UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

From the Institute of Information Systems
of the University of Lübeck
Director: Univ.-Prof. Dr. rer. nat. habil. Ralf Möller

Data Partitioning and Query Optimization in the Semantic Internet of Things

Dissertation for Fulfillment of Requirements for the Doctoral Degree
of the University of Lübeck
- from the Department of Computer Sciences -
Submitted by Alexander Benjamin Warnke from Henstedt-Ulzburg

Lübeck, 2023

First referee:	Prof. Dr. rer. nat. habil. Sven Groppe
Second referee:	Prof. Dr. rer. nat. habil. Josef Ingenerf
Date of oral examination:	July 14, 2023
Approved for printing.	Lübeck, July 17, 2023

Abstract

The Internet of Things (IoT) connects heterogeneous physical objects « Things » with the virtual world. This network allows the data exchange between arbitrary devices to enable more complex applications. The Semantic Web (SW) was invented to support machines in understanding human readable texts. Metadata is attached to the text to achieve this goal. In the Semantic Internet of Things (SIoT), the SW metadata concept is applied to the IoT data exchange. This metadata allows large amounts of data from different sources to be processed and stored together. The amount of sensors and, therefore, data is rapidly increasing. As the complexity and volume of data continue to increase, the existing DataBase Management Systems (DBMSs) are less and less able to meet the resulting requirements. The requirements for a DBMS used in the SIoT include handling heterogeneous hardware, performance, heterogeneous networks, communication protocols, parallelism, distributed storage, and distributed processing. Therefore, a new DBMS called Logisch und Physikalisch Optimierte Semantic Web Datenbank-Engine 3000 (LUPOSDATE3000) is designed with these topics in mind.

All of these requirements depend on the quality of the query plan, which in turn depends on the underlying data organization. Therefore, this work studies the data and query organization in a SIoT environment. This work is organized into three parts: The first part elaborates on the consequences of multiple local partition schemes regarding storage requirements and performance. It is shown that the optimal number of partitions depends on the amount of data and the number and selectivity of join operators. The second part shows that the network traffic can be reduced when the join order optimizer is granted access to routing information. The third part analyzes the cost and benefit of Machine Learning (ML)-based join order optimizers. We show that ML can optimize the join order faster than traditional approaches while maintaining a similar quality.

Kurzfassung

Das Internet of Things (IoT) verbindet heterogene physische Objekte « Things » mit der virtuellen Welt. Dieses Netzwerk ermöglicht den Datenaustausch zwischen beliebigen Geräten, um komplexere Anwendungen zu ermöglichen. Das Semantic Web (SW) wurde erfunden, um Maschinen beim Verstehen von menschenlesbaren Texten zu unterstützen. Um dieses Ziel zu erreichen, werden dem Text Metadaten hinzugefügt. Im Semantic Internet of Things (SIoT) wird das SW-Metadatenkonzept auf den IoT-Datenaustausch angewendet. Dies ermöglicht es, große Datenmengen aus unterschiedlichen Quellen gemeinsam zu verarbeiten und zu speichern. Die Menge an Sensoren und damit Daten nimmt rasant zu. Da die Komplexität und Menge der Daten weiter zunimmt, werden die bestehenden DataBase Management Systems (DBMSs) immer weniger den daraus resultierenden Anforderungen gerecht. Die Anforderungen an ein DBMS, das im SIoT verwendet wird, umfassen den Umgang mit heterogener Hardware, Leistung, heterogenen Netzwerken, Kommunikationsprotokollen, Parallelität, sowie verteilter Speicherung und Verarbeitung. Daher wurde ein neues DBMS namens Logisch und Physikalisch Optimierte Semantic Web Datenbank-Engine 3000 (LUPOSDATE3000) unter Berücksichtigung dieser Eigenschaften entwickelt.

Alle diese Anforderungen hängen von der Qualität des Abfrageplans ab, der wiederum von der zugrunde liegenden Datenorganisation abhängt. Daher werden in dieser Arbeit die Daten- und Abfrageorganisation in einer SIoT-Umgebung untersucht. Diese Arbeit ist in drei Teile gegliedert: Der erste Teil befasst sich mit dem Speicherbedarf sowie der Performance durch mehrerer lokaler Partitionierungsschemata. Es wird gezeigt, dass die optimale Anzahl von Partitionen von der Datenmenge und der Anzahl sowie Selektivität der Join Operatoren abhängt. Im zweiten Teil wird gezeigt, dass der Netzwerkverkehr reduziert werden kann, wenn der Joinreihenfolge-Optimierer Zugriff auf Routing-Informationen hat. Der dritte Teil analysiert die Kosten und den Nutzen von Machine Learning (ML)-basierten Joinreihenfolge-Optimierern. Es wird gezeigt, dass ML die Joinreihenfolge schneller als herkömmliche Ansätze optimieren kann, während eine ähnliche Qualität beibehalten wird.

Contents

1 Introduction **1**
1.1 Research Questions and Focus of this Work 2
1.2 Publications and Presentations 4
1.3 Organization of this Work 4

2 Fundamentals **7**
2.1 Semantic Web 7
2.1.1 Layer Cake of Semantic Web 8
2.1.2 UCS 9
2.1.3 IRI, URL, and URI 10
2.1.4 XML 11
2.1.5 RDF 12
2.1.6 TTL 14
2.1.7 SPARQL 15
2.1.8 RDFS 17
2.1.9 OWL 19
2.1.10 SOSA 20
2.1.11 SHACL 21
2.1.12 RIF 22
2.2 Internet of Things 23
2.3 Semantic Internet of Things 26
2.4 Semantic Web DBMS 26
2.4.1 Dictionary 27
2.4.2 Indexing Schemes 28
2.4.3 Data Partitioning and Data Distribution 29
2.4.4 DBMS and the Internet of Things 32
2.4.5 Join-Algorithms 34

I Effect of Multiple Partitioning Schemes on Storage Requirement and Performance 35

3 LUPOSDATE3000 37
3.1 Multi-Platform Capabilities . . . 37
3.2 Dictionary . . . 38
3.3 Pipelining . . . 40
3.4 Indices, Partitioning, and Distribution . . . 41
3.4.1 Compression . . . 41
3.5 Optimizers . . . 42
3.5.1 Greedy Join Order Optimizer . . . 44
3.5.2 Dynamic Programming Join Order Optimizer . . . 44
3.5.3 Interface for External Optimizers . . . 45

4 Experimental Setup 47
4.1 Other Semantic Web DBMS . . . 47
4.2 Public available Real World Datasets . . . 49

5 Flexible Data Partitioning 53
5.1 The Idea of the Flexible Data Partitioning Approach . . . 53
5.2 Evaluation . . . 55
5.2.1 Dataset and Queries . . . 55
5.2.2 Benchmarks . . . 57
5.3 Conclusion . . . 65

II Using Topology Information to Reduce the Network Traffic 67

6 Simulator 69
6.1 Fundamentals . . . 70
6.1.1 Simulator . . . 70
6.1.2 Routing . . . 73
6.2 The Simulator SIMORA . . . 76
6.2.1 Flexibility and Configuration . . . 76
6.2.2 Time Model . . . 77
6.2.3 Communication Model . . . 77
6.2.4 Network Stack . . . 78
6.2.5 API Design . . . 79
6.2.6 Predefined Topologies . . . 80
6.3 Evaluation . . . 80

6.4 Summary 81

7 Dynamic Content Multicast **83**
7.1 The Idea of Dynamic Content Multicast 84
7.1.1 Inverse Dynamic Content Multicast Tree 85
7.2 Application View 86
7.3 Scenario 86
7.4 Evaluate Cooperative Routing Protocols 87
7.5 Conclusion 88

8 Benchmark Scenario **89**
8.1 Fundamentals 90
8.2 The Benchmark Scenario 91
8.2.1 Setup Script 91
8.2.2 Ontology 93
8.2.3 Queries 94
8.2.4 Expandability 95
8.3 Summary 96

9 Topology Operator Distribution **97**
9.1 The Approach 98
9.1.1 Extended Routing Tables 99
9.1.2 Query Distribution 100
9.2 Evaluation 104
9.2.1 Insert Queries 104
9.2.2 Read Queries 107
9.3 Summary 110

III Machine Learning Based Join Order Optimization **111**

10 Machine Learning **113**
10.1 Related Work 115
10.1.1 Machine Learning Approaches 115
10.2 The proposed Algorithm 116
10.2.1 Generating Queries 116
10.2.2 The Machine Learning Algorithm 118
10.3 Evaluation 123
10.3.1 Environment 123
10.3.2 Evaluating Different Numbers of Triple Patterns 124

10.3.3 Evaluating Different Numbers of Training Steps 127
10.3.4 Evaluating what the Model has Learned 130
10.4 Evaluating ML on Network Traffic 130
10.5 Summary 134

11 Conclusion **135**
11.1 Summary of Contributions 135
11.2 Future Research Directions 136

IV Appendix **139**

A Benchmark Scenario SPARQL queries **141**

Bibliography **147**

List of Figures **165**

List of Abbreviations **171**

Curriculum Vitae **175**

List of Personal Publications **177**

Chapter 1

Introduction

The IoT connects heterogeneous physical objects - "Things" - with the virtual world. This network allows the data exchange between arbitrary devices to enable more complex applications. One core challenge is connecting many heterogeneous devices [50]. These heterogeneous devices are unavoidable in large projects like smart cities, where several organizations work together [136, 8]. Some of these devices have sensors that constantly provide new measured values. The devices must interact with each other to make the measured values usable. In addition, many of these devices are battery-powered. Transmitting data is one of the largest energy consumers [29]. Therefore, the transmitted data must be reduced as much as possible. However, most data must be stored to perform complex queries later, so sending some messages around is unavoidable. Since there is no uniform sensor standard in the IoT, there is also no fixed storage scheme. All data schemes can be converted to triples because triples are the most basic way to encode information.

The SW was invented to support machines in understanding human readable texts [13, 93]. However, metadata can be attached to arbitrary data as well. Ontologies are used to formalize the structure of knowledge.

In the SIoT, the SW technologies are applied to the IoT [98]. Therefore the metadata, which is the core of the SW, is used to annotate the data from the IoT to introduce a generic structure for sensors, actuators, and the environment. Triple stores can be used to store arbitrary directed graphs. This flexibility makes them ideal for the SIoT. The data must be stored in a physically distributed manner. In this context, there are a lot of open research questions to reduce network traffic and, thus, energy consumption. Also, validation of facts is possible. Contradictions in the data can be discovered and presented to the user. The ontologies and the metadata help to improve the quality of user queries because the machine has more context information [98].

SPARQL Protocol and Resource Description Framework (RDF) Query Language (SPARQL) is a query language commonly used to interact with triple stores. Due to the distributed data flow, the query evaluation must also be distributed. Some central operators are used frequently in SPARQL, as in many other languages. Except for the join operator, most operator implementations can be trivially parallelized. However, with the join operator, it is necessary to distribute the data so subtasks can be processed independently. Since the join operator is frequently used, several strategies exist. Thus the data can be stored partitioned, or only if necessary, selectively partitioned. Partitions can be formed after a single variable or by clustering algorithms. When partitioning, it is essential to consider the underlying data structure. For example, there are many more different object values than predicate values. Figure 4.3 supports this assumption later. The subject values often indicate a direct relationship. Depending on the application, there may be other relationships, so the strategy should always be flexible to match. Modern DBMS already reduce their network traffic for speed reasons. However, they do not focus on the network topology. Network topology is essential because sending multiple small data packets to another node that is far away will cause more package overhead than sending a large amount of data to a neighboring device. Therefore, the data distribution algorithm has to minimize the distances between data sources, data storage, and receivers. Therefore, new techniques are needed for networking between routing and application layers.

1.1 Research Questions and Focus of this Work

This work is motivated by three primary research questions:

- **What are the consequences of multiple partitioning schemes regarding storage space and the performance of the queries?** This question is answered in chapter 5. Due to the increased number of replications, the storage space requirements are increased along with the performance whenever much data is divided into many partitions. On the contrary, the performance is decreased if too many partitions are used for too few data. After the experiments, a function to estimate the optimal number of partitions is proposed.
- **How is the network traffic affected if the DBMS is granted access to the topology information of the routing protocol?** With more information about the environment, a more sophisticated execution plan can be created. These details, however, come at the

cost of a more complex plan generation process, which needs more time during the optimization phase. More details can be found in chapter 9.

- **How much cost and advantages are obtained using ML in query optimization?** ML algorithms feature a constant evaluation time of a given model, drastically improving the optimization phase's speed. However, a new training phase is introduced, which requires a lot of upfront computations. Also, the output quality is less predictable as ML naturally introduces some randomness. Chapter 10 shows the approach and the results.

Besides answering the main research questions, several secondary contributions are made. These contributions are necessary to solve the primary research questions, but their value is independent of them.

- The SPARQL DBMS LUPOSDATE3000 is open-source and publicly available (chapter 3) [158]. This DBMS is completely written in Kotlin, such that it can deal with heterogeneous Operating System (OS) environments. Another feature is that multiple components of the DBMS can be easily exchanged to experiment with new algorithms.

- The Simulator SIMulating Open Routing protocols for Application interoperability (SIMORA) (chapter 6) [159] can simulate entire applications within a network topology. The key feature is that the network interface can be modified such that the evaluation of the primary research questions becomes possible.

- Improved multicast techniques allow the distribution of similar data to multiple destinations using topology information within the application (chapter 7). Compared to a combination of state-of-the-art multicast and unicast, the number of packages and, thus, metadata is reduced, decreasing the overall traffic in the network.

- A scaleable benchmark scenario supporting network topologies (chapter 8). The primary contribution of the benchmark is that the data streams are decentralized, concurrent, and reproducible for any number of devices.

1.2 Publications and Presentations

I have implemented, evaluated and presented some key concepts of this dissertation at national and international conferences, workshops, and journals.

- The flexible partitioning approach (chapter 5) which is based on LUPOSDATE3000 (chapter 3) was published in 2021 at the Datenbanksysteme für Business, Technologie und Web (BTW) conference [162].
- The simulator SIMORA (chapter 6) with the Dynamic Content (DC) Multicast (chapter 7) was presented in 2022 at the International Conference on Fog and Edge Computing (ICFEC) [164].
- The benchmark scenario (chapter 8) was presented in 2022 in Proceedings of The International Workshop on Big Data in Emergent Distributed Environments: (BiDEDE) [161].
- The topology-driven join order optimization (chapter 9) was introduced in 2023 at the international database engineered applications symposium (IDEAS) [160].
- The ML approach for join order optimization (chapter 10) is submitted to the Journal of Web Semantics [163].

1.3 Organization of this Work

Figure 1.1 shows the dependencies between the chapters of this work. In the beginning, the DBMS LUPOSDATE3000 is introduced in chapter 3. This DBMS is used to compare and analyze the different approaches with each other.

The three main research questions are bundled with their required dependencies in parts, shown as rectangles in the figure. Chapter 5 considers an approach to improve the execution speed of SPARQL queries by using multiple partitioning schemes simultaneously. The intention is to gain speed up by selecting the most promising partitioning scheme at runtime based on the specific query. Then the topology-driven optimization approaches are explored in chapter 9. In focus are the changes to the network traffic, which are caused by different topologies, partitioning schemes, and join order optimizers. Finally, ML optimization techniques are introduced and compared in chapter 10 regarding their speed and the quality of the resulting query execution.

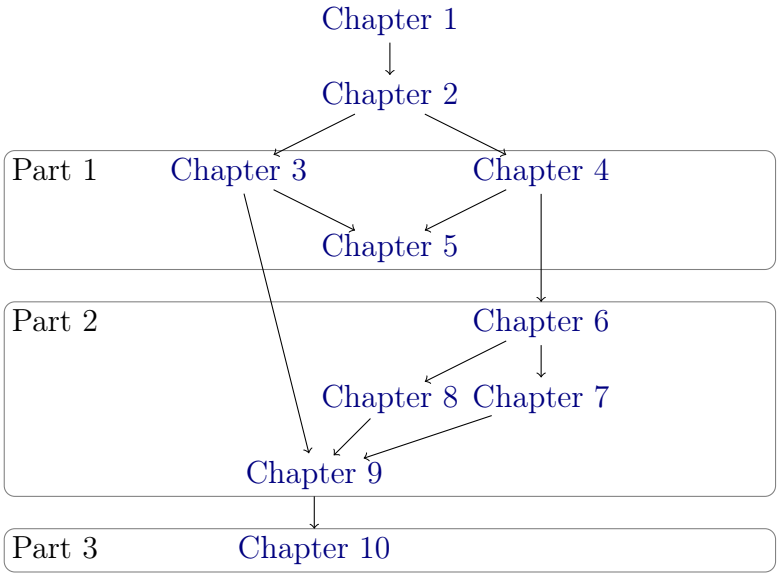


Figure 1.1: This figure shows the dependencies between the chapters of this document.

Definitions and foundations are introduced in chapter 2. These definitions are used throughout the following chapters. Chapter 4 specifies the hardware on which the benchmarks are executed. Additionally, the chapter introduces the used datasets, benchmarks, and competitive DBMS, which were used to evaluate the approaches. The simulator SIMORA is presented in chapter 6. This simulator is the foundation for all network-related evaluations. Next, DC Multicast is introduced in chapter 7. This approach can reduce network traffic further by using the topology knowledge of the routing protocol. Finally, a benchmark scenario is introduced in chapter 8. This benchmark includes simulating sensor devices to improve the evaluation quality in a distributed context.

Chapter 2

Fundamentals

This chapter explains in detail many basic definitions that will reappear in later chapters. These definitions belong to three fields: The SW with its ontologies and flexible data structures, the IoT with its heterogeneous devices and networks, and SW DBMS, which can store SW data.

2.1 Semantic Web

The worldwide connection of computers, known as the Internet, allows sharing of information globally within a short amount of time. Initially, the Internet was only usable by people who understood the technical foundations of providing and using resources. Over time, Internet use is becoming easier and more accessible, so even children can use it to exchange information. However, the same development, which made it easier for people to use the Internet, makes it more complicated for machines to understand and interact with the resources. Even though the links only connect one resource with another, humans can understand how the resources belong to each other. This understanding is possible because both humans share some background knowledge of how things work in the real world.

The SW is a concept in which information on the Internet is given meaning, making it more easily understandable and usable by computers and people [13]. The SW is built on the idea that the meaning of the information should be explicit and machine-readable rather than relying on human interpretation. This metadata allows computers to process and understand information more intelligently, creating new and innovative applications and services.

To achieve this, the SW needs a standardized way of expressing this metadata, which makes automated resource processing possible [13]. The basic

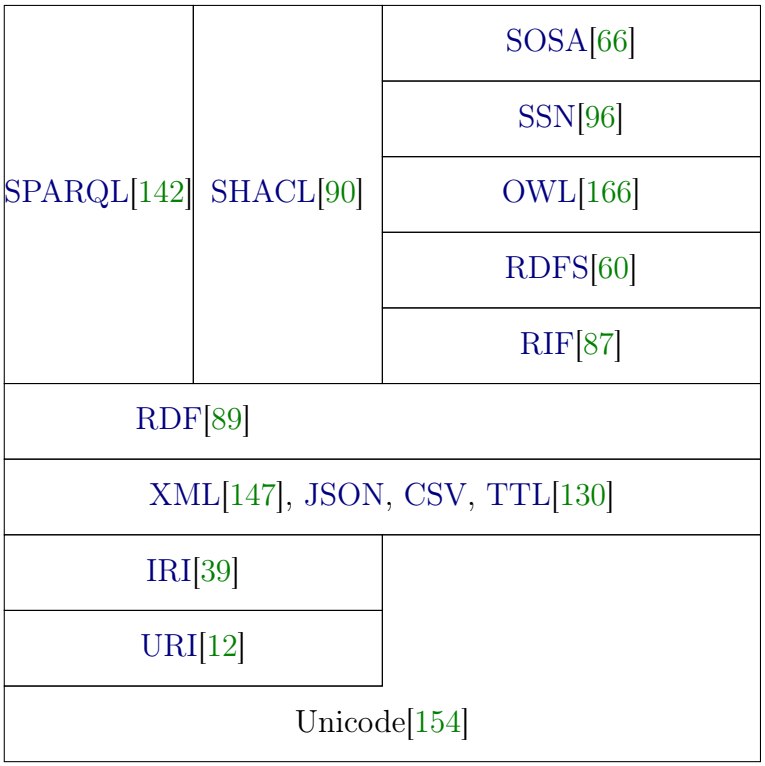


Figure 2.1: Layer Cake of SW

idea of the SW described by Tim Berners-Lee was to enable machines to process data in a context-aware way. The intention is to produce more context-aware answers [13]. To still allow everyone to benefit from this technology, the usage of SW is hidden from the user. To enable the widespread use of this technology, the World Wide Web Consortium (W3C) published standards for data formats and exchange protocols in the World Wide Web (WWW). The W3C continuously updates the standards and protocols, which implies that the specifications still have yet to reach their final state.

The SW is built on a set of technologies introduced in the following sections.

2.1.1 Layer Cake of Semantic Web

Figure 2.1 shows the overview of several SW technologies and their related standards. The bottommost layer of the SW is data storage and exchange. Since the existing resources on the Internet use a string-based encoding to encode their links and information, the SW also uses this to ensure compatibility with other resources. Since the universal character set (UCS) encoding

already considers different language sets, it is used for storage and exchangeability purposes. This string encoding is already standardized and used in many applications. For example, a Uniform Resource Locator (URL) can reference a specific web page on the Internet. For compatibility considerations, encoding an Internationalized Resource Identifier (IRI) matches an URL's encoding. This IRI can then define web pages and uniquely define resources.

The SW allows the information to be encoded in the Extensible Markup Language (XML) format to embed SW-related information into existing web pages. Besides the XML format, the data can be encoded in various text-based file formats, including JavaScript Object Notation (JSON) and comma-separated values (CSV). Additionally, the data can be encoded in the Turtle file format, which drops the compatibility, but enables a higher information density. This layer should ensure that the information is readable by humans and machines simultaneously.

The last layer in the context of the data description is the RDF layer. This layer finally allows the declaration of connections and statements, which help the machine understand the context.

On top of this coherent data layer, different functionalities can be provided. SPARQL is defined to access the data. This language defines multiple operators which enable Create, Read, Update and Delete (CRUD) operations. Since the purpose of the SW is to consider contextual knowledge, there are also RDF Schema (RDFS) and Web Ontology Language (OWL). These two provide a framework to express class hierarchies and ontologies. The rules can be exchanged with the Rule Interchange Format (RIF) layer, allowing the knowledge processor to infer additional knowledge not explicitly mentioned in the data. On top of this framework stack, the two ontologies Semantic Sensor Network (SSN) and Sensor, Observation, Sample, and Actuator (SOSA) are shown because these are used later in this thesis.

2.1.2 UCS

The universal character set (UCS) standard allows consistent encoding and representation of text. Due to this consistency, the handling of text within applications is simplified. The long-term goal of UCS is the collection of all meaningful letters from all over the world in any text form. This standard avoids the need for any local encoding scheme, which could lead to false representations of letters. The first version of UCS was released in 1991 [154]. The collection contained letters and signs which belonged to one or multiple languages. The current version of UCS *15.0.0* from 2022 contains 149186 characters with code points that cover 161 modern and historical scripts and

multiple symbol sets. The UCS standard is organized in planes. The first four bits of the data refer to the plane number, and the others refer to the character within the given plane. The first plane is the Basic Multilingual Plane (BMP). This plane covers most of the frequently used symbols. The other planes are called supplementary planes. As of version *15.0.0*, five plains contain symbols, and seven are named. The limitation to 16 planes comes from UTF-16, which limits the number of possible characters to 2^{20}. UTF-8 would allow the encoding of 2^{21} symbols with its current encoding, which equals 32 planes.

The last two planes are open for private use, which allows users to add their symbols at the price of incompatibility.

2.1.3 IRI, URL, and URI

The Internationalized Resource Identifier (IRI) uniquely identifies a resource by a given chain of characters and symbols [12]. The IRI originates from the Uniform Resource Identifier (URI) used in the WWW as a Uniform Resource Locator (URL). These URLs are also called web addresses. An URL refers to an existing location, while a URIs does not need to refer an existing resource. An URI follows a fixed scheme, as shown in figure 2.2. The difference between a URI and an IRI is the allowed character set. While an IRI can contain any character, a URI allows only American Standard Code for Information Interchange (ASCII) characters. An IRI can be transformed into a URI using the percent-encoding, which replaces non-ASCII chars with their hexadecimal representation, always indicated with a previous percent sign. Figure 2.3 shows an example URI which uses all the components. In the example, *https* specifies the resource scheme. By using the scheme, the client-side application can make assumptions about the content of the transmission. The next part is the username *benjamin*. The username can be used to verify that the requesting user has permission to view the resource. The third component specifies the server, which has the resource. In this example, it is *www.uni-luebeck.de*. The port must be specified if non-standard ports are used. When using standard ports, the specification is optional. With the port, it is possible to specify which application on the destination server should receive the query. The remaining components of the URI are interpreted within the application. The "path"-component in the example, *abc*, determines the resource within the application. Optional query parameters like *a=1* allow the generation of individual resources based on different parameters. Finally, the fragment component *#1* can be used to specify the position in the document. This component could refer to, e.g., captions in the document.

```
scheme ':' ['//' [userinfo '@'] host [':' port]] path
   ↪ ['?' query] ['#' fragment]
```

Figure 2.2: URI syntax

```
https://benjamin@www.uni-luebeck.de:1234/abc?a=1#1
```

Figure 2.3: URI example

The scheme part always specifies how to use the given resource. The optional *authority* part consisting of *userinfo*, *host*, and *port* describes the computer on which the resource can be found. The mandatory *path* segment specifies the name of the physical or logical resource on the previously given machine. Finally, the optional *query* and *fragment* segments can pass parameters to access a logical resource.

An example of IRI is given in figure 2.4. In this case, the IRI uniquely identifies a parking area. Even if this string matches the URL scheme, it does not refer to an existing web page.

2.1.4 XML

With standardized text encoding and a common way of referencing resources, the next step is a syntactical arrangement. The Extensible Markup Language (XML) layer provides a machine and human readable representation [147]. As the name suggests, the markup language distinguishes between keywords, annotations, and text. The design goals of XML emphasize simplicity, generality, and usability across the Internet. Although the design focused on documents, the format can express arbitrary data structures. The format is one of the most important formats used on the Internet because it is a foundation for Hypertext Markup Languge (HTML) to display the website content to users. Due to the flexibility, the XML used to display a website can be enhanced with additional invisible information, which the SW can use to provide further information to the machine.

```
https://parking#area
```

Figure 2.4: IRI example

2.1.5 RDF

Resource Description Framework (RDF) is a directed, labeled graph data format representing Web information [89]. It intends to connect resources in the WWW semantically. This layer is placed above the flexible XML encoding. A fact is always represented by a triple consisting of subject, predicate, and object. The triple can be interpreted as a sentence that describes the subject. The formal definition of a *Triple* is $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where I represents the set of IRIs, B the set of *blank nodes*, and L the set of *Literals*. Figure 2.5 shows the graphical representation of some RDF data. Since IRI was explained earlier, only the *blank nodes* and *Literals* are examined further.

Blank nodes are not globally unique but only within their document. Therefore these can be used to identify something which does not have or does not need a global identifier. When combining multiple documents, the *blank nodes* need to be renamed because they refer to different things no matter what they are called.

Literals represent values like strings, numbers, and dates. The standard allows the attachment of an explicit datatype or language to the string representation. However, both datatype and language tags are optional. The specification of a datatype allows the application to interpret the data meaningfully. Literals with a given datatype are also called typed literals. Similarly, literals with a given language tag are called language-tagged strings. Language-tagged strings can be implicitly assumed to be strings by any application. Data formats like Terse RDF Triple Language (TTL) do not specify a language-tagged string's datatype. When neither datatype nor language tags are given, the literals are called simple literals. In this case, most applications assume the string datatype. Literals can only occur in the object position of a triple.

Subjects represents specific resources that need to be described. When an subject is globally unique, an IRI directly references the resource. Otherwise, a *blank node* can be used as a placeholder to reference the resource. Every node in the graphical representation with an outgoing edge is an subject in at least one triple.

Objects can be any value type. When the type is an IRI or a *blank node*, then the object might be the subject of another triple, in which case the triple represents a connection between two things. If the object is a literal, it must be an attribute of the given subject. In the graphical representation, all nodes with incoming edges are at least once in the object position of a triple.

Predicates must be IRIs. They describe how the subjectd and objects are

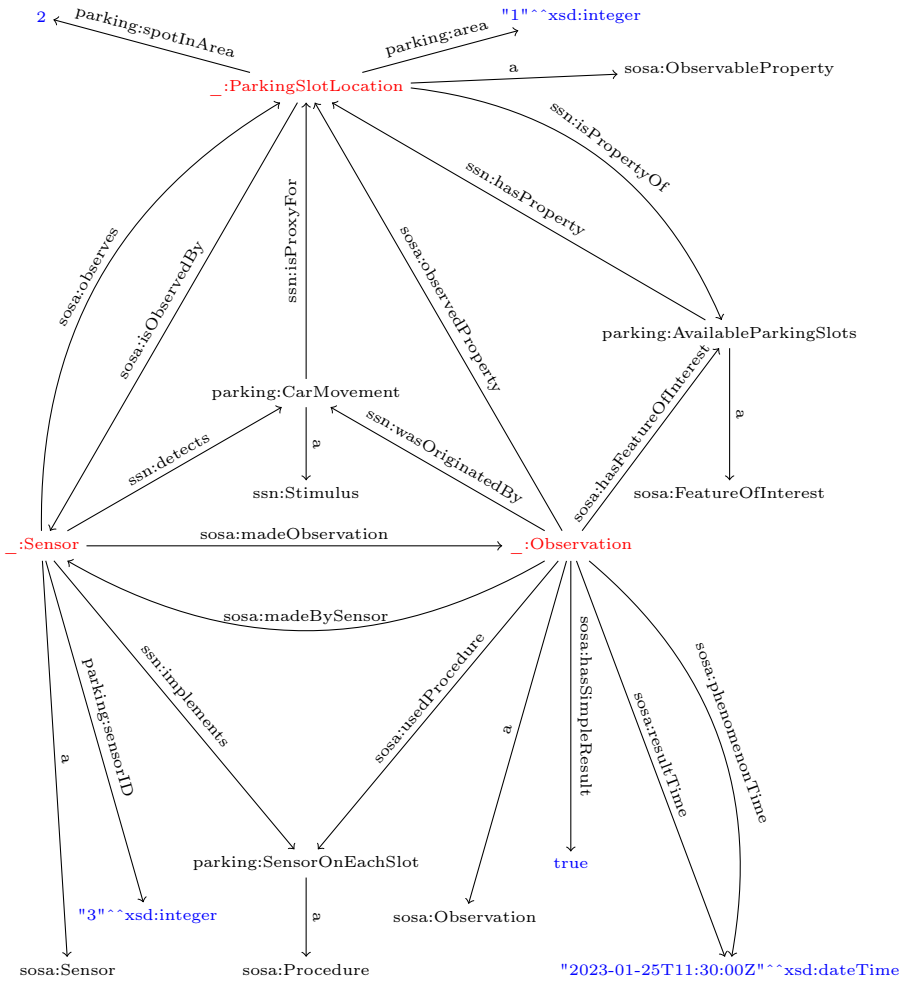


Figure 2.5: Graphical representation of RDF data. The blue nodes are Literals, and the red nodes are *blank nodes*. The black texts represent IRIs.

```
@prefix parking: <https://parking#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .
@prefix ssn: <http://www.w3.org/ns/ssn/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

parking:AvailableParkingSlots a sosa:FeatureOfInterest ;
ssn:hasProperty _:ParkingSlotLocation .
parking:CarMovement a ssn:Stimulus ;
ssn:isProxyFor _:ParkingSlotLocation .
parking:SensorOnEachSlot a sosa:Procedure .
_:ParkingSlotLocation a sosa:ObservableProperty ;
parking:area \"${area}\"^^xsd:integer ;
parking:spotInArea \"${spotInArea}\"^^xsd:integer ;
sosa:isObservedBy _:Sensor ;
ssn:isPropertyOf parking:AvailableParkingSlots .
_:Sensor a sosa:Sensor ;
parking:sensorID \"${sensorID}\"^^xsd:integer ;
sosa:observes _:ParkingSlotLocation ;
ssn:detects parking:CarMovement ;
ssn:implements parking:SensorOnEachSlot .

_:Sensor sosa:madeObservation _:Observation .
_:Observation a sosa:Observation ;
sosa:hasFeatureOfInterest parking:AvailableParkingSlots ;
sosa:hasSimpleResult \"${isOccupied}\"^^xsd:boolean ;
sosa:madeBySensor _:Sensor ;
sosa:observedProperty _:ParkingSlotLocation ;
sosa:phenomenonTime \"${sampleTime}\"^^xsd:dateTime ;
sosa:resultTime \"${sampleTime}\"^^xsd:dateTime ;
sosa:usedProcedure parking:SensorOnEachSlot ;
ssn:wasOriginatedBy parking:CarMovement .
```

Figure 2.6: Example TTL file.

connected. The graphical representation labels every connection between two nodes with a predicate.

2.1.6 TTL

Terse RDF Triple Language (TTL) is a file format used to represent RDF data [130]. It is a terse RDF syntax designed for readability and simplicity, often used to represent simple, small RDF graphs. The format is designed to be more concise and accessible to read than other RDF syntaxes, such as XML.

Figure 2.6 shows an example TTL file. In the first couple of lines, the frequently used prefixes are defined. The prefixes shorten the following entries and improve the TTL file's overall readability. Afterward, the data is defined. Only one triple is defined in each line to increase the readability. However, this is not required. Each triple consists of subject, predicate, and object. If multiple consecutive triples share the same subject, then the triples are

separated by a semicolon.

In the given example, the first block of triples contains static data, which is initialized once per sensor. The second block contains the data, which is generated, whenever the sensor reads a new sample.

TTL files use a simple and readable syntax that allows developers to express RDF data in a way that is easy to understand, making it a popular choice for representing data in linked data and SW applications. Consequently, TTL files can be used to exchange data between different DBMS. When using TTL files for data exchange, the data in the original DBMS is first exported to a TTL file, which can then be imported into the target DBMS. This process can be done using various RDF-related libraries and tools. Whenever a DBMS does not natively support RDF data, a middleware or plugin may be needed to import the data.

2.1.7 SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) is a query language and protocol for querying and manipulating RDF data [142]. It is similar to Structured Query Language (SQL) because it allows users to retrieve and manipulate data stored in an Relational DBMS (RDBMS). Still, it is designed specifically for querying RDF data. It can express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. The language contains capabilities for querying required and optional graph patterns, conjunctions, and disjunctions. SPARQL also supports aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs. Due to the graph structure of the underlying RDF data, the join operator is one of the most frequently used operators. This behavior is different in SQL because the data originates from tables that do not exist in SPARQL.

The basic structure of a SPARQL query is composed of a SELECT statement, which defines the variables that will be returned in the query results, followed by a WHERE clause. The WHERE clause explains the conditions the data must meet to be included in the query results. These conditions include a set of triple patterns defining the RDF data. Figure A.7 shows an example SPARQL query consisting of multiple elements.

SPARQL Prefix

In SPARQL, a prefix is a shorthand notation for a longer IRI that refers to a specific namespace in an RDF dataset. A namespace is a collection of

related resources identified by a shared prefix. The prefix notation allows using short and readable names for IRIs in SPARQL queries. For example, instead of using the full IRI *http://www.w3.org/ns/sosa/resultTime* to refer to the *resultTime* property in the SOSA vocabulary, the prefix *sosa* and the local name *resultTime* can be used in the query, like this: *sosa:resultTime*. A prefix in a SPARQL query needs to be defined using the PREFIX keyword, followed by the prefix and the IRI it represents. For example, the *sosa* prefix is defined in the first line of figure A.7. Multiple prefixes can be defined in a single query and used in any part of the query, including the SELECT- and WHERE-clauses and triple patterns. It is worth noting that the prefix notation is only a shorthand notation. The entire IRI is used internally by the SPARQL engine when it processes the query.

SPARQL Join

A join operator matches and combines data from different parts of an RDF dataset based on shared variables. The result of a join is a new set of variable bindings that combines the information from the matched triples.

A join operation is typically used in the WHERE clause of a SPARQL query, where multiple triple patterns are specified. The variables in the different triple patterns are matched based on the variable name, and the results are combined to form the final query results.

For example, suppose there are two triple patterns in the query, one that matches triples with the subject variable *?Observation* and the predicate *sosa:resultTime*, and another that matches triples with the subject variable *?Observation* and the predicate *sosa:hasSimpleResult*. In that case, the SPARQL engine will match the subject variable in both patterns and combine the results to form a new set of variable bindings, including the *resultTime* and the *simpleResult* of the matched observation.

A join operator can also be defined with more than two triple patterns. Additionally, as shown in figure A.7, subqueries can be part of the join.

SPARQL Group-by

In SPARQL, the GROUP BY clause groups the query results by one or more variables. The GROUP BY clause can be used with aggregate functions, such as COUNT, SUM, AVG, MIN, and MAX, to calculate group summary statistics.

The basic syntax for using the GROUP BY clause is to specify one or more variables after the keyword GROUP BY, separated by commas. In figure A.7, only one variable is specified.

When a query includes a GROUP-BY-clause, the SPARQL engine will first group the query results by the specified variables and then apply the aggregate functions to each group. In this example, the query groups the results by the *?ParkingSlotLocation* variable and calculates when the last information was received.

It is worth noting that a HAVING clause can filter the result of the GROUP BY clause.

SPARQL Filter

In SPARQL, the FILTER keyword is used to restrict the results of a query based on a Boolean expression. The FILTER keyword is used in the WHERE clause of a query to specify a condition that the results must meet to be included in the final query results. For example, the filter can be used whenever a simple equality test in a triple pattern is insufficient.

The basic syntax for using the FILTER keyword is to specify a Boolean expression after the keyword FILTER. The Boolean expression can include variables, literals, and built-in functions. For example, in figure A.8, the filter expression includes only certain parking areas in the result set.

The FILTER keyword is a powerful tool to refine a query's results and retrieve only the data that meets specific conditions. It also can help improve the query's performance by reducing the number of possible solutions before the join operation.

2.1.8 RDFS

RDF Schema (RDFS) is a set of vocabulary and rules describing the structure and constraints of RDF data [60]. RDFS provides a way to define classes, properties, and constraints that can be used to create a formal schema for an RDF dataset.

