



# Processing Linked Data on Lightweight Computing Devices

**Anh, Le-Tuan**

Submitted in fulfilment of the requirements  
for the degree of Doctor of Philosophy

**SUPERVISORS:**

**Dr. Conor Hayes**

**Dr. Danh Le Phuoc**

---

Insight Centre for Data Analysis, National University of Ireland, Galway

December 2020



*“If you optimise everything, you will always be  
unhappy.”*

Donald Knuth



# *Abstract*

The development of the Internet of Things (IoT) is enabling innovative and smarter domestic and commercial services. Due to the lack of skill and technology to make sense of IoT data, the full potential of the available information from IoT networks has not been achieved yet. It has long been recognised that standards for the interoperability and integration of data must play a key role in the value generated by IoT. Semantic technologies have been proposed to deal with data heterogeneity and to enable service interoperability in the IoT domain, and to underpin resource discovery, reasoning, and knowledge extraction. The Semantic Web research community has been attentive to the issues of data integration and interoperability raised in the IoT. Many efforts have constituted important milestones towards the integration of heterogeneous IoT platforms and applications. Linked Data model is now the preferred data model for semantic IoT data, and Resource Description Framework (RDF) engines have been proposed to be used as semantic integration gateways for the IoT.

It has been proposed that a decentralised integration paradigm fits better with the distributed nature of autonomous deployment of smart IoT devices. The idea is that moving RDF data processing closer to the edge of the network will reduce network overhead/bottlenecks and enable flexible and continuous integration of new IoT devices/data sources. The state-of-the-art RDF engines suffer performance degradation when they are required to work on IoT edge devices that are significantly under-equipped in terms of memory and CPU for supporting regular RDF engines.

In this thesis, we develop a scalable RDF engine for lightweight IoT devices, namely, RDF4Led. Our approach focuses on minimising memory consumption, maximising scalability, and processing performance. RDF4Led is designed with a RISC-style (Reduced Instruction Set Computer) design that simplifies the implementation with the purpose of reducing code footprint and memory footprint. We carefully redesign the RDF storage, indexing scheme and buffer manager to optimise flash I/O. We improve the *join* algorithm so that it significantly lowers the memory consumption used to evaluate SPARQL queries. Furthermore, we demonstrate the feasibility of RDF4Led in different application scenarios. Firstly, we show how to use RDF4Led to enable the integration of different personal data spaces on mobile devices. We also demonstrate how to integrate RDF stores and blockchain to protect the ownership of the shared RDF data on IoT edge nodes. As a result, we introduce a novel distributed RDF store which leverages blockchain benefits

for data publishers at the edges of networks. To the best of our knowledge, our system is the first of this kind. Finally, we introduce an federation mechanism to distribute the workload of RDF stream processing over IoT edge nodes. To enable such a mechanism, we integrate RDF4Led with CQELS to create Fed4Edge and RDF stream processing engine for IoT edge devices.

# *Acknowledgements*

First and foremost, I would like to thank my family, my parents, my wife, and my brother. Pursuing my studies would have been impossible without their unconditional love and encouragement. They have always been by my side no matter how far we were away from each other.

I would like to thank my two supervisors, Dr. Danh Le-Phuoc and Dr. Conor Hayes for their guidance and help. Danh has been a great teacher who has been giving me advices and directions since the first day of my journey. Conor has been a very dedicated advisor who was always patient with my regular language mistakes. I am really lucky to have you two in my career path.

I would also like to thank Prof. Manfred Hauswirth who was my master's thesis supervisor. With his support, I was awarded the IRC scholarship for this PhD thesis. As he moved to Germany, I was no longer his PhD student, however, he did not stop supporting me. I am fortunate that I can work with him and Danh in TU Berlin in the last year of my PhD.

I would also want to give many thanks to all my friends, Hoan Nguyen, Tuan Bui, Hung Nguyen, Ngoc Nguyen, Hau Bui for the great time in Galway.

I finally want to thank my colleagues in the INSIGHT Centre for Data Analytics for their help and support over the years: Martin, Aqeel, Qaiser, Yasar, Zia, Hugo, Tarek and many others. I will not forget the football matches and the board games we played together.

This work was created with the financial aid of Irish Research Council under Grant Number GOIPG/2014/917.



# Contents

<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>Contents</b>	<b>vii</b>
<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Problem Statement & Research Question . . . . .	3
1.3 Background Literature . . . . .	7
1.3.1 The Internet of Things . . . . .	7
1.3.1.1 The Semantic Web for the IoT . . . . .	8
1.3.1.2 Edge Computing in the IoT . . . . .	10
1.3.1.3 Distributed Ledger Technology and IoT . . . . .	11
1.3.2 RDF Data Management System . . . . .	12
1.3.2.1 RDF Dictionary . . . . .	12
1.3.2.2 Physical Organisation of RDF data . . . . .	13
1.3.2.3 Indexing Strategies . . . . .	15
1.3.2.4 SPARQL Processing . . . . .	16
1.3.2.5 Buffer Management . . . . .	17
1.3.2.6 RDF Stream Processing . . . . .	18
1.4 Contributions . . . . .	19
1.5 Publications . . . . .	20
1.6 Thesis structure . . . . .	22
<b>2 Pushing the Scalability of RDF Engines on IoT Edge Devices</b>	<b>23</b>
2.1 Introduction . . . . .	25
2.2 Background . . . . .	28
2.2.1 RDF and SPARQL . . . . .	28
2.2.2 Storing and Querying RDF Data . . . . .	30
2.3 Empirical Study . . . . .	32
2.3.1 Hardware Devices . . . . .	33

---

2.3.2	RDF Engines	34
2.3.3	Weather Dataset and RDF Schema	37
2.3.4	Experiment Design	38
2.3.5	Experiment Report and Findings	40
2.4	RISC-style Approach for Lightweight Edge Devices	46
2.4.1	Rationale of our System Design	46
2.4.2	Architectural View	47
2.5	Storage and Indexing	48
2.5.1	Storage Layout	48
2.5.2	Index Lookup	50
2.5.3	Writing Strategy	51
2.6	Buffer Manager	52
2.7	Adaptive Strategy for Iterative Join Execution	55
2.8	Evaluation Results	57
2.9	Conclusions	62
<b>3</b>	<b>Querying Heterogeneous Personal Information On The Go</b>	<b>63</b>
3.1	Introduction	65
3.2	Integration of Heterogeneous Personal Information	66
3.3	System Design and Implementation	70
3.3.1	System Architecture	70
3.3.2	High Performant and Low Memory Consumption RDF Store	72
3.4	Experimental Evaluation	74
3.4.1	Evaluation Setup	74
3.4.2	Evaluation Results	75
3.5	Related Work	79
3.6	Conclusions	80
<b>4</b>	<b>Incorporating Blockchain into RDF Store at The Lightweight Edge Devices</b>	<b>81</b>
4.1	Introduction	83
4.2	System Overview	83
4.3	Implementation and Deployment	86
4.4	Evaluations	88
4.5	Conclusion and Outlook	90
<b>5</b>	<b>Autonomous RDF Stream Processing for IoT Edge Devices</b>	<b>91</b>
5.1	Introduction	93
5.2	Continuous Federation with Autonomous RSP	94
5.2.1	Preliminaries: RDF Stream Processing with CQELS-QL	94
5.2.2	Dynamic subscription & discovery for autonomous RSP engines	96
5.2.3	Continuous Query Federation Mechanism	97
5.3	Design and Implementation	98
5.4	Evaluation and Analysis	101
5.4.1	Evaluation Setup	101

---

5.4.2	Experiments . . . . .	105
5.4.3	Results and Discussions . . . . .	107
5.4.4	Follow-up Challenges . . . . .	110
5.5	Related work . . . . .	111
5.6	Conclusion . . . . .	112
<b>6</b>	<b>Conclusion</b>	<b>113</b>
6.1	Contributions . . . . .	113
6.2	Future Work . . . . .	116
<b>A</b>	<b>SPARQL queries used in the experiments of Chapter 2</b>	<b>119</b>
<b>B</b>	<b>SPARQL queries used in the experiments of Chapter 3</b>	<b>125</b>
	<b>Bibliography</b>	<b>129</b>



# List of Figures

1.1	Thesis Structure . . . . .	22
2.1	The RDF schema describing the weather stations in the ISD dataset. . .	37
2.2	The RDF schema describing the weather observations in the ISD dataset.	38
2.3	Setup of the RDF data insertion and querying experiments. . . . .	39
2.4	Throughput test results of Jena TDB, RDF4J, Virtuoso on Gallileo Gen II, BeagleBone Black, Raspberry Pi Zero Raspberry Pi 2, and Raspberry Pi 3. <b>(a)</b> Insert throughput results on Gallileo Gen II; <b>(b)</b> Insert throughput results on Raspberry Pi Zero; <b>(c)</b> Insert throughput results on BeagleBone Black; <b>(d)</b> Insert throughput results on Raspberry Pi 2; <b>(e)</b> Insert throughput results on Raspberry Pi 3. . . . .	41
2.5	Querying test results of Jena TDB, RDF4J, Virtuoso and RDF4Led. <b>(a)</b> Query response time against 5 million triple dataset on Gallileo Gen II; <b>(b)</b> Query response time against 20 million triple dataset on BeagleBone Black; <b>(c)</b> Query response time against 50 million triple dataset on Raspberry Pi3.	43
2.6	Memory consumption of RDF4J, Jena TDB, Virtuoso. <b>(a)</b> Memory consumption of RDF engines in update throughput test; <b>(b)</b> Memory consumption of RDF engines in query evaluation. . . . .	45
2.7	Architecture overview of a RISC-style RDF engine. . . . .	47
2.8	Example of an SPO molecule. . . . .	49
2.9	Two-layers storage model consists of a Physical Layer and a Buffer Layer.	50
2.10	Example of a clustered write from Buffer Layer to Physical Layer . . . . .	51
2.11	Example of how data block is organised in the buffer queues. . . . .	52
2.12	Inserting throughput of RDF4Led compared to Jena TDB, RDF4J and Virtuoso on Gallileo Gen II, Raspberry Pi Zero and Raspberry Pi 3. <b>(a)</b> Insert throughput results on Gallileo Gen II; <b>(b)</b> Insert throughput results on Raspberry Pi Zero; <b>(c)</b> Insert throughput results on Raspberry Pi 3. . .	58
2.13	Query test results of Jena TDB, RDF4J, Virtuoso, and RDF4Led. <b>(a)</b> Query response time against 5 million triple dataset on Gallileo Gen II; <b>(b)</b> Query response time against 20 million triple dataset on Raspberry Pi Zero; <b>(c)</b> Query response time against 50 million triple dataset on Raspberry Pi 3.	60
2.14	Memory consumption of Jena TDB, RDF4J, Virtuoso, RDF4Led. <b>(a)</b> Memory consumption report of update throughput test; <b>(b)</b> Memory consumption report of query evaluation test. . . . .	61
3.1	Simple RDF graph integrated from data silos . . . . .	67
3.2	Consolidated and SameAs graph . . . . .	68

---

3.3	System architecture overview . . . . .	71
3.4	Update throughputs of RDF-OTG compared to TDBoid on HTC Desire, Samsung Galaxy Nexus and Nexus Tablet 7. (a) Update throughput results on HTC Desire; (b) Update throughput results on Galaxy; (c) Update throughput results on Nexus Tablet 7. . . . .	76
3.5	Query response time of RDF-OTG compared to TDBoid on HTC Desire, Samsung Galaxy Nexus and Nexus Tablet 7. (a) Query response time results on HTC Desire; (b) Query response time results on Galaxy; (c) Update throughput results on Nexus Tablet 7. . . . .	78
4.1	System overview of an incorporating of distributed RDF storage and Smart contracts. . . . .	84
4.2	General procedure . . . . .	85
4.3	Physical data organisation . . . . .	86
4.4	System deployment schema . . . . .	87
4.5	Acc. Throughput on 10-nodes cluster sizes . . . . .	88
4.6	Acc. Throughput on varying cluster sizes . . . . .	89
4.7	Query response time . . . . .	89
5.1	Overview architecture of Fed4Edge . . . . .	99
5.2	RDF schema for NCDC weather data . . . . .	101
5.3	The evaluation cluster of 85 Raspberry PI nodes . . . . .	105
5.4	Federation topologies . . . . .	106
5.5	Baseline calibration . . . . .	107
5.6	Federation topologies . . . . .	109

# List of Tables

2.1	Hardware configurations of the devices used in the experiments. . . . .	34
2.2	Characteristics of the RDF engines used in the experiments. . . . .	36
3.1	Android devices . . . . .	75
3.2	Memory consumption of mix queries/size of data . . . . .	77
3.3	Query response time (seconds) on maximum datasets for RDF-OTG . . . .	78



# Chapter 1

## Introduction

### 1.1 Motivation

The Internet of Things (IoT) proposes to connect a vast number of everyday devices (“things”) to the Internet to enable innovative and smarter domestic and commercial services ([Ashton 2009](#)). Many research initiatives have been proposed to accelerate the real-life deployment of the IoT ([Al-Fuqaha et al. 2015](#)). However, due to the lack of skills and technology to make sense of IoT data ([Bera 2019](#)), the full potential of the available information from IoT networks has not been achieved yet.

Because of the heterogeneous nature of IoT data, data integration becomes a difficult and labour-intensive task, and interoperability turns into a critical issue. It has long been recognised that standards for the interoperability and integration of data must play a key role in the value generated by IoT ([Vermesan et al. 2011](#)). Hereby, semantic technologies have been proposed to deal with data heterogeneity and to enable service interoperability in the IoT domain, and to underpin resource discovery, reasoning and knowledge extraction ([Barnaghi et al. 2012](#)).

The Semantic Web ([Berners-Lee et al. 2001](#)), known as an extension of the World Wide Web, aims to allow data to be interoperable over the Web. The Semantic Web research community has been attentive to the issues of data integration and interoperability raised in the IoT ([Androćec et al. 2018](#), [Noura et al. 2018](#)). Recent research has seen the development and deployment of ontologies to describe sensors, actuators and sensor readings ([Kaebisch et al. 2019a](#), [Haller et al. 2019](#)), semantic engines ([Gyrard et al. 2015](#)), and semantic reasoning agents ([Dell’Aglío et al. 2017](#)). These efforts have constituted important milestones towards the integration of heterogeneous IoT platforms and applications. Linked Data model is now the preferred data model for semantic IoT data ([Shi et al. 2018](#), [Le-Phuoc & Hauswirth 2018](#)); and Resource Description Framework (RDF) engines have been proposed to be used as semantic integration gateways for the IoT ([Kiljander et al. 2014](#), [Gyrard et al. 2015](#), [Desai et al. 2015](#)).

It has been proposed that a decentralised integration paradigm fits better with the distributed nature of autonomous deployment of smart IoT devices (Satyanarayanan 2017). The idea is that moving RDF data processing closer to the edge of the network will reduce network overhead/bottlenecks and enable flexible and continuous integration of new IoT devices/data sources.

The recent developments in the design of embedded hardware, e.g., ARM (Advanced RISC Machine) board (Smith 2008), led to the fact that lightweight computers have become cheaper, smaller, and more powerful. Their small size makes them easier to deploy or embed in other IoT devices placed on the edge of the network (called *edge devices*), which provides reasonable computing resources. However, the edge devices are significantly underequipped in terms of memory and CPU for supporting regular RDF engines. The state-of-the-art RDF engines suffer performance degradation when they are required to work on such devices (Le-Tuan 2016).

To process RDF data on IoT edge devices, extensive efforts will be required to optimise the computations as well as to handle the limited resources. However, if the devices process the data locally, they can be self-contained. Furthermore, data transmission costs can be dramatically reduced as it does not require the transfer of data from a device to a server or a cloud infrastructure. Working independently from a remote server also avoids the requirement for devices to maintain a frequent connection. Thus, the risks caused by intermittent connectivity can be reduced. As the device's data is not stored and processed on a remote server, the privacy and security concern is also reduced. Finally, by distributing that computation among a large number of edge devices, a greater computational scale can be achieved. These facts motivated us to investigate how to scale up RDF data processing on these IoT edge devices.

In this thesis, we develop a scalable RDF engine for lightweight IoT devices, namely, RDF4Led. Our approach focuses on minimising memory consumption, maximising scalability, and processing performance. RDF4Led is designed with a *RISC-style* (Reduced Instruction Set Computer) design that simplifies the implementation with the purpose of reducing *code footprint* and *memory footprint* (Chaudhuri & Weikum 2000). We carefully redesign the *storage*, *indexing schemes* and *buffer manager* to optimise *flash I/O*. We improve the *join* algorithm so that it significantly lowers the memory consumption used to evaluate SPARQL queries. Furthermore, we demonstrate the feasibility of RDF4Led in different application scenarios. Firstly, we show how to use RDF4Led to enable the integration of different personal data spaces on mobile devices (Le-Phuoc et al. 2014). We also demonstrate how to incorporate RDF store and *blockchain* (Swan 2015) to protect the ownership of the shared RDF data on IoT edge nodes. As a result, we introduce a novel distributed RDF store which leverages blockchain benefits for data publishers at the edges of networks (Le-Tuan et al. 2019). To the best of our knowledge, our system is the first of this kind. Finally, we introduce an *federation mechanism* to distribute the workload of RDF stream processing over IoT edge nodes. To enable such a mechanism, we integrate

RDF4Led with CQELS (Le-Phuoc et al. 2011) to create Fed4Edge (Nguyen-Duc et al. 2019) and RDF stream processing engine for IoT edge devices.

## 1.2 Problem Statement & Research Question

The motivation of the thesis leads to the broader research challenges that arise when trying to move RDF data processing to the edge of IoT networks. In this section, we state the research questions and discuss the research problems that the thesis aims to answer.

**RQ1: What are the root causes of the performance and scalability issues of the state-of-the-art RDF engines when they are required to work on IoT edge devices?**

Current state-of-the-art RDF engines are often optimised for machines with massive amounts of RAM and multiple high-performance disk arrays (Hauswirth et al. 2017). This abundance of resources enables them to store billion-triple datasets and answer complex SPARQL queries. In addition, their optimisation focuses on large, heterogeneous, and static RDF datasets. Lightweight edge devices are different from standard computing hardware in two major ways: (i) They have significantly smaller main memory and (ii) they are equipped with lightweight flash-based storage as secondary memory. Besides that, data processing on the edge of IoT networks operates in a dynamical environment with frequent data updates and changes in devices and sensors. Therefore, these RDF engines suffer from performance drawbacks on resource-constrained IoT edge devices (Le-Tuan 2016).

Similarly to designing a conventional database system, the algorithms and techniques that build an RDF engine have to be optimised to cope with the nature of the data (in this case RDF) and the particular hardware of the machine it runs on (Ullman et al. 2001, Owens 2011). Therefore, we first study how the hardware characteristics of the IoT edge devices influence the performance of an RDF engine. The results of this study would help us determine what should we optimise for an RDF engine for such lightweight devices.

**RQ2: How to create an RDF engine optimised for the hardware of IoT edge devices?**

RQ1 indicates that the scalability of PC-based RDF engines suffers dramatically when they run on lightweight edge devices. The performance degradation is due to the lack of main memory and the specific I/O behaviour of flash memory equipped with edge devices. To tackle these issues, the second research question investigates on how to create an RDF engine that is tailored built for the hardware of edge devices. To answer this research question, we divide it into subquestions to consider:

- **RQ2.1: How to reduce the code footprint and memory footprint of an RDF engine?**

On lightweight edge devices, the main memory is under-equipped to support a regular RDF engine (Le-Tuan 2016). In our study RQ1, several RDF engines can not be installed on the lightweight edge devices due to the memory limitation. Furthermore, we identify that the PC-based RDF engines, such as JenaTDB or RDF4J, do not scale well on the devices because they tend to use main memory without awareness of the limitation. Using too much memory on memory-constrained devices causes system paging behaviours or out-of-memory errors that heavily penalise the performance. Hence, reducing the code footprint and the memory footprint of an RDF engine is a vital requirement for creating a tailored-built RDF engine for edge devices.

