Informatics

Comparison of RDF Triplestores in a Kubernetes Environment

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Peter Bretterbauer, BSc
Matrikelnummer 01325562

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Reinhard Pichler

Wien, 27. Jänner 2024

Markus Peter Bretterbauer　　Reinhard Pichler

Informatics

Comparison of RDF Triplestores in a Kubernetes Environment

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Markus Peter Bretterbauer, BSc
Registration Number 01325562

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Reinhard Pichler

Vienna, 27th January, 2024

Markus Peter Bretterbauer Reinhard Pichler

Erklärung zur Verfassung der Arbeit

Markus Peter Bretterbauer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. Jänner 2024

Markus Peter Bretterbauer

Danksagung

An dieser Stelle möchte ich mich zuallererst bei meinem Betreuer Prof. Dr. Reinhard Pichler bedanken, welcher mich bei meinen konzeptionellen Problemen beim Verfassen der Diplomarbeit sehr gut unterstützt hat. Des Weiteren möchte ich mich für die schnelle Erreichbarkeit und die unkomplizierten Treffen bedanken.

Weiteren Dank richte ich an Helmut Bretterbauer für seine emotionale Unterstützung während des Schreibens dieser Diplomarbeit.

Schlussendlich möchte ich meinen Eltern Sonja Schmidt-Kloiber und Peter Bretterbauer für ihre Unterstützung während meines gesamten Studiums bedanken.

Acknowledgements

First, I want to thank my supervisor Prof. Dr. Reinhard Pichler, for his help and feedback during writing this thesis. I further want to thank him for always being available when I needed further advice.

I also want to thank Helmut Bretterbauer for his emotional support while writing this thesis.

Finally, I want to thank my parents, Sonja Schmidt-Kloiber and Peter Bretterbauer for their support during my studies.

Kurzfassung

Semantische Netzwerke modellieren Konzepte (z.B. Personen) und ihre Beziehungen. Diese Netzwerke werden häufig mithilfe des *Resource Description Frameworks* (RDF), einem W3C Standard, modelliert, wodurch ein Datengraph entsteht. Ein Vorteil von RDF ist, dass zusätzliches Wissen mittels Schlussfolgerungen beziehungsweise Regel-Ableitungen generiert werden kann. Heutzutage speichern Systeme mehrere Terabyte an Daten. Systeme, welche auf genau einer Maschine laufen, können mit solch großen Datenmengen oft nicht mehr umgehen, da diese Maschinen durch ihren verfügbaren RAM beziehungsweise ihre verfügbare Anzahl an CPU Cores limitiert sind. Es existieren bereits Lösungen, die ihren Triplestore auf mehrere Maschinen verteilen. Deren Technologievielfalt erschwert allerdings die anwendungsabhängige Auswahl. Manche dieser verteilten Systeme wurden außerdem mit einer fixen Anzahl an CPU Cores und einer fixen Anzahl an Worker-Nodes getestet, wodurch es unklar ist, wie sie sich in einem anderen Setting verhalten. Des weiteren ist unklar, inwiefern diese Systeme auf dem weitverbreiteten Framework für Container-Orchestrierung namens *Kubernetes* funktionieren und beispielsweise von dessen Elastizitäts-Funktionen profitieren können.

In dieser Arbeit addressieren wir die Auswahl eines optimalen Triplestores für Kubernetes. Im Zuge dessen definieren wir einen Anwendungsfall *Betrugsbekämpfung*. Wir spezifizieren neun funktionale und drei Performanz-Kriterien für die Evaluierung. Durch eine Literaturrecherche identifizierten wir drei vielversprechende Triplestores für die Cloud, welche auch Regel-Ableitungen unterstützen, nämlich *Apache Rya Accumulo*, *Apache Rya MongoDB* und *SANSA-Stack*, die wir in Kubernetes installieren. Schließlich evaluieren wir diese Triplestores mit den vorher definierten Kriterien und ermitteln, inwieweit sie für unseren Anwendungsfall geeignet sind. Für die Evaluierung der Performanz verwenden wir die Kriterien Daten-Ladezeit, Anfrage-Antwortzeit und Antwortzeit bei nebenläufigen Anfragen bei verschiedenen Datengrößen. Dabei wird auf den LUBM-Benchmark zurückgegriffen. Schließlich analysieren wir die Vor- und Nachteile der Systeme.

Wir zeigen, dass Apache Rya MongoDB die meisten funktionalen Anforderungen hinsichtlich unseres Anwendungsfalls unterstützt. Beim Hinzufügen von Resourcen skaliert Apache Rya MongoDB gut bezüglich nebenläufiger Anfragen. SANSA-Stack skaliert generell sehr gut mit den verfügbaren Resourcen, benötigt jedoch sehr viel RAM. Apache Rya Accumulo scheitert am Laden größerer Datensätze innerhalb einer angemessenen Zeit, weshalb wir nicht jeden Test für diesen Triplestore durchgeführt haben.

Abstract

Semantic networks are used in order to model concepts (e.g. persons) and their relations to each other. These networks are often modelled using the *Resource Description Framework* (RDF), a W3C standard, resulting in graph structured data. An advantage of using RDF as data model is that reasoning/rule inferencing can be applied in order to infer additional knowledge. On today's systems the amount of data of a knowledge graph can reach up to a few terabytes. Single machine systems reach their limits on those use cases due to memory limits and performance constraints. Some systems already exist which claim to have solved this issue by employing a triplestore in a distributed environment. However, these systems use different techniques which makes it difficult to decide which system shall be used for which use case. Also some systems are benchmarked using only a fixed setting of CPU cores or number of workers making it difficult to predict how they scale by altering these settings. Furthermore, it is unclear on how far these triplestores support running on the widely used container orchestrating framework *Kubernetes* and benefit from its elasticity capabilities.

In this work, we address the problem of selecting an optimal triplestore for a Kubernetes environment. For this, we define a use case *fraud detection* for which we will evaluate our candidate systems. We specify nine functional and three performance evaluation criteria in order to be able to evaluate triplestores. By literature search, we identified three promising triplestores for the cloud, which also support reasoning, namely *Apache Rya Accumulo*, *Apache Rya MongoDB* and *SANSA-Stack* and we show how to deploy them in a Kubernetes environment. We analyse these triplestores based on our defined functional and performance evaluation criteria and discuss to which extent they benefit our defined use case. In order to measure their performance, we measure the data loading time, the query response time and the response times for concurrent queries for different data sizes using the *LUBM* benchmark. Finally we analyse the advantages and drawbacks of each system.

We show that Apache Rya MongoDB fulfills the most functional requirements regarding our specified use case. In terms of performance, Apache Rya MongoDB scales well for concurrent access when adding more resources. SANSA-Stack in general scales well with more resources, however it requires a huge amount of memory. Apache Rya Accumulo fails to load bigger datasets in a reasonable time, which is why we did not run every test for this triplestore.

Contents

CHAPTER 1

Introduction

Large Triplestores. The *Resource Description Framework*[1] (RDF) which was proposed by the *World Wide Web Consortium* (W3C) has become widely used in the last decades. Data points in RDF are represented as subject-predicate-object triples where the predicate describes the relation of the subject to the object. Multiple data points are possibly interlinked what finally results in a knowledge graph. These triples are stored in specialized databases, called *triplestores*.

Triplestores can be queried by a query language similar to SQL, called *SPARQL Protocol And RDF Query Language*[2] (SPARQL) where a query basically consists of triple patterns. It is a standardized query language also proposed by W3C.

Finally, also reasoning can be applied to RDF data by specifying rules in order to derive additional triples. Popular rulesets are contained in the *Resource Description Framework Schema* (RDFS) and the *Web Ontology Language* (OWL). It is also possible to define custom rules in order to further customize triple generation. Reasoning is in general not trivial, for instance reasoning in OWL is undecidable [HPvH03]. However, decidable fragments of OWL exist, for example OWL Lite, which provide a sufficient subset of rules in order to define useful ontologies.

More and more companies, organisations (e.g. the DBPedia Association[3]) and even governments (e.g. the Office for National Statistics[4] from the United Kingdom) store data in an RDF-based knowledge graph [AAH16]. The amount of triples being openly accessible also increases steadily. In 2009 the amount of data in the Linked Open Data Cloud[5] was estimated to be around 4.7 billion triples [BHB09] and increased to around

[1] `https://www.w3.org/TR/rdf11-concepts/`
[2] `https://www.w3.org/TR/sparql11-overview/`
[3] `https://www.dbpedia.org/`, SPARQL: `https://dbpedia.org/sparql`
[4] `http://statistics.data.gov.uk`, SPARQL: `http://statistics.data.gov.uk/sparql`
[5] Statistics can be found at: `https://lod-cloud.net/`

150 trillion triples in 2020 [Opd21]. This imposes a challenge on systems which host many triples in how to store triples efficiently and how to let users query them within a small latency.

As the amount of data increases, single-machine triplestores become inefficient or even unfeasible due to constraints on available main-memory and CPUs. For example, a very popular[6] single machine RDF framework is the *Apache Jena framework*[7]. Jena provides capabilities to store and query RDF data in a triplestore called *TDB1* (Triple Database). Furthermore it supports RDFS and OWL to add extra semantics to the data. It also implements OWL and RDFS reasoners and also supports the configuration of custom inference rules. Finally, a SPARQL server can be deployed with *Apache Jena Fuseki*. However, being a single-machine framework, it has limitations like how much data can be held in memory due to memory restrictions on a single machine which can result in performance degradation and server-crashes. Also TDB1 is not designed to be accessed from multiple processes which limits its scalability. It claims to have solved the issue with version 1.1.0 but only "under most circumstances"[8]. This is a bottleneck for the data ingestion phase because the database cannot keep up with the amount of data being generated during this phase since data-ingestion can easily be scaled horizontally. A system which succeeds TDB1, called *TDB2* exists which improves some single machine operations on the database. However, it is still a single-machine database and therefore limits its ability to scale.

Furthermore, new data insertions may occur infrequently which results in wasted resources on single-machine frameworks which usually have a fixed amount of resources allocated and therefore cannot scale dynamically. For centralized databases currently this means that large amounts of computational resources are reserved to be able to accelerate the preprocessing phase in order for users to be able to search the data as soon as possible. But after the computationally intensive data-insertion and rule-inferencing phases, the database is mostly idle and over-provisioned until a new data-insertion job is issued. This issue could be alleviated by deploying the system in a cloud environment which features elasticity capabilities.

Current approaches. There have already been some approaches on how to achieve horizontal scalability. One of them is to use NoSQL stores as a storage backend, which sometimes offer native horizontal scalability like *Apache Accumulo*[9] or *Apache HBase*[10] which are based on the *Apache Hadoop*[11] project. Other distributed NoSQL stores include *Apache Cassandra*[12] which does not have a dependency on Hadoop. Triplestores which rely on such a storage backend include *Apache Rya* [PCR12] and *CumulusRDF* [Har11].

[6]Best open-source system on `https://db-engines.com/de/ranking/rdf+store` accessed at 3rd of August 2020

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

[8]`https://jena.apache.org/documentation/tdb/`

[9]`https://accumulo.apache.org/`

[10]`https://hbase.apache.org/`

[11]`https://hadoop.apache.org/`

[12]`https://cassandra.apache.org/`

Other systems rely on a distributed storage like the *Hadoop Distributed File System* (HDFS) from the Hadoop framework. Such systems include SANSA-Stack [LSB+17] and SHARD [RS10]. For data which is stored directly on HDFS, usually MapReduce or recently, Apache Spark can be used to query data. SHARD for instance transforms a SPARQL query into a MapReduce algorithm [RS10]. When relying on Apache Spark, with *Sparklify* and *Ontop* one can transform SPARQL queries to SQL queries and directly use Apache Spark SQL in order to query the data [SSGL19b, CCK+17]. This is used in the SANSA-Stack framework [SSGL19a].

Also some specialized stores exist like the column store for (clustered) *OpenLink Virtuoso* servers [Erl12, BEP14]. Virtuoso initially was implemented as a relational database system and was later extended to support RDF data with SPARQL query- and inference support [EM09].

We use a fraud-detection application where most of the data is loaded initially and then only few insertions/updates happen afterwards. The time until the data can be queried and also querying performance is crucial. However, it is unclear which triplestore solution should be chosen for that use case.

In other words, there are many triplestores which are backed by different frameworks, storage engines and supported features. Thus, it is unclear which solution should be chosen also for other use cases. Some triplestores claim to be better than others (e.g. Apache Rya is in general better than SHARD [PCR12]) but in many cases such evaluations do not exist.

In one evaluation, the authors classify different systems based on their capabilities and implementation techniques [KM15]. However, they do not cover some of the latest developments on distributed RDF databases (e.g. SANSA-Stack). Furthermore they do not cover performance, elasticity and workload evaluations on these databases. Therefore they lack of giving a potential user significant criteria on when to use which triplestore solution.

Additionally more and more companies are using Kubernetes as container orchestration framework in order to run their applications. Since Hadoop was developed for YARN, it is unclear how well systems which are based on Hadoop are supported in a Kubernetes environment together with e.g. its elasticity features. In Figure 1.1 it can be seen, that the interest in Kubernetes is increasing, while the interest in Hadoop is decreasing[13]. The bump after 2020 may occur because of the COVID-19 pandemic.

Goal. The goal of this thesis is to define functional and performance evaluation criteria in order for users to decide which system is best for various usage scenarios. Then we select three candidate systems which we evaluate based on the aforementioned criteria.

In order to understand the different triplestores, we conduct a literature research and study the examined systems. Then we install and configure those systems on a Kubernetes based

[13] https://trends.google.com/trends/explore?date=all&q=Hadoop,Kubernetes
[14] See footnote 13

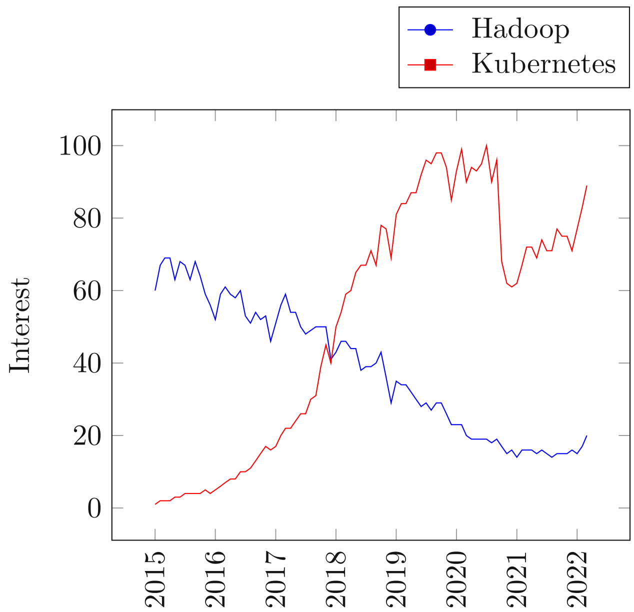

Figure 1.1: Kubernetes and Hadoop - Interest Comparison[14]

cloud and identify possible pitfalls and limitations. Afterwards we use two benchmarks in order to do performance evaluations for separate workloads and different settings of computation resources in a Kubernetes-based cloud. The performance for different query-shapes is benchmarked with the *Lehigh University Benchmark* (LUBM) [GPH05] which already provides suitable queries together with an ontology and configurable data sizes. Parallel access is benchmarked by using Apache JMeter which issues LUBM queries in parallel (following [PCR12]). Finally, we decide which system fits our use case best.

Results. We find, that none of the evaluated distributed triplestores supports whole OWL-Lite reasoning or custom inference rules, resulting in incomplete query answering using the LUBM benchmark. However, Apache Rya with a MongoDB backend performs well for increasing data sizes, although increasing resources does not always have a great impact in terms of query response times except for parallel access. Furthermore, with the MongoDB Kubernetes Operator, elasticity is (almost) natively supported. Apache Rya with an Accumulo backend on the other hand was not able to load the LUBM 20 dataset in a reasonable time in our test settings. The SANSA-Stack system scaled very well with the assigned resources. However, it would need much more resources than in our test settings to become faster in querying than Apache Rya (MongoDB). Also, it uses much more memory during our tests. Finally, although supporting SPARQL 1.1, SANSA-Stack does not support SPARQL updates, insertions and deletions. Choosing the optimal triplestore therefore depends on the requirements and the envisioned use case.

Structure. This thesis is structured as follows: Chapter 2 provides the most important definitions regarding RDF, Kubernetes and different database types. Then we define a use case in Chapter 3 for which our candidate systems will be evaluated. In Chapter 4 we define functional and performance evaluation criteria in order to be able to evaluate different systems. We present our candidate systems in Chapter 5 and evaluate them based on the aforementioned functional evaluation criteria. A performance evaluation for these systems is then conducted in Chapter 6. Finally, in Chapter 7 we conclude and give an outlook on further interesting research topics.

CHAPTER 2

Preliminaries

2.1 RDF

The *Resource Description Framework* (RDF) is a W3C recommended[1] data model for describing resources and their relationships as a directed knowledge graph. It is a basic building block for the semantic web. There are many public knowledge graphs like DBpedia[2], which provide their information in the form of RDF triples which can be queried by using *SPARQL* which is described in Section 2.1.2.

2.1.1 Notation

In RDF each data point consists of a triple, namely *subject*, *predicate* and *object*. A subject is a uniquely identified *resource*, represented by an *Internationalized Resource Identifier* (IRI) or a blank node, which will be described later. The predicate, which is also an IRI, describes the relation of the subject to an object. An object is again an IRI or a blank node and can also be a literal. There are a variety of formats in which RDF graphs can be represented as text like *Turtle*[3], *N-Triples*[4], *RDF/XML*[5] and *N3*[6]. We will use Turtle in our example since it has a very compact syntax. In order to standardize the usage of RDF in different domains, different vocabularies like FOAF[7] emerged. In order to ease the work on defining triples, there are some keywords which allow abbreviations in Turtle. The "@prefix" keyword specifies abbreviations of the form "a: <b>" where "a" is a string containing letters and "b" is an IRI. The prefix then

[1] `https://www.w3.org/TR/rdf11-concepts/`
[2] `https://wiki.dbpedia.org/`
[3] `https://www.w3.org/TR/turtle/`
[4] `https://www.w3.org/TR/n-triples/`
[5] `https://www.w3.org/TR/rdf-syntax-grammar/`
[6] `https://www.w3.org/TeamSubmission/n3/`
[7] `http://xmlns.com/foaf/spec/`

can be used in triples by using the abbreviation instead of the whole IRI (e.g. defining "@prefix abbr: <http://example.org#>" and using it in a subject of a triple declared with "abbr:Example" results in "<http://example.org#Example>"). Another abbreviation in Turtle for the declaration of triples can be applied by using the ";" keyword. With that keyword one can omit the subject for subsequent predicate-object tuples allowing the declaration of multiple predicates with their objects for one subject. An example of a simple graph is given in Listing 2.1.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://www.example.org/example#Bretterbauer>
        rdf:type foaf:Person ;
        foaf:givenName "Markus" ;
        foaf:familyName "Bretterbauer" .
```

Listing 2.1: RDF triple example (Turtle syntax)

The terms "rdf:" and "foaf:" represent the IRIs of the RDF and FOAF vocabulary respectively. "rdf:type" could be abbreviated by simply writing "a" instead. Written-out, the resulting triples are:

1. <http://www.example.org/example#Bretterbauer>
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
   <http://xmlns.com/foaf/0.1/Person>

2. <http://www.example.org/example#Bretterbauer>
   <http://xmlns.com/foaf/0.1/givenName> "Markus"

3. <http://www.example.org/example#Bretterbauer>
   <http://xmlns.com/foaf/0.1/familyName> "Bretterbauer"

An alternative representation for the resulting triples can be given as a directed graph, see Figure 2.1. In such a representation, subjects and objects are drawn as nodes, while predicates are drawn as edges.

Finally we also mention the concept of *blank nodes* which "indicate the existence of a thing, without using an IRI to identify any particular thing"[8]. Blank nodes can occur only as a subject or as an object of a triple and can be referenced only in that particular graph in which they are defined. Therefore, referencing the "same" blank node in another graph would reference another blank node. These type of nodes have several purposes: describing n-ary relationships[9], describing meta-informations of a triple, hiding sensitive information, expressing multi-relationships and defining resources for which a suitable

[8] https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/#blank-nodes
[9] https://www.w3.org/TR/swbp-n-aryRelations/

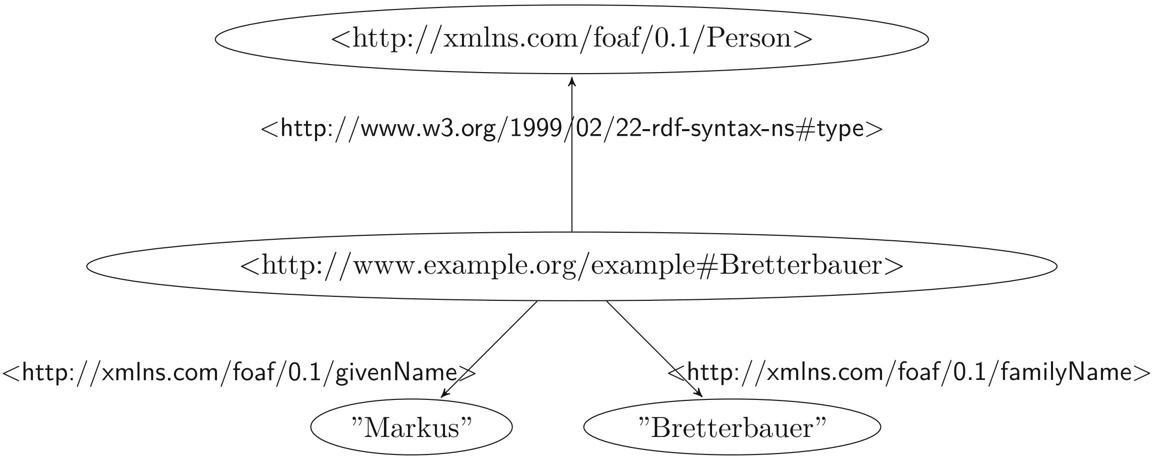

Figure 2.1: Triple Graph Example

URI temporarily cannot be identified but properties of that resources shall be stated [CZCG12].

2.1.2 SPARQL

In order to query a collection of triples, W3C recommends the *SPARQL*[10] query language. In SPARQL one primarily defines basic triple patterns in order to query the triplestore. More specifically, a query is constructed as follows[11]: First, one can define "PREFIX"es in order to abbreviate vocabulary domains when defining *Triple Patterns* (analogous to @prefix in RDF definitions in Turtle format, see Section 2.1.1). Then follows a "SELECT" clause where one specifies variables which in the end contain the found results. Finally follows a "WHERE" clause containing *Basic Graph Patterns* (BGPs) which consist of Triple Patterns (whitespace-separated list of triples). Additionally there are keywords which modify the result e.g. by ordering it ("ORDER BY") or by limiting ("LIMIT") the number of results. A complete list of result modifiers can be found in the W3C documentation[12]. In order to evaluate the query, the query engine performs pattern matching on those BGPs and assigns matches to the declared variables in the SELECT clause.

A simple example for querying 10 persons who know another person whose first name is "Markus" is given in Listing 2.2. The OPTIONAL keyword means that if an object for the property "foaf:familyName" does exist for a subject "?knowsMarkus", it is also returned, otherwise the output for the family name is simply empty. On the other hand, omitting the OPTIONAL keyword would only print those subjects having their family name persisted in the database. The number of results in this example is restricted to 10 using the LIMIT keyword.

[10] `https://www.w3.org/TR/sparql11-overview/`
[11] `https://www.w3.org/TR/rdf-sparql-query/`
[12] See footnote 11

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?knowsMarkus ?familyName WHERE {
        ?namedMarkus foaf:givenName "Markus".
        ?knowsMarkus foaf:knows ?namedMarkus.
        OPTIONAL { ?knowsMarkus foaf:familyName ?familyName }
} LIMIT 10
```

Listing 2.2: SPARQL Example

Version 1.1 of SPARQL also supports aggregate functions like COUNT, MIN, MAX, etc. in a fashion similar to SQL. Listing 2.3 for instance shows a query which outputs all persons together with the number of other persons they are known by.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?person (COUNT(?knows) as ?knownByCount) WHERE {
        ?person a foaf:Person .
        ?knows a foaf:Person .
        ?knows foaf:knows ?person .
} GROUP BY ?person
```

Listing 2.3: SPARQL Count Example

Furthermore support for insertions, updates and deletions was added. An example for an insertion is given in Listing 2.4 where a triple containing a gender is added to the database using the "INSERT DATA" keyword.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX example: <http://www.example.org/example#>

INSERT DATA {
        example:Bretterbauer foaf:gender "male" .
}
```

Listing 2.4: SPARQL Insert Example

Analogous to an insertion, a deletion is done by using the "DELETE DATA" keyword. Finally, an update is accomplished by using a "DELETE/INSERT" operation. An example is given in 2.5 where all occurrences of the given name "Marcus" are updated to "Markus".

The DELETE and INSERT statements are optional although one of them must exist, meaning that also insertions and deletions can be conditionally applied using the WHERE

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

DELETE { ?person foaf:givenName "Marcus". }
INSERT { ?person foaf:givenName "Markus". }
WHERE {
        ?person foaf:givenName "Marcus".
}
```

Listing 2.5: SPARQL Update Example

keyword.

2.1.3 Rule Systems

There are several rule systems in order to extend the expressiveness of RDF.

RDFS

The *Resource Description Framework Schema*[13] (RDFS) extends RDF by adding a vocabulary in order to be able to model ontologies. For instance it adds the support for defining classes. Therefore it uses the properties "rdfs:Class" in order to specify that a resource is a class and "rdfs:subClassOf" in order to define a class being a subclass of another class. Furthermore it contains for instance several entailment rules in order to derive subclass-information.

For example, if the triplestore contains both triples of Listing 2.6, then RDFS entailment would infer the triple "ex:Bretterbauer rdf:type foaf:Person" on the data (by using the entailment pattern *rdfs9*[14]).

RDFS also defines extensions to properties, for instance "rdfs:range" which restricts assigned objects for properties. To be more specific, for a property P and an object O, "P rdfs:range O" states that when using property P in a triple, the corresponding object must be an instance of class O. For example, "foaf:knows rdfs:range foaf:Person" states, that when using the property "foaf:knows", the corresponding object must be an instance of class "foaf:Person". A similar concept is "rdfs:domain" which similarly restricts the subject instead of the object of a triple.

For a full list of features, we refer to the W3C recommendation[15].

[13] https://www.w3.org/TR/rdf-schema/
[14] https://www.w3.org/TR/rdf11-mt/#patterns-of-rdfs-entailment-informative
[15] See footnote 13

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix ex: <http://www.example.org/example#>

ex:Bretterbauer rdf:type ex:Student .
ex:Student rdfs:subClassOf foaf:Person .
```

Listing 2.6: RDFS Example (Turtle)

OWL

The *Web Ontology Language*[16] (OWL) further extends RDFS by providing more expressive vocabulary and entailment rules for ontology creation. It is comprised of three sublanguages, namely *OWL Lite*, *OWL DL* and *OWL Full* (where OWL Lite $\subset$ OWL DL $\subset$ OWL Full) which differ mainly on how a reasoner can operate on the data. For instance, OWL Full is undecidable, therefore no complete reasoning can be guaranteed. On the other hand, OWL DL and OWL Lite are decidable subsets of OWL [HPvH03].

OWL Lite[17] is a subset of OWL which aims to provide a useful set of features while reducing the complexity for tool developers and representing a decidable subset of OWL Full [LN04]. It provides RDFS features together with features for stating (in-)equality of resources and can also be used to describe property characteristics and restrictions and also cardinality information items of associated values of a resource. For instance, an "owl:sameAs" property, which states resource equality, can be used to state that resources of different databases are in fact the same (e.g. database1:bretterbauer owl:sameAs database2:bretterbauer). The property characteristic "owl:inverseOf" states that a property is an inverse property of another property (e.g. example:hasTeacher owl:inverseOf example:hasStudent). Property restrictions restrict assigned values of properties. In order to to that, one specifies a restriction as superclass of another class which defines the restrictions which are applicable for that class. For example one could state that object values for a property example:hasStudent must be of class example:Student by using owl:someValuesFrom in context of a <owl:Restriction>, see Listing 2.7 for an example. Cardinalities can be used to state that for example a subject example:Teacher must teach at least one and at maximum thirty students using owl:minCardinality and owl:maxCardinality in context of an <owl:Restriction>. However, in OWL Lite cardinalities are restricted to the values "0" and "1".

The OWL DL[18] language contains all language features of OWL Full for which decidability and completeness can be guaranteed. It furthermore extends OWL Lite by dropping the

[16] https://www.w3.org/TR/owl-ref/
[17] https://www.w3.org/TR/owl-features/
[18] https://www.w3.org/TR/owl-guide/

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix ex: <http://www.example.org/example#>

_:hasStudentValuesRestriction rdf:type owl:Restriction .
_:hasStudentValuesRestriction owl:onProperty ex:hasStudent .
_:hasStudentValuesRestriction owl:someValuesFrom ex:Student .

ex:Teacher rdf:type owl:Class .
ex:Teacher rdfs:subClassOf _:hasStudentValuesRestriction .
```

Listing 2.7: OWL Restriction Example (Turtle)

cardinality constraint by allowing arbitrary numbers. For a complete list of features, see the W3C document[19].

Further fragments of OWL Full include *OWL Horst* (also called *OWL pD**). OWL Horst includes a subset of rules from RDFS and OWL Full while having weaker semantics than OWL Full. Its semantic is weaker in a sense that its entailment rules are not as strict as they are in OWL Full. For example, "owl:sameAs" in OWL Horst is treated as an equivalence relation instead of equality. In general, it derives less triples than OWL Full while having a weaker computational complexity [tH05, KP15].

The LUBM benchmark we use requires OWL Lite in order for its queries to return the complete answers [GPH05].

OWL 2

OWL 2 is an extension to OWL and is also a W3C recommendation[20]. It is fully compatible with OWL, which means that OWL (1) ontologies stay valid in OWL 2. Extensions in OWL 2 include richer datatypes, data ranges, asymmetric and reflexive properties, etc.[21] [GHM+08]. Furthermore, OWL 2 introduces the concept of *Profiles* which represent subsets of the OWL 2 language in order to benefit diverse application scenarios. It natively defines three profiles, namely *OWL 2 EL*, *OWL 2 QL* and *OWL 2 RL*. However, also OWL Lite can be seen as a profile in OWL 2.

The OWL 2 EL profile[22] trades expressiveness with performance allowing efficient reasoning in very large ontologies. OWL 2 QL is designed for applications where relational queries are used in order to search in the graph and where backward chaining is among the most common tasks. Conjunctive query answering can be done in LOGSPACE wrt.

[19] https://www.w3.org/TR/owl-features/
[20] https://www.w3.org/TR/owl2-overview/
[21] A list of new features can be found at https://www.w3.org/TR/owl2-new-features/
[22] https://www.w3.org/TR/owl2-profiles/

the size of the data. However, the expressiveness of this profile is very limited. Finally, OWL 2 RL offers scalable reasoning with more expressiveness. It should be particularly used when having a rather lightweight ontology and where data operations are directly done on RDF triples.

2.1.4 Other Rule Systems

There are further rule systems, like the generic Jena rule system where one can define custom rules. As a simple example, the Jena rule syntax[23] can be used to define a forward chaining rule in order to derive a new triple "b knows a" if "a knows b" as seen in Listing 2.8.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

[knows: (?A foaf:knows ?B) -> (?B foaf:knows ?A)]
```

Listing 2.8: Inference Example

Another system where it is possible to define custom rules is the *Semantic Web Rule Language (SWRL)*[24] which is a proposed language for the semantic web. It combines OWL DL and OWL Lite with the *Rule Markup Language (RuleML)* in order to be able to express rules.