RDFS defines a set of classes, including *rdfs:Class*, *rdfs:Resource*, and *rdfs:Literal*, and a bunch of properties, including *rdfs:domain*, *rdfs:range*,

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix parking: <https://parking#> .

parking:Observation rdfs:subClassOf rdfs:Resource .
parking:Sensor rdfs:subClassOf rdfs:Resource .
parking:ObservableProperty rdfs:Resource .

parking:sensorID rdfs:subPropertyOf rdfs:label .
parking:area rdfs:subPropertyOf rdfs:seeAlso .

parking:sensorID rdfs:domain parking:Sensor .
parking:area rdfs:range xsd:integer .
```

Figure 2.7: Example RDFS file.

rdfs:subClassOf, and *rdfs:subPropertyOf*, that can be used to create a hierarchical structure for classes and properties in an RDF dataset.

Likewise the *rdfs:subClassOf* property is used to specify that one class is a subclass of another. Finally, the *rdfs:subPropertyOf* property is used to specify that one property is a sub-property of another. These properties define a hierarchy of classes and properties, making it possible to express inheritance relationships between them.

The *rdfs:domain* and *rdfs:range* properties specify the domain and range of properties, respectively. The domain of a property is the class of resources that can be used as the subject of triples that use the property, and the range of a property is the class of resources that can be used as the object of triples that use the property.

RDFS also provides a way to specify constraints on using properties, such as cardinality constraints (e.g., minCardinality, maxCardinality) and type constraints (e.g., *rdfs:range*), which can be used to ensure that data is consistent and conforms to a specific schema.

RDFS is widely used in the RDF community, and many vocabularies and ontologies are defined using RDFS. This acceptance allows for greater interoperability between different RDF datasets and makes it easier to create and use powerful, reusable vocabularies for describing data on the web.

Figure 2.7 shows how RDFS can be used to specify components of the parking scenario. In this example, the *parking:Observation*, *parking:Sensor*, and *parking:ObservableProperty* are the RDFS class *rdfs:Resource* subclasses. *parking:sensorID* and *parking:area* are sub-properties of the RDFS properties *rdfs:label* and *rdfs:seeAlso*, respectively. Then the *rdfs:domain* and *rdfs:range* properties specify that the *parking:sensorID* property has a domain of *parking:Sensor*. The range of the *parking:area* property consists of all *xsd:integer*.

2.1.9 OWL

Web Ontology Language (OWL) is a language for representing ontologies on the WWW. Ontologies are formal representations of a set of concepts and relationships that exist in a particular domain of knowledge. They provide a way to define the vocabulary and structure of a specific domain and to express constraints and inferences that can be made from that vocabulary [166].

OWL has three different levels of expressiveness: OWL Lite, OWL DL, and OWL Full [166]. OWL Lite is the most basic level, providing a small set of constructs for defining classes, properties, and individuals. OWL DL and OWL Full share the same vocabulary, an extension of the vocabulary of OWL Lite. OWL DL is a more expressive level, providing a more extensive set of constructs and allowing for more complex reasoning. However, OWL DL enforces some restrictions, making distinguishing between things that appear in different contexts easier. Finally, OWL Full is the most expressive level, allowing for using constructs incompatible with reasoning.

OWL provides a rich set of constructs for defining classes, properties, individuals, and constraints, including:

- Class and property hierarchies using *rdfs:subClassOf* and *rdfs:subPropertyOf*
- Property characteristics such as functional, inverse functional, symmetric, transitive, and inverse properties
- Cardinality and existence restrictions on properties
- Logical operators such as intersection, union, and complement
- Built-in predefined classes and properties, such as *owl:Thing* and *owl:sameAs*

OWL also supports reasoning, allowing inferences to be made from ontology. Reasoning means that if the ontology implies a statement but is not explicitly stated, a reasoner can infer that statement. This inference makes it possible to automatically discover new information and relationships in the ontology and check for consistency and completeness [166].

OWL ontologies allow for greater interoperability and reuse of vocabularies across different applications and domains, enabling automated reasoning over the represented knowledge. This interoperability makes it a valuable tool for many applications, such as knowledge management, natural language processing, semantic search, and the SW.

OWL is based on RDF. Therefore, it is designed to be used alongside other W3C standards, such as RDF and RDFS. This design allows for easy integration of OWL ontologies with other RDF data and effortless linking of ontologies to other resources on the web. Furthermore, OWL ontologies can be represented in different serialization formats, such as RDF/XML [48], Turtle [25], and OWL/XML [140]. These representations make exchanging and sharing ontologies across other systems and platforms easy.

OWL 2 is fully backwards compatible to OWL [1]. Some new features are only syntactic sugar (e.g., disjoint union of classes), while others add new functionality such as asymmetric, reflexive, and disjoint properties.

2.1.10 SOSA

Sensor, Observation, Sample, and Actuator (SOSA) is a lightweight ontology representing sensor observations, measurements, and related information in RDF format [96, 66]. It is intended to provide a shared vocabulary for describing sensor observations and measurements in a machine-readable and human readable way.

SOSA provides a set of classes and properties for describing sensor observations and measurements, including:

- *sosa:Observation*, a class representing an observation made by a sensor
- *sosa:Result*, a class representing the result of an observation
- *sosa:hasResult*, a property linking observation to its result
- *sosa:madeBySensor*, a property linking observation to the sensor that made it
- *sosa:hasFeatureOfInterest*, a property linking observation to the feature of interest that it observed

SOSA aligns with other ontologies and vocabularies, such as SSN, which enables integration and linking sensor data with other domain-specific data and knowledge, making it helpful in building intelligent and interoperable sensor systems.

SOSA also allows for querying and reasoning over sensor data using SPARQL. This reasoning allows for discovering and understanding patterns and relationships in the sensor data and automatically generating new sensor data based on the existing data and ontological constraints.

```
PREFIX parking: <https://parking#> .
PREFIX sosa: <http://www.w3.org/ns/sosa/> .
PREFIX ssn: <http://www.w3.org/ns/ssn/> .
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> .

_:Observation a sh:NodeShape ;
 sh:targetClass sosa:Observation ;
 sh:property [
  sh:path sosa:madeBySensor ;
  sh:minCount 1 ;
  sh:maxCount 1 ;
  sh:datatype _:Sensor .
 ] .
_:Sensor a sh:NodeShape ;
 sh:targetClass sosa:Sensor ;
 sh:property [
  sh:path sosa:madeObservation ;
  sh:datatype _:Observation .
 ] .
```

Figure 2.8: Example SHACL file.

2.1.11 SHACL

Shapes Constraint Language (SHACL) defines a language, which can validate RDF graphs against conditions [90]. These conditions are provided as shapes; other constructs are expressed as an RDF graph. RDF graphs used in this manner are called shapes-graphs in SHACL, and the RDF graphs validated against a shapes graph are called data graphs. As SHACL shape graphs validate that data graphs satisfy a set of conditions, they can also be viewed as a description of the data graphs that meet these conditions. Such descriptions may be used for various purposes besides validation, including user interface building, code generation, and data integration.

Figure 2.8 defines two shapes. First, it defines an observation, and afterward, a sensor that can make such an observation. Then, the SHACL defines how those objects are related. Additionally, it can specify constraints such as the observation having to select exactly one sensor that made the observation.

2.1.12 RIF

The Rule Interchange Format (RIF) standard is maintained by the W3C, an international organization that develops and maintains open standards for the web. RIF aims to provide a standard interchange format for rules, making sharing and reusing rules across different applications and domains easier. It is mainly used to exchange rules among rule systems, particularly Web rule engines [87]. RIF focused on exchange rather than trying to develop a single one-fits-all rule language. The reason is that in contrast to other SW standards, such as RDF, OWL, and SPARQL, it was immediately apparent that a single language would not satisfy the needs of many popular paradigms for using rules in knowledge representation and business modeling.

Nevertheless, even rule exchange alone was recognized as a complex task. There are three categories of available rule systems: *first-order*, *action rules*, and *logic-programming*. The syntax and semantics of these systems only share little fragments. Moreover, systems have significant differences, even within the same paradigm.

The Working Group designed a family of dialects with rigorously specified syntax and semantics to obtain a simple-to-use notation. The family of RIF dialects is intended to be uniform and extensible. Therefore, dialects are expected to share the syntactic and semantic apparatus as much as possible. Extensibility here means it should be possible to define a new dialect, which extends an existing dialect by adding the desired functionality. When defined, these new RIF dialects would be non-standard but might eventually become standards.

RIF is more than just a format specification. However, the format concept is essential to how RIF is intended to be used. The medium of exchange between different rule systems is XML, a format for data exchange. Central to the idea behind rule exchange through RIF is that different systems will provide syntactic mappings from their native languages to RIF dialects and back. These mappings are required to be semantics-preserving. Thus, rule sets can be communicated from one system to another, provided the systems can talk through a suitable dialect they both support.

The example in figure 2.9 defines that an observation is made by a specific sensor, if that sensor made that observation. A modus ponens argument can logically derive this fact. The rule system can use this definition to infer additional knowledge.

```
Document(
  Prefix(cpt <http://example.com/concepts#>)
  Prefix(sosa <http://www.w3.org/ns/sosa/>)
  Group
  (
    Forall ?Sensor ?Observation (
        cpt:a1(?Sensor sosa:madeObservation ?
           Observation)
         :- cpt:a2(?Observation sosa:madeBySensor ?
            Sensor)
    )
  )
)
```

Figure 2.9: Example RIF file using RIF-Core.

2.2 Internet of Things

The graph in figure 2.10 shows that the speed of a single processor core has mostly stayed the same in the last 20 years. Due to physical limitations, the speed of a single core cannot be increased significantly. However, thanks to the ongoing hardware miniaturization, the space freed up can be used for new computing cores. Therefore, software must follow the new paradigm to efficiently use hardware capacities and satisfy the increasing demand for higher data processing performance. At the same time, the permanent storage available on the hardware side is also increasing. The additional storage can support the massive parallelization demanded by providing multiple variants of the original data. This change creates new challenges and opportunities for parallel and distributed processing. The increasing availability of low-cost, high-performance IoT devices and advances in technologies such as 5G and edge computing is contributing to the growth of the IoT [153]. IoT refers to the growing network of physical objects connected to the Internet that can collect and exchange data. These objects, known as smart devices, can include many devices such as smartphones, home appliances, vehicles, industrial machines, and more. The IoT allows these devices to communicate and share data with other systems, such as cloud-based servers, through the Internet. This communication enables new and innovative applications and services, such as remote monitoring, control and automation, real-time data analysis, and predictive maintenance, to name a few [28].

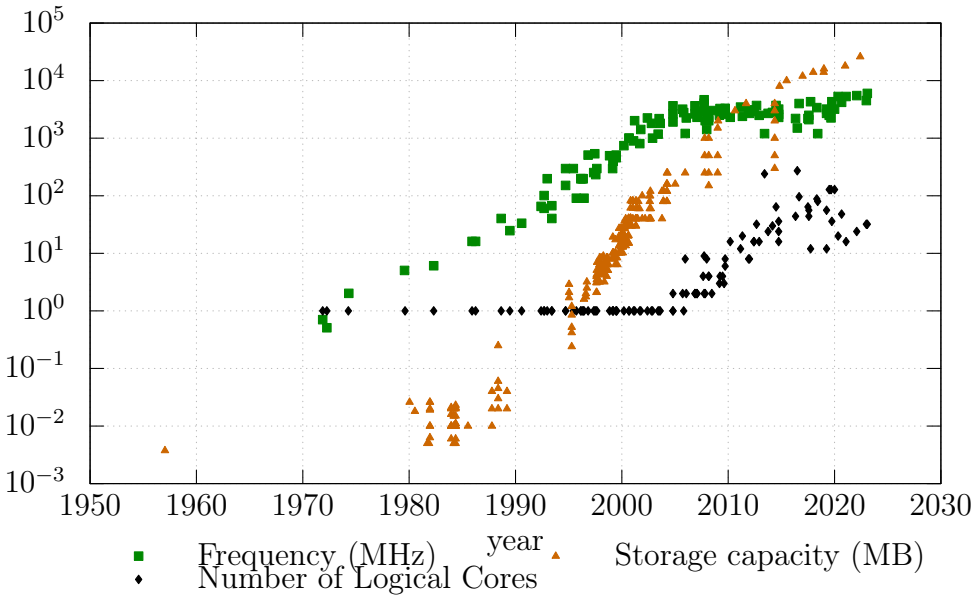


Figure 2.10: Central Processing Unit (CPU) performance and storage size over the last 70 years. CPU data modified from GitHub [135]. Storage data modified from Wikipedia [68]. This figure is incomplete and contains only data which is available to the public.

Some specific areas of growth in IoT hardware are:

- The increasing adoption of IoT sensors and actuators in industrial and manufacturing applications
- The rising demand for smart home devices and appliances
- The increasing adoption of IoT devices and solutions in the healthcare industry
- The growing demand for connected cars and other connected transportation systems.

Overall, IoT hardware prognoses suggest that the IoT market is expected to continue growing in the coming years, driven by the increasing adoption of IoT devices and solutions across various industries and the availability of advanced technologies.

The difficulty is that the hardware often offers little computing power. Especially in factories, many small devices can be networked together. Many of these devices have sensors that constantly provide new measured values.

It is possible to store all data in a central computing cluster within a factory. It is estimated that between 24 [38] and 100 [86] billion things will be connected to the IoT. So in the context of the IoT, the network is a crucial issue. Wireless Sensor Networks (WSNs) are networks that consist of many interconnected sensing devices. The devices are sometimes far apart in the context of weather or other environmental measurements. Therefore, most devices are connected to another machine rather than directly to the Internet. For battery-powered devices, energy is always a scarce resource. The available energy limits a device's lifetime and computing power. Transmitting data is one of the largest energy consumers in WSN [41]. Several approaches exist to optimize the routing protocols to reduce energy consumption during transmission [29]. However, sending less data is even better. Therefore, the amount of data to be transmitted must be reduced as much as possible. The topology of the devices must be considered to reduce the amount of data because it costs more energy to send a small piece of data over a long distance than a large amount of data over a short distance [41].

Since sensors can produce a large amount of data, data storage, and processing is an important research area. While current sensor data is sufficient for simple queries, complex analyses must also consider historical data. One possibility is to use cloud-based approaches. In this case, the sensors send the measured data directly to a cloud. In this scenario, the cloud can provide a large amount of computing power for the application. However, this leads to enormous latencies for trivial queries, as the data has to be sent over a long distance from the source to the cloud before any analysis can be performed. In the IoT domain, the network structure is also complex. Since the data is highly distributed, it is natural to store it in a distributed manner. Here, it is essential to distribute the data sensibly so that the queries can compute a solution as quickly and efficiently as possible. This distribution has given rise to edge computing, where medium-sized devices can be integrated into the network [24]. The amount of small devices, in turn, increases the effort to find the desired data.

2.3 Semantic Internet of Things

The combination of SW and IoT is called SIoT. It combines the machine-interpretable data semantics with many sensor devices of the IoT. The primary advantage is that devices of different manufacturers can be used together. All devices which support semantic data can benefit from the additional information. This unified representation allows for splitting responsibilities across multiple applications, such that a dedicated and optimized DBMS can focus on data storage and processing. The disadvantage is that the semantic enhancement requires extra bytes to be encoded, which increases the data volume to be sent around the network.

```
00 01 02 03
04 05 06 07
08 09 0a 0b
```

Figure 2.11: Binary sensor data.

```
<sensor4> <measurement> _:measure .
_:measure <temperature> 23 .
_:measure <time> "01.01.2023 08:05"^^<xsd:datetime> .
```

Figure 2.12: Sensor data encoded in TTL.

Figures 2.11 and 2.12 present the different encoding schemes. Both figures represent the same data. While manufacturers may choose another binary encoding, which others can not use, many applications can immediately use the SW version. A few applications may still need some conversation because of multiple possible ontologies, but the compatibility is improved overall.

2.4 Semantic Web DBMS

The SW is a concept in which information on the Internet is given meaning, making it more easier to understand and usable by computers and people. An SW DBMS can store and manage data in a format that computers and people can easily understand. According to a survey paper on query languages in the SW [10], SPARQL [142] is the primary RDF query language.

In a heterogeneous network, different devices are connected, often with varying levels of compatibility. An SW DBMS can help bridge the gap be-

tween these different systems by providing a shared data storage and management format. The DBMS allows data to be shared and used across different systems, regardless of their underlying technology.

Overall, SW DBMSs play a critical role in enabling interoperability and data sharing in heterogeneous networks, thus allowing the seamless integration of different devices. There is a strong relationship between the SPARQL query optimization and the indexing strategies used in the underlying triple store.

DBMS on heterogeneous hardware is already being explored. Hardware features like the CPU and the graphics processing unit (GPU) are considered in most research [22]. Heterogeneous distributed memory and data storage is another research focus [168]. Research also combines both heterogeneous processing and heterogeneous storage [104]. In addition to hardware, research is being done on combining different DBMSs [128].

2.4.1 Dictionary

Many DBMSs [115, 26, 16, 30] use dictionaries for their internal data representation. However, some DBMSs [148] use those dictionaries only for data types of varying length or when the datatype consumes much space, especially Binary Large Objects (BLOBs) and strings.

There are several names for the dictionary. E.g., in Apache Jena, this is called *the node table* [45], while PostgreSQL calls this *Large Object table* [58]. *Virtuoso* calls them dictionaries [150].

These bidirectional dictionaries store a mapping between a string and an integer representation. These integers can then be compactly stored in the indices of the DBMS. The DBMS's use of dictionaries can enormously reduce the required storage space, especially when using several indices.

Due to the massive volume of data in the most extensive datasets, 4-byte integers might not always have enough bits to encode the various values in a dataset. Therefore some DBMSs use ids with eight or more Bytes [26]. However, due to the extra bytes, it is also possible to encode some of the smaller, frequently used values directly inside this id representation [26]. This is done by defining one of the bits to be a flag. Whenever the flag is set, the remaining bits of the ID directly represent the value. This encoding requires that the value fits into the remaining bits. The candidates for inline encoding are small numbers, date and time values, and *blank nodes*.

2.4.2 Indexing Schemes

The two most essential indexing strategies are *RDF3X* [113, 115] and *Hexastore* [165].

RDF3X [113, 115] uses twelve indexes. A duplicated index exists for each subject, predicate, and object permutation, namely SPO, SOP, PSO, POS, OSP, and OPS. The abbreviation SPO stands for the sort order in which the triples are sorted first by subject, then by predicate, and finally, by object. The other sort orders differ in which triple component is sorted first, second, and last. Aggregate indexes SP, SO, PO, subject, predicate, and object are also created to store only the number of entries for each key. The individual values are stored in a dictionary and assigned integer IDs to save memory. B^+-trees are used to store these IDs within the indexes. The B^+-tree can then be used to determine the starting point of the triple patterns quickly. The data can then be read sequentially. In addition, this indexing scheme offers many possibilities for partitioning. The basic idea of the RDF-3X indexing scheme is the basis for many other approaches in many different research works.

Hexastore [165] uses an index for each subject, predicate, and object subset. These subsets are then the keys of a hash map. The remaining components that do not appear in the key are the values. Consequently, the six fully replicated indexes are S-PO, P-SO, O-SP, SP-O, SO-P, and PO-S. The hyphen in the names separates the components that are part of the key from the values. Sometimes a seventh index SPO is added if the hash consists of all three triple components. The values within a map entry can be stored and sorted. The original version of *Hexastore* was only developed for a central triple store. The most significant disadvantage is that it is tough to distribute the indexing. The advantage of *Hexastore* is that a single entry in the hash map directly contains the complete solution [123, 67].

After finding the first triple, both indexes allow easy iteration over an ordered list. In addition, both triple stores support the aggressive use of merging already sorted data retrieved from their indexes.

The Kowari approach [105] is very similar to the RDF3X. However, instead of B^+-trees, they use Adelson-Velskii and Landis (AVL) trees [105, 43]. They argue that in B^+-trees, an average of 25% of space is unused. In addition, this approach uses the three collation orders SPO, POS, and OSP. Consequently, the index fits into memory longer than the *RDF3X* approach, making it faster to execute.

In contrast to the above indices, which index the complete data, another work proposes to index only a part of the graph. The *Triag* [111] index looks for triangle patterns in the graph and stores them in a separate index.

This index allows an optimizer to replace multiple consecutive joins with a single range scan in this new index structure. Especially in the context of ontologies, this can lead to a high-performance improvement.

The Double Chain-Star approach [109] has a similar idea. They create an index for queries containing two "star shapes" connected by a path. The authors claim that the candidates for this kind of indexing can be efficiently computed. The data is then stored similarly to relational tables, with an arbitrary number of named columns. All queries that select a subset of these named columns can use an index scan instead of a costly join sequence so that the speed can be increased dramatically.

TripleBit uses an entirely different storage layout [171]. They encode the triple data as an ID-Chunk bit matrix. Additional to the range scans, which are used by the other approaches, this strategy allows for further filtering the possible join candidates directly in the triple store.

2.4.3 Data Partitioning and Data Distribution

In addition to parallel SPARQL processing, which uses multiple processor cores on a device, the data can be distributed to various nodes. This distribution, of course, requires the presence of multiple nodes and a fast connection between them to ensure high performance. Both variants involve the allocation of triples to a particular computing unit. This assignment is usually done using a deterministic algorithm. The main task of these algorithms is to achieve a uniform distribution among all computing units. Care must also be taken to minimize communication overhead when distributing data across multiple nodes. In the context of IoT, data distribution must additionally consider topology. There are a variety of approaches for partitioning and distributing triple data. All of these strategies have their advantages and disadvantages. When evaluating the strategies, it is noticeable that they are often optimized only for querying rather than inserting data. This observation can be derived from the fact that these methods send a large amount of data during the partitioning phase. If, during data partitioning, the data is partitioned so that equal values of a triple component are always processed on the same processor, then join operators can be computed independently. If this succeeds, it is sufficient to send intermediate results, reducing the network load and increasing the speed [53, 3, 173, 64]. Some variants of data distribution are presented below.

All cluster-based approaches have in common that a large part of the graph data must already be known at the beginning. The strongly connected nodes are then stored together in the same partition.

The connectivity between triples is computed in the molecule cover [83]

approach, such that closely related triples, called molecules, are assigned to the same node. Apart from the large initialization overhead, many joins can be computed independently and locally at a node in this way.

Another strategy is to create a separate table for each predicate [15]. These tables store the subject and object in independent arrays. Vectors make the connection between these tables and arrays of pointers. These vectors should achieve a higher storage efficiency. In addition, it is possible to mark the pointer vectors on which devices further data is located. As a result, this approach should cause less network traffic.

Another approach stores the triples in adjacency lists [172]. The data is distributed so that related data is stored near to each other. The advantage of this encoding is that it is immediately clear where the following data is when it is read. In this encoding, joins are calculated by traversing the data along the adjacency lists. Traversing lists avoids unnecessary intermediate results. However, it prevents the data from being read sequentially from the disk.

Another widely used clustering approach is duplicating data to obtain the k-hop property. In the k-hop strategy, the triples are first distributed using an arbitrary hash function. Then, all nodes up to k edges away from any original node are also stored in the partitions. Each system uses a slight variation, changing the value of k. The largest k used is three since this already duplicates many triples. This property allows queries containing path expressions up to a length of k to be evaluated locally on each node without communication [69, 3, 51]. The main drawback of the k-hop property is that some results are computed multiple times, which must be removed to obtain a valid result.

In addition, there are many other clustering strategies [133, 104, 137, 138, 95, 168].

Since all clustering algorithms work with paths in the graph, they work best when the data does not change. Since this work has a strong IoT focus, clustering algorithms are not considered further below.

Hash-based methods do not require an initialization phase. There are various options for what is hashed and how it is included in the hash function. Many combinations have already been presented and studied in numerous publications.

- When **hashing the subject, predicate, and object individually** [3], the data is duplicated multiple times, leading to a significantly higher memory requirement. In addition, this approach makes it possible to calculate many join operators locally.

- **Hashing by subject, predicate, and object** in all permutations is a different procedure. In contrast to the previous approach, all triple components are always used in the hash function [124].

- **S-O hashing** [69] assumes that almost all joins are S-S or S-O joins.

- Similar to the previous approach, subjects and objects are considered in the algorithm. However, hashing by subject is used to select the DBMS instance, and hashing by object is then used within the DBMS instance for partitioning [172]. Two-level data partitioning combines the low communication cost of S-S joins with the fast S-O join processing.

- **Hashing by subject** [15, 156, 138] is mainly about speeding up S-S joins. The parallel calculation of joins has the advantage of calculating their respective partial results without communication. Additionally, this strategy stores closely connected data at the same DBMS instance, which may yield further optimization possibilities.

- Another approach is **hashing by predicate** [67, 79, 9, 46]. Since the predicate is often specified as a constant in the queries, this results in a single partition containing the entire result set of a triple pattern. This partition scheme avoids including all partitions in the result calculation.

In the context of centralized in-memory DBMSs, many different data partitioning strategies exist.

Merge joins have a considerable performance advantage over hash joins. Therefore, another approach [4] partitions and sorts data on demand to deploy massively parallel merge joins anywhere. Although the approach targets RDBMSs, its results also apply to DBMS on the SW. Due to the additional sorting operations at runtime, this approach requires much available memory.

In another approach [53], partitioning threads are partitioning the join operator input data. In this way, each intermediate result can be partitioned into any number of partitions at runtime, but at the cost of runtime overhead for partitioning. If the input data for merge joins comes directly from the triple store, partitions can be accessed based on the ranges in B^+-trees. Since a range is determined based on the histogram of the corresponding triple pattern, the partition sizes of the triple design to be merged may need

to be balanced. In addition to their approach, the authors evaluate the performance gain from pipeline parallelism. They found that the main drawback of operator-based parallelism is the queues between operators, which require memory on the one hand and entail thread safety due to locking on the other.

2.4.4 DBMS and the Internet of Things

There can be several advantages to moving the entire DBMS to edge devices [24]. First, latency can be reduced because the data does not have to be transferred as far as before. The edge-devices capabilities can be used more efficiently because the data can be stored where it is created. From these two benefits comes another, namely a reduction in network traffic. As a side effect, communication hotspots on the network are reduced, leaving devices with more energy to compute something else. Among the disadvantages is that the DBMS must give up its requirement of complete global knowledge to adapt to the environment.

The biggest potential problem when running a DBMS on edge devices is the intermediate results, which can increase the data volume by orders of magnitude in the worst case. The amount of traffic depends directly on the SPARQL data queries. Whether DBMSs are suitable for integration into IoT networks will be evaluated later.

Regarding the distribution of data and storage, four different types of DBMSs can be defined in figure 2.13. In addition to the regular fully functional DBMS instances, others are added that are only involved in query processing, not data storage. This change ensures that devices with little or no memory can participate in query processing. It is also intended to reduce the amount of data sent over the network by reducing the total distance. These DBMSs are called ultralight DBMS instances in the following since they require a minimum amount of storage. All the devices connected by a mesh network in the figure could run a DBMS instance and store data. Sensors too weak to run a DBMS instance are not shown in the figure. They are indirectly shown with the icon next to the device which is connected to them. For simplicity, only the device in the upper right corner is connected directly to the Internet and a permanent power source. The four use cases presented each use the same network configuration and differ only in the DBMS design.

The upper left case corresponds to a classic centralized DBMS. All sensor data must be sent to the same DBMS instance, all data is stored there, and all query processing occurs on this one device. Since the device is connected to a permanent power supply, from a network perspective, this case corresponds to the point where all data is sent to the cloud, except for the latency between

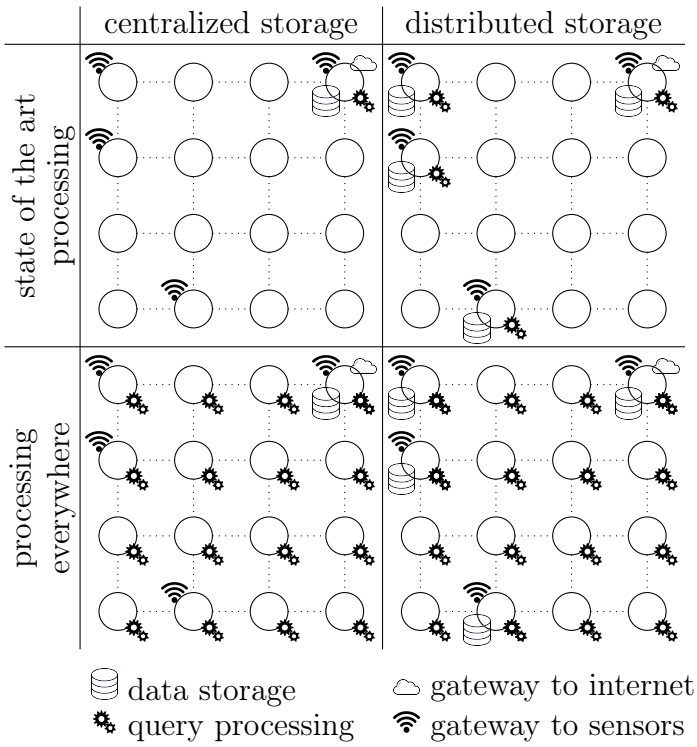


Figure 2.13: Possible combinations of distributed and centralized DBMS storage and processing

the gateway and the servers in the cloud.

The top-right case shows a classic distributed DBMS. A DBMS is created for each edge device directly connected to sensors. Depending on the data distribution model of the DBMS, it is now possible that no communication is required before the first query. However, due to query processing optimizations, it is likely, the DBMS can only store some of the data locally where it was collected. Therefore, it distributes the data to all available DBMS instances so that it can be read more quickly later.

Next, consider the third case at the bottom left of the figure. The data is initially stored centrally in a single DBMS instance, but then the data is distributed across the network again for query processing. This approach is the worst case for the amount of data transferred over the network because the data is transmitted over the network multiple times. The data is transmitted once to store the data and then again for each query. On the other hand, more available processing power can be used, which could reduce the query response time.

The last case at the bottom right of the figure combines the best of the

previous points. Data transfer during insertion can be reduced by storing data closer to the sensors. In addition, the DBMS can use a multicast algorithm to insert and retrieve data more efficiently.

2.4.5 Join-Algorithms

Since the join operator greatly influences the performance and memory requirement, several implementations exist.

Among all implementations, the merge-join is the fastest and most memory-friendly strategy because it reads the input streams only once. It is also able to discard inputs without a matching join partner immediately. However, it requires the input data to be already sorted.

A hash-join reads the data and stores it in internal hash maps. Several variations exist about which and how many input sources are stored in the hash map. If the hash-join stores multiple inputs in hash maps, one hash map is required for each input source. Due to the hash map, the storage requirement is quite high. Therefore it has no restrictions on the input order.

A third variant can be used when one join-input source is complex, and the other join source is coming from the triple store. In this variant, the complex join-input retrieves one row after the other, and for each row, the triple store is queried with the exact range to generate the output. This strategy reduces the memory requirement. However, this might be slower due to repeating new requests to the triple store.

Part I

Effect of Multiple Partitioning Schemes on Storage Requirement and Performance

Chapter 3

LUPOSDATE3000

LUPOSDATE3000 [158] is the successor of Logisch und Physikalisch Optimierte Semantic Web Datenbank-Engine (LUPOSDATE) [54]. The DBMS was designed for extensibility, making it easy to replace many components for research purposes. There are several requirements and properties of an IoT DBMS which are shown in the following sections.

3.1 Multi-Platform Capabilities

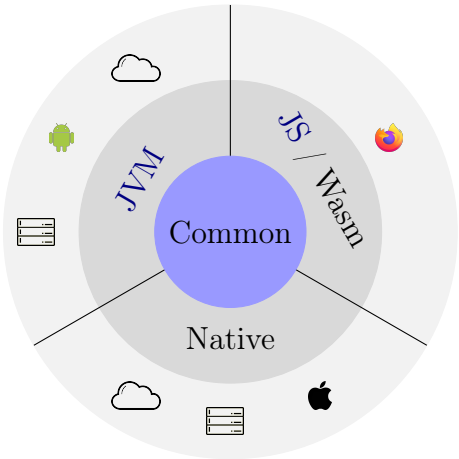


Figure 3.1: Kotlin multi-platform capabilities. Figure adapted from [44].

While the old LUPOSDATE was implemented entirely in Java, the new LUPOSDATE3000 is implemented in *Kotlin*. This language yields multiple advantages. With the focus on IoT, the DBMS must run on as many

devices as possible, as shown in figure 3.1. *Kotlin* includes a Java Virtual Machine (JVM) target that covers all devices that can run *Java*. In addition, *Kotlin* allows JavaScript (JS) so that any *Kotlin* program can run directly in browsers. Finally, *Kotlin* can output native binary code for many platforms. For each *Kotlin* target, it is possible to include existing libraries. Because of the many targets, *Kotlin* can run on desktop computers, servers, cell phones, and browsers. At the time of writing, the *Kotlin* native garbage collector is relatively slow. This issue makes *Kotlin*'s JVM target the fastest, but since the language is very new, new features and improvements are regularly added. Since the JVM target is the fastest, all benchmarks in this document use the JVM target unless otherwise noted. *Kotlin* has several modern language features, such as coroutines, inline functions, type inference, and a garbage collector for all targets, including native binaries.

3.2 Dictionary

The DBMS uses a dictionary to map the values to an internal eight-byte ID representation to reduce the required storage space. This encoding significantly reduces the amount of memory required.

In LUPOSDATE3000, there are two dictionary implementations. An in-memory dictionary has the advantage of a much higher speed for small databases because, as the name suggests, it is entirely inside the main memory. In the implementation, there is a hash-map structure, which maps the values to integer IDs. The other way around is implemented by an array, where the ID is used as an index to access the corresponding value. One disadvantage of this implementation is the memory requirement, which scales with the DBMS's data. Another disadvantage is that during the startup and shutdown of the DBMS, the whole dictionary must be converted and stored, which leads to higher startup and shutdown times.

The other implementation is based on memory pages actively swapped to disk. This swapping reduces the memory requirement but decreases the performance due to disk access. The implementation for the data to ID lookup uses tries. One example trie is shown in figure 3.2. The advantage of this data structure is its compression [151, 18]. The more words are added, the more prefixes appear and the higher the compression ratio. Since RDF heavily uses prefixes to structure its ontologies, it is likely, that there are a few very frequent prefixes in the dictionary. The other direction concatenates all the values together with a jump table, which defines which ID is stored where.

According to the specification, *blank nodes* have no fixed value. Therefore,

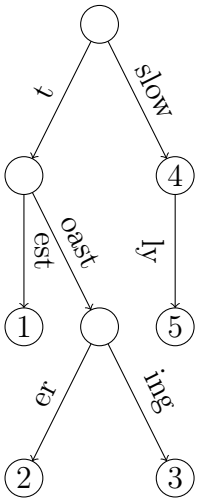


Figure 3.2: Example trie which contains the mappings test → 1, toaster → 2, toasting → 3, slow → 4 and slowly → 5

blank nodes do not need to be included in the dictionary. Instead, *blank nodes* are represented directly as IDs. The task of distributed, collision-free allocation of new *blank nodes* can be trivially solved by having each device manage its range of IDs. With some partitioning strategies, it is possible for all inserted data to be stored entirely locally and partitioned correctly in this way without the need for communication. Since most operators in a DBMS do not require the textual representation of values, these IDs are used as long as possible. Dictionary accesses are unavoidable when outputting the results. Smaller values are encoded directly in these eight bytes, so it is unnecessary to contact a remote dictionary for translation to id representation, to reduce the number of network accesses to the dictionary. This approach works for the most commonly used values, such as small integers, boolean values, and timestamps. However, it does not work for IRIs, which are required for each triple in the predicate position. Fortunately, each sensor uses a fixed algorithm to output its values. Therefore, each sensor can be translated into a small sensor-specific ontology. The collection of all these sensor ontologies is tiny compared to an entire database, so that this dictionary part can be replicated to any device in the network. These observations show that triples can be created with a minimal dictionary overhead. No distributed dictionary access is needed as long as only values from the ontology are used along with small primitive values. Some ontologies make much use of *blank nodes*. While this is good because fewer dictionary queries are required, it leads to a new problem when these *blank nodes* need to be queried to insert new data. The LUPOSDATE3000 DBMS allows internal *blank node* IDs to

be specified directly in the query to avoid a sensor constantly using insert-where statements, which would require a lot of network communication. This change allows the sensor to query these IDs once and then use them for further queries. An additional dictionary cache is created on each device with the most recently used ID values that neither contain small inline values nor are included in the sensor ontologies. The experiments resulted in low network communication due to the dictionary during DBMS initialization. More importantly, almost no dictionary access is required during subsequent use.

3.3 Pipelining

Pipeline-based programming is often implemented with iterators. DBMS results contain multiple columns (variables) and rows (results). Consequently, there are two types of iterators. Row iterators deliver the entire next row with all its variables at once. When column iterators are used, there is one iterator for each column. Multiple independent iterators allow the receiver to request the columns independently from each other. LUPOSDATE3000 uses both column and row iterators. LUPOSDATE3000 prefers column iterators. The reason is that when column iterators are used, there is no need for buffers within the receiving operators.

Additionally, when reading directly from the triple store, it is possible to skip a vast number of rows at once, such that those values are never read from the disk, which improves performance. Skipping values is possible, for example, when a merge-join or a filter operator reads directly from the storage. However, there are exceptions to the usage of column iterators. For example, the order-by-operator reads its input using row iterators because the values of a row have to stay together regardless of the ordering. The same applies to the reduce operator, which always reads the entire row to eliminate duplicates.

3.4 Indices, Partitioning, and Distribution

LUPOSDATE3000 stores the triples in multiple indices similar to *RDF3X*. However, the original *RDF3X* aggregation indexes are absent. Therefore, depending on the configuration, arbitrary collation orders are present or disabled.

Regarding data partitioning and distribution, LUPOSDATE3000 supports arbitrary combinations of triple components that can be included in the hash functions. In the experiments, the hash function only performs the modulo operator on the integer IDs. This trivial hash function already provides a uniform distribution because the dictionary assigns ascending numbers.

The indices are explained in more detail in chapter 5.

3.4.1 Compression

Inside each index, the data is ordered. The ordering of data property yields multiple features. First, duplicate elimination is trivial when the data is sorted during insertion. It is only required to look at consecutive elements. Second, the data can be compressed very well.

0xFF000001	0xFF000002	0xFF000003	triple 1
0xFF000001	0xFF000004	0xFF000005	triple 2
0xFF000002	0xFF000002	0xFF000006	triple 3
0xFF000003	0xFF000002	0xFF000007	triple 4

Figure 3.3: Index compression procedure step 1.

0xFF000001	0xFF000002	0xFF000003	triple 1
0x00000000	0x00000006	0x00000006	triple 1 $\oplus$ 2
0x00000003	0x00000006	0x00000003	triple 2 $\oplus$ 3
0x00000001	0x00000000	0x00000001	triple 3 $\oplus$ 4

Figure 3.4: Index compression procedure step 2.

Assuming each triple is stored as three 8-byte integers, the raw triple requires 24 bytes - for each active index as shown in figure 3.3. Due to the ordering, the triples next to each other are similar. Then $\oplus$ is applied to the integer representation of the previous triple and the current triple. When similar values are xored with each other, than the result has many leading zeroes. In this example, the result is shown in figure 3.4. The final step is

to count the nonzero bytes of each triple component. These counters are stored in one header byte followed by the nonzero part of the triple. Since the header byte has only 8 bits available, and there are $9^3 = 729$ possible sets of counters, an additional compression must be applied here as well. In LUPOSDATE3000, there are 2 bits for the first triple component and 3 bits for the second and third triple components. Whenever a counter is too large to fit into the desired number of bits, it is increased to 8, such that in those cases, leading zeros need to be stored as well. However, in practice, this only occurs occasionally. The difference between two triples requires about 3 bytes on average, which yields a storage reduction of factor 8. A similar encoding is used in *RDF3X* [115]. However their system only considers 4-byte integers, and subtracts values instead of applying $\oplus$.

These simple compression strategies need nearly no overhead during read and write operations. Since fewer data must be read from the disk, the speed may be increased as already shown in several research results [115, 167, 141]. Also, many operators profit from an id-based data abstraction. One of the most important operators, which profits from the representation, is the join operator because it can simply compare numbers with each other instead of complex strings, which is much faster and simpler to implement [113]. On the other hand, the dictionary yields a slower evaluation within all operators, which need to access the values directly. Those are sorting, filtering, and binding new values because these operators must convert the ids to the values they represent. This conversion is an extra step that is otherwise not necessary.

3.5 Optimizers

The overall optimization pipeline can be seen in figure 3.5. First, the query is syntactically parsed to construct a syntax tree. Then some redundant language features are striped to obtain a simple tree. Afterward, the query is optimized logically. In this phase, the ordering of the operators is changed. Constant expressions like filters and variable-bindings are evaluated. In this phase, the join order is also optimized. LUPOSDATE3000 has integrated a greedy optimizer and an optimizer that uses dynamic programming. In addition, the DBMS also features an interface for external join order optimizers.

Finally, the physical optimization phase starts. The logical operators are replaced with the desired physical implementations in this phase. Afterward, the operators are assigned to devices, where they are finally executed. The executable for intermediates outputs the number of intermediate results instead of the actual result.

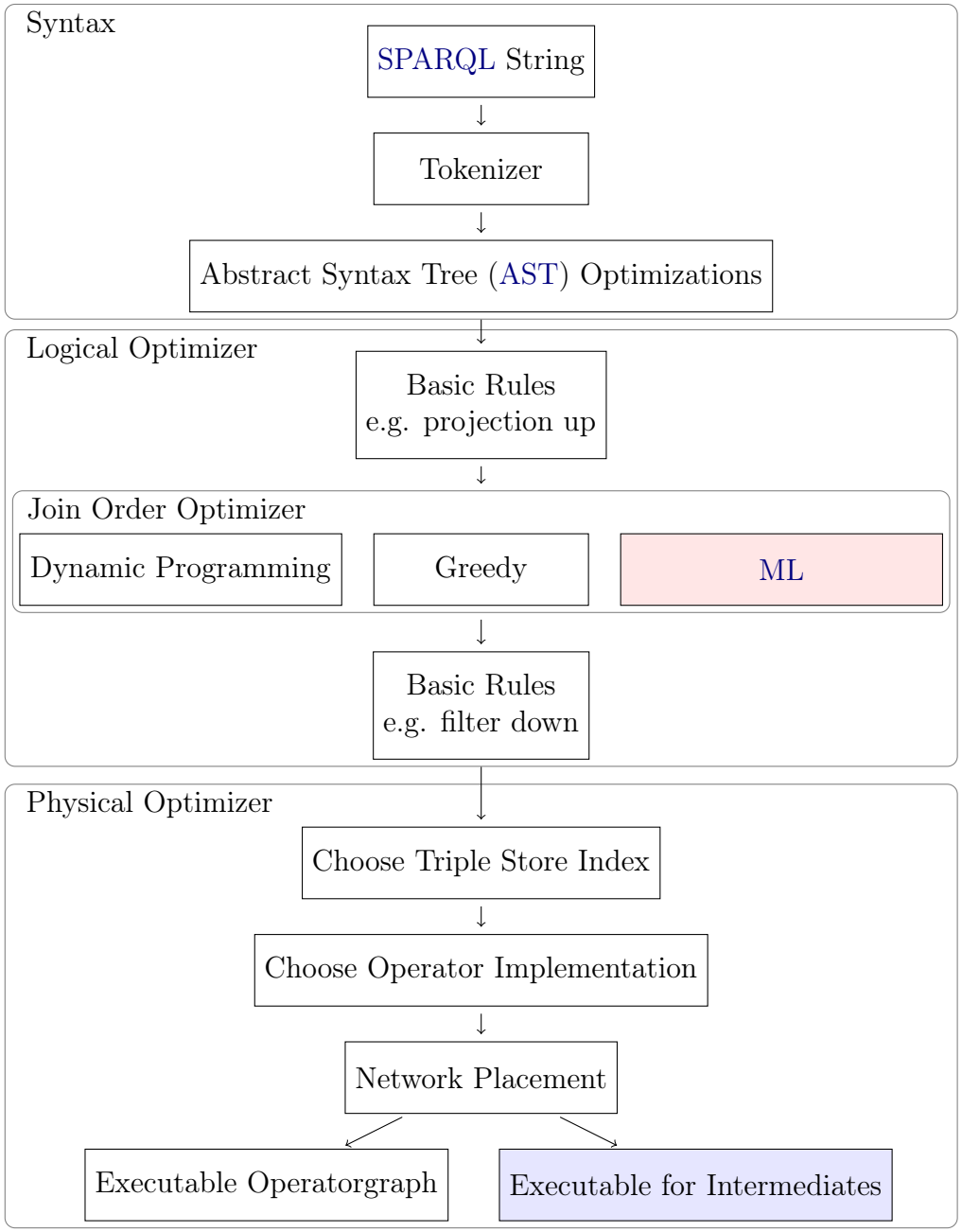


Figure 3.5: Query processing pipeline in LUPOSDATE3000.

3.5.1 Greedy Join Order Optimizer

LUPOSDATE3000 's greedy join order optimizer uses minimalistic histograms with only one bucket. As a result, basic histograms can be extracted directly from the data without additional storage structures. It also allows the optimizer always to use a current histogram. The optimization process consists of several steps: The optimizer collects all input relations to get an overview of what needs to be merged. Then, these inputs are grouped by variable names to identify the star-shaped joins. The idea is that merge joins can be used more frequently this way. In addition to a star-shaped pattern, many identical variable names indicate that the join is likely to reduce the output size. Next, inputs are concatenated so that the estimated cardinality of the output always remains as small as possible. The groups with different variables must be merged in the final step. In doing so, the optimizer tries assembling subgroups with at least one variable in common. Choosing inputs with a shared variable prevents the optimizer from choosing the Cartesian product as long as the SPARQL query allows it. The advantage of this optimization strategy is that the time required to find a join tree is in $O(n \cdot \log n)$, which is the time needed to sort the inputs by their estimated intermediate results.

3.5.2 Dynamic Programming Join Order Optimizer

This optimizer enumerates all possible join orders and chooses the optimal solution. The idea of dynamic programming is that the optimal solution for a problem is composed of the optimal solutions of its partial problems [11]. For join order optimization, simpler sub-problems occur in multiple complex problems. Therefore, it is sufficient to calculate simple problems once and reuse its result multiple times. Figure 3.6 shows an example where four inputs should be joined. Each two-component solution is reused three times, and the original inputs are referenced seven times each. The more inputs that should be joined, the higher the number of reused partial results.

The optimizer based on dynamic programming generates better join trees than the greedy optimizer. However, the time required to create the join tree is $O(n \cdot 2^r)$ [80], where r is the number of inputs to be joined, and n is the number of join orders. The number of possible join orders is defined as $\frac{(2 \cdot n-2)!}{(n-1)!}$ [52]. Since both functions increase exponentially or factorially, this type of optimization is not used in LUPOSDATE3000 for more than 18 inputs.

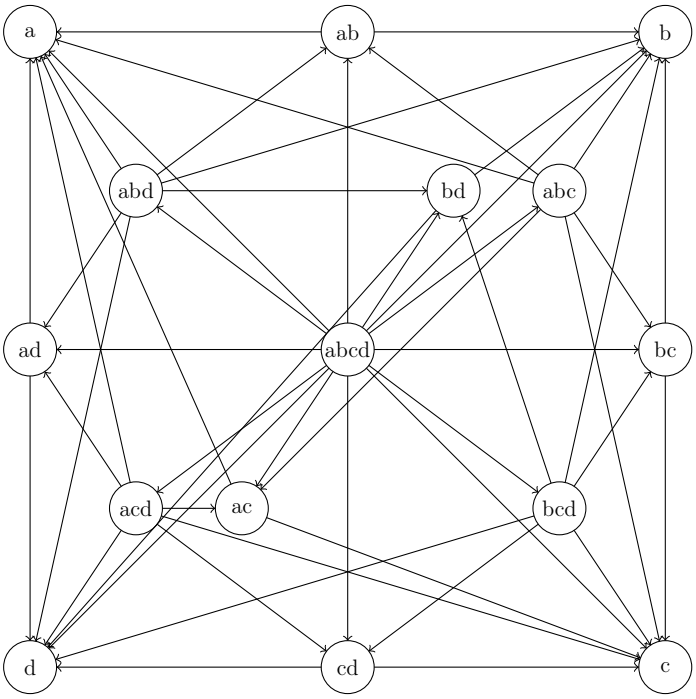


Figure 3.6: Dynamic programming is reusing partial solutions several times.

3.5.3 Interface for External Optimizers

In addition, an interface allows an external application to force a specific join tree during execution. The interface simplifies interaction with the Python programming language, which plays a central role in the ML community. This interface is essential for using ML algorithms in chapter 10 to optimize join orders.

Chapter 4

Experimental Setup

The benchmarks are run on a server OS with Ubuntu *22.04*. The CPU is an *Intel i9-10900K* with a clock rate of up to 5.1 GHz on ten cores with 20 threads. 128 GB of Random-Access Memory (RAM) is installed, although most benchmarks require less than 32 GB. *Kotlin 1.8.255@24.Okt.2022* was used to compile LUPOSDATE3000 and SIMORA. Since the *Kotlin* JVM target is the fastest, all benchmarks use this target unless otherwise noted. The JVM target uses *Java 18* in the server edition. The *stable baselines3 1.5.0* framework with the maskable Proximal Policy Optimization (PPO) modification [77] is used for ML. The ML scripts run on Python *3.9.7*.

4.1 Other Semantic Web DBMS

This chapter introduces other DBMSs that will be used later in evaluating different approaches. None of these DBMSs use ML to optimize the join order. A short overview of these DBMS can be seen in figure 4.1.

DBMS	Language	join tree	licensing	features
Apache Jena	JVM	left-deep	open source	centralized database
Blaze graph	JVM		open source	fully replicated cluster, federated system
Virtuoso	C++		open source	SQL backend
RDF3X	C++	bushy	open source	
Ontotext GraphDB	C++	left-deep	commercial	

Figure 4.1: SW DBMS feature overview.

Apache Jena [26] Version 3.14.0, *Blaze-graph* Version 2.1.6, and *Virtuoso* [145], Revision *840b468fc400a254eab0eb20f1afde6ca3c2220d* were chosen as comparative DBMSs because they are the most commonly used open-source RDF DBMSs according to an RDF DBMS ranking [81]. *Apache Jena* is an RDF store that runs in the JVM. The *Apache Jena TBD* component, responsible for storing triples, is aimed exclusively at centralized DBMSs. Triples are stored in B^+-trees. The *Apache Jena* DBMS is a fully open-source DBMS [84]. This DBMS supports all SPARQL functions. However, the *Apache Jena* Query Optimizer only generates left-deep join trees, so many join orders are not considered.

Blaze-graph also runs in the JVM. The development started later than *Apache Jena*. The DBMS can be configured as a fully replicated cluster to achieve higher query throughput. In addition, multiple DBMS instances can also be combined as a federated system. Indexes are stored in B^+-trees influenced by the *Google BigTable* system.

Virtuoso is written in C++. Only the *Virtuoso* RDF interface is used for evaluation. It allows comparison with a compiled DBMS and is intended to show that garbage-collected languages are not inherently slow.

The *RDF3X* DBMS is used [115, 114]. This DBMS introduced the original *RDF3X* triple-store layout used in many DBMS implementations today. In addition, the join order optimizer creates bushy join trees.

The *Ontotext GraphDB* is a commercial SPARQL DBMS [118]. Similar to *Apache Jena*, the optimizer only generates left-deep join trees. However, the number of intermediate results generated is significantly lower than *Apache Jena*.

4.2 Public available Real World Datasets

Several real world dataset collections are available for research purposes. The purpose of real world datasets during benchmarks is to confront the DBMS with inconsistent complex data structures. Figure 4.2 shows an overview of the used datasets. In addition to the raw text file size, the compressed internal data size is also shown.

dataset	size of turtle file	size of processed data	number of triples	number of dictionary entries
Wordnet	0.4 GiB	0.4 GiB	2637168	1291219
Yago1	0.9 GiB	0.8 GiB	21383706	12752436
Barton	9.5 GiB	1.4 GiB	35184003	10830905
Yago2	5.8 GiB	4.2 GiB	123689922	54351098
Yago3	8.5 GiB	5.4 GiB	142608259	72644117
Yago2s	9.5 GiB	4.4 GiB	151474901	42599960
BTC2019	38.0 GiB	12.9 GiB	256059356	82631100
Yago4	474.0 GiB	82.8 GiB	2489858800	571715647
BTC2010	624.9 GiB	53.2 GiB	1426828906	279151232

Figure 4.2: Overview of real world data sets.

The early large datasets have been published in conjunction with the Billion Triples Challenge (BTC). The latest dataset in this series is the BTC *2019* [74], which contains about 250 million facts. This dataset collects freely available data from hundreds of Internet domains. However, there is no unified structure since the data was only copied together.

The Barton [2] dataset contains library data. Since the data is a collection from several libraries, there are different conventions.

Wordnet [42] is a computerized linguistic dataset. Linguistic researchers created the dataset. Since the dataset was not collected automatically, it is much smaller than other SW datasets. On the other hand, this dataset has a clear structure and goal so that different algorithms can be evaluated against this data.

Yet Another Great Ontology (YAGO) *4* [152] is a knowledge base that contains data from Wikipedia structured according to the ontology of Wordnet. Similar to BTC, several versions of this dataset are publicly available: YAGO *1* [149], YAGO *2* [76], and YAGO *2s* [14]. However, unlike the BTC dataset, a strict ontology is evident. With around 2.5 billion facts, this is one of the most extensive datasets available.

Even though the above data structures follow an ontology, for demonstration and experimentation purposes, it is helpful to use synthetic data as well.

Synthetic data has the advantage that it can be varied in size and structure. Synthetic data allows algorithms to be tested and optimized in controlled environments before encountering complex data. Similar inputs can make detecting connections between the data and the performance easier.

One generator for synthetic data is SPARQL Performance Benchmark (SP^{2}B) [139, 17]. This generator generates bibliography data, and citation graphs are simulated.

Based on these data sets, it can be shown that regardless of the content and origin, all of these data sets share a part of their metadata structure. As shown in figure 4.3, knowledge about this structure allows us to improve several partitioning, distribution, and join order optimization strategies. For example, the analysis shows that the number of different predicate is always the smallest, from which it follows that in each case, a few values are used very frequently. In the graphic, the predicate line is always shown at the bottom. For the subject, most are associated with fewer than 100 values. For objects, most are used only ten times, while a few values, such as small numbers, are used more frequently. For values composed of subject and object, there are almost no duplicates. The Barton dataset stands out here because it contains various library systems. A book has the same authors no matter which library it is in, but how the link is named differs. With these simple statistics, predictions can be made about the quality of the data. However, even more insights can be gained here for DBMS. For example, for each triple pattern, the variable and constant fields can be used to predict how many results there will be. This structural knowledge is beneficial for optimizing join order and data partitioning.

Since these datasets are available to everyone, they are often used to illustrate some features of DBMS. Therefore, these datasets are also used in this work.

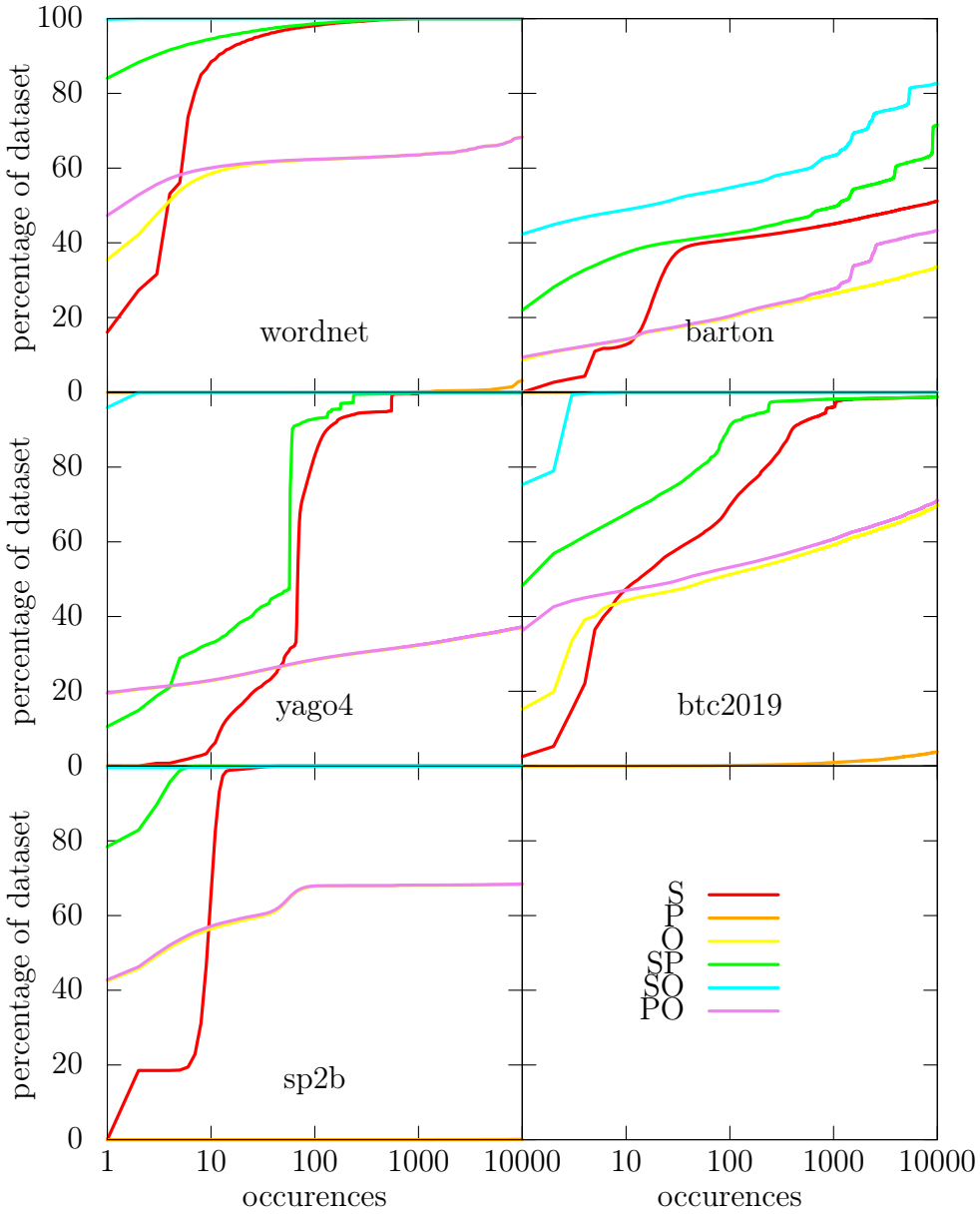


Figure 4.3: The figure shows the cumulative distribution function $f(X < x)$. The X-axis shows the number of triples that share the same value at the columns specified by the legend entry. The Y-axis represents the percentage of the triples in the triple store, which share their value with, at most, X triples. The names of the graphs consist of the constant values of the triple pattern. For example, the predicate graph shows the relation for triple patterns of type $?s < p > ?o$, where the predicate is a constant.

Chapter 5

Flexible Data Partitioning

The two most commonly used join implementations are hash joins and merge joins. Merge joins can achieve much higher speeds and use much less memory but require that the inputs be sorted beforehand. To compute joins in parallel, the data must also be partitioned by at least one join column. This partitioning can be done at runtime or in the triple store. In general, more threads are worthwhile for larger data sets than for smaller data sets. This chapter deals with the approach of flexible partitioning in triple stores. This approach uses the appropriate number of partitions to make the resulting partitioning overhead as tiny as possible.

5.1 The Idea of the Flexible Data Partitioning Approach

The data must be distributed over an arbitrary number of partitions to support parallel data processing. The first problem is to select suitable key columns for partitioning. The second problem is to optimize the number of partitions for this key. If too many partitions are used, the overhead of partitioning and merging is greater than the benefit of parallel processing. On the other hand, if too few partitions are used, fair data distribution is impossible. As a result, the hardware cannot be fully utilized. Later in the evaluation, this is supported by figures 5.4 to 5.6.

If only merge joins are used, the actual computation is so fast that the overhead caused by live partitioning cannot be effectively compensated [53]. Therefore, the materialization of different partitions within an index is proposed. These partitions can then be used as parallel inputs to multiple merge-join threads without incurring runtime overhead.

Figure 4.3 shows that each triple pattern implies a drastically different

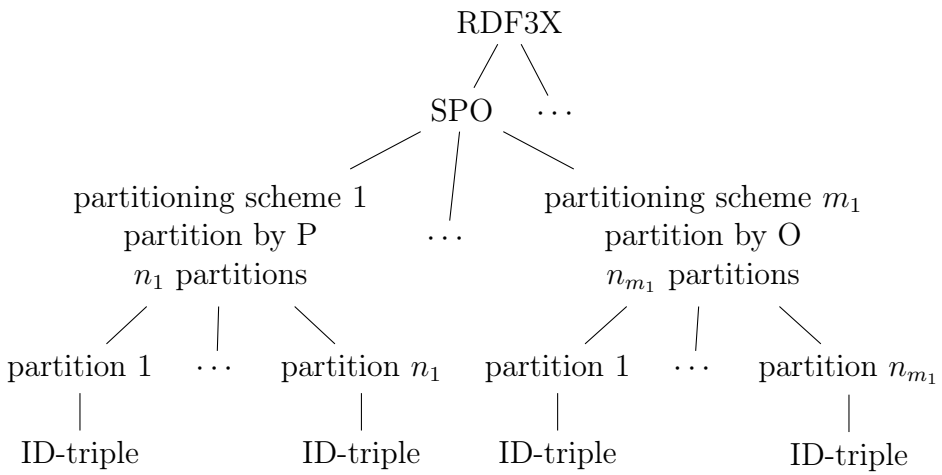


Figure 5.1: Structure of DBMS implementation.

number of triples. Ontology knowledge can be used to specify even more different clusters in terms of the necessary computational pipeline. Therefore, it is not optimal to always use a fixed number of partitions. One possible solution is to use several different partitioning schemes for each index. Multiple indices enable flexible choices depending on the data properties at runtime. Also, the number of partitions used can be chosen during query optimization. This choice allows much finer control over effective parallelism. Figure 5.1 shows a structural example of the proposed triple-store implementation. It is important to note that each partitioning scheme may choose a different hash function and amount of partitions. These additional indexes are all stored on disk to avoid partitioning overhead at runtime.

Each partitioning scheme occupies disk space. The more schemes are defined, the more disk space is required. Therefore, only the most effective schemes should be stored. Unfortunately, this leads to a different problem since many partitions must be considered when planning the join order.

To allow more efficient implementations, each number of partitions must be a power of 2. The reason for the efficiency is that if a different number of partitions are to be joined, the index with the higher number of partitions can continue to merge two partitions until the number of partitions matches. The merging of partitions works as long as the hash function is the same. It is also required that the modulo operator is finally used to trim the hash value to the number of partitions. Multiple partitions allow multiple threads to merge independent partitions simultaneously without any communication between threads. Avoiding thread communication eliminates the need for locks, increasing effective speed.

5.2 Evaluation

Many performance aspects must be considered to determine how the number or presence of an additional partitioning layer affects the performance of the DBMS. If only centralized DBMSs are considered, there are already enough interrelated features that this evaluation focuses only on centralized DBMSs. The same concepts can be transferred to distributed DBMSs. However, the overall evaluation is more complicated in a distributed setting.

5.2.1 Dataset and Queries