To support the small footprint requirement, it is important to include only the (database) functionalities that are required (Nori 2007). In computer architecture, the RISC (Reduced Instruction Set Computer) philosophy is to keep the hardware simple by reducing the complexity of the instruction set and to provide only necessary instructions. Inspired by the RISC philosophy, Chaudhuri & Weikum (2000) indicates that the use of RISC-style could reduce coupling among the components of a database system. Moreover, Neumann & Weikum (2008) shows that RISC-style features necessary for an RDF engine can be implemented around data access and *join* operations. The processing load and resource consumption of an RDF engine are mainly caused by these operations. Thus, we investigate an RISC-style design RDF engine that focuses on optimising data accesses and join operations to boost the performance of the engine. The other components of the engine can be simplified to reduce its code footprint and memory footprint.

- **RQ2.2: How to create a flash-friendly physical storage and buffer manager for RDF data?**

The study RQ1 also indicates that the inefficiency is not only due to the lack of main memory, but also due to a less efficient disk-based data indexing structure and caching mechanism on flash-based storage.

IoT edge devices are equipped with flash memory as secondary storage because it has attractive features such as small size, shock resistance, and low-power consumption. In comparison to hard disks, flash-based storage devices have faster random accesses but fail to provide fast random write (Ajwani et al. 2008). Flash memory stores information in arrays of semiconductor memory cells which are organised into pages and pages are grouped into blocks. A page is the smallest unit that can be read or written to flash memory. With flash memory, updating a single page in a block is not possible. Instead, first the the whole block must be erased and then the updated data can be written to this block. *Erase* is an operation particular to flash-based memory. Thus, *write-in-place* operations that update a single piece of data in a block consist of two operations on the entire block: *erase* and *write*. Common

indexing techniques which are designed for magnetic disks do not manage well this *erase-before-write* limitation. As a result, they suffer from slow write performance when managing data on flash memory (Ajwani et al. 2008). For instance, the commonly used B+tree indexing structure in RDF triple stores does not work well on flash-based storage (Ho & Park 2016).

The performance of random write can be improved by aligning writes to blocks (Ajwani et al. 2008) and applying appropriate buffer management techniques (Jin et al. 2012). Careful design of physical storage, indexing schema and buffer management is vital for the performance of data accesses of an RDF engine (Owens 2011). Therefore, we explore how to apply these principles to build a flash-friendly physical storage and flash-friendly buffer manager for our RDF engine.

- **RQ2.3: How to minimise memory consumption for SPARQL?**

The most resource-intensive task of answering a SPARQL query is to perform the *graph pattern matching* over the RDF dataset. The graph matching operator executes a series of join operations between RDF triples that match the triple patterns. Join operations have the greatest impact on the overall performance of a SPARQL query engine, typically requiring a large number of comparison operations that can only be done efficiently if records are stored in memory. The join performance can be tuned by optimisation algorithms which plan the optimal join order and join algorithms (Neumann & Weikum 2008, Tsialiamanis et al. 2012). These approaches assume that memory is always available during the execution of a chosen query plan. However, in light-weight computing devices, memory is critically low and, as such, the memory resources available to an RDF engine are unreliable, e.g., a surge in the number of network connections to the device might drain the available memory for all other running processes. Lack of memory may block join operations that require temporary virtual memory such as *hash joins* or *sorted-merge joins* and thus hurts the overall performance of the query engine or probably crashes the engine.

*Materialisation* techniques that write intermediate join results to storage are an attractive solution for the issue of memory shortage (Ullman et al. 2001). However, on flash storage, writing is much slower if a random write operation happens. Furthermore, only a limited number of erase operations can be applied to a block of flash memory before it becomes unreliable and fails. Therefore, to adapt to the low memory available on IoT edge devices, it is required to investigate a join algorithm that minimises the memory used for executing a SPARQL query and makes the best use of our indexing scheme.

**RQ3: How feasible would it be to enable such an RDF engine for IoT edge devices?**

The second research question concerns how to enable a scalable RDF engine on IoT edge devices. The third research question aims to measure the usefulness of such an RDF

engine on IoT edge devices. Therefore, we investigate three applications of our RDF engine on IoT edge devices: (i) integration of personal data from heterogeneous sources on mobile devices; (ii) incorporating RDF storage with blockchain on IoT edge devices; (iii) cooperative federation RDF stream processing over the edge network.

- **RQ3.1: How to use an RDF engine to enable the integration of different personal data spaces on mobile devices?**

Integrating with *personal information* on mobile devices is becoming an essential requirement for many IoT applications (Miranda et al. 2015, Atzori et al. 2017). Therefore, it is required to maintain an up-to-date, comprehensive, and consolidated view of personal information across heterogeneous personal information spaces on mobile devices Le-Phuoc et al. (2014).

To integrate personal data from different data sources, several approaches proposed a unified data model for transforming heterogeneous data formats to RDF underpinned by agreed-upon vocabularies (FOAF, SIOC, vCard, etc) (Bojars et al. 2008). The RDF data model can be used to describe people and link their social relationships and the content relevant to them. As mobile devices have a lot in common with IoT edge devices in terms of hardware characteristics, we are able to apply the same approach to enable a scalable RDF engine for mobile devices.

However, personal identifier consolidation is challenging when integrating personal data from multiple data sources with the Linked Data approach. A person may have multiple identifiers (IDs) on different personal data spaces. When they are integrated in a single data space, these IDs have to be interlinked and unified to represent a unique person. A solution for this problem is to use entailment regimes<sup>1</sup> or to apply ID consolidation approach (Dong & Halevy 2005). However, mobile devices do not have enough memory and CPU for the reasoning process. Therefore, a data normalisation approach that can deal with ID-consolidation and ambiguity issues without complex generic reasoning is required.

- **RQ3.2: How to enable the incorporation of distributed RDF stores and blockchain to protect the ownership of RDF data on IoT edge devices?**

RDF stores provide a simple abstraction for publishing and querying data, which is becoming a norm in data sharing practice. Enabling RDF stores on edge devices empowers the decentralised architecture of data publishing for IoT-driven systems. However, how the owners of edge devices can control and benefit their published data is still unclear. To take control of one's data, there must be an ability to accurately account for ownership, and similarly keep a record of all transactions, exchanges, and permissions. Regarding this challenge, distributed ledger technology can provide a layer of transparency and accountability for data ownership and transactions. Thus, the arrival of blockchain technology promises to allow users to

---

<sup>1</sup><https://www.w3.org/TR/sparql11-entailment/>

control who uses their data, when their data is used and their compensation for it. Therefore, there are emerging interests in marrying RDF stores and blockchain technology to realise the desirable but speculative benefits of distributed ledger-powered data sharing at the edge of the network.

- **RQ3.3: How to enable an RDF stream processing (RSP) engine on an IoT edge node and how feasible it is to create a cooperative federation mechanism for RSP over an edge network?**

RDF Stream Processing (RSP) (Sakr et al. 2018) extends the RDF data model, enabling the capture and processing of heterogeneous streaming sensor sources under a unified data model. An RSP engine usually supports a continuous query language based on SPARQL, for example, CQELS-QL (Le-Phuoc et al. 2011) or C-SPARQL (Barbieri et al. 2009). Hence, an edge device equipped with an RSP engine could play the role of an autonomous data processing gateway. Such an autonomous gateway can coordinate the actions with other peers connected to it to execute a data processing pipe in a collaborative fashion. However, there has not been any in-depth study on how such a decentralised processing paradigm would work with edge devices. Therefore, in line with the research on pushing RDF data processing to edge networks, we investigate how to realise this edge computing paradigm by extending an RSP engine, in this case CQELS (Le-Phuoc et al. 2011), as a continuous query federation engine to enable a decentralised computation architecture for IoT edge devices.

## 1.3 Background Literature

### 1.3.1 The Internet of Things

The term “*Internet of Things*” (IoT) was first used in a presentation by Kevin Ashton when presenting his work in The Auto-ID Lab<sup>2</sup>. His idea was to link the RFID (radio frequency identification) system to the Internet. Ashton (2009) mentioned “The Internet of Things has the potential to change the world, just as the Internet did. Maybe even more so”. Since then, the IoT has gained much attention from researchers in both academia and industry. IoT applications such as *smart city* (Arasteh et al. 2016), *intelligent agriculture* (Zamora-Izquierdo et al. 2019) or *smart healthcare* (Baker et al. 2017) have made a profound impact in business and society (Langley et al. 2020).

The IoT does not have a unique definition; several research communities and business alliances have defined the IoT in their own terms. For example, the ITU (International Telecommunication Union) states that “The IoT is a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual)

---

<sup>2</sup><https://autoidlabs.org/>

things based on existing and evolving interoperable information and communication technologies”<sup>3</sup>. Whereas, Gartner defines the IoT as “the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment”<sup>4</sup>. Hence, the IoT refers to an ecosystem of interconnected physical objects that are accessible through the Internet. The physical objects perceive their surrounding environment, connect and exchange data among each other, and communicate with people via computer-based systems.

To build up the IoT, a variety of researchers, business alliances, and standardisation bodies have approached the initiative from different perspectives (Minerva et al. 2015). Atzori et al. (2010) summarised that the IoT is a result of the convergence of three main visions: *thing-oriented vision*, *Internet-oriented vision*, and *semantic-oriented vision*. The thing-oriented vision concentrates on developing things such as sensors, actuators, and smart objects, as well as methods for registering, tracing, and collecting their contexts (locations, status, etc.). The main focus of the Internet-oriented vision is to enable connectivity and the Internet for resource-constrained devices, for example, developing lower-power wireless technologies (Chettri & Bera 2019) or simplifying the Internet Protocol (IP) to adapt to resource-constrained devices (Gershenfeld & Cohen 2006, Dunkels & Vasseur 2008). The semantic-oriented vision addresses how heterogeneous objects and their generated data can be managed. Their efforts focus on developing standards for describing objects and their data, hence, they can interoperate and their data can be easily processed and analysed (Ganzha et al. 2017).

### 1.3.1.1 The Semantic Web for the IoT

The key to the power of the IoT is the ability to provide real-time data from many different sources (Aggarwal et al. 2013). One major challenge is that the underlying data from different sources is heterogeneous, very large scale, and distributed. It is difficult for data consumers to use the IoT data if the data is not clearly described. Thus, there are required frameworks to describe IoT data in a sufficient intuitive way, so that the data becomes easily usable. Such description and querying frameworks have been offered by the Semantic Web (Berners-Lee et al. 2001).

The Semantic Web technologies developed by the World Wide Web Consortium (W3C) such as Resource Description Framework (RDF), SPARQL, and Web Ontology Language (OWL) originally have been aimed to allow data to be interoperable over the web. RDF is a graph-based data model and SPARQL is the query language to query RDF dataset. The details of these technologies and examples of their usage in the IoT are described in Chapter 2. OWL is a language for representing an ontology which is “an explicit, machine-readable specification of a shared conceptualization” (Studer et al. 1998). The common components of an ontology include: (i) *Individuals (or instances)* which define

<sup>3</sup><https://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx>

<sup>4</sup><https://www.gartner.com/en/information-technology/glossary/internet-of-things>

concrete objects, things such as devices and sensors; (ii) *Classes* that define abstract groups, sets, or collections of objects, things; (iii) *Attributes* of an object which are the things that the object relates to. An attribute can be a class or an individual; (iv) *Relations* which specify how an object is related to other objects. In general, the set of relationships define the semantics in a domain of interest; (v) *Events* represent the change of the attributes of the relationship.

Recently, there has been a clear indication that using Semantic Web solutions for IoT interoperability is gaining attention from the IoT research community (Seydoux et al. 2017, Noura et al. 2018). There have been several IoT research projects which apply semantic technologies to improve semantic interoperability, such as SPITFIRE (Pfisterer et al. 2011), OpenIoT (Soldatos et al. 2015) and FIESTA-IoT (Serrano et al. 2017). Moreover, based on the idea of “sensing of a service” which enables the access to sensor data through standard service technologies, De et al. (2011) proposed a semantic description model for the services to promote the interoperability among them. Hence, the primary motivations to use semantics in IoT interoperability research is to provide resource and service descriptions (Andročec et al. 2018).

The Semantic Web technologies have been integrated into the IoT domain for various purposes, for example, description and search of things and IoT services (Rhayem et al. 2020). Many ontologies have been specifically developed for the IoT, such as W3C Semantic Sensor Network (SSN) (Haller et al. 2019), SAREF (Daniele et al. 2015), and OpenIoT (Soldatos et al. 2015). There have been also many ready-to-use ontologies in various domains such as healthcare, transportation, and logistics (Ganzha et al. 2017). Among the existing ontologies in IoT domain, the SSN ontology has been the strongest adoption and has inspired other projects (Ganzha et al. 2017).

Web of Things (WoT) is a refinement of the IoT that integrates the IoT and Web technologies, and it aims to enable accesses to the IoT via Web interfaces (Guinard & Trifa 2009). The WoT architecture and WoT building blocks have been standardised by the W3C. To solve the interoperability issues for WoT applications and platforms, the WoT has been integrated with the Semantic Web standards (Gyrard et al. 2017). For example, the WoT Thing Description building block uses the SSN ontology to describe the semantic of a specific IoT device and how it interacts with other IoT devices (Charpenay et al. 2016).

Since the IoT environment has intrinsic characteristics, for example, the constraints of IoT devices and the distributed nature of IoT deployments, there have been several efforts to adapt the Semantic Web technologies to the IoT environment. For example, Su et al. (2015) compared the resource consumption of different serialisation formats of RDF data with respect to the size, the encoded messages in each format, and the CPU cycle required to produce these messages. To reduce the communication cost, Loseto et al. (2016) proposed an extension of Linked Data Platform<sup>5</sup> that

---

<sup>5</sup><https://www.w3.org/TR/ldp/>

works with CoAP (Constrained Application Protocol)<sup>6</sup>. Additionally, to reduce memory consumption, Bazoobandi et al. (2015) proposed a method to compress RDF data in memory. Poslad et al. (2015) and Desai et al. (2015) suggested applying computation offloading that split an application into processes and distributed the processes to different nodes in IoT networks. Furthermore, RDF Stream Processing (RSP) (Sakr et al. 2018) extends the RDF data model, enabling the capture and processing of heterogeneous streaming sensor sources under a unified data model. An RSP engine usually supports a continuous query language based on SPARQL, e.g. C-SPARQL (Barbieri et al. 2010) and CQELS-QL (Le-Phuoc et al. 2011).

### 1.3.1.2 Edge Computing in the IoT

The growth of the IoT has led to the requirement of substantial volumes of data storage and processing capabilities. When IoT devices have limited storage and computational resources, cloud computing has virtually unlimited capabilities in terms of storage and processing power (Botta et al. 2016). Therefore, cloud computing has been integrated with the IoT to bridge the gap of the lack of storage and computation of IoT devices (Zhou et al. 2013, Truong & Dustdar 2015). However, the common integrating approaches, which directly connect IoT devices and upload all IoT data to the cloud, encounter network latency and traffic issues (Zhang et al. 2015, Shi et al. 2016). Most of IoT data can be discarded immediately after being processed. Therefore, the bandwidth used to transmit the data to the cloud may be used unnecessarily. Pushing all data into the cloud will dominate the upstream network traffic, meanwhile, broadband networks have more downstream bandwidth than upstream bandwidth. Moreover, some IoT applications require rapid response time, unfortunately, cloud-based solutions may lead these applications to be less operable due to unexpected network latency. Therefore, the edge computing paradigm that shifts the computation to the edge of the IoT network is gaining more attention.

(Shi et al. 2016) explain that *edge computing* refers to “the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services”. The authors define the “edge” as “any computing and network resources along the path between data sources and cloud data centres”. Edge-based IoT systems may include a large number of edge devices such routers, gateways, switchers, sensors and actuators placed between the end user and cloud. Edge nodes can be placed at a fixed point such as street lights, houses; or they can be mobile such as smart phones or connected vehicles. These edge nodes can compute, transmit, and store the temporal data. The end users can connect to these edge nodes to obtain applications or services. As edge nodes are placed close to end users, they can support latency-sensitive applications and services. Moreover, in the situation

---

<sup>6</sup><https://coap.technology/>

that more powerful computing and storage capabilities are required, the edge node can dedicate part of the workload to a cloud (Xu et al. 2019).

### 1.3.1.3 Distributed Ledger Technology and IoT

Edge computing can reduce network overhead and enable flexible and continuous integration of IoT devices and data sources (Satyanarayanan 2017). Nonetheless, the security and privacy issues (e.g., authentication, intrusion detection, access control, etc.) in edge computing architectures are emerging challenges for the IoT (Yu et al. 2018). Edge nodes operate in different sites, thus, it is difficult to ensure the integrity, information protection, anonymity assessment, and freshness of the original data (Hossain et al. 2015). Furthermore, different edge nodes are managed by different owners, making it difficult to account for ownership of the data produced from the edge nodes (Yang et al. 2017). Therefore, the IoT requires a mechanism that allows edge nodes to securely share information and reach consensus in a fully decentralised schema.

Distributed ledger technology (DLT) has attracted attention in recent years as an innovative approach to security and privacy (Walport et al. 2016). DLT refers to the technologies for the storage, distribution, and exchange of data between users over private or public distributed computer networks. It provides a layer of transparency and accountability for data ownership and transactions by employing immutable ledgers that cannot be modified. The distributed ledgers are responsible for maintaining the trust by tracking the ownership of different nodes in the network. Therefore, it does not require any centralized servers to manage and ensure the trust between communicating nodes.

One of the popular types of DLTs is blockchain (Pilkington 2016) which was introduced with Bitcoin cryptocurrency (Nakamoto 2008) to solve the double spending problem. The blockchain is designed as a distributed and decentralised ledger of transactions. To store a transaction in the ledger, the majority of participating nodes in the blockchain network should agree and record their consent. The transactions are grouped into blocks which are chained together. To chain the blocks together, each block encompasses a timestamp and a hash function to the previous block. The hash function validates the integrity and nonrepudiation of the data inside the block.

An interesting and well-known application of the distributed ledger technology is the smart contract (Szabo 1997) that can be used to implement the agreements between mutual parties. A smart contract simply is a program that is automatically executed when the predetermined conditions are met. The program may entail the computation of complex logics such as decision making through voting, crowd funding, and workflow management. The immutability and verifiability of smart contract conditions are securely maintained by the consensus mechanism of the distributed ledger.

On a blockchain, the deployment of a smart contract occurs by sending a transaction (Antonopoulos & Wood 2018). The transaction includes the compiled code for the

smart contract and a receiver address. Users interact with the smart contract through transactions that can invoke the smart contract. For example, a transaction might result in executing the compiled code by sending coins from one user to another. The most popular blockchain for running smart contracts is Ethereum (Wood et al. 2014). On Ethereum, smart contracts are typically written in a Turing-complete programming language called Solidity (Dannen 2017), and compiled into low-level bytecode to be executed by the Ethereum Virtual Machine (Antonopoulos & Wood 2018).