We also mention the *Shapes Constraint Language (SHACL)*[25] rule system which can be used to validate RDF graphs against a set of conditions. As an example, it can be used to constrain the number of social security numbers for persons to exactly one, therefore making the graph more reliable.

2.1.5 Reasoning

Reasoning or *Entailment* is the process of automatically deriving information from a given knowledge base (collection of triples). In order to do this, one specifies rules on which triple patterns must exist in a database to derive further triples.

There are basically two approaches in rule-based reasoning. One approach is called *forward chaining*. In this approach, all triples are inferred and materialized before a query is issued to the system. Forward chaining has the advantage that, compared to backward chaining, query answering is fast since all inferred triples are already saved to the triplestore and no further computation is needed [Rus16]. However, the process is potentially time consuming since reasoning on the whole data needs to be done at the very beginning after the data is loaded, resulting in a delayed access to the data. Also forward chaining needs to be applied every time after triples are added or updated. Furthermore,

[23] https://jena.apache.org/documentation/inference/
[24] https://www.w3.org/Submission/SWRL/
[25] https://www.w3.org/TR/shacl/

forward chaining may become unfeasible since potentially large datasets are derived from comparatively few data. For instance [HZU+12] states that 33052 equivalent entities (declared with the property "http://www.w3.org/2002/07/owl#sameAs") suffice to infer over one billion triples. There are already several implementations for forward chaining in distributed environments including the usage of MapReduce [UKOvH09, UKM+12] and Spark jobs [KP15].

The other approach is called *backward chaining* where inferred triples are computed on demand while processing a SPARQL query. This is usually done by query-rewriting where an issued query is transformed into another query which includes the inferred results [UvHSB11]. The advantage of this approach is that after the data is loaded, no reasoning needs to be applied before accessing the data. Also there is no need to infer potential triples of insertions and updates immediately since a rewritten query automatically considers the whole dataset. However, the huge drawback of that approach is that queries become large and potentially time consuming to answer. Furthermore inferred triples may become computed multiple times since the resulting inferred triples are not saved to the triplestore. Implementations of backward chaining include QueryPIE [UvHSB11].

In order to benefit from each of the mentioned advantages and to minimize each of the disadvantages, hybrid reasoners exist. Reasoners which use both forward- and backward-chaining include the OWL reasoners in the Virtuoso Universal Server and the Apache Jena project [Rus16].

2.2 Kubernetes

Kubernetes[26] is an open source orchestration tool for containerized applications in a cloud and was originally developed by Google. It eases the management of applications by providing features like scaling, self-healing, load balancing and many others. Kubernetes is supported by various cloud vendors like Google Cloud Platform[27] (GCP), Microsoft Azure[28], Red Hat Openshift[29] and Amazon AWS[30]. We will use the Kubernetes engine in GCP for our evaluations.

Kubernetes basically runs containerized applications and abstracts the assignment of computation resources. Instead of creating virtual machines with a specific configuration by hand, one just needs to assign computation resources to Kubernetes and let the applications define their resource configuration. Kubernetes then automatically reserves the defined resources for the application and runs it in that setting. An example for such

[26] https://kubernetes.io/
[27] https://cloud.google.com/kubernetes-engine
[28] https://azure.microsoft.com/en-us/services/kubernetes-service/
[29] https://www.redhat.com/en/technologies/cloud-computing/openshift
[30] https://aws.amazon.com/eks/

a system is the *GKE Autopilot*[31] alleviating the user from provisioning dedicated nodes.

A *Container* is a virtualization which bundles all software/programs which are needed for the intended application to run. Thus, in order to run an application in Kubernetes, it must be wrapped into a *Container-Image* (e.g. with Docker[32]). The image must then be uploaded into a *registry* and finally can be referenced in the Containers configuration in order for the Container to run the image.

In Kubernetes, everything is created in a declarative manner. Thus, a developer chooses what she wants and how the system shall behave, and the system tries to follow the description, hiding the detailed implementation from the developer. Such a description/-configuration is usually written in *YAML*[33].

Kubernetes supports different "objects" for deploying applications which support different characteristics a developer can choose from.

A *Pod*[34] is the smallest deployable unit in Kubernetes. It can host multiple containers which run their assigned programs. Pods consume configurable amounts of *virtual* CPUs (vCPUs) and memory, therefore an efficient handling of Pods is crucial in achieving a high utilization of resources. In this thesis, the terms CPU and vCPU are used interchangeably. For those, one defines "resource requests" for containers in the Pods YAML configuration and the Kubernetes system then allocates those resources from its pool of resources. The benefit of using virtual CPUs is that one also can for instance define 0.5 vCPUs for a Pod and thus only pay for half of a core when not much computing speed is needed.

PersistentVolumeClaims[35] (PVC) can be used to request *PersistentVolume*s (PV) from Kubernetes. Therefore, one basically specifies the storage capacity needed and the storage class, which is used to support multiple storage 'qualities' (usually in terms of speed), in the PVC and Kubernetes tries to find a suitable PV which is then bound to the PVC. Also different reclaiming policies are supported which indicate how the PV shall behave when its corresponding PVC is deleted, e.g. delete the data of the PV or retain it. The capacity of a PV does not need to be fixed, it can also be set as expandable in order to be able to expand the capacity in the future.

Deployments[36] can then be used to handle the lifecycle of Pods, starting with automatic rollouts, replication, readiness and self-healing. For a deployment, one can also setup a *HorizontalPodAutoscaler* (HPA) or a *VerticalPodAutoscaler* (VPA). The HPA is used to add or remove Pods dynamically in order to fit a specified target workload, while the VPA is used in order to dynamically set good values for CPU and memory based on the workload of a Pod. A variation of a Deployment is a *StatefulSet*[37] where one can

[31] https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview
[32] https://www.docker.com/
[33] https://yaml.org/
[34] https://kubernetes.io/docs/concepts/workloads/pods/
[35] https://kubernetes.io/docs/concepts/storage/persistent-volumes/
[36] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[37] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

specify a "volumeClaimTemplate" in its configuration in order for (multiple) Pods to request storage for themselves. In a Deployment on the other hand one can only request a shared storage for all managed Pods.

To make the application reachable from within Kubernetes or from the outside world, one needs a *Service*[38] object. This object specifies the type of service (e.g. *LoadBalancer* in order to be reached from extern or *ClusterIP* if it only shall be reached from within the cloud) and the ports to which requests shall be forwarded to the application.

Finally, we mention Kubernetes *Operators*[39], which is a pattern that aims to fully automate the management of an application. For an operator, one creates *Custom Resource Definitions* (CRD) in a declarative manner which specify application relevant configurations. CRDs are completely user-defined and therefore there are no limitations on what a user can declare in these. Concrete instances of CRDs are called *Custom Resource*s (CR). For example, a CRD could define a "worker" property which accepts numbers as value, representing the number of workers. Then, one can develop an operator, which reacts on (changes to) CRs. A reaction for instance can be, that it scales up a Deployment when the value of the "worker" property changes from "2" to "3".

Figure 2.2 shows the most important concepts used in this paper. The only physical objects in this diagram are the *Node*s which are physical machines in a Kubernetes cluster. These nodes run several Pods which contain Containers. They also have a label assigned in order for other objects to be able to reference them. In these containers, images are executed. In this figure, the Pods run as parts of Deployments and StatefulSets. If a Deployment or a StatefulSet is deleted, their managed Pods also become deleted. The Deployment manages one PVC, while the Pods of the StatefulSet manage their own PVCs. Finally, the Services in this example can be used in order for applications to communicate with the Pods.

[38] https://kubernetes.io/docs/concepts/services-networking/service/
[39] https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

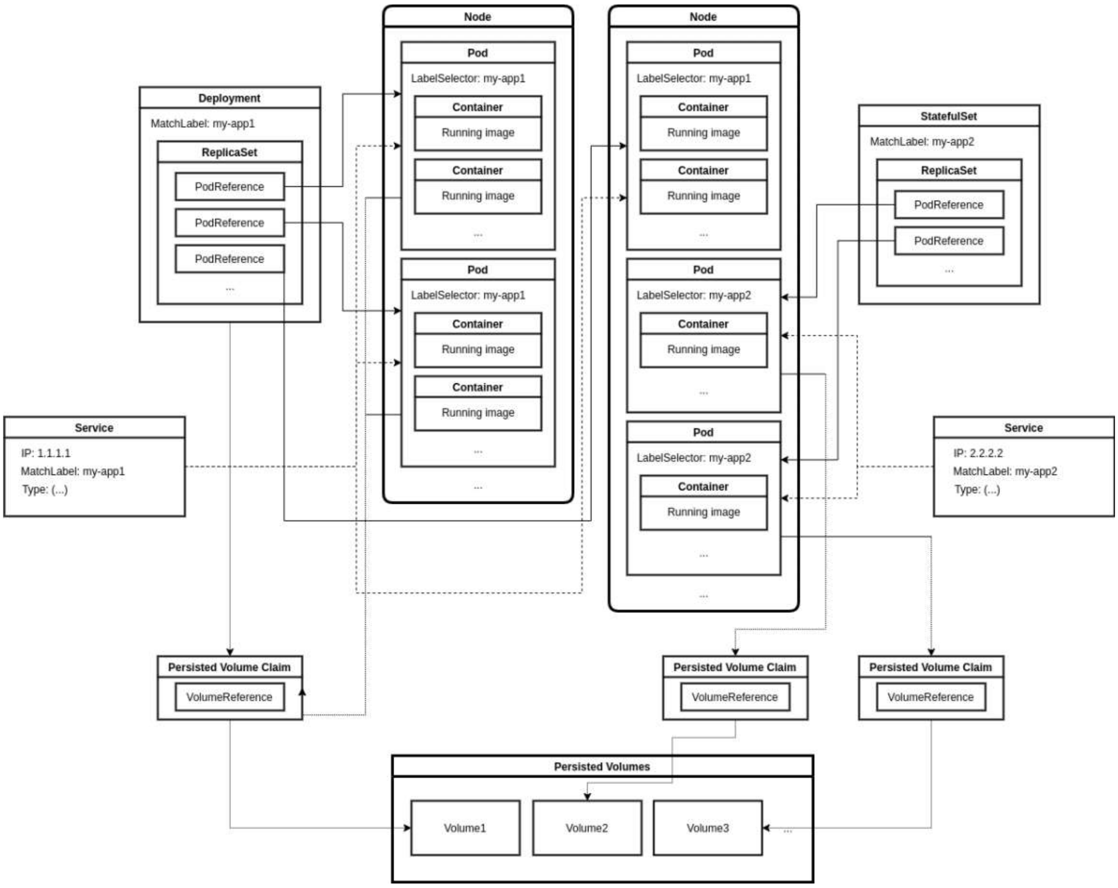

Figure 2.2: Kubernetes Objects Overview

2.3 Triplestores

The querying performance is a crucial aspect when choosing a triplestore. As the amount of data increases, the response time for a query in general also increases since all matching query patterns need to be found in the data. Also the data loading time is a relevant aspect regarding performance for a triplestore since it indicates how much time a system needs in order for users to start querying the data.

2.3.1 Storage

One aspect, which impacts the performance for a triplestore, is the storage backend used in order to store data. For instance, *Jena SDB* stores data into an SQL database [Wil06] [WSKR03]. Its successor, *Jena TDB* is a specialized storage backend which stores the data directly on the disk. Finally, systems like *Apache Rya* store data in a NoSQL database like Apache Accumulo. In this chapter we describe different approaches on how RDF data is stored in recent systems.

Relational databases

Relational databases are the most popular[40] database type. Those databases benefit from many years of development and experience. In those databases, data is stored in tables on which a schema is employed. A popular standardized query language for those structural data is *SQL*.

NoSQL

NoSQL databases store data in a structured way, but without employing a schema on the data. They are often used when big continuous data streams need to be saved, since they usually have a better performance than SQL databases [JA20, LM13]. Furthermore, they tend to be simpler to scale horizontally. However, many NoSQL stores do not support ACID consistency. Recent developments on the other hand show that ACID compliant NoSQL stores are possible [LSEA16].

Distributed Storage - HDFS

With the creation of *Apache Hadoop*, distributed storages like the *Hadoop Distributed File System* (HDFS) [SKRC10] emerged. It is the open-source implementation of the *Google File System* [GGL03] and is capable of storing very large datasets and provides a fault-tolerant file system by implementing redundancy while running on commodity hardware.

HDFS consists of *NameNodes*, which store metadata and provide access from clients, and *DataNodes* which store the actual data. To improve durability, data is replicated three times by default, but this is configurable. Replication does not only increase durability, but also increases the read-performance, since data can be read from multiple disks simultaneously.

2.3.2 Partitioning

Partitioning plays an important role when efficiently accessing data in a database. A naive approach for storing triples is to store them into a relational table with three columns for subject, predicate and object. However, in many use cases, many properties are queried for one resource (star-queries), which results in a big amount of joins. An early approach on improving the handling of star-queries is storing triples in a so-called *Property Table* (PT), where the first column represents the subject and every other column all of the known possible predicates [Wil06]. Therefore, one row stores the subject together with every object, according to the predicate-columns of the table. This reduces the number of joins required for a query and also helps a query optimizer to collect statistics about the data in order to improve query ordering. However, as one can imagine, many NULL values occur in this setup, since normally, a subject is not connected to an object for

[40] `https://db-engines.com/en/ranking`

every predicate which is present in the dataset. Furthermore, this approach is not very scalable since data resides in one, possibly large, table.

A scalable approach in partitioning RDF data is *Vertical Partitioning* (VP) [AMMH07]. In this approach, a new table for every property in the data is generated. A table therefore consists only of two columns, namely the subject and the object while the name of a table represents the property. The advantage of VP is, that otherwise big tables are divided into smaller ones and these smaller tables can be distributed to different nodes. Furthermore the size of each table usually does not become too big which means that the whole table can be loaded into memory. However, for some properties (e.g. the "rdf:type" property) the tables can still grow to a decent size. The disadvantage of this approach is, that when querying every property for a subject, many joins, possibly across many nodes, are needed resulting in a communication overhead. This is also true when writing several properties for a subject.

2.3.3 Benchmarks

There are several benchmarks which are commonly used to evaluate triplestores. They typically are able to generate datasets with adjustable sizes. One widely used benchmark is the *Lehigh University Benchmark* (LUBM) which uses a university ontology [GPH05]. It consists of 14 test queries written in SPARQL. Some of these queries assume that OWL Lite inferencing is supported in order to return the correct results. The benchmark contains a tool, namely *UBA*, which is used to create data for variable university sizes. For example, LUBM 1 (one university) contains around 100000 triples, while LUBM 50 already contains around 7 million triples [GPH05]. The performance for one of our candidate system, Apache Rya Accumulo (see section 5.1), was evaluated with LUBM 15000 which involved 2.1 billion triples [PCR12].

Another popular benchmark is the *Waterloo SPARQL Diversity Test Suite* (WatDiv), which claims to produce more realistic data for benchmarking [AHÖD14]. It supports defining a user-defined dataset by their dataset description language[41]. The benchmark contains tools for data and query generation for which one can specify the number of query templates to be generated and also the maximum number of triple patters in the generated queries. For basic testing, they already provide a set of 20 query templates[42]. Furthermore, the data generator supports a scale factor in order to scale the number of triples to generate. For example, the WatDiv data generator with scale factor 1 will produce around 100000 triples. The data generated linearly scales with the scale factor provided [AHÖD14].

Other benchmarks include the *Berlin SPARQL Benchmark* (BDSM) which is settled in an e-commerce domain and the SP^{2}Bench benchmark which uses the *Digital Bibliography & Library Project*[43] (DBLP) as a domain for its dataset [DKSU11].

[41] https://dsg.uwaterloo.ca/watdiv/watdiv-schema-tutorial
[42] https://dsg.uwaterloo.ca/watdiv/#tests
[43] https://dblp.org/

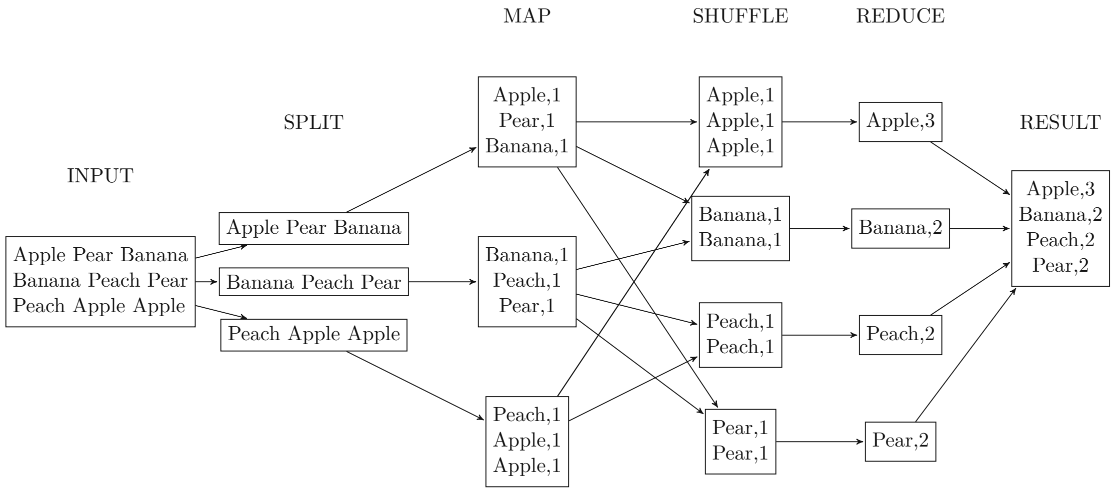

Figure 2.3: MapReduce WordCount Example

2.4 Distributed Computation Methods

2.4.1 MapReduce

MapReduce is a highly scalable computation framework originally developed by Google [DG04]. It basically processes key-value pairs in three steps: Map, Shuffle and Reduce. The Map step is a user-defined function on a list of key-value pairs, which can be computed in parallel on every connected node for the data on that node. Afterwards, the data is "shuffled" such that values with the same key are transported to the same node. Finally during Reduce, a user-defined function is computed for each data point with the same key which is again parallelisable since different keys can be computed at different nodes. A popular example of counting words in MapReduce is given in Figure 2.3 [Läm08, DG04]. First, the input is divided into smaller subsets which are distributed to possibly different nodes. In the Map phase, a user-defined function assigns "1" to each of the words, each word now being the key of its data point. Then, in the Shuffle phase, each data point with the same key is sent to the same node, where another user-defined function is defined, which in our case adds up the values that were assigned beforehand in the Map phase. Finally, the resulting word-counts are collected.

An open source implementation of MapReduce is implemented in *Apache Hadoop*[44]. Hadoop uses HDFS, which was already mentioned in Section 2.3.1, in order to store data.

However, every time when issuing a MapReduce operation, the system needs to read and write data from and to potentially slow disks, which causes a performance bottleneck making the framework more suitable for batch-processing than for interactive processing [ZCF+10].

[44] https://hadoop.apache.org/

2.4.2 Apache Spark

In order to make processing distributed data more interactive, the *Apache Spark* framework was developed [ZCF+10]. It is built around *Resilient Distributed Datasets* (RDDs), which are immutable data collections. These can be held in-memory on different nodes while providing prevention mechanisms against dataloss. Spark also supports the use of *DataFrames* which are basically RDDs of structured records [AXL+15].

Data quering can be done similarly to MapReduce and HDFS can be used to load and store data. Spark also supports a query language named *Spark SQL*, which is similar to normal SQL making the system very accessible for a variety of developers [AXL+15]. Spark SQL makes use of the known schema of DataFrames in order to process them. For example, one can define a DataFrame collection of "students" with a property "age". DataFrames support operations to query such collections, for example in order to query all students who are older than 25, the following Scala code can be used: "students.where(students("age") > 25)" [AXL+15]. All common relational operations (select, where, join, group-by) are supported by Spark SQL. Therefore, an SQL query just needs to be translated into DataFrame operations in order to process a query in Spark SQL. Furthermore, *Spark Streaming* was developed to support stream-processing within Spark [ZDL+12].

When compared to MapReduce, experiments show a performance gain of one up to two orders of magnitude [FHA18].

The architecture[45] of a Spark cluster is given in Figure 2.4. The *driver program* contains the main application. In this application, one must additionally specify a *SparkContext* which holds Sparks cluster configuration (e.g. number of worker nodes, the image which the worker nodes shall execute, the hostname of the cluster manager, etc.). On startup, the driver program provides the *cluster manager* with its SparkContext in order for the cluster manager to start the worker nodes. There are different cluster managers supported by Spark, namely Standalone, Apache Mesos, Hadoop YARN and Kubernetes. We will make use of the last one, the Kubernetes cluster manager. Thus, when the driver program starts, it contacts the Kubernetes cluster manager which will then create Pod objects running containerized Spark workers. Finally, when the driver program submits a Spark application, the application is divided into single tasks which are distributed to the worker nodes. The worker nodes execute the tasks and return the results to the driver program.

2.4.3 Apache Flink

Apache Flink is a stream processing framework which supports both, bounded and unbounded streams [CKE+15]. A stream is a directed acyclic graph consisting of producers and consumers. Producers ingest data like sensor-data into the stream while

[45] `https://spark.apache.org/docs/3.0.1/cluster-overview.html`
[46] See footnote 45

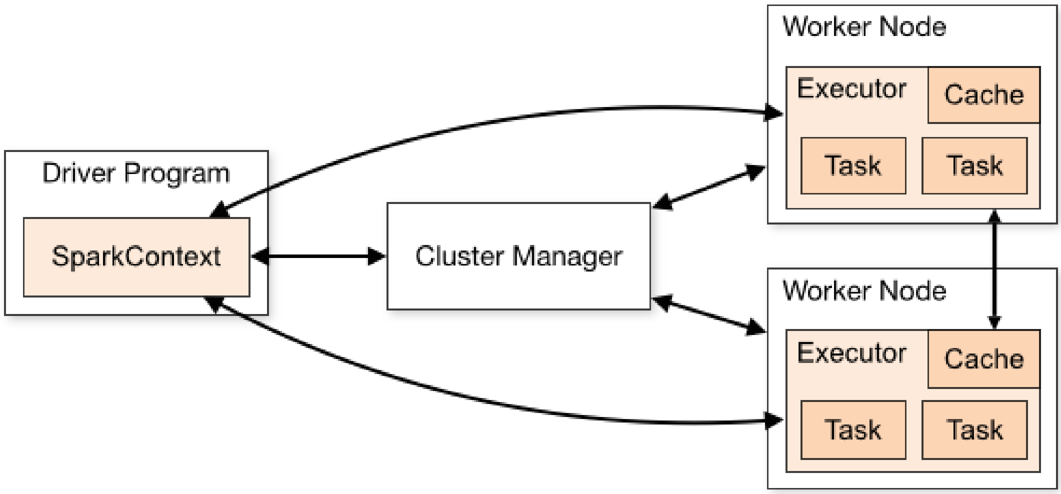

Figure 2.4: Spark Cluster Overview[46]

consumers possibly perform operations on the data or just store the data into a distributed storage (e.g. HDFS). An important feature of Flink is the *exactly one* semantics such that it is guaranteed that a data point which was ingested into the stream is processed by it exactly once, even during node-failures.

Apart from streaming, Flink also contains libraries for graph processing and SQL-like operations.

CHAPTER 3

Use Case: Fraud Detection

Fraud detection applications are gaining much interest for governments and governmental organisations. In 2016, the European Commission estimated that around 150-170 billion euros of value added tax were not collected by the member states of the EU. 50 billion euros of which were allegedly defrauded by criminal organisations[1,2]. In 2019 the amount was estimated to still be around 134 billion euro[3]. Also in other fields like healthcare, it is estimated that alone in the USA, over 30 billion dollar were lost in 2018 due to improper payments which also include fraud attempts [BLR+20]. Fraud detection applications therefore are becoming increasingly important in order to be able to efficiently battle fraud.

In Austria, the governmental *Anti-Fraud Office*, which operates under the Federal Ministry of Finance[4], is responsible for the fight against fraud. The *Financial Police*, a business unit inside the Anti-Fraud Office, is responsible for detecting "tax evasion, social fraud, organised shadow economy and illegal gambling"[5]. Another important business office inside the Anti-Fraud Office is *Tax Investigation* which is responsible for detecting organised tax fraud by detecting and combating fraud schemes.

A criminal process is started when someone reports a potential crime to the authorities or when the authorities themselves become aware of a potential crime[6]. During preliminary

[1] https://www.euractiv.com/section/euro-finance/news/commission-revolutionises-vat-to-tackle-fraud/

[2] https://www.europarl.europa.eu/RegData/etudes/STUD/2021/697019/IPOL_STU(2021)697019_EN.pdf

[3] https://ec.europa.eu/taxation_customs/news/vat-gap-eu-countries-lost-eu134-billion-vat-revenues-2019-2021-12-02_en

[4] https://www.bmf.gv.at/en/the-ministry/internal-organisation/Anti-Fraud-Office-.html

[5] See footnote 4

[6] https://www.oesterreich.gv.at/themen/dokumente_und_recht/strafrecht/1/Seite.2460103.html

proceedings, authorities try to collect enough evidence in order to prove the crime. For this, a search and seizure procedure can be executed during which evidences regarding the crime, like documents and message protocols (e.g. e-mails and chats), are collected[7]. Investigators then examine these evidences in order to prove the crime. When the investigations show that indeed a crime was committed, the authorities bring a charge against the accused subject. During the following trial, the authorities submit their collected evidences in order to prove the crime. Finally, the court decides if the accused subject is guilty based on the evidence provided.

Since data for such legal cases can reach up to some terabytes (for example, 12 terabytes of data were seized during the investigations for the Wirecard bankruptcy [Kob]), this data is preprocessed in order for investigators to be able to browse through the data more efficiently. Relevant data can be all sorts of documents like bills, e-mails, conversation logs, chats (from diverse chat-platforms and different mobile devices) and other unstructured data. In order to combine all that data into one coherent view, software like *Intella*[8] is used in order to automatically extract all sorts of information from this (semi-/unstructured) data and further construct a knowledge graph which then can be efficiently browsed and queried by investigators. Such a knowledge graph is typically stored in a triplestore.

Common tasks in order to construct a knowledge graph include *entity extraction*, where entities like persons or organisations are automatically extracted from text data and *relationship extraction*, which links entities to other entities for instance by simple pattern matching or natural language processing of text data [LQHL17, DGPN13].

For our use case of an application used for "Fraud Detection", we define the following requirements:

1. Application and Framework
   There exists a standalone application which uses the popular[9] Apache Jena framework for storing a knowledge graph as RDF into a triplestore. It therefore is desirable that a distributed triplestore also uses this framework in order to ease the adaption of the existing source code.

2. Data Changes
   Based on new insights, investigators are able to dynamically update the data while they are investigating. They need to have access to the most recent data because outdated information may not accurately reflect the current state of affairs. Therefore, any changes to the data, such as insertions, updates, or deletions based on new insights, must be immediately reflected in subsequent requests in the dataset. Furthermore, since reasoning is applied to the data, it is also important that the reasoner always has access to the latest data. This ensures that the most accurate

[7] https://www.oesterreich.gv.at/themen/dokumente_und_recht/strafrecht/5/Seite.2460404.html
[8] https://www.vound-software.com/pro
[9] https://db-engines.com/de/ranking/rdf+store

and up-to-date insights are used in the investigation process. The data storage system therefore must be designed in a way that it can quickly and accurately reflect changes, ensuring that investigators always have the most current and relevant data at their disposal.

3. Reasoning
   In order to derive further information from the existing dataset, reasoning is applied. We define, that OWL Lite support is sufficient. This language fragment contains RDFS features like classes and properties together with basic inferencing for sub-classes and sub-properties. It also provides more advanced constructs in order to build ontologies suitable for investigations. Furthermore, the ability for investigators to create custom inference rules is crucial, as it allows them to automate the creation of information needed for their investigations.

4. Parallel Access
   Possibly multiple investigators are accessing the data in parallel in order to speed up the investigation. Therefore the triplestore needs to be able to handle parallel access without (much) performance degradation. However, since it is a private system, the number of concurrent accesses to the triplestore is limited.

The existing system shall be transformed into a distributed system in a Kubernetes cloud, since cloud environments have the following, non exhaustive list of characteristics:

1. Elasticity
   Cloud environments provide the ability to easily scale up or down based on the current workload. This applies to both, assigned CPUs and number of workers. Adding more resources, for instance, is relevant when reasoning is applied to the data in order to complete this step faster. Furthermore, the workload increases if many investigators access the system concurrently. Elasticity ensures that the system maintains acceptable performance characteristics even if the workload increases.

2. Cost
   Cloud services typically charge customers based on the resources assigned. Therefore, if the workload is low for a specific case, together with the aforementioned scalability feature, it is possible to save costs by scaling down the resources.

3. Reliability
   The cloud infrastructure typically provides self-healing features in order to automatically recover applications from a failure state.

CHAPTER 4

Evaluation Criteria

We aim to provide assistance in selecting a system for managing large triplestores, with regard to the use case defined in Chapter 3. Due to our use case, we are only considering systems that support OWL-Lite. Furthermore, the source code of these systems needs to be open-source.

In this chapter, functional and performance evaluation criteria will be identified and discussed in order to evaluate systems. In Section 4.1 we will discuss functional evaluation criteria, which determine the characteristics of a system, such as its support for SPARQL, its support for reasoning and the availability of good documentation. Furthermore, in Section 4.2 we will identify important metrics in order to measure the performance of a system. Examples are data load time, query time, and queries per seconds in a concurrent access of the triplestore.

4.1 Functional Evaluation Criteria

In order for users and developers to decide, which triplestore they shall employ, some criteria need to be defined in order to be able to make an informed choice. We identified the following functional evaluation criteria for people to decide which triplestore shall be used for their specific use cases.

1. Framework [KM15]
   The software framework, that a software product is using, provides the foundation for developing software applications on top of it. The choice of framework used by a triplestore is relevant for projects that already use a specific framework to build their existing solution. This can ease integration and reduce development time, as well as make it more efficient for the developers to continue using a framework they are already familiar with.

2. Documentation
   In order to be able to use, adapt and extend the system, the amount and quality of the available documentation is crucial. A survey[1] carried out by GitHub in 2017 found out, that "incomplete or confusing documentation" is the number one problem encountered in open source software. Good documentation therefore can reduce time and cost in order to be able to use and extend the software. Bad documentation, on the other hand, may even deter potential users from wanting to use the product. Furthermore, it may reduce the product's live span since less developers may be willing to put effort in maintaining or extending the source code[2].

3. Storage
   The choice of a storage backend used in a software product is important because each backend has its own advantages and trade-offs. For example, some backends support high availability while others support high consistency. The choice can have significant implications for the performance, scalability and reliability of the software product. Furthermore, also other features, such as support for (automated) backups and licence fees, may also be important for users of a triple store.