```
PREFIX b: <http://benchmark.com/>
SELECT * WHERE {
  ?s b:p0 ?o0 .
  ?s b:p1 ?o1 .
  ?s b:p2 ?o2 .
}
```

Figure 5.2: SPARQL *A1*.

```
PREFIX b: <http://benchmark.com/>
SELECT * WHERE {
  ?s b:p0 ?o0 .
  ?s b:p1 ?o1 .
  ?o1 b:p2 ?o2 .
}
```

Figure 5.3: SPARQL *A2*.

To get clear indications of how exactly partitioning affects execution times, simple queries such as those shown in figures 5.2 and 5.3 are used. *A1* is a template for the benchmarks, which use up to 16 consecutive merge joins. *A2* enforces a hash join since joins are to be made on different variables.

Only synthetic data is used for this evaluation, so properties such as selectivity and result size can be tightly controlled. Furthermore, since the proposed changes have nothing to do with join order optimization, side effects caused by the applied optimization should be avoided. Therefore, the

generated triple structure is the same for each triple pattern in the query, so the join order does not matter.

For selectivities less than 1, a fixed n indicates how many triples should be skipped after a triple involved in a join is generated. Then, this procedure is repeated for each basic triple pattern during data generation. This results in each join operator's selectivity of $\frac{1}{1+n}$. The size of the generated data sets is expected to scale with the join operators. When generating such datasets, m blocks of triples are formed, each block containing the triples to be concatenated. In the following, n and m are restricted to powers of 2. This property is also called selectivity in the experiments since it specifies the factor by which the number of rows within the joins changes.

During this evaluation, all graphs are labeled with the number of results rows instead of the commonly used number of input rows. This way, a uniform workload can be achieved across all threads used. In addition, this prevents a situation where low selectivity combined with a low number of input triples results in unnoticeable few result rows being computed. Having results is essential because too few cannot be distributed evenly among the threads, making evaluation more difficult.

Because of the new partitioning scheme, there are many ways in which a simple query can be executed. For example, it is possible to compute *A1*, shown in figure 5.2, with two merge joins. The following compares the performance of 1, 2, 4, 8, and 16 partitions for each join operator and the triple-store iterators. The options that require a merge join to change the partitioning of its two inputs at runtime have been removed, allowing for $2^2 \cdot 4^3 = 256$ different operator graphs.

Using the same number of partitions in all operator graphs gives the best results since no partitions need to be merged. In addition, different synthetic datasets with uniform or non-uniform data distribution were also considered in the experiments. In the non-uniform synthetic datasets, the join inputs differences are magnified so that one triple pattern provides up to 128 times more data to the following join operators. These patterns change the overall query evaluation time but not the ranking of the optimal partitioning in the operator graph.

Depending on where exactly the number of partitions changes in the operator graph, the evaluation speed is not much lower than operator graphs that use only one number of partitions. This outcome shows that using the same number of partitions is always helpful but unnecessary.

The *A2* shown in figure 5.3 joins two different variables, which means that the second join cannot be a merge join. While the first query removed all variants where the two inputs were partitioned differently, it is necessary to partition them differently. Now the optimizer has to consider more options

to merge these three inputs. One way is to merge the partitions after the first join and then change the partitioning of the result in time for the second join. The other option is to leave the partitioning as it is, meaning the second join is not partitioned by its join column. In this case, the second join must read all unpartitioned inputs from the triple store to continue to get valid results. The measurements show that, in this case, it is fastest to use the second option, where non-partitioned data is read multiple times from the store.

5.2.2 Benchmarks

This section is divided into two parts.

Focus on Merge join Operator

This benchmark focuses on the performance of the merge-join operator. To this end, several DBMS features have been disabled or bypassed. First, the operator graphs, including the partitions used, are hard-coded to examine their impact on query performance. The benchmark code is included in the LUPOSDATE3000 binary to avoid the overhead of the Hypertext Transfer Protocol (HTTP) interface. Finally, the result is computed only as a series of integer IDs since the dictionary queries that would otherwise be required to convert these IDs to strings would introduce significant runtime variations. This benchmark only intends to show the impact on query evaluation when the data structure and size change. Since applying the above changes to other existing DBMSs is complicated, this part will only be evaluated within LUPOSDATE3000.

In the tests, several properties have an impact on the query time. The first observation is that a more significant number of rows leads to higher speedups. This speedup comes from the sequential query's initialization phase, which takes less than the total computation time. The selectivity of joins is also necessary because when more rows are filtered away, the next join operator has less work to do, resulting in faster processing. The number of CPU cores used is significant because, due to the in-memory benchmark setup, all experiments are both memory and CPU limited. Using more partitions than CPU cores does not make sense as long as the data is evenly distributed. LUPOSDATE3000 parallelizes its query evaluation based on partitions only. Therefore, the maximum achievable speedup is equal to the number of partitions. Multiple consecutive joins on the same join columns can increase the throughput since the intermediate results do not have to be serialized or cached when merging.

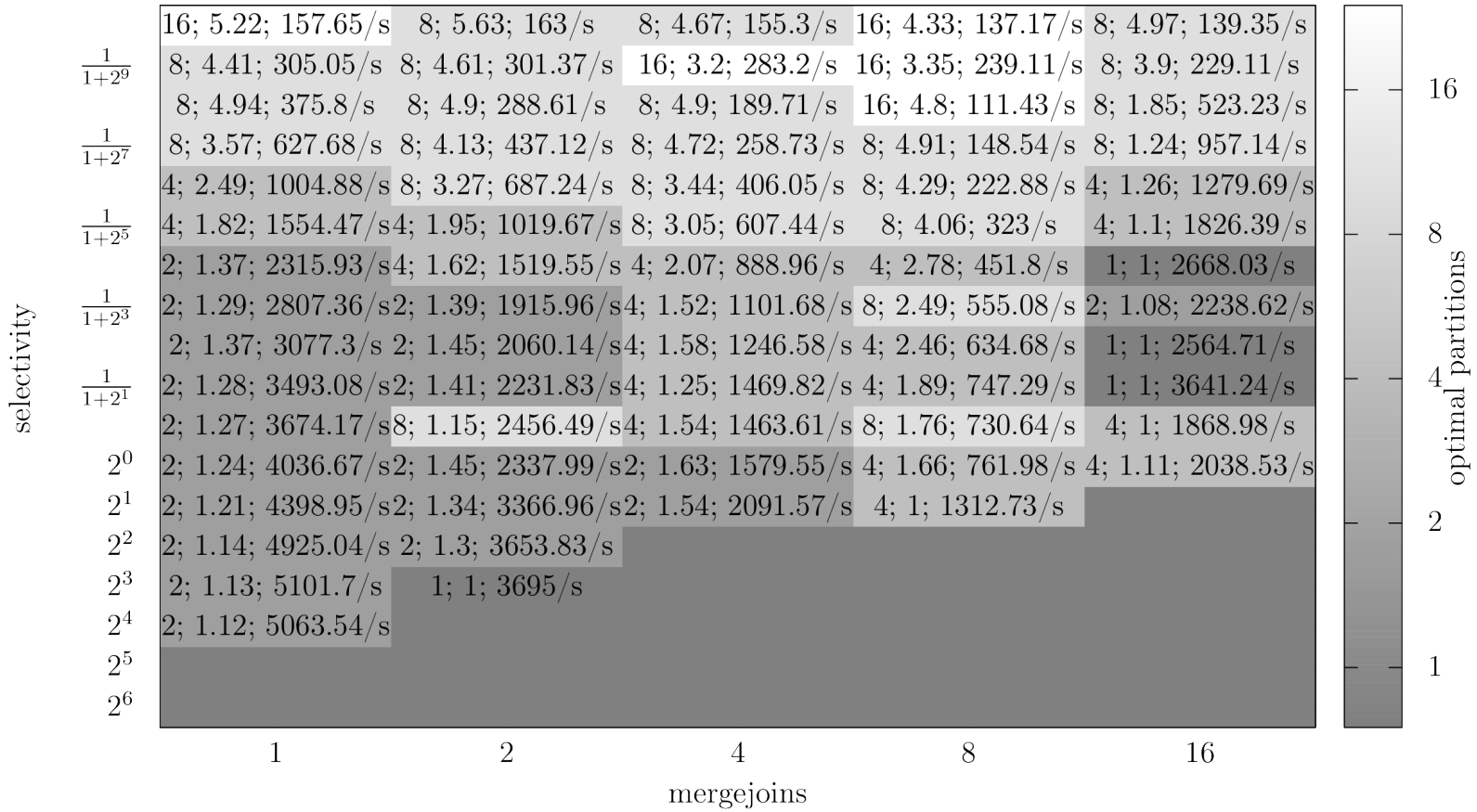


Figure 5.4: An optimal number of partitions depends on the number and the selectivity of merge joins for 512 result rows. Therefore, the labels follow the form *a;b;c*, where *a* is the optimal number of partitions, *b* is the speedup compared to no partitions, and *c* is the queries per second when no partitions are used.

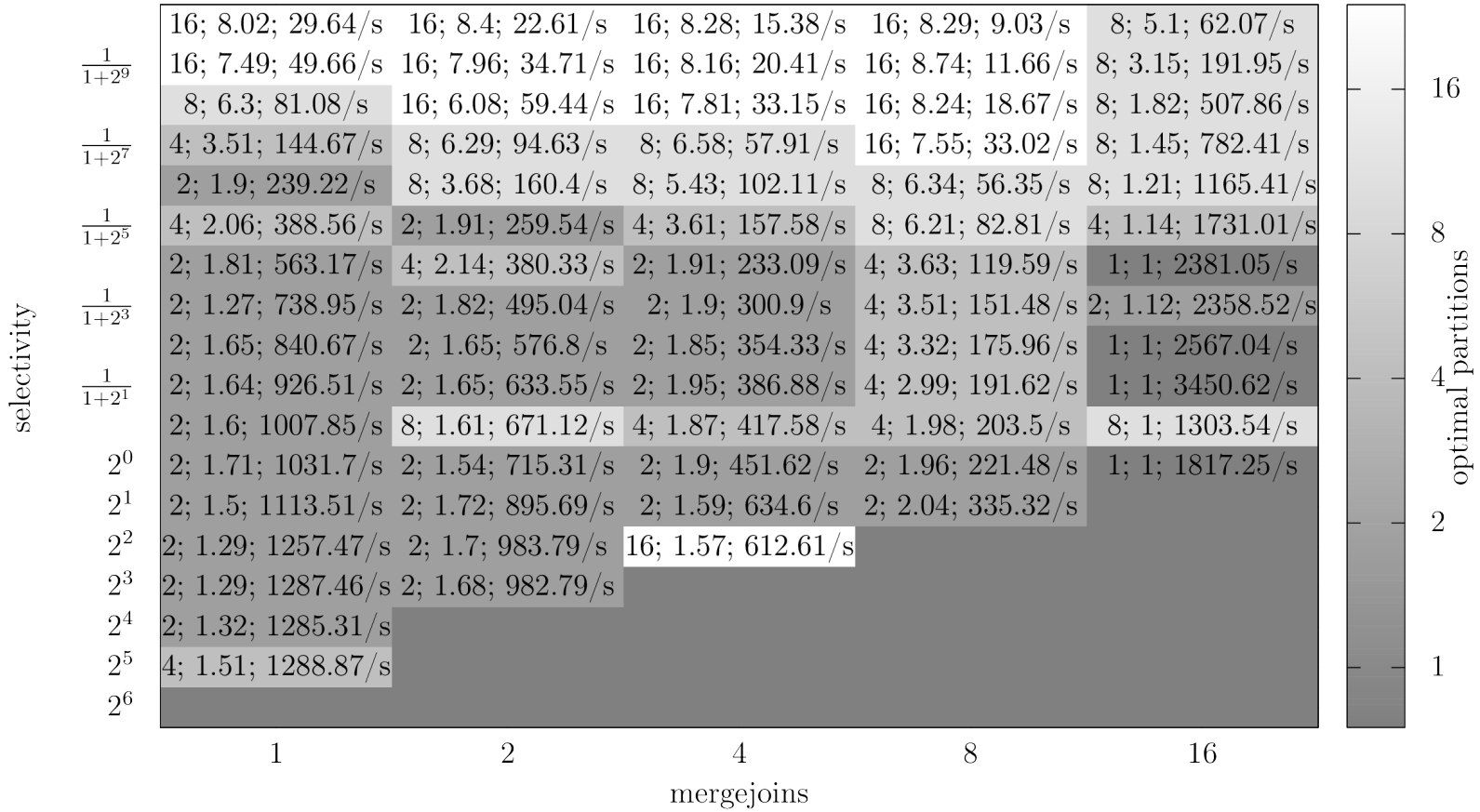


Figure 5.5: An optimal number of partitions depends on the number and the selectivity of merge joins for 2048 result rows. Therefore, the labels follow the form *a;b;c*, where a is the optimal number of partitions, b is the speedup compared to no partitions, and c is the queries per second when no partitions are used.

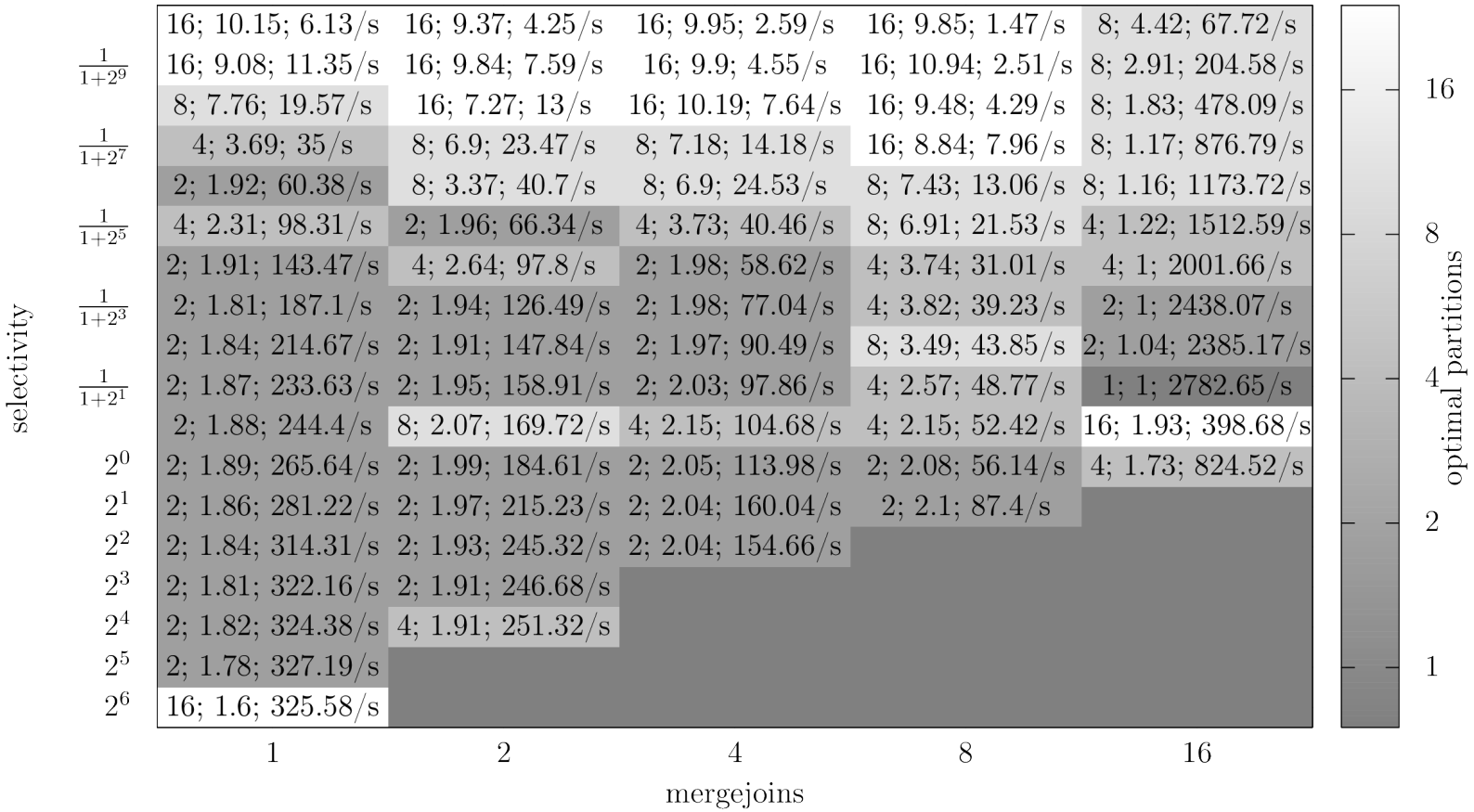


Figure 5.6: An optimal number of partitions depends on the number and the selectivity of merge joins for 8192 result rows. Therefore, the labels follow the form *a;b;c*, where a is the optimal number of partitions, b is the speedup compared to no partitions, and c is the queries per second when no partitions are used.

The SPARQL template shown in figure 5.2 is used to create the queries. Then the number of generated triple patterns is changed according to the desired number of joins.

Figures 5.4 to 5.6 show that all three properties, selectivity, output rows, and several joins, affect the optimal number of partitions in query evaluation.

Experiments are missing in the lower right of each figure because the targeted number of output rows cannot be generated because of a massive increase in rows within the join operator chain, which is higher than the targeted number of output rows. The low optimal partition counts in the lower left portion of each figure are due to the same reason. Due to the deficient number of triples in memory, partitioning there does not make sense.

As expected, the optimal number of partitions is proportional to the workload. The more triples removed due to low selectivity, the more triples must be present in the memory in the first place. The same is true for the number of merge joins. The more different triple patterns are to be joined, the more input triples must be defined. The last property, increasing the number of output rows, naturally requires rising input. Conversely, the result would be similar if the number of input rows was fixed. In this case, the optimal number of partitions would be proportional to the number of output rows.

The results of this benchmark are used to predict the best partitioning scheme for the query optimizer. The polynomial $a \cdot x + b \cdot y + c \cdot z + d \cdot x^2 + e \cdot y^2 + f \cdot z^2 + g$ with x as the number of result rows, y as the number of joins, and z as the expected selectivity are chosen as the basis function. This function was chosen because it promises good results for predicting the fastest partitioning, and the process can be evaluated quickly in the query optimization phase. The function with the constants used, as they can be calculated from the measurements, can be seen in figure 5.7. The auxiliary function $h(z)$ is used here to convert the selectivity values to a similar range of numbers as all other variables. The value is rounded to discrete partitions in the final step. The predictive function $p(x, y, z)$ computes most numbers of partitions optimally. Nevertheless, the mean square error between the predicted partitions in figure 5.7 and the calculated partitions from figures 5.4 to 5.6 is 4.3030. This error is relatively small compared to the many known value pairs that must be fitted.

Even though the above benchmark was evaluated on different synthetic datasets with other queries, the same effects apply to actual data. A different subgraph is accessed each time a query uses another constant, such as another predicate. Each subgraph in an actual data set can contain a different number of triples, as shown in figure 4.3. When two different subgraphs are randomly selected and joined, there is a very different selectivity within the join operators and a different number of output rows by changing

$$h(z) = -log_2(z) \tag{5.1}$$
$$f(x,y,z) = 0.0025 \cdot x + 1.4827 \cdot y + 1.1277 \cdot h(z) \tag{5.2}$$
$$+ 0.0906 \cdot y^2 + 0.0279 \cdot h(z)^2 - 3.3696 \tag{5.3}$$
$$p(x,y,z) = 2^{\lfloor \log_2(f(x,y,z)) \rfloor} \tag{5.4}$$

Figure 5.7: Prediction function for the number of partitions to use

only the query. This observation confirms the assumption that the optimal partitioning scheme changes by changing the query. This changing optimal partitionings, in turn, require partition selection at runtime.

Comparison to other DBMS

In this section, the same queries are used for evaluation. However, this time several different DBMS implementations are compared. All DBMS functions are enabled in this part, and the HTTP SPARQL endpoints are used.

The LUPOSDATE DBMS has been configured to use either its in-memory storage or a disk-based *RDF3X* storage layout. Consequently, the results of both the in-memory storage and the disk-based *RDF3X* storage layout are presented since the internal implementation of the triple stores is entirely independent. All other DBMSs use their default parameters as suggested in their documentation.

The performance measurements of the *Blazegraph* DBMS suffer from substantial measurement inaccuracies. All other DBMSs have a minimal variation in the time required for the same query. To counteract these inaccuracies, the experiments are repeated ten times - and the average measurement values are shown in the graph.

Ten merge joins are performed with different data sets for the experimental comparison. These data sets are generated to obtain the target selectivity for each merge operator. The figures 5.8 and 5.9 show the results of this comparison.

Two significant representatives were chosen, and the graphs show changing selectivities in each case. Both figures show different effects. The experiments of the Jena DBMS and the in-memory LUPOSDATE in the 128 result rows configuration are dominated by the effect that a fixed output size with decreasing selectivity requires increasing input data.

The sequential performance of LUPOSDATE3000 decreases while the time required for partitioned evaluation remains the same for selectivities

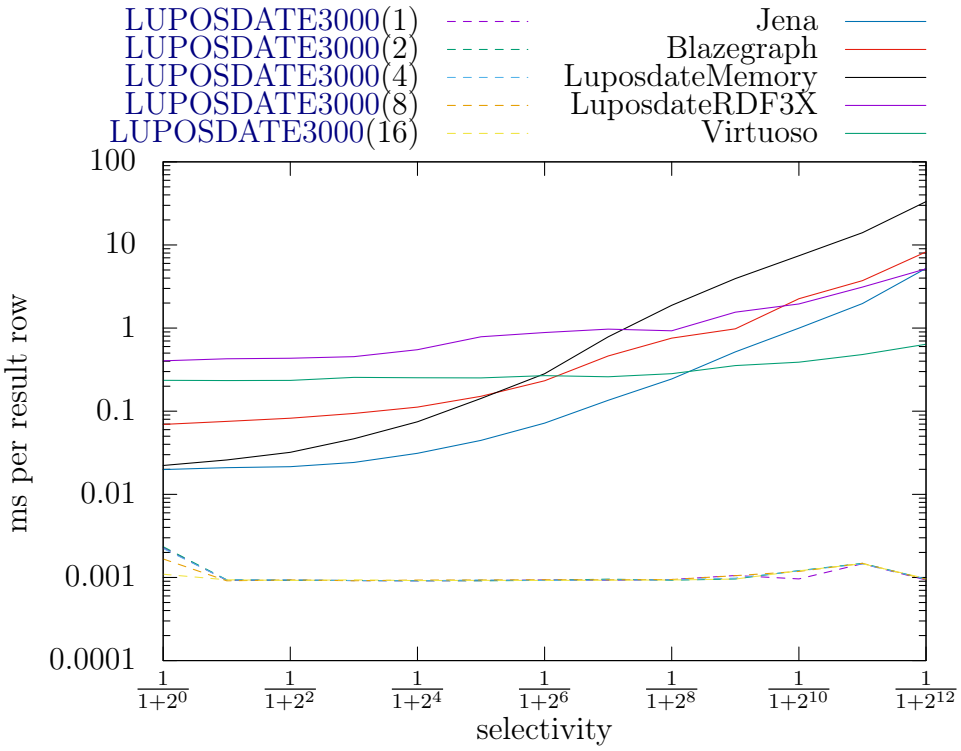


Figure 5.8: Performance of *A1* with ten merge joins on different DBMSs, with a result size of 128 result rows. The numbers in the brackets, for example, LUPOSDATE3000(8), show the number of used partitions.

above $\frac{1}{1+2^9}$. However, the execution times for LUPOSDATE3000 include the overhead for endpoint communication and materialization of the string representations of the values. Nevertheless, a speedup of up to a factor of 1.81 is still achieved compared to no partitioning. This speedup shows that the correct number of partitions is essential for query evaluation.

All other configurations, *Virtuoso*, LUPOSDATE with *RDF3X*, and partitioned LUPOSDATE3000, require the same evaluation time, completely independent of the selectivity of the join operators since, for minimal data, the DBMS needs most of the time for static initialization.

Using 32768 result rows, figures 5.8 and 5.9 shows utterly different performance characteristics, although the only difference in the benchmark setup is the higher number of result rows. *Virtuoso* and LUPOSDATE with *RDF3X* remain unaffected by the change in selectivity. This outcome indicates that encoding the final results' output limits these two DBMSs. In contrast to before, the time required per row decreases with decreasing selectivity for all LUPOSDATE3000 configurations and *Blazegraph*. This effect is because the

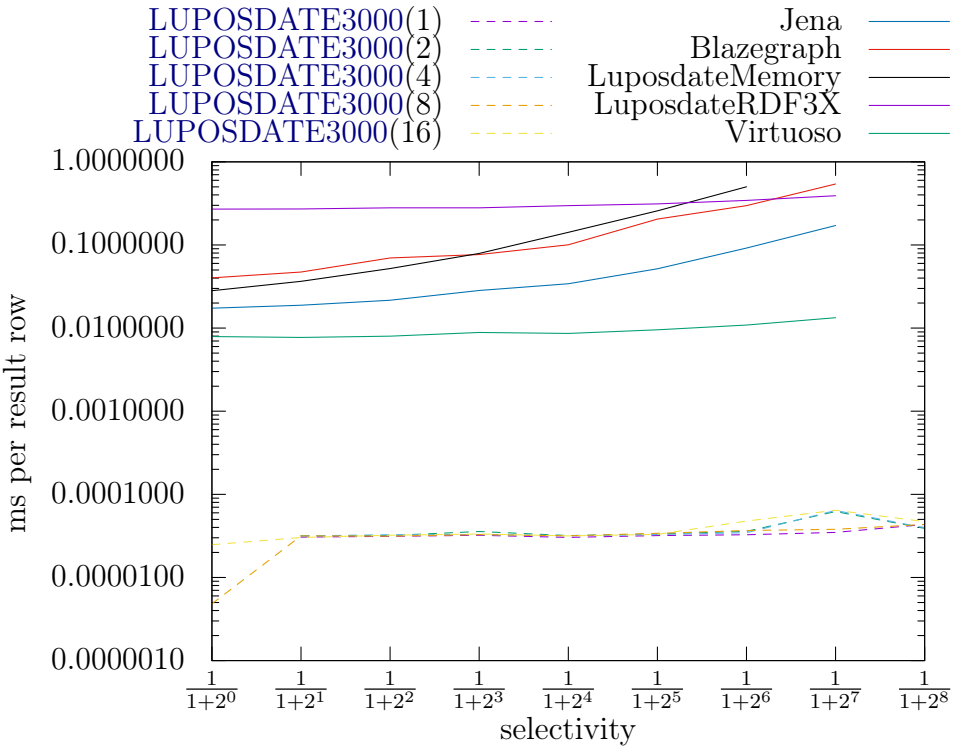


Figure 5.9: Performance of *A1* with ten merge joins on different DBMSs, with a result size of 32768 result rows. The numbers in the brackets, for example, LUPOSDATE3000(8), show the number of used partitions.

static initialization time becomes smaller than the total evaluation time. As a result, more necessary computations during the evaluation phase increase the overall speed per result row. The in-memory variant of LUPOSDATE suffers from poor memory management, mainly because no dictionary is used to map string representations to integer identifiers. In all other DBMSs, dictionaries reduce memory requirements so that out-of-memory errors are avoided during the triple-load phase.

Although the previous benchmark showed that more partitions are better with many result rows, this benchmark with 32768 result rows shows that faster calculations are possible when fewer partitions are used. At the same time, both benchmarks show that the sequential execution is slower than the parallel execution. This effect is probably due to the different benchmark environments. In particular, the sequential text output requires synchronization between the threads, which was unnecessary for the previous benchmark.

5.3 Conclusion

This chapter investigated the factors affecting parallel SPARQL query processing performance. The focus was on data parallelism. According to the experiments, performance depends on the amount of data in memory, the available hardware, and the structure of the query being processed. Furthermore, to avoid the overhead caused by additional partitioning phases, the triple store should materialize several independent partitioning schemes and select the best among them on the fly. Experimental analysis and concepts were presented on how a DBMS can optimize its query processing in the context of multiple partitioning schemes.

Part II

Using Topology Information to Reduce the Network Traffic

Chapter 6

Simulator

This work aims to create and optimize complex applications in IoT environments. Insight into the relationship between the environment and real world applications must be gained while evaluating these optimizations. IoT applications should minimize latency and overall data traffic by shortening transmission paths. Data reduction is primarily achieved by storing data close to the source. With the help of clever memory allocation, data can be sent over the network in a better-compressed form. For this clever memory allocation, the application must know the network and the communication paths. Ultimately, only the application itself is capable of making the optimal fine-tuning. Generic approaches can cover many cases, but no generic system can prune implicit and, thus, redundant information. The application needs an up-to-date overview of the surrounding network for fine-tuning. This information is already processed in the routing protocol algorithms running on each device. An interface between the routing and the application layer is proposed to avoid duplicated implementations. Later in this chapter, it will be shown that this interface can reduce the network load of the entire network. Therefore, a virtual environment is needed to test, evaluate and compare innovative applications. Existing Network simulators often focus on the lower communication layers. An overview of existing simulators will be given in figure 6.1. Specialization allows simulators to focus on a specific topic. As a result, often only fictitious and abstract applications can be simulated. In addition, the scenarios for network simulators do not include resource-intensive applications such as a DBMS.

Consequently, the software stack of the simulators does not consider this scenario either. Another argument for the need for a virtual environment is the heterogeneity of wireless sensor networks. This heterogeneity complicates the feasibility and also the repeatability of experiments. Finally, comparability is problematic since an identical network structure must be reconstructed.

This network structure requirement makes it almost impossible for other research groups to confirm the results. Even within a research group, this can be a problem because the hardware has to be replaced or removed after a few years due to defects. In addition, the hardware environment must be very flexible to support different use cases. It makes no sense to build a fixed environment because it is not reusable. However, repeating the experiment with a modified topology or different device characteristics makes sense for debugging and analysis reasons. Using real hardware has the disadvantage that it is difficult and expensive to configure different scenarios with different network topologies.

In summary, there needs to be a simulator that can simulate the lower communication layers but still focus on the application layer. This simulated environment could then be used to develop and test applications that can fully exploit heterogeneous hardware. However, current applications focus on homogeneous hardware since only such simulators are available. Therefore, this chapter presents the new simulator SIMORA. This simulator should improve the development of innovative IoT applications.

6.1 Fundamentals

In this section, several existing simulators are presented. Afterward, some representative routing protocols are shown.

6.1.1 Simulator

Figure 6.1 contains a brief comparison of the main features of several existing simulators. After this brief overview, these simulators are presented in more detail in the following subsections. Several simulators support edge computing scenarios. Two main groups can be distinguished: those with an abstract application model and those that can evaluate real world applications based on byte-level network simulation.

Abstract Cost Model Simulators

First, the simulators that use an abstract application model with fixed predefined scenarios are compared. These simulators focus heavily on routing protocols. None of these simulators can evaluate real applications.

FogNetSim++ [131] allows the modeling and simulation of predefined fog computing scenarios. *IoTSim-Osmosis* [7] focuses on the composition of applications. Applications are modeled as graphs of microservices that are

Simulator	Language	Routing Protocols	IoT Routing Protocols	Real Data	External Apps	cooperation between routing and application
CloudSim [23]	JVM					
COOJA [119]	C, JVM	✓[0]	✓[0]	✓[1]	✓[1]	
EdgeCloudSim [146]	JVM					
FogBed [34]	Python			✓[0]	✓[3]	
FogNetSim++ [131]	C++	✓[0]				
iFogSim [63]	JVM					
IotSimEdge [85]	JVM	✓[0]				
IoTSim-Osmosis [7]	JVM	✓[0]				
Mininet [117]	Python			✓[0]		
MyiFogSim [100]	JVM					
NS-3 [33]	C++, Python	✓[1]	✓[1]	✓[1]	✓[2]	
PureEdgeSim [108]	JVM					
Shawn [92]	C++	✓[0]	✓[0]	✓[0]	✓[1]	
YAFS [97]	Python					
SIMORA	*Kotlin*	✓[0]	✓[0]	✓[0]	✓[1]	✓[1]

Figure 6.1: Feature comparison of network simulators. 0: without programming effort, 1: via an interface, 2: via file descriptor, 3: via Docker

then distributed in the computing pyramid. *YAFS* [97] is a simulator for Fog computing. The main contribution is the dynamic allocation of application components, device mobility, and device failure modeling. *CloudSim* [23] is implemented in Java and is the basis for many event-driven IoT simulators. *CloudSim*'s model includes keywords such as data center, host, and Virtual Machine (VM) that are generally not found in IoT environments but must still be used to model IoT scenarios. *EdgeCloudSim* [146] extends *CloudSim* with a simulation tool specifically for edge computing scenarios using a modular architecture. It supports modeling Wireless Local Area Network (WLAN) and Wide Area Network (WAN) link delays, positioning mobile devices and access points, and execution of configuration-defined tasks. *PureEdgeSim* [108] is another extension of *CloudSim*. It is a simulation

framework for evaluating the performance of fog, edge, and cloud computing scenarios, focusing on success rate, delays, and energy consumption. *iFogSim* [63] also extends *CloudSim*. It simulates physical fog devices such as VMs, gateways, sensors, actuators, connectivity between them, and communication capacities such as memory, processor, storage, uplink, and downlink bandwidth. The data dependencies of fog applications are modeled as Directed Acyclic Graphs (DAGs). The simulator manages fog application placement and scheduling. *MyiFogSim* [100] is an extension of *iFogSim* to manage VM migration for mobile users. It provides wireless network access points and positioning capabilities. It also adds strategy and policy migration to the simulated applications.

Byte Code Precise Simulators

Some simulators focus on byte code accurate simulation of network packets. This model allows real applications to run on the simulator. There are several ways in which precisely these applications interact with the environment. *NS-3* [33] allows various protocols to be implemented over the Transmission Control Protocol (TCP)-Internet Protocol (IP) stack. The simulator allows real applications to communicate with each other, but the communication is done through file descriptors, which is very slow when moving large amounts of data. *COOJA* [119] is a simulation framework for *Contiki*, an OS for low-power wireless sensor devices. In this simulator, devices are modeled in too much detail, making it too slow for simulating applications with millions of lines of code. Finally, *FogBed* [34] is an extension of the *Mininet emulator* [117]. Its main feature is the simulation of real applications with virtualization in Docker containers. The advantage of Docker containers is that real applications can be run without modifications. The disadvantage is that these Docker containers are heavier than simulators containing the entire program in a binary file.

Cross-platform applications cannot be tested because none of these simulators work across platforms. Similarly, none of these simulators can also be used in the browser. However, the ability to run the simulator in the browser is essential because it allows the simulator to be used on student hardware for educational purposes. This way, there is no need for a strong cluster behind the teaching.

Virtualization Tools

Another approach to simulating network environments is virtualization tools. VMs allow rigid separation between multiple machines. However, this approach comes with considerable overhead when simulating small devices with tiny applications. Modern virtualization tools such as Docker significantly reduce the overhead compared to real VMs for small applications. However, the virtualization tool itself manages the network connections. Therefore, only the already existing network Application Programming Interface (API) functions can be used. However, this chapter is about experimenting with the network API and adding some experimental features. Therefore, these virtualization techniques cannot be used to experiment with the cooperation between the routing protocol and the application.

6.1.2 Routing

There is already much research on the topic of routing. Therefore, there are several existing routing protocols. These routing protocols can be classified based on the properties shown in figure 6.2. Low Power and Lossy Networks (LLNs) are networks in which devices and links are constrained. Constraint means that the devices in these networks operate with limited processing power, memory, and energy. In addition, the interconnecting links have high data loss rates, low data rates, and instability [21]. Therefore, decentralized, dynamic, local, and on-demand algorithms are preferred in IoT. These algorithms can be more energy friendly, require fewer resources since they have only local knowledge, and adapt more quickly to environmental changes.

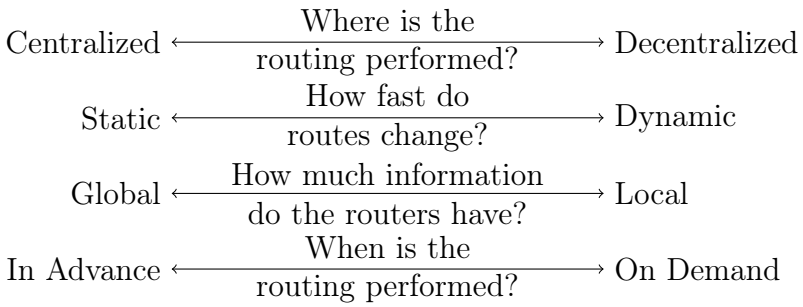


Figure 6.2: Classification of routing protocols. Figure adapted from [106].

This chapter focuses not on the protocols but on their impact on the applications built on top of them. First, however, a brief overview of the essential protocols is given.

One of the first researches to provide IPv6 routing over the IEEE 802.14.5 standard [110] was done by Internet Engineering Task Force (IETF) and the IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) Working Group. After that, other research with the same goal was proposed: Hi-Low [169], Lightweight On-demand Ad hoc Distance-vector Routing Protocol (LOAD) [27], Dynamic MANET On Demand (DYMO)-Low [88], hybrid routing protocol (HYDRO) [36] , and LOAD next generation (LOADng) [31].

After defining the routing requirements for LLNs in urban [102], building automation [37], home automation [20], and industrial [127], the Routing Over Low-Power and Lossy Networks (ROLL) working group proposed the Routing Protocol for Low power and Lossy Networks (RPL) protocol [21]. Although most new routing protocols for LLNs were inspired by the aforementioned existing protocols, RPL and LOADng were mainly used as a reference for recent research on routing for LLNs [144].

LOADng [31] is a reactive protocol based on the Ad-hoc On-demand Distance Vector (AODV) routing protocol [125] but adapted for LLNs. In this case, a device establishes a route only when forwarding information. For this purpose, the source device sends an Route Requests (RREQ) message. Once the destination is reached, it sends an Route Replies (RREP) message to the source. In this case, LOADng does not use an intermediate RREP message. In addition, an RREP acknowledgement (ACK) message may be required to establish a bidirectional path. Each device updates its routing tables with the information of received packets used in forwarding. Finally, a Route Errors (RERR) message reports a problem in setting up the route.

The IETF has defined RPL [21] as the standard for LLNs. It is a proactive distance vector routing. It organizes a network topology as a DAG divided into Destination Oriented DAGs (DODAGs), where one DODAG per sink is called a root or border router. RPL creates the DODAGs using the Objective Function (OF), which defines a metric for routing purposes. The root device starts the DODAG formation by sending DODAG Information Object (DIO) messages. When a device joins the DODAG based on the policy defined in the OF, it establishes its rank along with its preferred parent en route to the root device. Destination Advertisement Object (DAO) messages are then used to establish an upward path to the root device. Intermediate devices can resend a DIO message. Although RPL is designed for MultiPoint to Point (MP2P) upward forwarding, it can also support point to multipoint (P2MP) downward forwarding. This chapter focuses on the distributed and multicast functions in the routing process, focusing on the latest research on multicast routing for LLNs.

Stateless Multicast RPL Forwarding (SMRF) [116] was proposed to enable multicast forwarding in RPL networks. However, SMRF only allows

downlink multicast traffic in a DODAG, which is helpful for service discovery and network management. Since each device receives packets from its preferred parents and can forward a package once, uniquely identifying a package is unnecessary. SMRF uses cross-layer optimization for multicast transmissions.

Enhanced Stateless Multicast RPL Forwarding (ESMRF) [40] provides an enhancement to multicast transmission. Since SMRF can only send multicast transmissions downstream, ESMRF can send multicast transmissions downstream and upstream in a DODAG. When a device needs to send a multicast packet, it is encapsulated with Internet Control Message Protocol version 6 (ICMPv6) and sent to the root device, which forwards the multicast packet to the destination.

Multicast Protocol for Low-Power and Lossy Networks (MPL) [78] uses a trickle algorithm to organize control and data multicast transmission. MPL avoids using a multicast forwarding topology through propagation within an MPL domain.

To improve energy efficiency and bandwidth utilization, Bi-Directional Multicast Forwarding Algorithm (BMFA) [122] proposes to improve SMRF to enable uplink and downlink multicast transmission.

Bidirectional Multicast RPL Forwarding (BMRF) combines the best features of RPL-multicast and SMRF to provide bidirectionality, link-layer broadcast, and unicast. In unicast mode, BMRF uses the RPL multicast method; in broadcast mode, it uses the same method as SMRF. A mixed mode is possible, where the unicast and broadcast modes are mixed depending on the number of interested subordinate devices.

The simulator provides an All Shortest Path (ASP) Routing to simulate the optimal case for unicast. In this case, the routing tables are created using Floyd-Warshall based on the global topology. For each routing protocol, the sum of all distances on the path is used as OF.

Since RPL is the established standard in IoT [144], the simulator also supports this protocol. Another feature of RPL is that it constructs a tree-shaped routing network on the nodes, similar to the SPARQL optimizer that often builds tree-shaped operator graphs. Finally, RPL is used in storing mode so that the DBMS can query the routing protocol for the next hops with a DBMS.

The aim is to distribute the operator graph to the nodes using multicast messages. For this purpose, the operator graph is to be assigned to the nodes so that the operator graph is adapted to the network topology. Unfortunately, the existing Application Layer Multicast (ALM) explicitly wants to hide the underlying network topology from the application [49]. Since this is precisely the information that should be used, traditional ALM cannot be used.

6.2 The Simulator SIMORA

SIMORA is intended to help with the development of IoT DBMSs as well as other applications. Therefore, the main goal is to simulate a heterogeneous IoT network on the one hand and full-fledged applications on the other. The timing and network packet model must be chosen accordingly to simulate full-fledged applications. In addition, the application should respond to its environment regarding network topology, transmission cost, and device characteristics. Therefore, the simulator must provide an interface for interaction between the routing and application layers.

Furthermore, a portion of the network stack must be simulated to relay information from routing to the application. In addition, the simulator should allow the simulation of different environments to provide better insight into the influences of the environment on the application. Finally, the simulator must be flexible to adapt to new requirements quickly.

Since no existing simulator allows interoperation between applications and routing protocols, a new simulator called SIMORA is introduced. The simulator has several functions, each of which will be presented in one of the following sections.

6.2.1 Flexibility and Configuration

The simulator was developed as a tool for evaluating advanced approaches for applications in the context of the IoT. A modular approach is taken in the simulator to enable high extensibility.

The simulator is written in *Kotlin*, allowing applications to be integrated into any programming language. In addition, this allows the simulator to be used in many environments, as Kotlin has multiple targets, including JS, JVM, and native.

The application only needs to implement the appropriate interfaces provided by the simulator to be run within the virtual environment. The interfaces are mainly related to initialization and sending and receiving messages. Existing network interfaces are not supported since the goal of the simulator is to add new functionality to the communication layer.

Any compatible application can be configured directly in the configuration file passed to the simulator. In this file, the user can freely specify which application should run on which device. It is also possible to use generated parts of the configuration file when initializing the application. These generated variables are important because the configuration specifies patterns where a single statement in the configuration file creates multiple similar devices. In pattern-based topology specifications, information about inher-

itance must be available at the initialization phase of the application. For example, the placement of sensors may be divided into regions, and different properties may exist within these regions.

For analysis purposes, SIMORA was developed to measure many relevant details of applications in the context of the IoT. In particular, SIMORA measures the number and size of individual messages grouped by type. The application itself can freely choose these message types. Furthermore, each application can define its tagged network packets so that statistics can be used to quickly understand which packet type is causing which network traffic.

SIMORA takes into account the length of the communication path in its output. These insights allow for analyzing the application in more detail to minimize the overall network traffic. Another essential aspect is latency. SIMORA can measure the application's response times so that the performance of different implementations can be compared under the same workload.

6.2.2 Time Model

There are two possible time models, event-driven and time-based.

The time-based model has the advantage of a continuously increasing simulation clock. However, due to the combination of routing and applications, the time intervals between events vary greatly, making this approach impractical for the simulator.

Therefore, SIMORA follows the event-driven simulation model, which is more suitable for simulating the application layer. It allows easy integration of complex applications. At runtime, there is a virtual clock. This clock is updated according to the next event so that the possibilities are entirely decoupled from the actual execution time.

In addition, SIMORA can measure the execution time of the processing that follows the reception of a network packet. This measured time is scaled according to the configuration so that each application considers the locally available processing capabilities of the simulated devices.

6.2.3 Communication Model

The lowest simulated layer is the packet layer. Routing is performed based on these packets. From the simulator's point of view, each packet only needs to implement the interface. Therefore, to speed up the simulation, sending binary packets is unnecessary. However, this can be useful to test the correctness of the distributed application itself. However, for non-binary packets,

the simulator only knows the approximate size given by this interface, so it should be specified as precisely as possible.

6.2.4 Network Stack

```
interface IActuator{
    fun setMiddleware(m: IMiddleware)
    fun startUp()
    fun shutDown()
    fun receive(pck: IPayload) : IPayload?
}
```

Figure 6.3: Network Stack Actuator interface.