Integrating the distributed ledger technology with the IoT can bring several benefits. For instance, distributed ledger technology can provide a layer of transparency and accountability for data ownership and transactions on the edge of IoT. Therefore, this work includes an integration of RDF store and distributed ledger technology to realise the benefits of distributed ledger-powered data sharing at the edge of the network (see Chapter 4). In our work, we implemented smart contracts with Solidity and deployed and evaluated our system on a private Ethereum network.

### 1.3.2 RDF Data Management System

An RDF data management system or an RDF engine for short, is a purpose-built database for the storage and querying of RDF data. While traditional database systems require predefined schemes of data that can be asserted on them, RDF engines allow arbitrary assertion of information in the form of triples. The flexibility of RDF data model has advantages in terms of accessing and interoperating between heterogeneous data sources, however, the lack of a predefined structure generates challenges for designing the physical storage and query processing for RDF data. This section introduces background knowledge on database techniques used in RDF data management systems.

#### 1.3.2.1 RDF Dictionary

RDF data may be very verbose due to the use of string identifiers that are representing the RDF resources. Therefore, to store RDF data in a compact way, many RDF engines often normalise RDF resources into unique integers. This technique reduces the storage space required for RDF triples and makes comparisons (for joins) more efficient. The links between the original strings and encoded integers are usually stored in a data structure called *RDF dictionary* (Curé & Blin 2014).

The basic concept of RDF dictionary is to provide a bijection to map between RDF resources (URIs, literals, blank nodes) and their compact identifier IDs (e.g., integers). The RDF dictionary provides two basic operations: *string-to-id* and *id-to-string*. The string-to-id operation is often called when inserting RDF data into storage or parsing a SPARQL query. Whereas, the id-to-string operation is executed to translate back the encoded IDs in the results of a SPARQL to their original format. If the IDs are not

cached, each id-to-string operation may require a seek operation which can be costly. As a consequence, different approaches in the dictionary are used for optimising the usage of the two operations. For example, SW-Store (Abadi et al. 2009), Hexastore (Weiss et al. 2008), or RDF-3X (Neumann & Weikum 2010) employ two different data structures to implement the string-to-id and id-to-string operations. A B<sup>+</sup>tree is used for the string-to-id operation, while an array is used for id-to-string operation. The array supports constant time and direct access that improve the *id look-up* performance when the query produces many results.

### 1.3.2.2 Physical Organisation of RDF data

The physical organisation of RDF data is an importance topic as it has a large impact on the read performance, write performance, and space utilisation of an RDF engine. In this section, we provide an overview of the storage organisation of the state-of-the-art RDF engines.

A straightforward approach for RDF data storage is to store RDF statements in a table with a three-column schema (a column for the subject, predicate, and object respectively). This approach is commonly used in the world of RDF engines. The popular systems such as 3Store (Stephen & Nicholas 2003), Jena (Wilkinson et al. 2003), and Redland (Beckett 2002), all store RDF data in back-end databases that utilise table-like data structures. 3Store is built on top of the MySQL relational engine; and to query the data, it translates SPARQL queries into SQL queries. Meanwhile, Jena, Redland implement their own databases. This approach is relatively simple to implement, however, they scale poorly as complex queries with multiple triple patterns require many *self-joins* over a single large table.

To alleviate the problem of storing RDF statements in a single table, Wilkinson (2006) and Broekstra et al. (2002) introduced the concept of *property tables*. In this approach, a separate table is assigned to every property; and RDF statements are grouped and stored in the tables associated with their properties. This approach was used in several systems such as Sesame (Broekstra et al. 2002), Jena2 (Wilkinson 2006), and 4Store (Harris et al. 2009). While this model can reduce the number of self-joins, it has several disadvantages. For instance, it requires the information of the schema of RDF data to be predefined. Furthermore, adding a new property requires adding a new table. Therefore, the flexibility of the schema is lost, and this approach limits the benefits of using RDF.

SW-Store (Abadi et al. 2007) presented an alternative approach to the property table named *vertical partitioning* that speeds up the queries. SW-Store organises RDF statements in  $n$  two-column tables, where  $n$  is the number of unique properties in the data. The first column stores the subjects that are described by that property, while the second column stores the object values. SW-Store uses a column-oriented database C-store (Stonebraker et al. 2005) to maintain the tables.

As the RDF data model is a graph data model, it was also suggested to store and process RDF data by graph data structure and algorithms. The systems following the graph-based idea include TripleT (Fletcher & Beck 2009), gStore (Zou et al. 2014), Diplodocus (Wylot & Cudré-Mauroux 2015). However, many graph algorithms are known to be complex in terms of implementation and optimisation.

There have been several ideas that use bit matrix structure to create compact representations for RDF data. For example, BitMat (Atre et al. 2010) uses three-dimensional bit matrix for the compact representation of RDF data in memory. In this approach, an RDF dataset is stored in a three-dimensional (subject, predicate, object) bit matrix. An RDF triple in the RDF dataset is considered a three-dimensional element in the matrix. The element is a bit encoding the presence or absence of the RDF triple in the RDF dataset. SPARQL queries over the RDF dataset can be processed by using bit-wise AND, OR operations on the bit matrix. Meanwhile, TripleBit (Yuan et al. 2013) uses two-dimensional bit matrix storage structure. In this approach, each column corresponds to a predicate and rows are subjects or objects. In the matrix, a set bit denotes the presence of a subject-predicate or object-predicate pair. There are only two set bits per column. This storage structure enables TripleBit to use merge joins extensively for processing SPARQL queries.

The RDF engines, such as YARS (Yet Another RDF Store) (Harth & Decker 2005), Hexastore (Weiss et al. 2008), and RDF-3X (Neumann & Weikum 2008), employ an index-permuted storage system. The main idea of this approach is that providing indexes on the subject, predicate, and object of a triple in different orders (permutations) allows fast access to any part of the triple. In these systems, the string representations of RDF resources are replaced with unique integers using RDF dictionary technique. Indexes are built on the shorter encoded values and cover all major access types to triple query patterns (see Section 2.2). For example, the object variable triple query pattern with bound subject and predicate (subject predicate ?object) can be searched on the subject-predicate-object index. Hexastore and RDF-3X maintain all six possible permutations of subject, predicate, and object in six separate indexes. However, using three indexes on three cyclic orderings of a triple's components can cover all triple query patterns (Harth & Decker 2005).

Moreover, there have been several works to store RDF data in distributed database systems and in cloud infrastructures. For example, Jena-HBase (Khadilkar et al. 2012) integrates Jena and Apache Hadoop<sup>7</sup>, whereas AMADA (Aranda-Andújar et al. 2012) is an RDF data management platform based on Amazon Web Services.

---

<sup>7</sup><https://hadoop.apache.org/>

### 1.3.2.3 Indexing Strategies

Indexing is a search data structure which allows efficient retrieval of specific pieces of data from a dataset. Organising RDF triples in optimal structures, as presented in the previous section, gives access to the triples that match a triple pattern more efficiently. There are often many triples that match a triple pattern. Hence, if it is required to scan the whole dataset to find each matched triple, the search still remains very slow. Therefore, efficient indexing strategies for storing RDF triples are required.

Different indexing strategies can be used for different purposes. For example, hash maps or hash tables are data structures that are commonly used for indexing data in memory-based storage (Ullman et al. 2001). A part of the data is hashed and the hash value is used as the key search for this data. The data is stored at the memory location corresponding to the hash value. Hash-based data structures offer good insert and search performance as their time complexity is  $O(1)$ . RDF engines, such as Sesame (Broekstra et al. 2002) or Jena (Seaborne 2010) use hash maps to store RDF data in the main memory.

Hash-based data structures are not sufficient to build disk-based RDF storage. SPARQL query evaluation requires matching RDF triples to the RDF query patterns. As hash-based indexing does not guarantee two similar items to be stored next to each other. RDF triples that match a triple query pattern are not guaranteed to be stored in the same disk block or a sequence of disk blocks. Consequently, the number of disk seek and disk read operators is unnecessarily high when retrieving the matched RDF triples.

The B-tree is commonly used as the index structure for disk-based storage (Ullman et al. 2001). A B-tree refers to a self-balancing tree data structure in which each node can have more than two child nodes. The largest number of child nodes that a node can have is called *fanout*. The fanout of a B-tree defines the height of the tree, which is proportionate to  $\log_{fanout}$ . In practice, the size of a node in the tree is kept equal to the size of a data block in the disk. Thus, when a node is accessed, its entire contents are read into the main memory. As the disk block size is quite large, the height of the tree is often low. The number of disk seek operations required to search for an item within a B-tree equals the height of the tree. Thus, only a few disk seek operations are required for a search on a B-tree.

The B<sup>+</sup>-tree (Comer 1979) is a variant of B-tree that stores the pointers to actual data in the leaf nodes, and leaf nodes are linked to allow sequential traversal over them. Thus, a B<sup>+</sup>-tree supports range searching much better than hash-based data structure (Ullman et al. 2001). As a result, B<sup>+</sup>-tree is used exclusively in many RDF engines that support storing data persistently on the hard disk, for example, JenaTDB (Owens et al. 2008), Hexastore (Weiss et al. 2008) and RDF3X (Neumann & Weikum 2008).

Furthermore, the choice of an indexing data structure of a data management system is also influenced by the storage mechanism (e.g., RAM, disk-based storage or flash-based storage) that the system runs on. Unfortunately, B-tree variants do not work

well on flash-based storage which is often used in IoT edge devices (Ho & Park 2016). Flash memory has erase-before-write limitations that makes random writes on flash-based storage inefficient (Ajwani et al. 2008). The frequent random writes required for a B-tree degrades its efficiency on flash-based storage. The main principle on the design of flash memory access methods is avoiding in-place updates and random writes (Ajwani et al. 2008). Several works, such as LA-tree (Agrawal et al. 2009) or FD-tree (Li et al. 2010) have applied this principle to improve the performance of tree data structure for flash-based storage. In our work, we also apply the same principle to design the index for RDF4Led (see Section 2.5). We organise the data into block units whose size is equal to the flash-erase block size. We use an in-memory caching mechanism to cache the atomic data and to cluster the write operations to improve the write performance. However, to minimise the amount of memory required to maintain the indexes, we use the Block Range Index (BRIN) approach <sup>8</sup>.

#### 1.3.2.4 SPARQL Processing

SPARQL is a query language which was developed to query RDF datasets. Since RDF is a graph-based data model, SPARQL is designed as a graph-matching query language. SPARQL queries include graph query patterns, conjunctions, disjunctions, and optional patterns as formalised in (Arenas & Pérez 2011). We briefly introduce SPARQL query in Section 2.2.

A SPARQL query is processed in the following sequential steps. Firstly, the SPARQL query string is translated into an internal format that the query processor can further process. This step is known as query parsing, and the internal format is often in the form of an algebra operator tree. In this step, an RDF dictionary may be used to replace the string representations of the RDF resources in triple query patterns. The second step is to select the candidate physical operators for query operations such as index scanning or joining. The physical operators are selected by considering the information such as the physical data structure on disk or the availability of indexes to speed up the operation. In the final step, a set of potential query plans are created. The cost for executing each query plan is defined by a cost model. The cost may be weighted by, for example, the number of disk accesses required, memory usage, or time delay. The query optimiser picks the query plan that is best suited to the optimisation goal.

Many join algorithms can be used to implement the join operators of a SPARQL query processor (Owens 2011). For example, Jena TDB (Tuple Database) <sup>9</sup> implements *index nested loop join* algorithm which joins two sets by using two nested loops (DeWitt et al. 1993). For each item in one set (called the outer input), it searches the entire other set (called the inner input) to find the compatible items. With the available index in the inner input, the search can speed up as there needs only a few comparisons instead of

---

<sup>8</sup><https://www.postgresql.org/docs/9.5/brin-intro.html>

<sup>9</sup><https://jena.apache.org/documentation/tdb/>

scanning the whole inner set. The index nested loop join algorithm is especially effective in the situation that the outer input is small and the inner input is very large. As it does not require temporary virtual memory, the index nested loop join is very useful for applications that are required to optimise memory consumption (Graefe 1993).

*Hash join* is a join algorithm that is performed using a hash table (Ullman et al. 2001). This algorithm is executed in two phases: build phase and probe phase. In the build phase, it creates a hash table to index the first input. In the second phase, it probes each item in the second input and compares against the hash table to produce the output. In both phases, join attributes are hashed and used as the key to create the hash table and for comparison. This algorithm does not require indexing of inputs and scales in a linear manner with the size of the inputs (Ullman et al. 2001).

*Sort merge join* algorithm requires the items in two input sets to be sorted by the join attributes (Ullman et al. 2001). Therefore, the join can be executed by scanning of both inputs and comparing on the join attributes to produce join results. Sort merge join is always faster than hash join or nested loop join if the inputs are already sorted correctly. However, in the case where the inputs have to be sorted, the performance of this algorithm largely relies on the performance of the sort. The sort merge join algorithm was used in the implementation of Hexastore (Weiss et al. 2008) and RDF-3X (Neumann & Weikum 2008). They used all six possible index permutations to make the greatest use of the merge join over the sorted index list.

Selecting the best suited algorithm for implementing the join operators for a query processor is clearly important. Different join algorithms require different amounts of resources for their optimal performance (Owens 2011). These join operations contribute the most to the time and resource consumption for the evaluation of the SPARQL queries. The state-of-the-art query planners and query optimisations often select join operators and join order without taking resource consumption (memory consumption) into account. Of course, in unlimited capability, storage and processing power environments such as cloud infrastructure, minimising the time spent on the join is the highest priority. However, for resource-constrained IoT edge devices, the resource consumption of the joins must be considered first because inappropriate resource usage may result in system failure. Therefore, to reduce the computational cost for RDF4Led, we use a nested execution model (Graefe 2003) to avoid caching the intermediate results of joins. We process the joins in one-tuple-at-a-time fashion (Avnur & Hellerstein 2000). In each run, we can adaptively choose the next triple pattern to probe and scan (see Section 2.7).

### 1.3.2.5 Buffer Management

In a database system, the buffer manager is responsible for caching data in the main memory to reduce the number of disk I/O, and it guarantees that the cached data fit in the available main memory (Ullman et al. 2001). Thus, the buffer manager helps a

database system to avoid reading and writing a data block from memory to disk multiple times, and to keep the system safe from out-of-memory errors.

Data in the buffer manager is often organised into fixed-size memory blocks. The size of a memory block equals to the size of a disk block. When the system requests a data block (which is often indexed), the buffer manager checks if the requested data block is in the main memory. If the requested data block is not found in the main memory, then it sends a read request to the physical storage. If free space is needed to hold the requested data block, the buffer manager releases a buffered data block from the main memory. If the buffered data in the block has not been changed (*clean* block), the data is simply evicted from this memory block. On the other hand, if the buffered data has been updated (*dirty* block), the data is rewritten back to disk.

The buffer management employs *replacement policies* to decide which data block in the memory is set free to make space for newly requested data. For example, the least recently used (LRU) policy releases the block that has not been read or written to for the longest time (Ullman et al. 2001). The idea of the policy is that the more recently accessed data blocks have a higher probability to be accessed in the future.

Flash-based storage devices also require new buffer replacement algorithms (Graefe 2007). *Clean First LRU (CFLRU)* is an alternative version of LRU that priorities the release of the clean blocks first as the writing cost for these blocks is zero (Park et al. 2006). *Clean-First Dirty-Clustered (CFDC)* enhances the write performance of CFLRU by clustering dirty blocks into single writes (Ou et al. 2009). Adaptive Double LRU (AD-LRU) (Jin et al. 2012) aims to prioritise both the frequency and recency by storing data blocks in two LRU queues: a hot queue and a cold queue. The recently accessed data blocks are kept in the hot queue. If the hot queue is full, it moves the least frequently accessed data block to the cold queue. If the cold queue is full, it writes the least frequently accessed data block to disk.

In this thesis, we use an approach similar to AD-LRU to build a flash-friendly buffer manager for RDF4Led (see Section 2.6). We also organise data blocks in a hot queue and a cold queue. However, we use an alternative *replacement policy* to make the buffer manager suitable for RDF data.

### 1.3.2.6 RDF Stream Processing

A data stream is an unbounded and continuously generated sequence of time-stamped data from a data source. The RDF stream is modelled by extending the definition of RDF graph (see Section 2.2) with temporal annotations which are point-based labels or interval-based labels. The point-based label is used to annotate that an RDF graph is valid at a timestamp. Whereas, interval-based label is used to express that an RDF graph is valid in a time window (Dell'Aglio et al. 2016).

Several continuous query languages were developed for RDF stream data, for example, Stream SPARQL (Bolles et al. 2008), C-SPARQL (Barbieri et al. 2010), and CQELS-QL (Le-Phuoc et al. 2011). The query languages extend SPARQL by adding new syntaxes to support continuous queries. For example, Stream SPARQL extends the SPARQL 1.0 grammar by adding *DataStreamClause* and a clause for defining time windows. CQELS-QL defines its grammar based on SPARQL 1.1 grammar and supports generating RDF streams from the query output by using the CONSTRUCT keyword.

The design of RDF stream processing engine (RSP engine) can be classified into two categories: *black box* and *white box* (Le-Phuoc et al. 2011). The RSP systems that use a black box architecture delegate the processing to other stream/event processing systems. They have to translate their query language to the query language of the back end system. For instance, C-SPARQL is built on top of ESPER<sup>10</sup>. It has to transform RDF streams to relational streams before any further processing. Building a RSP following the black box architecture can reduce the development cost, however, this kind of system has difficulty in optimising as it does not have full control of the back end system.

On the other hand, CQELS (Le-Phuoc et al. 2011), uses a white box architecture. It defines its own native processing model and physical operators, such as sliding window, join and triple pattern matching. CQELS provides a flexible query execution framework with the query processor dynamically adapting to changes in the input data. Furthermore, using the white box architecture allows the RSP the flexibility to be extended (Le-Phuoc et al. 2015). For example, in a version of CQELS for cloud computing infrastructure, it uses Apache Storm<sup>11</sup> and Apache HBase<sup>12</sup> for coordinating parallel execution processes. In this thesis, we integrate CQELS with RDF4Led to a lightweight RSP for IoT edge devices called Fed4Edge (Nguyen-Duc et al. 2019).

## 1.4 Contributions

The most significant contributions of the research presented in this thesis can be summarised as follows:

- **An empirical study of PC-based RDF engines running on IoT edge devices (Chapter 2).**

To answer RQ1, we conduct a performance analysis of PC-based RDF engines when they run on single board computers that are representative of IoT edge device hardware configurations. The study shows that PC-based RDF engines suffer performance issues. The analysis indicates that the lack of memory and the specific I/O behaviour of flash-based storage negatively influence the algorithms designed

---

<sup>10</sup><https://www.espertech.com/esper/>

<sup>11</sup><https://storm.apache.org/>

<sup>12</sup><https://hbase.apache.org/>

for PC-based RDF engine. This study is published in (Le-Tuan 2016) and (Le-Tuan et al. 2020).

- **An RDF engine for lightweight edge devices (Chapter 2).**

To overcome the shortcomings found in RQ1, we develop RDF4Led, a lightweight RDF engine that is tailored built for IoT edge devices. RDF4Led is implemented on a RISC-style architecture to minimise the memory footprint and code footprint (RQ2.1). To boost the read/write performance of RDF4Led, we concentrate on developing a flash-friendly RDF physical storage and flash-friendly RDF data buffer management (RQ2.2). Furthermore, we implement a join algorithm that optimises the memory consumption when executing a SPARQL query (RQ2.3). This work was published in (Le-Tuan et al. 2018) and (Le-Tuan et al. 2020).