4. SPARQL Support
   The supported SPARQL version is important for triplestores because it determines the expressiveness and capabilities of the queries that can be evaluated. SPARQL 1.0 supports significantly fewer features than SPARQL 1.1. Since version 1.1 it is possible to insert, update and delete data from the store. Furthermore, aggregate functions are not supported in version 1.0, which restricts the ability to write complex queries (see Section 2.1.2). Therefore, this criterion is relevant depending on the use case in mind. Furthermore, systems may have limited support for different SPARQL features.

5. Reasoning Fragment
   The supported reasoning fragments determine which information can be derived from the given knowledge base. Systems may support multiple reasoning fragments out-of-the-box, allowing users to decide which reasoning fragment to apply to their data. Users may want to decide which reasoning fragment shall be applied on the data, as different use cases may have time constraints for the triplestore to become accessible (forward chaining) or for queries to return results (backward chaining), see Section 2.1.5.

6. Support for Custom Inference Rules
   The support for custom inference rules allows users to create rules specialized for their use case. This allows users to infer new information from existing data based on their specific needs. For instance, when a knowledge graph contains triples for

[1] https://opensourcesurvey.org/2017/

[2] https://opensource.googleblog.com/2018/10/building-great-open-source-documentation.html

bills and money transactions, one could formulate a rule which states that if there is a transaction for a bill, then the bill has been paid.

7. Compression
   Compression can affect the performance and storage requirements of a system. By reducing the size of the data that needs to be stored and retrieved, compression can alleviate the amount of required storage and improve the performance of the system. This means that slow I/O operations only have to retrieve smaller amounts of data, which can make it faster to access and query the data.

8. Ease of Deployment
   Ease of deployment directly impacts the time required to integrate the system into an existing infrastructure. We use Kubernetes as container orchestration framework. The evaluated systems may already provide a deployment option on Kubernetes. If they do not, we identify the pitfalls when adapting them for Kubernetes.

9. Elasticity
   Elasticity refers to the ability of a system to dynamically scale its resources based on the current workload. This can help to ensure that the system's performance does not degrade during a sudden increase in workload. Furthermore, cloud providers charge customers for allocated resources, so efficient resource utilization may be crucial in order to minimize costs. Therefore it is important to evaluate whether a system is capable of scaling dynamically.

4.2 Performance Evaluation Criteria

We use the popular *Leheigh University Benchmark* (LUBM) [GPH05] for data generation and performance evaluations. It comprises a university ontology with configurable data sizes and is used in order to test various query shapes against the data in order to evaluate the performance of triplestores. LUBM includes a flexible data generator called *UBA*. This tool allows users to specify the number of universities for which they want to generate data. For example, one could configure it to generate data for 20 universities. We use an enhanced version[3] which allows parallelism during data generation in order to improve the data generation performance. Furthermore it supports the generation of data in different formats like N-Triples. We use the default configuration with index=0 and seed=0. We choose this benchmark over WatDiv or BDSM since it specifically assumes OWL Lite support which is required by our use case.

In order to determine the performance of a system, we use the following metrics.

1. Data load time [PCR12].
   Here we measure the data ingestion time up to the moment when the system becomes available for querying. This may also include the time needed for reasoning on the

[3] https://github.com/rvesse/lubm-uba

data. Due to different partitioning strategies and possible network-communication between the nodes, this test is also relevant when determining the scaling factor when increasing the number of worker nodes. In order to do this, we will use the system's web endpoint for inserting data and measure the time until the request is completed. This endpoint will be accessed from a separate Pod in Kubernetes, where the LUBM data, which will be stored in the triplestore, will also be located.

2. Query time [SGK+19][PMNH18].
Here we measure the average and median query time with standardized queries in order to measure the system's response times. For that we use the *UBT* tool of the LUBM benchmark. During the test, the queries are executed sequentially and the average and the median response time of 5 requests per query is taken (following [SGK+19]). The test will be issued from a separate Pod accessing the triplestores web-endpoint.

3. Queries per second and latency [PCR12][SD17].
This metric simulates a concurrent triplestores access from multiple clients in order to evaluate how the system performs under stress when multiple requests are issued simultaneously. We choose 8 as the number of parallel clients, which was also the setup in [PCR12]. We will use Apache JMeter in a separate Pod in order to simulate concurrent client requests to the triplestore.

We will measure several non-functional metrics in order to determine the footprint of the triplestores while the aforementioned tests are performed. To do this, we use the Kubernetes metrics[4] of GCP's cloud monitoring to take our measurements. GCP provides a dashboard to examine the observed metrics and also offers an option to download a metric in a specific time frame. Each data point is measured in a 1 minute interval.

1. Network communication (peak) [PMNH18].
In a distributed triplestore, data is stored across multiple nodes and communication between these nodes may be required for storing and querying the data. As a result, the amount of network communication needed by a triplestore is an important criterion to consider when evaluating its performance. In GCP, we use the "kubernetes.io/pod/network/sent_bytes_count" metric, which measures the bytes sent during a specific interval.

2. Memory usage (peak) [PMNH18].
Memory usage can affect the performance and scalability of a distributed triplestore. Each node in the system has its own memory limitations. Depending on how much data is stored in-memory on a node, it may need to swap data in and out of disk storage, which can significantly slow down its operations. In extreme cases, a node may even run out of memory and crash. As the size of the data stored in the

[4] https://cloud.google.com/monitoring/api/metrics_kubernetes

triplestore grows, the memory requirements for storing and processing data may also increase. Measuring the memory usage during the execution of our benchmarks can provide valuable information about the system's performance and scalability. For measuring the memory usage, we use the "kubernetes.io/container/memory/used_bytes" metric in GCP in order to determine the memory usage during a test.

3. Storage size [RDE+07].
   This metric is used since storage is also a cost factor for a system. Triplestores may improve their performance by storing data multiple times or by providing exhaustive indexing. Some systems may also support data compression to reduce storage requirements. Determining the needed storage for a triplestore is therefore an important value to consider. We measure this value by observing the "kubernetes.io/pod/volume/used_bytes" metric in GCP.

For a detailed performance comparison, the systems will be compared using the following setups:

1. LUBM 1 and 20 [SSGL19b].
   We will use LUBM with 1 and 20 universities in order to evaluate how each triplestore will adapt to various data sizes. Larger datasets will not be evaluated since our candidate systems already claim that they can be used for large datasets (see Sections 5.1, 5.2 and 5.3). Also, large data sizes would be infeasible when evaluating with only few and weak worker nodes whose settings will be introduced in the next items. We use the N-Triples format for the generated files. The data size of these LUBM settings is given in Table 4.1. It can be seen, that both in terms of the number of triples and size, the LUBM 20 dataset is about 27 times bigger than the LUBM 1 dataset.

2. Number of worker nodes: 1, 2 and 4 [SGK+19].
   Testing the horizontal scalability of a distributed triplestore by performing tests on 1, 2 and 4 worker nodes for each LUBM dataset can help to determine how well the system scales in terms of performance when adding more workers. If a system scales well, it may be desirable to add more worker nodes in order to improve performance, depending on the use case.

3. CPU cores: 1, 2 and 4 [Daw16].
   The vertical scalability of a triplestore is evaluated by varying the number of CPU cores of each worker. This allows us to evaluate how well the system scales in terms of performance when adding more cores to each node. In some cases, horizontal scaling may result in the network communication between nodes reaching the network's upper limit capacity. To mitigate this, it is possible to scale vertically by adding more cores to worker nodes. Also here, if a system scales well vertically, adding CPU cores to workers may be desirable depending on the use case. We evaluate configurations with 1, 2 and 4 cores.

LUBM	Triple Count	Size
1	103076	18MB
20	2781362	491MB

Table 4.1: LUBM data sizes

Each node will have assigned as much memory, as the framework documentation of each system suggests. When we encounter out-of-memory errors, we adjust the assigned memory and repeat that test.

In order to measure the performance of parallel clients, we use Apache JMeter[5] which can issue our queries in parallel by using threads.

We perform our performance measurements in the *Google Kubernetes Engine (GKE)* in an "autopilot" cluster. We chose the compute class for our cluster to be "Scale-Out" in order to be able to choose our CPU platform[6]. To the end, we configure our cluster to use AMD EPYC 7B13[7] with a base frequency of 2.45GHz by setting the CPU architecture to "amd64". Furthermore, GKE specifies no limit for the network bandwidth between internal nodes[8].

[5]https://jmeter.apache.org/
[6]https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-compute-classes
[7]https://cloud.google.com/compute/docs/cpu-platforms
[8]https://cloud.google.com/vpc/docs/quota#per_instance

CHAPTER 5

Candidate Systems

5.1 Apache Rya Accumulo

Apache Rya is a distributed RDF store which is based upon a NoSQL store such as Apache Accumulo or MongoDB. It was promoted[1] to a Top-Level Project of the Apache Software Foundation in September 2019 and therefore is a very promising triplestore solution. We will use Accumulo as backend in this chapter which is the original storage backend for Rya (**R**DF **y**(and) **A**ccumulo[2]).

It has been shown that Rya can answer most LUBM queries for the LUBM 15000 dataset in under a second even with multiple clients accessing the store concurrently [PCR12, PCR15]. However, they used a strong setting with 10 Hadoop DataNodes and 10 Accumulo Tablet Servers, each node having 8 cores with 16GB of memory. Thus it occupies many computing resources even when the store is not being accessed.

5.1.1 Storage

Accumulo[3] is a distributed NoSQL store which is modeled after Googles Bigtable [PCR12]. It uses tables to store key-value pairs and uses HDFS as underlying storage. Also data compression is supported.

Each row in an Accumulo table is lexicographically sorted by key, enabling fast retrieval of a row [KAB+14]. In order to distribute the data on different nodes, tables are split into possible multiple sub-tables, called *tablets*. These tablets are finally distributed across the available HDFS DataNodes.

[1] https://blogs.apache.org/foundation/entry/the-apache-software-foundation-announces56
[2] https://github.com/apache/rya#overview
[3] https://accumulo.apache.org/1.10/accumulo_user_manual.html

In order to create rows for Accumulo in Rya, RDF data points are translated to key-value pairs according to Table 5.1 [PCR12]. Hence the whole triple is saved into the *Row ID* part of the key. The *Qualifier* and *Value* fields stay empty. The benefit of storing the whole data in the Row ID is that Accumulo sorts and partitions its data based on that column. This results in a faster access for data which is probable to be accessed together. For instance all triples regarding a subject "<http://example.org/example#Bretterbauer>" are stored in close proximity because of the lexicographical ordering of the Row ID. Furthermore because of the grouping based on the Row ID, those triples also have a high probability to be stored on the same tablet server. An example of data partitioning can be seen in Figure 5.1. There, a single table is split into four tablets, where it is ensured that all columns for a particular row can be found on the same tablet. These tablets are then persisted on (possibly) different Tablet Servers. Finally Rya employs three table indices, namely *SPO, POS, OSP* in order to improve querying performance. These indices are easily created by ordering "subject,predicate,object,type" in the Row ID according to the given index. For example, the Row ID for the POS (= Predicate-Object-Subject) table index is built by concatenating "predicate,object,subject,type".

Key					**Value**
Row ID	Column			Timestamp	
	Family	Qualifier	Visibility		
subject,predicate,object,type	graph name		visibility	timestamp	

Table 5.1: Key-Value pair in Accumulo [PCR12]

Accumulo basically consists of three components, a *master server*, at least one *tablet server* and at least one *garbage collector*. An architecture diagram can be found in Figure 5.2 [PKAS17].

The master server is basically responsible for balancing the load across potentially multiple Tablet Servers[5]. This is done by distributing tablets to different Tablet Servers. It also handles client requests for creating, editing and deleting tables. Furthermore the master server is responsible for reacting to failures in Tablet Servers and for their recovery.

Tablet Servers represent the worker nodes and are responsible for managing a subset of all tables. They basically handle writes and reads from clients. Writes are performed initially in a write-ahead log in order to be able to recover from node failures. This log is periodically flushed to HDFS.

Zookeeper maintains configuration information and provides distributed synchronization. It is also used to distribute[6] a secret key to the master and tablet servers. Accumulo services require this secret to create and verify delegation tokens. These tokens are

[4] https://accumulo.apache.org/1.10/accumulo_user_manual.html#_data_management

[5] https://accumulo.apache.org/1.10/accumulo_user_manual.html#_architecture

[6] https://accumulo.apache.org/1.10/accumulo_user_manual.html#_delegation_tokens_2

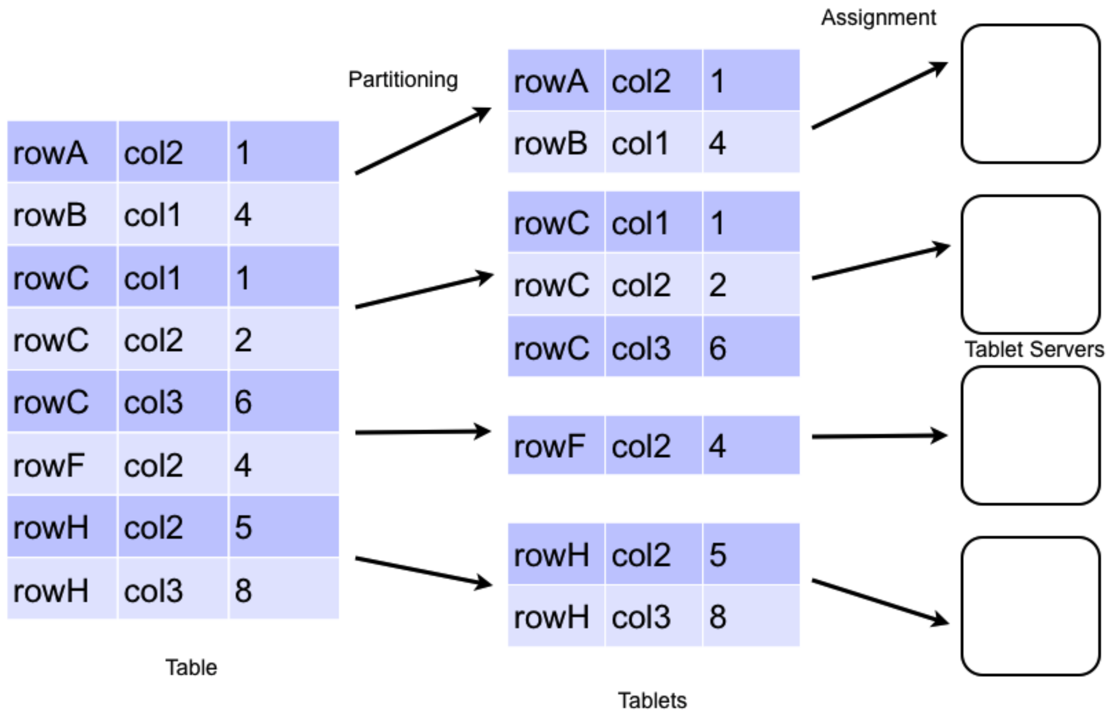

Figure 5.1: Accumulo Data Distribution[4]

needed for authentication and authorization of users. The master for instance requires to know whether a user is authorized to create or delete tables, while tablet servers need to know if a user is allowed to write data.

Further components include garbage collectors which periodically identify files in HDFS which are no longer needed and delete them from the system. One can also deploy *tracers* and *monitors* where tracers write timing information to Accumulo tables and monitors provide statistics about the Accumulo database. Finally, clients can interact with Accumulo over Zookeeper.

5.1.2 Query- and Inferencing

Query processing in Rya is done by using the Eclipse RDF4J framework, which is the successor of the OpenRDF Sesame framework [PCR12]. For simple querying, a SPARQL query is translated into a query evaluation plan and the best suited index-table(s) to lookup from are selected.

For example, when querying all students who study computer science at TU Vienna, one could formulate the SPARQL given in Listing 5.1. The resulting query pattern for the first part of the query is (*,ex:studiesAt,ex:TUVienna) and for the second part (*,rdf:type,ex:ComputerScienceStudent). Due to the query patterns being (*,p,o),

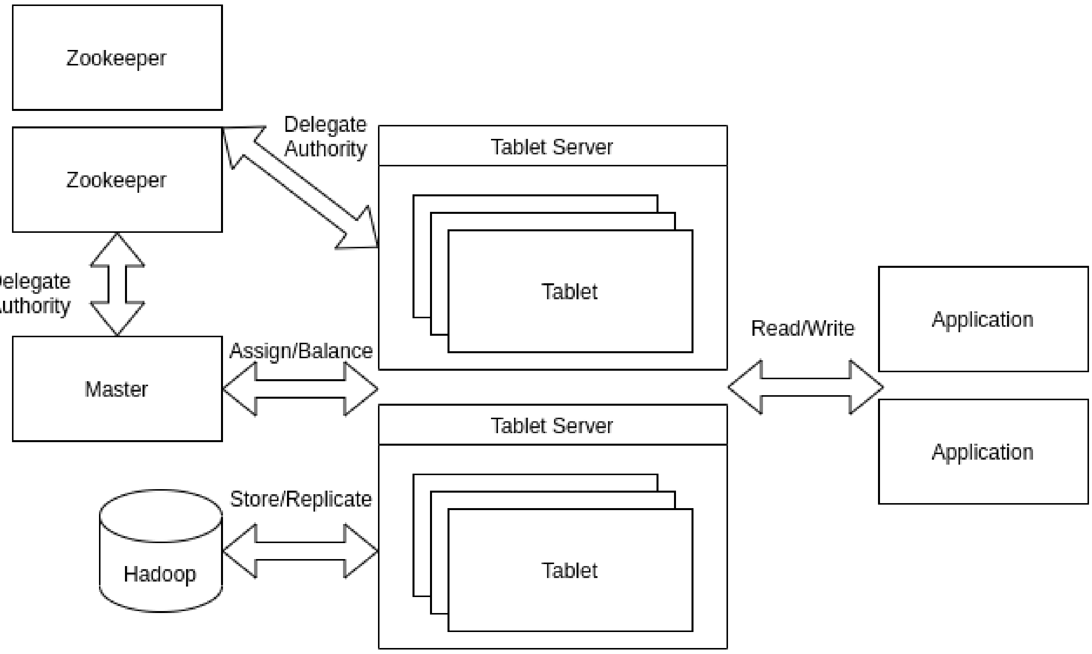

Figure 5.2: Accumulo Architecture Overview [PKAS17]

Accumulos POS table is used [PCR12]. First, a range scan on the key "ex:studiesAt, ex:TUVienna" is performed on that table where all students, which study at TU Vienna, are returned. Then, a simple scan for row-existence is performed for all students since the pattern becomes (?student,p,o), where ?student is a bound variable from the previous step. These queries are thus performed on the SPO table.

```
PREFIX ex: <http://example.org/>

SELECT ?student WHERE {
  ?student ex:studiesAt ex:TUVienna .
  ?student a ex:ComputerScienceStudent .
}
```

Listing 5.1: SPARQL Example

The tablet servers of Accumulo process requests using an iterator framework which also allows tablet servers processing entries in parallel [SOTY13]. The (filtered) results of all tablet servers are sent and and finally combined at Rya.

Further query performance improvements are accomplished by using parallel joins and also a "Batch Scanner" was implemented, which improves access of tables. Also statistics about data in order to improve the ordering of joins are collected.

When data is loaded into Rya, it triggers MapReduce jobs in order to infer additional relationships and stores the resulting triples into the database [PCR15]. These jobs run as long as new relationships are found and terminate otherwise. Furthermore, query expansion is used for some inferencing rules during querying.

An alternative to using MapReduce was proposed in [PCR15]. They extend the TinkerPop Blueprints implementation of the OpenRDF Sesame Api in order to construct and cache the resulting inferred graph in-memory at the master node of Rya. Apache TinkerPop[7] is a graph computing framework which uses *Gremlin* as query language. This reduces the amount of time needed for finding relationships in the data since Accumulo does not need to be queried and query expansion can be done locally.

Rya exposes endpoints to load and to query (via SPARQL) data. In order to import data in the N-Triples format, we use the endpoint "/web.rya/loadrdf?format=N-Triples". The data itself is sent via a POST request and with "Content-Type: text/plain" to this endpoint. Querying is done using the endpoint "/web.rya/queryrdf?query.infer=true&query=<URL-encoded query>". The path states that inferencing is enabled during querying.

5.1.3 Deployment

In order to deploy the system we must consider the following components: Apache Zookeeper, Apache Hadoop, Apache Accumulo and finally Apache Rya. Rya, in time of this writing in version 4, only supports Accumulo in version 1, which itself only supports Hadoop in version 2.

We use the already dockerized Apache Zookeeper 3.4[8] without modifications. For HDFS in version 2, we use the Hadoop-Stack Docker images from the Big Data Europe initiative[9] and translated their "docker-compose.yml" into corresponding Kubernetes objects. The default heap size for Hadoop DataNode is 1GB, therefore we set the memory for that Pod to 1GB and the number of vCPUs to 0.25 since each core should be able to handle up to 4 disks[10]. They define volumes for NameNodes, DataNodes and history servers, therefore we create StatefulSets for those services. For the other services, we create Deployments. Furthermore they define port bindings which we translate into Kubernetes Services. The service object for the NameNode as example is given in Listing 5.2. Additionally they define "SERVICE_PRECONDITION"s in that YAML file, which we translate into Kubernetes init-containers in order to check if the preconditioned service has already been started. This is done by utilizing the *BusyBox* docker image in version 1.32[11] by issuing a simple nslookup for the preconditioned service. An example for such an init-container is given in Listing 5.3.

[7] https://tinkerpop.apache.org/
[8] https://hub.docker.com/_/zookeeper
[9] https://github.com/big-data-europe/docker-hadoop
[10] https://accumulo.apache.org/1.10/accumulo_user_manual.html#_hardware
[11] https://hub.docker.com/_/busybox

```
kind: Service
apiVersion: v1
metadata:
  name: namenode
  labels:
    k8s-app: namenode
spec:
  ports:
    - name: tcp-9870-9870-ngr2r
      protocol: TCP
      port: 9870
      targetPort: 9870
    - name: tcp-9000-9000-b8kj4
      protocol: TCP
      port: 9000
      targetPort: 9000
    - name: tcp-50070-50070
      protocol: TCP
      port: 50070
      targetPort: 50070
  selector:
    k8s-app: namenode
  type: ClusterIP
  clusterIP: "None"
  sessionAffinity: None
```

Listing 5.2: NameNode Service

To deploy Accumulo, we download version 1.10 and encapsulated it into a docker image together with Java 8, Zookeeper 3.6.2 (client) and Hadoop. As base image we choose "centos:7"[12]. Hadoop and Zookeeper are needed inside the image since the root directories of those frameworks are referenced in Accumulos "/conf/accumulo-site.xml" and "/conf/accumulo-env.sh". During the docker-build we also build Accumulos native library for performance improvements[13]. We configured accumulo to use up to 3GB of heap-storage per instance. Furthermore, since we only need to be able to scale the workers, we encapsulated all accumulo-components, except for the tablet-server (slave) into a single docker image. For the tablet server, another docker image was created. Their difference lies in their startup and the specification of the location of other services. For Accumulo one specifies the hostnames of all slaves in a dedicated "slaves" file. In order for that to be configurable in Kubernetes, we create another ConfigMap which

[12] https://hub.docker.com/_/centos

[13] https://accumulo.apache.org/1.10/accumulo_user_manual.html#_native_map

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  spec:
    template:
      spec:
        initContainers:
        - name: init-datanode
          image: 'busybox:1.32'
          command:
            - sh
            - '-c'
            - >-
              until nslookup namenode;
              do echo waiting for namenode; sleep 2;
              done
(...)
```

Listing 5.3: Init-Container example

is injected into the master node of accumulo. Furthermore, since the accumulo-slave is configured for up to 3GB heap usage, the slave-pods memory is set to 3.5GB. The number of cores for the slave-pods are varied during the tests. Finally, we assigned one core and 1GB of memory to the collection of accumulo-components, since most of the work should be done by the slaves.

For Rya, we download version 4.0.1 and compile it with JDK 8. The resulting ".war" file is encapsulated into a docker container together with Zookeeper 3.6.2. As base image we choose "tomcat:jdk8"[14] since the .war file needs to be deployed in a web-server. Furthermore JDK 8 is included in that base image. We also modified the startup of the tomcat server by setting "-Xms512m" and "-Xmx16384m" in order to allow the application to use 512MB up to 16GB of memory. The upper limit was a setting used by [PCR12]. Furthermore we set the number of cores to 2. In Kubernetes we also specified an init-container for Rya in order to wait for all accumulo-slaves to be ready. This is necessary since Rya creates all tables during startup and no rebalancing occurs when a new slave is being started afterwards.

The resulting Kubernetes deployment schema is depicted in Figure 5.3. However, for the purpose of simplification, we omitted the corresponding Kubernetes objects for the remaining Hadoop and Zookeeper components in that diagram. The Apache Rya webserver is connected to the Accumulo master node over the corresponding Service object. The Accumulo master reads from its attached ConfigMap the hostnames of the

[14] https://hub.docker.com/_/tomcat

Accumulo slaves and connects to them also via the regarding Service object. Finally, the slaves connect to the Hadoop DataNodes (by fetching the hostnames from Zookeeper) which all have PersistedVolumeClaim objects assigned. Finally, PersistedVolumes are assigned to match the requirements of the PersistedVolumeClaims where the data of the DataNodes is stored.

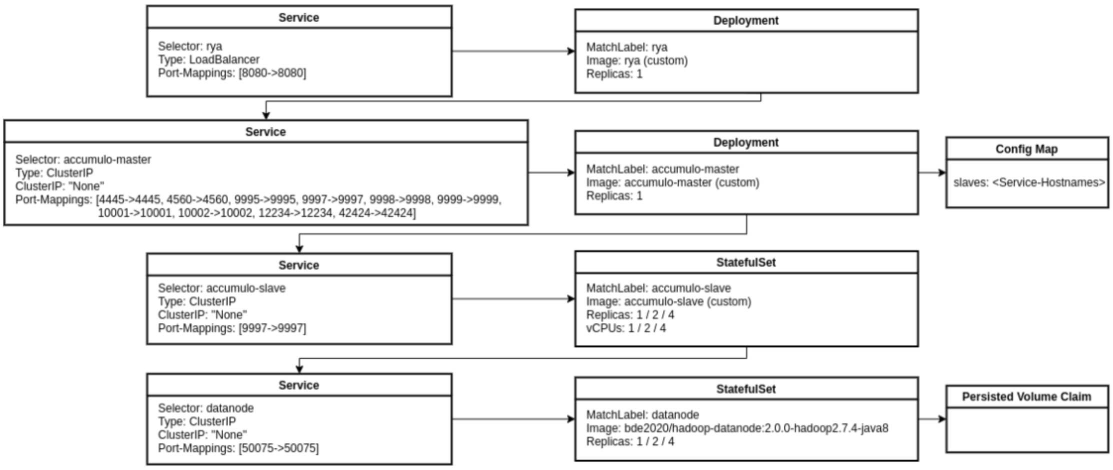

Figure 5.3: Apache Rya Accumulo Deployment

5.1.4 Functional Evaluation

In this section we evaluate the functional characteristics of Apache Rya Accumulo, according to our definition in Section 4.1.

1. Framework
   The system uses the Eclipse RDF4J framework as foundation to process RDF data. Therefore, the currently used system of our use case would need to be adapted in order to support also the RDF4J framework.

2. Documentation
   The documentation[15] of Apache Rya contains useful examples on how to deploy, load and query data. Also source code examples are provided. Therefore, we find the documentation of this system to be useful.

3. Storage
   Apache Accumulo uses Hadoop as background storage, which is a highly consistent[16] framework. That means that always the latest data is seen by a user of the system. Also, regarding partition tolerance, the NameNodes and DataNodes are designed

[15]https://github.com/apache/rya/blob/master/extras/rya.manual/src/site/markdown/index.md

[16]https://hadoop.apache.org/docs/r3.3.6/hadoop-project-dist/hadoop-common/filesystem/introduction.html#Consistency

to be highly available[17], which means that the system continues to function even when some nodes cannot communicate among themselves any more. However, regarding availability, the system does not always guarantee an answer to a request[18]. Regarding our use case this would meet our requirements.

4. SPARQL Support
   Apache Rya fully supports SPARQL 1.1 [PCR15]. Therefore there are no limitations for our use case.

5. Reasoning Fragment
   Apache Rya only supports simple reasoning including seven rules[19]. For our use case, at least OWL-Lite support is required.

6. Support for Custom Inference Rules
   Apache Rya Accumulo does not support custom inference rules. In order to support them, one could extend the InferenceEngine[20] in Rya.

7. Compression
   Data compression is supported for Apache Rya Accumulo[21]. The supported[22] compression types are: gz[23], snappy[24] and lzo[25]. Therefore it is assumed, that data can be stored in a compact form.

8. Ease of Deployment
   Apache Rya Accumulo is relatively hard to deploy, which also lies in the amount of components working together. As described in Section 5.1.3 there are four independent software products involved, namely Apache Zookeeper, Apache Hadoop, Apache Accumulo and Apache Rya itself. Together with several version requirements of the components among themselves in the deployment, it can be quite cumbersome to do it successfully. Even more, as each of these components needs at least one Kubernetes object in order to function. Therefore, it is not easy to be deployed in Kubernetes.

9. Elasticity
   Finally, when evaluating the elasticity characteristic of the system, we find that it

[17] https://hadoop.apache.org/docs/r3.3.6/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html

[18] https://hadoop.apache.org/docs/r3.3.6/hadoop-project-dist/hadoop-common/filesystem/introduction.html#Operations_and_failures

[19] https://github.com/apache/rya/blob/master/extras/rya.manual/src/site/markdown/sm-infer.md

[20] https://github.com/apache/rya/blob/master/sail/src/main/java/org/apache/rya/rdftriplestore/inference/InferenceEngine.java

[21] https://accumulo.apache.org/1.10/accumulo_user_manual.html#_introduction

[22] https://accumulo.apache.org/1.10/accumulo_user_manual.html#_table_file_compress_type

[23] https://www.gnu.org/software/gzip/

[24] http://google.github.io/snappy/

[25] http://www.oberhumer.com/opensource/lzo/

somewhat lacks to be able to dynamically scale because of the following reasons. Accumulo stores the hostnames of all tablet servers in a single slaves-file (see Section 5.1.3) which is injected into the master node of Accumulo. Thus the number (and hostnames) of the tablet servers needs to be known beforehand and cannot be adapted dynamically. For up- and downscaling this means, that the hostnames would need to be adapted in that file and the Accumulo's master node would need to react to changes in this file. Accumulo 2 already implements such a functionality[26]. It also implements enhancements like rebalancing the tablets of the tablet servers on such cases. Thus it seems to be sufficient to let Apache Rya support Accumulo 2. Then, the master- and worker nodes need to access a shared volume with that slaves-file (in Accumulo 2 it is called "tservers") and upon worker creation and deletion the file needs to be updated, based on the Pods DNS hostname.

As a summary, the triplestore supports many functional features we require for our use case. It has good documentation, suitable storage characteristics, SPARQL 1.1 support and support for data compression. However, it lacks functionality in several aspects for our use case. First, our source code would need to be adapted to support RDF4J. It does not support all OWL-Lite rules and also does not support custom inference rules. The triplestore itself is scalable in terms of adding worker nodes and storage, but it lacks the elasticity feature because of the aforementioned reason. Furthermore, the deployment of the system presents a significant challenge.

5.2 Apache Rya MongoDB

Analogous to Chapter 5.1, in this chapter we will again analyse Apache Rya, but this time with a MongoDB backend. The advantage of using MongoDB in a Kubernetes context is that it does not rely on Hadoop components and Zookeeper which also incorporate a specific order on which the components need to be started. Furthermore, with Kubernetes Service objects, another concept on how to address applications, namely via hostnames, is available instead of using Zookeeper. There has been no performance evaluation on Rya with MongoDB to the best of our knowledge.

5.2.1 Storage

MongoDB is an ACID compliant NoSQL document store which is developed by MongoDB Inc. [MMW+21]. It stores its documents in the *BSON* format, which is an extension and binary representation of the popular JSON format. An example[27] of a *document* is given in Listing 5.4. A document is uniquely identified by an *ObjectId* and can contain multiple embedded documents [MTS+21]. The embedded sub-document in the example is the "personalinfo" part. In order to reference the student from the example in another

[26] https://accumulo.apache.org/docs/2.x/administration/in-depth-install
[27] https://docs.mongodb.com/manual/core/data-modeling-introduction/#document-structure

document, one just needs to include: student_id: ObjectId(”a12bc28dffa13315bcca1a25”) into the other document, where ”student” (in student_id) is the name of the document and the value is the ObjectId of the entry.