```
interface IMiddleware{
    fun addActuator(child: IActuator)
    fun registerTimer(time: Long, entity: ITimer)
    fun flush()
    fun send(dest: Int, pck: IPayload)
    fun resolveHostName(name: String) : Int
    fun getNextFeatureHops(dest: IntArray, filterKey: Int) : IntArray
    fun closestDeviceWithFeature(filterKey: Int) : Int
}
```

Figure 6.4: Network Stack Middleware Interface.

Figures 6.3 and 6.4 show the most important interfaces of the network stack. Each layer implements the actuator interface from the device to the application. Conversely, each layer implements the middleware interface from the application to the device. Consequently, the device does not need an actor interface, and the application does not require a middleware interface, but everything in between must implement both. These interfaces create a bidirectional connection between successive layers. The functions in figure 6.3, line 2, and figure 6.4, line 2 enable this communication. The naming of the functions in figure 6.3, lines 3 and 4 is self-explanatory.

The functions implement timers in figure 6.4, line 3 and 4. Each layer in the network stack can register a timer to implement simple repetitive events, such as sensors periodically acquiring and sending current readings.

Applications mainly generate packets. The packet's destination address must be defined to send a packet (function figure 6.4, line 5). Therefore, each device receives an automatically generated number during its initialization phase. The simulated devices can be named in the configuration so that name resolution (function figure 6.4, line 6) is possible. Named devices

are implemented only for simplified initialization. Name resolution is read directly from the configuration without any accurate simulation.

The network packets then pass through a simulated configurable application stack. Each step in this stack can recode the packet, group it with others, compress it, or perform another user-defined function. At the end of this stack, the packet is passed to a routing protocol implementation that calculates where to send the package. The result of precisely this calculation can be queried by the application (figure 6.4, lines 7 and 8).

The packet is then routed over multiple hops to its final destination. Finally, the packet is forwarded to the device's local routing layer to implement routing over multiple hops, which can decide whether to forward the packet to another device or process it locally.

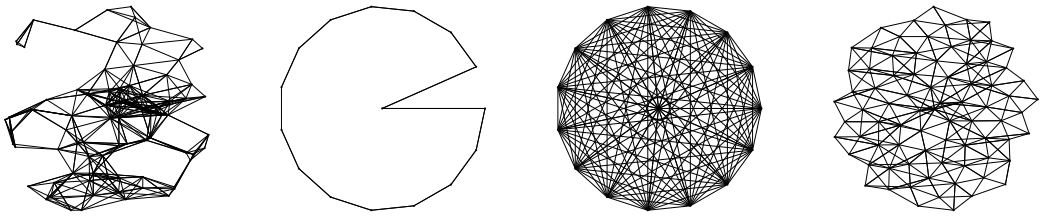
When the packet arrives at its destination device, it traverses the application stack in reverse order (function figure 6.3, line 5) until any part of the application stack or an application consumes it. While processing the packet, the application may also send new packets.

One of these network layers ensures that the network packets arrive in the same order as they were sent. The order of network messages is significant for DBMSs. For example, a table must be created before data is inserted into that table.

Currently, the simulator does not simulate packet loss. This package loss is irrelevant to the application evaluation since there is nothing the application can do about it. Choosing a different route is the only way to work around packet loss. Choosing another route is calculated by the routing protocol, which then informs the application of the current route. Since the focus is on the application layer, packet loss will not be considered further in the remainder of this paper.

6.2.5 API Design

Simulator applications can only explicitly send and receive packets, as shown in figures 6.3 and 6.4. For both stateless and packet-based applications, this is the natural network perspective. Common APIs such as Message Passing Interface (MPI) also provide a stateless API so that these applications can be used in the simulator with only minor modifications. However, the application's network module must be rewritten for socket-based applications to handle the packets.



(a) Random (b) Ring (c) Full Mesh (d) Uniform

Figure 6.5: Topologies as provided by the simulator.

6.2.6 Predefined Topologies

Due to the heterogeneity in IoT, it is challenging to perform standardized experiments with defined network structures. Using real hardware has the disadvantage that it is time-consuming and costly to configure different scenarios with different network topologies. Furthermore, different hardware in different environments limits repeatability. This environmental requirement makes it almost impossible to confirm experimental results for research purposes. SIMORA supports several predefined network topologies, which can be seen in figure 6.5. In addition, topologies can also be defined manually. These predefined topologies can be nested and combined to create complex patterns.

Different communication technologies can be used for each nesting level. This configuration allows the simulation of multilayer network architectures. For example, many routing protocols assume that a particular device is connected to the Internet, which is then used as the starting point. For this reason, and to allow nesting, each predefined topology is centered around a device.

6.3 Evaluation

The only measurable parameter related only to the simulator itself is the time it takes to start. The performance during runtime will be evaluated in later chapters with more complex applications and configurations. The startup time of SIMORA is closely related to the number of devices to be simulated. Since the simulator is intended to be used in IoT-like scenarios, it must be possible to simulate many devices. It turns out that almost all of the time is spent initializing the routing protocol. Therefore, figure 6.6 compares the two implemented routing protocols and the *Kotlin* targets. The used topology does not matter because the routing protocols must consider all possibilities.

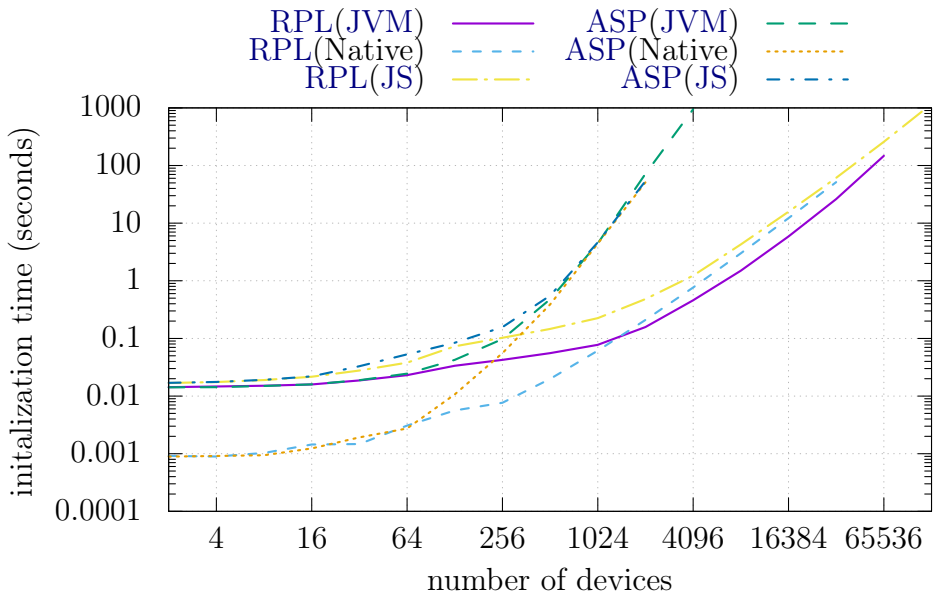


Figure 6.6: This figure shows the SIMORA's setup time for different numbers of devices.

While ASP routing has an initialization time complexity of $O(n^3)$ due to Floyd-Warshall, RPL has a time complexity of about $O(m \cdot \log(n))$. The different *Kotlin* targets also have other performance characteristics. The native *Kotlin* target suffers particularly from a slow garbage collector. On the other hand, the JVM requires much more RAM than any other target.

6.4 Summary

The simulator provides the necessary functions to simulate fully parallel data input. Many existing applications could benefit from improved test and evaluation environments. A simulator can verify the scalability of applications to many data inputs and outputs in the standard build pipeline. Routing algorithms know various network topologies as they form the basic communication layer between devices. However, applications are not aware of these topologies.

Chapter 7

Dynamic Content Multicast

Messages can be distinguished according to the number of senders and recipients involved. The simplest message type is one-to-one, where a sender sends a single message to a receiver. This type is also referred to as unicast. Unicast is useful when the information is only needed at a single destination device.

However, sometimes the same information is to be sent to multiple recipients. Here, a distinction is made between a one-to-all message, which is also called a broadcast. Furthermore, a message to be sent to several recipients is called multicast.

Sending similar information to multiple destinations in the context of DBMSs can make sense. For example, when data is inserted, the data is replicated and distributed by a hash function that sends the same data to multiple devices on the same network. In this scenario, portions of the message can be reused to be sent once and read many times along the way of the package. This message is read on each device on the network route, and only the data that is still needed later is forwarded in each case to send as little data as possible. This approach is referred to as DC Multicast in the following.

7.1 The Idea of Dynamic Content Multicast

In this section, the DC multicast is explained with an example. The corresponding topology can be seen in figure 7.1. All gray nodes in the image correspond to the filter specified by the application. The others do not. For simplicity, only the relevant network connections are shown.

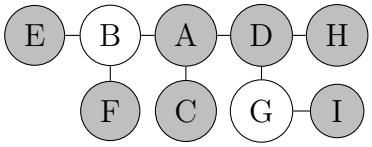


Figure 7.1: Example network layout for demonstrating DC multicast.

To create a multicast tree, the application must first pass a list of destination addresses to the function in figure 6.4, line 7.

Suppose that A wants to send some data to C, E, F, H, and I to cover all possible variations. The function then sends the data to each of these addresses.

Then the routing layer can compute the multicast tree, considering all devices. Finally, it removes intermediate devices from the multicast tree if they do not meet the application's criteria.

In this example, devices B and G are removed. Devices E and F receive unicast messages from A because device B cannot read the contents of the packet.

After removing G, I is directly reachable from D. It does not matter how much or what data is transmitted in this case.

Finally, the application obtains the mapping shown in figure 7.2, which assigns each destination to the next hop on the path to that device.

destination	C	E	F	H	I
next hop	C	E	F	D	D

Figure 7.2: mapping of destinations to next hops.

This mapping lets the application know which destinations have the same next hop.

The data for devices H and I can be combined into one message sent to D. Device D can read the message entirely and create new messages that are then forwarded. This understanding allows the entire message to be compressed and encrypted, even if the content is intended for different recipients. This knowledge also allows the application to interpret the content of

the message. In this example, this avoids sending configurations overwritten by more specific configurations, thereby reducing the size of the message. An example of a packet structure is shown in figure 7.3.

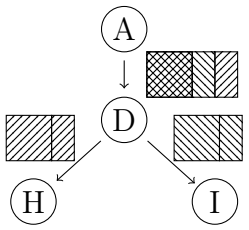


Figure 7.3: Example for DC multicast package structure.

This knowledge allows D to split an incoming packet into two filtered packets sent to H and I. The main advantage over sending multiple independent packets is that the application can infer which parts should be sent to which destination, so this does not need to be transmitted. Unfortunately, the current multicast layers cannot provide this functionality because, from their point of view, they are just bits and bytes that could be identical.

7.1.1 Inverse Dynamic Content Multicast Tree

The multicast tree can also be used in reverse order. However, this is only possible with the cooperation of the application and routing layers. Let us take the same network layout in figure 7.1 as an example. Suppose A wants to count all the data from nodes C, E, F, H, and I.

Two phases are required to obtain the information. First, A must inform all nodes that information should be sent. Then, each node must send its response back to A.

In the first phase, dynamic multicast does not provide any advantage because an identical message is sent to many destinations. This task also works with current multicast protocols.

In the second phase, device D can handle the responses from H and I and only forwards the final result back to A. Of course, there are better options than simply aggregating two numbers. However, this can be extended to any complex stream processing where the application can evaluate the complex processing directly on the devices on the optimal communication path. Depending on the network and application, this can significantly reduce traffic.

7.2 Application View

Applications can improve their network communication using a suitable multicast tree. The application must be able to specify criteria as to which user-defined properties the participating devices must fulfill to make optimal use of multicast. Only the routing protocol can calculate this optimal multicast tree, considering the topology and all hardware aspects. SIMORA provides the application with an interface function, as shown in function figure 6.4, line 7. For performance reasons, each filter is initialized once during initialization, where it is given a fixed ID that can be used later. For example, one possible filter condition is to select all devices which run an instance of a specific application.

7.3 Scenario

Since a DBMS is a very complex scenario, this chapter uses a much smaller example to evaluate the concept and benefits of DC Multicast. Later, this strategy will be shown and reused in the LUPOSDATE3000 DBMS. As a simple use case, suppose that all IoT device configurations should be changed according to the preferred settings stored on the user's mobile device when the user enters his home.

All devices share a global configuration, such as the user ID. Then, many devices can be divided into groups, each with an additional global group configuration set. For example, all speakers can have a maximum shared volume, and all lamps can share a brightness setting. For simplicity, let us assume that the global part of this message is 128 bytes, each group shares another 64 bytes, and finally, each device gets 32 specific bytes that are not needed by any other device. This message is sent to 32 devices divided into four groups of eight devices each. Each topology defines 128 devices.

7.4 Evaluating the Influence of Cooperation between Routing and Application Layer

All combinations of the parameters routing protocol, topology, and packet type are compared to compare the effect of DC Multicast. The result can be seen in figure 7.4. The simulator implements RPL and ASP routing protocols and four basic topologies Ring, Random, Full, and Uniform. The message types to be compared are unicast, broadcast, and multicast. For multicast, there are two variants: State Of the Art (SOA) multicast, where the same messages are sent to multiple hosts, and the new DC multicast. In SOA multicast, messages are split as much as possible. In real applications, where the global section may contain default configurations that the individual sections can override, the message could be reduced even further. The effects of all three parameters must be compared at once because they strongly influence each other, so it is impossible to analyze them individually. The most obvious observation is that the total network traffic in the fully connected network is lower than in any other topology because each device can communicate directly without forwarding messages.

Consequently, in this topology, the unicast approach is most appropriate. However, this is not a realistic wireless network due to full connectivity. The ring topology has the fewest connections of all the topologies. As a result, many messages must be forwarded multiple times. In this context, the broadcast has the disadvantage that the later devices in the communication path receive too much information.

On the other hand, the unicast method sends the same data multiple times to the devices near the sender. Only the DC multicast method can reduce the number of data sent because it does not send the same data twice at the beginning. Simultaneously, it reduces the number of data sent at a greater distance from the sender. The overhead of the prior art multicast method is due to the increased number of messages, where each message must contain its destinations further down the multicast tree. The uniform topology is very similar to the random network design. The main difference is that local clusters can occur in the random topology. Regardless of what the topology is or how the routing is specified, the broadcast method is always the worst case because too much unnecessary data is sent. The multicast implementation, which can use both routing and application knowledge, can reduce the amount of data sent by a factor between 2 and 6.

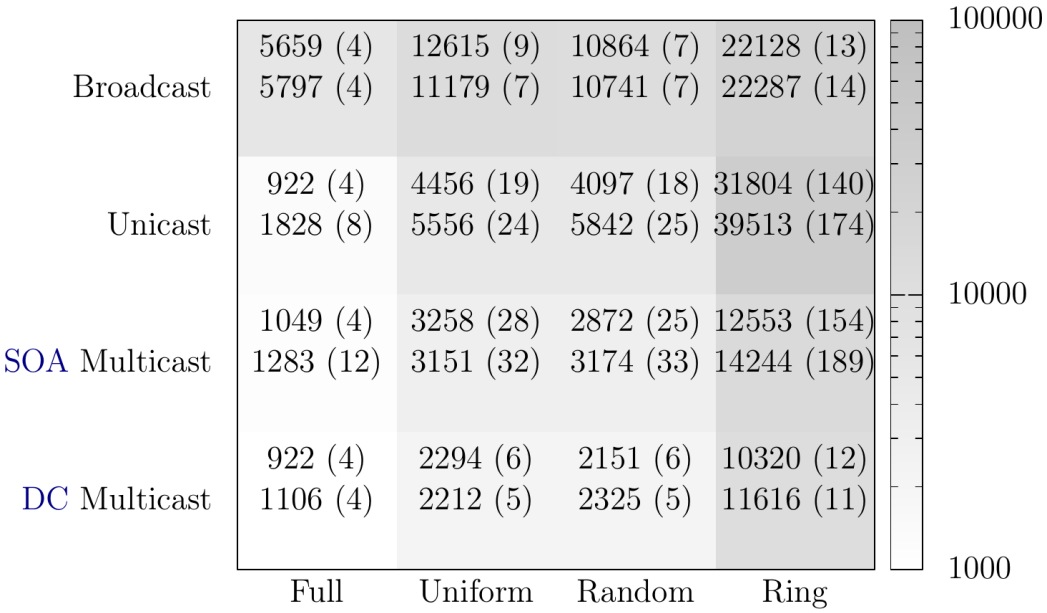


Figure 7.4: Amount of KB sent in the network. The number of messages sent (in thousands) is shown in brackets. All numbers refer to the transport layer messages sent between two devices, aggregated over all devices. The upper value represents ASP routing, and the lower is RPL routing.

7.5 Conclusion

This chapter presents a simple IoT scenario for distributing configurations to IoT devices. Then, the impact of different topologies and routing protocols on the total number of transport layer messages and their size was measured and compared. Advanced techniques such as DC Multicast can thus reduce the number of messages and the amount of data sent. In the experiments, the number of messages can be reduced by 17 or 94 %, and the number of bytes transmitted over the network can be reduced by up to 29 % compared to conventional multicast techniques. Even though a straightforward scenario was chosen for illustrative purposes, the method can also be applied to more complex applications. More complex applications will be demonstrated in the following chapters.

Chapter 8

Benchmark Scenario

Many DBMSs claim to support SIoT scenarios. Several benchmarks compare the behavior of DBMSs in different application scenarios. The main goal of many benchmarks is to show how fast DBMS queries can be evaluated. These benchmarks already consider many aspects, such as query variations to avoid caching full results and data streams to simulate changing data sets over time [139, 5, 61, 126, 17]. However, in the context of the IoT, applications must be as energy efficient as possible. One aspect of energy efficiency is to send as little data as possible. The realization that network traffic slows down the evaluation of queries is already considered in many DBMSs [53, 69, 3, 51, 173, 64]. As a result, there are already many approaches to reducing network traffic. However, the importance of the network topology is always neglected. One must compare different network topologies to measure the amount of data sent. To compare topologies, it is essential to examine how the data was inserted into the DBMS since different indexing variants produce very different communication patterns. Distributed data insertion is a crucial aspect of sensor data in IoT. After analyzing the most commonly used SPARQL benchmarks, none is designed to insert data into the DBMS in a distributed manner.

Therefore, a new benchmark framework is presented in this chapter to gain insights into the interoperability between routing and application, and thus effectively reduce communication costs. Various DBMS indexes are evaluated regarding how much traffic they generate in a fully distributed environment. Because repeatable tests in a distributed environment are the target, the benchmark scenario is built to work in conjunction with SIMORA. SIMORA can be used to simulate and compare a variety of network topologies. This variety allows for determining performance relationships between the DBMS, the indexing strategy, and the topology. This benchmark is based on the idea that the speed of a DBMS should not be the only performance

measure. For example, suppose the DBMS consumes fewer resources than the application can use them. Ultimately, the entire system is faster and not just one component.

8.1 Fundamentals

There are several existing SPARQL benchmark scenarios. The simplest benchmark type consists of three steps: data generation, data import, and the evaluation of the query performance.

The most widely used benchmark for this principle is the Berlin SPARQL Benchmark (BSBM) [17]. The benchmark simulates an e-commerce system with many products and functions. The benchmark changes the queries' constants to prevent DBMS caching. In this way, queries for many products and functions are simulated. Another benchmark is the Lehigh benchmark [61]. Here, a university domain ontology is simulated, consisting of courses taught by professors and attended by students. The primary purpose of this benchmark is to compare reasoning strategies based on the defined ontology. Finally, the SP^{2}B benchmark simulates Digital Bibliography & Library Project (DBLP) data [139]. The main component of the dataset is the relationships between articles, journals, and authors. Due to the simple data structure, this benchmark can compare join order optimizers and the execution of joins and filters.

More realistic benchmarks use static data streams with timestamps instead of a single static data import. These data streams can either be actual sensor readings or generated. The *SR* benchmark uses the SSN ontology [175]. The data describes actual weather measurements since 2002. A SPARQL Performance Benchmark (SP2Bench) is a synthetic benchmark [126]. The data model describes a social network. The benchmark contains three primary data streams: locations, posts, and image upload notifications. The *City bench* data corpus contains sensor data from a city in Denmark [5]. The benchmark contains traffic counters, such as available parking spaces.

However, all these existing benchmarks send data to a single central DBMS instance with a running SPARQL endpoint. None of these benchmarks consider distributed DBMS or network topologies. The sensors and the DBMS would be in different locations in the real world. These geographically distributed locations make a big difference to the DBMS as the differences between indexing strategies become much more relevant. Therefore, a centralized coordination device is a significant disadvantage. Data distribution must start at the data generation stage to compare indexing strategies based on other metrics, such as network load caused. Especially

in the IoT context, DBMS strategies requiring less data restructuring should be preferred because they reduce network load.

All existing benchmarks could be modified to evaluate distributed DBMS. However, in the case of actual data, the data must be heavily preprocessed to be streamed to a distributed DBMS over time. In the case of generated data, the entire data generation process must be rewritten so that these generators can produce parallel data streams. Therefore, this new benchmark scenario that generates data in a distributed manner is proposed.

8.2 The Benchmark Scenario

City bench inspires the definition of our benchmark scenario [5]. The *City bench* data corpus contains, among other things, actual parking occupancy data. On the one hand, this actual data enables the simulation of a realistic environment. On the other hand, it complicates the adaptation of the scenario to different topologies. The sensor data follows a fixed data structure, meaning the structural difference between actual and generated data is negligible. Therefore, this benchmark defines virtual sensors that generate synthetic data instead of actual sensor data, similar to *Citybench* 's parking lot occupancy data. Each virtual sensor periodically sends its current parking lot occupancy status to the closest DBMS. Since this benchmark is more than just sending data to a DBMS instance, the configuration must consider several properties.

8.2.1 Setup Script

Together with SIMORA, a script is provided that simplifies the topology creation. The benchmark topology creates a separate parking area for each full DBMS. The *size* parameter defines the number of these full DBMS instances. Within each parking area, there are ten parking spaces by default. Each of these parking spaces, in turn, has a sensor attached to it, which generates five data packets. The final select queries wait until all sensors have sent all data packets to obtain a consistent result. It would be possible to start the read queries while the sensors operate. However, this would result in the DBMS having a different state for each topology. Therefore, it would be much more difficult to compare the results of the different topologies. This setup of a nested topology can be seen in figure 8.3. Additional devices are added there for the network to be connected to both the random and uniform networks. These devices would also be necessary for an actual city, as shown in figure 8.2. Two additional devices are added for each complete DBMS

in the generated topology. Depending on the configuration, these additional devices may also run DBMS instances. However, these additional instances may only participate in query processing, not data storage, to simulate further heterogeneity. The spacing between devices is such that each device has an average of 10 direct neighbors. In ring-shaped topologies, additional devices would increase the communication volume by a constant factor. In a fully meshed network, it also makes no sense to add additional devices because these other devices would never be on the shortest path between two devices. At the time of the last read request, there were $d \cdot (15 + s \cdot 10)$ triples in the DBMS, where d is the number of DBMS instances and s is the number of measurements per sensor. Figure 8.1 shows some statistics about the generated scenarios.

databases	databases without store	sensors	sensor samples	triples
4	8	40	200	2060
16	32	160	480	8240
128	256	1280	6400	65920

Figure 8.1: Statistics about used topologies. DBMSs without stores are only available for topologies that are random and uniform.

In addition to the number of DBMS instances, the configuration script can be used to set other properties that cannot simply be derived from a single number. The routing algorithm and the topology layout are the options that relate directly to the network topology. There are a few more options that relate to the DBMS itself:

- Where is the operator mapped to a device?
- Are there DBMS instances without storage on the intermediate devices?
- What type of multicast will be used?
- What index strategy should be used?

All these properties are necessary to compare all combinations with each other. Since the configuration script creates a JSON file, fine tuning can be done later.

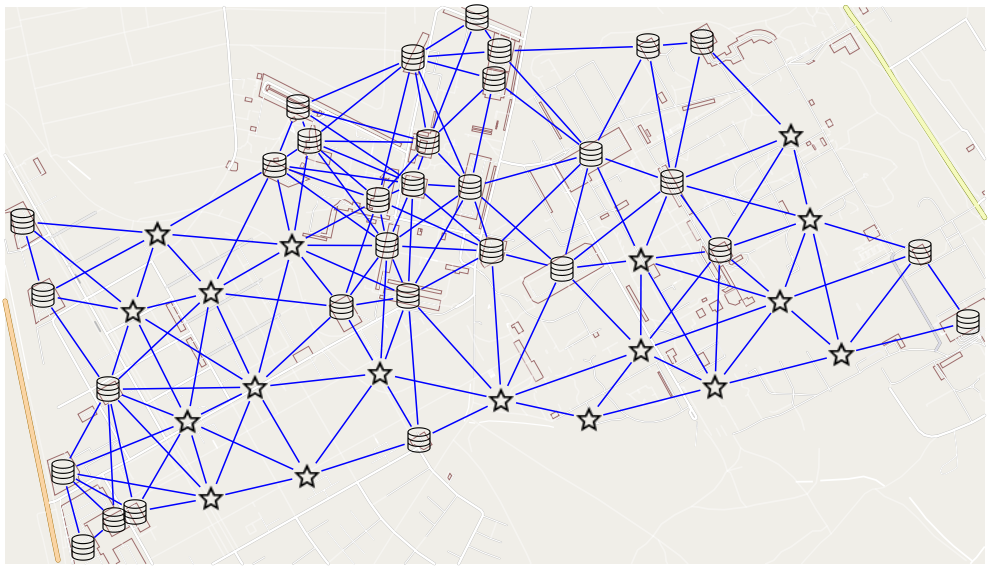
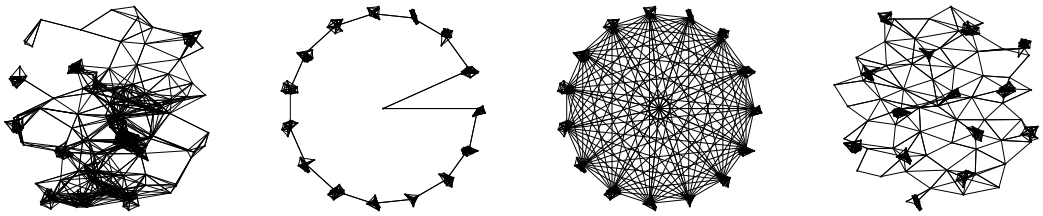


Figure 8.2: Map of the city around the campus taken from OpenStreetMap [65]. A DBMS instance is placed in each parking area. The stars are used as additional devices required to connect the whole network. Like figure 2.13, the DBMS can be further classified as having or not having an attached storage.

8.2.2 Ontology

The binary data of the sensor must first be converted into a graph form before it can be inserted into the SW DBMS. The SOSA ontology's *Sensor*, *ObservableProperty*, and *Observation* classes are used as a basis [32]. *Blank nodes* are used for the subject, giving the DBMS more options for later optimization. This freedom comes from allowing the DBMS to freely assign *blank nodes* so that all data can be stored locally without communication. The *Sensor* and *ObservableProperty* classes contain static information about the location of the parking lot and are, therefore, only initialized once. The *Observation* class contains the actual measurement data, such as the occupancy status of a parking space and the time of measurement. Another reason for using many *blank nodes* is that the variable data in this way contains only ontology vocabulary and primitive values. This choice means that the use of the distributed dictionary and the associated traffic can be significantly reduced. With this definition, the properties of SIoT DBMSs can be compared during computation without the data output being a significant part of the communication.

The structure of the generated data can be seen in figure 2.5.



(a) Random (b) Ring (c) Full Mesh (d) Uniform

Figure 8.3: The topologies as the topology generation script define them. The dark-appearing zones are parking areas. The center device and the devices in the parking areas run a DBMS instance. The other devices in the parking areas contain sensors. The remaining devices are just filling devices to connect to the network.

8.2.3 Queries

This benchmark defines nine SPARQL queries on this topic to simulate a variety of realistic queries against the data corpus. These queries can be found in part IV. In addition, different SPARQL features are used to evaluate various parts of the DBMS:

- At the beginning, figure A.1 shows how the data is inserted into the database. Therefore the insert query is shown so the reader can understand how the data is shaped. The particular idea is that all subjects of sensor samples are *blank nodes*. This structure allows some advanced data placement strategies.

- Then *Q1* (figure A.2) selects the entire dataset. This query does not contain any special operators which could be optimized. Therefore it can be used as a baseline for how the DBMS instances interact.

- *Q2* (figure A.3) retrieves a list of all parking areas. The result size of this query is independent of the number of sensor samples.

- *Q3* (figure A.4) counts the number of parking spots in the parking area 9. Compared to the previous query, another triple pattern is added, so now join ordering needs to be considered. However, the join is star-shaped, so the differences between different orderings have a low impact.

- *Q4* (figure A.5) counts the number of samples from a specific parking spot. This query adds path joins to the query. This path prevents

plans, consisting of only merge joins, from being used and thus requires a good join ordering.

- *Q5* (figure A.6) finds out when the last sample from a specific sensor was sent. It is the first query in this benchmark, which contains two star-shaped joins connected by a path. Here the query optimizer must detect the shapes correctly such that the joins can benefit from partitioning and local executions whenever the indexing strategy allows it.

- *Q6* (figure A.7) asks for the state of every parking spot in an area and the timestamp of its last measurement. A nested SELECT clause introduces a GROUP BY clause in the middle of the join tree. This structure complicates the join order, making obtaining good statistics harder.

- *Q7* (figure A.8) asks for the state of every parking spot in multiple areas and the timestamp of their last measurements. A FILTER operation is introduced, which can affect cardinalities within or before the join tree.

- *Q8* (figure A.9) counts the number of free parking spots in a specific area. Here two GROUP BY clauses are nested within each other, containing joins. The problem with the join ordering algorithms is the less reliable cost estimation.

- *Q9* (figure A.10) joins everything together to showcase many joins. Finally, three star-shaped joins are connected by multiple bidirectional paths. The idea is to complicate the query without restricting the join order optimizers' possibilities. Indeed later evaluations will show that this enables advanced optimizations.

8.2.4 Expandability

This scenario can be extended later in several ways. For example, the sensor devices could add more static data and measurement properties. This static data would increase the data generated for each sensor. In addition, it would be possible to use different types of sensors simultaneously, with each sensor sending another data format.

8.3 Summary

Current well-known benchmarks insert their data into a single DBMS instance. However, this hides the differences between the various partitioning strategies, which only occur when the DBMS is used in a distributed manner. In the newly proposed benchmark framework, all sensor data is dynamically generated and distributed, allowing the benchmark to adapt to any topology. Moreover, this is the only way for the benchmark to exploit a distributed DBMS's features fully. Combined with SIMORA, the runtime of queries on different DBMSs and indexing strategies can be compared alongside how much data is sent for this purpose.

Chapter 9

Topology Driven Operator Graph Distribution

The network's topology can play an essential role in a distributed scenario. In High Performance Computing (HPC), the servers are often connected by fat tree networks [129]. As a consequence, not all devices can communicate at full speed at the same time. In IoT, topology can take very different forms from the perspective of individual devices. There is already research that considers heterogeneous network topologies [103]. However, the most crucial consideration is to compute as much data as possible on connected devices. Current partitioning algorithms are based on the assumption of homogeneous hardware. Since this is not the case in sensor networks, DBMSs should not be operated in sensor networks without modifications. A particular problem here is that the transmission paths are different. Basic assumptions such as that join operators should be performed on devices with more data become invalid. All data flow paths must be considered to determine the optimal processing path.