- **A framework for personal data integration on mobile phones based on a lightweight RDF engine (Chapter 3).**

To prove the feasibility of our RDF engine, we develop and evaluate a framework for integrating personal data from heterogeneous data sources on mobile devices (RQ3.1). The integration framework is built upon Linked Data technology, and our RDF engine is used as the core component that performs the data integrating tasks. We implement an alternative data normalisation approach that can deal with ID consolidation without using a complex generic reasoner. The framework was published in (Le-Phuoc et al. 2014).

- **An integration of blockchain and distributed RDF store (Chapter 4)**

We develop the first prototype of the integration of distributed RDF store and blockchain technology for the decentralised edge network (RQ3.2). We integrate RDF4Led with the distributed file system IPFS to enable the share of RDF data across edge nodes. We use a Ethereum blockchain network to protect the ownership of the shared RDF data. We published this work in (Le-Tuan et al. 2019).

- **A federated RDF stream processing engine for IoT edge devices (Chapter 5)**

We integrate RDF4Led and CQESL (Le-Phuoc et al. 2011) to create Fed4Edge, a federated RDF stream processing for IoT edge devices (RQ3.3). Fed4Edge enables the coordination of edge devices' resources to process query processing pipelines by cooperatively delegating a partial workload to their peers. This work is published in (Nguyen-Duc et al. 2019)

## 1.5 Publications

Contributions of this work have been published in peer-reviewed conference and journal papers as follows:

- Anh Le-Tuan, Conor-Hayes, Manfred Hauswirth, Danh Le-Phuoc, "Pushing the scalability of RDF engines on IoT edge devices", Journal Sensors 2020.
- Anh Le-Tuan, Darshan Hingu, Manfred Hauswirth, Danh Le-Phuoc, "Incorporating blockchain into RDF store at lightweight edge devices", International Conference on Semantic System, 2019.
- Manh Nguyen Duc, Anh Le-Tuan, Jean-Paul Calbimonte, Manfred Hauswirth, Danh Le-Phuoc, "Autonomous RDF Stream Processing for IoT Edge Devices", Joint International Semantic Technology Conference, 2019 (\*).
- Anh Le-Tuan, Conor Hayes, Marcin Wylot, Danh Le-Phuoc. "RDF4Led: an RDF engine for lightweight edge devices", International Conference on the Internet of Things 2018 (\*).
- Anh Le-Tuan. "Linked Data processing for embedded devices". Proceedings of the Doctoral Consortium at International Semantic Web Conference 2016.
- Danh Le-Phuoc, Anh Le-Tuan, Gregor Schielle, Manfred Hauswirth, "Querying heterogeneous personal information on the go", International Semantic Web Conference 2014.

(\* *Best paper nominee.*)

In addition, the following papers have been produced during the study:

- Daniele Dell'Aglio, Danh Le Phuoc, Anh Le-Tuan, Muhammed Intizar Ali, Jean-Paul Calbimonte, "On a web of data streams". Decentralizing the Semantic Web Workshop at International Semantic Web Conference 2017.
- Danh Le-Phuoc, Minh Dao-Tran, Chan Le Van, Anh Le Tuan, Manh Nguyen Duc, Tuan Tran Nhat, Manfred Hauswirth, "Platform-Agnostic Execution Framework Towards RDF Stream Processing", RDF Stream Processing Workshop at Extended Semantic Web Conference 2015.
- Danh Le Phuoc, Minh Dao-Tran, Anh Le Tuan, Manh Nguyen Duc, Manfred Hauswirth, "RDF Stream Processing with CQELS Framework for Real-time Analysis", International Conference on Distributed Event-Based Systems 2015.

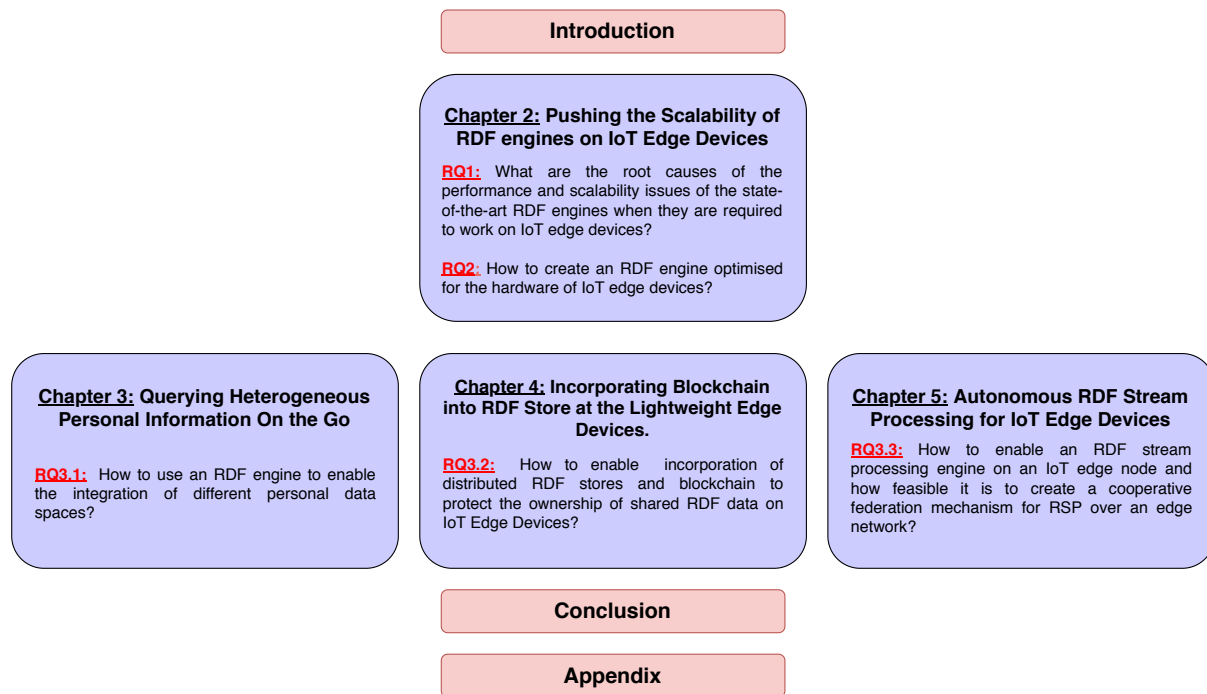


FIGURE 1.1: Thesis Structure

## 1.6 Thesis structure

The diagram in [Figure 1.1](#) outlines the broad structure of this thesis. This diagram outlines which research questions are addressed in each chapter.

The present chapter includes the justification for this research, the objectives of this research, and a background literature review on the Internet of Things and the Semantic Web. The rest of the thesis is organised as follows: [Chapter 2](#) includes the detail of our work on RDF4LED an RISC-Style RDF engine for lightweight IoT edge devices. [Chapter 3](#) presents the framework for integration heterogeneous personal information on mobile phone. [Chapter 4](#) presents the prototype to marry RDF store with blockchain. [Chapter 5](#) discusses a federation mechanism for RDF stream processing on IoT edge network. Finally, [Chapter 6](#) concludes this thesis and points out future work.

## Chapter 2

# Pushing the Scalability of RDF Engines on IoT Edge Devices

The work outlined in this chapter was published in:

**Le-Tuan, A.**, Hayes, C., Wylot, M. and Le-Phuoc, D., 2018, October. “RDF4Led: an rdf engine for lightweight edge devices”. In Proceedings of the 8th International Conference on the Internet of Things (pp. 1-8).

**Le-Tuan, A.**, Hayes, C., Hauswirth, M. and Le-Phuoc, D., 2020. “Pushing the Scalability of RDF Engines on IoT Edge Devices”. *Sensors*, 20(10), p.2788.

## Abstract

Semantic interoperability for the Internet of Things (IoT) is enabled by standards and technologies from the Semantic Web. As recent research suggests a move towards decentralised IoT architectures, we have investigated the scalability and robustness of RDF (Resource Description Framework) engines that can be embedded throughout the architecture, in particular at edge nodes. RDF processing at the edge facilitates the deployment of semantic integration gateways closer to low-level devices. Our focus is on how to enable scalable and robust RDF engines that can operate on lightweight devices. In this paper, we have first carried out an empirical study of the scalability and behaviour of solutions for RDF data management on standard computing hardware that have been ported to run on lightweight devices at the network edge. The findings of our study show that these RDF store solutions have several shortcomings on commodity ARM (Advanced RISC Machine) boards that are representative of IoT edge node hardware. Consequently, this has inspired us to introduce a lightweight RDF engine, which comprises an RDF storage and a SPARQL processor for lightweight edge devices, called RDF4Led. RDF4Led follows the RISC-style (Reduced Instruction Set Computer) design philosophy. The design constitutes a flash-aware storage structure, an indexing scheme, an alternative buffer management technique, and a low-memory-footprint join algorithm that demonstrates improved scalability and robustness over competing solutions. With a significantly smaller memory footprint, we show that RDF4Led can handle 2 to 5 times more data than popular RDF engines such as Jena TDB (Tuple Database), and RDF4J, while consuming the same amount of memory. In particular, RDF4Led requires 10%–30% memory of its competitors to operate on datasets of up to 50 million triples. On memory-constrained ARM boards, it can perform faster updates and can scale better than Jena TDB, and Virtuoso. Furthermore, we demonstrate considerably faster query operations than Jena TDB and RDF4J.

## 2.1 Introduction

The Internet of Things (IoT) proposes to connect a vast number of everyday devices (“things”) to the Internet to enable innovative and smarter domestic and commercial services (Ashton 2009). These devices range from physical objects (Mattern & Floerkemeier 2010) such as smart phones, smart watches, environmental sensors, to virtual objects (Nitti et al. 2015), such as tickets and agendas. In fact, any online service that has a unique identifier and is accessible on the Internet may be connected as a “thing” to the network. The heterogeneity of devices, services, and requirements has driven major research initiatives to accelerate the real-life deployment of IoT technologies (Akpakwu et al. 2017). In 2019, Gartner (2019) reported that 4.81 billion IoT devices were active and predicted that the number would reach 5.8 billion by the end of 2020. In 2018, nearly two and a half exabytes of data were generated every day (Marr 2018). However, it was also observed that 80% of companies lacked skills and technology to make sense of the data provided by the IoT devices (Bera 2019). Consequently, a clear challenge is how to make the best use of the substantial amount of information available from IoT networks.

It has long been recognised that standards for the integration and analysis of data must play a key role in the value generated by IoT (Vermesan et al. 2011). The Semantic Web, known as an extension of the World Wide Web, aims to allow data to be interoperable over the Web (Berners-Lee et al. 2001). Semantic technologies have been proposed to deal with data heterogeneity and to enable service interoperability in the IoT domain (Atzori et al. 2010), and to underpin resource discovery, reasoning and knowledge extraction (Barnaghi et al. 2012). Recent research has seen the development and deployment of ontologies to describe sensors, actuators and sensor readings (Kaebisch et al. 2019a), semantic engines (Gyrard et al. 2015) and semantic reasoning agents (Dell’Aglio et al. 2017). These efforts have constituted important milestones towards the integration of heterogeneous IoT platforms and applications

The Resource Description Framework (RDF) (see Section 2.2) is now the preferred data model for semantic IoT data (Shi et al. 2018, Le-Phuoc & Hauswirth 2018) and RDF engines have been used as semantic integration gateways for the IoT (Kiljander et al. 2014). While existing centralised cloud solutions offer flexible and scalable options to deal with different degrees of data integration (Gyrard et al. 2015), in the context of IoT, a cloud infrastructure as a single processing node may lead to network latency issues (Zhang et al. 2015). Directly pushing IoT data to the cloud may have several disadvantages because it is estimated that only 10% of the preprocessed IoT data is worth saving for analysis (Marr 2018), as most of the intermediate data can be discarded immediately. As broadband networks have more downstream bandwidth than upstream bandwidth, large uploads of raw sensor data, for example, may quickly dominate the upstream network traffic.

Furthermore, real-time IoT applications may suffer due to the resulting network latency between cloud and end-user devices. As such, it has been proposed that a decentralised

integration paradigm fits better with the distributed nature of autonomous deployment of smart devices (Satyanarayanan 2017). The idea is that moving the RDF engine closer to the edge of the network, and to the sensor nodes and IoT gateways, will reduce network overhead/bottlenecks and enable flexible and continuous integration of new IoT devices/data sources (Munir et al. 2017).

Thanks to recent developments in the design of embedded hardware, e.g., ARM (Advanced RISC Machine) boards (Smith 2008), lightweight computers have become cheaper, smaller, and more powerful. For example, a Raspberry Pi Zero <sup>1</sup> or a C.H.I.P computer <sup>2</sup> costs less than 15 Euros and is comparable in size to a credit card. Such devices are powerful enough to run a fully functional Linux distribution that are efficient in power consumption. Their small size makes them easier to deploy or embed in other IoT devices (e.g., sensors and actuators), which provides reasonable computing resources. Furthermore, they can be placed on the network edge, as *edge devices*, i.e., data-processing gateways that interface with outer networks. For example, such an integration gateway device may be used for an outdoor adhoc sensor network. This gateway can be easily fitted into a lamp pole on a street or at a traffic junction, sharing a power source powered by a small solar panel.

Despite their advantages in power consumption, size, and cost-effectiveness, lightweight edge devices are significantly underequipped in terms of memory and CPU for supporting regular RDF engines. Of the 100 billion ARM chips shipped so far, (Segars 2017) and even much more in the coming future, only a small fraction, e.g., 0.1%, will need a special class of RDF engine optimised for this environment. Nevertheless, this still equates to 100 million devices, which motivated us to design and build an RDF engine optimised for the hardware constraints of lightweight edge devices.

Lightweight edge devices are different from standard computing hardware in two major ways: (i) They have significantly smaller main memory and (ii) they are equipped with lightweight flash-based storage as secondary memory. To manage large static RDF datasets, standard stand-alone or cloud-based RDF engines apply sophisticated indexing mechanisms that consume a large amount of main memory and are expensive to update. Our empirical study (see Section 2.3) shows that applying the same approach to resource-constrained devices causes high numbers of page faults or out-of-memory errors, which heavily penalises the system performance. Flash memory devices are smaller in size, lighter, more shock resistant, and consume much less power than computers with disks or HDD. However, the I/O behaviour of flash-based storage, especially the erase-before-write limitation (Bouganim et al. 2009), degrades the efficiency of disk-based data indexing structure and caching mechanism (Graefe 2007). The majority of existing RDF engines do not cater for storage of this kind.

Inspired by the RISC design (Reduced Instruction Set Computer) of ARM computing boards, in this paper, we introduce a RISC-style approach similar to (Neumann & Weikum

<sup>1</sup><https://www.raspberrypi.org/products/raspberry-pi-zero/>

<sup>2</sup><http://chip.jfpossibilities.com/docs/chip-pro.html>

2008) to address these shortcomings and others found in our empirical study. In contrast to (Neumann & Weikum 2008), we focus on minimising memory consumption, scalability (maximising data processing), and processing performance. Our approach is based on a redesign of the storage and indexing schemes to optimise flash I/O. This has led to an improved *join* algorithm with significantly lower memory consumption. As a result, our RDF engine, RDF4Led, has a small code footprint (4MB) and outperforms RDF engines such as Jena TDB, and RDF4J. The experiments in Section 2.8 show that RDF4Led requires less than 30% of the memory used by competing RDF engines when operating on the same scale of data.

In summary, our contributions are as follows:

1. We intensively study the scalability of the PC-based RDF engines running on IoT lightweight edge devices.
2. We introduce a RISC-Style RDF engine design based on observations drawn from an empirical study of the performance of PC-based RDF engines running on lightweight edge devices.
3. We develop a flash-friendly indexing data structure, a flash-friendly buffer management technique and a low-memory-footprint *join* algorithm to store and query RDF data on lightweight IoT edge devices.
4. We implement our prototype in Java, and evaluate it to show the performance gains of our approach.

This paper is structured as follows: In Section 2.2, we present the fundamentals of the RDF data model, provide a short introduction to SPARQL queries, and describe how to build and RDF storage and how to query RDF data. Section 2.3 investigates the performance of RDF engines on standard hardware when they are ported to run on lightweight edge devices without optimisation. Then we introduce our RISC-style approach to build an RDF engine for lightweight edge devices, called RDF4Led, and overview its architecture in Section 2.4. In the following sections, we present in detail the design of our flash-aware storage and our new indexing structure (Section 2.5), buffer management (Section 2.6) and the algorithm for dynamically computing joins (Section 2.7). In Section 2.8, we discuss the results of our evaluation of RDF4Led with other engines on different types of devices. Finally, in Section 2.9, we present our conclusions and an outlook for future work.

## 2.2 Background

### 2.2.1 RDF and SPARQL

The Resource Description Framework (RDF) is a graph-based data model that has been developed for the representation of properties and relationships of web resources. Initially, the development of RDF was intended to contribute to the Semantic Web (Berners-Lee et al. 2001), however, its usage is now much wider than that. RDF is a promising standard to represent IoT data which usually refers to the attributes of the phenomena observed by things and the relations among things.

RDF presents data by using statements in a similar way to using natural language to express facts. A statement is given as a triple consisting of a subject, a predicate, and an object. The subject denotes an entity, the predicate denotes a property (relation), and the object denotes an entity or a value. Put simply, a triple states that a subject has a relation to the object or a subject has an attribute whose value is the object. Listing 2.1 presents an RDF example describing a wind sensor W01 and its observations, i.e., *":windSensorW01 is a sensor which observes wind speed"* and *":windSensorW01 measured the wind speed to be 30km/h at timestamp 2019-10-03T08:04:50"*. In the example, the RDF triples are represented in Turtle - a standard format for serialising RDF data. RDF can also be serialised in other standard formats like RDF/XML or JSON-LD.

```
1 :windSensorW01 a  sosa:Sensor;
2                 sosa:observes  :windSpeedRate;
3                 sosa:madeObservation  :winspeedObs01.
4 :winspeedObs01 sosa:observedProperty  :windSpeedRate;
5                 sosa:hasSimpleResult  "30 km/h";
6                 sosa:resultTime  "2019-10-03T08:04:50"^^xsd:dateTime.
```

LISTING 2.1: An example of RDF statements (in Turtle format) describing a wind speed sensor and a wind speed observation.

According to the RDF standard, in an RDF triple, the subject is an International Resource Identifier (IRI) to denote a named entity, or a blank node to express an anonymous entity. The predicate is always an IRI and the object can be an IRI, a blank node, or a literal. As a consequence, there are two types of RDF triples: literal triples for describing an entity's properties and RDF links for denoting the relationships of two entities. In Listing 2.1, the triples in line 5 and line 6 are literal triples which contain RDF literals as their objects. RDF literals can be basic or complex, e.g., integer, float, or datetime, to define values such as string, number or time. RDF links, on the other hand, consist of three IRIs. The predicate IRI connecting two entities has a type to describe the relationships between two things. These types are defined in ontologies. For example, triples in lines 1-4 are RDF links, and the relationships are defined in the SSN ontology (Haller

et al. 2019). Hence, an RDF dataset or an RDF Graph is a set of RDF triples  $\langle s, p, o \rangle \in (I \cup B) \times I \times (I \cup B \cup L)$ , where I, B and L are sets of IRIs, blank nodes and literals.