```
Document: student

{
  _id: ObjectId("a12bc28dffa13315bcca1a25"),
  studentnumber: "01325562",
  personalinfo: {
    firstname: "Markus",
    lastname: "Bretterbauer"
  }
}
```

Listing 5.4: MongoDB Embedded Documents Example

Apache Rya MongoDB stores triples by assigning the relevant triple parts to corresponding keys[28], see for example in Listing 5.5. Also some more fields are stored, like the hash-value of the corresponding subject value, but we omit these for simplicity reasons. The ”<idValue>” is basically generated by concatenating subject, predicate and object of the triple. The result is then converted into a string representation of the hexadecimal values of each byte of the concatenated string. The other ”<value>”s represent the corresponding parts of the triple. For more details we refer to the SimpleMongoDBStorageStrategy.java file of the source code[29].

```
{
  _id: ObjectId("<idValue>"),
  subject: "<value>",
  predicate: "<value>",
  object: "<value>",
  (...)
}
```

Listing 5.5: MongoDB Triple Document

Apache Rya MongoDB also creates three indices[30]. for its data, namely SPO, POS and OSP in order to improve querying performance. For this, MongoDB provides a ”createIndex()” operation which takes the keys to build an index for as arguments (e.g.

[28]https://github.com/apache/rya/blob/master/dao/mongodb.rya/src/main/java/org/apache/rya/mongodb/dao/SimpleMongoDBStorageStrategy.java
[29]See footnote 28
[30]See footnote 28

"subject", "predicate" and "object"). The values of these fields are then sorted in ascending order and stored in a B-tree for faster access.

Documents are aggregated to form a *collection*, which is comparable to a table in relational databases. These collections however do not need to follow a specific schema. A *database* finally contains potentially multiple collections.

A MongoDB cluster is comprised of a primary node, secondary nodes and an arbiter node [MMW+21], see Figure 5.4. When issuing a write operation to MongoDB, the primary node writes the data and all secondary nodes will replicate the write in order to achieve high availability. The arbiter node is responsible for selecting a new master node in case the current master fails.

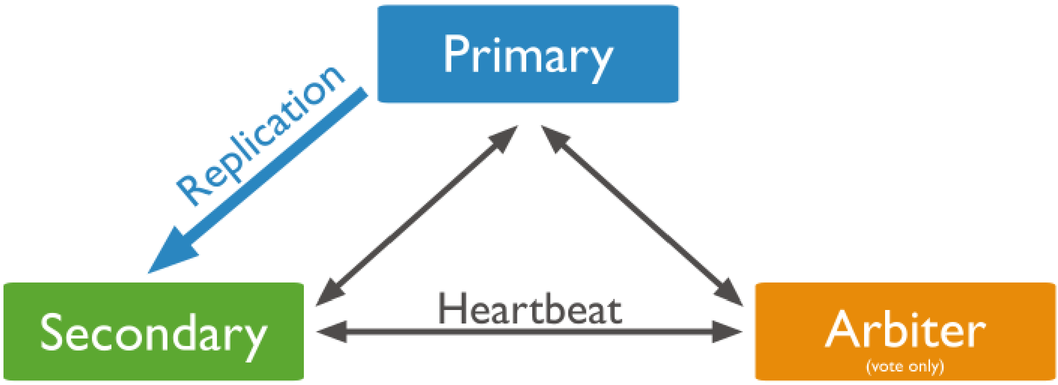

Figure 5.4: MongoDB Cluster [MMW+21]

MongoDB has support for a concept called *sharding* where the data is divided into subsets and these are then distributed across multiple *shards* [MMW+21]. The big advantage of sharding is that when requesting data which resides on a single shard, the system only has to lookup a subset of the data decreasing the response time[31]. Also the overall storage usage is decreased since it is not necessary for each shard to hold the whole dataset and an arbitrary number of shards can be added to the system. A disadvantage of sharding is that queries are more complex to handle across multiple shards and there needs to be a server which merges all the results into a single result set.

5.2.2 Query- and Inferencing

Since Apache Rya MongoDB also uses the Eclipse RDF4J framework, we refer to Section 5.1.2 for details on query building and index selection. Analogous to Apache Rya Accumulo, Apache Rya MongoDB uses iterators[32] in order to collect the results of a

[31] https://www.mongodb.com/features/database-sharding-explained

[32] https://github.com/apache/rya/tree/master/dao/mongodb.rya/src/main/java/org/apache/rya/mongodb/iter

query. The web endpoint in order to load and query data is also the same as with Apache Rya Accumulo.

However, the documentation itself lacks information on how Apache Rya MongoDB exactly handles joins and how it performs inferencing.

5.2.3 Deployment

For the deployment of MongoDB, we use the official MongoDB Community Kubernetes Operator[33]. For the database instances, it can use every MongoDB docker instance[34] available. Furthermore it features the creation of replica sets together with up- and downscaling. The operator is even able to scale the replica sets during reads and writes. However, sharded clusters are currently not supported by the MongoDB Community Kubernetes Operator.

In order to connect Apache Rya to the MongoDB instance, we adapt its configuration according to its manual[35]. However, the inferencing parameters during querying seemed not to work by the same means as for the Accumulo database backend, thus we modified the source code a little in order to apply inferencing on all queries by default.

For our scaling measurements, we alter the "members" field (1/2/4) in the MongoDBCommunity Custom Resource[36] and also specify resource requests and limits (cpu: 1/2/4, memory: 3000M) for the "mongod" container. However, we did not override the resources for the "mongodb-agent" container.

The deployment scheme can be seen in Figure 5.5. The Deployment of the MongoDB Kubernetes Operator directly manages the Statefulset object according to the beforehand mentioned MongoDBCommunity Custom Resource. This Statefulset creates associated Pods according to its configuration and finally Rya is accessing this Pods via the MongoDB Service object.

5.2.4 Functional Evaluation

In this section we evaluate the functional characteristics of Apache Rya MongoDB, according to our definition in Section 4.1.

1. Framework
   Analogously to Apache Rya Accumulo this system uses the Eclipse RDF4J framework as foundation to process RDF data so the currently used system of our use case would need to be adapted.

[33] https://github.com/mongodb/mongodb-kubernetes-operator/blob/v0.7.1/README.md
[34] https://hub.docker.com/_/mongo/
[35] https://github.com/apache/rya/tree/rya-4.0.1-rc1
[36] https://github.com/mongodb/mongodb-kubernetes-operator/blob/master/config/samples/mongodb.com_v1_mongodbcommunity_specify_pod_resources.yaml

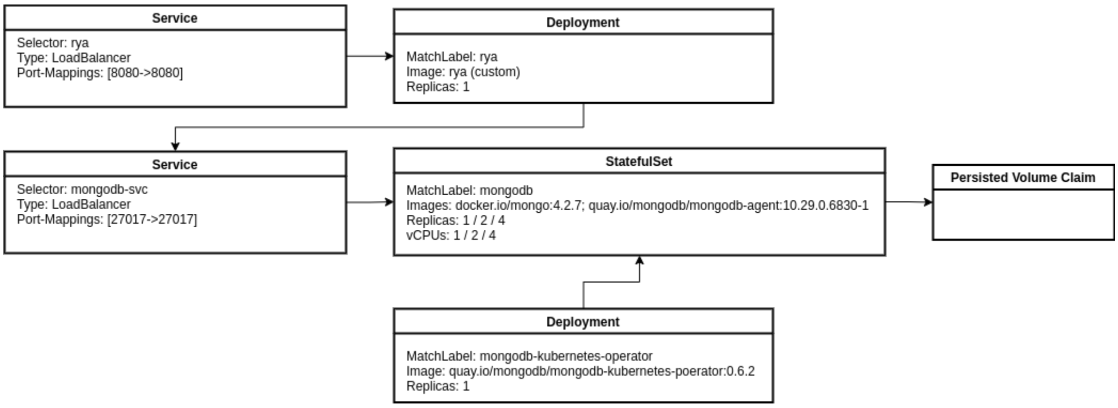

Figure 5.5: Apache Rya MongoDB Deployment

2. Documentation
   The documentation of Apache Rya MondoDB also includes examples on how to configure Apache Rya with a MongoDB backend. Therefore the documentation also provides adequate information on how to install the system. However, there is very sparse documentation how Apache Rya exactly behaves with the MongoDB backend.

3. Storage
   MongoDB per default is a strongly consistent storage[37], which means, that writes are immediately seen by users. Furthermore, if the primary node does not communicate with its secondary nodes for a given period of time, a new primary nodes becomes elected[38], making it partition tolerant. However, it is not guaranteed that the system always answers a request since no upper limit on how long the election process, in order to elect a new primary node in case of a failure, is defined[39]. This would meet our requirements defined in our use case.

4. SPARQL Support
   MongoDB does not limit Apache Ryas support for SPARQL 1.1. Therefore it supports our use case.

5. Reasoning Fragment & Support for Custom Inference Rules
   Regarding the supported reasoning fragments and the support for custom rules, the same statements presented for Apache Rya Accumulo in Section 5.1.4 also apply for this system. Therefore, further development is needed in order to support our use case.

[37] https://www.mongodb.com/jepsen
[38] https://www.mongodb.com/docs/manual/replication/#automatic-failover
[39] See footnote 38

6. Compression
   Data compression is supported[40] by MongoDB. The supported compression types are: snappy[41], zlib[42] and zstd[43]. Therefore, data can be stored in a compact form.

7. Ease of Deployment
   Compared to the Accumulo backend, Apache Rya MongoDB is much easier to configure and to deploy because of the existence of the *MongoDB Community Kubernetes Operator*[44]. However, not all features are supported by this operator. For instance, sharding is only supported with the *MongoDB Enterprise Kubernetes Operator*[45].

8. Elasticity
   Finally, regarding the elasticity feature, the MongoDB Community Kubernetes Operator already supports dynamic horizontal scaling of workers[46]. Therefore, one should only need to define a Kubernetes *Horizontal Pod Autoscaler* in order to support elasticity.

As a summary, this system provides suitable storage characteristics, SPARQL 1.1 support, and support for data compression. Furthermore it is quite easy to deploy and has built-in support for elasticity. However, it lacks documentation and we also would need to adapt our source code in order to support RDF4J. Furthermore only limited reasoning capabilities are supported by the system.

5.3 SANSA-Stack

SANSA-Stack "is a big data engine for scalable processing of large-scale RDF data"[47] which uses Apache Spark and Apache Flink in order to distribute the data for various operations in order to achieve horizontal scalability [LSB+17]. It combines frameworks from the distributed machine learning field with frameworks from the semantic technology field in order to get the benefits of both like horizontal scalability and RDF modelling (see Figure 5.6).

The framework consists of several libraries in order to handle RDF data. These are called *Read/Write RDF/OWL Library*, *Querying Library*, *Inference Library*, *Machine Learning Library* and *Datalake Library*. It uses HDFS (or a local file system) as storage, but also SQL-, NoSQL- and other custom data-sources can be used by utilizing its Datalake Library [MGS+19].

[40] https://www.mongodb.com/docs/v4.4/core/wiredtiger/#compression
[41] http://google.github.io/snappy/
[42] http://www.zlib.net/
[43] https://github.com/facebook/zstd
[44] https://github.com/mongodb/mongodb-kubernetes-operator
[45] https://github.com/mongodb/mongodb-enterprise-kubernetes
[46] https://github.com/mongodb/mongodb-kubernetes-operator#supported-features
[47] http://sansa-stack.net/

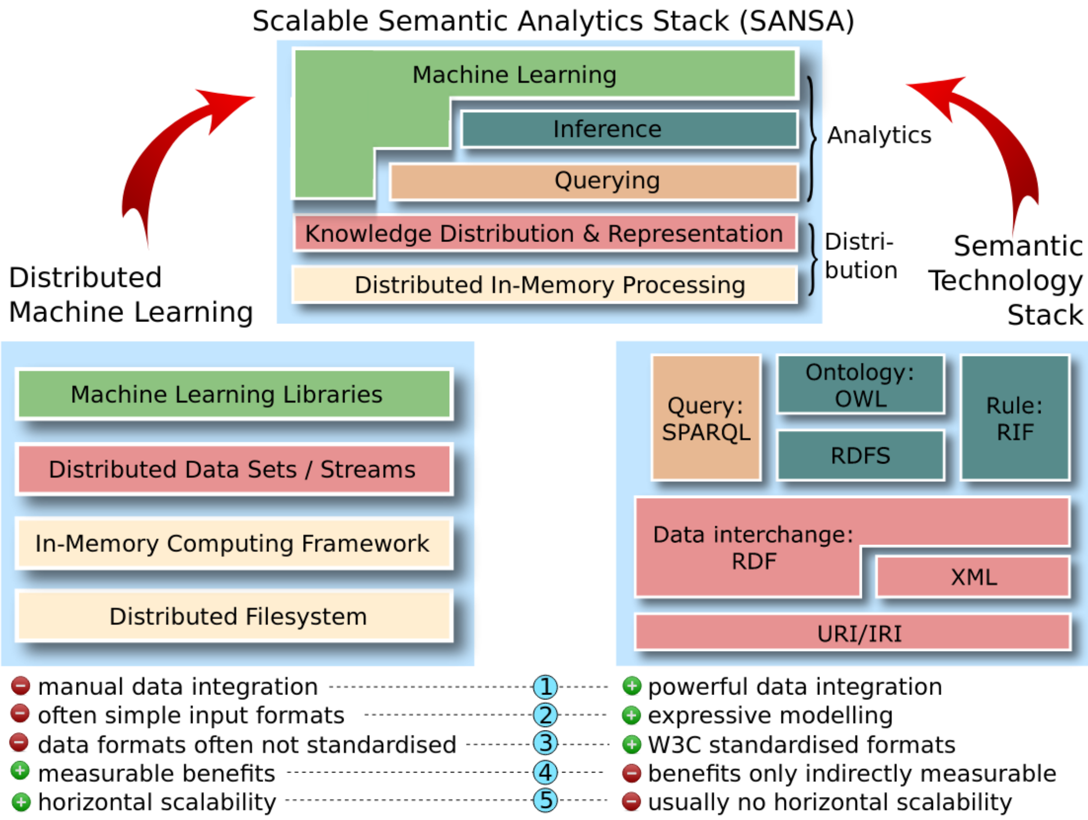

Figure 5.6: SANSA-Stack Vision [LSB+17]

5.3.1 Storage, Query- and Inferencing

By using the Read/Write Library, one can store and read RDF data to and from HDFS (and other sources using its Datalake library). Several serialization formats are supported, like N-Triples, RDF/XML, N quad and Turtle. Data is directly written to HDFS in the specified format. Also data partitioning can be performed with this library. By default, vertical partitioning is applied on the data [LSB+17].

The Querying Library consists of methods in order to transform SPARQL queries into Spark and Flink programs, which can be natively executed by these programs [LSB+17]. These transformations are basically SPARQL-to-SQL transformations based on the SQL dialects used by Spark and Flink. For these transformations, SANSA can use Sparqlify (SPARQL 1.0) and Ontop[48] (SPARQL 1.1) [SSGL19a].

The Inference Library is used in order to apply forward reasoning on the data. Currently RDFS and OWL-Horst rulesets are supported but it is planned to also support more subsets of OWL [LSB+17]. Lastly, the Machine Learning Library encompasses algorithms

[48] https://github.com/SANSA-Stack/SANSA-Stack/releases/tag/v0.7.1

which are designed for graph analysis. However, this library falls outside the scope of relevance for the present thesis.

5.3.2 Deployment

In order to deploy a SANSA-Stack application on Kubernetes, we again deploy Apache Hadoop from the BDE project (analogous to Section 5.1.3) on Kubernetes. For the SANSA-Stack itself we develop a simple web service which provides the needed functionality (storing and querying data) using the SANSA-Stack libraries. It also functions as the Spark driver which creates and manages Spark executors. We use the SANSA-Stack libraries in the latest version available during the writing of this thesis, namely 0.8.0-RC1. As dependencies we need Apache Spark 3.0.1 and the Scala library in version 2.12.10.

Figure 5.7 shows a schematic picture of our SANSA-Stack deployment. The web service will contact the cluster manager (in our case it uses the Kubernetes API) in order to create Spark executors, the worker nodes, during its startup. For the executors, we need to create another Docker image, a Spark image with the required dependencies of the SANSA-Stack libraries in its class path. The number of executors and their number of vCPU cores is varied during the experiments. However, determining the amount of needed memory is not trivial in Spark, especially during inferencing and parallel querying. We will go into detail on this in Section 6.3. Again, the other Hadoop components were omitted in this image for the sake of simplicity.

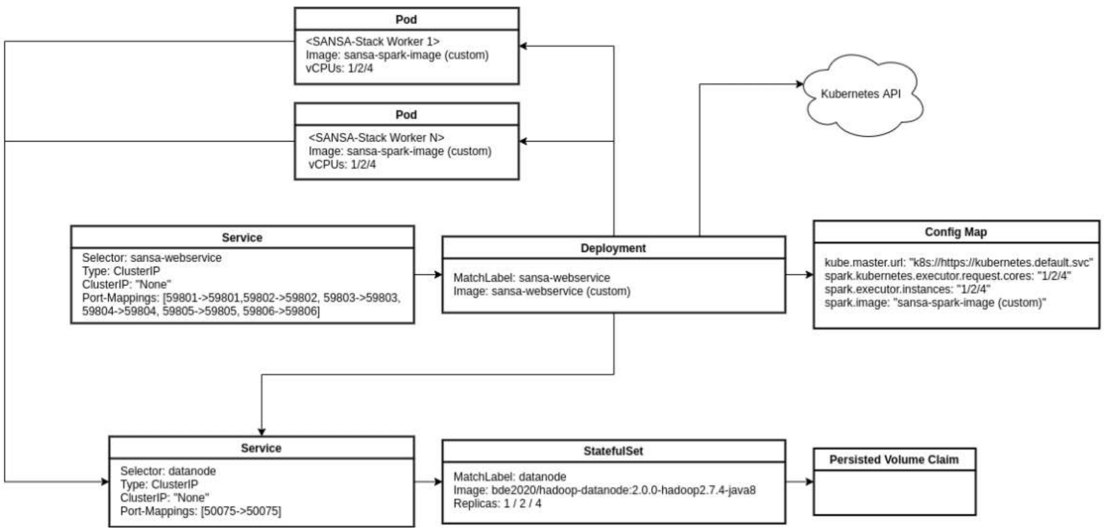

Figure 5.7: SANSA-Stack Deployment

5.3.3 Functional Evaluation

In this section we evaluate the functional characteristics of SANSA-Stack according to our definition in Section 4.1.

1. Framework
   SANSA-Stack uses the Jena framework in order to process RDF data. Since in our use case we also use Jena, we may easily be able to integrate this system into the existing code.

2. Documentation
   The documentation of this system provides useful examples on how to load and query data. Source code examples are provided. However, almost no documentation is given on how to deploy the system in general and specifically in a Kubernetes environment.

3. Storage
   Hadoop is a highly consistent framework with high partition tolerance (see Section 5.1.4) which is very suitable for our use case.

4. SPARQL Support
   Although SANSA-Stack supports SPARQL 1.1 by using Ontop, a SPARQL to SQL rewriter, insertions, updates and deletions via SPARQL are not supported by the system. This would need to be implemented for our use case.

5. Reasoning Fragment
   The system supports RDFS and OWL-Horst rule sets but not OWL Lite. Therefore we would need to extend the systems capabilities.

6. Support for Custom Inference Rules
   SANSA-Stack does not support custom inference rules. But there may be support for custom rules by extending the given rule sets[49]. However, it is uncertain if this would work[50].

7. Compression
   Data compression is supported[51] by SANSA-Stack, although it is not specified to which extent. However, it is therefore assumed, that data can be stored in a compact form.

8. Ease of Deployment
   SANSA-Stack is very hard to deploy since also here many components are working together, namely Apache Zookeeper, Apache Hadoop, Apache Spark and SANSA-Stack itself. Furthermore, there are very strict specifications on which versions of each component to use. Furthermore, as mentioned earlier, the documentation for deploying the system is almost inexistent. Therefore, it is very difficult to deploy the system in Kubernetes.

[49] https://github.com/SANSA-Stack/SANSA-Stack/blob/v0.8.0-RC1/sansa-inference/sansa-inference-common/src/main/scala/net/sansa_stack/inference/rules/RuleSets.scala

[50] https://github.com/SANSA-Stack/SANSA-Stack/issues/112

[51] https://github.com/SANSA-Stack/SANSA-Stack/releases/tag/v0.6.0

9. Elasticity
   Elasticity is not fully supported by SANSA-Stack. Spark currently does not support dynamic resource allocation[52]. Therefore one needs to specify the number of cores and workers beforehand. The Spark Context then is created based on this configuration and is currently not changeable. However, according to their website this is a planned feature.

As a summary regarding our given use case, the system uses the required framework, it has good documentation regarding its usage, suitable storage characteristics and support for data compression. However, it lacks deployment documentation, full SPARQL 1.1 support, insufficient inferencing support, it is very difficult to deploy and lacks the elasticity feature.

5.4 Further Systems

Apart from the mentioned candidate triplestores, there has already been much effort in creating distributed triplestores. For example, the open-source triplestore *Halyard*[53] uses Apache HBase, which is a NoSQL database for Hadoop [SN16], as storage backend. It uses the Eclipse RDF4J framework which supports SPARQL 1.1 queries. The store supports rule inferencing over RDFS. It has been shown that Halyard supports handling petabytes of RDF data. However, the last commit[54] for this project has been on 5th of December 2019. Therefore this project seems to have been discontinued.

Another open-source distributed triplestore is *4store*[55] [HLSL09]. It is comprised of processing nodes which handle parsing of RDF data and SPARQL 1.0 queries, and (multiple) storage nodes which contain non-overlapping portions of the whole data. It supports backward reasoning on a subset of RDFS [SCH$^+$11]. Furthermore it has been shown that this triplestore can handle up to 15 billion triples. However, this project seems also to have been discontinued since the last commit[56] was issued at 28th of March 2017.

The last open-source distributed triplestore we found is *CumulusRDF* [Har11]. This triplestore's backend is Apache Cassandra, another distributed NoSQL database. The system supports SPARQL 1.1 queries, but no information about reasoning capabilities could be found. Also this project seems to have been discontinued. The last commit[57] was issued at 14th of April 2016.

Proprietary distributed triplestores include *Amazon Neptune*[58] [BCG$^+$18]. It is a cloud

[52] `https://spark.apache.org/docs/3.0.1/running-on-kubernetes.html#kubernetes-features`
[53] `https://github.com/Merck/Halyard`
[54] `https://github.com/Merck/Halyard/commits/master`
[55] `https://github.com/4store/4store`
[56] `https://github.com/4store/4store/commits/master`
[57] `https://github.com/cumulusrdf/cumulusrdf/commits/master`
[58] `https://aws.amazon.com/neptune/`

service which is hosted on Amazon Web Services (AWS) and supports SPARQL 1.1. According to the authors, this triplestore is capable to scale for more than 100 billion triples. However, also no information about the support for reasoning could be found for this triplestore.

Another proprietary distributed system is *Virtuoso Universal Server*, a hybrid database system which supports storing multiple data formats like relational data as well as (schema-less) RDF data [Erl12]. It supports SPARQL as well as reasoning for a subset of OWL[59]. A Virtuoso cluster consists of shared-nothing servers in order to be able to scale out. It has been shown that Virtuoso can handle terabytes of RDF triples. Also a free version for this database exists, however clustering is only supported in the proprietary version.

Stardog[60] is another proprietary distributed triplestore system. It is a specialized system consisting of (multiple) *Stardog Servers* and Zookeeper instances[61]. Zookeeper is responsible to maintain a list of cluster members. Clients access the Stardog servers through a load balancer. This triplestore supports SPARQL 1.1 and reasoning for several OWL2 profiles. A Kubernetes deployment option is available[62].

GraphDB[63] is a proprietary, scalable RDF database for which a Kubernetes deployment option is available. The triplestores architecture is a master-worker system where the master is responsible to act as a load balancer between the clients who access the triplestore and the worker nodes. It is also responsible to ensure that the data is consistent between all workers. It supports SPARQL 1.1 together with reasoning on the OWL Horst fragment.

AllegroGraph[64] is a proprietary distributed triplestore which has a Kubernetes deployment option. It supports SPARQL 1.1 and reasoning for a subset of OWL[65] (e.g. OWL 2 RL, RDFS). In this triplestore, the data is sharded and stored in (multiple) specialized servers.

AnzoGraph[66] is a proprietary, horizontally scalable in-memory graph database[67]. The triplestore is comprised of master and worker nodes which access a shared storage (i.e. NFS). A deployment option for Kubernetes is available. It supports SPARQL 1.1 together with reasoning on a subset of OWL 2 RL.

MarkLogic[68] is a distributed multi-model database, which is capable of storing RDF triples (together with documents like JSON/XML or even relational data). It is comprised of

[59] http://docs.openlinksw.com/virtuoso/rdfsparqlruleintro/