Query distribution optimization strategies can be classified similarly to routing protocols. This classification can be seen in figure 9.1. The data must be stored, for example, by using a DBMS to access more than just the sensors' current values. In the context of the IoT, DBMSs are confronted with many different sensors and, thus, many different data formats. The schemas of an RDBMS need to be more flexible for this variety of data formats. Therefore, in the SIoT domain, the information is encoded in triples because this encoding allows the combining of arbitrary data sources. Additionally are standardized ontologies such as SOSA [32]. These ontologies allow us to handle a zoo of heterogeneous devices and support their data formats [55].

There are already several research efforts on SIoT DBMSs [172, 133, 82, 67, 70, 134, 64, 143]. However, there are many open challenges in the dis-

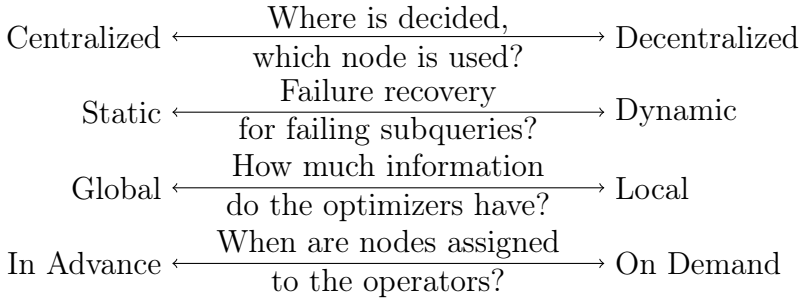


Figure 9.1: Classification of distributed query optimizers.

tributed processing and storage of data in the SIoT context. Modern sensors send their measured values to a central computer cluster over the Internet [72, 176, 35]. This centralization makes it impossible for applications to guarantee low latencies because the data must be transmitted over the Internet. In addition, all data from the sensor network must be sent externally, which can lead to security and privacy issues. Since only a few devices are directly connected to the Internet, a few devices have to forward many more messages, which means that the available energy in these devices is consumed more quickly.

The current DBMS assumes that the hardware is homogeneous in the initial optimization. Since this is not the case in sensor networks, DBMSs cannot operate in sensor networks without modifications. Keywords such as homogeneous hardware and server cluster imply that the pairwise communication latencies between all DBMS instances in the network are assumed to be the same. Again, this is different in a physically distributed network. A particular problem here is that the transmission paths are different. Basic assumptions such as that join operators should be performed on devices with more data become invalid. All data flow paths must be considered to determine the optimal type of processing.

9.1 The Approach

The central task of all routing protocols is the ability to calculate an efficient route to a given destination. Many routing protocols, such as RPL in storing mode, use a local routing table to accomplish their tasks. In this chapter, the routing protocol is combined with the query distribution optimizer of a DBMS. Therefore, additional information is stored in the routing tables to determine the following DBMS instance on the way to the destination. The widely used RPL protocol has been extended to include the position of the

following DBMS instance in the DIO messages. This change could also be applied to any other routing protocol.

Since routing is essential for the remainder of this work, these changes and their effects are explained in detail. A simple network topology is shown in figure 9.2.

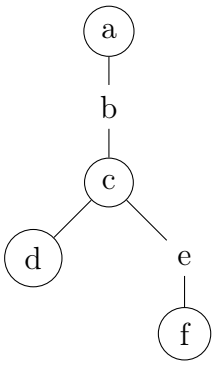


Figure 9.2: Example network topology. The nodes with a visible circle around them are edge nodes with a locally running DBMS instance. The others are just routers.

9.1.1 Extended Routing Tables

Since routing is an essential requirement for this chapter, the changes and their effects are explained in detail. An example of the extended routing tables can be seen in figure 9.3.

	a	b	c	d	e	f
a	a(a)	b(c)	b(c)	b(c)	b(c)	b(c)
b		b	c(c)	c(c)	c(c)	c(c)
c			c(c)	d(d)	e	e(f)
d				d(d)		
e					e	f(f)
f						f(f)

Figure 9.3: The routing tables as RPL uses them. The first column indicates which device owns the row. The first row shows the destinations, and the values indicate to which next hop the message should be forwarded. The values in the parentheses are added to indicate the next hop with a DBMS instance.

If there is no entry for the next DBMS in the routing table next to the next hop, then there is no DBMS on the way to this node. These empty fields in the routing table are irrelevant because, from the DBMS's point of view, it would make no sense to ask the routing protocol for the next DBMS on the way to any device. The DBMS only asks for the next DBMS towards another DBMS peer.

If an empty cell is in the routing table, the original routing protocol forwards the message to the parent node. In these cases, it would be problematic if the routing protocol is asked for the next device with a DBMS since it does not know it. An instance of the DBMS is placed on the DODAG root node defined by the RPL to solve this problem. If all complex queries are entered into the DBMS at the DODAG root node, then all next DBMSs along the entire path to the leaves of the network tree are always known. Another solution would be to use multiple DODAGs in parallel - one DODAG for each DBMS. There are already examples where multiple DODAGs are used in parallel [99]. Whenever the next DBMS is unknown, a message can be transmitted directly to the final destination, which means some optimization potential is unused.

9.1.2 Query Distribution

Figure 9.4 shows the differences between the operator placement and join order optimization strategies. Figure 9.5 shows the processing pipeline of a SPARQL query. This entire chapter explains the details.

First, the SPARQL string request must be converted into an operator graph. Therefore the string is tokenized and undergoes a logical optimization process. In state-of-the-art, the definition of the final join order is part of this process. This strategy is called the static approach in the following. However, it is only possible to accurately compute the best join order by calculating it. Hence, the result of the join order optimizer is always only an approximation of what is likely to be better according to the optimization goal. The optimization goal is often defined as the estimated execution time, but other metrics, such as the network traffic, are also achievable goals. In figure 9.4, this is shown as *static (a)*. In addition, some optimizers rely on variable name patterns, shown as *static (b)*, to reduce the network traffic and latency during optimization.

information used			strategies			
where	which	when	static (a)	static (b)	routing	routing assisted
coordinator	cost statistics	on query received	yes	yes		yes
coordinator	join shapes	on query received	implicit		yes	yes
every node	network topology	on query-part received	optional	optional	optional	optional
every node	relocate operators	output > input				

Figure 9.4: Short overview about which and where information is used to change the join order or the location of execution.

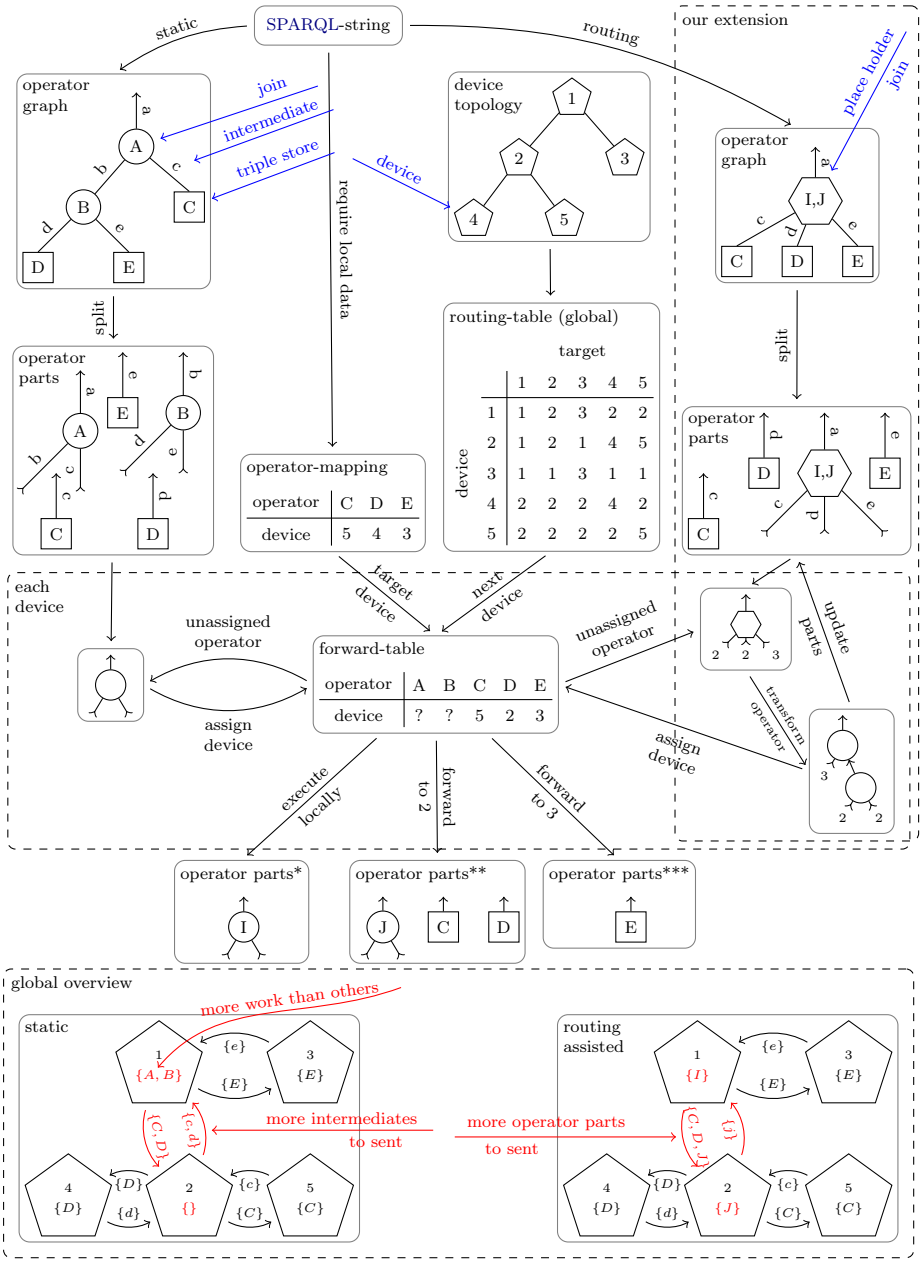


Figure 9.5: Routing-assisted join order optimization in comparison to state-of-the-art static join order optimization.

A placeholder join operator indicates that multiple inputs should be joined without specifying which order to let the routing choose which join order should be used. For example, a join placeholder-tree is used to avoid cartesian products. Therefore, only the identity of variable names in the query must be considered, such that the CPU overhead is negligible. We call this the *routing* approach.

Even if the state-of-the-art join order optimizer returns an explicit join order, several join orders are often so similar that the optimizer makes a random choice among similar results. We take advantage of this random decision. Instead of choosing an arbitrary join order, the placeholder join operator is used. We call this the *routing-assisted* approach. The figure does not explicitly show this because it combines *static* and *routing* approaches.

No matter how this operator graph looks, it is split into many parts, one for each operator. At each point where the operator graph is split, matching send and receive operators are added, which are indicated by arrows in the figure. These send-and-receive operators ensure that the operator graph can be executed in a distributed manner.

Afterward, these parts must be assigned to the devices. The operators for triple store access must be executed on specific devices since they require local data. In the figure, this is displayed by the operator-mapping table. All other operators could be executed anywhere. The local routing table of the current device is used to shorten the data paths. In the example, device one is only connected to devices 2 and 3. Therefore, it cannot send all triple-store operators directly to their destination. Therefore, for each triple-store operator, the next device through which it must be routed to reach its destination is computed. This functionality must be used inside the routing protocol as well. Therefore, it is helpful if the routing protocol exposes these functions.

An operator's inputs are considered to check for the other operators where to send them. If all inputs are sent to the same device, that operator is also sent there. This procedure is repeated until a destination is assigned to each operator. Operators whose inputs are sent on different devices are calculated locally on the current device. In the case of the placeholder operator, the placeholder is split whenever different devices are directly involved. This step is repeated on each device to which operators are passed until all operators are to be executed locally in terms of that device. In this way, the operator graph is adapted to the topology.

During or after the evaluation of a join operator, the possibility to undo the decision of sending the operator further down the network whenever the join result is larger than its inputs is added. In this case, the input streams and the operator are forwarded to the parent DBMS instance. This

relocation increases the CPU workload because the partial join has to be calculated again, but this reduces the network load, as fewer data needs to be sent. This extension can be applied to every operator placement strategy. The effect of this extension for all approaches is shown in the evaluation.

In the presented example, there are two joins. From the overall network's point of view, an additional operator must be sent due to the routing approach, but a stream of intermediate results can be omitted. As for the message size, an operator has a fixed number of bytes. However, the size of a data stream always depends on the number and structure of the stored data. So, the approach always yields better results, provided the joins have similar selectivities. This requirement is also why a pure routing-based join order optimization is not considered in the evaluation. In this case, the join order is not influenced by an estimation of the selectivities. Consequently, the used join orders can and will produce substantial intermediate results so that the queries no longer terminate in a reasonable time.

9.2 Evaluation

The configuration generation script with the size parameter 128 generates the topology used in the evaluation. In this evaluation, the longest measurement took 26 minutes for a benchmark case with a large ring topology where the indexing algorithm sent the data over long distances.

The number of bytes sent in the following figures appears large for such a small number of triples. This high number is because the message size is counted each time a message moves from one device to another, which can happen many times for a single message depending on the topology. As a result, self-messages do not contribute to the number of bytes sent.

All data is first entered using sensors to obtain fair and reproducible results. Then the read queries are performed. The simulator and also the DBMS allow simultaneous execution. Still, it is much more challenging to analyze the numbers because each query would give different answers for each scenario since the timing of the read and insert queries are different.

9.2.1 Insert Queries

First, the behavior of different indexing strategies during the INSERT phase is compared.

Looking only at the amount of data sent, grouped by topologies, figure 9.6 shows that the fully connected network sent the least. In reality, however, a fully connected network is unrealistic. The ring network performs well when

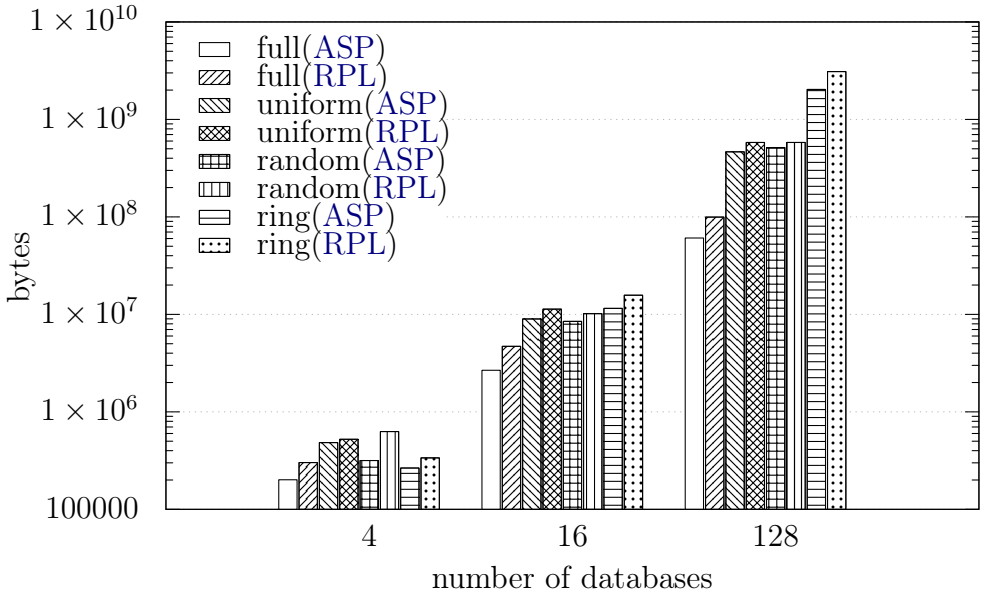


Figure 9.6: Several bytes are sent during INSERT by topology and routing protocol.

there are a small number of devices but performs worse when there are more devices. In this evaluation, the uniform topology behaves similarly to the random topology, especially for more extensive networks.

Next, the influence of the routing protocol on the data traffic is considered. This influence is shown in figure 9.6. In the case of the RPL protocol, a single DODAG tree is used where data must take longer paths to its final destination. All routing tables for the shortest path are created using the Floyd-Warshall algorithm.

Figure 9.7 shows the sent data volume grouped by topology, several DBMSs, and an indexing strategy.

The ontology must be loaded in all cases, and the sensor data must be sent from the sensor to the nearest DBMS instance. In addition, during initialization and simulation, some static data must be sent regardless of the indexing strategy.

There are many different hashing strategies, each with different strengths and weaknesses. For example, in the centralized strategy, there is no data distribution at all. Instead, each triple is stored in the same DBMS instance. However, all triples must be sent to this central DBMS, which causes more network activity than if the data were stored locally. In addition, the more replicas an index uses, the more data must be sent when writing to the index.

In key based hashing, different combinations of columns are hashed and

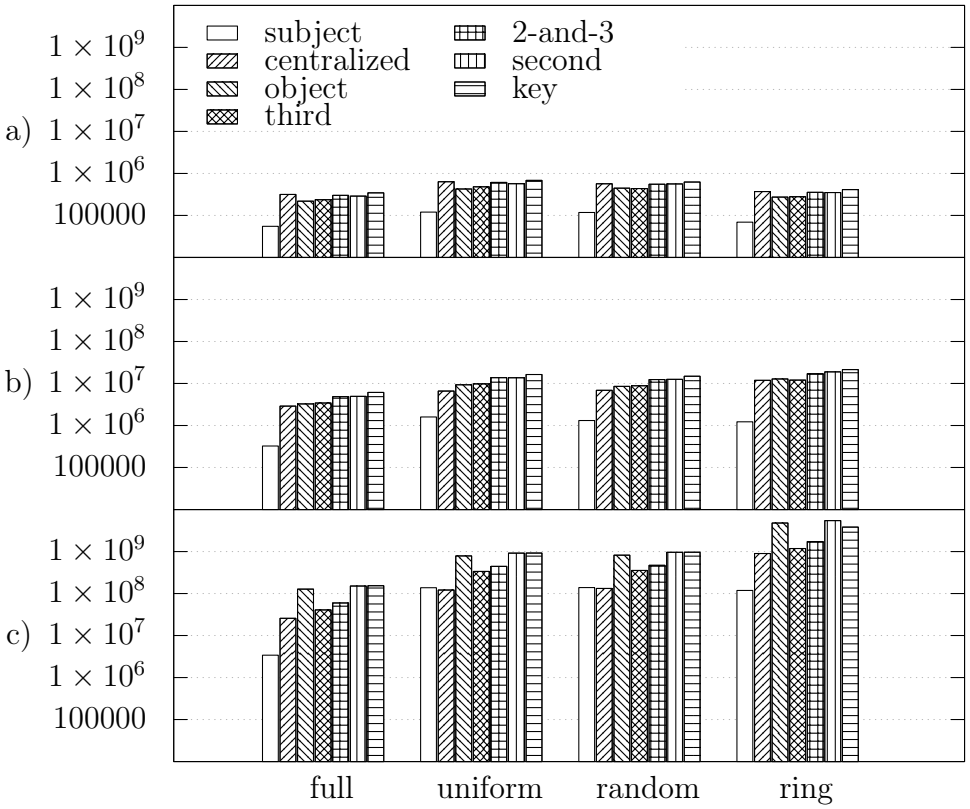


Figure 9.7: Several bytes are sent during INSERT by topology and hash strategy. From top to bottom, the results for a) 4, b) 16, and c) 128 DBMSs are shown.

distributed. Because of the different subject, predicate, and object subsets, other values are included in the hash. As a result, this hashing strategy has sent the data to many different devices. This strategy can improve query speed and data security but also increase traffic. With increased DBMS instances, data traffic is among the most expensive indexes.

Among the indices based on specific triple pattern parts, partitioning by subject with a factor of about 10 is the best strategy in this benchmark. The reason is that almost all triples have *blank nodes* at the subject position. When these *blank nodes* are hashed, everything is stored locally on the device that receives the insertion request. Therefore, partitioning by object is not helpful for this benchmark data. Partitioning by the second or third column, as defined by sort order, is also poor in combination with this data layout. Partitioning by multiple columns achieves the highest level of data security since most triples are stored on six different devices. However, it increases

the data traffic considerably.

9.2.2 Read Queries

The benchmark in chapter 8 defines nine queries that read from the triple store. The benchmark queries have been illustrated and explained in figures A.1 to A.10.

The randomized topology is used for all benchmarks because it is the most realistic. In addition, the other standard simulator topologies are unsuitable for the intended comparisons.

Figure 9.8 shows the results of the benchmarks. This figure layout is entirely new. Therefore, the layout is explained first. All combinations of all the options shown in the legend are executed to create this figure. Then the total network traffic can be calculated. This number of bytes evaluates to 100% in the innermost circle. The remaining option with the most significant impact is always chosen from the inside to the outside. This subset is the new 100% for the further ring segment below this option. From the center of the circles outward, the graph shows the decision trees designed to show the essential options and how much influence they have compared to each other. When the differences caused by the remaining options are close to zero, then no further circle segments are shown. Consequently, only effective options are shown, leading to the partial disk shapes in the graph.

The figure shows that the amount of data is almost always in the middle of the circles. This option determines how many data samples each sensor generates, which scales the data in the DBMS. Of course, in productive environments, how much data is stored in a DBMS cannot be chosen. However, this does indicate how the DBMS will behave with different-sized data sets.

During initialization, the sensors generate data sent to the DBMS and stored there. Besides the obvious information that more sensor samples implicate more traffic, it can be observed that the DBMS partitioning scheme significantly changes the amount of data sent. When the keys from the hexastore index are used for data distribution, almost ten times more data must be sent than with subject-hash partitioning. Additional hops refer to DBMS instances involved in computations but not storing data. Even though these DBMS instances are not involved in storage, they can help send multicast messages with DC, which reduces network traffic by eliminating duplicates during the data send process.

Q1 selects the entire data. Since no join operators and nothing else could be optimized, the query only scales with the amount of data.

The result of *Q2* does not depend on the number of sensor samples, so this is the only query not affected by the amount of data. However, network

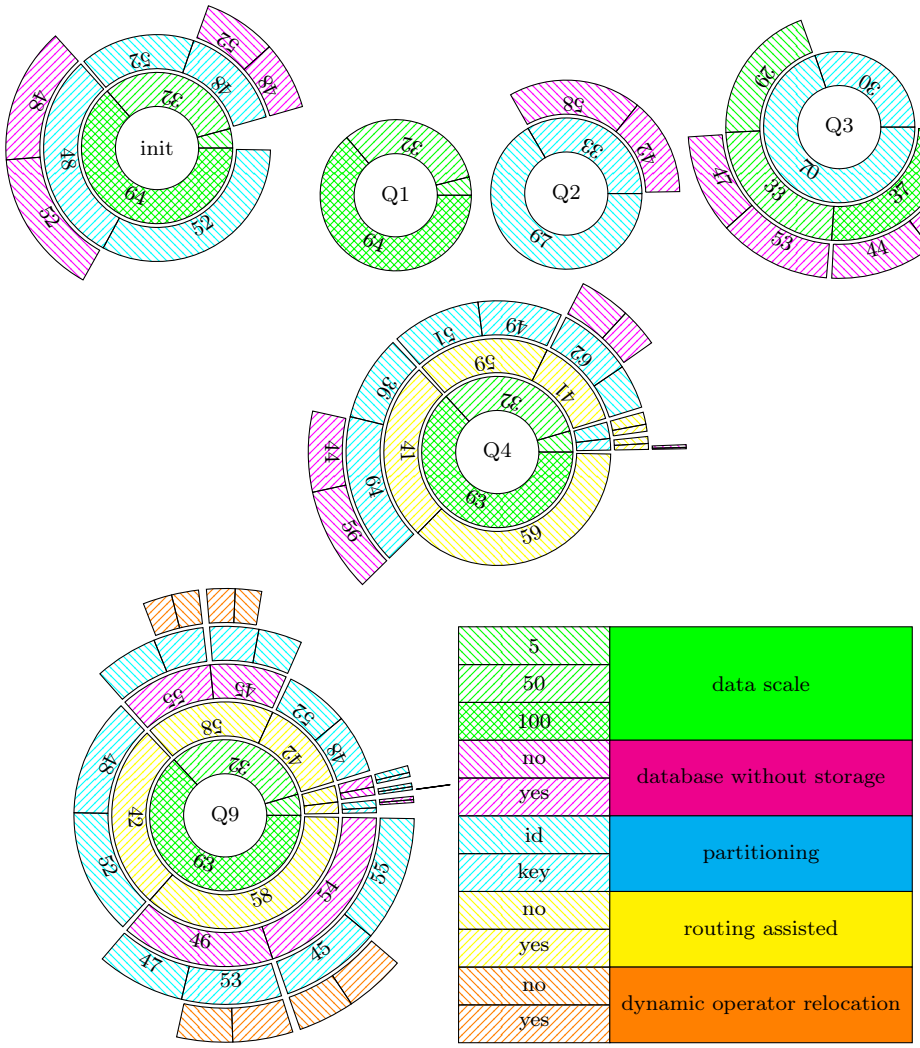


Figure 9.8: Transferred data for benchmark queries in percent. A smaller amount of data is better. Each circle shows which decision leads to how much data for which option. The decisions further inside the circle show the most significant effect.

traffic is strongly influenced by the partitioning used in the DBMS. If the data is partitioned by subject, then all DBMS instances must participate in the computation, whereas only a few are used with key-based partitioning. Key-based partitioning benefits from additional DBMS instances because the join operator can be evaluated earlier, reducing the amount of data sent over the network. In the case of key-based partitioning, the topology-aware join order generates less traffic than the topology-unaware optimizer. Together with *Q1*, this confirms that the visualization strategy works as expected.

Q3 is the most extreme example, where key-based partitioning significantly reduces the amount of data transferred. The small size of the data is related to the small number of DBMS instances involved in the result computation. While this reduces network traffic, it also increases the load on a few devices, which can be another problem in an IoT environment. In *Q4*, the routing-assisted join order can reduce network traffic in key-based partitioning. *Q5* to *Q8* do not show anything conceptually new. Therefore they are not shown.

We have added *Q9* to the benchmark. The query joins most triples together so that each sensor sample is converted into a single line. This results in a much larger result set with many more star joins. It can be seen that the routing-assisted join order reduces network traffic. Also, the dynamic relocation of operators can be used here whenever the result of a join is more significant than its inputs. The measurements show that this can reduce network traffic by up to four percent in the larger data scale setups and up to 23% when only a few triples are present in the store.

In summary, the topology strongly influences the amount of data sent. However, the topology is predetermined and cannot be changed by software in an actual application. The second most substantial influence is the partitioning scheme, which decides where the data is stored. This decision is crucial when inserting and retrieving data, as it affects where operator graphs must be sent and how far intermediate results must be transmitted. Like the topology, once the partitioning scheme is chosen, it cannot be changed unless heavy repartitioning is triggered, adding massive one-time network traffic.

Among the changeable things for each query is the routing-assisted optimizer, which adjusts the join order to match the topology. Some modified join orders are better, while others are worse than the static optimizer. Therefore, it is not optimal to always use one algorithm or the other. Whether or not the decision to use routing-assisted join order optimization is reasonable depends on the similarity of the estimated intermediate results. The routing-assisted join order will likely improve the result if the estimate is similar. In contrast, significant differences in estimated cardinalities should be statically optimized to reduce overall traffic and computation time.

9.3 Summary

Modern DBMSs already minimize their network traffic to increase their execution speed. However, these DBMSs rarely consider the path that the data travels. This path is critical because a larger piece of traffic traveling a short path is better overall than a small piece of data traveling a very long path. This work proposed combining DBMSs and routing protocols to solve this problem. Furthermore, this strategy can be executed in a fully decentralized manner with a relatively small overhead during the initialization of the routing protocol.

Part III

Machine Learning Based Join Order Optimization

Chapter 10

Machine Learning

Join order optimization is a central and complex problem in modern DBMS [73]. The problem of selecting a good join order becomes more complicated when there are more inputs since the number of possible join trees is $\frac{(2 \cdot n-2)!}{(n-1)!}$ [52], where n denotes the number of inputs to be joined.

While in RDBMSs, the number of tables and, thus, the number of joins in SQL queries is relatively small [101, 177], the number of joins in SPARQL queries is typically much higher due to the simple concept of triples in RDF. For example, reports of more than 50 join operators exist in a practical application [59]. Other applications also require a high number of joins [121, 120]. Therefore, when the number of joins is high, join order optimization becomes much more critical in SPARQL queries. Other studies report that the join operator is one of SPARQL's most commonly used operators [19].

Due to many possible join sequences, several strategies have been developed to deal with this problem.

For example, there are several variants of greedy algorithms [6, 94]. In this category of algorithms, the goal is to select two relations as quickly as possible that yield the fewest intermediate results. This procedure is then repeated until all inputs have been merged. In practice, however, it may sometimes be faster to compute a slightly larger intermediate result if a faster merging algorithm can be used.

Another strategy is dynamic programming [6, 94]. This strategy assumes that the optimal solution can be constructed from the optimal solutions of the subproblems. Consequently, it is unnecessary to traverse the entire search space of the join tree, but only a portion of it, without missing the desired solution. Another issue in optimizing the join order is that not only the execution of the query plan but also the optimization of the query takes time [59]. This issue is also shown in figure 10.1.

Therefore, a balance must be found between the time required to find a

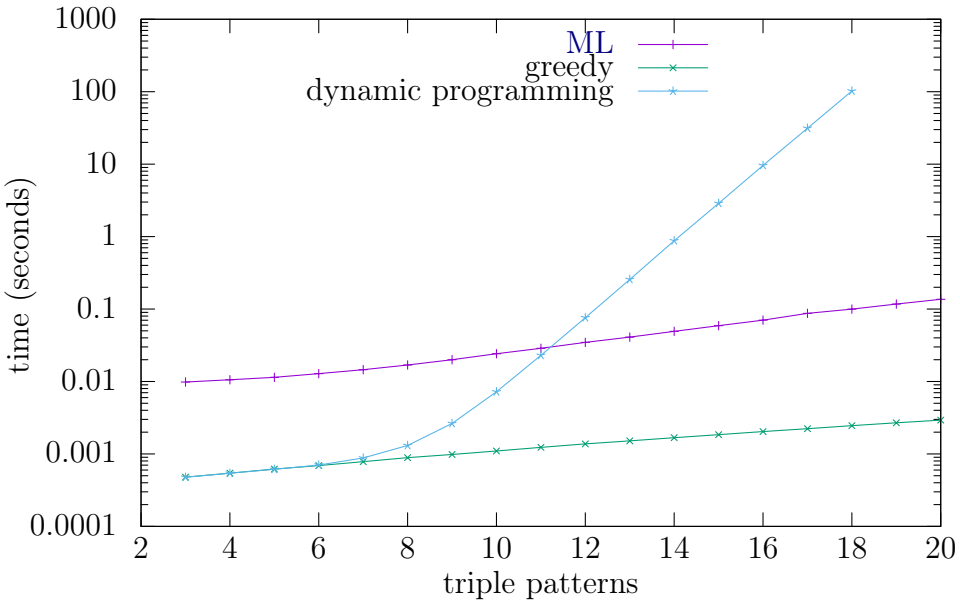


Figure 10.1: The time needed to construct an optimized join tree with a given number of inputs to join in the LUPOSDATE3000 DBMS. The optimization of queries was repeated until at least 60 seconds were spent for each join size, then the average time per query was calculated.

good plan and the time needed to execute the plan. For example, greedy algorithms have a shorter planning time, but the execution is often longer. Conversely, dynamic programming often suggests a better query plan but takes more time, especially for many joins in SPARQL queries.

There are already approaches that use ML in optimizing join order [157, 107, 174, 47, 62, 71, 73]. The idea is that a machine-learning approach only requires a little time to evaluate a given model. The time required is shifted to a separate learning phase. A learned model can be reused for many queries if the data does not change significantly. The advantages of a short optimization time can be combined with a short execution time, provided that the estimates are reasonable. However, these existing approaches only work for SQL DBMSs [157, 107, 73]. The remaining approaches only estimate the runtime of a query without optimizing the query itself [174, 47, 62, 71]. This chapter presents how ML can optimize the join order in SPARQL queries. The focus is on optimization for a large number of joins.

10.1 Related Work

This section presents several existing machine-learning approaches related to the proposed algorithm.

10.1.1 Machine Learning Approaches

There are various approaches for optimizing SQL queries in conjunction with ML. However, research [174] has shown that such techniques cannot directly translate into SPARQL environments.

The Reinforcement Learning Join Order Optimizer (ReJOIN) approach [107] receives SQL queries and chooses the best join order from the available subset. The ReJOIN enumerator is based on a reinforcement learning approach. The backbone of ReJOIN is the PPO algorithm to enumerate the join trees. ReJOIN was trained on the Internet Movie Database (IMDb), which contains 113 queries [132]. Of these, 103 queries were used for training and 10 for analysis. This approach does not work for SPARQL because to capture tree structure data, they encode each binary subtree as a row vector of size n, where n is the total number of relations in the database. When SQL tables are translated into predicates of SPARQL, this vector would contain thousands of elements, mostly zero. They create a square matrix of size n for each episode to capture critical information about join predicates. When using large datasets, this would not fit into memory. When using smaller datasets, this may fit into memory. However, this would be too large to be useful as machine-learning model input.

Another work [157] uses reinforcement learning to choose a data storage structure for the data automatically. Suitable indices are also calculated. Finally, suitable join orders are specified for these indices.

Another article [170] explores join order selection by integrating both reinforcement learning and long short-term memory. Graph neural networks are used to capture the structures of joined trees. In addition, this work supports multi-alias table names and database schema modification. To encode join inputs, they use vectors of size n, where n is the number of tables. Additionally, they use a quadratic matrix of size n to encode what should be joined with each other. Similar to the ReJOIN approach, this fails for SPARQL due to the massive size of this matrix.

Another research group proposed a Fully Observed Optimizer based on the PPO Algorithm (FOOP) [73]. This work uses a data-adaptive learning query optimizer to avoid the enumeration of join orders. Thus it is faster than dynamic programming algorithms. Similar to the previous optimizers, they also use bit vectors which consist of one bit for every column in every

table. Additionally, their state is a matrix where one dimension equals such a vector, and the other is scaled with the number of joins in a query. The memory consumption is much better than in the other approaches, but still, it scales with the number of predicates in SPARQL, which is too large.

Other approaches [174, 47, 62, 71] predict query performance directly, detached from any data knowledge, by looking at the logs of previously executed queries. In their approach, the authors consider execution time as an optimization objective [71]. First, they encode the queries in terms of feature vectors. Then they measure the distance of the new feature vector to known feature vectors to predict the new execution time.

Some approaches [174, 62] focus only on predicting how long and how many CPUs will be used without explicitly computing the join trees. For this purpose, SPARQL queries are converted to feature vectors.

The proposed approach is unique to all other techniques because it outputs an executable join tree. Furthermore, during the generation of the join tree, the required memory depends on the number of joins, not the number of predicates. Finally, the proposed approach is optimized for SPARQL instead of SQL.

10.2 The proposed Algorithm

The algorithm consists of several parts. First, the algorithm is presented to generate the training and evaluation queries. Then, Reinforcement learning for Join Order Optimization in SPARQL (ReJOOSp) is presented and explained.

10.2.1 Generating Queries

For training and evaluating the ML models, the SPARQL [142] queries have to contain precisely a certain number of joins. The idea is that queries that contain only join operators make it easier to interpret the result. Unfortunately, actual queries are not helpful for ML because they are too similar. After all, they change only a few constants or are too different because actual queries can contain other SPARQL elements besides joins. Also, getting enough queries to use for training can be challenging. Finally, there are queries with many joins, but applications that use so many joins use fewer variations, so again, there need to be more queries.

The optimizer could adapt to these elements, but the outcome is harder to interpret for comparison purposes. Existing benchmarks use a small set of queries unsuitable for ML. Benchmark queries originating from templates

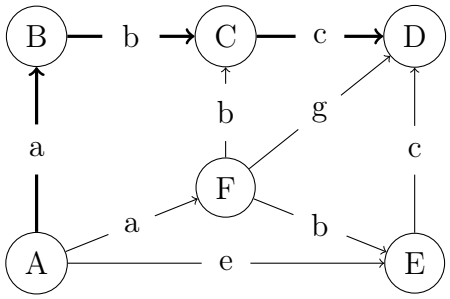


Figure 10.2: Example graph to show how queries can be generated.

are not considered because ML would learn the explicit query structure such that it can not adapt to different query templates. We experimented with several strategies to generate useful SPARQL queries containing exactly n triple patterns. We discovered that it is essential that the result set of each query is not empty. The problem with the empty result set is that ML prefers the join trees that quickly detect the empty result. In practice, almost every SPARQL SELECT query will return a result [19].

The best strategy is to think of the triples as a directed graph, as shown in figure 10.2. Then a random subgraph is chosen with n distinct edges in this graph. For this example, let us pick the bold edges AB, BC, and CD with their predicate A, B, and C, respectively. The algorithm is wholly randomized, so the chosen subgraphs can have any shape, including paths, star patterns, circles, and combinations. By definition, the generated query structure will always follow the shapes in the data, which means that actual queries could also use similar ones.

The target is that the model generalizes for arbitrary queries. Therefore different shapes were not considered independently of each other. Besides that, assuming that every predicate is used only once per subject, the join order does not matter if multiple star joins are compared with each other. When there are a few duplicate predicates, there might be slight differences between execution plans, but they are also not significant. When only path joins are considered, the best execution plan starts with one triple pattern and repeatedly connects the next section of the path to it. No ML is needed to find a good execution plan in this case. The only interesting queries contain a mixture of different patterns because the join order is nontrivial.

After selecting the subgraph, it is converted into a SPARQL query by replacing all node labels with variable names. The generated query for this example can be seen in figure 10.3. By construction, this guarantees at least one solution. Even though RDF stores allow for very flexible data structures, there are repetitive structures, so queries generated using this approach will