SPARQL is a query language which was developed to query RDF datasets. Since RDF is a graph-based data model, SPARQL is designed as a graph-matching query language. SPARQL queries include graph query patterns, conjunctions, disjunctions, and optional patterns as formalised in (Arenas & Pérez 2011). This ability is essential for semantic integration of heterogeneous data. SPARQL also supports aggregation, filters, limits and federation, etc. Listing 2.2 shows an example SPARQL query Q to search for the highest wind speed per day in a month measured by wind speed sensors.

```
1 PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX sosa:<http://www.w3.org/ns/sosa/>
3
4 SELECT ?sensor ?month ?day (max(?windSpeed) as ?maxWindSpeed)
5
6 WHERE
7 {
8   ?sensor      a              sosa:Sensor;
9               sosa:observes   :winspeedObs01;
10              sosa:madeObservation ?winObs.
11  ?winObs      sosa:hasSimpleResult ?windSpeed
12              sosa:resultTime     ?time.
13 }
14 GROUP BY ?sensor (month(?time) as ?month) (day(?time) as ?day)
```

LISTING 2.2: An example of a SPARQL query that returns the strongest wind speed in each day of month.

A SPARQL query is of the form  $H \leftarrow B$ , where B, the body of the query, is an RDF graph query pattern matched against an RDF graph, and H, the head of the query, defines how to construct the answer of the query (Arenas & Pérez 2011). In Listing 2.2, the body of Q is the text from line 8 to line 12. The graph query pattern of query Q is the matching condition to search for the RDF subgraphs containing the information of wind speed sensors and their observations. The matched subgraph is similar to the RDF graph presented in the previous example (Listing 2.1). The head of Q in line 4 indicates that the max aggregation is applied to the found wind speed values grouped by sensors, days and months.

A graph query operation is performed by matching the variables in its query pattern. In practice, this operation is expressed as a query pattern that is composed of triple patterns and joining matched triples. A triple pattern is also a variation of an RDF triple in which  $s, p, o$  can be variables. A set of triple patterns is called a basic graph pattern (BGP) and can be modelled as a directed graph. A BGP can have different shapes (e.g., star shape,

linear shape, snowflake shape, and complex shape) which influences the performance of graph matching operations (Aluç et al. 2014).

### 2.2.2 Storing and Querying RDF Data

To deal with the data heterogeneity, IoT data can be annotated with RDF by semantic gateways (Desai et al. 2015) or semantic information broker (Kiljander et al. 2014). The process of mapping IoT data into RDF, known as “semantisation”, consists of three steps: collecting sensor data, enriching the raw data, and generating RDF statements to semantically annotate the data (Shi et al. 2018). Therefore, it is essential to manage the RDF data generated from the IoT mediator nodes efficiently.

The goal of RDF data management is to facilitate efficient storage and query processing of RDF data. RDF data management has received considerable attention within the Semantic Web community. As a result, there are many works focusing on RDF storage and SPARQL query processing (Hauswirth et al. 2017). An RDF engine, which contains an RDF storage and a SPARQL processor, can be classified into two types: non-native RDF engines and native RDF engines.

A non-native RDF engine is built on top of a traditional database management system (Hauswirth et al. 2017). For example, 3store (Stephen & Nicholas 2003) and Oracle (Chong et al. 2005) use components of a relational database management system to build their RDF storage and SPARQL query processors. In these systems, RDF statements are stored in a single table of three columns corresponding to three constituents of an RDF statement  $(s, p, o)$ . An index is added to each column to speed up the lookup operations. This approach, known as the triple-table approach, scales poorly due to many self-joins over this single, possibly very large table when executing complex queries (Owens 2011). Therefore, the so-called “property table technique” was introduced to reduce the number of self-joins. Here, RDF statements are stored in many tables and each table includes a subject and several predicates. Therefore, triples that have the same subject can be retrieved without an expensive join operation. Sesame (Broekstra et al. 2002), Jena2 (Wilkinson et al. 2003) are among the RDF engines using this approach. An alternative to the property table approach was suggested in SW-Store (Abadi et al. 2009) which organises RDF datasets in two-column tables. Each table contains the subject and object of the triples which have the same predicate. The tables are stored in the column-oriented database C-store (Stonebraker et al. 2005). Moreover, there have been several works to store RDF data in distributed database systems and in cloud infrastructures. For example, Jena-HBase (Khadilkar et al. 2012) integrates Jena and Apache Hadoop<sup>3</sup>, whereas AMADA (Aranda-Andújar et al. 2012) is an RDF data management platform based on Amazon Web Services.

---

<sup>3</sup><https://hadoop.apache.org/>

A native RDF engine, on the other hand, is designed and optimised from scratch to manage RDF data. Instead of adapting RDF concepts to the concepts that are native to the underlying database systems, a native RDF engine is fully optimised for persisting and querying RDF data. Naturally, the design of native RDF storage is heavily influenced by traditional database design. For example, Virtuoso (Erling & Mikhailov 2009) or 4Store (Harris et al. 2009) store RDF statements in a table-like structure. In these systems, RDF data is presented as “RDF quads” consisting of 4 elements: subject, predicate, object, and graph id (or model in 4Store). Each of these attributes can be indexed in different ways to improve query execution performance. Other RDF systems, e.g., YARS (Yet Another RDF Store) (Harth & Decker 2005), Hexastore (Weiss et al. 2008) and RDF-3X (Neumann & Weikum 2008), employ index-permuted storage systems. In particular, the string representations of RDF resources are replaced with unique integers. Indexes are built on the shorter encoded values and cover all major accesses types to triple query patterns. As the RDF data model is a graph data model, it was also suggested to store and process RDF data by graph data structures and algorithms. The systems following the graph-based idea include TripleT (Fletcher & Beck 2009), DOGMA (Bröcheler et al. 2009), Diplodocus (Wylot & Cudré-Mauroux 2015). However, many graph algorithms are known to be complex in terms of implementation and optimisation.

Most of these solutions for RDF data management focus on scalability in dataset size and query complexity for standard computer hardware and cloud infrastructures equipped with large main memory and multiple disks. Their efficiency and scalability is achieved by choosing appropriate index structures and join techniques for large main memory machines (DeWitt et al. 1984), whereas, IoT edge devices are characterised by small memory and flash-based secondary storage. Our previous results (Le-Tuan 2016, Le-Tuan et al. 2018) have already indicated that, on the resource-constrained computers, these engines suffer from performance issues, pointing to the requirement to build a customised RDF data management for this type of devices. There have been several efforts to move RDF data processing to small devices. For instance, Mobile RDF <sup>4</sup>, a lightweight RDF framework, provides simple APIs for creating, parsing and serialising RDF data. However, RDF graph modifications are not supported (no update functionality). AndroJena <sup>5</sup> is an adoption of Jena that offers all functionalities of the original Jena framework. However, the implementation of AndroJena has ignored the fact that RDF data processing on resource-constrained devices has different requirements which results in significant scalability issues. The second version of RDF On-The-Go (Le-Phuoc et al. 2014), which was our first effort to build a native RDF storage for Android OS, can store up to 5 million triples on a common hardware configuration Android phone. Two recent notable works are the Wiselib tuplestore (Hasemann et al. 2014) and  $\mu$ RDF Store (Charpenay et al. 2017) designed for very constrained memory devices which are intended to store only a few thousands of RDF triples.

---

<sup>4</sup><https://www.hedenus.de/rdf/>

<sup>5</sup><https://github.com/lencinhaus/androjena>

## 2.3 Empirical Study

In the current state of the art, many RDF engines can manage from billion to trillion triple datasets ([Hauswirth et al. 2017](#)). To achieve this scale, these RDF engines must be executed on powerful computers equipped with hundreds of GBs of RAM and many CPUs. Clearly, these approaches suffer from performance drawbacks on resource-constrained IoT edge devices. In addition, these RDF engines are optimised for storing and querying of large, heterogeneous, and rather static RDF datasets, whereas IoT data is more dynamic and less heterogeneous. In this section, we study the performance drawbacks of RDF engines that were directly ported to run on lightweight edge devices. The findings of this study are the input for the design of our RDF engine optimised for lightweight IoT edge devices presented in [Section 2.4](#).

The empirical study is conducted in a simulated scenario of a weather data management system specifically created for the IoT domain. To enable semantic interoperability in this IoT system, the weather data is described in RDF and can be queried with SPARQL. In the IoT domain, RDF is often used to semantically annotate the metadata of IoT platforms, systems, and devices such as the location of an IoT device or the specifications of sensors deployed on an IoT platform; the metadata of observations such as types of observations or units of measurement; and observation timestamps and actual readings.

Traditionally, an IoT system that manages semantic weather data can be deployed in a centralised fashion with a 3-layer architecture. At the lowest layer, the IoT devices are wireless sensors or actuators deployed to collect environment data such as temperature, humidity, etc. The collected data can be transmitted wirelessly to the second layer via a number of wireless protocols. The middle layer consists of the IoT gateways functioning as protocol translators that transmit the collected data to the upper layer. Via long-range wired or wireless networks, the data then can be transferred to the third layer which can be powerful servers or cloud infrastructures. At the third layer, the collected data can be mapped to RDF. This layer can also provide SPARQL endpoints to allow users to query the RDF data.

However, it is argued that to reduce the network traffic, to scale up the system, and to support real-time operation, the data should be preprocessed (i.e., annotated, aggregated and filtered) and queried on the IoT gateways themselves ([Desai et al. 2015](#)). This means that RDF engines for these gateway devices must exist and be efficient on their hardware. Hence, the raw data can be mapped to RDF and can be queried by using SPARQL query from the middle layer. In this study, we set up RDF engines on resource-constrained single board computers that are representative of IoT gateway hardware configurations. We use sensor readings from the Integrated Surface Database (ISD) of the National Climate Data Center (NCDC) as sample data. The data is mapped to RDF and is stored locally on these devices. On each device, for each RDF engine, we test how much RDF can be stored and how fast the data can be inserted and queried. The details of

the hardware configurations of our testing devices, RDF engines, RDF schema, testing SPARQL queries, and the experiments are presented in the following sections.

### 2.3.1 Hardware Devices

Recent technological advances of embedded processors have increased the processing capabilities of IoT edge devices. Based on their computational capabilities, IoT devices can be categorised into low-end devices and high-end devices. The low-end IoT devices which are very constrained in terms of energy, CPU (less than 100MHz), and memory capacity (less than 100 kB) resources. Popular examples of devices in this category include Arduino <sup>6</sup>, Zolertia <sup>7</sup>, OpenMote node <sup>8</sup>, etc. The second category consists of high-end IoT devices which include single-board computers such as Intel Galileo <sup>9</sup>, Raspberry Pi <sup>10</sup>, Beagle Bone board <sup>11</sup> or smartphones. They have enough resources and adequate capabilities to run software based on traditional operating systems such Linux or BSD (Berkeley Software Distribution).

We conduct our empirical study on five types of high-end IoT devices: Intel Galileo Gen II (GII), Raspberry Pi Zero W (RPi0), Raspberry Pi 2 version B (RPi2), Raspberry Pi 3 (RPi3), and Beagle Bone Black (BBB). They were chosen because of their popularity and thus proof of concept in the IoT domain. Furthermore, they are good representatives of resource constraints of IoT gateways. The configurations of each device used in the experiments are summarised in [Table 2.1](#).

The Intel Galileo was developed by Intel and was the first IoT device that could run a complete Linux system. The Intel Galileo was designed as to be Arduino-compatible, which enables Arduino sensors and actuators to be used out of-the-box. Therefore, it can be considered as a composition of two sets of hardware components, i.e., the Arduino hardware and the “mini PC” hardware. The Intel Galileo is equipped with RAM, Flash memory, a mini SD card reader, an Ethernet adaptor and a Intel Quark x86 CPU. Galileo came out in two versions: Intel Galileo Gen 1 and Intel Galileo Gen 2. The Gen 2 boards are more powerful than Gen 1 boards as they have some improved hardware. In the experiments we use only Galileo Gen 2 hardware.

BeagleBoard was originally developed and introduced by Texas Instruments in 2008. By using the OMAP3530 system-on-a-chip technology, the BeagleBoard board is a good platform for various demonstration scenarios in the IoT world and it was regarded as a giant step to bring the microcontrollers to a fully fledged microcomputer. BeagleBone Black is a small credit card-sized board that was launched in 2013 at a price of 55 EUR.

---

<sup>6</sup><https://www.arduino.cc>

<sup>7</sup><https://zolertia.io>

<sup>8</sup><https://openmote.com>

<sup>9</sup><https://www.arduino.cc/en/ArduinoCertified/IntelGalileo>

<sup>10</sup><https://www.raspberrypi.org>

<sup>11</sup><https://www.beagleboard.org/bone/>

Despite its small size and low cost, the BeagleBone Black is equipped with 512 MB RAM, a 1 GHz clock ARM Context-A8 processor, and 2 GB eMMC flash memory.

The Raspberry Pi is another well-known, low-cost type of single board computer that was developed by the University of Cambridge. The Raspberry Pi Zero was introduced in 2015 and features an ARM Cortex A8 single core CPU with 1.0 GHz and 512 MB RAM. In 2017, the Raspberry Pi Zero W was launched, a newer version of the Pi Zero with Wi-Fi and Bluetooth capabilities. The Raspberry Pi 2, which is more powerful than Pi Zero, has an ARM Cortex A7 CPU with a 0.9 GHz CPU cycles/core, quad-cores, and 1 GB of RAM. The Raspberry Pi 3 is equipped with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor, with 512 KB shared L2 cache and 1 GB of RAM. The Raspberry Pi runs Raspbian OS, a free operating system based on Debian Linux and optimised for the Raspberry Pi hardware. Despite the availability of 64-bit CPUs on Raspberry Pi 3, at the time this work was done, Raspbian OS with 64-bit has not been released yet.

TABLE 2.1: Hardware configurations of the devices used in the experiments.

Device	GII	RPi0	BBB	RPi2	RPi3
Cost	65 EUR	15 EUR	55 EUR	35 EUR	45 EUR
CPU	model	Quark	ARM A8	ARM A7	ARM A53
	freq.	0.4 GHz	1.0 GHz	900 MHz	1.2 GHz
	$n_{cores}$	1	1	4	4
RAM	256 MB	512 MB	512 MB	1 GB	1 GB
Storage	Transcend MicroSD 16GB class 10 (40 MB/s)				
OS	Yocto 1.4 Poky Linux Distribution	Rasp. Lite	Debian 7.0	Rasp. Lite	Rasp. Lite

### 2.3.2 RDF Engines

We selected three RDF engines that can be set up on the above devices: Apache Jena, RDF4J, and Virtuoso. The technical specifications of each engine used in this study are reported in [Table 2.2](#).

Apache Jena <sup>12</sup> is a well-known open-source framework for RDF data processing implemented in Java. To support persistent RDF storage, Apache Jena provides a native RDF storage called Jena TDB (Tuple Database). Jena TDB stores RDF terms (nodes) in a node table and employs multiple indexes for RDF triples. The data in the node table and the indexes are organised in fixed length key and fixed length value B<sup>+</sup>Trees. Jena TDB is able to operate on both 32-bit and 64-bit systems. However, it performs better on a 64-bit machine because its caching mechanism requires more memory. On a 32-bit JVM,

<sup>12</sup> <https://jena.apache.org/>

the size of the dataset might be limited because Java addressing cannot grow beyond 1.5 GB. To deal with this limitation, TDB employs an in-heap LRU cache of B<sup>+</sup>Tree blocks. Thus, it is recommended to configure JVM with at least 1GB for Jena TDB to achieve the sufficient performance.

RDF4J <sup>13</sup> (formerly Sesame (Broekstra et al. 2002)) is another RDF data processing framework implemented in Java and is available as open-source software. Native Store is the persistent RDF storage of RDF4J. In addition, this component communicates with other stores via SAIL API (Storage and Inference Layer). Native Store is designed to support medium datasets (e.g., 100 million triples) on common hardware. It uses direct disk I/O and employs on-disk indexes to speed up queries. Again, B-Trees are used for indexing RDF statements and each RDF statement is stored in multiple indexes. By default, the Native Store uses two indexes: subject-predicate-object-context(spoc) and predicate-object-subject-context. Indexes can be added or dropped on demand to speed up querying or saving disk space.

Virtuoso <sup>14</sup> is developed by OpenLink Software Inc, and is available both as an open-source and a commercial version. Different from the other engines, Virtuoso uses a relational database back-end storage to store RDF and is implemented in C++. Virtuoso is well known as a traditional relational database supporting RDF data and as a SPARQL-to-SQL solution to manage RDF data. The older version of Virtuoso stores RDF data in a row-wise format storage. Meanwhile, column-wise format storage has been adopted since version 7. In a row-based storage, RDF datasets are stored as collections of RDF quads that consist of graph ids, subjects, predicates, and objects in a single table. From version 7, Virtuoso only operates on 64-bits OS. Thus, we have compiled and set up the Virtuoso 6 open-source version for the evaluation.

---

<sup>13</sup><https://rdf4j.org/>

<sup>14</sup><https://virtuoso.openlinksw.com/>

TABLE 2.2: Characteristics of the RDF engines used in the experiments.

	Technical Characteristics				
	Developed in Language	Backend DB	File Access	Data Structure	Version
Jena TDB	Java	native store	File Caching	B <sup>+</sup> tree LRU Cache	3.14.0
RDF4J Native Store	Java	native store	n/a	BTree	3.1.0
Virtuoso Open-Source	C++	row store	n/a	B <sup>+</sup> Tree	6.1.8

### 2.3.3 Weather Dataset and RDF Schema

As mentioned above, we use the ISD (Integrated Surface Dataset) dataset <sup>15</sup> as the sample dataset for our experiments. The ISD dataset is one of the most prominent weather datasets that contains weather observations collected from 20 thousand weather stations from all over the world since 1901. The observations include measurements such as temperature, wind speed, wind angle, etc. Additionally, the observations also contain the timestamps when these measurements were made.

To describe the metadata of weather stations such as location, deployed sensors, and observations in RDF, we use the Semantic Sensor Network (SSN) ontology, the Quantity Kinds and Units (QUDT) ontology, Geo Name (GeoNames) and Basic Geo (WGS84). Figure 2.1 illustrates a sample of the RDF schema used to describe the metadata of the weather stations in the ISD dataset.

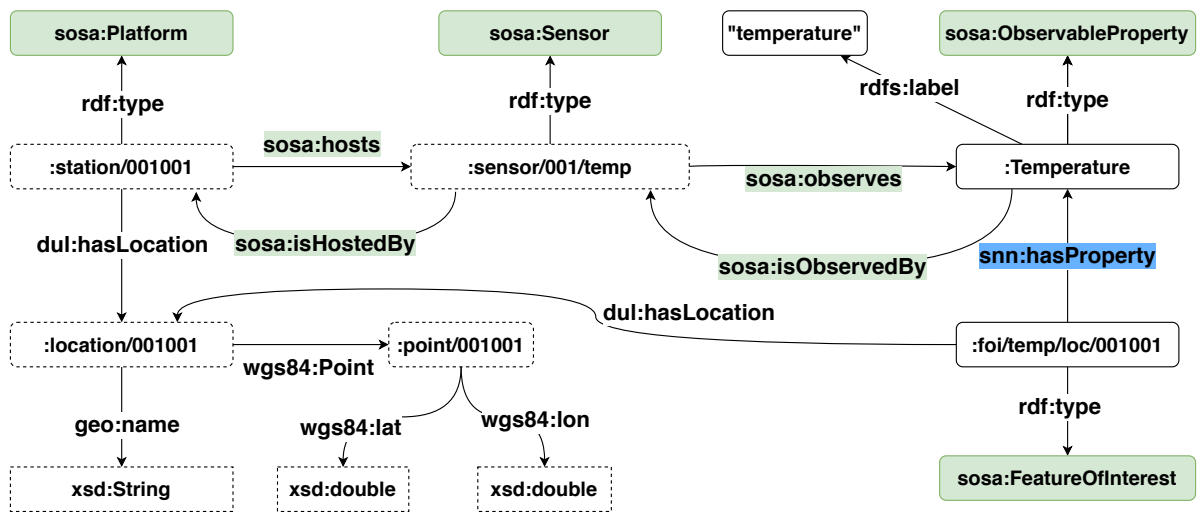


FIGURE 2.1: The RDF schema describing the weather stations in the ISD dataset.