[60] https://www.stardog.com/
[61] https://docs.stardog.com/
[62] https://github.com/stardog-union/helm-charts
[63] https://www.ontotext.com/products/graphdb/
[64] https://allegrograph.com/
[65] https://allegrograph.com/products/allegrograph/
[66] https://cambridgesemantics.com/anzograph/
[67] https://docs.cambridgesemantics.com/anzograph/v2.5/userdoc/home.htm
[68] https://www.marklogic.com/

database servers which store the actual data and middleware servers which communicate with the database servers and receive requests via a REST API. It supports SPARQL 1.1 together with inferencing on RDFS and OWL Horst.

Finally, we mention *Dydra*[69] which is a cloud-based database platform. It is hosted by Datagraph GmbH. It supports SPARQL 1.1 but no information about reasoning could be found.

5.5 Functional Discussion

Our functional evaluations of our candidate systems show that none of these systems fulfills every functional evaluation criterion, as seen in Table 5.2. Apache Rya MongoDB fulfills five criteria (plus two partially) out of nine, followed by Apache Rya Accumulo, which fulfills four criteria (plus one partially) out of nine. SANSA-Stack only fulfills three out of nine evaluation criteria. However, it partially fulfills three additional criteria.

All of our candidate systems have strongly consistent storage with high partition tolerance. Furthermore, all support data compression techniques in order to improve performance and storage requirements. On the other hand, none of our candidate systems support custom inference rules and also OWL Lite is not fully supported which limits the reasoning capabilities for our use case.

Apache Jena, our framework of choice, is only used by SANSA-Stack. The other two systems use Eclipse RDF4J. To support the Jena framework in these cases, we would need to adapt the source code.

Regarding the documentation, only Apache Rya Accummulo has good documentation regarding all aspects. The documentation for Apache Rya MongoDB lacks describing its behavior with the MongoDB backend. The documentation of SANSA-Stack completely lacks information on how to deploy the system in general.

SPARQL 1.1 support is given for all systems. However, with SANSA-Stack it is not possible to perform insert, update, and delete queries with its SPARQL implementation. This feature would need to be added for our defined use case.

Of our candidate systems, only Apache Rya MongoDB was easy to deploy because it involves only few components and there are already Kubernetes Operators available in order to further ease the deployment of the system. Apache Rya Accumulo and SANSA-Stack on the other hand both involve many different components with a specific version requirement for these components. Furthermore, SANSA-Stack, as already mentioned, delivers no information on how to deploy the system making these two systems difficult to deploy.

Finally, the elasticity feature is only supported by Apache Rya MongoDB by its MongoDB Community Kubernetes Operator. Apache Rya Accumulo currently does not support

[69] `https://docs.dydra.com/dydra`

elasticity, however it may become supported when it starts supporting Accumulo in version 2. Support for elasticity is also planned in a later version of Spark and SANSA-Stack.

We conclude, that Apache Rya MondoDB is our framework of choice regarding its functional evaluation since it fulfills the most criteria, followed by Apache Rya Accumulo and finally SANSA-Stack.

	Apache Rya Accumulo	Apache Rya MongoDB	SANSA-Stack
Framework	X	X	✓
Documentation	✓	~	~
Storage	✓	✓	✓
SPARQL Support	✓	✓	~
Reasoning Fragment	~	~	~
Support for Custom Inference Rules	X	X	X
Compression	✓	✓	✓
Ease of Deployment	X	✓	X
Elasticity	X	✓	X

Table 5.2: Summary Functional Evaluation

CHAPTER 6

Performance Evaluation

In this chapter we present the performance evaluation results of our observed systems. As already mentioned in Section 4.2, we use the Kubernetes metrics[1] of GCPs cloud monitoring in order to collect our measurements. When measuring the network communication, we use the "kubernetes.io/pod/network/sent_bytes_count" metric. It measures the bytes sent during a specific interval. The memory usage is determined by using the "kubernetes.io/container/memory/used_bytes" metric. For the used storage, we use the "kubernetes.io/pod/volume/used_bytes" metric. Each data point was measured in a 1 minute interval. All tests are conducted within the cloud in order to be independent from a network connection to the cloud.

6.1 Apache Rya Accumulo

Evaluation results for the loading times during the load of the LUBM datasets are given in Figure 6.1. The raw results, which also contain memory-usages, storage-sizes and the network-communication peaks, are given in Table 8.1 in the Appendix. As one can see, the loading time does not benefit when we increase the number of cores or the number of workers. The more workers we add, the more likely it is that the loading time even increases about linearly with the number of workers. The reason for the increase seems to be based in the splitting of its tables into tablets. In Table 6.1, in which we show the LUBM 1 experiment with 2 workers having 2 cores each, the tablets are distributed evenly across the nodes (accumulo-slave-0 with 5 tablets and accumulo-slave-1 with 4 tablets). However, all entries for each index (SPO, POS, OSP) were transferred to the second tablet server which results in the second tablet server holding all the data whereas the first tablet server holding no data (except for some metadata information). In contrast to Table 6.2, which shows the experiment with 2 workers having 4 cores each, the tablets

[1] https://cloud.google.com/monitoring/api/metrics_kubernetes

were distributed differently. One is holding 6 tablets while the other one is holding only 3. However, the data is distributed more evenly since one tablet seems to hold one and the other tablet holding two indices. The loading time however seems to become higher, the more distributed the data becomes (2 workers with 2 cores have a loading time of 1441s, while 2 workers with 4 cores have a loading time of 2687s).

The memory usage during loading however does not depend on the splitting of the tables onto tablet servers, see Figure 6.2 (or Table 8.1 in the Appendix). It mostly depends on the number of workers used in the experiments. For LUBM 1, the memory consumption increases about 22% when increasing the number of workers from one to two and about 31% when increasing the number of workers from two to four in the two-core setting.

In terms of consumed storage, it first about doubles when increasing the number of workers from one to two and about 68% when increasing the number of workers from two to four (see Table 8.1). This is probably due to the default replication factor of three. The number of cores again do not affect the consumed storage.

Interestingly, the network communication does not seem to become greatly affected when we change the number of workers or the number of cores (see Table 8.1). However, the variation of network communication results in the four-worker setting is quite high.

For the LUBM 20 dataset we conducted only few experiments since the loading time in the best case was already about 10 hours and even increased to about 30 hours when testing with four workers. In the original paper, data ingestion into the system was achieved using Accumulo's Bulk Import MapReduce job [PCR12]. We, on the other hand, use the already provided web REST endpoint[2] for loading the data since it seems to be the standard way for Accumulo to load data over the web. This may explain the very long loading times in our experiments.

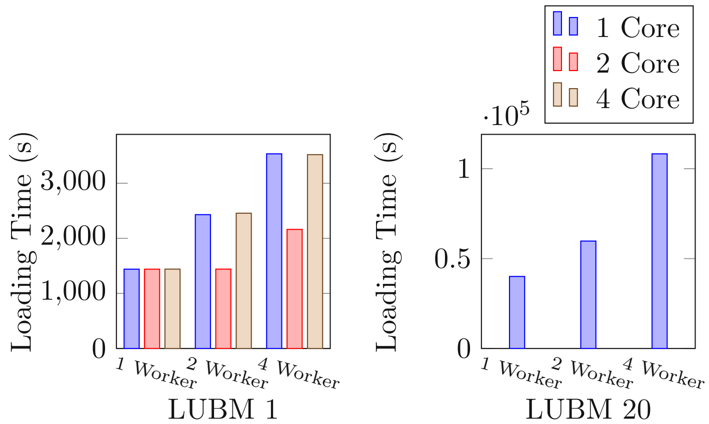

Figure 6.1: Apache Rya Accumulo: Loading Data - Loading Time

[2]https://github.com/apache/rya/blob/master/extras/rya.manual/src/site/markdown/loaddata.md#web-rest-endpoint

Server	Hosted Tablets	Entries
Tablet Server 1	5	187
Tablet Server 2	4	302.51K

Table 6.1: Apache Rya Accumulo: 2 Workers 2 Core

Server	Hosted Tablets	Entries
Tablet Server 1	6	101.11K
Tablet Server 2	3	201.68K

Table 6.2: Apache Rya Accumulo: 2 Workers 4 Cores

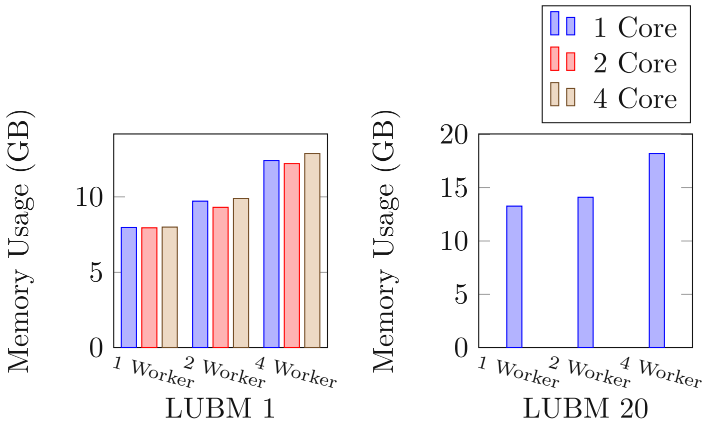

Figure 6.2: Apache Rya Accumulo: Loading Data - Memory Usage

The querying results for LUBM 1 and LUBM 20 are shown in Figures 6.3 and 6.4 respectively. As we mentioned earlier, for the LUBM 20 dataset only few experiments were conducted because of the very long data loading times. For the detailed results, we refer to the Appendix (Tables 8.2 to 8.15). For the LUBM 1 dataset it seems that no setting is clearly the best. Two cores however often seem to clearly benefit the query performance for this dataset which can be seen at the results for the queries: 1, 3, 5, 7, 8, 10, 11, 12, and 13. Four cores in some cases even seem to increase the response time compared to one core. However, when we look at the few results for the LUBM 20 dataset, we see that increasing the worker nodes clearly reduces the query response time for most of the queries when having one core assigned. The queries 2, 5, 7, 8, 9, 10 and 13 however timed out after one hour.

We also measure the memory consumption during the whole run of the benchmark. The results for the LUBM 1 dataset can be found in Figures 6.5 and 6.6. Adding more cores to a single worker configuration affects the memory consumption only a little bit. However, adding more workers to a single core configuration results in a clearly visible increase in memory consumption.

We also conducted an experiment to show how the number of HDFS DataNodes affect the

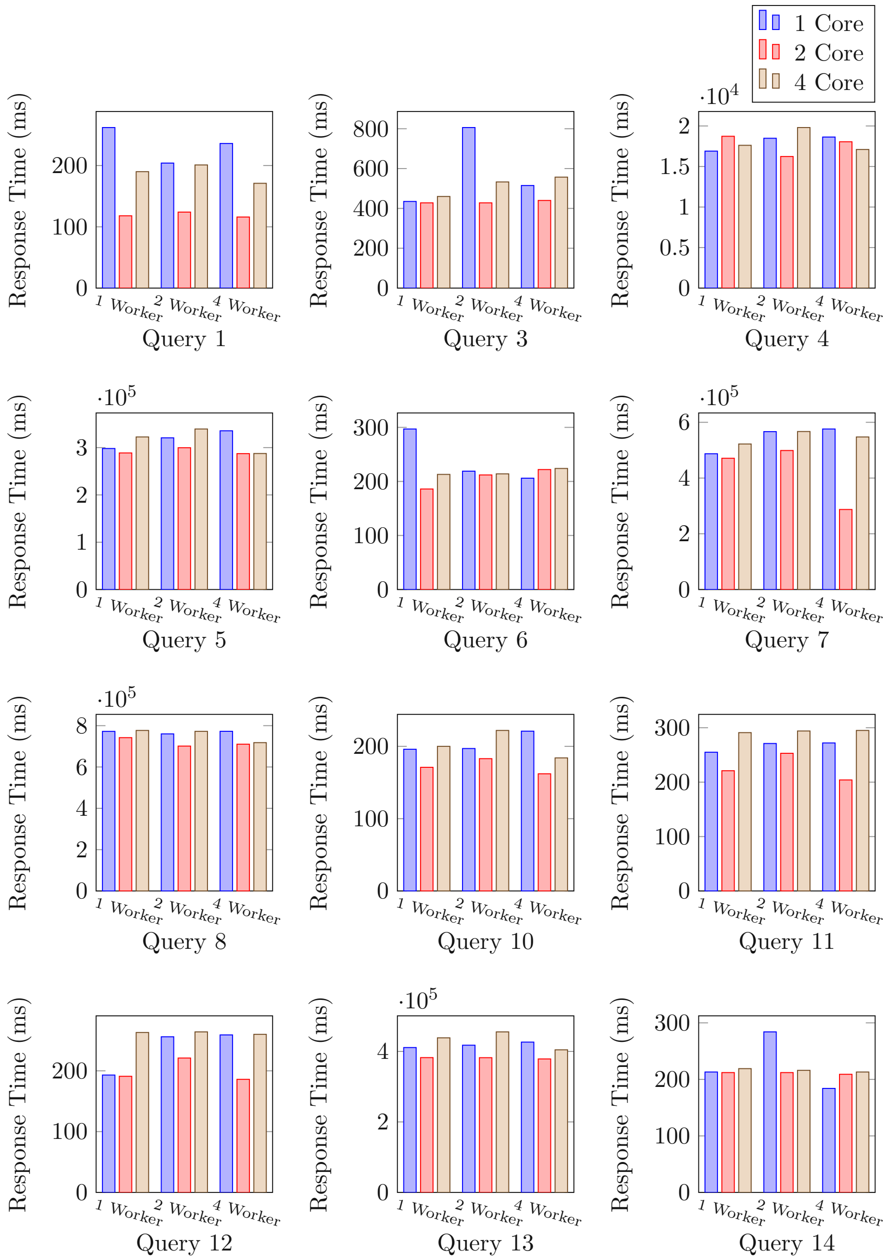

Figure 6.3: Apache Rya Accumulo: LUBM 1 Median Query Response Times

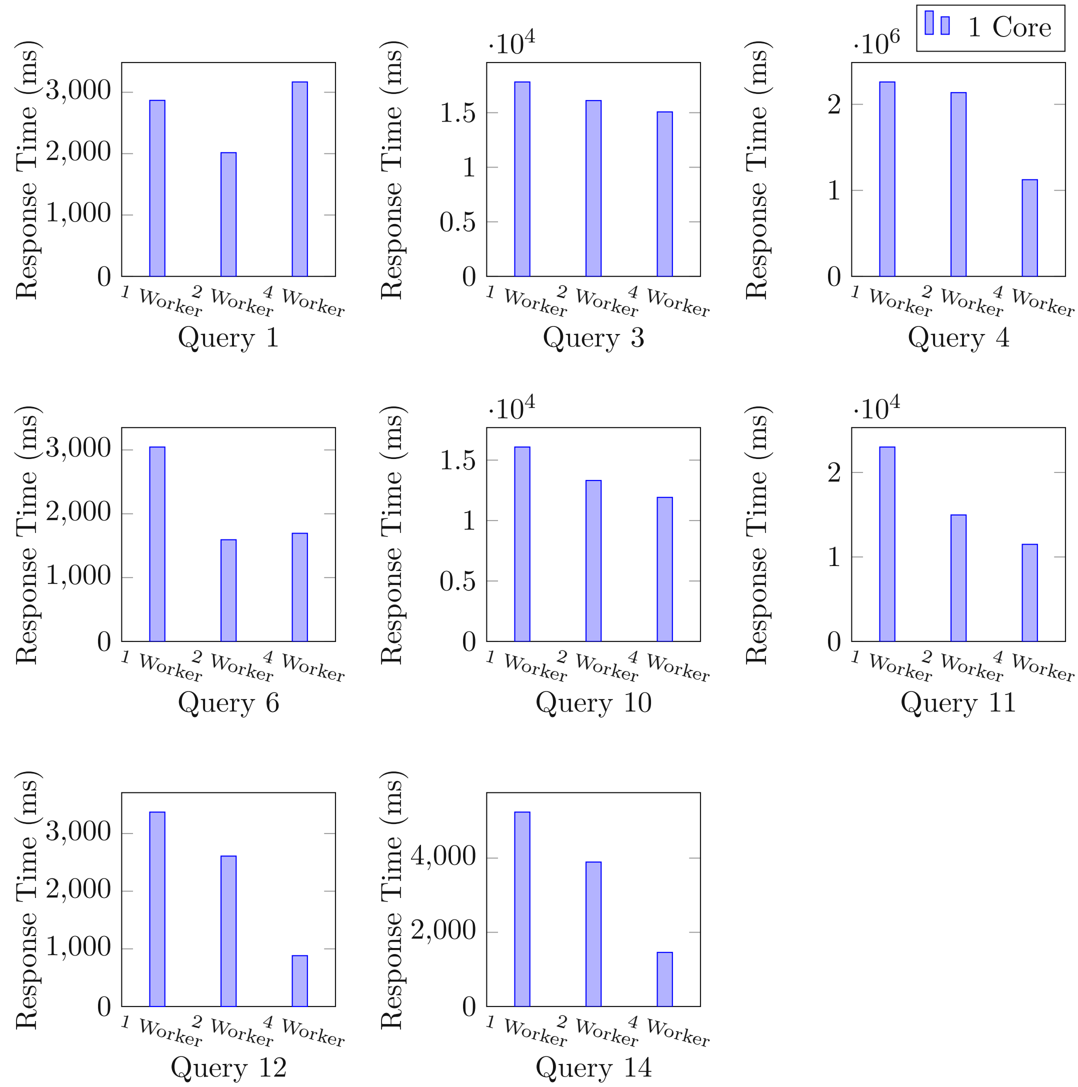

Figure 6.4: Apache Rya Accumulo: LUBM 20 Median Query Response Times

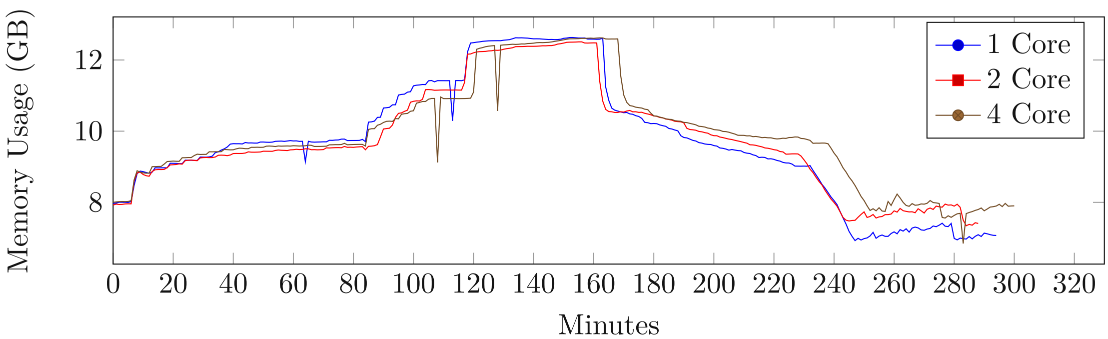

Figure 6.5: Apache Rya Accumulo: LUBM 1 - 1 Worker Memory Usage

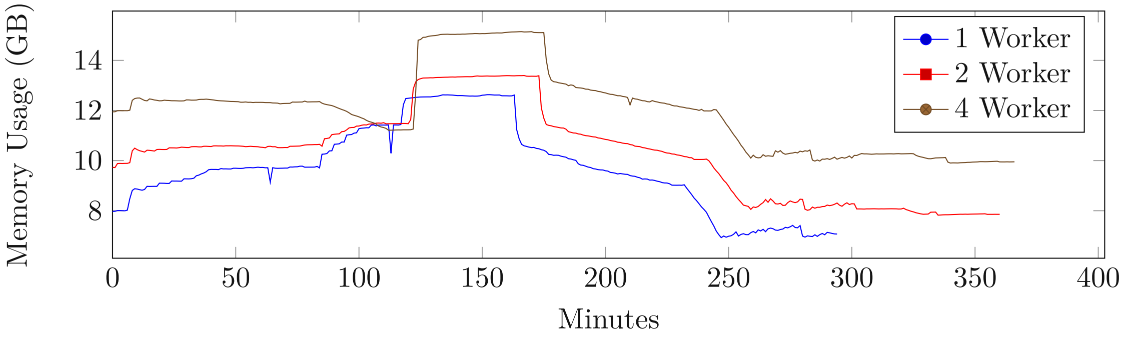

Figure 6.6: Apache Rya Accumulo: LUBM 1 - 1 Core Memory Usage

response times of queries (see Table 8.16 in the Appendix). Each row contains the query and its average- and median response times for one and two DataNodes. In [PCR12] the amount of worker nodes and data nodes were equal, each also having the same amount of CPU cores assigned. However, we set the number of cores for a DataNode to 0.25, because a DataNode with one core can handle up to 4 disks[3] and a datanode in our setting has exactly one disk assigned. The experiment shows that scaling the datanode alongside the worker nodes results in slower response times for most of the queries. Also, the storage size increases with the DataNodes which can be seen in Table 8.1. Since storage is cheap and in order to be comparable to the original measurements, we still increase the number of DataNodes together with the number of worker nodes.

We do not measure the queries per second performance here, since the overlong loading times for bigger datasets already makes this system unsuitable for our use case.

6.2 Apache Rya MongoDB

In order to load the data, we use the same endpoint as for the evaluation with the Apache Accumulo backend.

The result of loading a LUBM 1 and a LUBM 20 dataset into the triplestore can be seen in Figure 6.7. The raw data can be found in the Appendix (Table 8.17). The increase in the dataset size from LUBM 1 to LUBM 20 results in a much higher loading time. For example, when comparing the highest loading times, the loading time for the LUBM 20 dataset is about 60 times higher when having four workers with one core although the dataset being only 27 times bigger (see Section 4.2). When comparing the least loading times, it is still a 38 times increase when having one worker with four cores.

Generally it can be seen, that adding more cores to the system results in a reduced loading time. On the other hand, adding more workers results in a higher loading time. However, when we look at the settings of four cores for each number of workers, we see that the loading times almost equalize. For example, in the LUBM 1 dataset, one worker

[3] https://accumulo.apache.org/1.10/accumulo_user_manual.html#_hardware

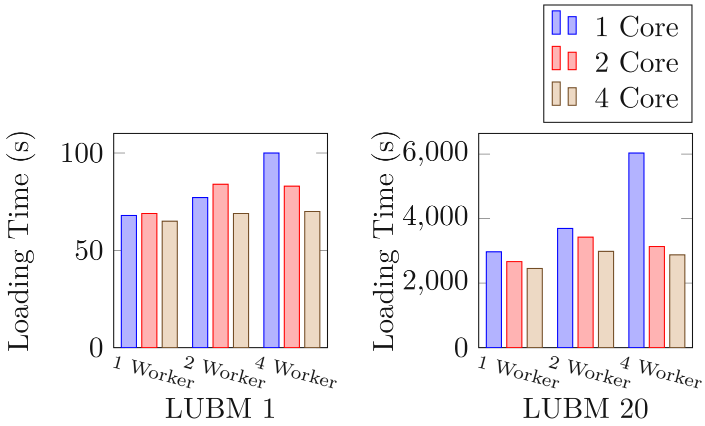

Figure 6.7: Apache Rya MongoDB: Loading Data - Loading Time

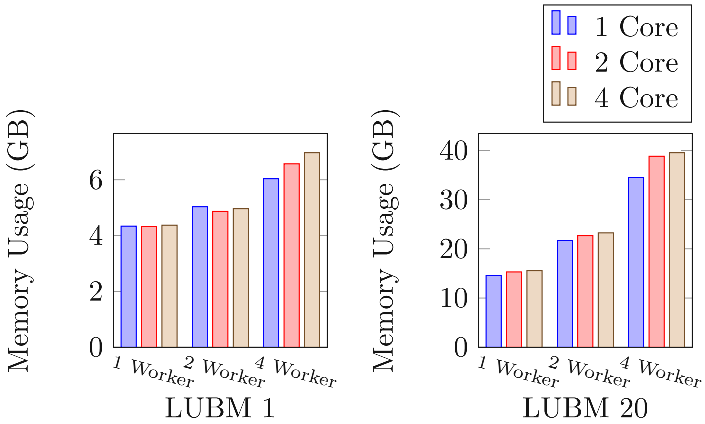

Figure 6.8: Apache Rya MongoDB: Loading Data - Memory Usage

with one core needs 68 seconds to load the data, while four workers need 100 seconds which is a 47% increase. However, one worker with four cores needs 65 seconds to load the data, whereas four workers with four cores need only 70 seconds which results in only an 8% increase of loading time. Similarly, in the LUBM 20 dataset the increase of loading time reduces from 103% to 17%.

The memory usage only slightly increases with the number of cores, see Figure 6.8 (or Table 8.17 in the Appendix). The highest increase can be seen in the settings of four workers when comparing one core per worker with four cores per worker which is about 15% for the LUBM 1 and LUBM 20 datasets. The increase of memory usage is higher when we increase the number of workers. For the highest memory usages (4 worker 4 cores), the increase between one and four workers amounts to about 59% for the LUBM 1 dataset and to 154% for the LUBM 20 dataset.

An almost linear increase can be seen in the storage size when we observe the number of workers. When we increase the number of cores, still a small increase of storage size can be seen.

Finally, when we observe the network communication, the amount of data transferred mostly correlates with the number of workers. For the two cores settings, the increase of network communication amounts to about 28% between one and two workers and to about 77% between two and four workers for the LUBM 1 dataset. The differences for the LUBM 20 datasets are 30% and 76% respectively. We chose the setting with two cores since variating the number of cores results in an inconsistent increase or decrease in terms of transmitted bytes per second.

The results for the query median response time evaluation for the LUBM 1 and LUBM 20 datasets are given in Figures 6.9 and 6.10 respectively. The raw results are again given in the Appendix (Tables 8.18 to 8.31).

The results for LUBM 1 show that there are queries which clearly benefit by adding more cores to the system (queries 5, 8, 10 and 13). For example, for query 5 one can reduce the query response time by about 23% when assigning four cores instead of one core to the setting with four workers. However, most of the queries finally benefit when having four cores assigned (queries 3, 4, 5, 6, 7, 8, 10, 11, 12 and 13). Using more workers in the system most of the times increases the query response times.

In terms of memory consumption during querying, Figure 6.11 shows for the LUBM 1 dataset that adding more cores to a system with one worker hardly affects the amount of memory used by the system for our test run. However, there is a visible increase when adding more workers to a system with one core, see Figure 6.12. The LUBM 20 runs show a slight increase of memory usage when adding more cores (Figure 6.13) and an even bigger increase when adding more workers (Figure 6.14).

Finally, we present our evaluations when issuing parallel requests to Rya. As stated in Section 4.2, we use 8 parallel clients each issuing 63 requests which makes in total 504 requests per query. For the LUBM 1 dataset we chose queries 1, 3, 4, 10, 11, 12 and 14 since they have a reasonable response time. We tested the weakest against the strongest configuration in order to emphasise the differences. The results for the LUBM 1 dataset can be seen in Figure 6.15. Contrary to sequentially executing the queries, executing (the same query) in parallel hugely benefits the median response time by adding more cores and workers. The median response times are more than halved and the throughputs are more than doubled in the stronger configuration. For example, the median response time of "Query 4" can be reduced from 16618ms to 5306ms, which is a reduction of 68% and the throughput is increased from 0.48 queries per second to 1.5 queries per second which is an increase of 213%. The median response time for "Query 14" on the other hand increased from 400ms to 414ms which is a slight increase of about 4% but the throughput still increased from 17.87 queries per second to 18.55 queries per second. For the LUBM 20 dataset, we removed queries 3 and 4 since their response time is too long for this test and added query 6 instead. Also here, some of the queries hugely benefit by adding more resources, see Figure 6.16. However, the median response times and throughputs of query 6 and 14 even suffer by adding more resources. For example, the median response time

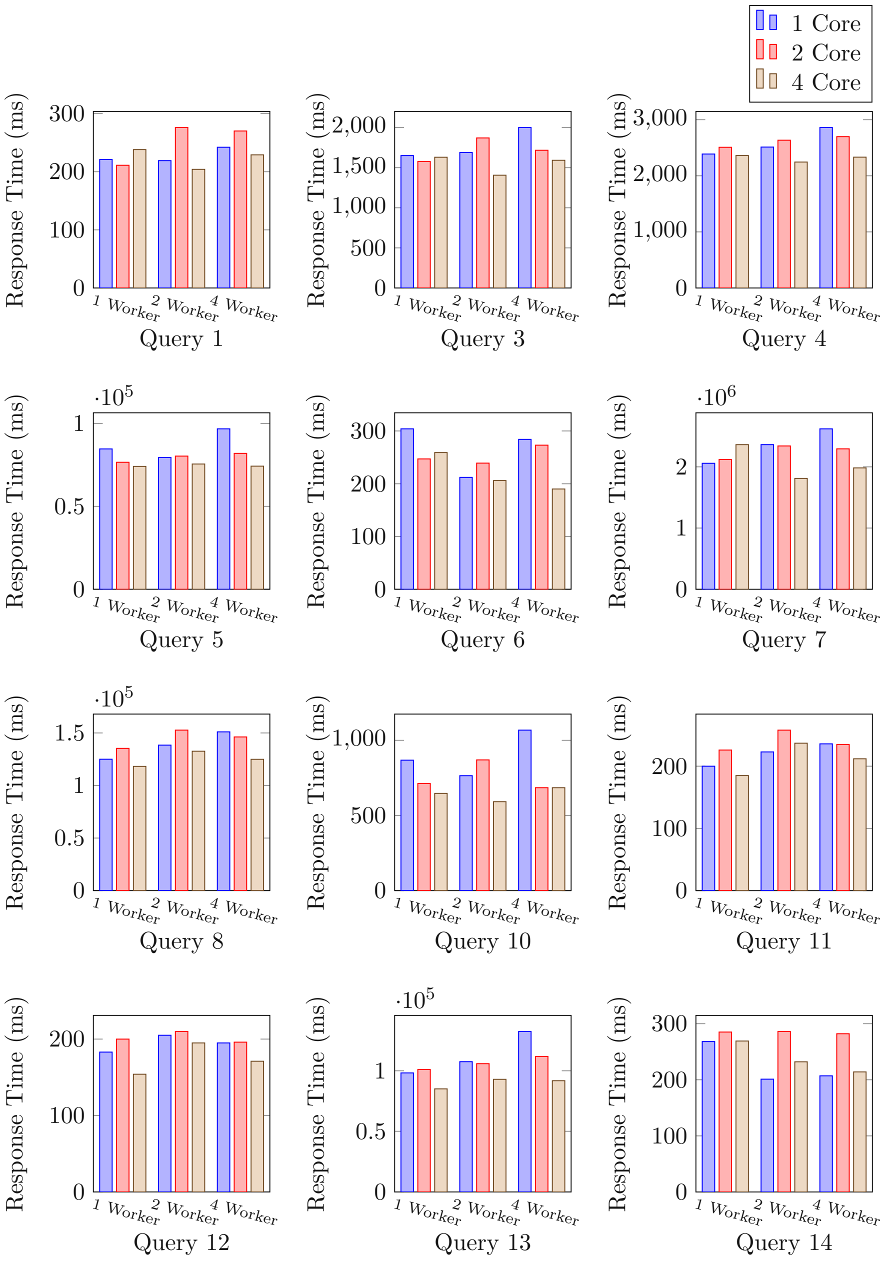

Figure 6.9: Apache Rya MongoDB: LUBM 1 Median Query Response Times

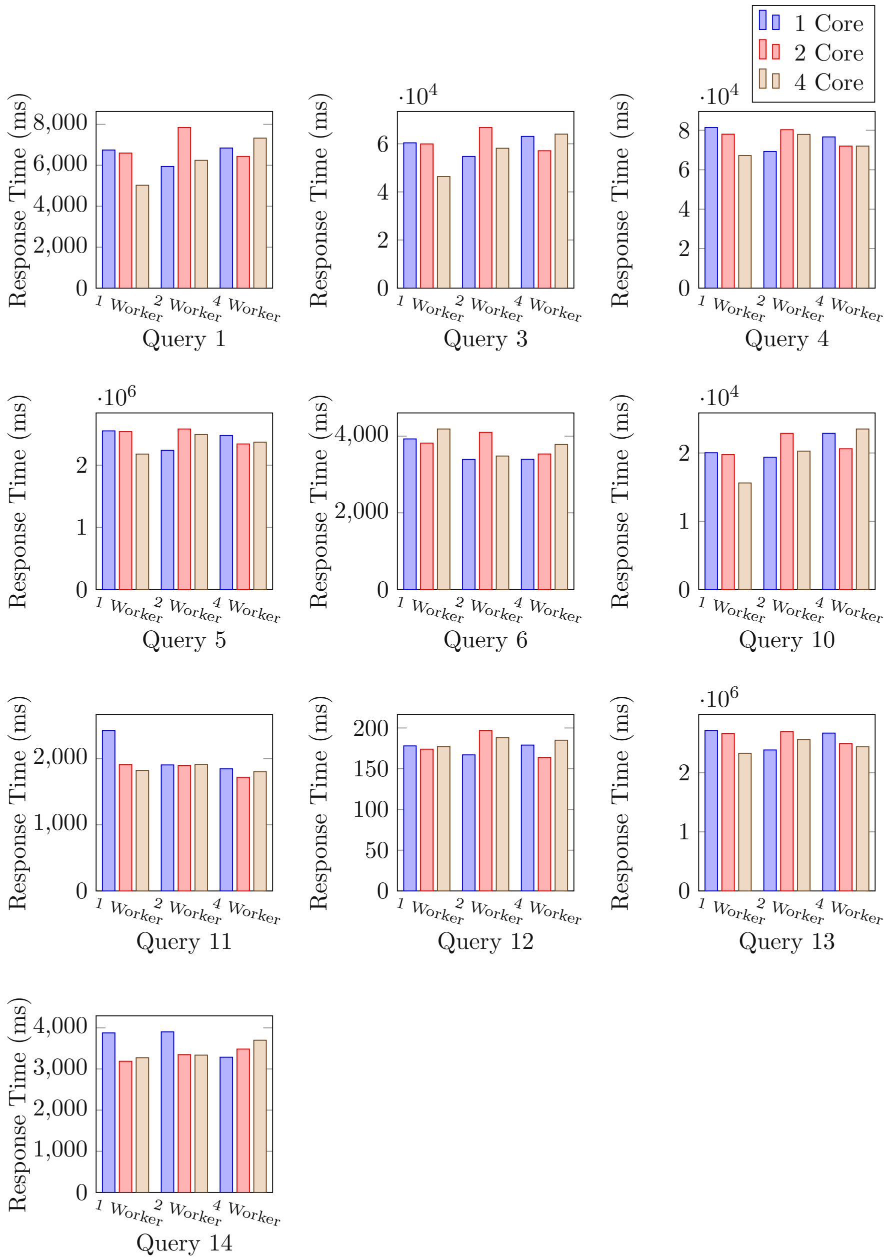

Figure 6.10: Apache Rya MongoDB: LUBM 20 Median Query Response Times

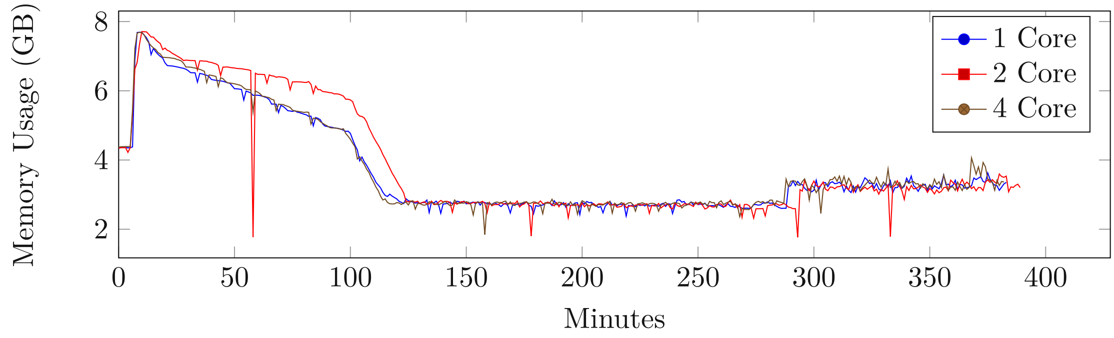

Figure 6.11: Apache Rya MongoDB: LUBM 1 - 1 Worker Memory Usage

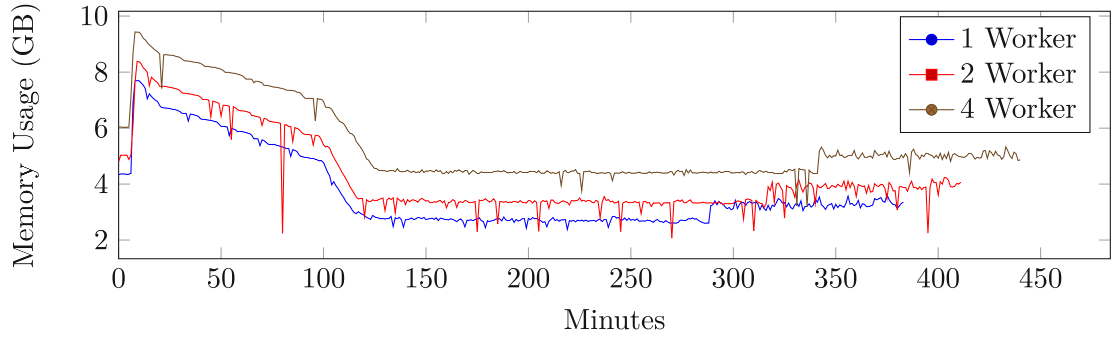

Figure 6.12: Apache Rya MongoDB: LUBM 1 - 1 Core Memory Usage

for "Query 10" decreased from 247113ms to 66809ms which is a decrease of about 73% and the throughput increased from 0.03 queries per second to 0.12 queries per second. However, the response time for "Query 14" increased from 10099ms to 12062ms which is an increase of 19%. The throughput also decreased from 0.78 queries per second to 0.66 queries per second which is a decrease of 15%.

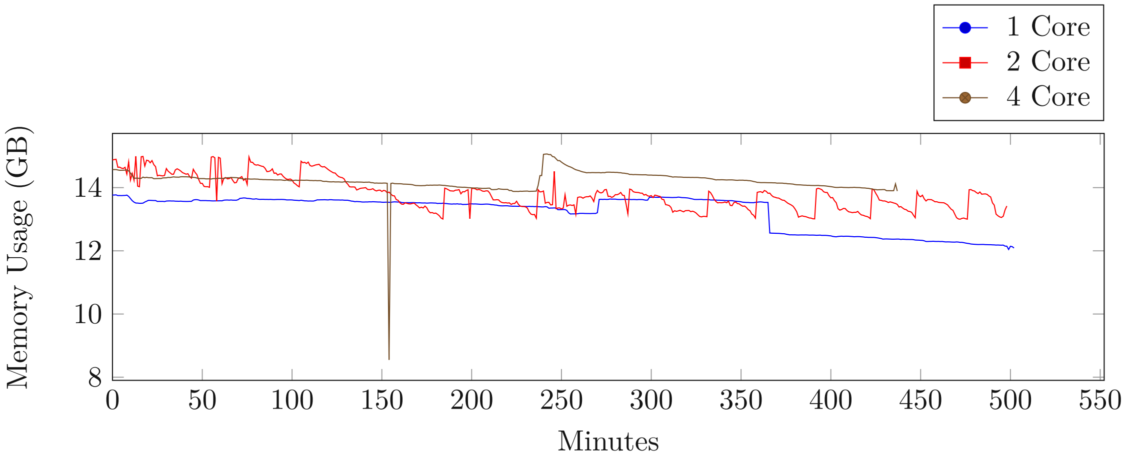

Figure 6.13: Apache Rya MongoDB: LUBM 20 - 1 Worker Memory Usage

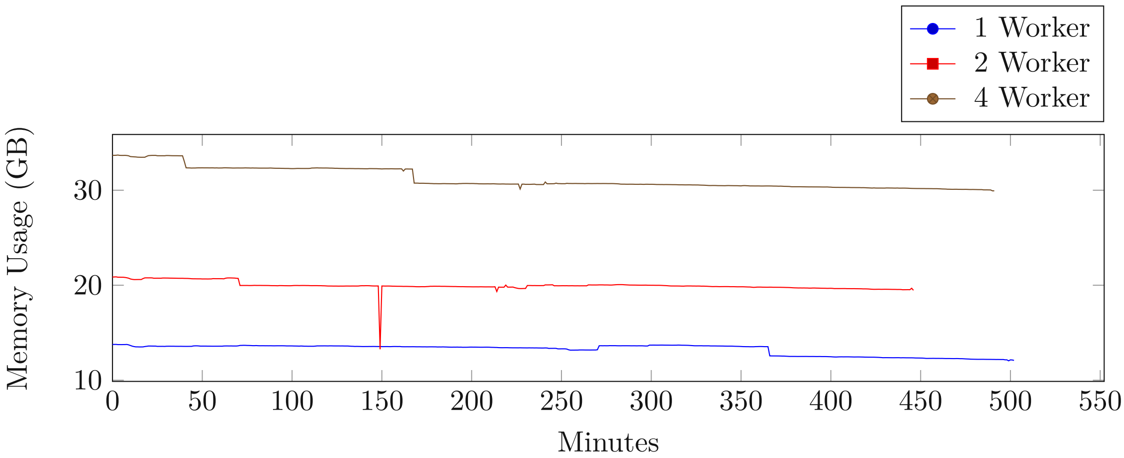

Figure 6.14: Apache Rya MongoDB: LUBM 20 - 1 Core Memory Usage

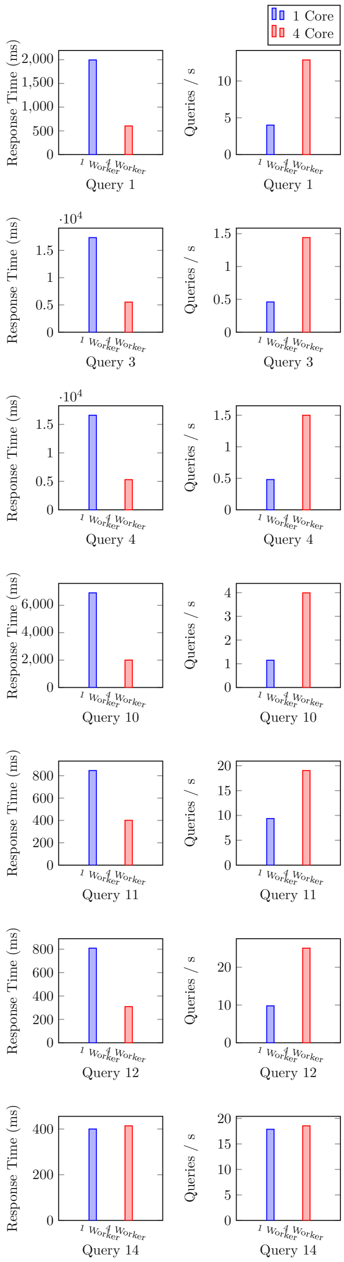

Figure 6.15: Apache Rya MongoDB: LUBM 1 Parallel Queries

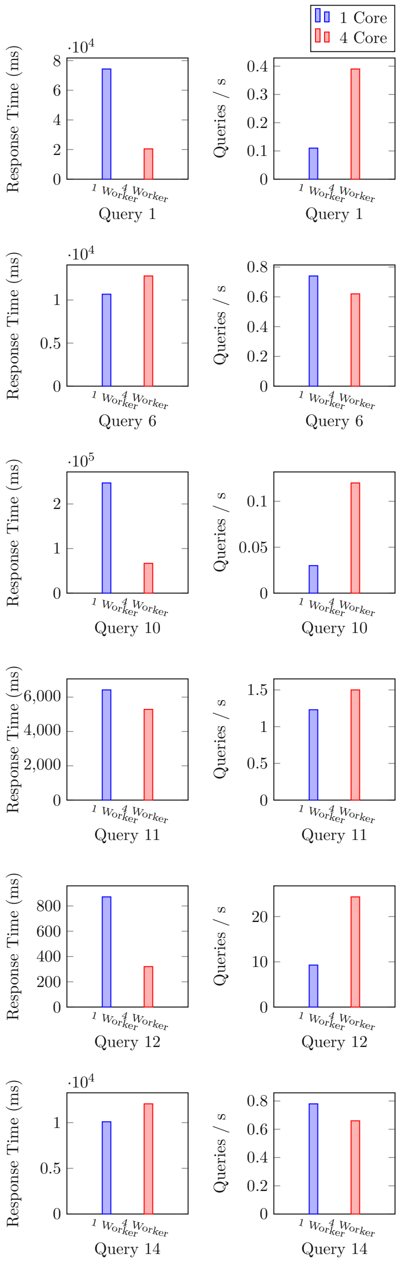

Figure 6.16: Apache Rya MongoDB: LUBM 20 Parallel Queries

6.3 SANSA-Stack

The results of loading the LUBM 1 and LUBM 20 datasets are given in Figure 6.17. The raw results can be seen in the Appendix (Table 8.32). It can be seen that the loading time benefits from both, more workers and more cores. Fixating the number of workers to one for the LUBM 1 dataset results in an about 20% loading time decrease when using two cores instead of one and again about 38% when further increasing the cores from two to four. Also when fixating the number of cores to one, the decrease of loading time is about 14% and about 8% when increasing the number of workers from one to two and from two to four respectively. The overall decrease from the weakest setting to the best is about 50%. The relative decrease in loading time is even bigger when observing the LUBM 20 dataset. Fixating the number of workers to one results in a loading time decrease of about 40% and 41% when increasing the number of cores from one to two and two to four respectively. Fixating the number of cores to one we see a decrease of about 45% and again of about 27% when increasing the number of workers from one to two and from two to four respectively. For the LUBM 20 dataset, the overall reduction of loading time which we could observe for our settings is about 81%.

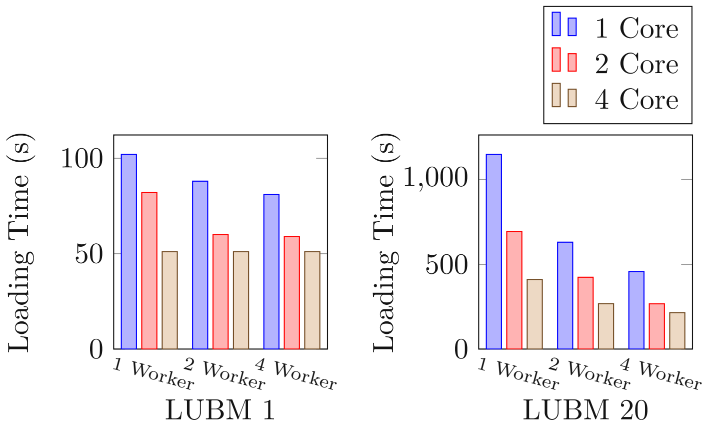

Figure 6.17: SANSA-Stack: Loading Data - Loading Time

The memory usage during loading the data increases greatly with the number of workers and also with the number of cores, see Figure 6.18 (or Table 8.32 in the Appendix). For example, for the LUBM 1 dataset, the memory consumption of the best performing four-worker setting increased about 14% when using two cores instead of one. Increasing the cores to four, the memory consumption further increased about 23%. The increase was even more drastic for the LUBM 20 dataset. There, the increase amounts to about 132% when using two cores instead of one and increases another 26% when using four cores instead of two. In all of our loading experiments, the last setting used the highest amount of memory, namely about 122GB. Overall, the difference in memory consumption between the weakest and the strongest setting amounts to 335% for the LUBM 20 dataset.