```
SELECT * WHERE { ?A a ?B . ?B b ?C . ?C c ?D . }
```

Figure 10.3: Example generated query.

```
SELECT * WHERE {
  ?article dc:creator ?author .
  ?article swrc:journal ?journal .
  ?journal dc:title ?title .
  ?journal swrc:volume ?volume .
}
```

Figure 10.4: Example SPARQL query.

likely return multiple results. The same applies to the real world as well. Some queries are intended to retrieve a few results like "Who published x?", and others are expected to return many results like "What is published by x?". The same applies to the generated queries. Some queries return one result, some return up to 20 results, and some return a considerable amount. However, since the query generation is randomized, this can only be guaranteed with a much more complex query generator, which would need more time to prepare the model's training.

10.2.2 The Machine Learning Algorithm

While the number of tables in SQL is small, the number of predicates in SPARQL is very high. This observation makes a significant difference in possible encoding for ML. In SQL, it is enough to say which tables should be joined. On the other hand, in SPARQL, the triple patterns must be explicitly connected, dramatically increasing the input size for ML and making it harder for the model to learn what is essential.

Figure 10.4 shows a simple example of a SPARQL query that retrieves the authors of an article along with the title and volume of the journal in which the article appears. A plausible approach would be first to connect the triple patterns (4) and (5), assuming that there are more articles than journals. Next, the triple patterns (2) and (3) could be joined, allowing a second independent star-shaped join. Finally, both intermediate results must be joined.

The *Stable Baselines 3* [75] framework is used because there are many more open-source contributors than in other frameworks. Especially there is

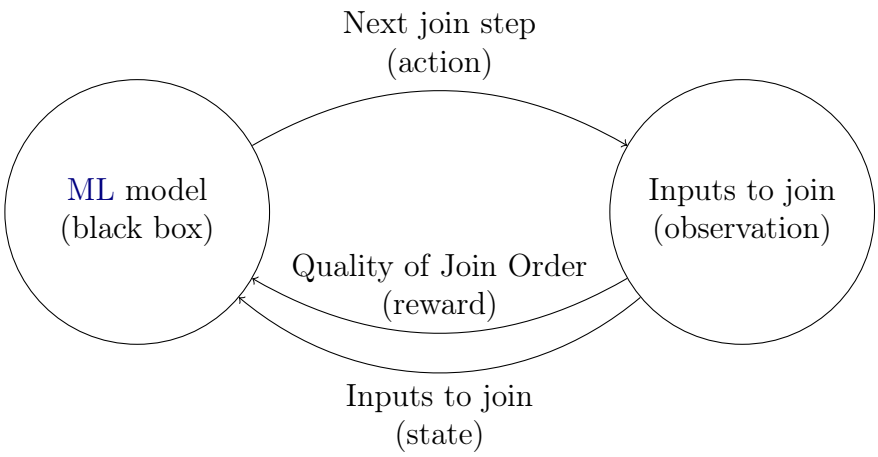


Figure 10.5: Reinforcement learning.

a community addon [77] extending the built-in PPO model with the possibility of masking. Applying masks to the available actions is crucial for the model to produce good join trees. We use the default MultiLayer Perceptron (MLP) policy with two layers and 64 nodes, also used in other join order optimizing articles [91]. Using more layers increases the number of learning steps required since the model must first learn how to use these hidden layers. We need to keep the training phase short. Otherwise, the changes in the dataset will immediately render the newly trained model useless.

We want to treat the ML model as a black box. This simplification allows us to focus on the encoding of the query and the calculation of a reward. Additionally, this simplifies many different experiments. A simplified view of the ML framework is shown in figure 10.5. The model's actions modify the environment and the join inputs. ReJOOSp exposes a reward to the model when the join order is complete.

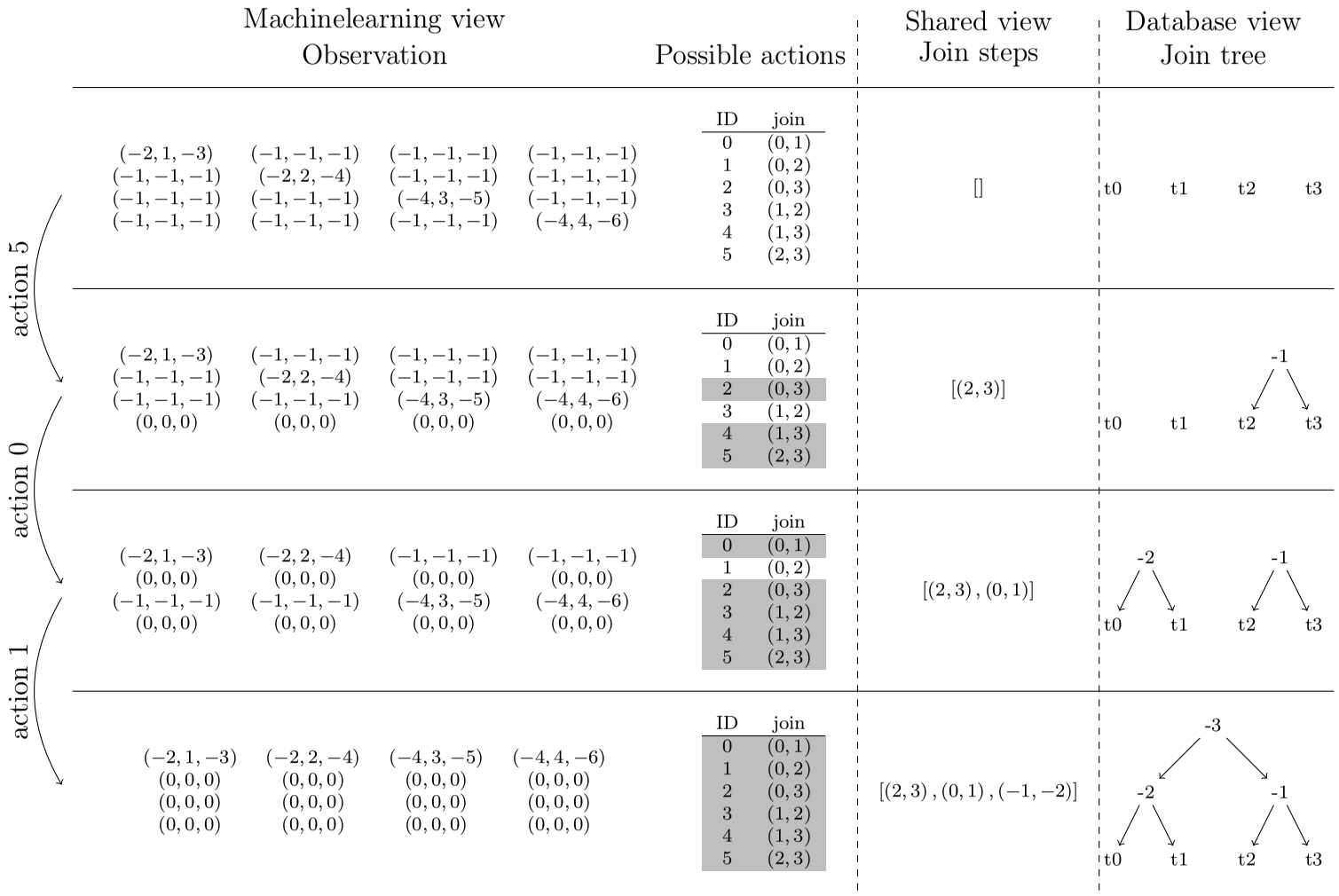


Figure 10.6: Overview of join order optimization

The internal state of the essential variables is shown in figure 10.6 to explain ReJOOSp in detail. This figure should give an overview of how the join tree is constructed. Next, the details of ReJOOSp are explained in the following. Each row in the figure shows a time step.

The essential step in the initialization is the definition of the observation matrix. This matrix is created from a transformation of the SPARQL query. First, the literals of the SPARQL query are replaced with numbers from a dictionary. This dictionary assigns a unique positive number to each literal, starting at 1. Since the same dictionary is used for every query, the model can learn the cardinalities of the query constants. Next, another dictionary is used for the variables for each query. Finally, the variables are assigned negative numbers to distinguish between literals and variables, starting with -2. Because the variables are always renamed to a sequence of negative numbers, the model can also learn variables' patterns in the queries to calculate the best join order. As shown in figure 10.7, the resulting number sequence can be obtained by applying the above transformations to the query.

This list is transformed into the diagonal of a square matrix to obtain the observation matrix. This matrix is necessary for the neural network to remember which inputs have already been connected. The remaining cells of this square matrix are filled with $(-1, -1, -1)$. The matrix size is chosen to represent the largest desired join. This choice means the matrix size is quadratic in the number of joins. Even if this seems to be a large matrix, it is still tiny compared to approaches from RDBMSs, where the matrix size is quadratic to the number of tables. The unused part of the matrix is filled with $(0, 0, 0)$. The purpose of the matrix is that each join is mapped to a sequence of numbers of the same size. The DBMS itself does not need this observation matrix.

The possible actions are defined as a list of pairs of rows in the matrix that could be joined. To further reduce the ML search space, the difference between the left and right input of the join operator is ignored. For the example in figure 10.4 with four triple patterns, this list of possible actions can be seen in the top row of figure 10.6 next to the observation matrix. In figure 10.6, the invalid actions are marked in gray.

$$[(-2, 1, -3), (-2, 2, -4), (-4, 3, -5), (-4, 4, -6)]$$

Figure 10.7: SPARQL query of figure 10.4 transformed into a number sequence.

The ML model now receives the matrix and the action list and selects a new action until only one action is possible. Then this last action is applied, and the join tree is complete. Finally, for each selected action, the step function is executed.

The step function first calculates which two inputs are to be joined. Then, the action ID is looked up in the action list to decide which inputs were chosen.

Then the observation matrix is updated. Therefore, the values from the second join input matrix line are copied to the first join input matrix line. All fields with values of $(-1, -1, -1)$ are skipped so that existing data is not overwritten. Finally, the entire row is set to $(0, 0, 0)$ from the right join input.

Then the currently selected join step is appended to the list of join steps. This list of join steps is the target of ReJOOSp, as it is the only variable passed to the DBMS at the end. The DBMS can use this list to build the join tree shown in the right column of figure 10.6.

This phase is done when the entire matrix contains only the value $(0, 0, 0)$ except for the first row. The matrix always ends up in this state, regardless of the chosen join order.

Once the join tree has been computed, a reward must be returned to the model. The reward is intended to reflect the quality of the join tree compared to other previously run join trees. Calculating the reward for joins with up to 5 inputs is easy as it is possible to traverse all the join trees and keep statistics on the good and the bad ones. However, since the model is intended to work with more significant joins, such as 20 join inputs, this approach will only work as it can collect statistics on all available join trees. Therefore, the default optimizer is used first to initialize the statistics for each SPARQL query. These statistics only map the query and the total number of intermediate results. These statistics do not contain specific join orders.

Furthermore, whenever a new reward during training is calculated, these statistics are extended with the intermediate results produced by the current join plan. All these statistics about queries and their generated intermediate result counts are stored in a separate SQL DBMS. These stats will continually improve as the model is trained. After the training, these statistics are removed since they consume much storage space. When the model needs to be retrained, these statistics would be outdated, so there is no reason to keep them after the training. The reward calculation function can be seen in figure 10.8. The variables v_{max} and v_{min} refer to the min and max of the intermediate results for this specific query. In contrast, $v_{current}$ refers to the intermediate results for the current join plan. Even if the actual best and worst case is unknown, this estimation is good enough to train the model.

$$r = \begin{cases} 0 & \text{if } v_{max} = v_{min} \\ -10 & \text{if } v_{current} = null \\ min\left(10, -\log\left(\frac{v_{current} - v_{min}}{v_{max} - v_{min}}\right)\right) & \text{otherwise} \end{cases}$$

Figure 10.8: The reward function uses v_{min} and v_{max} from the statistics, which refer to the currently known best and worst-case numbers of intermediate results. $v_{current}$ may receive a *null* value when it runs into a timeout. This *null* value is not considered when calculating the known worst case.

A logarithmic reward function is used along with a minimum to compensate for the potentially high variation between different join trees. Generally, good join trees should be rewarded, while those with very high intermediate result counters or timeouts are punished.

10.3 Evaluation

In this chapter, the quality of the ML optimizer is evaluated.

10.3.1 Environment

The synthetic SP^{2}B dataset [139] generator is used to produce an approximately 2^{20} triples dataset. This amount of triples is chosen because the data is large enough to show differences between join orders. At the same time, the dataset is small enough, so poor join orders still have a chance of being calculated completely. Additionally, the WordNet [155] real world dataset is used, which contains 2637168 triples. The queries are generated, as explained in section 10.2.1. All the queries contain only basic triple patterns, which should be joined together.

The training phase took between 1 and 20 hours, depending on the number of triple patterns to join. At the beginning of the training, the execution time is dominated by the timeout for a single query because the more joins are to be optimized, the higher the probability of choosing a terrible join order. When dealing with many joins, bad decisions during the join order optimization yield execution plans that never terminate. Every learning approach, which needs to execute some sample queries during training, will suffer from this same problem. Therefore, this is considered a short training time. The triple store size also influences the training time requirement,

mainly because poor join orders become very bad due to increased data for every join iterator.

First, the time needed to optimize the join tree is compared in figure 10.1. The speed of the optimizers only depends on the number of joins in a single query because all optimizers have the same heuristics available. The greedy optimizer is the fastest one, but the evaluation in the following sections will show that its generated join trees are bad. On the other hand, the dynamic programming optimizer needs much more time to optimize join trees. At approximately 11 join inputs, the ML-based optimizer is faster than the dynamic programming optimizer during the generation of the join tree. ReJOOSp is, therefore, faster during the optimization phase. The following sections will focus on the quality of the generated join orders.

10.3.2 Evaluating Different Numbers of Triple Patterns

The number of joins used during training is shown on the x-axis in the right half of figure 10.9. The y-axis shows the number of joins when evaluating the models. In the left part of figure 10.9, other DBMS's explain functions are used to compare their optimization capabilities with ML. *GraphDB* and *Apache Jena* only produce deep left join trees. The *Apache Jena* DBMS produces the worst query plans in the context of synthetic data, while *RDF3X* produces the worst results in actual data, as shown in figure 10.10. Despite this, the *RDF3X* DBMS uses bushy join trees, which could allow parallel processing of independent subtrees. However, this is insufficient in the *WordNet* dataset, as the join orders generated are terrible compared to all other optimizers. Next, two different join order optimizers are implemented in LUPOSDATE3000. First, a greedy join order optimizer always chooses the two entries with the smallest estimated result. Second, in LUPOSDATE3000, a join order optimizer for dynamic programming is implemented. Here the quality of the join order is better, but its performance drops drastically with a higher number of joins which can be seen in figure 10.1.

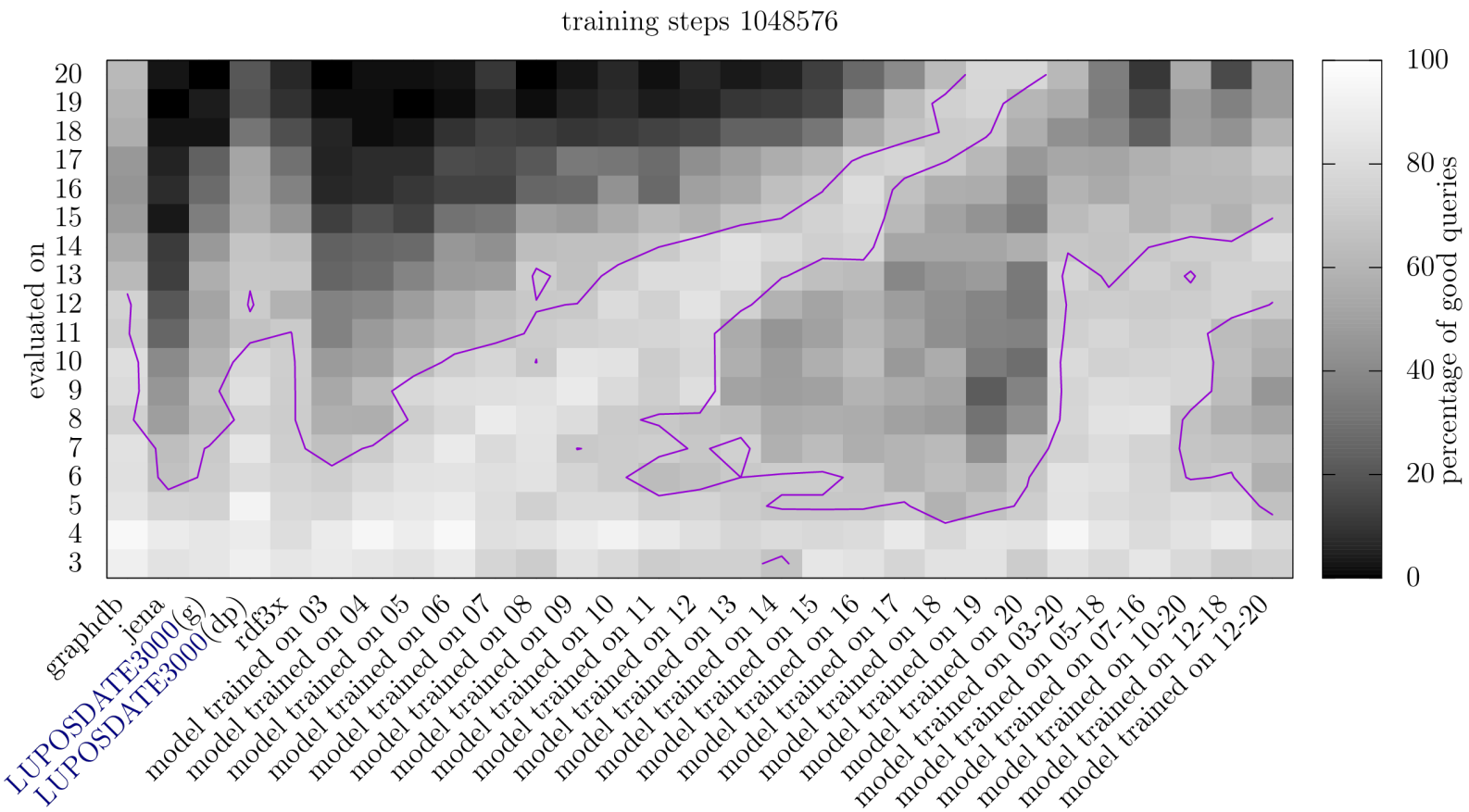


Figure 10.9: SP^{2}B-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70 % of the queries receive good join trees.

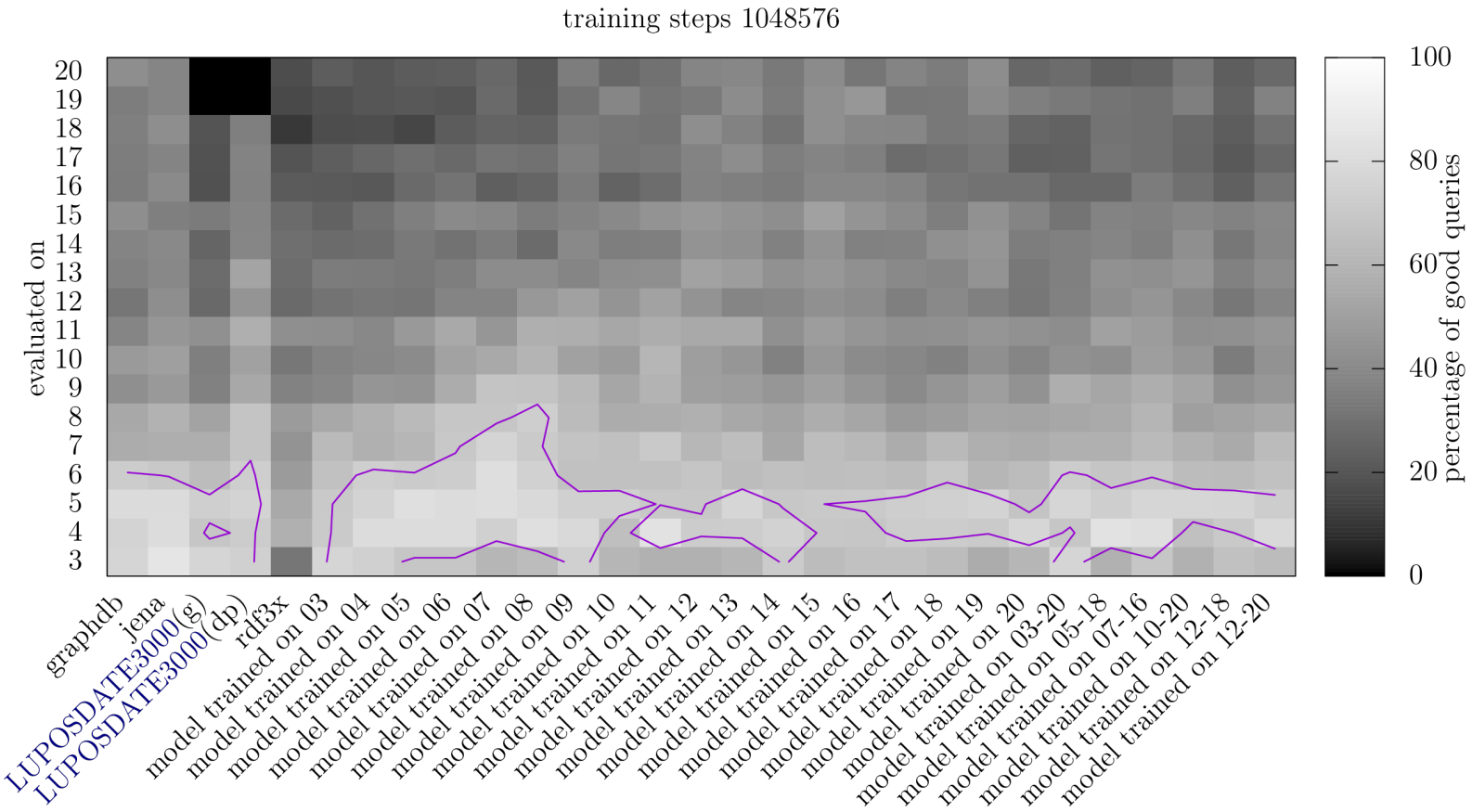


Figure 10.10: WordNet-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70 % of the queries receive good join trees.

None of the deterministic synthetic data approaches can produce reproducible good join orders for many triple patterns. None of the join order optimizers tested on actual data achieved good results for more than eight joins. Each trained model can optimize joins with up to six entries relatively well. On the right side of the diagram, different ML optimizers trained with varying numbers of join inputs are shown. As expected, these trained models provide plausible results over a broader range of inputs when used on synthetic data sets. In the context of actual data, the mixed models show no improvement over other models. However, these results are inferior to specially trained models in both data sets, meaning the user is better off training multiple models for the desired purpose. Next, the figure shows that the models perform better for the number of join inputs they used for training. While this result is obvious, each model can also optimize join trees of comparable size from $n - 3$ to $n + 3$ very well. In the case of synthetic data, this can be seen in the graph. The measurements show a similar result with actual data, which is much less noticeable in the graphic because the training effect is weak overall.

10.3.3 Evaluating Different Numbers of Training Steps

In figure 10.11, multiple models for queries with different numbers of joins are trained. All models were tested with queries containing 16 triple patterns. The y-axis shows how long each model was trained. This figure should highlight how long the training must last to achieve good results. The results of some join order optimizers without ML are shown on the left. We show that ML on the synthetic data set produces better results than traditional query optimizers. However, the quality of the ML-generated join orders is slightly worse for the real world dataset. In addition, figure 10.12 shows that no optimizer can reliably generate good join orders. The synthetic data set shows that the models must be trained in at least 2^{17} steps to achieve good results often. The quality distribution for the *WordNet* dataset is similar, but the overall quality is lower.

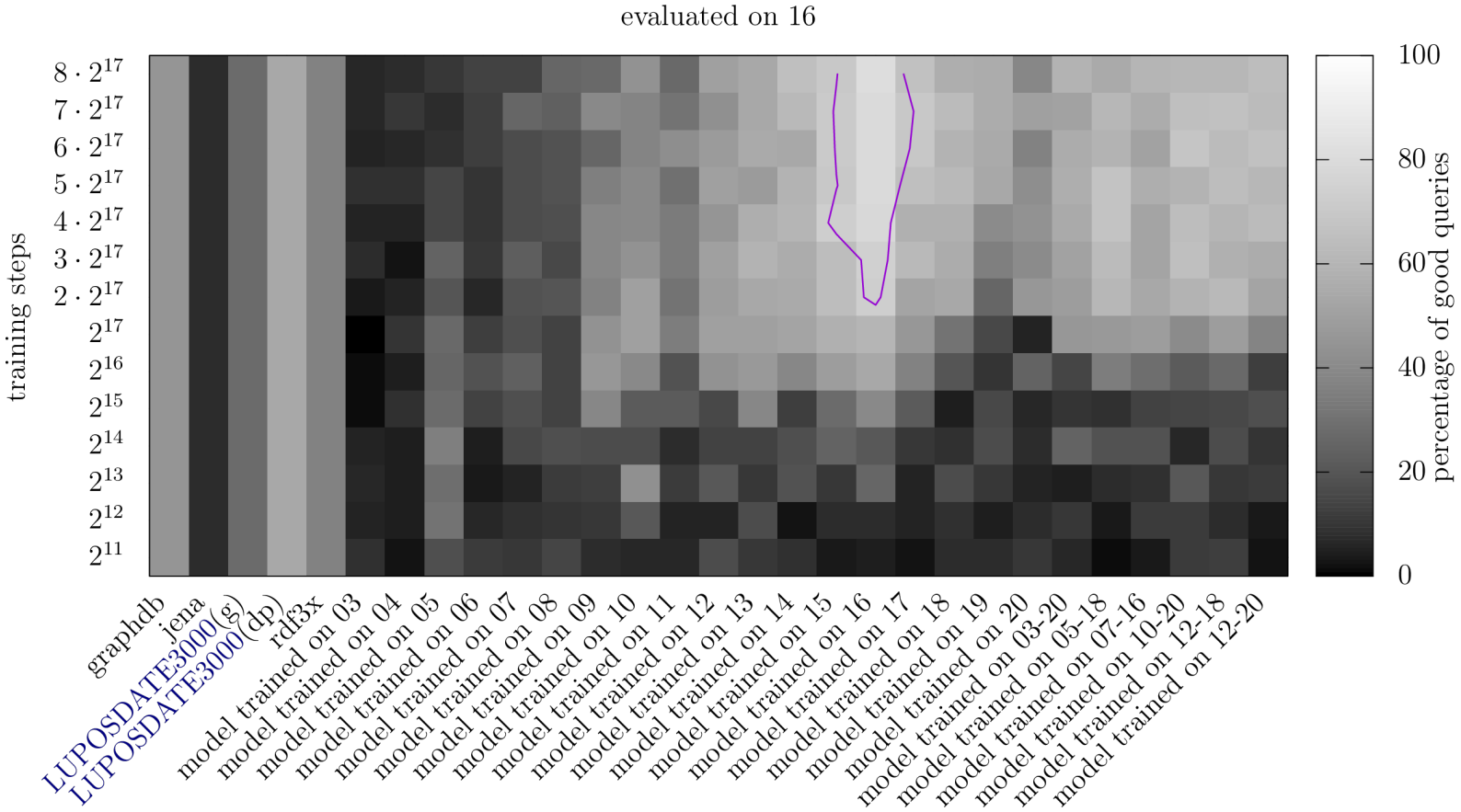


Figure 10.11: SP^{2}B-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70 % of the queries receive good join trees.

Figure 10.12: WordNet-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case.

Furthermore, one can see that only the models trained with a similar number of joins can produce good join trees. In the diagrams on the right, models with mixed numbers of triples are trained. These models should be usable for a broader range of queries. However, the results show that these models for synthetic data can only be used for larger joins after a longer training time. With real world data, these mixed models do not work at all. It is better to explicitly train a new model for each desired join tree size.

10.3.4 Evaluating what the Model has Learned

Synthetic datasets often receive good join orders, so it is unlikely that the quality of the join order is purely random. We looked closely at what join order is used to prevent the model from learning one join order per number of joins. For 3 and 4 triple patterns, only half of the possible join orders are used. The remaining join orders swap the left and right operands while keeping the same structure. When five or more triple patterns should be joined, no join tree is used in more than 1.5 % of the queries. Among the join orders for 20 input triples, no join order is used in more than 0.03 %. From this, it can be concluded that either the model learns the cardinality of the predicate or learns to recognize patterns in the query structure and avoid Cartesian products.

10.4 Evaluating Machine Learning on Network Traffic

In the previous section, ML is applied to the number of intermediate results. This strategy is similar to state-of-the-art join order optimizers. This section improves models by training them on the network traffic from specific queries instead of the intermediate results. SIMORA is used to simulate the DBMS in a reproducible randomized topology. This topology was chosen because it is the most realistic one.

Additionally, the simulator allows the application of all the previous chapters' strategies simultaneously to investigate the best possible variations of each strategy. The distributed parking scenario benchmark generates 9923 triples to obtain and compare plausible network loads. Since the insertion is independent of the join order, the data transmission to store the data is not included. Due to the simulation component, the execution is much slower, so only joins with up to six triple patterns can be analyzed. The smaller number of triple patterns allows as a side effect for a much shorter training phase. During all measurements, the DBMS distributed the data based

on subject hashes because this allows many star-shaped join operators to be executed locally. Figure 10.13 shows the reward received by the model after x iterations. After about 8192 steps, each model frequently receives a high reward. Frequent high rewards indicate that the model is done with the training phase. Similar to the previous section, the training and test phase queries are disjoint. The evaluation of the evaluation-query set confirms that the quality of the join orders does not significantly improve with longer training.

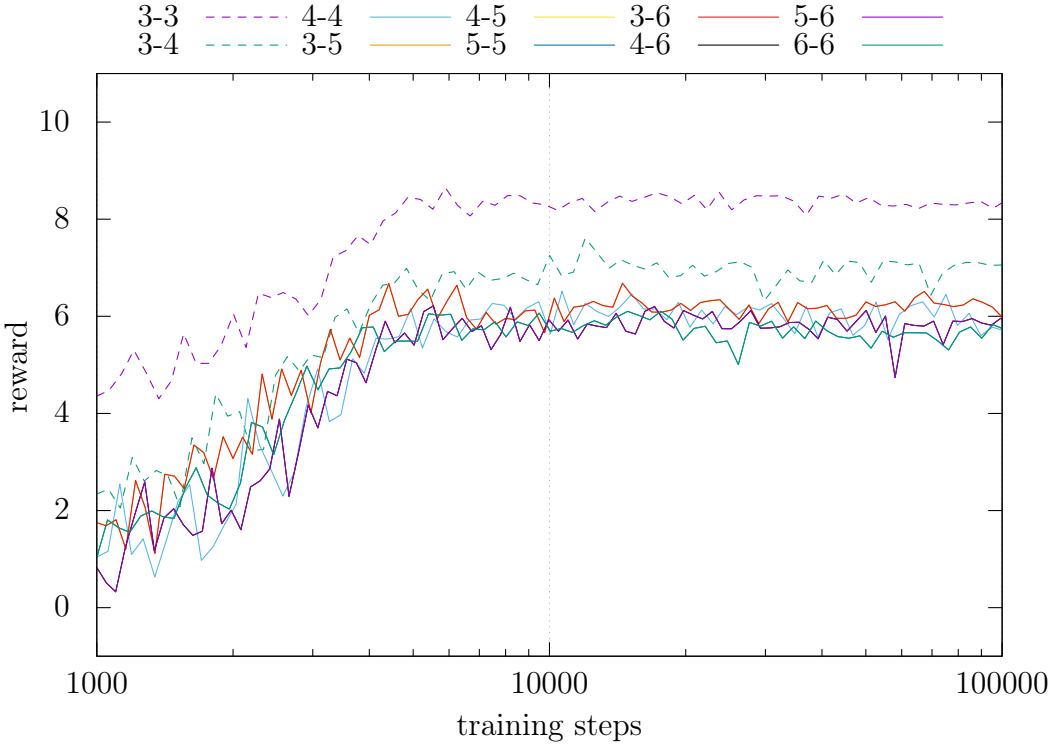


Figure 10.13: This figure shows the reward for the model after x training steps. The displayed graphs use an average running function reset after logarithmic increasing distances. The graph names of the models contain the number of triple patterns used during training.

Having at most six triple patterns has the advantage of executing all possible join orders and storing their intermediate results and network traffic. A timeout of one minute is applied. Additionally, all join orders with Cartesian products are immediately aborted. Whenever a query fails this way, twice the worst-case number of intermediate results and network traffic is assumed during reward calculation. This complete statistical data is used to

evaluate the quality of the join orders calculated by the different algorithms. First, the results for the evaluation based on intermediate results are shown in figure 10.14.

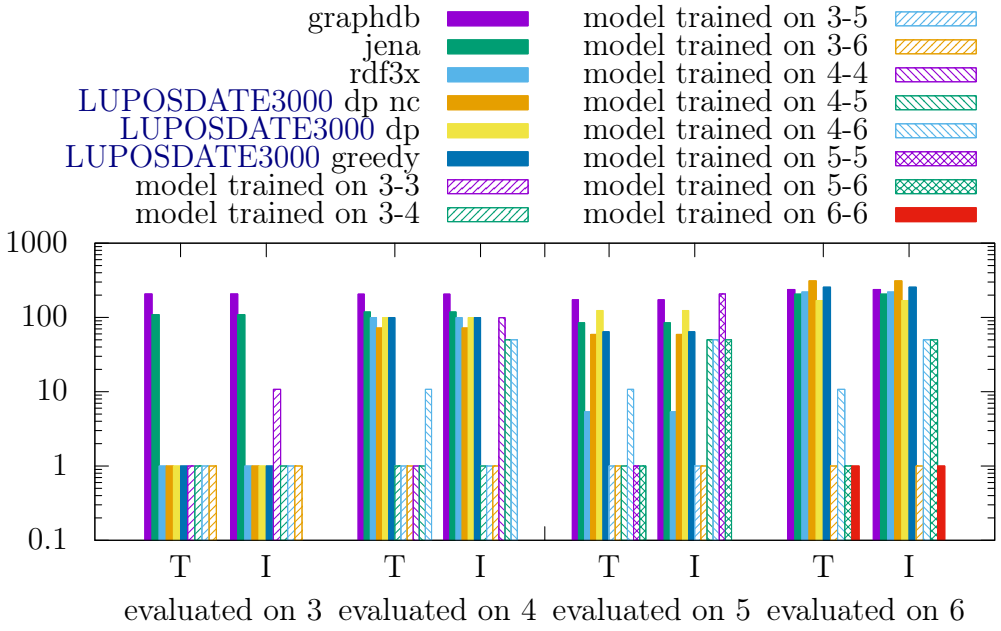


Figure 10.14: All optimizers were evaluated based on the produced intermediate results. The X-axis shows which evaluation on how many triple patterns was performed. *I* means Intermediate-results, and *T* means network-Traffic. The Y-Axis shows the average factor between the best join order and the chosen join order of a given join order optimizer. A factor of 1 is achieved if the optimizer always chooses the best case. The models are trained on joins with the number of triple patterns specified in their labels.

The optimizers from *Jena* and *GraphDB* consider only left deep join orders. However, if only three triple patterns exist, this still matches all possibilities Nevertheless, these two optimizers always produce about 100 times more intermediates than necessary.

The optimizer of *RDF3X* achieves much better results. It can always select the best case for three triple pattern queries and frequently for five triple patterns. However, the four and six triple pattern queries achieve bad results like the previous optimizers.

LUPOSDATE3000 provides two optimizers, dynamic programming (*dp* in the figure) and greedy. The dynamic programming optimizer can optionally cluster the triple patterns by variable names, which reduces the search space,

and increases the optimization speed. In the figure, the optimizer marked with *nc* is not using clustering by variable names. In the context of three triple patterns, all LUPOSDATE3000 built-in optimizers always choose the best join order. However, with more triple patterns, the optimizer achieves a similar quality as the other optimizers without ML.

The ML models, when trained on network traffic, almost always choose the optimal join order. However, when the reward function during training is based on intermediate results, the optimizer creates unnecessary work.

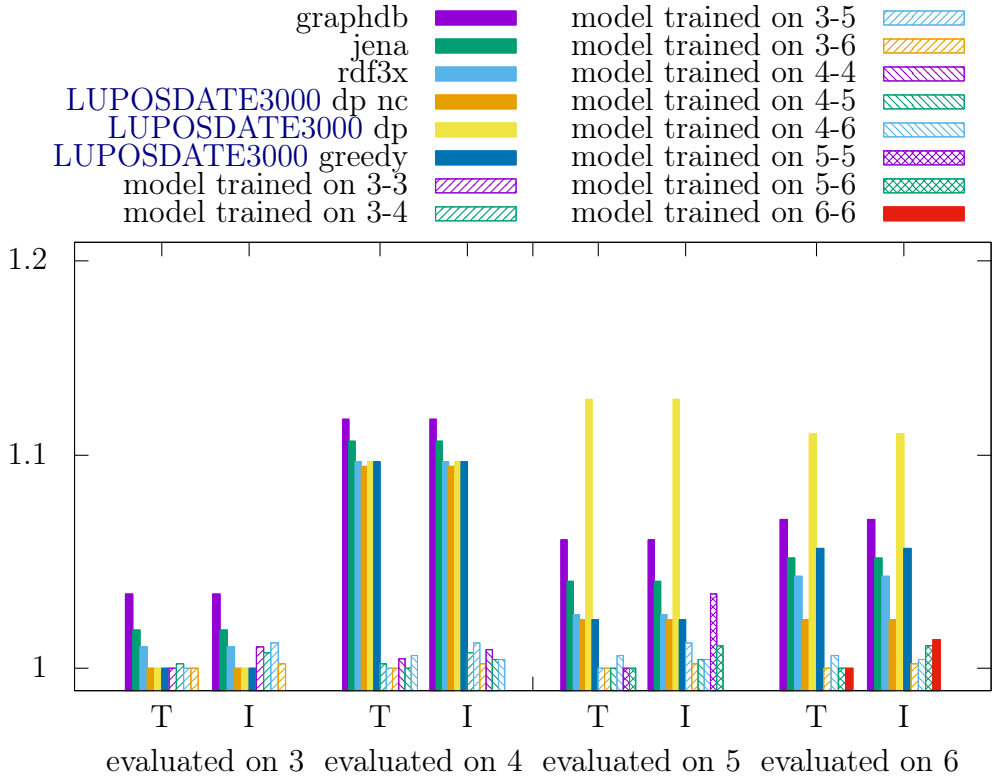


Figure 10.15: All optimizers were evaluated based on the network traffic. The X-axis shows which evaluation on how many triple patterns was performed. *I* means Intermediate-results, and *T* means network-Traffic. The Y-Axis shows the average factor between the best join order and the chosen join order of a given join order optimizer. A factor of 1 is achieved if the optimizer always chooses the best case. The models are trained on joins with the number of triple patterns specified in their labels.

Next, figure 10.15 shows the results for the same queries, but this time the ranking is based on network traffic. The overhead factor is generally

much lower than the previous figure showing the intermediate results. This reduction is achieved by the dynamic operator relocation, which first executes the join. Then the bytes of the input are compared with those of the output. Finally, the option with fewer bytes is chosen and sent over the network.

Consequently, the only overhead is the operator graph size, which is sent twice for every device on the path. Since only the network traffic counts in this comparison, and the smaller number is always chosen, this is no advantage for any specific optimizer. Without this optimization, the bad join orders would be much worse than they are now. It is also possible that the ranking between good and bad join orders changes by disabling this feature. However, in that case, the absolute number of bytes sent through the network will increase, so those join orders can not be called optimal anymore.

The interesting part of this figure is that the trained model almost always achieves less network traffic than the state-of-the-art static optimizers.

When the models are trained on network traffic, their result is almost optimal. They are slightly better than those models which were trained on intermediate results. The best possible results were achieved with the model, which trained on all possible mixed triple pattern sizes. Consequently, the figure also shows that overfitting occurs if the models are trained on small query sets, slightly decreasing the quality.

10.5 Summary

This chapter first presented different strategies for systematically generating queries for training ML models. Then, a new ML approach was presented to optimize join trees in a SPARQL DBMS. After about 1 million training steps, the learned models lead to excellent join sequences in about 80 % of the cases for synthetic data. On the other hand, no optimizer can produce good join sequences for real world data in more than 50 % of the queries. Unlike traditional join tree optimizers, ML allows these join trees to be computed in linear time. While most trained models in the literature learn execution time directly, this approach focuses on the number of intermediate results. Since the number of intermediate results is closely related to the execution time, the execution time is also good. Moreover, the generated query plans are very memory friendly. Finally, the evaluation shows that models trained on network data can increase the quality of the results even further.

Chapter 11

Conclusion

This dissertation focuses on the quality of the query plan, which in turn depends on the underlying data organization. Therefore several data and query organization strategies in a SIoT environment have been studied.

11.1 Summary of Contributions

Overall, the contributions of this thesis can be summarized as follows.

Multiple partitioning schemes were compared with each other. Also, multiple partitioning schemes were applied at the same time, which increased the replication level of the DBMS. Increased replications, and thus options for the join order optimizer, can effectively increase the query performance. However, replications require additional storage space. The analysis also shows that not all partitioning schemes are accessed with the same frequency, so less often used indices can be omitted, saving some storage space. The evaluation shows that too many partitions can also decrease the performance because the overhead for launching threads and merging the results becomes slower than just calculating the result. After the experiments, a function to estimate the optimal number of partitions is proposed. This function considers the amount of data and the number and selectivity of the joins in a query.

DBMS with access to the topology information of the routing protocol. With more information about the environment, a more sophisticated execution plan can be created. After the plan is generated, the topology information can be used again to distribute the operator graph in the network. Here it becomes possible to place the operators on the natural path of the data, which reduces the overall distance the data needs to travel. However, this topology information is usually unavailable. Therefore this

approach currently works only in the simulator SIMORA.

ML and join order optimization ML algorithms feature a constant evaluation time of a given model, drastically improving the optimization phase's speed. However, a new training phase is introduced, which requires a lot of upfront computations. Also, the output quality is less predictable as ML naturally introduces some randomness. State-of-the-art join order optimizers also generate a lot of not optimal join orders. Nevertheless, in combination with topology information, the ML models can produce more efficient join trees than state-of-the-art dynamic programming algorithms since those do not consider network traffic in their cost function.

11.2 Future Research Directions

LUPOSDATE3000 implements only hash-based data distribution. Nevertheless, there already exist other clustering-based distribution strategies. Those strategies can be expected to send more data during the clustering phase because the DBMS needs to decide what to store where. Later during the data retrieval, however, the network traffic might be reduced.

This work analyses heterogeneous network topologies. It also distinguishes between DBMS instances with and without physical data storage. However, heterogeneous hardware properties of the other components, e.g., CPU, GPU, RAM, and storage space, are not considered. Considering completely heterogeneous hardware in one DBMS and how to utilize it efficiently might be interesting.

LUPOSDATE3000 only implements the features necessary to perform the evaluations mentioned in this dissertation. However, other SPARQL features, e.g., inference, still need to be implemented. Furthermore, to apply inference, the rules must be distributed - and updated - yielding additional research questions.

This work only considers an SW DBMS. However, graph, relational, multimodel, and other DBMS types are not considered. In those DBMS, similar data management questions will occur. Due to the different models, each DBMS can ensure different additional properties, e.g., a relational DBMS has fixed data types per column. Those additional properties might further prune irrelevant intermediate results to reduce network traffic and CPU load. Right now, the SW data is stored in a triple store. Another possible research topic could be automatically inferring relational data schemes and storing those triples in automatically inferred relational tables. Those tables could reduce the number of joins in the SW queries. Additionally, by placing multiple triples in a relational table, the common subject could be eliminated, such

that the required storage space might be reduced. Due to the more complex data structure, however, the compression might need to be improved, e.g., by some heuristics about how much data could be stored together or not.

Currently, the DBMS considers one query at a time and optimizes each query independently. However, if there are multiple queries, some might share some intermediate results, which could be reused. In addition, continuous queries might introduce another type of query, which could share or reuse some intermediate results.

The simulator SIMORA could be enhanced to use multiple threads at once. This enhancement can then be used to increase the performance and, thus, the possible amount of data in the DBMS during simulation. Higher amounts of available data might yield different data streams during processing, which might be optimized differently.

LUPOSDATE3000 applies ML in order to optimize the join order. However, the quantum computing hardware is getting cheaper, smaller, and more efficient, so they might also be used for join ordering in SW DBMS.

Part IV

Appendix

A Benchmark Scenario SPARQL queries

This section shows the SPARQL queries for the benchmark defined in chapter 8. The interesting aspects of each of these queries are also explained there.