In the ISD dataset, each weather station is assigned a unique station ID. Thus, to create a unique IRI to refer to a weather station, we concatenate a unique prefix and the station ID. For example, the IRI *station:001001* is used to refer to the weather station 001001. Following the specification of the SSN ontology, a weather station is described as a platform that hosts multiple sensors or devices. For instance, the IRI *station:001001* is described to have type *sosa:Platform* and hosts a sensor whose resource is *sensor:001/temp*. The location of the station is described by using GeoNames and WGS84. The class *sosa:ObservableProperty* is used to define the phenomena and properties that a sensor can observe. For instance, the resource *:Temperature* refers to an observable property which is the temperature. The temperature sensor *sensor:001/temp* is described as a *sosa:sensor* that observes the *:Temperature*.

The RDF schema of a sensor reading is shown in Figure 2.2. A resource referring to a sensor reading is described as an observation by using the *sosa:Observation* class. The

<sup>15</sup><https://www.ncdc.noaa.gov/isd>

type of observation is expressed by using the *sosa:observedProperty* property. Defining an observation and its type is similar to how a sensor and its observable phenomena. For example, the temperature observation *Observation/001* is defined as a *sosa:Observation* whose observed property is *Temperature*. The timestamp of an observation is defined by using the Date and Time data type of the XML Schema Definition Language(XSD) and assigned to the observation with the predicate *sosa:resultTime*. The actual reading of an observation is described with the predicate *sosa:hasSimpleResult* or more explicitly with the predicate *sosa:hasResult*. In this example, the unit of the temperature reading is presented in Celsius degree. Furthermore, the *sosa:FeatureOfInterest* vocabulary is used to enhance the expressiveness of an observation. For instance, the observation in the example is known as a temperature observation at location *location/001*. With this RDF schema, approximately 80–90 RDF triples are required to map an observation from the ISD dataset to RDF.

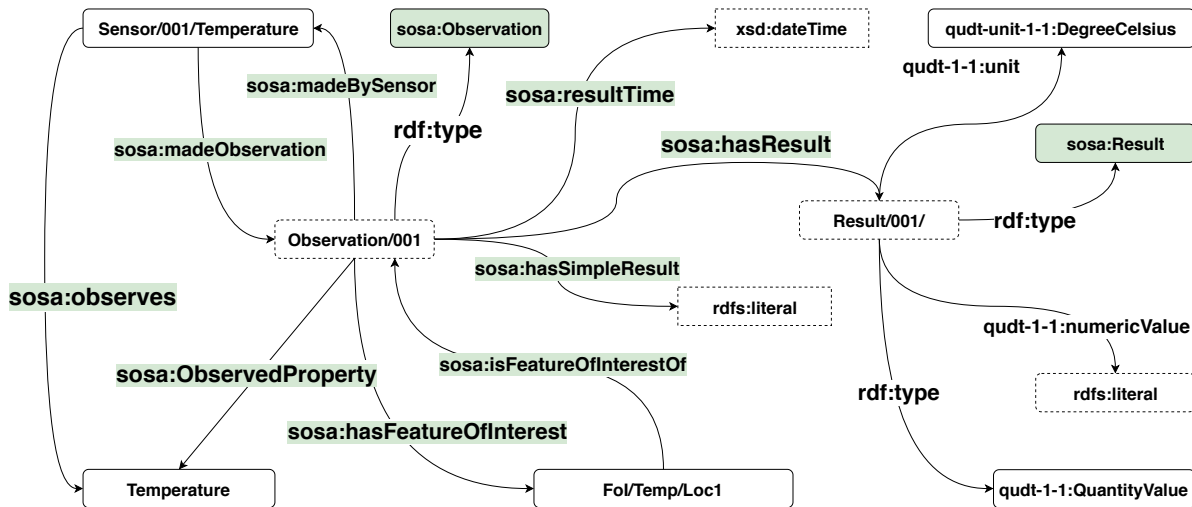


FIGURE 2.2: The RDF schema describing the weather observations in the ISD dataset.

The queries used in the experiments were created following the design of the WATDIV benchmark (Aluç et al. 2014) to test the performance of each SPARQL query processor on different shapes of BGP. Eleven query templates were created and can be found in our Github repository<sup>16</sup>. The query templates are categorised into three groups: linear (L), star (S), and snowflake (F). The BGPs of the query templates in the linear group contain multiple low degree join triple patterns, whereas the BGPs in the star-shaped queries are composed of single high degree join triple patterns. The snowflake-shaped queries have a mix of low degree joins and high degree joins.

### 2.3.4 Experiment Design

We evaluated the selected RDF engines with three experiments. First, we test the update throughput of each engine on each device. As presented in our scenario, on high-end

<sup>16</sup><https://github.com/anhlt18vn/sensor2020>

devices, these RDF engines may serve for embedded semantic data management that supports semantic gateway services (Desai et al. 2015) or semantic information brokers (Kiljander et al. 2014) architectural styles. Hence, they are required to deal with dynamic data flows from the sensors. Then, we test the query response time. Finally, we measure the memory consumption of the RDF engines when these engines perform storing and querying operations.

Figure 2.3 depicts the setup of our experiments. The ISD data is read and mapped into RDF with the RDF schema described in Section 2.3.3 by the *ISD-2-RDF Wrapper*. The processes of inserting and querying the generated RDF data to/from the *RDF Engine* are managed by the *RDF Data Insert and Query Monitor*. This component is also responsible for recording the performance of these processes. The complete source code of the implementation of our experiments can be found in our Github repository<sup>16</sup>.

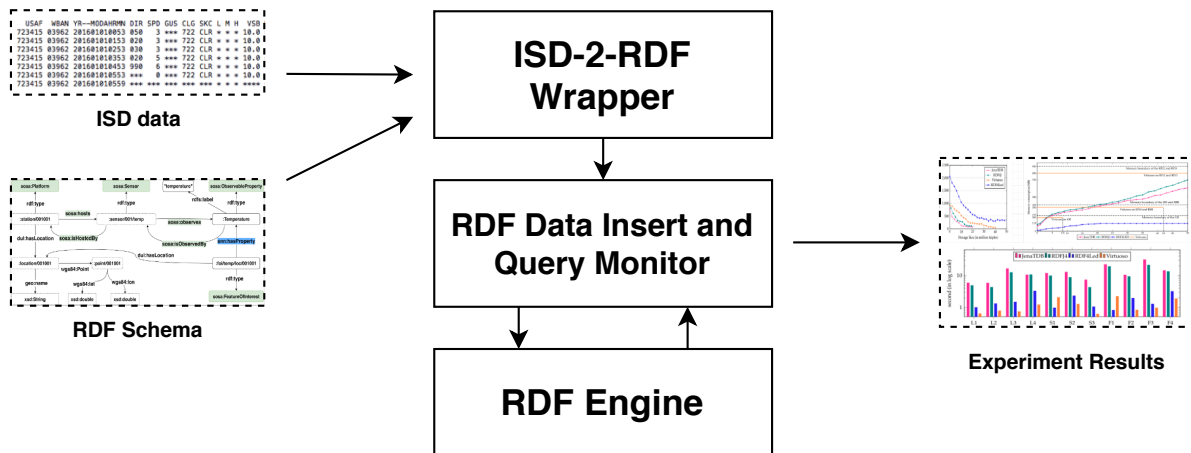


FIGURE 2.3: Setup of the RDF data insertion and querying experiments.

Exp1 - Update throughput: The first experiment is to test how much new data the system can incrementally update with a certain underlying RDF store corresponding to each hardware configuration. We simulate the process of data growth by gradually adding more data to the system. We measure the rate of inserting data (triples/s) and query response time until the system crashes or until the speed is below 80 triples/s (whichever happened first). If the system cannot update 80 triples/s, that means it is not able to update one observation/s. We extract data from 25 weather stations from the ISD dataset in the last six months of the year 2019. The number of observations is approximately 600 thousands, and the size of the generated RDF dataset is about 50 million RDF triples.

Exp2 - Query evaluation: In the second experiment, we test the query response times of each engine. On each device, we choose the dataset with a scale which all engines can store. For each dataset, for each query template, we generate 100 queries. We record the maximum, minimum, and average time that these engines need to answer each type of query. The generated queries for each dataset are 1100 (11 query templates) in total.

Exp3 - Memory consumption: In the third experiment, we measure the memory consumption of three system configurations, when they perform the insertion and query. The

experiment runs the queries repeatedly and records the maximum memory heap that the operating system allocates. Note that the memory consumption is device-independent. To evaluate the impact of the data size on memory consumption, the test is conducted on the Raspberry Pi 3 with ten different sized datasets and 10 sets of queries according to each dataset. The scale ranges from 5 million triples to 50 million triples.

### 2.3.5 Experiment Report and Findings

[Figure 2.4](#) illustrates the results of **Exp1**, in which we measured the update throughput of Virtuoso, Jena TDB and RDF4J on five types of lightweight computing devices. On GII, RPi0, and BBB, none of the RDF engines could finish inserting the dataset of 50 million RDF triples. They crashed in the middle of the test due to an “out of memory” error. For instance, on GII (see [Figure 2.4\(a\)](#)), Virtuoso was able to insert 9 million RDF triples, Jena TDB and RDF4J stopped after inserting 5 million RDF triples. Due to the similar hardware settings, the scalability behaviours of these RDF engines on RPi0 (see [Figure 2.4\(b\)](#)) and on BBB (see [Figure 2.4\(c\)](#)) were similar. On both devices, Virtuoso could store 40 million RDF triples, whereas Jena TDB and RDF4J were only able to store 20 million RDF triples. On the other hand, on the RPi2 and RPi3, with more powerful computational capabilities, all three RDF engines could finish the test and store up to 50 million RDF triples (see [Figure 2.4](#)).

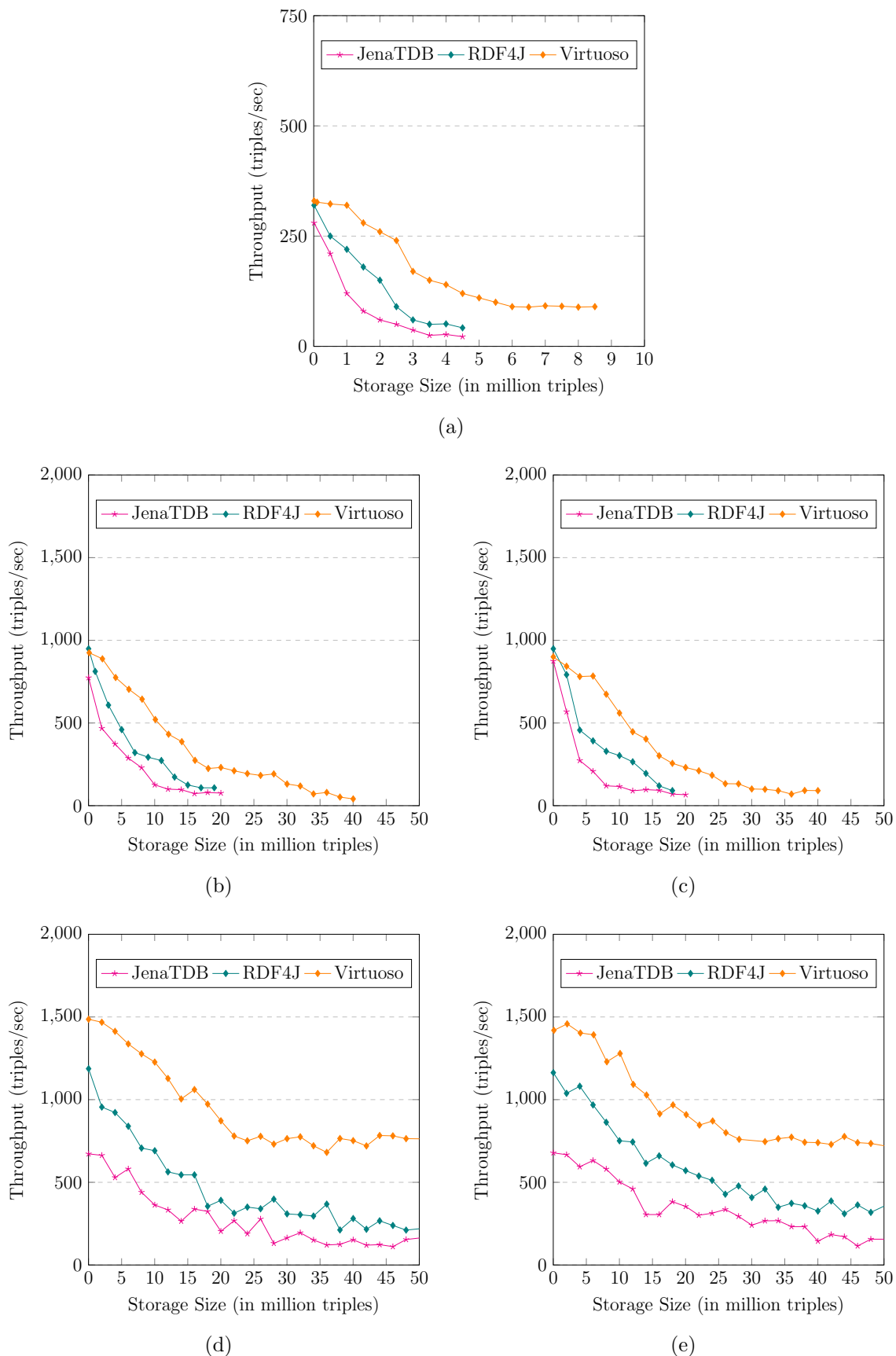


FIGURE 2.4: Throughput test results of Jena TDB, RDF4J, Virtuoso on Gallileo Gen II, BeagleBone Black, Raspberry Pi Zero Raspberry Pi 2, and Raspberry Pi 3. (a) Insert throughput results on Gallileo Gen II; (b) Insert throughput results on Raspberry Pi Zero; (c) Insert throughput results on BeagleBone Black; (d) Insert throughput results on Raspberry Pi 2; (e) Insert throughput results on Raspberry Pi 3.

In general, the update throughput of the three engines decreased when the size of their storage increased. On GII, after inserting 3 million triples, the update throughput of Virtuoso was 250 triples/s (approx. 3 observations per second), whereas Jena TDB and RDF4J only could insert 50 triples/s (less than 1 observation per second). Before crashing, Virtuoso's update speed was less than 100 RDF triples per second, whereas the update speeds of Jena TDB and RDF4J were only 20 and 50 triples/second respectively. For RPi0 and BBB, the inserting speed of Virtuoso was up to 600–900 triples/s in the first 10 million triples. However, Virtuoso's speed dropped dramatically when the storage size reached 15 million triples. Its speed remained 350 triples/s, which is only half of its peak speed. The update behaviour of Jena TDB and RDF4J was similar to that of Virtuoso. However, with the same storage size, the speed of Jena TDB and RDF4J was less than half of the speed of Virtuoso. On RPi2 (see [Figure 2.4\(d\)](#)) and RPi3 (see [Figure 2.4\(e\)](#)), the insertion throughput of the RDF engines was much higher and dropped slower than that on RPi0 and BBB. At the beginning of the test, Virtuoso inserted data with a speed of up to 1300–1400 triples/s. The insertion speed of Virtuoso decreased to 700–900 triples/s later when the storage size was up to 40 million RDF triples. Again, the insertion speed of Jena TDB and RDF4J with RPi2 and RPi3 was 2–3 times slower than that of Virtuoso.

[Figure 2.5](#) reports the results of the **Exp2** in which we compared the query response time of the RDF engines. For each type of device, we used the datasets that all engines could handle. For instance, we used datasets of 5 millions, 20 millions, and 50 millions of RDF triples to conduct the second test on GII (see [Figure 2.5\(a\)](#)), BBB (see [Figure 2.5\(b\)](#)) and RPi3 (see [Figure 2.5\(c\)](#)) respectively. In general, all RDF engines could answer the tested SPARQL queries. Among the three RDF engines, Virtuoso was always the fastest to return the answer for every query. In most cases, Jena TDB and RDF4J were able to answer these queries in less than 10 seconds. However, to answer the more complicated queries, e.g., the snow flake shape query F3 took roughly a minute.