Analogous to Apache Rya Accumulo, the consumed storage increases almost linearly

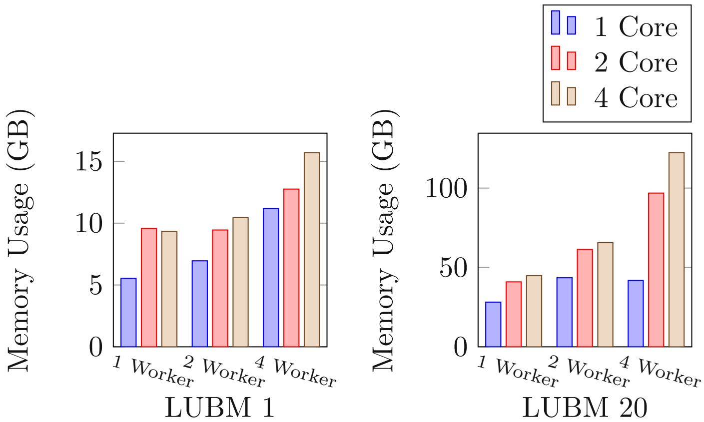

Figure 6.18: SANSA-Stack: Loading Data - Memory Usage

with the number of workers. The network communication underlies high fluctuations but generally also mostly increases with the number of workers.

The results for the median query response time experiments for the LUBM 1 and LUBM 20 datasets are given in Figures 6.19 and 6.20 respectively. The raw results of the experiments are given in the Appendix (Tables 8.33 to 8.46). The query response times of all queries clearly benefit from both, an increased number of cores and workers. For instance, for "Query 4" in the LUBM 1 dataset, the median response time for the system with 1 worker decreased about 34% when using two cores instead of one and again by about 43% when using four cores instead of two. When we fixate the number of cores to one, we see a decrease of about 31% and about 33% when we increase the worker from one to two and four respectively. The overall response time reduction which could be achieved for this query by adding more resources is about 65%. For the LUBM 20 dataset, the overall reduction which could be achieved by adding more resources even was about 86%. Thus, this system is highly scalable both, in terms of the number of workers and the number of cores.

The memory consumption during some LUBM 1 querying benchmark runs can be found in Figures 6.21 and 6.22. The memory consumption increase by adding more cores closely resembles the memory consumption when adding more workers. About the same holds for the memory consumptions for the LUBM 20 benchmarks, which can be found in Figures 6.23 and 6.24.

The measurements of the parallel access to the triplestore for LUBM 1 are given in Figure 6.25. The differences between using one worker with one core each and four workers with four cores each is huge. For example, the median response time for "Query 1" is reduced from 50826ms to 5844ms which amounts to a reduction of 88.5%. The throughput analogously is more than 8 times higher than in the weaker configuration. This response time and throughput behaviour can be seen in all the executed queries. When we perform the same experiment for the LUBM 20 dataset, many Pods go out of

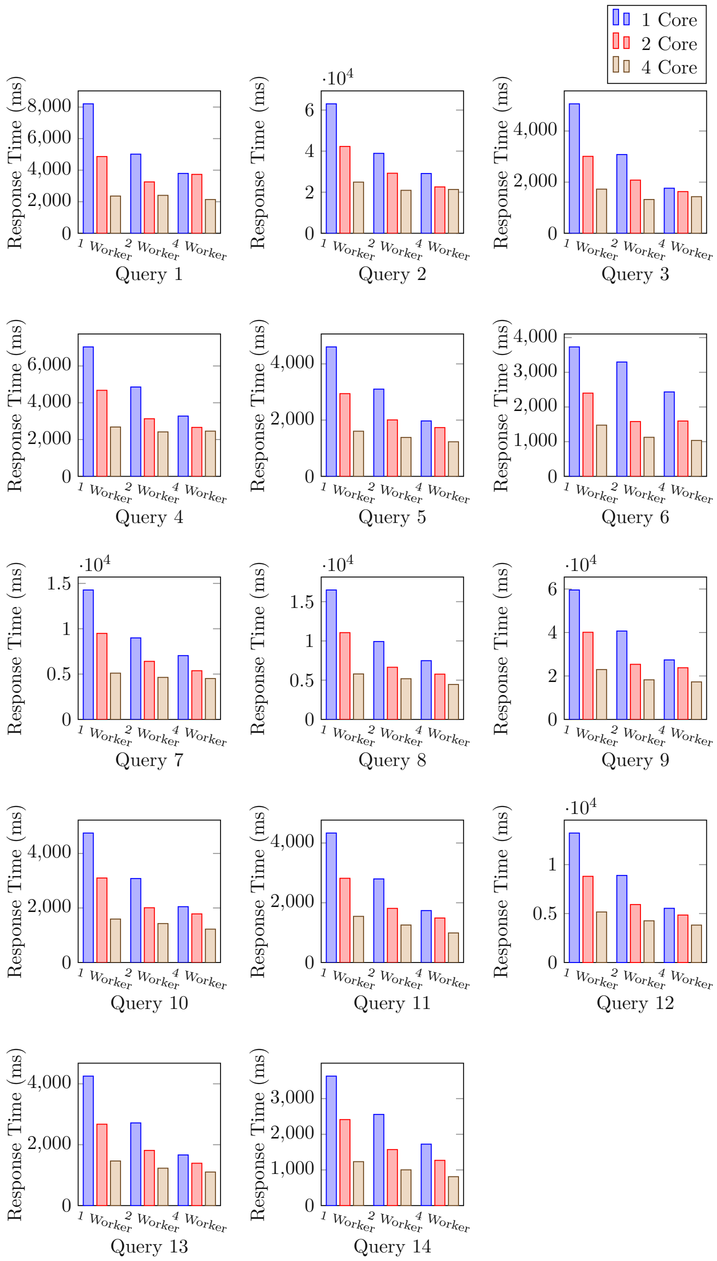

Figure 6.19: SANSA-Stack: LUBM 1 Median Query Response Times

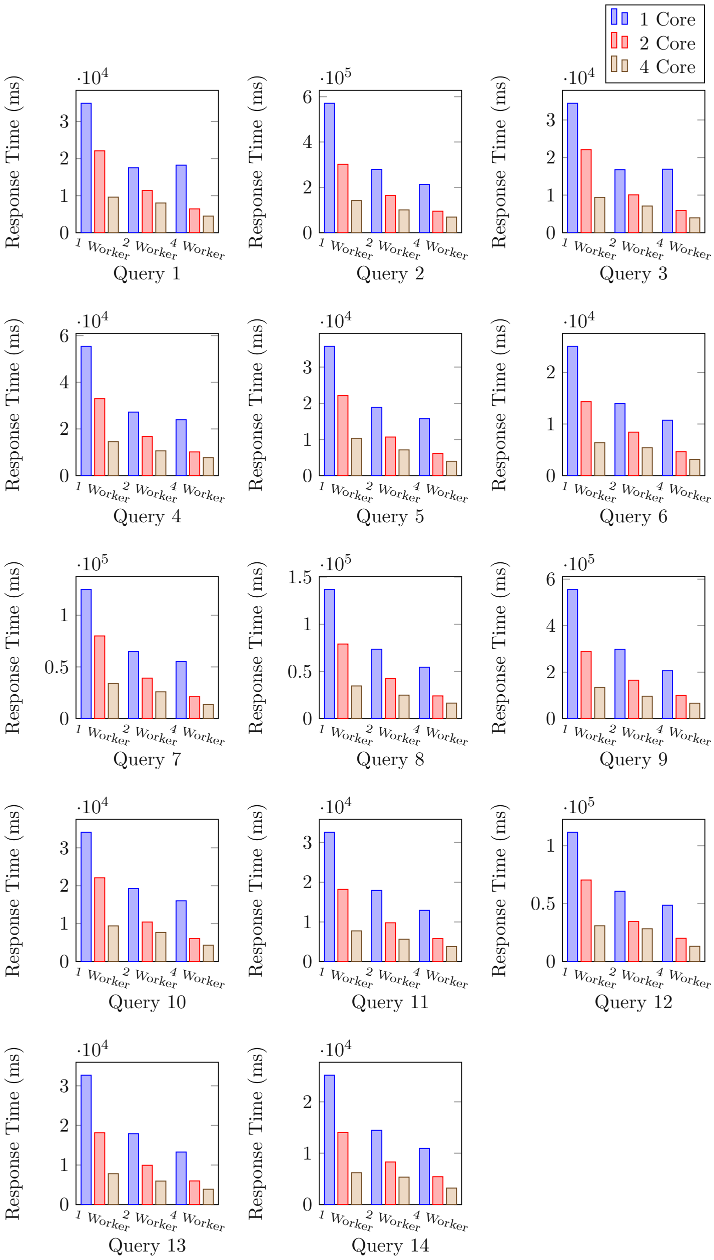

Figure 6.20: SANSA-Stack: LUBM 20 Median Query Response Times

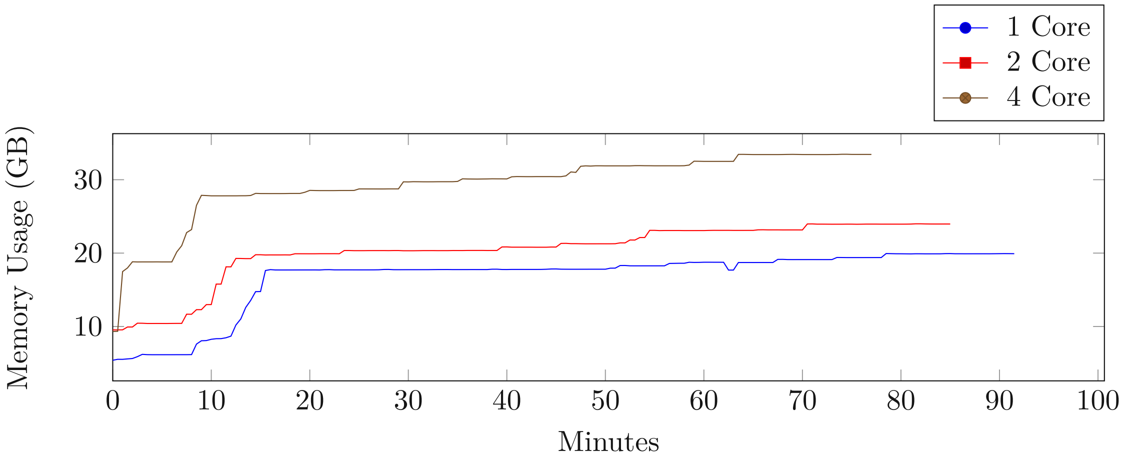

Figure 6.21: SANSA-Stack: LUBM 1 - 1 Worker Memory Usage

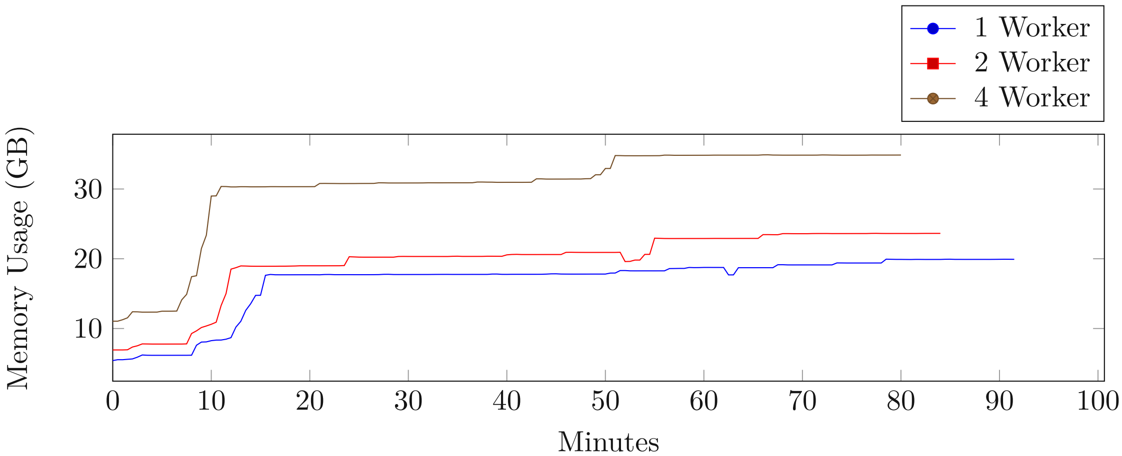

Figure 6.22: SANSA-Stack: LUBM 1 - 1 Core Memory Usage

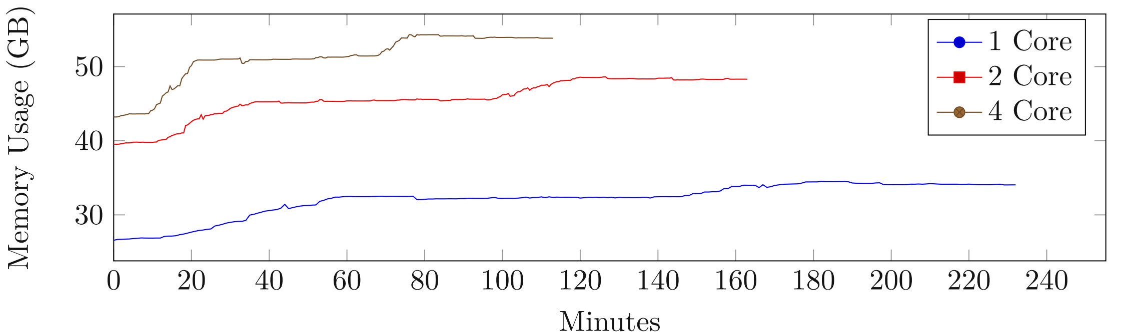

Figure 6.23: SANSA-Stack: LUBM 20 - 1 Worker Memory Usage

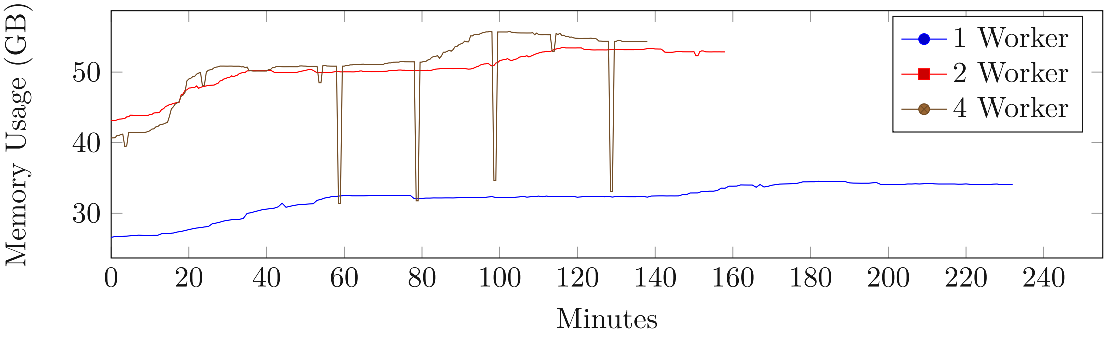

Figure 6.24: SANSA-Stack: LUBM 20 - 1 Core Memory Usage

memory for some queries, making the test for the specific query unreliable. The results for the tests were no worker went out of memory can be seen in Figure 6.26. In general we can see here again, that adding more resources results in much faster response times. However, GKE limits[4] the amount of assignable memory to a Pod regarding its assigned CPU cores. Therefore, the tests for queries 3, 10 and 11 resulted in out-of-memory errors in the case of assigning only one core.

[4]`https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-resource-requests#compute-class-min-max`

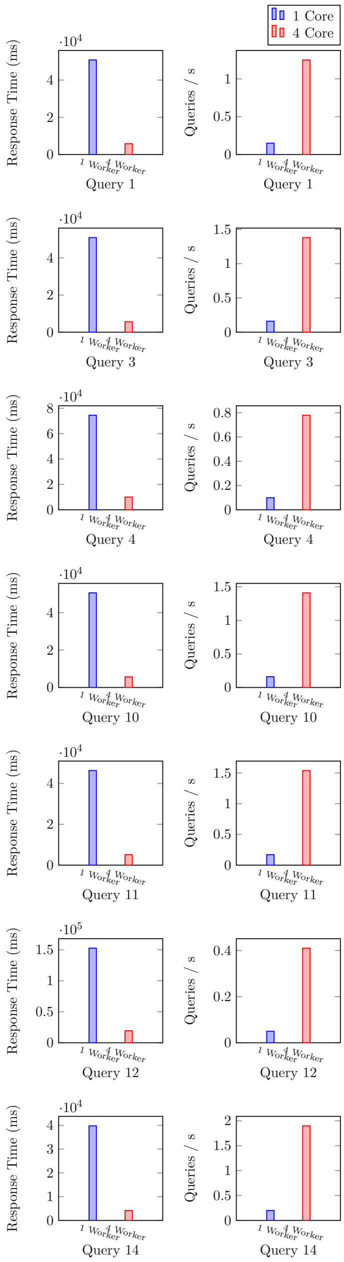

Figure 6.25: SANSA-Stack: LUBM 1 Parallel Queries

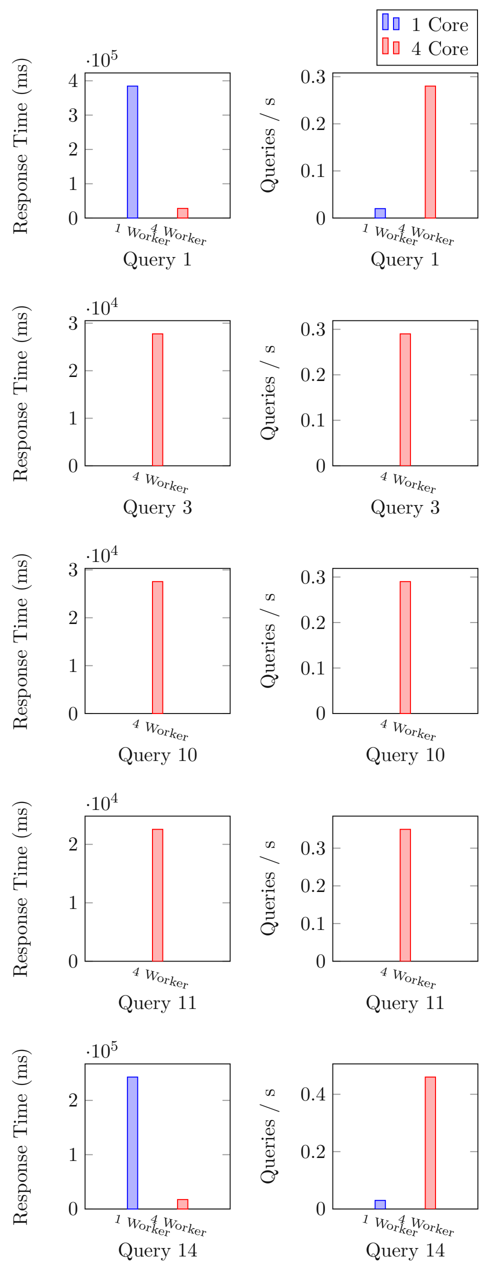

Figure 6.26: SANSA-Stack: LUBM 20 Parallel Queries

6.4 Performance Discussion

Our performance evaluations show, that Apache Rya Accumulo has by far the poorest loading time performance compared to Apache Rya MongoDB and SANSA-Stack. In some configurations, Apache Rya Accumulo requires nearly one hour to load the LUBM 1 dataset, whereas Apache Rya MongoDB and SANSA-Stack both complete loading the LUBM 1 dataset in less than two minutes. For the LUBM 20 dataset, the loading time for Apache Rya Accumulo increases to almost 30 hours, in contrast to one hour and 40 minutes for Apache Rya MongoDB and 19 minutes for SANSA-Stack. Additionally, Apache Rya Accumulo's loading performance declines when scaling along with available resources, including both the number of cores and workers, which generally decrease the loading performance. On the other hand, Apache Rya MongoDB's loading performance generally improves when adding more resources. However, the most significant improvement in loading time when adding resources is observed with SANSA-Stack. Due to the unacceptable loading times for Apache Rya Accumulo for the LUBM 20 dataset, we did not conduct the data-loading test for all configurations and also skipped the experiments regarding concurrent access.

When comparing the response times for the individual LUBM queries, Apache Rya Accumulo performs comparably to Apache Rya MongoDB. For the LUBM 1 dataset there are queries (for instance for Query 10) in which Apache Rya Accumulo answers around four times faster than Apache Rya MongoDB and vice versa (for instance, for Query 8). When comparing them for the LUBM 20 dataset, Apache Rya Accumulo is generally faster than Apache Rya MongoDB. However, a significant drawback is that Apache Rya Accumulo times out after one hour when answering Queries 5 and 13 while Apache Rya MongoDB can answer them. Nonetheless, there are LUBM queries that both systems cannot answer within one hour. When comparing both systems to SANSA-Stack for the LUBM 1 dataset, they are faster answering most queries than SANSA-Stack. Some queries however can be answered significantly faster by SANSA-Stack, for instance Query 7. For larger datasets, SANSA-Stack is considerably faster than both systems. Moreover, SANSA-Stack can answer all LUBM queries in under one hour. However, this could also be due to the different reasoning support of our candidate systems and SANSA-Stack being an in-memory system.

In our tests for query response times regarding the scaling of cores and workers, Apache Rya Accumulo primarily benefits when using the LUBM 20 dataset, whereas the response times for Apache Rya MongoDB do not seem to improve significantly when adding more resources. On the other hand, the response times for SANSA-Stack greatly improve when adding more resources.

Finally, when issuing concurrent requests, the individual response times for Apache Rya MongoDB seem to improve when adding more resources. The same holds true for SANSA-Stack.

However, SANSA-Stack comes with a significant drawback. It requires a substantial amount of memory to function. Figure 6.27 shows the maximum memory usage for all compared systems in the "4 worker with 4 cores each" setting during our LUBM query test runs. For the LUBM 1 dataset, the maximum memory consumption of SANSA-Stack is nearly eight times higher than that of Apache Rya MongoDB. This difference decreases to 3.5 times for the LUBM 20 dataset. Since we did not run the query tests with this setting for Apache Rya Accumulo for the LUBM 20 dataset, this test is omitted in the figure. During our benchmark runs, it occasionally happened that workers crashed because they ran out of memory. This was due to the fact that we had to predefine the amount of memory allocated to the workers, which did not always align with the actual memory requirements. However, Spark always successfully recovered and the issued queries even returned the correct answers also during restarts of workers. The runs in which out-of-memory errors occurred still had to be repeated with increased memory resources in order to ensure more consistent measurements.

The determination of the superior system for our performance evaluation between Apache Rya MongoDB and SANSA-Stack is inconclusive. Apache Rya MongoDB offers decent data loading and querying performance, combined with reasonable memory requirements. On the other hand, SANSA-Stack scales better with the available resources and can answer all queries without any timeouts. It also performs better with larger datasets than Apache Rya MongoDB. However, it consumes significantly more memory than Apache Rya MongoDB. Apache Rya Accumulo fails to meet our performance requirements due to its excessively long data loading durations.

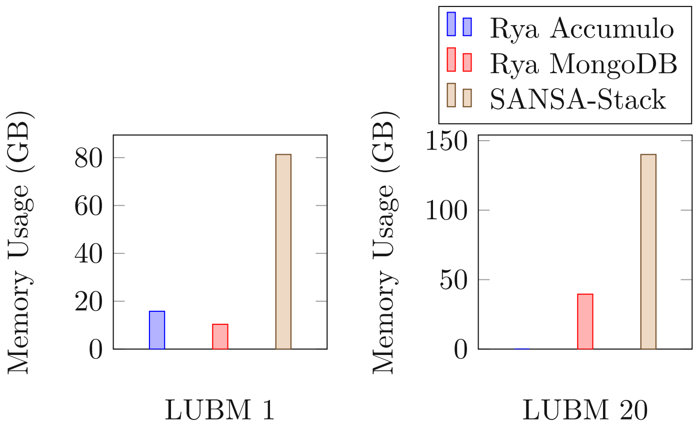

Figure 6.27: Query Memory Usage Comparison - 4 Worker 4 Core

CHAPTER 7

Conclusion and Future Work

This thesis addresses the challenge of selecting an optimal, large-scale, open source triplestore system with reasoning capabilities, specifically for the Kubernetes container orchestration framework. We evaluate our candidate systems with respect to the "fraud detection" use case, which we consider a typical application for large triplestores.

First, we defined the use case fraud detection and the requirements of a system for battling fraud. This includes the used framework, the required behavior of the storage when data changes occur, the necessary reasoning support and the access characteristics. We further described how a standalone system can benefit when migrating to a distributed setting in a cloud environment.

Then, we defined functional and performance evaluation criteria in order to evaluate distributed triplestores for general use cases. The defined functional evaluation criteria include the used framework, the available documentation, the storage characteristics, SPARQL support, the supported reasoning fragment together with the support for custom inference rules, its support for compression, the ease of deployment and elasticity support. We defined general performance criteria on how to evaluate distributed triplestores, including data load time, query time and queries per second and latency when concurrently accessing the triplestore. Also we defined metrics to observe like memory usage, storage size and network communication. Finally we described our test setups for the evaluation of our candidate systems which include different setups for the number of cores, the number of workers and the used datasets.

By conducting a literature research, we identified and evaluated the most recent open-source distributed triplestores. There have already been some attempts on developing open-source systems for distributed triplestores. The only projects which are in active development are Apache Rya and the SANSA-Stack to the best of our knowledge. Other systems like Halyard and CumulusRDF exist, but development seems to have discontinued according to their last commit dates[1,2]. For none of our candidate systems there is an out-of-the box deployment option for Kubernetes available.

We evaluated Apache Rya with both, an Apache Accumulo and a MongoDB backend. Accumulo uses Hadoop as backend framework and storage, while MongoDB is a NoSQL database. We also evaluated the SANSA-Stack framework which uses Apache Spark for its operations. We showed how those systems can be deployed in a Kubernetes environment. Then we conducted a functional evaluation for these triplestores.

We found, that none of these triplestores fulfills every functional requirement for our defined use case. Apache Rya MongoDB fulfills the most requirements. Both Apache Rya Accumulo and Apache Rya MongoDB are built upon the Eclipse RDF4J framework, while SANSA-Stack is implemented using the Jena framework which fits our described use case. The best documentation available comparing these systems is provided for Apache Rya Accumulo, while the other systems lack descriptions of their behavior or instructions on how to deploy them. All of our candidate systems use consistent, partition tolerant storage. A lack of SPARQL support is only observed for SANSA-Stack since it does not allow insert, update and delete requests. None of the observed systems fully supports OWL Lite, nor allows rule-inferencing for custom rules. Storage compression techniques are applied for all of the evaluated systems. Apache Rya MongoDB is the only system that is easy to deploy due to the availability of a MongoDB Kubernetes Operator, while the other systems are particularly difficult to deploy because of the quantity of the involved systems together with their specific version requirements and, especially regarding SANSA-Stack, the lack of documentation regarding deployment. The elasticity feature is only present for Apache Rya MongoDB, however, for the other systems, this feature is planned.

We evaluated our candidate systems regarding the aforementioned performance criteria. For Apache Rya we observed that the MongoDB backend for Apache Rya performs similar to the Accumulo backend regarding query response time. However, in our test settings, Accumulo particularly fails when loading the system in an acceptable duration, especially when the number of workers increases. We observed data loading times for the LUBM 20 dataset of around 30 hours with four workers, which is why we did not evaluate Apache Rya Accumulo for the LUBM 20 dataset with the two-core and four-core setting. Apache Rya does not scale well when adding more resources to the system except for parallel access. The evaluation of SANSA-Stack shows, that the performance of SANSA-Stack is worse than that of Apache Rya with the MongoDB backend for most queries when assigning only one core and one worker. However, SANSA-Stack scales well in terms of performance when adding more resources. Thus it outperforms Apache Rya especially for larger datasets having more cores and workers assigned. A huge drawback for SANSA-Stack is that it needs much more memory than the other systems.

Our triplestore of choice regarding our use case is therefore Apache Rya MongoDB since it

[1] `https://github.com/Merck/Halyard`
[2] `https://github.com/cumulusrdf/cumulusrdf`

fulfills the most functional evaluation criteria and has decent performance with reasonable memory requirements.

In order to improve those systems, we suggest the support for custom inference rules. Furthermore, when using SANSA-Stack, the support for dynamically altering the stored triples with SPARQL is a required feature. Finally, a Kubernetes Operator should be created in order to provide a convenient deployment option on Kubernetes and in order to allow Kubernetes to efficiently manage the systems.

The limitation of this work is, that we did not test for larger datasets than LUBM 20 and larger configurations than four cores and four workers. Some of these evaluations were already conducted for Apache Rya Accumulo ([PCR12, PCR15]), but no performance evaluations were found for Apache Rya MongoDB and SANSA-Stack. Therefore it is unclear how more hardware resources would affect loading and query times. However, our evaluations especially show the scaling characteristics of the candidate systems, which are particularly relevant for optimizing costs in a cloud environment.

Future work could evaluate these systems in also larger settings in order to observe if their performance characteristics change having more cores or workers assigned. Furthermore regarding Apache Rya MongoDB, since sharding is not supported by the MongoDB Community Kubernetes Operator, we suggest conducting experiments with the MongoDB Enterprise Kubernetes Operator, where sharding is supported. Finally, we suggest that our evaluated systems should implement OWL Lite support and custom rule inferencing.

CHAPTER 8

Appendix

LUBM	#Nodes	CPU	Loading Time	Memory usage (peak)	Storage size	Network Communication (peak)
1	1	1	1440s	7974952960	105463808	307000,727929304
		2	1440s	7946125312	103780352	345342,115073072
		4	1441s	8000950272	105443328	294565,814875181
	2	1	2430s	9723092992	211165184	285068,391538227
		2	1441s	9319403520	210870272	374823,528877659
		4	2456s	9902333952	211197952	282558,913088357
	4	1	3534s	12416897024	354668544	193447,479575813
		2	2162s	12210991104	354430976	412750,248592362
		4	3519s	12887072768	354463744	345250,500427719
20	1	1	40062s	13273423872	1862766592	349966,49010586
		2	Not tested			
		4	Not tested			
	2	1	59749s	14108393472	1850679296	311832,650542109
		2	Not tested			
		4	Not tested			
	4	1	108265s	18211215770	536588288	279910,482742213
		2	Not tested			
		4	Not tested			

Table 8.1: Apache Rya Accumulo: Loading Data

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
		1	340ms/262ms	110652,189888332
	1	2	218ms/118ms	119471,637110507
		4	266ms/190ms	113735,540994465
		1	334ms/204ms	142469,620592019
1	2	2	261ms/124ms	128196,1304059
		4	364ms/201ms	122213,970291005
		1	362ms/236ms	114938,621235416
	4	2	243ms/116ms	129656,574557054
		4	307ms/171ms	134778,204189425
		1	3166ms/2868ms	
	1	2	Not tested	
		4	Not tested	
		1	2087ms/2016ms	
20	2	2	Not tested	
		4	Not tested	
		1	3695ms/3168ms	
	4	2	Not tested	
		4	Not tested	

Table 8.2: Apache Rya Accumulo: LUBM Query 1

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
		1	Timeout	43093170,9269924
	1	2	Timeout	42242534,9719198
		4	Timeout	48397092,6468884
		1	Timeout	53463196,2128541
1	2	2	Timeout	58103709,1887223
		4	Timeout	56800203,8377365
		1	Timeout	49412779,3605122
	4	2	Timeout	71759927,8797756
		4	Timeout	63613084,1963508
		1	Timeout	
	1	2	Not tested	
		4	Not tested	
		1	Timeout	
20	2	2	Not tested	
		4	Not tested	
		1	Timeout	
	4	2	Not tested	
		4	Not tested	

Table 8.3: Apache Rya Accumulo: LUBM Query 2

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
		1	523ms/435ms	762758,152589176
	1	2	447ms/428ms	859822,334431946
		4	507ms/460ms	804939,466408188
		1	764ms/806ms	692375,764832282
1	2	2	476ms/428ms	806531,066607459
		4	565ms/533ms	860789,631972605
		1	542ms/515ms	821453,751813999
	4	2	459ms/440ms	955739,372512881
		4	654ms/557ms	1021493,2060913
		1	17967ms/17817ms	
	1	2	Not tested	
		4	Not tested	
		1	15987ms/16113ms	
20	2	2	Not tested	
		4	Not tested	
		1	15025ms/15074ms	
	4	2	Not tested	
		4	Not tested	

Table 8.4: Apache Rya Accumulo: LUBM Query 3

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
		1	16993ms/16894ms	27243471,6435691
	1	2	18627ms/18726ms	27069700,7935753
		4	17514ms/17612ms	22864391,7439025
		1	18288ms/18480ms	31580598,5849171
1	2	2	16190ms/16232ms	23355124,4544782
		4	19883ms/19803ms	22963180,0631264
		1	18387ms/18632ms	26211043,5804908
	4	2	17966ms/18050ms	25774928,0558132
		4	17092ms/17098ms	26566382,3624026
		1	2261088ms/2259958ms	
	1	2	Not tested	
		4	Not tested	
		1	2121573ms/2136241ms	
20	2	2	Not tested	
		4	Not tested	
		1	1096298ms/1123997ms	
	4	2	Not tested	
		4	Not tested	

Table 8.5: Apache Rya Accumulo: LUBM Query 4

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	293920ms/298089ms	45016039,0084263
		2	289427ms/288694ms	40091420,0375215
		4	323173ms/322521ms	36095569,6065421
	2	1	325027ms/320622ms	43134226,0811849
		2	299648ms/299681ms	49144535,3712632
		4	337203ms/339311ms	36851183,9587443
	4	1	338645ms/335473ms	35407290,2562849
		2	277021ms/287251ms	49656139,1697948
		4	285292ms/287499ms	50674455,758243
20	1	1	Timeout	
		2	Not tested	
		4	Not tested	
	2	1	Timeout	
		2	Not tested	
		4	Not tested	
	4	1	Timeout	
		2	Not tested	
		4	Not tested	

Table 8.6: Apache Rya Accumulo: LUBM Query 5

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	353ms/297ms	148381,042051223
		2	279ms/186ms	169086,534260674
		4	289ms/213ms	118070,851902265
	2	1	352ms/219ms	166104,02174493
		2	300ms/212ms	161168,560126121
		4	320ms/214ms	141937,941337047
	4	1	334ms/206ms	150525,481986534
		2	307ms/222ms	178591,190865433
		4	305ms/224ms	149265,952738316
20	1	1	3440ms/3045ms	
		2	Not tested	
		4	Not tested	
	2	1	2012ms/1593ms	
		2	Not tested	
		4	Not tested	
	4	1	2074ms/1694ms	
		2	Not tested	
		4	Not tested	

Table 8.7: Apache Rya Accumulo: LUBM Query 6

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	486808ms/487118ms	43307179,5824119