```
PREFIX parking: <https://parking#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
INSERT DATA {
  parking:AvailableParkingSlots a sosa:FeatureOfInterest .
  parking:AvailableParkingSlots ssn:hasProperty _:ParkingSlotLocation .
  parking:CarMovement a ssn:Stimulus .
  parking:CarMovement ssn:isProxyFor _:ParkingSlotLocation .
  parking:SensorOnEachSlot a sosa:Procedure .
  _:ParkingSlotLocation a sosa:ObservableProperty .
  _:ParkingSlotLocation parking:area \"${area}\"^^xsd:integer .
  _:ParkingSlotLocation parking:spotInArea \"${spotInArea}\"^^xsd:integer .
  _:ParkingSlotLocation sosa:isObservedBy _:Sensor .
  _:ParkingSlotLocation ssn:isPropertyOf parking:AvailableParkingSlots .
  _:Sensor a sosa:Sensor .
  _:Sensor parking:sensorID \"${sensorID}\"^^xsd:integer .
  _:Sensor sosa:observes _:ParkingSlotLocation .
  _:Sensor ssn:detects parking:CarMovement .
  _:Sensor ssn:implements parking:SensorOnEachSlot .

  _:Sensor sosa:madeObservation _:Observation .
  _:Observation a sosa:Observation .
  _:Observation sosa:hasFeatureOfInterest parking:AvailableParkingSlots .
  _:Observation sosa:hasSimpleResult \"${isOccupied}\"^^xsd:boolean .
  _:Observation sosa:madeBySensor _:Sensor .
  _:Observation sosa:observedProperty _:ParkingSlotLocation .
  _:Observation sosa:phenomenonTime \"${sampleTime}\"^^xsd:dateTime .
  _:Observation sosa:resultTime \"${sampleTime}\"^^xsd:dateTime .
  _:Observation sosa:usedProcedure parking:SensorOnEachSlot .
  _:Observation ssn:wasOriginatedBy parking:CarMovement .
}
```

Figure A.1: SPARQL Benchmark Query Insert. The variables enclosed in *${}* highlight which values are changed for each sensor sample.

```
SELECT ?s ?p ?o WHERE {
  ?s ?p ?o.
}
```

Figure A.2: SPARQL Benchmark *Q1*. Select everything.

```
PREFIX parking: <https://parking#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT DISTINCT ?area WHERE {
  ?ParkingSlotLocation a sosa:ObservableProperty .
  ?ParkingSlotLocation parking:area ?area .
}
```

Figure A.3: SPARQL Benchmark *Q2*. Retrieve a list of all parking areas.

```
PREFIX parking: <https://parking#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT (COUNT(DISTINCT ?spotInArea) AS ?count) WHERE {
  ?ParkingSlotLocation a sosa:ObservableProperty .
  ?ParkingSlotLocation parking:area 9 .
  ?ParkingSlotLocation parking:spotInArea ?spotInArea .
}
```

Figure A.4: SPARQL Benchmark *Q3*. Count the number of parking spots in the parking area 9.

```
PREFIX parking: <https://parking#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT (COUNT(?Observation) AS ?count) WHERE {
  ?ParkingSlotLocation a sosa:ObservableProperty .
  ?ParkingSlotLocation parking:area 6 .
  ?ParkingSlotLocation parking:spotInArea 1 .
  ?Observation a sosa:Observation .
  ?Observation sosa:observedProperty ?ParkingSlotLocation .
}
```

Figure A.5: SPARQL Benchmark *Q4*. Count the number of samples from a specific parking spot.

```
PREFIX parking: <https://parking#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT (MAX(?resultTime) AS ?latestDate) WHERE {
  ?ParkingSlotLocation a sosa:ObservableProperty .
  ?ParkingSlotLocation parking:area 7 .
  ?ParkingSlotLocation parking:spotInArea 1 .
  ?Observation a sosa:Observation .
  ?Observation sosa:observedProperty ?ParkingSlotLocation .
  ?Observation sosa:resultTime ?resultTime .
}
```

Figure A.6: SPARQL Benchmark *Q5*. Find out when the last sample from a specific sensor was sent.

```
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX parking: <https://parking#>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?spotInArea ?isOccupied ?lastObservedAt WHERE {
  ?Observation sosa:resultTime ?lastObservedAt .
  ?Observation sosa:hasSimpleResult ?isOccupied .
  ?Observation sosa:observedProperty ?ParkingSlotLocation .
  ?ParkingSlotLocation parking:spotInArea ?spotInArea .
  {
    SELECT(MAX(?resultTime) AS ?lastObservedAt) ?ParkingSlotLocation WHERE {
      ?ParkingSlotLocation a sosa:ObservableProperty .
      ?ParkingSlotLocation parking:area 9 .
      ?Observation a sosa:Observation .
      ?Observation sosa:observedProperty ?ParkingSlotLocation .
      ?Observation sosa:resultTime ?resultTime .
    }
    GROUP BY ?ParkingSlotLocation
  }
}
```

Figure A.7: SPARQL Benchmark *Q6*. Ask for the state of every parking spot in an area and the timestamp of its last measurement.

```
PREFIX parking: <https://parking#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?area ?spotInArea ?isOccupied ?lastObservedAt WHERE {
  ?ParkingSlotLocation parking:area ?area .
  ?ParkingSlotLocation parking:spotInArea ?spotInArea .
  ?Observation sosa:observedProperty ?ParkingSlotLocation .
  ?Observation sosa:resultTime ?lastObservedAt .
  ?Observation sosa:hasSimpleResult ?isOccupied .
  {
    SELECT(MAX(?resultTime) AS ?lastObservedAt) ?ParkingSlotLocation WHERE {
      ?ParkingSlotLocation a sosa:ObservableProperty .
      ?ParkingSlotLocation parking:area ?area .
      ?Observation a sosa:Observation .
      ?Observation sosa:observedProperty ?ParkingSlotLocation .
      ?Observation sosa:resultTime ?resultTime .
      FILTER (?area IN (9, 8, 2))
    }
    GROUP BY ?ParkingSlotLocation
  }
}
```

Figure A.8: SPARQL Benchmark *Q7*. Ask for the state of every parking spot in multiple areas and the timestamp of their last measurements.

```
PREFIX parking: <https://parking#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT (COUNT(?ParkingSlotLocation) AS ?count ) WHERE {
  ?ParkingSlotLocation parking:spotInArea ?spotInArea .
  ?Observation sosa:observedProperty ?ParkingSlotLocation .
  ?Observation sosa:resultTime ?lastObservedAt .
  ?Observation sosa:hasSimpleResult "false"^^xsd:boolean .
  {
    SELECT(MAX(?resultTime) AS ?lastObservedAt) ?ParkingSlotLocation WHERE {
      ?ParkingSlotLocation a sosa:ObservableProperty .
      ?ParkingSlotLocation parking:area 9 .
      ?Observation a sosa:Observation .
      ?Observation sosa:observedProperty ?ParkingSlotLocation .
      ?Observation sosa:resultTime ?resultTime .
    }
    GROUP BY ?ParkingSlotLocation
  }
}
```

Figure A.9: SPARQL Benchmark *Q8*. Count the number of free parking spots in a specific area.