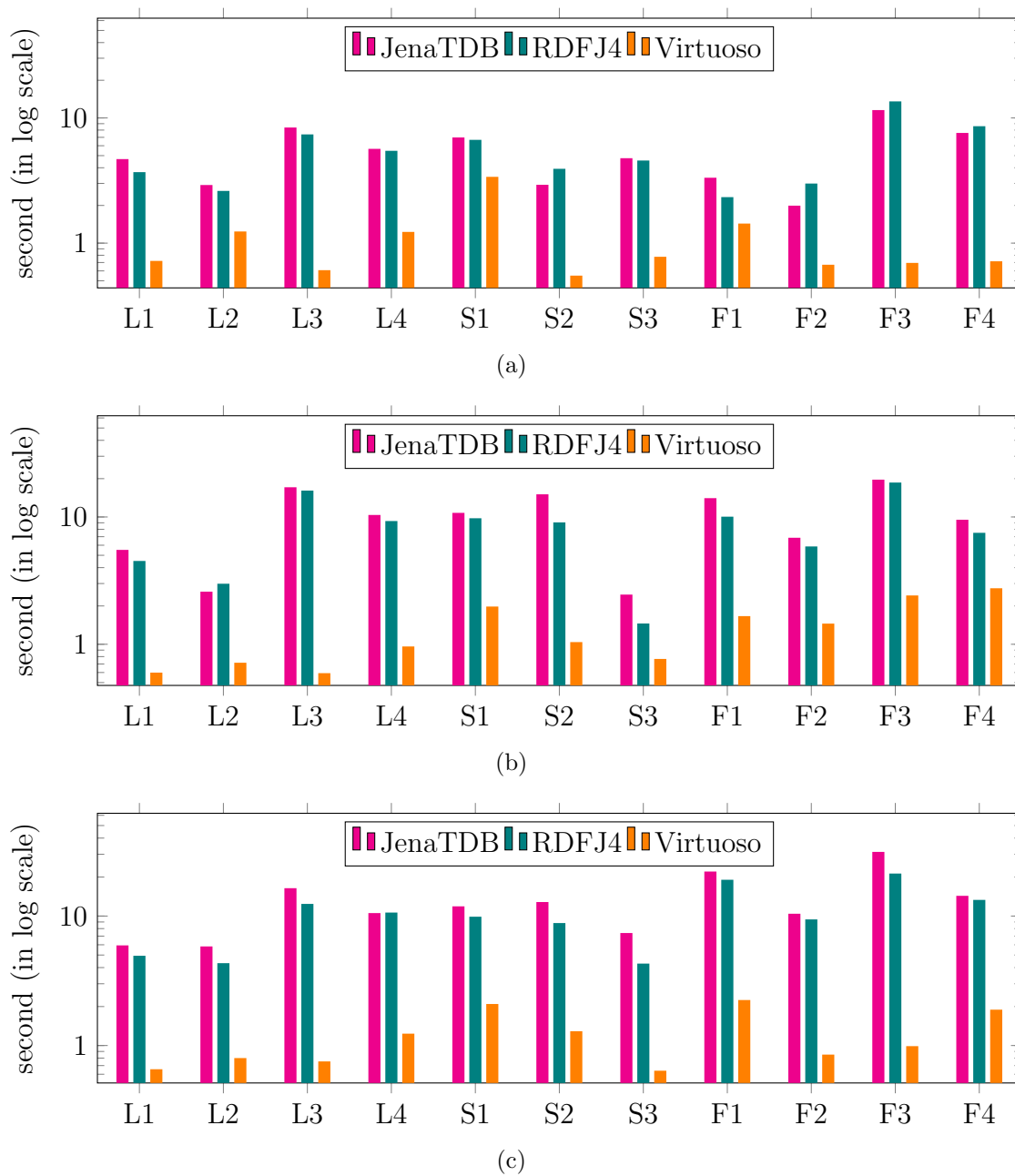
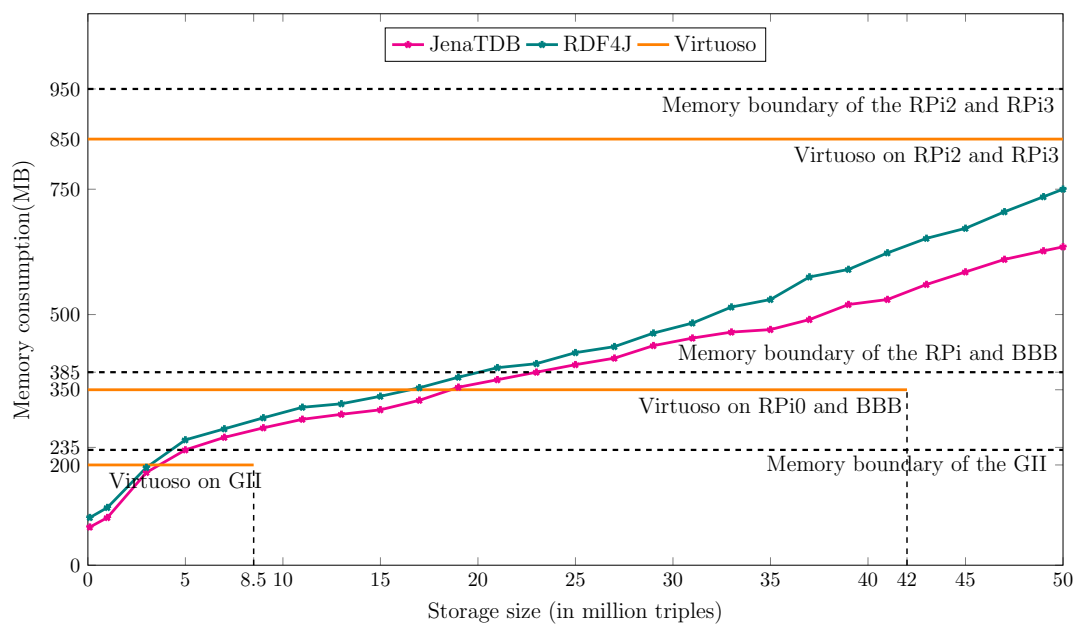
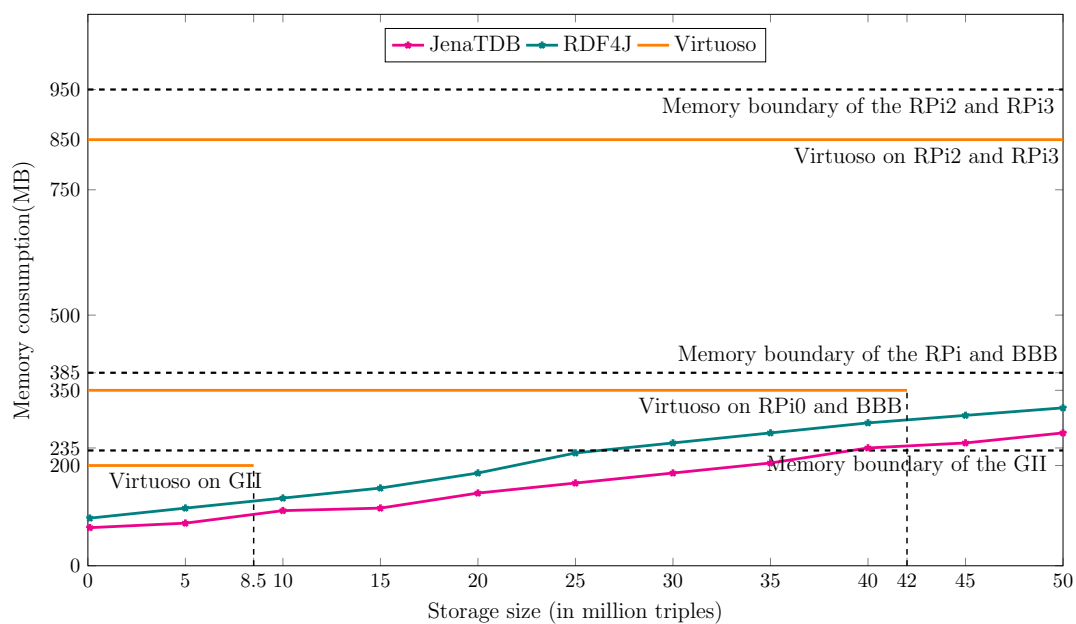


FIGURE 2.5: Querying test results of Jena TDB, RDF4J, Virtuoso and RDF4Led. (a) Query response time against 5 million triple dataset on Gallileo Gen II; (b) Query response time against 20 million triple dataset on BeagleBone Black; (c) Query response time against 50 million triple dataset on Raspberry Pi3.

The difference in scalability and performance of the three engines can be explained by their memory usage, which is reported in [Figure 2.6](#). Note that a part of the memory is occupied by the operating system. Therefore, the maximum available memory for the application is always lower than the size of RAM. For instance, there is only 230 MB available memory on the GII, nearly 380 MB on P0 and BBB and 950 MB in RPi2 and RPi3. The memory consumption of Jena TDB and RDF4J gradually increased according to the size of the storage. In the throughput test (see [Figure 2.6\(a\)](#)), the memory consumption of Jena TDB and RDF4J rose up to 230 MB, 380 MB and 650 MB after inserting 5 million, 20 million, and 50 million RDF triples, respectively. The memory usage histogram of Jena TDB and RDF4J explains why they ran out of memory when operating on GII, RPi0, and BBB. In contrast, the memory buffer of Virtuoso was statically set depending on the maximum RAM available on each device. For instance, it was set to 200 MB on GII, 350 MB on BBB and RPi0 and 850 MB on RPi2 and RPi3. On the same device, Virtuoso had better scalability than Jena TDB and RDF4J because it handled the buffer memory better. For example, writing data from memory to the secondary storage to claim back memory space could help Virtuoso enlarge its storage. Virtuoso was able to store up to 8.5 million RDF triples on GII, and near 40–42 million RDF triples on RPi0 and BBB. Writing data into the secondary memory to create more room for caching new data is a widely used technique in conventional database management systems ([Ullman et al. 2001](#)). By using this technique, it can be explained why the insertion speed of Virtuoso dropped dramatically and was heavily penalised on the devices with less memory. However, compared to Jena TDB and RDF4J, Virtuoso used bigger buffer memory to cache more data in the main memory, which explains why in our experiments Virtuoso could update data and answer the queries faster.



(a)



(b)

FIGURE 2.6: Memory consumption of RDF4J, Jena TDB, Virtuoso. (a) Memory consumption of RDF engines in update throughput test; (b) Memory consumption of RDF engines in query evaluation.

## 2.4 RISC-style Approach for Lightweight Edge Devices

### 2.4.1 Rationale of our System Design

From our empirical study, we clearly see that lightweight edge devices are different from standard computing hardware in respect to two characteristics that determine the performance of an RDF store: (i) They have a significantly smaller amount of main memory and (ii) they are equipped with flash-based storage as secondary memory and storage. Besides that, data processing on the network edges operates in a dynamical environment with frequent data updates and changes in devices and sensors.

Typically, RDF engines are optimised for machines with massive amounts of RAM and multiple high-performance disk arrays. This abundance of resources enables them to store billion-triple datasets and answer complex SPARQL queries. However, our empirical study has shown that these RDF engines suffer from significant performance problems when they run on edge devices. To manage large static RDF datasets, these RDF engines use sophisticated indexing mechanisms that consume huge amounts of main memory and incur high update costs. Using too much memory on memory-constraint devices may cause system paging behaviours or out-of-memory errors that heavily penalise the performance or harm the robustness. Such inefficiency is due to the lack of main memory and a less efficient disk-based data indexing structure and caching mechanism on flash-based storage.

In comparison to hard disks, flash-based storage devices have faster random accesses but fail to provide fast random writes ([Ajwani et al. 2008](#)). Flash memory stores information in arrays of semiconductor memory cells which are organised into pages and pages are grouped into blocks. A page is the smallest unit that can be read or written to flash memory. With flash memory, updating a single page in a block is not possible. Instead, first the the whole block must be erased and then the updated data can be written to this block. Erase is an operation particular to flash-based memory. Thus, the write-in-place operations, that update a single piece of data in a block, consist of two operations on the entire block: erase and write. Common indexing techniques which are designed for magnetic disks do not manage well this erase-before-write limitation. As a result, they suffer from slow write performance when managing data on flash memory ([Bouganim et al. 2009](#)). For instance, the commonly used B<sup>+</sup>Tree indexing structure in RDF triple stores does not work well on flash-based storage ([Ho & Park 2016](#)). However, the performance of random write can be improved by aligning writes to blocks ([Bouganim et al. 2009](#)) and applying appropriate buffer management techniques ([Jin et al. 2012](#)).

The RISC-style design philosophy described in ([Neumann & Weikum 2008](#)) implements the features necessary for an RDF engine around data access and join operations. On top

of that, the processing load and resource consumption are mainly caused by these operations. Thus, we will focus our design efforts on efficient components for these operations and use simple implementations for the rest with the purpose of reducing software size.

## 2.4.2 Architectural View

In general terms, the architecture of an RDF engine can be viewed as shown in [Figure 2.7](#). At the bottom layer, an RDF engine has a *Physical RDF Storage* functioning as the secondary memory to store persistent data. The Physical RDF Storage is often coupled with a *Buffer Manager* to manage in-memory data. The Buffer Manager caches the data in use to reduce disk accesses when writing to the Physical Storage or being read by the *Query Executor*. Typically, an RDF engine will use a *Dictionary* to translate the string-based RDF resources identifiers into encoded identifiers in the form of integers or longs. The Dictionary is often coupled with an *Input handler*, a *Query Parser* to encode the RDF resources in RDF documents or SPARQL queries, and with an *Output handler* to return the original form of RDF resources. This technique reduces the storage space required for RDF triples and makes comparisons (for joins) more efficient.

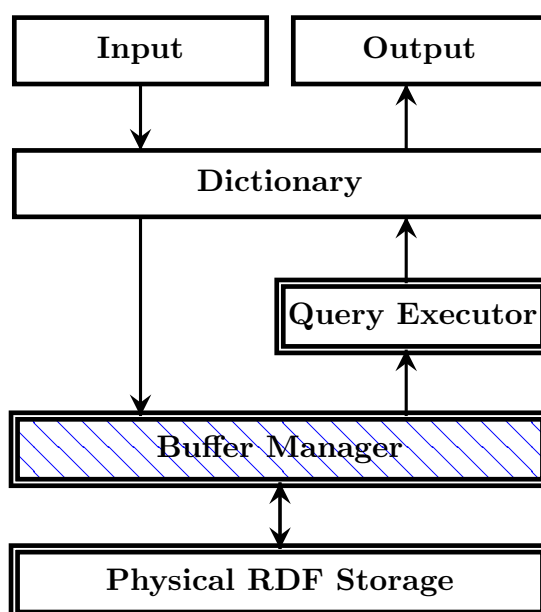


FIGURE 2.7: Architecture overview of a RISC-style RDF engine.

For our RDF engine, RDF4Led, we reuse the same architecture of traditional RDF engines. We reuse the Dictionary technique to transform RDF resources into encoded integers. The string representations of the RDF resources are kept separately on the flash memory. The key components that differentiate our approach from traditional RDF engines are the Physical RDF Storage, the Buffer Manager, and the Query Executor. They are specifically optimised for lightweight edge devices.

In an RDF engine, the algorithms and techniques that comprise the Physical Storage, the Buffer Manager, and the Query Executor, have to be optimised to the nature of the data (in this case RDF) and the particular hardware of the machine it runs on (Ullman et al. 2001, Owens 2011). To reduce the storage space, we used a very compact format to store a list of RDF triples known as “RDF molecules.” To adapt to the specific flash I/O behaviour, the molecules are organised into block units whose size is equal to the flash-erase block size. On top of that, we used an in-memory caching mechanism to cache the atomic data and to cluster the write operations to improve the write performance. To reduce the memory required to maintain the indexes of data in the flash storage, we used an alternative index structure that is based on the Block Range Index (BRIN) approach<sup>17</sup>. The basic idea of BRIN is to summarise the information of a data block on persistent storage (e.g., its location) into a small tuple. The result is that we can minimise the amount of memory required to maintain the indexes.

The results of our empirical study indicate that managing the memory usage of an RDF engine is the key factor to achieve robustness and scalability. The Buffer Manager is used to buffer updated data or to cache data read from Physical Storage. Its primary role is to keep the engine from crashing by unexpected out-of-memory exceptions. When needed, it flushes data to the Physical RDF Storage to reclaim free memory. Write operations are prioritised by a buffer replacement policy designed to reduce the number of overwrites on the same data block as well as the number of read operations from the Physical Storage.

To answer a SPARQL query, it is required to perform graph matching operations. These operations are the joins of the RDF triples that match triple query patterns. Among the operations to answer a SPARQL query, the graph matching operation is the most resource-intensive one. To reduce the computational cost, our approach is to avoid caching the intermediate results of joins. The Query Executor uses a nested execution model (Graefe 2003) to join, and it processes the join in one-tuple-at-a-time fashion (Avnur & Hellerstein 2000). In each run, the Query Executor adaptively chooses the next triple pattern to probe and scan. The Buffer Manager is also tightly coupled with the Query Executor to provide cached data in the buffer for the efficient use of memory.

## 2.5 Storage and Indexing

### 2.5.1 Storage Layout

Our RDF storage and indexing model combines the permuted index and molecule-based storage model. We use 3 permuted indexes: SPO (subject-predicate-object), POS, and OSP, which is sufficient to cover all possible query patterns. For example, the SPO layout can be used for triple query patterns with a bound subject (s ? ?) and bound subject-predicate (s p ?). Although using all six possible permutation combinations may answer

<sup>17</sup><https://www.postgresql.org/docs/9.5/brin-intro.html>

complex queries more effectively, using only three indexes consumes less storage space and decreases the cost of updates, i.e., we must update only three data structures instead of six, which is crucial for flash storage.

Our design consists of a *Physical Layer* and a *Buffer Layer*. The Physical Layer stores data directly on flash storage (Physical RDF Storage) and the Buffer Layer operates in the main memory (Buffer Manager) and has the following roles: (i) grouping and caching atomic data updates before writing a block; (ii) indexing the data stored on the Physical Layer ; and (iii) caching recently used data for read performance. This allows us to group multiple updates within a block into one erase-and-write operation and to improve read performance through the cache.

**Physical Layer:** To achieve high compression of triples on the flash storage, we leverage the molecule-based storage model. RDF molecule is a hybrid data structure. It stores a compact list of properties and objects related to a subject which is the root of the molecule. Molecule clusters are used in two ways: to logically group sets of related resources and to physically collocate information related to a given subject. Physically, we represent a molecule as a list of co-located integers corresponding to S, P, and O as shown in Figure 2.8. By this, we avoid storing repetitive values multiple times. Moreover, we enable further data compression, e.g., by storing deltas of sorted integers instead of full values.

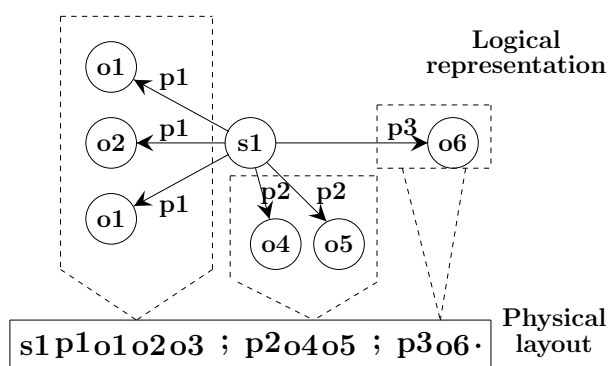


FIGURE 2.8: Example of an SPO molecule.

In the Physical Layer, we store sorted molecules into contiguous pages (read units) which are grouped into blocks (erase units). Moreover, all entities in molecules are also sorted to improve the search performance. In the example shown in Figure 2.9, each block in the Physical Layer contains four pages and each page stores molecules.

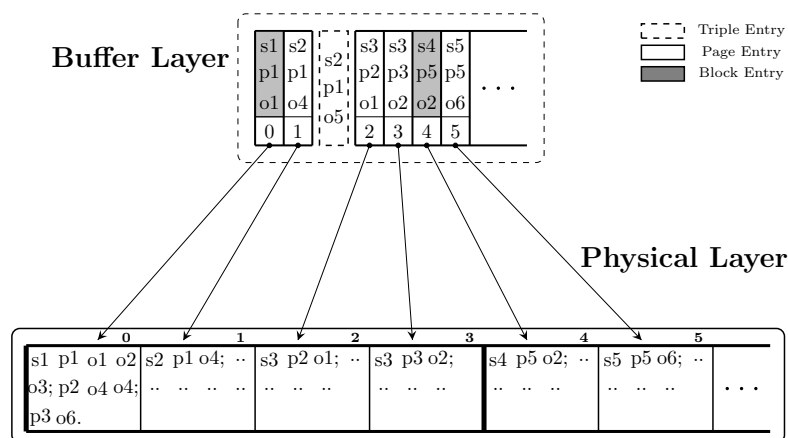


FIGURE 2.9: Two-layers storage model consists of a Physical Layer and a Buffer Layer.

**Buffer Layer:** Similarly to the idea of BRIN, the Buffer Layer summarises the information of the data in the Physical Layer. In the Buffer Layer, to keep the reference to a data page, we cache the first triple of the first molecule in the page. The reference to a data block is the first page of the data block. Figure 2.9 depicts an example of the indexing structure that we use for storing an SPO index. The Buffer Layer maps the references of the pages and blocks to their physical addresses. Thus, it acts as an index of the data that locates in the Physical Layer. We distinguish three types of entries in the Buffer Layer: triple entries, page entries, and block entries. A *page entry* is an entry that refers to the beginning of a page in the Physical Layer, it contains the first triple in the first molecule of a page. A *block entry* is a page entry with an extra field indicating that this page is the first page of a block. A *triple entry* contains an atomic triple and a value indicating whether this triple has been modified. In Figure 2.9, the grey columns represent block entries and the white columns represent page entries. For fast lookup operations on the Buffer Layer, all triples are sorted lexicographically. Moreover, we maintain the logical order of the triples as in the Physical Layer. This allows us to group and commit sequential pages containing modified triples and belonging to the same block in one write operation.