		2	472915ms/471012ms	44451481,6544599
		4	523121ms/522249ms	39489792,7810117
	2	1	568705ms/566572ms	37939066,8073298
		2	498355ms/498892ms	41369279,570527
		4	564852ms/567005ms	39938438,5665566
	4	1	580319ms/576033ms	36039346,7726836
		2	480880ms/479546ms	43838648,9439807
		4	549454ms/547491ms	40089879,6786588
20	1	1	Timeout	
		2	Not tested	
		4	Not tested	
	2	1	Timeout	
		2	Not tested	
		4	Not tested	
	4	1	Timeout	
		2	Not tested	
		4	Not tested	

Table 8.8: Apache Rya Accumulo: LUBM Query 7

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	768542ms/772319ms	40391653,0877632
		2	745606ms/742050ms	47689722,0205612
		4	776481ms/776890ms	46426374,4643879
	2	1	750151ms/760091ms	53450662,9381631
		2	703522ms/701203ms	54067077,9126318
		4	771960ms/772471ms	49554355,2427148
	4	1	773778ms/772707ms	44597789,7338796
		2	711792ms/710388ms	50708636,1592896
		4	722785ms/717887ms	53590807,0070549
20	1	1	Timeout	
		2	Not tested	
		4	Not tested	
	2	1	Timeout	
		2	Not tested	
		4	Not tested	
	4	1	Timeout	
		2	Not tested	
		4	Not tested	

Table 8.9: Apache Rya Accumulo: LUBM Query 8

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	Timeout	25476044,3018071
		2	Timeout	26789491,5563976
		4	Timeout	23509629,0095953
	2	1	Timeout	22811609,3556631
		2	Timeout	25645569,1241769
		4	Timeout	22694756,2067612
	4	1	Timeout	22720485,3760204
		2	Timeout	27091225,1760622
		4	Timeout	22846454,7892069
20	1	1	Timeout	
		2	Not tested	
		4	Not tested	
	2	1	Timeout	
		2	Not tested	
		4	Not tested	
	4	1	Timeout	
		2	Not tested	
		4	Not tested	

Table 8.10: Apache Rya Accumulo: LUBM Query 9

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	250ms/196ms	265101,376153812
		2	237ms/171ms	303370,624416173
		4	242ms/200ms	232340,589545299
	2	1	252ms/197ms	330927,078570588
		2	231ms/183ms	279313,455261671
		4	272ms/222ms	304435,983241573
	4	1	246ms/221ms	205232,155006103
		2	213ms/162ms	333479,977681532
		4	231ms/184ms	323385,767702771
20	1	1	16222ms/16084ms	
		2	Not tested	
		4	Not tested	
	2	1	13548ms/13316ms	
		2	Not tested	
		4	Not tested	
	4	1	12285ms/11918ms	
		2	Not tested	
		4	Not tested	

Table 8.11: Apache Rya Accumulo: LUBM Query 10

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	339ms/255ms	54448,3300581072
		2	249ms/221ms	65636,7475932081
		4	349ms/291ms	60627,4938358032
	2	1	317ms/271ms	68447,6398508332
		2	298ms/253ms	60776,2094503487
		4	338ms/294ms	66836,9995401283
	4	1	324ms/272ms	77488,3179594939
		2	262ms/204ms	92683,8217479473
		4	367ms/295ms	66500,8939581605
20	1	1	22093ms/23006ms	
		2	Not tested	
		4	Not tested	
	2	1	16091ms/14969ms	
		2	Not tested	
		4	Not tested	
	4	1	11011ms/11491ms	
		2	Not tested	
		4	Not tested	

Table 8.12: Apache Rya Accumulo: LUBM Query 11

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	244ms/193ms	55845,1117899279
		2	230ms/191ms	61245,7338215324
		4	321ms/263ms	59255,4649750063
	2	1	301ms/256ms	81645,0394213954
		2	269ms/221ms	56618,7490933469
		4	309ms/264ms	70786,8870926279
	4	1	304ms/259ms	32664,0957464501
		2	231ms/186ms	75894,6581628564
		4	293ms/260ms	66224,5033234814
20	1	1	3076ms/3372ms	
		2	Not tested	
		4	Not tested	
	2	1	2620ms/2609ms	
		2	Not tested	
		4	Not tested	
	4	1	938ms/882ms	
		2	Not tested	
		4	Not tested	

Table 8.13: Apache Rya Accumulo: LUBM Query 12

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	413211ms/411009ms	32560652,3749952
		2	384661ms/382491ms	37172668,2412753
		4	428729ms/438621ms	37290463,7981141
	2	1	413270ms/417699ms	42112368,6626268
		2	383921ms/382296ms	44658390,9736003
		4	451453ms/455457ms	38553312,3364641
	4	1	422309ms/426473ms	29588597,520496
		2	365419ms/378644ms	50570086,1202489
		4	394412ms/404597ms	39023901,6368894
20	1	1	Timeout	
		2	Not tested	
		4	Not tested	
	2	1	Timeout	
		2	Not tested	
		4	Not tested	
	4	1	Timeout	
		2	Not tested	
		4	Not tested	

Table 8.14: Apache Rya Accumulo: LUBM Query 13

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	339ms/213ms	145161,526132728
		2	339ms/212ms	165009,704754316
		4	353ms/219ms	111464,603685582
	2	1	338ms/284ms	118622,715930658
		2	317ms/212ms	117702,246013447
		4	288ms/216ms	143049,742084132
	4	1	271ms/184ms	157493,505392268
		2	318ms/209ms	170221,873609354
		4	302ms/213ms	147539,196107679
20	1	1	5062ms/5241ms	
		2	Not tested	
		4	Not tested	
	2	1	3832ms/3894ms	
		2	Not tested	
		4	Not tested	
	4	1	1821ms/1458ms	
		2	Not tested	
		4	Not tested	

Table 8.15: Apache Rya Accumulo: LUBM Query 14

Query	1 DataNode	2 DataNodes
1	309ms/210ms	334ms/204ms
3	492ms/458ms	764ms/806ms
4	17373ms/17412ms	18288ms/18480ms
5	286264ms/286083ms	325027ms/320622ms
6	278ms/202ms	352ms/219ms
7	463658ms/463986ms	568705ms/566572ms
8	648650ms/646602ms	750151ms/760091ms
10	242ms/182ms	252ms/197ms
11	304ms/243ms	317ms/271ms
12	250ms/211ms	301ms/256ms
13	374179ms/369592ms	413270ms/417699ms
14	305ms/203ms	338ms/284ms

Table 8.16: Apache Rya Accumulo HDFS DataNodes Comparison, 2 Worker Nodes, 1 Core

LUBM	#Nodes	CPU	Loading Time	Memory usage (peak)	Storage size	Network Communication (peak)
1	1	1	68s	4340326400	439369728	1302820,6931661
		2	69s	4334477312	453038080	1953068,21854447
		4	65s	4374806528	458264576	1716130,61778304
	2	1	77s	5032648704	907685888	2427356,57638691
		2	84s	4872949760	918110208	2507574,57576804
		4	69s	4961943552	920891392	2261177,04580166
	4	1	100s	6037192704	1817161728	3140445,28102813
		2	83s	6571970560	1807040512	4429971,82317289
		4	70s	6968836096	1813286912	4379663,99717526
20	1	1	2967s	14588370944	3429859328	2336163,7059455
		2	2662s	15303348224	3444846592	2124875,07069863
		4	2458s	15547527168	3540492288	2279904,12580098
	2	1	3700s	21741756416	6505222144	2588678,88911067
		2	3426s	22667374592	6617493504	2755691,56773982
		4	2988s	23247155200	6679003136	2785289,18904595
	4	1	6033s	34521821184	12139347968	2991615,52290344
		2	3136s	38847557632	13500780544	4860525,73328698
		4	2873s	39539056640	13957869568	4801100,34320181

Table 8.17: Apache Rya MongoDB: Loading Data

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	365ms/221ms	136955,761982904
		2	342ms/211ms	149049,781905566
		4	387ms/238ms	84875,2437947993
	2	1	374ms/219ms	121410,84044215
		2	393ms/276ms	104962,852234319
		4	324ms/204ms	144175,275496564
	4	1	350ms/242ms	104764,700253609
		2	377ms/270ms	144595,676297685
		4	349ms/229ms	140616,857534029
20	1	1	8376ms/6747ms	3343348,40676462
		2	8194ms/6599ms	2127651,36821752
		4	6311ms/5023ms	2539834,21083876
	2	1	7452ms/5940ms	2711758,25326666
		2	10264ms/7844ms	3524329,80948824
		4	7554ms/6241ms	2410599,31628869
	4	1	8741ms/6844ms	2941825,23905881
		2	8100ms/6431ms	2888968,04678308
		4	8801ms/7328ms	3021095,5724748

Table 8.18: Apache Rya MongoDB: LUBM Query 1

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	Timeout	2519794,20701836
		2	Timeout	2463513,63702613
		4	Timeout	2584611,71190445
	2	1	Timeout	2359869,42070276
		2	Timeout	2013771,53525652
		4	Timeout	2399328,17766255
	4	1	Timeout	2434305,24311539
		2	Timeout	2185566,46786417
		4	Timeout	2510418,95259194
20	1	1	Not tested	
		2	Not tested	
		4	Not tested	
	2	1	Not tested	
		2	Not tested	
		4	Not tested	
	4	1	Not tested	
		2	Not tested	
		4	Not tested	

Table 8.19: Apache Rya MongoDB: LUBM Query 2

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	1675ms/1650ms	3269086,24430871
		2	1649ms/1576ms	3315838,76905641
		4	1689ms/1629ms	3231407,22016674
	2	1	1691ms/1689ms	3231002,72904836
		2	1963ms/1870ms	3086799,66014197
		4	1489ms/1405ms	3366644,69641175
	4	1	2038ms/2000ms	2707153,12194204
		2	1742ms/1716ms	2723350,58458461
		4	1645ms/1591ms	3009432,12484649
20	1	1	60447ms/60400ms	9091254,22119593
		2	60312ms/59950ms	8039782,78066853
		4	45339ms/46359ms	10135664,6077795
	2	1	55055ms/54717ms	8771044,02150533
		2	66182ms/66750ms	7810555,49598149
		4	58874ms/58127ms	8848524,85307248
	4	1	63746ms/63053ms	8128512,87086741
		2	57771ms/57111ms	9304243,57807995
		4	64220ms/64017ms	7947287,02472985

Table 8.20: Apache Rya MongoDB: LUBM Query 3

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	2438ms/2387ms	2637236,66562088
		2	2544ms/2507ms	2374725,99102024
		4	2420ms/2360ms	2525343,7651423
	2	1	2557ms/2511ms	2490742,75403019
		2	2659ms/2633ms	2421374,43151614
		4	2290ms/2244ms	2700690,56121627
	4	1	2888ms/2860ms	2330330,72341312
		2	2705ms/2697ms	2252467,46110231
		4	2346ms/2331ms	2437486,35378082
20	1	1	81379ms/81400ms	2451374,07007163
		2	78085ms/78041ms	2565548,44271189
		4	68324ms/67184ms	3008171,80940515
	2	1	68922ms/69249ms	2838180,30073598
		2	80474ms/80316ms	2549412,35103479
		4	78158ms/77908ms	2588470,58188942
	4	1	76887ms/76661ms	2812760,47056939
		2	71814ms/72009ms	2853143,99368111
		4	72202ms/71994ms	2791315,03930661

Table 8.21: Apache Rya MongoDB: LUBM Query 4

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	81928ms/84696ms	4677384,92179356
		2	76723ms/76621ms	4914949,17737786
		4	74082ms/74137ms	5383136,89057516
	2	1	79053ms/79501ms	4743283,76440674
		2	80489ms/80354ms	4863216,82016697
		4	75667ms/75581ms	4972100,10240849
	4	1	94683ms/96818ms	4095028,82615256
		2	82102ms/81996ms	4694816,10601333
		4	74626ms/74282ms	5001782,58689911
20	1	1	2553195ms/2550530ms	2546249,04960453
		2	2541732ms/2537961ms	2387990,53405188
		4	2192389ms/2178086ms	2819858,15752795
	2	1	2237474ms/2239400ms	2704885,85238862
		2	2584368ms/2581007ms	2385606,95052436
		4	2491054ms/2491504ms	2622443,38352121
	4	1	2472008ms/2476355ms	2448702,85691262
		2	2346615ms/2340410ms	2613895,11118982
		4	2377259ms/2370082ms	2748703,38760926

Table 8.22: Apache Rya MongoDB: LUBM Query 5

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	418ms/304ms	2342100,63688442
		2	342ms/247ms	3023336,63962908
		4	344ms/259ms	2752271,51650953
	2	1	316ms/212ms	2545735,27831662
		2	309ms/239ms	2587364,66592729
		4	282ms/206ms	2376006,32087106
	4	1	364ms/284ms	2319604,55383563
		2	355ms/273ms	2235554,43223721
		4	276ms/190ms	2331219,7325818
20	1	1	4220ms/3927ms	8541141,65764212
		2	3792ms/3819ms	7572822,18656589
		4	4044ms/4186ms	5880707,74792671
	2	1	3841ms/3393ms	7445704,15050764
		2	4355ms/4101ms	6654808,28426898
		4	4128ms/3481ms	8714242,15954032
	4	1	3846ms/3398ms	7879979,83448959
		2	3923ms/3534ms	7709444,2572839
		4	4160ms/3781ms	6997136,24598095

Table 8.23: Apache Rya MongoDB: LUBM Query 6

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	2037909ms/2057472ms	14697314,2893539
		2	2099895ms/2119559ms	14733179,8919973
		4	2046898ms/2051196ms	14819682,1580444
	2	1	2358421ms/2362478ms	14749970,2142217
		2	2333756ms/2340971ms	13677128,946793
		4	1812895ms/1810744ms	17316147,6343472
	4	1	2641297ms/2620506ms	12111158,7143456
		2	2327775ms/2293966ms	13065179,924067
		4	1989323ms/1982799ms	14834122,8116925
20	1	1	Not tested	
		2	Not tested	
		4	Not tested	
	2	1	Not tested	
		2	Not tested	
		4	Not tested	
	4	1	Not tested	
		2	Not tested	
		4	Timeout	

Table 8.24: Apache Rya MongoDB: LUBM Query 7

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	125388ms/124984ms	5576824,32699673
		2	136956ms/135329ms	4404504,8589746
		4	118262ms/118184ms	4918015,27574762
	2	1	138720ms/138396ms	4388154,14452728
		2	153153ms/152681ms	3982054,78787111
		4	130353ms/132596ms	4638144,67350759
	4	1	151042ms/151048ms	3974182,13713269
		2	146637ms/146206ms	4004685,90277627
		4	124719ms/124907ms	4660250,6514517
20	1	1	Not tested	
		2	Not tested	
		4	Not tested	
	2	1	Not tested	
		2	Not tested	
		4	Not tested	
	4	1	Not tested	
		2	Error occurred	
		4	685112ms/697311ms	

Table 8.25: Apache Rya MongoDB: LUBM Query 8

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	Timeout	50639255,4415597
		2	Timeout	47877089,5035284
		4	Timeout	50480307,9227594
	2	1	Timeout	47309901,203827
		2	Timeout	49325510,8390949
		4	Timeout	50192809,4207847
	4	1	Timeout	49654413,1401307
		2	Timeout	43648160,3431275
		4	Timeout	44333507,8084431
20	1	1	Not tested	
		2	Not tested	
		4	Not tested	
	2	1	Not tested	
		2	Not tested	
		4	Not tested	
	4	1	Not tested	
		2	Not tested	
		4	Not tested	

Table 8.26: Apache Rya MongoDB: LUBM Query 9

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	861ms/868ms	47954433,2954167
		2	791ms/713ms	42958048,1319759
		4	696ms/647ms	43513781,0808759
	2	1	795ms/765ms	43742370,8920082
		2	893ms/870ms	40887407,1553253
		4	632ms/592ms	44413344,5619527
	4	1	1080ms/1068ms	43193241,3262154
		2	812ms/763ms	41745158,6779297
		4	706ms/685ms	40775130,2214301
20	1	1	19910ms/20033ms	7443516,47172824
		2	20141ms/19759ms	6246143,8340605
		4	15541ms/15611ms	7252663,82980093
	2	1	19483ms/19389ms	6031704,68393311
		2	23155ms/22883ms	6952167,54128568
		4	20717ms/20278ms	7055984,59104539
	4	1	22267ms/22889ms	6163076,24307005
		2	22180ms/20616ms	6992039,25872083
		4	23622ms/23517ms	6956122,95615508

Table 8.27: Apache Rya MongoDB: LUBM Query 10

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	296ms/200ms	47765140,7534747
		2	301ms/226ms	46906097,1458241
		4	241ms/185ms	45229280,5752283
	2	1	283ms/223ms	43662597,2439706
		2	299ms/258ms	43612096,6996962
		4	269ms/237ms	40525402,7575994
	4	1	283ms/236ms	44752865,2830721
		2	288ms/235ms	43021502,2477167
		4	251ms/212ms	40397776,4259194
20	1	1	2311ms/2425ms	497172,306398477
		2	2010ms/1910ms	418557,234428887
		4	1891ms/1821ms	357256,049896656
	2	1	1980ms/1904ms	413101,346135914
		2	2147ms/1898ms	591624,691455805
		4	2060ms/1914ms	510149,427153526
	4	1	1987ms/1845ms	459967,484871172
		2	1876ms/1718ms	422531,398389347
		4	1943ms/1800ms	394135,905496269

Table 8.28: Apache Rya MongoDB: LUBM Query 11

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	220ms/183ms	44931735,9272209
		2	237ms/200ms	40590497,9583637
		4	185ms/154ms	44072264,9569635
	2	1	228ms/205ms	44796108,3000057
		2	236ms/210ms	44810404,7181532
		4	219ms/195ms	41835428,6950643
	4	1	234ms/195ms	47379371,3920447
		2	232ms/196ms	41079702,7889945
		4	218ms/171ms	38470861,9047317
20	1	1	229ms/178ms	27131,9311823688
		2	218ms/174ms	21947,4197483541
		4	219ms/177ms	19234,3118245674
	2	1	211ms/167ms	25673,6283959865
		2	247ms/197ms	33381,0408310942
		4	214ms/188ms	32061,3256713258
	4	1	211ms/179ms	41807,5079771694
		2	195ms/164ms	40473,9720279793
		4	214ms/185ms	40900,8243713257

Table 8.29: Apache Rya MongoDB: LUBM Query 12

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	98197ms/98210ms	45419674,6171273
		2	101941ms/101057ms	46354076,5998821
		4	84811ms/85051ms	49052506,7277415
	2	1	107749ms/107471ms	46744319,7251503
		2	109062ms/105825ms	47177547,0669564
		4	94011ms/92898ms	45752717,8346743
	4	1	131269ms/132389ms	42171382,8323031
		2	111295ms/111789ms	43064459,6105818
		4	93305ms/91779ms	41099399,6342401
20	1	1	2716423ms/2717166ms	2608201,41389611
		2	2682252ms/2668077ms	2430049,03312238
		4	2324683ms/2330832ms	2927146,78536985
	2	1	2384449ms/2386296ms	2726723,9390317
		2	2705016ms/2699740ms	2524773,41868955
		4	2568054ms/2561520ms	2800484,19451476
	4	1	2652388ms/2671752ms	2579080,29265448
		2	2500734ms/2495277ms	2699549,01028574
		4	2438669ms/2439684ms	2892080,68099369

Table 8.30: Apache Rya MongoDB: LUBM Query 13

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	379ms/268ms	43723103,5576159
		2	376ms/285ms	46853570,8349834
		4	309ms/269ms	46848784,5947482
	2	1	330ms/201ms	49629329,5588675
		2	375ms/286ms	49920966,9503264
		4	308ms/232ms	44250117,0459405
	4	1	314ms/207ms	42975055,0115523
		2	347ms/282ms	43925388,5646074
		4	291ms/214ms	38411708,6694044
20	1	1	4121ms/3877ms	6097051,28026586
		2	3835ms/3188ms	7582771,10447894
		4	3592ms/3274ms	6105243,67825752
	2	1	4044ms/3902ms	7453972,74271396
		2	3985ms/3350ms	10064967,4890475
		4	3820ms/3337ms	7578596,06697413
	4	1	3708ms/3286ms	6287822,28250223
		2	3840ms/3484ms	7003706,51211377
		4	4033ms/3699ms	7007934,17259532

Table 8.31: Apache Rya MongoDB: LUBM Query 14

LUBM	#Nodes	CPU	Loading Time	Memory usage (peak)	Storage size	Network Communication (peak)
1	1	1	102s	5529919488	56164352	1442421,92817144
		2	82s	9567404032	63258624	1022987,07937882
		4	51s	9334587392	63242240	1724170,17452415
	2	1	88s	6954217472	126484480	3435404,20880214
		2	60s	9445199872	139104256	3898822,48325863
		4	51s	10445852672	124837888	3869730,94104516
	4	1	81s	11181125632	241725440	3273918,14567899
		2	59s	12752384000	225091584	4213914,55112675
		4	51s	15700897792	225533952	5090040,17310004
20	1	1	1149s	28154044416	715661312	14575578,5022696
		2	694s	40923213824	691200000	13634798,7621221
		4	411s	44824068096	691220480	17202175,4383775
	2	1	631s	43549376512	1431318528	29369493,0338758
		2	424s	61316096000	1431367680	27081102,6316787
		4	268s	65593061376	1382612992	30839833,6014025
	4	1	458s	41782460416	2111492096	33511184,5477478
		2	267s	96843898880	2185142272	43981292,5918687
		4	215s	122339913728	2185551872	53848230,7222331

Table 8.32: SANSA-Stack Loading Data

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	14339ms/8205ms	2495381,58737327
		2	9261ms/4863ms	867550,522111752
		4	5602ms/2359ms	2912969,44403835
	2	1	10566ms/5015ms	3605274,94140156
		2	5454ms/3257ms	3038714,62801133
		4	4213ms/2405ms	1812252,78956578
	4	1	8353ms/3794ms	2868911,73619213
		2	6131ms/3734ms	2612872,28389436
		4	3729ms/2139ms	2503078,68738697
20	1	1	79546ms/34919ms	11987938,1279602
		2	47153ms/22100ms	20779686,8351557
		4	29015ms/9587ms	29570003,1517028
	2	1	42268ms/17532ms	22980199,0873366
		2	25380ms/11410ms	25119273,771702
		4	15572ms/8005ms	32178558,3788683
	4	1	33877ms/18214ms	29340121,7496039
		2	14358ms/6409ms	41419995,2796473
		4	9437ms/4469ms	33480042,5634029

Table 8.33: SANSA-Stack: LUBM Query 1

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
		1	101338ms/63038ms	3677858,42402121
	1	2	65138ms/42287ms	7125288,74910138
		4	34942ms/24907ms	9916127,92397149
		1	67125ms/38926ms	6653910,62695076
1	2	2	40998ms/29237ms	10803006,8817199
		4	28502ms/20923ms	14149577,1717669
		1	48437ms/29104ms	8231224,04951853
	4	2	34895ms/22583ms	11672745,7570307
		4	26058ms/21338ms	16799519,6904768
		1	587747ms/570856ms	11439127,7922541
	1	2	320728ms/301653ms	14521747,5538566
		4	151276ms/141969ms	26104569,3804879
		1	306684ms/278891ms	20463586,0436843
20	2	2	177057ms/164717ms	26549306,0376723
		4	108020ms/100290ms	31162338,3128211
		1	232928ms/213213ms	26423194,049269
	4	2	103584ms/94705ms	38825634,4612809
		4	73009ms/68707ms	48602279,3145918

Table 8.34: SANSA-Stack: LUBM Query 2

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
		1	5711ms/5063ms	1154560,53318338
	1	2	3515ms/3009ms	696509,819471174
		4	2012ms/1726ms	1601692,86522079
		1	3873ms/3079ms	1342082,28261702
1	2	2	2332ms/2079ms	1606652,21167155
		4	1533ms/1321ms	1120549,01040025
		1	2223ms/1762ms	1525252,3300819
	4	2	1823ms/1630ms	1581828,55649756
		4	1840ms/1432ms	1687216,4094863
		1	45190ms/34447ms	10240417,7142224
	1	2	28359ms/22116ms	10640373,0180557
		4	12444ms/9407ms	12442919,5545704
		1	22392ms/16788ms	11242465,5609034
20	2	2	13302ms/10065ms	9861407,77554696
		4	8850ms/7089ms	11245409,2405144
		1	20678ms/16884ms	13109177,8425588
	4	2	7756ms/5944ms	12356344,7559766
		4	4865ms/3941ms	14245380,3888508

Table 8.35: SANSA-Stack: LUBM Query 3

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	12211ms/7035ms	2992887,83705179
		2	7570ms/4677ms	1372153,98764653
		4	4106ms/2682ms	4093956,00201962
	2	1	8276ms/4856ms	4098242,32652979
		2	4557ms/3131ms	3129513,30207763
		4	3210ms/2417ms	2363285,88224412
	4	1	5410ms/3275ms	3721133,0059273
		2	4057ms/2661ms	3831939,77459848
		4	3235ms/2457ms	3272176,41673161
20	1	1	97303ms/55408ms	12288117,9679419
		2	55531ms/33003ms	21608957,2129321
		4	31735ms/14565ms	26841473,7501516
	2	1	50127ms/27206ms	23904152,2049459
		2	28773ms/16826ms	30170540,6581861
		4	17491ms/10610ms	34425819,3790103
	4	1	35890/23942ms	35038352,0798031
		2	16572ms/10162ms	32838042,5789704
		4	10879ms/7714ms	43948945,3806678

Table 8.36: SANSA-Stack: LUBM Query 4

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	4632ms/4603ms	552708,117749289
		2	3178ms/2941ms	658451,889719888
		4	1607ms/1605ms	903159,16495858
	2	1	3140ms/3101ms	637520,636398128
		2	2454ms/2007ms	856802,404192299
		4	1678ms/1384ms	967779,699932039
	4	1	2113ms/1971ms	748391,890371216
		2	1994ms/1734ms	1071522,16874328
		4	1251ms/1230ms	1093602,9940922
20	1	1	35683ms/35760ms	2379674,46243117
		2	22276ms/22165ms	3748773,66067343
		4	10346ms/10335ms	5242682,65565581
	2	1	19056ms/18904ms	4231498,38427144