```
PREFIX parking: <https://github.com/luposdate3000/parking#>
PREFIX sosa: <http://www.w3.org/ns/sosa/>
PREFIX ssn: <http://www.w3.org/ns/ssn/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT * WHERE {
  ?Observation sosa:hasSimpleResult ?isOccupied .
  ?Observation sosa:madeBySensor ?Sensor .
  ?Observation sosa:observedProperty ?ParkingSlotLocation .
  ?Observation sosa:phenomenonTime ?sampleTime .
  ?Observation sosa:resultTime ?sampleTime2 .
  ?Sensor sosa:madeObservation ?Observation .
  ?ParkingSlotLocation parking:area ?area .
  ?ParkingSlotLocation parking:spotInArea ?spotInArea .
  ?ParkingSlotLocation sosa:isObservedBy ?Sensor .
  ?Sensor parking:sensorID ?sensorID .
  ?Sensor sosa:observes ?ParkingSlotLocation .
}
```

Figure A.10: SPARQL Benchmark *Q9*. Join everything together to showcase many joins.

Bibliography

[1] Jie Bao (RPI) et al. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. W3C Recommendation. https://www.w3.org/TR/owl2-overview/. W3C, Dec. 2012.

[2] Daniel J. Abadi et al. "Scalable Semantic Web Data Management Using Vertical Partitioning". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 411–422. ISBN: 9781595936493. DOI: 10.5555/1325851.1325900.

[3] Ibrahim Abdelaziz et al. "A survey and experimental comparison of distributed SPARQL engines for very large RDF data". In: *Proceedings of the VLDB Endowment* 10.13 (Sept. 2017), pp. 2049–2060. DOI: 10.14778/3151106.3151109.

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively parallel sort-merge joins in main memory multi-core database systems". In: *Proceedings of the VLDB Endowment* 5.10 (June 2012), pp. 1064–1075. DOI: 10.14778/2336664.2336678.

[5] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. "CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets". In: *The Semantic Web - ISWC 2015*. Springer International Publishing, 2015, pp. 374–389. DOI: 10.1007/978-3-319-25010-6_25.

[6] J. Allam. "Evaluation of a greedy join-order optimization approach using the IMDB dataset". MA thesis. Universität Magdeburg, 2018.

[7] Khaled Alwasel et al. "IoTSim-Osmosis: A framework for modeling and simulating IoT applications over an edge-cloud continuum". In: *Journal of Systems Architecture* 116 (June 2021), p. 101956. DOI: 10.1016/j.sysarc.2020.101956.

[8] Dimitrios Amaxilatis et al. “Advancing Experimentation-as-a-Service Through Urban IoT Experiments”. In: *IEEE Internet of Things Journal* 6.2 (Apr. 2019), pp. 2563–2572. DOI: 10.1109/jiot.2018.2871766.

[9] Medha Atre et al. “Matrix "Bit" loaded”. In: *Proceedings of the 19th international conference on World wide web*. ACM, Apr. 2010. DOI: 10.1145/1772690.1772696.

[10] James Bailey et al. “Web and Semantic Web Query Languages: A Survey”. In: *Reasoning Web*. Ed. by Norbert Eisinger and Jan Maluszynski. Springer Berlin Heidelberg, 2005, pp. 35–133. DOI: 10.1007/11526988_3.

[11] Richard Bellman. “Dynamic Programming”. In: *Science* 153.3731 (July 1966), pp. 34–37. DOI: 10.1126/science.153.3731.34.

[12] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396. http://www.rfc-editor.org/rfc/rfc2396.txt. RFC Editor, Aug. 1998. DOI: 10.17487/rfc2396.

[13] Tim Berners-Lee, James Hendler, and Ora Lassila. “The Semantic Web”. In: *Scientific American* 284.5 (May 2001), pp. 34–43. DOI: 10.1038/scientificamerican0501-34.

[14] Joanna Biega, Erdal Kuzey, and Fabian M. Suchanek. “Inside YAGO2s”. In: *Proceedings of the 22nd International Conference on World Wide Web*. ACM, May 2013. DOI: 10.1145/2487788.2487935.

[15] Dimitris Bilidas and Manolis Koubarakis. “In-memory parallelization of join queries over large ontological hierarchies”. In: *Distributed and Parallel Databases* 39.3 (June 2020), pp. 545–582. DOI: 10.1007/s10619-020-07305-y.

[16] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. “Dictionary-based order-preserving string compression for main memory column stores”. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. SIGMOD ’09. New York, NY, USA: ACM, June 2009, pp. 283–296. ISBN: 9781605585512. DOI: 10.1145/1559845.1559877.

[17] Christian Bizer and Andreas Schultz. “The Berlin SPARQL Benchmark”. In: *International Journal on Semantic Web and Information Systems* 5.2 (Apr. 2009), pp. 1–24. DOI: 10.4018/jswis.2009040101.

[18] Christopher Blochwitz et al. "An optimized radix-tree for hardware-accelerated dictionary generation for semantic web databases". In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Dec. 2015, pp. 1–7. DOI: 10.1109/reconfig.2015.7393291.

[19] Angela Bonifati, Wim Martens, and Thomas Timm. "An analytical study of large SPARQL query logs". In: *Proceedings of the VLDB Endowment* 11.2 (Oct. 2017), pp. 149–161. DOI: 10.14778/3149193.3149196.

[20] A. Brandt, J. Buron, and G. Porcu. *Home Automation Routing Requirements in Low-Power and Lossy Networks*. Tech. rep. Internet Engineering Task Force, Apr. 2010, pp. 1–17. DOI: 10.17487/rfc5826.

[21] A. Brandt et al. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. Tech. rep. Cisco Systems, Mar. 2012. DOI: 10.17487/rfc6550.

[22] David Broneske. "Adaptive Reprogramming for Databases on Heterogeneous Processors". In: *Proceedings of the 2015 ACM SIGMOD on PhD Symposium*. ACM, May 2015. DOI: 10.1145/2744680.2744685.

[23] Rodrigo N. Calheiros et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and Experience* 41.1 (Aug. 2010), pp. 23–50. DOI: 10.1002/spe.995.

[24] Keyan Cao et al. "An Overview on Edge Computing Research". In: *IEEE Access* 8 (2020), pp. 85714–85728. DOI: 10.1109/access.2020.2991734.

[25] Gavin Carothers and Eric Prudhommeaux. *RDF 1.1 Turtle*. W3C Recommendation. https://www.w3.org/TR/2014/REC-turtle-20140225/. W3C, Feb. 2014.

[26] Jeremy J. Carroll et al. "Jena". In: *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters - WWW Alt. '04*. ACM Press, 2004. DOI: 10.1145/1013367.1013381.

[27] Jian-Ming Chang et al. "The 6LoWPAN ad-hoc on demand distance vector routing with multi-path scheme". In: *IET International Conference on Frontier Computing. Theory, Technologies and Applications*. IET, 2010, pp. 204–209. DOI: 10.1049/cp.2010.0562.

[28] Shanzhi Chen et al. "A Vision of IoT: Applications, Challenges, and Opportunities With China Perspective". In: *IEEE Internet of Things Journal* 1.4 (Aug. 2014), pp. 349–359. DOI: 10.1109/jiot.2014.2337336.

[29] Yunxia Chen and Qing Zhao. "On the lifetime of wireless sensor networks". In: *IEEE Communications Letters* 9.11 (Nov. 2005), pp. 976–978. DOI: 10.1109/lcomm.2005.11010.

[30] Eugene Inseok Chong et al. "An Efficient SQL-Based RDF Querying Scheme". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 1216–1227. ISBN: 1595931546.

[31] Thomas Clausen, Jiazi Yi, and Ulrich Herberg. "Lightweight On-demand Ad hoc Distance-vector Routing - Next Generation (LOADng): Protocol, extension, and applicability". In: *Computer Networks* 126 (Oct. 2017), pp. 125–140. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2017.06.025.

[32] Michael Compton et al. "The SSN ontology of the W3C semantic sensor network incubator group". In: *Journal of Web Semantics* 17 (Dec. 2012), pp. 25–32. ISSN: 1570-8268. DOI: 10.1016/j.websem.2012.05.003.

[33] ns-3 Consortium. *ns-3: Network Simulator*. URL: https://www.nsnam.org/ (visited on 05/17/2023).

[34] Antonio Coutinho et al. "Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing". In: *2018 IEEE International Conference on Communications (ICC)*. IEEE, May 2018, pp. 1–7. DOI: 10.1109/icc.2018.8423003.

[35] Sanjit Kumar Dash et al. "Sensor-Cloud: Assimilation of Wireless Sensor Network and the Cloud". In: *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer Berlin Heidelberg, 2012, pp. 455–464. DOI: 10.1007/978-3-642-27299-8_48.

[36] Stephen Dawson-Haggerty, Arsalan Tavakoli, and David Culler. "Hydro: A Hybrid Routing Protocol for Low-Power and Lossy Networks". In: *2010 First IEEE International Conference on Smart Grid Communications*. IEEE, Oct. 2010, pp. 268–273. DOI: 10.1109/smartgrid.2010.5622053.

[37] P. De, N. Riou, and W. Vermeylen. *Building Automation Routing Requirements in Low-Power and Lossy Networks*. Tech. rep. Internet Engineering Task Force, June 2010. DOI: 10.17487/rfc5867.

[38] Amol Dhumane, Rajesh Prasad, and Jayashree Prasad. "Routing issues in internet of things: a survey". In: *Proceedings of the international multiconference of engineers and computer scientists*. Vol. 1. 2016, pp. 16–18. ISBN: 978-988-19253-8-1.

[39] M. Duerst and M. Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC 3987. http://www.rfc-editor.org/rfc/rfc3987.txt. RFC Editor, Jan. 2005. DOI: 10.17487/rfc3987.

[40] Khaled Qorany Abdel Fadeel and Khaled El Sayed. "ESMRF". In: *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems*. ACM, May 2015. DOI: 10.1145/2753476.2753479.

[41] Szymon Fedor and Martin Collier. "On the Problem of Energy Efficiency of Multi-Hop vs One-Hop Routing in Wireless Sensor Networks". In: *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*. Vol. 2. IEEE, 2007, pp. 380–385. DOI: 10.1109/ainaw.2007.272.

[42] Christiane Fellbaum. *WordNet*. Ed. by Christiane Fellbaum. Language, Speech, and Communication. Cambridge, MA: The MIT Press, 1998. ISBN: 978-0-262-06197-1. DOI: 10.7551/mitpress/7287.001.0001.

[43] Caxton C. Foster. "A generalization of AVL trees". In: *Communications of the ACM* 16.8 (Aug. 1973), pp. 513–517. ISSN: 0001-0782. DOI: 10.1145/355609.362340.

[44] Kotlin Foundation. *Kotlin Multiplatform*. URL: https://kotlinlang.org/docs/multiplatform.html (visited on 04/20/2023).

[45] The Apache Software Foundation. *Apache Jena - TDB Architecture*. URL: https://jena.apache.org/documentation/tdb/architecture.html (visited on 05/17/2023).

[46] Luis Galárraga, Katja Hose, and Ralf Schenkel. "Partout". In: *Proceedings of the 23rd International Conference on World Wide Web*. ACM, Apr. 2014. DOI: 10.1145/2567948.2577302.

[47] Archana Ganapathi et al. "Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning". In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE, Mar. 2009, pp. 592–603. DOI: 10.1109/icde.2009.130.

[48] Fabien Gandon and Guus Schreiber. *RDF 1.1 XML Syntax*. W3C Recommendation. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/. W3C, Feb. 2014.

[49] A. Garyfalos, K. Almeroth, and J. Finney. "A comparison of network and application layer multicast for mobile IPv6 networks". In: *Proceedings of the 6th international workshop on Modeling analysis and simulation of wireless and mobile systems - MSWIM '03*. ACM Press, 2003. DOI: 10.1145/940991.941003.

[50] Vangelis Gazis et al. "Short Paper: IoT: Challenges, projects, architectures". In: *2015 18th International Conference on Intelligence in Next Generation Networks*. IEEE, 2015, pp. 145–147. DOI: 10.1109/icin.2015.7073822.

[51] Francois Goasdoue et al. "CliqueSquare: Flat plans for massively parallel RDF queries". In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE, Apr. 2015, pp. 771–782. DOI: 10.1109/icde.2015.7113332.

[52] Alexey Goncharuk. *Introduction to the Join Ordering Problem*. URL: https://www.querifylabs.com/blog/introduction-to-the-join-ordering-problem (visited on 05/17/2023).

[53] Jinghua Groppe and Sven Groppe. "Parallelizing join computations of SPARQL queries for large semantic web databases". In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, Mar. 2011. DOI: 10.1145/1982185.1982536.

[54] Sven Groppe. *Data Management and Query Processing in Semantic Web Databases*. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-19357-6.

[55] Sven Groppe. "Emergent models, frameworks, and hardware technologies for Big data analytics". In: *The Journal of Supercomputing* 76.3 (Feb. 2018), pp. 1800–1827. DOI: 10.1007/s11227-018-2277-x.

[56] Sven Groppe, Rico Klinckenberg, and Benjamin Warnke. "Generating Sound from the Processing in Semantic Web Databases". In: *Open Journal of Semantic Web (OJSW)* 8.1 (2021), pp. 1–27. ISSN: 2199-336X. URL: http://nbn-resolving.de/urn:nbn:de:101:1-202201161833054484370 4.

[57] Sven Groppe, Rico Klinckenberg, and Benjamin Warnke. "Sound of databases". In: *Proceedings of the VLDB Endowment* 14.12 (July 2021), pp. 2695–2698. DOI: 10.14778/3476311.3476322.

[58] The PostgreSQL Global Development Group. *Documentation - PostgreSQL 15*. URL: https://www.postgresql.org/docs/current/largeobjects.html (visited on 05/17/2023).

[59] Andrey Gubichev and Thomas Neumann. "Exploiting the query structure for efficient join ordering in SPARQL queries". In: *EDBT*. 2014.

[60] Ramanathan Guha and Dan Brickley. *RDF Schema 1.1*. W3C Recommendation. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/. W3C, Feb. 2014.

[61] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems". In: *Journal of Web Semantics* 3.2-3 (Oct. 2005). Selcted Papers from the International Semantic Web Conference, 2004, pp. 158–182. ISSN: 1570-8268. DOI: 10.1016/j.websem.2005.06.005.

[62] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. "PQR: Predicting Query Execution Times for Autonomous Workload Management". In: *2008 International Conference on Autonomic Computing*. IEEE, June 2008, pp. 13–22. DOI: 10.1109/icac.2008.12.

[63] Harshit Gupta et al. "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments". In: *Software: Practice and Experience* 47.9 (June 2017), pp. 1275–1296. DOI: 10.1002/spe.2509.

[64] Sairam Gurajada et al. "TriAD". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, June 2014. DOI: 10.1145/2588555.2610511.

[65] M. Haklay and P. Weber. "OpenStreetMap: User-Generated Street Maps". In: *IEEE Pervasive Computing* 7.4 (Oct. 2008), pp. 12–18. DOI: 10.1109/mprv.2008.80.

[66] Armin Haller et al. "The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation". In: *Semantic Web* 10.1 (Dec. 2018). Ed. by Pascal Hitzler, pp. 9–32. ISSN: 1570-0844. DOI: 10.3233/sw-180320.

[67] Mohammad Hammoud et al. "DREAM". In: *Proceedings of the VLDB Endowment* 8.6 (Feb. 2015), pp. 654–665. DOI: 10.14778/2735703.2735705.

[68] Hankwang. *Datei:Hard drive capacity over time.svg*. URL: https://de.wikipedia.org/wiki/Datei:Hard%5C_drive%5C_capacity%5C_over%5C_time.svg (visited on 05/17/2023).

[69] Razen Harbi et al. "Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning". In: *The VLDB Journal* 25.3 (Feb. 2016), pp. 355–380. DOI: 10.1007/s00778-016-0420-y.

[70] Razen Harbi et al. "Evaluating SPARQL queries on massive RDF datasets". In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015), pp. 1848–1851. DOI: 10.14778/2824032.2824083.

[71] Rakebul Hasan and Fabien Gandon. "A Machine Learning Approach to SPARQL Query Performance Prediction". In: *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. Vol. 1. IEEE, Aug. 2014, pp. 266–273. DOI: 10.1109/wi-iat.2014.43.

[72] Mohammad Mehedi Hassan, Biao Song, and Eui-Nam Huh. "A framework of sensor-cloud integration opportunities and challenges". In: *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*. ACM, Feb. 2009. DOI: 10.1145/1516241.1516350.

[73] Jonas Heitz and Kurt Stockinger. "Join Query Optimization with Deep Reinforcement Learning Algorithms". In: *ArXiv* abs/1911.11689 (2019). DOI: 10.48550/arXiv.1911.11689.

[74] José-Miguel Herrera, Aidan Hogan, and Tobias Käfer. "BTC-2019: The 2019 Billion Triple Challenge Dataset". In: *Lecture Notes in Computer Science*. Ed. by Chiara Ghidini et al. Springer International Publishing, 2019, pp. 163–180. DOI: 10.1007/978-3-030-30796-7_11.

[75] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. May 2023.

[76] Johannes Hoffart et al. "YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia". In: *Artificial Intelligence* 194 (Jan. 2013), pp. 28–61. DOI: 10.1016/j.artint.2012.06.001.

[77] Shengyi Huang and Santiago Ontañón. "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms". In: *The International FLAIRS Conference Proceedings* 35 (May 2022). DOI: 10.32473/flairs.v35i.130584.

[78] J. Hui and R. Kelsey. *Multicast Protocol for Low-Power and Lossy Networks (MPL)*. Tech. rep. Internet Engineering Task Force, Feb. 2016. DOI: 10.17487/rfc7731.

[79] Mohammad Husain et al. "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing". In: *IEEE Transactions on Knowledge and Data Engineering* 23.9 (Sept. 2011), pp. 1312–1327. DOI: 10.1109/tkde.2011.103.

[80] Balakrishna R. Iker and Arun N. Swami. "Method for optimizing processing of join queries by determining optimal processing order and assigning optimal join methods to each of the join operations". U.S. pat. 5345585. Sept. 1994.

[81] solid IT gmbh. *DB-Engines Ranking of RDF Stores*. URL: https://db-engines.com/en/ranking/rdf+store (visited on 05/17/2023).

[82] Hajira Jabeen et al. "DISE: A Distributed in-Memory SPARQL Processing Engine over Tensor Data". In: *2020 IEEE 14th International Conference on Semantic Computing (ICSC)*. IEEE, Feb. 2020, pp. 400–407. DOI: 10.1109/icsc.2020.00079.

[83] Daniel Janke, Steffen Staab, and Martin Leinberger. "Data placement strategies that speed-up distributed graph query processing". In: *Proceedings of The International Workshop on Semantic Big Data*. ACM, June 2020. DOI: 10.1145/3391274.3393633.

[84] Apache Jena. *Apache Jena Open - Source Edition*. https://dlcdn.apache.org/jena/binaries/apache-jena-4.5.0.tar.gz. May 2023.

[85] Devki Nandan Jha et al. "IoTSim-Edge: A simulation framework for modeling the behavior of Internet of Things and edge computing environments". In: *Software: Practice and Experience* 50.6 (Jan. 2020), pp. 844–867. DOI: 10.1002/spe.2787.

[86] Gabor Kecskemeti et al. "Modelling and Simulation Challenges in Internet of Things". In: *IEEE Cloud Computing* 4.1 (Jan. 2017), pp. 62–69. DOI: 10.1109/mcc.2017.18.

[87] Michael Kifer and Harold Boley. *RIF Overview (Second Edition)*. W3C Note. https://www.w3.org/TR/2013/NOTE-rif-overview-20130205/. W3C, Feb. 2013.

[88] K Kim et al. *Dynamic MANET On-demand for 6LoWPAN (DYMO-low) Routing, Internet Engineering Task Force*. Tech. rep. Internet-Draft, Jun. 2007, work in progress, 2007.

[89] Graham Klyne and Jeremy Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation. https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. W3C, Feb. 2004.

[90] Dimitris Kontokostas and Holger Knublauch. *Shapes Constraint Language (SHACL)*. W3C Recommendation. https://www.w3.org/TR/2017/REC-shacl-20170720/. W3C, July 2017.

[91] Sanjay Krishnan et al. "Learning to Optimize Join Queries With Deep Reinforcement Learning". In: *ArXiv* abs/1808.03196 (2018). DOI: 10.48550/arXiv.1808.03196.

[92] A. Kröller et al. "Shawn: A new approach to simulating wireless sensor networks". In: *Design, Analysis, and Simulation of Distributed Systems 2005, Part of the SpringSim 2005*. Apr. 2005.

[93] G. Kuck. "Tim Berners-Lee's Semantic Web". In: *SA Journal of Information Management* 6.1 (Dec. 2004). ISSN: 1560-683X. DOI: 10.4102/sajim.v6i1.297.

[94] Hai Lan, Zhifeng Bao, and Yuwei Peng. "A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration". In: *Data Science and Engineering* 6.1 (Jan. 2021), pp. 86–101. DOI: 10.1007/s41019-020-00149-7.

[95] Kisung Lee and Ling Liu. "Scaling queries over big RDF graphs with semantic hash partitioning". In: *Proceedings of the VLDB Endowment* 6.14 (Sept. 2013), pp. 1894–1905. DOI: 10.14778/2556549.2556571.

[96] Maxime Lefrançois et al. *Semantic Sensor Network Ontology*. W3C Recommendation. https://www.w3.org/TR/2017/REC-vocab-ssn-20171019/. W3C, Oct. 2017.

[97] Isaac Lera, Carlos Guerrero, and Carlos Juiz. "YAFS: A Simulator for IoT Scenarios in Fog Computing". In: *IEEE Access* 7 (2019), pp. 91745–91758. DOI: 10.1109/access.2019.2927895.

[98] Qi Li et al. "A Semantic Collaboration Method Based on Uniform Knowledge Graph". In: *IEEE Internet of Things Journal* 7.5 (May 2020), pp. 4473–4484. DOI: 10.1109/jiot.2019.2960150.

[99] Xinxin Liu et al. "Load balanced routing for low power and lossy networks". In: *2013 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, Apr. 2013, pp. 2238–2243. DOI: 10.1109/wcnc.2013.6554908.

[100] Márcio Moraes Lopes et al. "MyiFogSim". In: *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*. ACM, Dec. 2017. DOI: 10.1145/3147234.3148101.

[101] Hongjun Lu, Hock Chuan Chan, and Kwok Kee Wei. "A survey on usage of SQL". In: *ACM SIGMOD Record* 22.4 (Dec. 1993), pp. 60–65. DOI: 10.1145/166635.166656.

[102] Ed. M. Dohler et al. *Routing Requirements for Urban Low-Power and Lossy Networks*. Tech. rep. France Telecom R&D, May 2009. DOI: 10.17487/rfc5548.

[103] Werner Mach and Erich Schikuta. "Optimized Workflow Orchestration of Database Aggregate Operations on Heterogenous Grids". In: *2008 37th International Conference on Parallel Processing*. IEEE, Sept. 2008, pp. 214–221. DOI: 10.1109/icpp.2008.12.

[104] Werner Mach and Erich Schikuta. "Performance analysis of parallel database sort operations in a heterogenous Grid Environment". In: *2007 IEEE International Conference on Cluster Computing*. IEEE, 2007, pp. 525–533. DOI: 10.1109/clustr.2007.4629279.

[105] David Makepeace et al. *An Indexing Scheme for a Scalable RDF Triple Store*. URL: https://redirect.cs.umbc.edu/courses/691s/papers/RDFScalableIndexing.pdf (visited on 05/17/2023).

[106] Johann Mantler. "Simulation Framework for Distributed Database Query Processing in the Semantic Internet of Things". MA thesis. Universitaet zu Luebeck, 2021.

[107] Ryan Marcus and Olga Papaemmanouil. "Deep Reinforcement Learning for Join Order Enumeration". In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, June 2018. DOI: 10.1145/3211954.3211957.

[108] Charafeddine Mechalikh, Hajer Taktak, and Faouzi Moussa. "PureEdgeSim: A simulation framework for performance evaluation of cloud, edge and mist computing environments". In: *Computer Science and Information Systems* 18.1 (2021), pp. 43–66. DOI: 10.2298/csis200301042m.

[109] Marios Meimaris and George Papastefanatos. "Double Chain-Star: an RDF indexing scheme for fast processing of SPARQL joins". In: *International Conference on Extending Database Technology*. 2016. DOI: 10.5441/002/edbt.2016.78.

[110] G. Montenegro et al. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. Tech. rep. Internet Engineering Task Force, Sept. 2007, pp. 1–30. DOI: 10.17487/rfc4944.

[111] Hubert Naacke and Olivier Curé. "On Distributed SPARQL Query Processing Using Triangles of RDF Triples". In: *Open Journal of Semantic Web (OJSW)* 7.1 (2020), pp. 17–32. ISSN: 2199-336X. URL: http://nbn-resolving.de/urn:nbn:de:101:1-202011221833331672109.

[112] Nitin Nayak et al. "Constructing Optimal Bushy Join Trees by Solving QUBO Problems on Quantum Hardware and Simulators". In: *Proceedings of The International Workshop on Big Data in Emergent Distributed Environments*. New York, NY, USA: ACM, June 2023. DOI: 10.1145/3579142.3594298.

[113] Thomas Neumann and Gerhard Weikum. "RDF-3X". In: *Proceedings of the VLDB Endowment* 1.1 (Aug. 2008), pp. 647–659. DOI: 10.14778/1453856.1453927.

[114] Thomas Neumann and Gerhard Weikum. *RDF-3X*. https://gitlab.db.in.tum.de/dbtools/rdf3x. May 2023.

[115] Thomas Neumann and Gerhard Weikum. "The RDF-3X engine for scalable management of RDF data". In: *The VLDB Journal* 19.1 (Sept. 2009), pp. 91–113. DOI: 10.1007/s00778-009-0165-y.

[116] George Oikonomou, Iain Phillips, and Theo Tryfonas. "IPv6 Multicast Forwarding in RPL-Based Wireless Sensor Networks". In: *Wireless Personal Communications* 73.3 (June 2013), pp. 1089–1116. DOI: 10.1007/s11277-013-1250-5.

[117] Rogerio Leao Santos de Oliveira et al. "Using Mininet for emulation and prototyping Software-Defined Networks". In: *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE, June 2014, pp. 1–6. DOI: 10.1109/colcomcon.2014.6860404.

[118] Ontotext. *GraphDB 10.2*. https://graphdb.ontotext.com/documentation. May 2023.

[119] Fredrik Osterlind et al. "Cross-Level Sensor Network Simulation with COOJA". In: *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*. IEEE, Nov. 2006, pp. 641–648. DOI: 10.1109/lcn.2006.322172.

[120] Simon Paasche and Sven Groppe. "Enhancing data quality and process optimization for smart manufacturing lines in industry 4.0 scenarios". In: *Proceedings of The International Workshop on Big Data in Emergent Distributed Environments*. ACM, June 2022. DOI: 10.1145/3530050.3532928.

[121] Simon Paasche and Sven Groppe. "Generating SPARQL-Constraints for Consistency Checking in Industry 4.0 Scenarios". In: *Open Journal of Internet Of Things (OJIOT)* 8.1 (2022), pp. 80–90. ISSN: 2364-7108. URL: https://www.ronpub.com/ojiot/OJIOT_2022v8i1n08_Paasche.html.

[122] Georgios Z. Papadopoulos et al. "BMFA: Bi-Directional Multicast Forwarding Algorithm for RPL-based 6LoWPANs". In: *Interoperability, Safety and Security in IoT*. Springer International Publishing, 2017, pp. 18–25. DOI: 10.1007/978-3-319-52727-7_3.

[123] Nikolaos Papailiou et al. "H_2RDF+: High-performance distributed joins over large-scale RDF graphs". In: *2013 IEEE International Conference on Big Data*. IEEE, Oct. 2013, pp. 255–263. DOI: 10.1109/bigdata.2013.6691582.

[124] Peng Peng et al. "Processing SPARQL queries over distributed RDF graphs". In: *The VLDB Journal* 25.2 (Jan. 2016), pp. 243–268. DOI: 10.1007/s00778-015-0415-0.

[125] C.E. Perkins and E.M. Royer. "Ad-hoc on-demand distance vector routing". In: *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, 1999, pp. 90–100. DOI: 10.1109/mcsa.1999.749281.

[126] Danh Le-Phuoc et al. "Linked Stream Data Processing Engines: Facts and Figures". In: *The Semantic Web – ISWC 2012*. Springer Berlin Heidelberg, 2012, pp. 300–312. DOI: 10.1007/978-3-642-35173-0_20.

[127] Kristofer Pister et al. "RFC 5673: Industrial Routing Requirements in Low-Power and Lossy Networks". In: *IETF* RFC 5673 (Oct. 2009).

[128] Christopher Popfinger. "Realisation of Active Multidatabases by Extending Standard Database Interfaces". In: *Workshop on Foundations of Databases (Grundlagen von Datenbanken)*. June 2003, pp. 40–44. URL: https://dbs.cs.uni-duesseldorf.de/publikationen.php?&pubid=1.

[129] Bogdan Prisacari et al. "Bandwidth-optimal all-to-all exchanges in fat tree networks". In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ICS '13. New York, NY, USA: ACM, June 2013, pp. 139–148. ISBN: 9781450321303. DOI: 10.1145/2464996.2465434.

[130] Eric Prud'hommeaux and Gavin Carothers. *RDF 1.1 Turtle*. W3C Recommendation. https://www.w3.org/TR/2014/REC-turtle-20140225/. W3C, Feb. 2014.

[131] Tariq Qayyum et al. "FogNetSim+: A Toolkit for Modeling and Simulation of Distributed Fog Environment". In: *IEEE Access* 6 (2018), pp. 63570–63583. DOI: 10.1109/access.2018.2877696.

[132] Greg Rahn. *Join Order Benchmark (JOB)*. https://github.com/gregrahn/join-order-benchmark.git. May 2023.

[133] Kurt Rohloff and Richard E. Schantz. "Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store". In: *Proceedings of the fourth international workshop on Data-intensive distributed computing*. ACM, June 2011. DOI: 10.1145/1996014.1996021.

[134] Kurt Rohloff and Richard E. Schantz. "High-performance, massively scalable distributed systems using the MapReduce software framework". In: *Programming Support Innovations for Emerging Distributed Applications*. ACM, Oct. 2010. DOI: 10.1145/1940747.1940751.

[135] Karl Rupp. *microprocessor-trend-data*. https://github.com/karlrupp/microprocessor-trend-data. May 2023.

[136] Luis Sanchez et al. "SmartSantander: IoT experimentation over a smart city testbed". In: *Computer Networks* 61 (Mar. 2014). Special issue on Future Internet Testbeds – Part I, pp. 217–238. ISSN: 1389-1286. DOI: 10.1016/j.bjp.2013.12.020.

[137] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. "PigSPARQL". In: *Proceedings of the International Workshop on Semantic Web Information Management*. ACM, June 2011. DOI: 10.1145/1999299.1999303.

[138] Alexander Schätzle et al. "S2RDF". In: *Proceedings of the VLDB Endowment* 9.10 (June 2016), pp. 804–815. DOI: 10.14778/2977797.2977806.

[139] Michael Schmidt et al. "SP^{2}Bench: A SPARQL Performance Benchmark". In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE, Mar. 2009, pp. 222–233. DOI: 10.1109/icde.2009.28.

[140] Peter Patel Schneider, Masahiro Hori, and Jerome Euzenat. *OWL Web Ontology Language XML Presentation Syntax*. W3C Note. https://www.w3.org/TR/2003/NOTE-owl-xmlsyntax-20030611/. W3C, June 2003.

[141] Falk Scholer et al. "Compression of inverted indexes For fast query evaluation". In: *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. SIGIR '02. New York, NY, USA: ACM, Aug. 2002, pp. 222–229. ISBN: 1581135610. DOI: 10.1145/564376.564416.

[142] Andy Seaborne and Steven Harris. *SPARQL 1.1 Query Language*. W3C Recommendation. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/. W3C, Mar. 2013.

[143] Bin Shao, Haixun Wang, and Yatao Li. "Trinity". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, June 2013. DOI: 10.1145/2463676.2467799.

[144] José V. V. Sobral et al. "Routing Protocols for Low Power and Lossy Networks in Internet of Things Applications". In: *Sensors* 19.9 (May 2019), p. 2144. DOI: 10.3390/s19092144.

[145] OpenLink Software. *Virtuoso Open - Source Edition*. git://github.com/openlink/virtuoso-opensource.git. May 2023.

[146] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. "EdgeCloudSim: An environment for performance evaluation of Edge Computing systems". In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, May 2017, pp. 39–44. DOI: 10.1109/fmec.2017.7946405.

[147] Michael Sperberg-McQueen et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. https://www.w3.org/TR/2008/REC-xml-20081126/. W3C, Nov. 2008.

[148] M. Stonebraker and M. Olson. "Large object support in POSTGRES". In: *Proceedings of IEEE 9th International Conference on Data Engineering*. USA: IEEE Comput. Soc. Press, 1993. DOI: 10.1109/icde.1993.344046.

[149] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. "Yago". In: *Proceedings of the 16th international conference on World Wide Web*. ACM, May 2007. DOI: 10.1145/1242572.1242667.

[150] Technical Support. *Announcing Virtuoso Open-Source Edition, Version 6.1.2*. URL: https://lists.w3.org/Archives/Public/public-lod/2010Jul/0302.html (visited on 05/17/2023).

[151] Wojciech Szpankowski. "Patricia tries again revisited". In: *Journal of the ACM* 37.4 (Oct. 1990), pp. 691–711. ISSN: 0004-5411. DOI: 10.1145/96559.214080.

[152] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian Suchanek. "YAGO 4: A Reason-able Knowledge Base". In: *The Semantic Web*. Springer International Publishing, 2020, pp. 583–596. DOI: 10.1007/978-3-030-49461-2_34.

[153] Torsten Teubler. "Ein namenszentrischer Ansatz zur Realisierung von Diensten im Internet der Dinge". PhD thesis. Zentrale Hochschulbibliothek Lübeck, 2019.

[154] Inc. Unicode. *History of Unicode Release and Publication Dates*. URL: http://unicode.org/history/publicationdates.html (visited on 05/17/2023).

[155] Princeton University. *Princeton WordNet 3.1*. http://WordNet-rdf.princeton.edu/static/WordNet.nt.gz. May 2023.

[156] María-Esther Vidal et al. "Efficiently Joining Group Patterns in SPARQL Queries". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 228–242. DOI: 10.1007/978-3-642-13486-9_16.

[157] Hongzhi Wang et al. "April". In: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. CIKM '20. New York, NY, USA: ACM, Oct. 2020, pp. 3465–3468. ISBN: 9781450368599. DOI: 10.1145/3340531.3417422.

[158] Benjamin Warnke. *Luposdate3000*. https://github.com/luposdate3000/luposdate300. May 2023.

[159] Benjamin Warnke. *SIMORA*. https://github.com/LUPOSDATE3000/SIMORA. May 2023.

[160] Benjamin Warnke, Sven Groppe, and Stefan Fischer. "Distributed SPARQL queries in collaboration with the routing protocol". In: *International Database Engineered Applications Symposium Conference (IDEAS 2023), May 5–7, 2023, Heraklion, Crete, Cyprus*. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 979-8-4007-0744-5. DOI: 10.1145/3589462.3589497.

[161] Benjamin Warnke et al. "A SPARQL benchmark for distributed databases in IoT environments". In: *Proceedings of The International Workshop on Big Data in Emergent Distributed Environments*. New York, NY, USA: ACM, June 2022. DOI: 10.1145/3530050.3532929.

[162] Benjamin Warnke et al. “Flexible data partitioning schemes for parallel merge joins in semantic web queries”. In: *Datenbanksysteme für Business, Technologie und Web (BTW), 19. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme", Dresden, Germany*. Ed. by Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 237–256. DOI: 10.18420/btw2021-12.

[163] Benjamin Warnke et al. “ReJOOSp: Reinforcement learning for join order optimization in SPARQL”. Manuscript submitted for publication. May 2023.

[164] Benjamin Warnke et al. “SIMORA: SIMulating Open Routing protocols for Application interoperability on edge devices”. In: *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*. Taormina (Messina), Italy: IEEE, May 2022, pp. 42–49. DOI: 10.1109/icfec54809.2022.00013.

[165] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. “Hexastore”. In: *Proceedings of the VLDB Endowment* 1.1 (Aug. 2008), pp. 1008–1019. DOI: 10.14778/1453856.1453965.

[166] Christopher Welty and Deborah McGuinness. *OWL Web Ontology Language Guide*. W3C Recommendation. https://www.w3.org/TR/2004/REC-owl-guide-20040210/. W3C, Feb. 2004.

[167] H. E. Williams. “Compressing Integers for Fast File Access”. In: *The Computer Journal* 42.3 (Mar. 1999), pp. 193–201. ISSN: 0010-4620. DOI: 10.1093/comjnl/42.3.193.

[168] Shengqi Yang et al. “Towards effective partition management for large graphs”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, May 2012. DOI: 10.1145/2213836.2213895.

[169] Hong Yu and Jingsha He. “Improved hierarchical routing over 6LoWPAN”. In: *2011 IEEE 3rd International Conference on Communication Software and Networks*. IEEE, May 2011, pp. 377–380. DOI: 10.1109/iccsn.2011.6013616.

[170] Xiang Yu et al. “Reinforcement Learning with Tree-LSTM for Join Order Selection”. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2020, pp. 1297–1308. DOI: 10.1109/icde48307.2020.00116.

[171] Pingpeng Yuan et al. "TripleBit". In: *Proceedings of the VLDB Endowment* 6.7 (May 2013), pp. 517–528. ISSN: 2150-8097. DOI: 10.14778/2536349.2536352.

[172] Kai Zeng et al. "A distributed graph engine for web scale RDF data". In: *Proceedings of the VLDB Endowment* 6.4 (Feb. 2013), pp. 265–276. DOI: 10.14778/2535570.2488333.

[173] Changchun Zhang, Lei Wu, and Jing Li. "Optimizing Distributed Joins with Bloom Filters Using MapReduce". In: *Communications in Computer and Information Science*. Ed. by Tai-hoon Kim et al. Springer Berlin Heidelberg, 2012, pp. 88–95. DOI: 10.1007/978-3-642-35600-1_13.

[174] Wei Emma Zhang et al. "Learning-based SPARQL query performance modeling and prediction". In: *World Wide Web-internet and Web Information Systems* 21.4 (Oct. 2017), pp. 1015–1035. DOI: 10.1007/s11280-017-0498-1.

[175] Ying Zhang et al. "SRBench: A Streaming RDF/SPARQL Benchmark". In: *The Semantic Web – ISWC 2012*. Springer Berlin Heidelberg, 2012, pp. 641–657. DOI: 10.1007/978-3-642-35176-1_40.

[176] Chunsheng Zhu et al. "Multi-Method Data Delivery for Green Sensor-Cloud". In: *IEEE Communications Magazine* 55.5 (May 2017), pp. 176–182. DOI: 10.1109/mcom.2017.1600822.

[177] Zainab Zolaktaf, Mostafa Milani, and Rachel Pottinger. "Facilitating SQL Query Composition and Analysis". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, May 2020. DOI: 10.1145/3318464.3380602.

List of Figures

1.1 This figure shows the dependencies between the chapters of this document. . . . 5

2.1 Layer Cake of SW . . . 8
2.2 URI syntax . . . 11
2.3 URI example . . . 11
2.4 IRI example . . . 11
2.5 Graphical representation of RDF data. The blue nodes are Literals, and the red nodes are *blank nodes*. The black texts represent IRIs. . . . 13
2.6 Example TTL file. . . . 14
2.7 Example RDFS file. . . . 18
2.8 Example SHACL file. . . . 21
2.9 Example RIF file using RIF-Core. . . . 23
2.10 CPU performance and storage size over the last 70 years. CPU data modified from GitHub [135]. Storage data modified from Wikipedia [68]. This figure is incomplete and contains only data which is available to the public. . . . 24
2.11 Binary sensor data. . . . 26
2.12 Sensor data encoded in TTL. . . . 26
2.13 Possible combinations of distributed and centralized DBMS storage and processing . . . 33

3.1 Kotlin multi-platform capabilities. . . . 37
3.2 Example trie which contains the mappings test → 1, toaster → 2, toasting → 3, slow → 4 and slowly → 5 . . . 39
3.3 Index compression procedure step 1. . . . 41
3.4 Index compression procedure step 2. . . . 41
3.5 Query processing pipeline in LUPOSDATE3000. . . . 43
3.6 Dynamic programming is reusing partial solutions several times. . . . 45

4.1 SW DBMS feature overview. 47
4.2 Overview of real world data sets. 49
4.3 The figure shows the cumulative distribution function $f(X < x)$. The X-axis shows the number of triples that share the same value at the columns specified by the legend entry. The Y-axis represents the percentage of the triples in the triple store, which share their value with, at most, X triples. The names of the graphs consist of the constant values of the triple pattern. For example, the predicate graph shows the relation for triple patterns of type $?s < p > ?o$, where the predicate is a constant. 51

5.1 Structure of DBMS implementation. 54
5.2 SPARQL *A1*. 55
5.3 SPARQL *A2*. 55
5.4 An optimal number of partitions depends on the number and the selectivity of merge joins for 512 result rows. Therefore, the labels follow the form *a;b;c*, where *a* is the optimal number of partitions, *b* is the speedup compared to no partitions, and *c* is the queries per second when no partitions are used. . . . 58
5.5 An optimal number of partitions depends on the number and the selectivity of merge joins for 2048 result rows. Therefore, the labels follow the form *a;b;c*, where *a* is the optimal number of partitions, *b* is the speedup compared to no partitions, and *c* is the queries per second when no partitions are used. . . . 59
5.6 An optimal number of partitions depends on the number and the selectivity of merge joins for 8192 result rows. Therefore, the labels follow the form *a;b;c*, where *a* is the optimal number of partitions, *b* is the speedup compared to no partitions, and *c* is the queries per second when no partitions are used. . . . 60
5.7 Prediction function for the number of partitions to use 62
5.8 Performance of *A1* with ten merge joins on different DBMSs, with a result size of 128 result rows. The numbers in the brackets, for example, LUPOSDATE3000(8), show the number of used partitions. 63
5.9 Performance of *A1* with ten merge joins on different DBMSs, with a result size of 32768 result rows. The numbers in the brackets, for example, LUPOSDATE3000(8), show the number of used partitions. 64

6.1 Feature comparison of network simulators. 0: without programming effort, 1: via an interface, 2: via file descriptor, 3: via Docker 71
6.2 Classification of routing protocols. Figure adapted from [106]. 73
6.3 Network Stack Actuator interface. 78
6.4 Network Stack Middleware Interface. 78
6.5 Topologies as provided by the simulator. 80
6.6 This figure shows the SIMORA's setup time for different numbers of devices. 81

7.1 Example network layout for demonstrating DC multicast. . . 84
7.2 mapping of destinations to next hops. 84
7.3 Example for DC multicast package structure. 85
7.4 Amount of KB sent in the network. The number of messages sent (in thousands) is shown in brackets. All numbers refer to the transport layer messages sent between two devices, aggregated over all devices. The upper value represents ASP routing, and the lower is RPL routing. 88

8.1 Statistics about used topologies. DBMSs without stores are only available for topologies that are random and uniform. . . 92
8.2 Map of the city around the campus taken from OpenStreetMap [65]. A DBMS instance is placed in each parking area. The stars are used as additional devices required to connect the whole network. Like figure 2.13, the DBMS can be further classified as having or not having an attached storage. 93
8.3 The topologies as the topology generation script define them. The dark-appearing zones are parking areas. The center device and the devices in the parking areas run a DBMS instance. The other devices in the parking areas contain sensors. The remaining devices are just filling devices to connect to the network. 94

9.1 Classification of distributed query optimizers. 98
9.2 Example network topology. The nodes with a visible circle around them are edge nodes with a locally running DBMS instance. The others are just routers. 99

9.3 The routing tables as RPL uses them. The first column indicates which device owns the row. The first row shows the destinations, and the values indicate to which next hop the message should be forwarded. The values in the parentheses are added to indicate the next hop with a DBMS instance. . 99
9.4 Short overview about which and where information is used to change the join order or the location of execution. 101
9.5 Routing-assisted join order optimization in comparison to state-of-the-art static join order optimization. 102
9.6 Several bytes are sent during INSERT by topology and routing protocol. 105
9.7 Several bytes are sent during INSERT by topology and hash strategy. From top to bottom, the results for a) 4, b) 16, and c) 128 DBMSs are shown. 106
9.8 Transferred data for benchmark queries in percent. A smaller amount of data is better. Each circle shows which decision leads to how much data for which option. The decisions further inside the circle show the most significant effect. 108

10.1 The time needed to construct an optimized join tree with a given number of inputs to join in the LUPOSDATE3000 DBMS. The optimization of queries was repeated until at least 60 seconds were spent for each join size, then the average time per query was calculated. 114
10.2 Example graph to show how queries can be generated. 117
10.3 Example generated query. 118
10.4 Example SPARQL query. 118
10.5 Reinforcement learning. 119
10.6 Overview of join order optimization 120
10.7 SPARQL query of figure 10.4 transformed into a number sequence. 121
10.8 The reward function uses v_{min} and v_{max} from the statistics, which refer to the currently known best and worst-case numbers of intermediate results. $v_{current}$ may receive a *null* value when it runs into a timeout. This *null* value is not considered when calculating the known worst case. 123
10.9 SP^{2}B-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70 % of the queries receive good join trees. 125

10.10WordNet-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70 % of the queries receive good join trees. 126
10.11SP^{2}B-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. The magenta line surrounds the area, where 70 % of the queries receive good join trees. 128
10.12WordNet-Dataset. The chart shows the percentage of queries that require at most twice as many intermediate results as the known best case. 129
10.13This figure shows the reward for the model after x training steps. The displayed graphs use an average running function reset after logarithmic increasing distances. The graph names of the models contain the number of triple patterns used during training. 131
10.14All optimizers were evaluated based on the produced intermediate results. The X-axis shows which evaluation on how many triple patterns was performed. *I* means Intermediate-results, and *T* means network-Traffic. The Y-Axis shows the average factor between the best join order and the chosen join order of a given join order optimizer. A factor of 1 is achieved if the optimizer always chooses the best case. The models are trained on joins with the number of triple patterns specified in their labels. 132
10.15All optimizers were evaluated based on the network traffic. The X-axis shows which evaluation on how many triple patterns was performed. *I* means Intermediate-results, and *T* means network-Traffic. The Y-Axis shows the average factor between the best join order and the chosen join order of a given join order optimizer. A factor of 1 is achieved if the optimizer always chooses the best case. The models are trained on joins with the number of triple patterns specified in their labels. . 133

A.1 SPARQL Benchmark Query Insert. The variables enclosed in *${}* highlight which values are changed for each sensor sample. 141
A.2 SPARQL Benchmark *Q1*. Select everything. 142
A.3 SPARQL Benchmark *Q2*. Retrieve a list of all parking areas. 142
A.4 SPARQL Benchmark *Q3*. Count the number of parking spots in the parking area 9. 142

A.5 SPARQL Benchmark *Q4*. Count the number of samples from a specific parking spot. 142
A.6 SPARQL Benchmark *Q5*. Find out when the last sample from a specific sensor was sent. 143
A.7 SPARQL Benchmark *Q6*. Ask for the state of every parking spot in an area and the timestamp of its last measurement. . 143
A.8 SPARQL Benchmark *Q7*. Ask for the state of every parking spot in multiple areas and the timestamp of their last measurements. 144
A.9 SPARQL Benchmark *Q8*. Count the number of free parking spots in a specific area. 144
A.10 SPARQL Benchmark *Q9*. Join everything together to showcase many joins. 145

List of Abbreviations

6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
ACK	acknowledgement
ALM	Application Layer Multicast
AODV	Ad-hoc On-demand Distance Vector
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASP	All Shortest Path
AST	Abstract Syntax Tree
AVL	Adelson-Velskii and Landis
BLOB	Binary Large Object
BMFA	Bi-Directional Multicast Forwarding Algorithm
BMP	Basic Multilingual Plane
BMRF	Bidirectional Multicast RPL Forwarding
BSBM	Berlin SPARQL Benchmark
BTC	Billion Triples Challenge
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
CSV	comma-separated values
DAG	Directed Acyclic Graph
DAO	Destination Advertisement Object
DBLP	Digital Bibliography & Library Project
DBMS	DataBase Management System
DC	Dynamic Content

DIO	DODAG Information Object
DODAG	Destination Oriented DAG
DYMO	Dynamic MANET On Demand
ESMRF	Enhanced Stateless Multicast RPL Forwarding
FOOP	Fully Observed Optimizer based on the PPO Algorithm
GPU	graphics processing unit
HTML	Hypertext Markup Languge
HTTP	Hypertext Transfer Protocol
HPC	High Performance Computing
HYDRO	hybrid routing protocol
ICMPv6	Internet Control Message Protocol version 6
IETF	Internet Engineering Task Force
IMDb	Internet Movie Database
IP	Internet Protocol
IRI	Internationalized Resource Identifier
IoT	Internet of Things
JSON	JavaScript Object Notation
JS	JavaScript
JVM	Java Virtual Machine
LLN	Low Power and Lossy Network
LOADng	LOAD next generation
LOAD	Lightweight On-demand Ad hoc Distance-vector Routing Protocol
LUPOSDATE3000	Logisch und Physikalisch Optimierte Semantic Web Datenbank-Engine 3000
LUPOSDATE	Logisch und Physikalisch Optimierte Semantic Web Datenbank-Engine
ML	Machine Learning
MLP	MultiLayer Perceptron
MP2P	MultiPoint to Point
MPI	Message Passing Interface

MPL	Multicast Protocol for Low-Power and Lossy Networks
OF	Objective Function
OS	Operating System
OWL	Web Ontology Language
P2MP	point to multipoint
PPO	Proximal Policy Optimization
RAM	Random-Access Memory
RDBMS	Relational DBMS
RDFS	RDF Schema
RDF	Resource Description Framework
RERR	Route Errors
RIF	Rule Interchange Format
ROLL	Routing Over Low-Power and Lossy Networks
RPL	Routing Protocol for Low power and Lossy Networks
RREP	Route Replies
RREQ	Route Requests
ReJOIN	Reinforcement Learning Join Order Optimizer
ReJOOSp	Reinforcement learning for Join Order Optimization in SPARQL
SHACL	Shapes Constraint Language
SIMORA	SIMulating Open Routing protocols for Application interoperability
SIoT	Semantic Internet of Things
SMRF	Stateless Multicast RPL Forwarding
SOA	State Of the Art
SOSA	Sensor, Observation, Sample, and Actuator
SP^{2}B	SPARQL Performance Benchmark
SPARQL	SPARQL Protocol and RDF Query Language
SP2Bench	A SPARQL Performance Benchmark
SSN	Semantic Sensor Network

SQL	Structured Query Language
SW	Semantic Web
TCP	Transmission Control Protocol
TTL	Terse RDF Triple Language
UCS	universal character set
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
W3C	World Wide Web Consortium
WAN	Wide Area Network
WLAN	Wireless Local Area Network
WWW	World Wide Web
XML	Extensible Markup Language
YAGO	Yet Another Great Ontology
WSN	Wireless Sensor Network

Curriculum Vitae

Personal Information

Name	Alexander Benjamin Warnke
Birthday	February 16, 1994
Place of birth	Hamburg, Germany
Nationality	German

Professional Experience

01/2020 - Present	Research Assistant at the Institute of Information Systems, University of Lübeck
02/2017 - 10/2017	Student Assistant at the Institute for *scientific calculation* (DKRZ) at the University of Hamburg
03/2015 - 04/2015	Student Assistant at Valtech as a software developer
07/2014 - 09/2014	Voluntary internship at Valtech as a software developer

Education

10/2014 - 12/2019	University of Hamburg, Degree: M.Sc
07/2004 - 06/2014	Alstergynasium Henstedt-Ulzburg

List of Personal Publications

- Benjamin Warnke et al. “A SPARQL benchmark for distributed databases in IoT environments”. In: *Proceedings of The International Workshop on Big Data in Emergent Distributed Environments*. New York, NY, USA: ACM, June 2022. DOI: 10.1145/3530050.3532929

- Benjamin Warnke et al. “Flexible data partitioning schemes for parallel merge joins in semantic web queries”. In: *Datenbanksysteme für Business, Technologie und Web (BTW), 19. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme", Dresden, Germany*. Ed. by Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 237–256. DOI: 10.18420/btw2021-12

- Benjamin Warnke et al. “SIMORA: SIMulating Open Routing protocols for Application interoperability on edge devices”. In: *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*. Taormina (Messina), Italy: IEEE, May 2022, pp. 42–49. DOI: 10.1109/icfec54809.2022.00013

- Benjamin Warnke, Sven Groppe, and Stefan Fischer. “Distributed SPARQL queries in collaboration with the routing protocol”. In: *International Database Engineered Applications Symposium Conference (IDEAS 2023), May 5–7, 2023, Heraklion, Crete, Cyprus*. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 979-8-4007-0744-5. DOI: 10.1145/3589462.3589497

- Sven Groppe, Rico Klinckenberg, and Benjamin Warnke. “Sound of databases”. In: *Proceedings of the VLDB Endowment* 14.12 (July 2021), pp. 2695–2698. DOI: 10.14778/3476311.3476322

- Sven Groppe, Rico Klinckenberg, and Benjamin Warnke. “Generating Sound from the Processing in Semantic Web Databases”. In: *Open*

Journal of Semantic Web (OJSW) 8.1 (2021), pp. 1–27. ISSN: 2199-336X. URL: http://nbn-resolving.de/urn:nbn:de:101:1-202201161833054484370 4

- Nitin Nayak et al. "Constructing Optimal Bushy Join Trees by Solving QUBO Problems on Quantum Hardware and Simulators". In: *Proceedings of The International Workshop on Big Data in Emergent Distributed Environments*. New York, NY, USA: ACM, June 2023. DOI: 10.1145/3579142.3594298