		2	11446ms/10672ms	3767552,8649109
		4	7166ms/7117ms	4868602,94594521
	4	1	15787ms/15758ms	4627823,49527721
		2	6206ms/6161ms	6306564,93978829
		4	4031ms/3985ms	4593207,316082

Table 8.37: SANSA-Stack: LUBM Query 5

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	3964ms/3732ms	520542,256482143
		2	2583ms/2399ms	490159,125489977
		4	1665ms/1476ms	667369,75184184
	2	1	3337ms/3296ms	492172,011670426
		2	1667ms/1580ms	542953,8282665
		4	1168ms/1126ms	621903,082583487
	4	1	2459ms/2434ms	624129,017546244
		2	1597ms/1595ms	657266,024927765
		4	1072ms/1036ms	742773,332680927
20	1	1	25004ms/25056ms	2237402,01983455
		2	14540ms/14346ms	3146271,10583885
		4	6625ms/6367ms	4083928,50254555
	2	1	14362ms/13992ms	3487171,89886202
		2	8604ms/8424ms	3672803,66802508
		4	5535ms/5392ms	4610075,00062863
	4	1	11314ms/10740ms	3833788,99208415
		2	4938ms/4633ms	5252624,4995814
		4	3450ms/3164ms	4938381,29385017

Table 8.38: SANSA-Stack: LUBM Query 6

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	18610ms/14275ms	2070673,32612768
		2	12012ms/9489ms	2739226,33577578
		4	6280ms/5103ms	3991813,00100232
	2	1	11557ms/8992ms	3065453,08383996
		2	7715ms/6406ms	3100933,76494745
		4	5217ms/4638ms	3755743,32237851
	4	1	8313ms/7038ms	2739947,36568639
		2	6264ms/5372ms	3961559,1248445
		4	5406ms/4510ms	5439925,69908238
20	1	1	132416ms/125057ms	7991988,84833546
		2	84681ms/79916ms	12201388,0362817
		4	36821ms/33992ms	15436962,9352183
	2	1	69974ms/64851ms	13086488,5822233
		2	41852ms/39177ms	12712133,5704921
		4	26750ms/25914ms	17210318,6573723
	4	1	57440ms/55278ms	13024484,5857013
		2	22305ms/21171ms	23696062,9131857
		4	14908ms/13555ms	22901805,8844563

Table 8.39: SANSA-Stack: LUBM Query 7

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	16182ms/16490ms	1601391,22862088
		2	11262ms/11054ms	2552202,04966405
		4	5857ms/5795ms	3320609,19790012
	2	1	10172ms/9921ms	2186760,17647531
		2	6887ms/6646ms	3008436,85958055
		4	5260ms/5186ms	3501543,02134466
	4	1	7349ms/7485ms	2938459,96287078
		2	5676ms/5767ms	3811098,24248926
		4	4558ms/4456ms	4236924,21467031
20	1	1	136259ms/136997ms	2370703,958233
		2	76969ms/78998ms	4447835,83252583
		4	34545ms/34624ms	9625545,71415499
	2	1	73038ms/73550ms	5679514,76067321
		2	43591ms/42611ms	9603197,13864304
		4	24882ms/24907ms	15569688,8088328
	4	1	53648ms/54501ms	8348205,9542526
		2	24477ms/24080ms	18572834,7881652
		4	16369ms/16472ms	22743298,5758818

Table 8.40: SANSA-Stack: LUBM Query 8

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	72213ms/59562ms	3056166,92330204
		2	47709ms/40103ms	4600538,17835142
		4	26124ms/22945ms	9024771,28491825
	2	1	46834ms/40664ms	4923568,11751164
		2	30070ms/25364ms	8562683,06087256
		4	21142ms/18235ms	11294541,2246019
	4	1	31865ms/27385ms	7211330,21734858
		2	26112ms/23784ms	11054745,3920012
		4	19378ms/17283ms	16192123,1730062
20	1	1	561694ms/556405ms	7200862,3229212
		2	293284ms/289926ms	10530519,3912947
		4	136972ms/134694ms	17341316,8472034
	2	1	301483ms/298656ms	12587800,7549032
		2	171179ms/165461ms	13936286,1510468
		4	100166ms/96661ms	26152516,9827094
	4	1	211662ms/205885ms	16103902,9448193
		2	99619ms/100061ms	30583274,4197032
		4	67034ms/66593ms	38259702,4198761

Table 8.41: SANSA-Stack: LUBM Query 9

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	4834ms/4747ms	629274,338477849
		2	3070ms/3099ms	766264,424522223
		4	1613ms/1593ms	1015801,35011845
	2	1	3118ms/3079ms	743260,705669509
		2	2048ms/2008ms	880802,291086888
		4	1539ms/1428ms	914850,020291169
	4	1	2137ms/2047ms	798642,724925841
		2	1783ms/1783ms	755302,182800727
		4	1332ms/1223ms	1220532,41635919
20	1	1	34163ms/34109ms	2206541,29834259
		2	22152ms/22099ms	3690751,17954012
		4	9553ms/9428ms	5309954,1977485
	2	1	19164ms/19243ms	3947866,20451657
		2	10493ms/10453ms	3713435,05705063
		4	7696ms/7673ms	4869369,7132966
	4	1	16007ms/16037ms	4368935,58806451
		2	6298ms/6065ms	6145225,57599801
		4	4538ms/4340ms	5439639,11815674

Table 8.42: SANSA-Stack: LUBM Query 10

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	4406ms/4333ms	511458,177139896
		2	2849ms/2819ms	678051,367095816
		4	1666ms/1545ms	898293,585804383
	2	1	2845ms/2798ms	675756,232451663
		2	1785ms/1812ms	779464,660856768
		4	1351ms/1255ms	806488,003356865
	4	1	1715ms/1738ms	617344,140414167
		2	1493ms/1489ms	926991,251446128
		4	1232ms/990ms	939965,785577755
20	1	1	32834ms/32586ms	2261221,05027386
		2	18131ms/18194ms	3112322,29109496
		4	7786ms/7746ms	4348748,18195972
	2	1	17906ms/17926ms	3694029,22512221
		2	9883ms/9778ms	3764785,35528651
		4	5680ms/5649ms	4927302,71617643
	4	1	13013ms/12916ms	3306436,4981586
		2	5865ms/5786ms	3684849,42745276
		4	3758ms/3801ms	5015095,7283231

Table 8.43: SANSA-Stack: LUBM Query 11

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
		1	13770ms/13223ms	1377148,10160144
	1	2	9067ms/8804ms	2266405,8167108
		4	5498ms/5164ms	3186922,3834235
		1	8732ms/8895ms	1475221,38964588
1	2	2	5999ms/5921ms	2296080,33504958
		4	4784ms/4261ms	2808626,13481941
		1	5551ms/5536ms	2169051,02460966
	4	2	5143ms/4848ms	3230540,13988307
		4	4346ms/3822ms	3615695,5513029
		1	112473ms/111673ms	2696074,53933473
	1	2	71582ms/70479ms	4858370,5906871
		4	31033ms/30942ms	10840215,447151
		1	59690ms/60729ms	5298836,48473577
20	2	2	34034ms/34496ms	7905465,28431223
		4	28630ms/28358ms	10549026,4008102
		1	48885ms/48725ms	6805615,73139439
	4	2	19809ms/20169ms	12189134,6099048
		4	13408ms/13229ms	14336663,5559989

Table 8.44: SANSA-Stack: LUBM Query 12

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
		1	4892ms/4252ms	1206303,1206153
	1	2	3121ms/2673ms	701518,515251984
		4	1698ms/1465ms	1051580,93620636
		1	3340ms/2716ms	1574358,81894028
1	2	2	2055ms/1812ms	1436254,61329568
		4	1425ms/1228ms	1053812,1767783
		1	2051ms/1664ms	1348223,68963789
	4	2	1731ms/1391ms	1275639,61928854
		4	1328ms/1102ms	1765071,58820156
		1	43468ms/32668ms	8567086,84994654
	1	2	24887ms/18155ms	8021078,09640238
		4	11122ms/7809ms	14920293,0464032
		1	23515ms/17905ms	10559395,0989329
20	2	2	13000ms/9928ms	11274874,0719658
		4	7989ms/5948ms	11800741,6224488
		1	17448ms/13282ms	10544349,84655
	4	2	8654ms/5987ms	10575495,6757433
		4	4791ms/3878ms	13564889,7169469

Table 8.45: SANSA-Stack: LUBM Query 13

LUBM	#Nodes	CPU	Query time average/median	Network Communication (peak)
1	1	1	3735ms/3631ms	403081,277348628
		2	2483ms/2412ms	531686,611568046
		4	1276ms/1232ms	424993,30879698
	2	1	2665ms/2557ms	573241,366427509
		2	1671ms/1571ms	632099,772817458
		4	1080ms/1003ms	637060,262616723
	4	1	1748ms/1723ms	621778,125126033
		2	1318ms/1269ms	561395,477577906
		4	912ms/811ms	866952,246632068
20	1	1	25241ms/25173ms	2265035,7629269
		2	14349ms/14022ms	3538685,36312557
		4	6477ms/6210ms	4076708,14369514
	2	1	14317ms/14448ms	2982212,64465444
		2	8339ms/8301ms	4490006,80100038
		4	5643ms/5343ms	3859976,53525898
	4	1	10898ms/10932ms	4453711,28339243
		2	5426ms/5433ms	5140770,40851332
		4	3294ms/3219ms	4124916,02964805

Table 8.46: SANSA-Stack: LUBM Query 14

Query	1 DataNode	2 DataNodes
1	9561ms/4345ms	8818ms/4061ms
2	60862ms/35628ms	58967ms/35050ms
3	3351ms/2623ms	3084ms/2570ms
4	7154ms/4698ms	6944ms/4283ms
5	2548ms/2508ms	2942ms/2968ms
6	2525ms/2869ms	2456ms/2638ms
7	11099ms/9265ms	10053ms/8031ms
8	9237ms/9150ms	8931ms/8913ms
9	41493ms/34144ms	39007ms/32310ms
10	2698ms/2667ms	2707ms/2638ms
11	2388ms/2389ms	2531ms/2537ms
12	7526ms/7325ms	7526ms/7510ms
13	2680ms/2258ms	2721ms/2264ms
14	2135ms/1991ms	2133ms/2195ms

Table 8.47: SANSA-Stack HDFS DataNodes Comparison, 2 Worker Nodes, 1 Core

Bibliography

[AAH16] Witold Abramowicz, Sören Auer, and Tom Heath. Linked data in business. *Bus. Inf. Syst. Eng.*, 58(5):323–326, 2016.

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

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422. ACM, 2007.

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[BCG+18] Bradley R. Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, Simone Rondelli, Alexander Ryazanov, Michael Schmidt, Kunal Sengupta, Bryan B. Thompson, Divij Vaidya, and Shawn Wang. Amazon Neptune: Graph data management in the cloud. In Marieke van Erp, Medha Atre, Vanessa López, Kavitha Srinivas, and Carolina Fortuna, editors, *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018*, volume 2180 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[BEP14] Peter A. Boncz, Orri Erling, and Minh-Duc Pham. Experiences with Virtuoso cluster RDF column store. In *Linked Data Management*, pages 239–259. Chapman and Hall/CRC, 2014.

[BHB09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[BLR+20] Theodora S. Brisimi, Vanessa López, Valentina Rho, Marco Luca Sbodio, Gabriele Picco, Morten Kristiansen, John Segrave-Daly, and Conor Cullen. Ontology-guided policy information extraction for healthcare fraud detection. In Louise Bilenberg Pape-Haugaard, Christian Lovis, Inge Cort Madsen, Patrick Weber, Per Hostrup Nielsen, and Philip Scott, editors, *Digital Personalized Health and Medicine - Proceedings of MIE 2020, Medical Informatics Europe, Geneva, Switzerland, April 28 - May 1, 2020*, volume 270 of *Studies in Health Technology and Informatics*, pages 879–883. IOS Press, 2020.

[CCK+17] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[CZCG12] Lei Chen, Haifei Zhang, Ying Chen, and Wenping Guo. Blank nodes in RDF. *J. Softw.*, 7(9):1993–1999, 2012.

[Daw16] Omer Dawelbeit. *Investigating elastic cloud based RDF processing*. PhD thesis, University of Reading, Berkshire, UK, 2016.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[DGPN13] Francesco Draicchio, Aldo Gangemi, Valentina Presutti, and Andrea Giovanni Nuzzolese. FRED: from natural language text to RDF and OWL in one click. In Philipp Cimiano, Miriam Fernández, Vanessa López, Stefan Schlobach, and Johanna Völker, editors, *The Semantic Web: ESWC 2013 Satellite Events - ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers*, volume 7955 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2013.

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

[EM09] Orri Erling and Ivan Mikhailov. Virtuoso: RDF support in a native RDBMS. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Semantic Web Information Management - A Model-Based Perspective*, pages 501–519. Springer, 2009.

[Erl12] Orri Erling. Virtuoso, a hybrid RDBMS/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.

[FHA18] Md. Nowraj Farhan, Md. Ahsan Habib, and Arshad Ali. A study and performance comparison of MapReduce and Apache Spark on Twitter data on Hadoop cluster. *International Journal of Information Technology and Computer Science*, 10:61–70, 07 2018.

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

[GHM+08] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter F. Patel-Schneider, and Ulrike Sattler. OWL 2: The next step for OWL. *J. Web Semant.*, 6(4):309–322, 2008.

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

[Har11] A. Harth. CumulusRDF: Linked data management on nested key-value stores. 2011.

[HLSL09] Steve Harris, Nick Lamb, Nigel Shadbolt, and Garlik Ltd. 4store: The design and implementation of a clustered RDF store. *Proc. SSWS*, 01 2009.

[HPvH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: the making of a web ontology language. *J. Web Semant.*, 1(1):7–26, 2003.

[HZU+12] Aidan Hogan, Antoine Zimmermann, Jürgen Umbrich, Axel Polleres, and Stefan Decker. Scalable and distributed methods for entity matching, consolidation and disambiguation over linked data corpora. *J. Web Semant.*, 10:76–110, 2012.

[JA20] Benymol Jose and Sajimon Abraham. Performance analysis of NoSQL and relational databases with MongoDB and MySQL. *Materials Today: Proceedings*, 24:2036–2043, 2020. International Multi-conference on Computing, Communication, Electrical & Nanotechnology, I2CN-2K19, 25th & 26th April 2019.

[KAB+14] Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Matthew Hubbell, Peter Michaleas, Julie Mullen, Andrew Prout, Albert Reuther, Antonio Rosa, and Charles Yee. Achieving 100, 000, 000 database inserts per second using Accumulo and D4M. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, pages 1–6. IEEE, 2014.

[KM15] Zoi Kaoudi and Ioana Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1):67–91, 2015.

[Kob] OLG Koblenz. Beschluss vom 30.03.2021 - 5 ws 16/21.

[KP15] Je-Min Kim and Young-Tack Park. Scalable OWL-Horst ontology reasoning using SPARK. In *2015 International Conference on Big Data and Smart Computing, BIGCOMP 2015, Jeju, South Korea, February 9-11, 2015*, pages 79–86. IEEE Computer Society, 2015.

[Läm08] Ralf Lämmel. Google's MapReduce programming model - revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.

[LM13] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. pages 15–19, 08 2013.

[LN04] Thorsten Liebig and Olaf Noppens. OntoTrack: Combining browsing and editing with reasoning and explaining for OWL Lite ontologies. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, volume 3298 of *Lecture Notes in Computer Science*, pages 244–258. Springer, 2004.

[LQHL17] Hao Lian, Zemin Qin, Tieke He, and Bin Luo. Knowledge graph construction based on judicial data with social media. In *14th Web Information Systems and Applications Conference, WISA 2017, Liuzhou, Guangxi Province, China, November 11-12, 2017*, pages 225–227. IEEE, 2017.

[LSB+17] Jens Lehmann, Gezim Sejdiu, Lorenz Bühmann, Patrick Westphal, Claus Stadler, Ivan Ermilov, Simon Bin, Nilesh Chakraborty, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, and Hajira Jabeen. Distributed semantic analytics using the SANSA stack. In *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II*, volume 10588 of *Lecture Notes in Computer Science*, pages 147–155. Springer, 2017.

[LSEA16] Ayman E. Lotfy, Ahmed I. Saleh, Haitham A. El-Ghareeb, and Hesham A. Ali. A middle layer solution to support ACID properties for NoSQL databases. *J. King Saud Univ. Comput. Inf. Sci.*, 28(1):133–145, 2016.

[MGS+19] Mohamed Nadjib Mami, Damien Graux, Simon Scerri, Hajira Jabeen, Sören Auer, and Jens Lehmann. Squerall: Virtual ontology-based access to heterogeneous and large data sources. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II*, volume 11779 of *Lecture Notes in Computer Science*, pages 229–245. Springer, 2019.

[MMW+21] Pedro Martins, Francisco Morgado, Cristina Wanzeller, Filipe Sá, and Maryam Abbasi. MongoDB, Couchbase, and CouchDB: A comparison. In Álvaro Rocha, Hojjat Adeli, Gintautas Dzemyda, Fernando Moreira, and Ana Maria Ramalho Correia, editors, *Trends and Applications in Information Systems and Technologies - Volume 2, WorldCIST 2021, Terceira Island, Azores, Portugal, 30 March - 2 April, 2021*, volume 1366 of *Advances in Intelligent Systems and Computing*, pages 469–480. Springer, 2021.

[MTS+21] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, Dimitrios Zissis, and Dimosthenis Anagnostopoulos. MongoDB vs PostgreSQL: A comparative study on performance aspects. *GeoInformatica*, 25(2):243–268, 2021.

[Opd21] Andreas L. Opdahl. Knowledge graphs and natural-language processing. *CoRR*, abs/2101.06111, 2021.

[PCR12] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: a scalable RDF triple store for the clouds. In *1st International Workshop on Cloud Intelligence (colocated with VLDB 2012), Cloud-I '12, Istanbul, Turkey, August 31, 2012*, page 4. ACM, 2012.

[PCR15] Roshan Punnoose, Adina Crainiceanu, and David Rapp. SPARQL in the cloud using Rya. *Inf. Syst.*, 48:181–195, 2015.

[PKAS17] Dr. Yusuf Perwej, Bedine Kerim, Mohammed Adrees, and Osama Sheta. An empirical exploration of the Yarn in big data. *International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868 , Foundation of Computer Science FCS, New York, USA*, Volume 12:Page 19– 29, 12 2017.

[PMNH18] Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. Dynamic data exchange in distributed RDF stores. *IEEE Trans. Knowl. Data Eng.*, 30(12):2312–2325, 2018.

[RDE+07] Kurt Rohloff, Mike Dean, Ian Emmons, Dorene Ryder, and John Sumner. An evaluation of triple-store technologies for large data stores. In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, OTM Academy Doctoral Consortium, MONET, OnToContent, ORM, PerSys, PPN, RDDS, SSWS, and SWWS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part II*, volume 4806 of *Lecture Notes in Computer Science*, pages 1105–1114. Springer, 2007.

[RS10] Kurt Rohloff and Richard E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *SPLASH Workshop on Programming Support Innovations for Emerging Distributed Applications (PSI EtA - ΨΘ 2010), October 17, 2010, Reno/Tahoe, Nevada, USA*, page 4. ACM, 2010.

[Rus16] Michael Ruster. Large-scale reasoning with OWL. *CoRR*, abs/1602.04473, 2016.

[SCH+11] Manuel Salvadores, Gianluca Correndo, Steve Harris, Nick Gibbins, and Nigel Shadbolt. The design and implementation of minimal RDFS backward reasoning in 4store. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*, volume 6644 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2011.

[SD17] Daniel Seybold and Jörg Domaschka. Is distributed database evaluation cloud-ready? In *New Trends in Databases and Information Systems - ADBIS 2017 Short Papers and Workshops, AMSD, BigNovelTI, DAS, SW4CH, DC, Nicosia, Cyprus, September 24-27, 2017, Proceedings*, volume 767 of *Communications in Computer and Information Science*, pages 100–108. Springer, 2017.

[SGK+19] Gezim Sejdiu, Damien Graux, Imran Khan, Ioanna Lytra, Hajira Jabeen, and Jens Lehmann. Towards a scalable semantic-based distributed approach for SPARQL query evaluation. In *Semantic Systems. The Power of AI and Knowledge Graphs - 15th International Conference, SEMANTiCS 2019, Karlsruhe, Germany, September 9-12, 2019, Proceedings*, volume 11702 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 2019.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[SN16] Adam Sotona and Stefan Negru. How to feed Apache HBase with petabytes of RDF data: An extremely scalable RDF store based on Eclipse RDF4J framework and Apache HBase database. In *Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016), Kobe, Japan, October 19, 2016*, volume 1690 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[SOTY13] Scott M. Sawyer, B. David O'Gwynn, An Tran, and Tamara Yu. Understanding query performance in Accumulo. In *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*, pages 1–6. IEEE, 2013.

[SSGL19a] Claus Stadler, Gezim Sejdiu, Damien Graux, and Jens Lehmann. Querying large-scale RDF datasets using the SANSA framework. In *Proceedings of the ISWC 2019 Satellite Tracks (Posters & Demonstrations, Industry, and Outrageous Ideas) co-located with 18th International Semantic Web Conference (ISWC 2019), Auckland, New Zealand, October 26-30, 2019*, volume 2456 of *CEUR Workshop Proceedings*, pages 285–288. CEUR-WS.org, 2019.

[SSGL19b] Claus Stadler, Gezim Sejdiu, Damien Graux, and Jens Lehmann. Sparklify: A scalable software component for efficient evaluation of SPARQL queries over distributed RDF datasets. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II*, volume 11779 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2019.

[tH05] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *J. Web Semant.*, 3(2-3):79–115, 2005.

[UKM+12] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri E. Bal. WebPIE: A web-scale parallel inference engine using MapReduce. *J. Web Semant.*, 10:59–75, 2012.

[UKOvH09] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable distributed reasoning using MapReduce. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, volume 5823 of *Lecture Notes in Computer Science*, pages 634–649. Springer, 2009.

[UvHSB11] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri E. Bal. QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham

Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 730–745. Springer, 2011.

[Wil06] Kevin Wilkinson. Jena property table implementation. 11 2006.

[WSKR03] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8, 2003*, pages 131–150, 2003.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[ZDL+12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13, 2012*. USENIX Association, 2012.