## 2.5.2 Index Lookup

To efficiently retrieve RDF triples that match a triple pattern, we execute a lookup on the index layout in which triples are stored and sorted on the bound elements of the triple pattern. For example, a search for the matched triple of a triple pattern  $(s, p, ?)$  is executed on the SPO index layout, or the triples that match the pattern  $(?, p, o)$  are answered with POS index layout. Thus, we can answer a triple pattern query by using a single range scan on the corresponding index layout. For instance, in the sorted SPO index layout, the matched triples of the triple pattern  $(s, p, ?)$  are of the form  $(s, p, o_i)$  and are located between  $(s, p, o_{min})$  and  $(s, p, o_{max})$ , where  $o_{min}$  and  $o_{max}$  are the smallest and the highest identifiers within the layout. As triples are sorted, the matched



buffer to storage, we move either data belonging to block 0, i.e., triples (03, 06, 01), (03, 06, 04), or data belonging to block 1, i.e., triples (26, 24, 04), (30, 28, 21), so that only one block is erased. In case we choose triples (18, 10, 12) and (30, 28, 21), the system has to erase two blocks before writing data. Moving data from the Buffer Layer to the Physical Layer is an essential optimisation problem of maximising the number of triples we can write within a minimal number of blocks to minimise the number of erase operations. Obviously, block 0 is written before block 1 as its density is greater than that of block 1. The higher the density a block has, the less chance the next arriving triple will have to be inserted into that block.

## 2.6 Buffer Manager

Similar to a conventional database system, the Buffer Manager is responsible for handling all requests for data blocks (see [Section 1.3.2.5](#)). If the requested data block already exists in the main memory, it passes the block’s reference to the requester. If not, it fetches the data block from the flash drive (Physical Layer) and loads it into the main memory (Buffer Layer). It also decides which data block will be kept in the main memory. To perform efficiently, the Buffer Manager requires a suitable replacement strategy to create space for a new data block and write back to Physical Storage data blocks which are no longer needed.

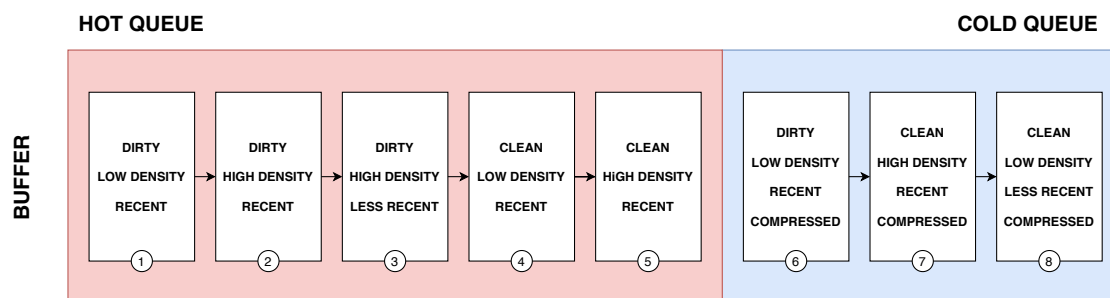


FIGURE 2.11: Example of how data block is organised in the buffer queues.

Our Buffer Manager is designed similarly to AD-LRU ([Jin et al. 2012](#)), a flash-friendly data buffering technique. We organise the data blocks in two queues, the hot queue and cold queue (see [Figure 2.11](#)). In the hot queue, we keep the blocks that have been recently accessed. When releasing memory, we will not write them to the Physical Layer immediately, but to the cold queue. We organise the data in a flat fashion in the hot queue. We keep sorted triples instead of molecules to speed up atomic updates as well as index look-up operations. To save memory, the cold queue contains the molecule blocks (blocks in compressed form) that are ready to be written to the Physical Layer.

[Algorithm 1](#) illustrates the process of accessing a data block from the Buffer Manager. For requesting a given data block, the requester sends its block entry (of the Buffer Layer) to the Buffer Manager. First, the Buffer Manager looks up for the reference to

**Algorithm 1:** Access to a data block from buffer**input** : blockEntry : Buffer Layer block entry**output:** Block : reference to requested data block

---

```

1 Function fetchFromBuffer(blockEntry)
2   Block  $\leftarrow$  lookUpInBuffer(blockEntry);
3   /* If the requested block is not in buffer then read it from Physical Layer */
4   if isNull(Block) then
5     /*Check available memory for reading the data block*/
6     memoryCheck();
7     /*Read data from Physical Layer*/
8     addrtoRead  $\leftarrow$  computeAddress(blockEntry.getBlockId());
9     Block  $\leftarrow$  readBlock(addrtoRead);
10    Block  $\leftarrow$  decompress(Block);
11    addOrAdjustHotQueue(Block);
12    return Block;
13  else
14    /* If the block from Cold queue */
15    if isInColdQueue(Block) then
16      /* prepare space for decompressing compressed block */
17      memoryCheck();
18      Block  $\leftarrow$  decompress(Block);
19      /* Adjust recent order of block in the Hot Queue */
20      addOrAdjustHotQueue(Block);
21    return Block;

```

---

the requested data block in the main memory (line 2). If the requested data block does not exist in the main memory, the Buffer Manager checks if there is sufficient memory available for holding a new data block (lines 4–6). During the process of checking memory availability, exiting data blocks in the memory may be written to the Physical Layer to reclaim space for new blocks (see [Algorithm 2](#)). After having been read, the new data block is “decompressed” into its flat form, i.e., decomposed into individual triples, and added to the hot queue, while its reference is returned (lines 8–12). If the data block already existed in main memory, the Buffer Manager will check if it is in the the hot queue or in the cold queue. Note that the organisation of the hot queue and the cold queue is similar, the only difference is that hot queue holds uncompressed data while the cold queue holds compressed data. If the data block is in the cold queue, the system again prepares the space for decompressing the data block (lines 15–18). Before returning the reference of the data block, its position in the hot queue is adjusted to mark the data block as being recently accessed (lines 20–21).

[Algorithm 2](#) illustrates the memory checking procedure which is called in the first algorithm (line 6 and line 17). This algorithm guarantees that the system does not run out of memory during runtime. As we divide the buffer into the hot queue and the cold queue, the hot/cold ratio defines how many blocks are held in each queue. Depending on the

current hot/cold ratio, the Buffer Manager decides to move a data block from the hot queue to the cold queue, or to evict a data block from the cold queue and write it to the Physical Layer. For instance, when the available memory is not sufficient for holding a “flattened” data block (line 2), the system computes the current hot/cold ratio (line 4) and if the current ratio is higher than the predefined ratio, the data is evicted from the hot queue and moved to the cold queue. The system compresses the evicted data block from the hot queue and puts it into the cold queue (lines 6–10). Otherwise, if the current ratio is not higher than the predefined ratio, the data is evicted from the cold queue. If the evicted data block is modified, the system writes it back to the Physical Layer.

---

**Algorithm 2:** Check the available memory and evict data from the buffer if need

---

```

1 Function memoryCheck()
2   while  $mem_{avail} < blockSize$  do
3     /*compute current hot/cold ratio */
4      $ratio \leftarrow \frac{queue_{Hot}.size}{queue_{Cold}.size}$ ;
5     /* If current hot/cold ration is higher than the given ratio */
6     if  $ratio > ratio_{const}$  then
7       /* Move data from hot queue to cold queue */
8        $Block_{flatten} \leftarrow queue_{hot}.pop()$ ;
9        $Block_{compressed} \leftarrow compress(Block_{flatten})$ ;
10       $addOrAdjustColdQueue(Block_{compressed})$ ;
11    else
12       $Block_{toWrite} \leftarrow queue_{cold}.pop()$ ;
13      /*If the block is modified then write it to physical layer*/
14      if  $isDirty(Block_{toWrite})$  then
15         $block_{id} \leftarrow Block_{toWrite}.getId()$ ;
16         $addr_{toWrite} \leftarrow computeAddress(block_{id})$ ;
17         $writeBlock(Block_{toWrite}, addr_{toWrite})$ ;

```

---

Dividing the buffer into a hot queue and a cold queue and to move data blocks between them is our first strategy to delay write operations when the system needs more memory. However, in the worst case, the Buffer Manager still has to write the data to Physical Storage. To make the Buffer Manager more efficient, we keep data blocks in each queue in the order illustrated in Figure 2.11. This ordering follows the following priorities:

1. the clean/unmodified blocks;
2. the higher density blocks, defined by the ratio between the number of triples in a block and the capacity of the block i.e.,  $density_{BlockA} = \frac{\#triples_{BlockA}}{capacity_{BlockA}}$ ;
3. the least recently used blocks.

This prioritisation allows us to keep dirty/modified blocks in the buffer as long as possible to delay write operations and group more updates into dirty/modified blocks. In case we need to release memory, we always refer to the cold queue and start from the beginning of the priority list. Consequently, we prioritise clean/unmodified blocks to be evicted. Since we do not have to perform any write operation for them, we can just release the memory they occupy. Then, we prioritise blocks that contain many triples, i.e., high-density blocks, as they group multiple updates into one erase-write operation. Finally, the traditional LRU strategy is used to avoid writing the same block multiple times in a row.

## 2.7 Adaptive Strategy for Iterative Join Execution

The most resource-intensive task of answering a SPARQL query is to perform the graph pattern matching over the RDF dataset. The graph matching operator executes a series of join operations between RDF triples that match the triple patterns. Join operations have the greatest impact on the overall performance of a SPARQL query engine, typically requiring a large number of comparison operations that can only be done efficiently if records are stored in memory.

The join performance can be tuned by optimisation algorithms which plan the optimal join order and join algorithms (Stocker et al. 2008, Neumann & Weikum 2008, Tsialiamanis et al. 2012). These approaches assume that memory is always available during the execution of a chosen query plan. However, in light-weight computing devices, memory is critically low and, as such, the memory resources available to an RDF engine are unreliable, e.g., a surge of the number of network connections to the device might drain the available memory for all other running processes. Lack of memory may block join operations that require temporary virtual memory such as hash joins or sorted-merge joins, and thus hurts the overall performance of the query engine or probably crashes the engine.

Materialisation techniques that write intermediate join results to storage are an attractive solution for the issue of memory shortage (Ullman et al. 2001). However, on flash storage, writing is much slower if a random write operation happens. Furthermore, only a limited number of erase operations can be applied to a block of flash memory before it becomes unreliable and fails.

To minimise the memory required for executing a SPARQL query and making the best use of the indexing scheme introduced in Section 2.5, we adopt the *one-tuple-at-a-time* approach to compute the join. This approach can reduce the memory consumption as no virtual temporary memory is required to buffer the intermediate join results. The basic idea of the algorithm to compute the join of a graph pattern is as follows: A mapping solution (mapping for short) is continuously sent to visit each triple pattern of the graph pattern. In each visit, it searches for triples matching the triple pattern. For each matched

triple, variables in the triple pattern and the corresponding value in the triple are added to the mapping. The mapping with new values will be sent to visit the next triple pattern, or be returned as the query result when all triple patterns have been visited. The pseudo code of the join propagation algorithm is given in [Algorithm 3](#).

---

**Algorithm 3:** Join propagation
 

---

```

1 Function propagate( $\mu$ ,  $\mathbb{P}$ )
   input :  $\mu$  : mapping,  $\mathbb{P}$  : set of triple query patterns
   output:  $\mu$  : mapping
2 if isEmpty( $\mathbb{P}$ ) then
3   return  $\mu$ ;
4  $p \leftarrow$  findNextPattern( $\mu$ ,  $\mathbb{P}$ );
5  $p_{key} \leftarrow$  createKey( $\mu$ ,  $p$ );
6  $\mathbb{T} \leftarrow$  indexScan( $p_{key}$ );
7  $\mathbb{P}' \leftarrow \mathbb{P} \setminus \{p\}$ ;
8 for  $t \in \mathbb{T}$  do
9    $\mu \leftarrow$  bindMapping( $t$ ,  $p$ );
10  propagate( $\mu$ ,  $\mathbb{P}'$ );
11   $\mu \leftarrow$  resetMapping( $t$ ,  $p$ );

```

---

The propagate( $\mu$ ,  $\mathbb{P}$ ) function is used to recursively propagate the input mapping. The function starts with an empty mapping  $\mu$  and a set of unvisited triple patterns  $\mathbb{P}$ . For each run, it checks and returns the input mapping as a result, if there is no triple pattern left to visit (line 1–2). Based on the given input mapping, it looks for the optimal unvisited triple query pattern to visit (line 3). To search for triples matching pattern  $p$ , an index search key  $p_{key}$  is created by replacing the variables in  $p$  according to  $\mu$  (line 4). For each matched triple  $t$ , the corresponding variables and values are bound into the mapping. Then another propagation of the mapping to the remaining unvisited triple patterns is initiated (lines 7–10).

In each run of the propagation algorithm, the function findNextPattern( $\mu$ ,  $\mathbb{P}$ ) is called to find the optimal triple pattern to execute the propagation (see [Algorithm 4](#)). For each triple pattern  $p$  in  $\mathbb{P}$ , the set of triple query patterns, the function searches for a triple pattern that shares variables with the input mapping  $\mu$  (line 4). For each shared pattern found, an index search key pattern,  $p_{key}$ , is created (line 5). An index lookup on  $p_{key}$  is executed to search for the upper bound and lower bound positions of the set of matching triples in the index, as described in the previous section (line 6). The size of the index lookup  $\mathbb{I}$  is defined as the range between the upper bound and lower bound positions (lines 7–9). The function returns the triple pattern that has the minimal size of the index lookup (line 11).

The join propagation algorithm is similar to the nested iterations. Nested loop join is often argued to have a poor performance as it does not attempt to prune the number of comparisons. However, supported by an efficient index scheme, an index nested loops join

**Algorithm 4:** Find next pattern

---

```

1 Function findNextPattern( $\mu$ ,  $\mathbb{P}$ )
  input :  $\mu$  : mapping,  $\mathbb{P}$  : set of triple query patterns
  output:  $P$  : triple query pattern
2    $p_{next} \leftarrow null$ ;
3    $s_{min} \leftarrow Integer_{max}$ ;
4   for  $p \in \mathbb{P}$  do
5     if  $isShared(\mu, p)$  then
6        $p_{key} \leftarrow createKey(\mu, p)$ ;
7        $I \leftarrow indexLookup(p_{key})$ ;
8        $s \leftarrow sizeOf(I)$ ;
9       if  $s < s_{min}$  then
10         $s_{min} \leftarrow s$ ;
11         $p_{next} \leftarrow p$ ;
12  return  $p_{next}$ ;

```

---

can perform as well as other join algorithms (Graefe 2003). With the design of our storage, the index lookup can be done mostly within the buffer layer, only two extra I/O operations may be required. The visitor pattern, that sends a mapping to visit each triple pattern and to execute the index lookup, reduces the extra memory for the joins as only a mapping is kept in the main memory. This mechanism also enables the adaptivity of the joins. The function  $findNextPattern(\mu, \mathbb{P})$  decides which triple pattern the mapping should visit first. Similarly to the routing policy of stream processing engines, e.g., Eddies (Avnur & Hellerstein 2000) or CQELS (Le-Phuoc et al. 2011), this function defines the propagating policy to achieve a certain optimisation purpose. In our case, we attempt to minimise the number of propagations by choosing the shortest index scan in each run. Note that this is also the key place for adding sophisticated optimisation algorithms, e.g., adaptive caching algorithm to be discussed in our future work.

## 2.8 Evaluation Results

Operating systems for a specific type of device are usually customised and optimised to meet a specific hardware configuration. For example, by default, Raspbian is installed on the Raspberry Pi Zero while a Galileo Gen II is running the Yocto 1.4 Poky Linux distribution. A Java virtual machine is available on most edge devices and Java is platform independent. Hence, we choose to implement our approach in Java to take advantage of its “compile once run anywhere” property that enables the portability of our RDF4Led engine.

RDF4Led is developed by reusing Jena TDB code base and following the RISC-style design as presented in Section 2.4.1. We only selected the required components and

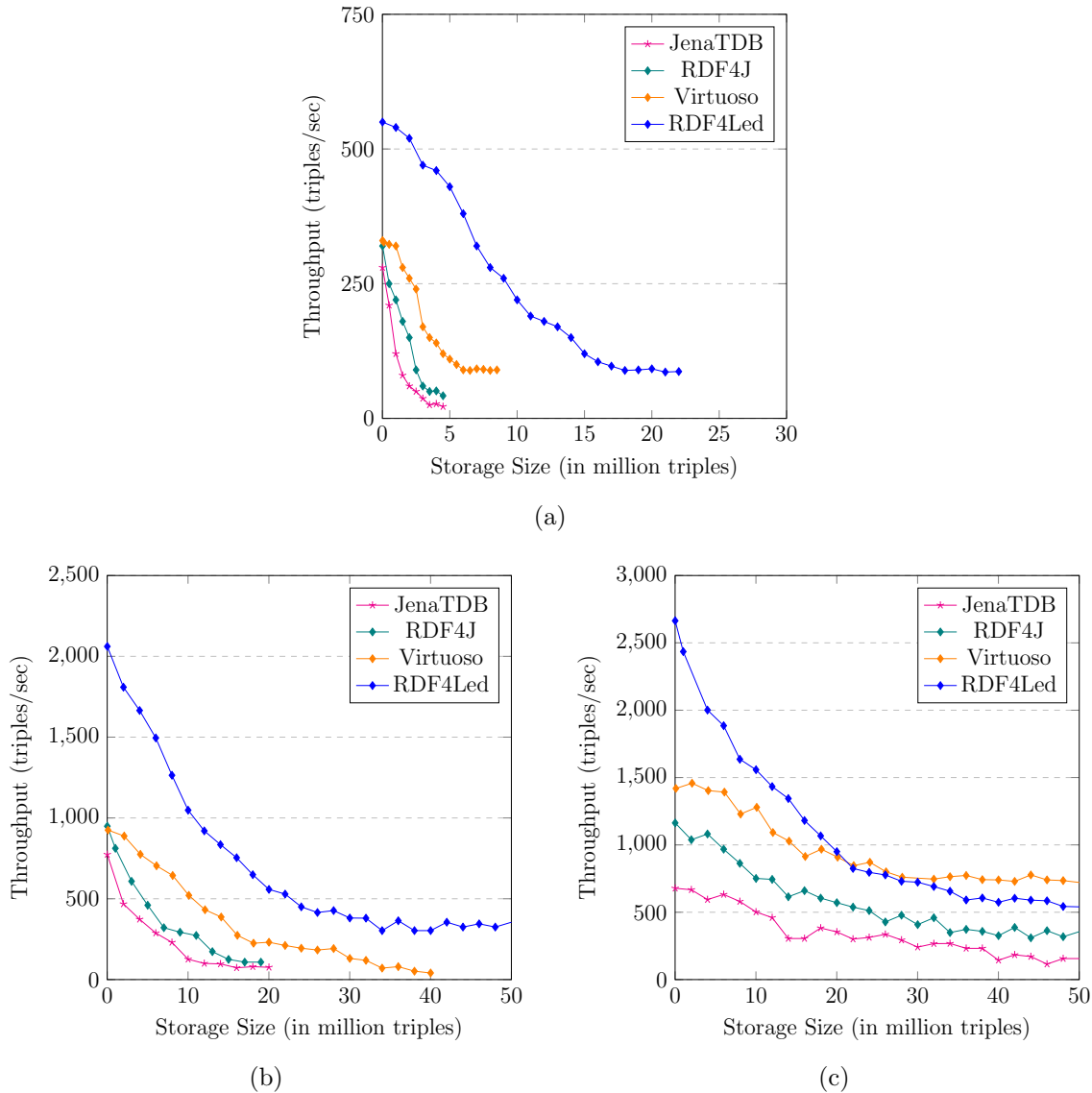


FIGURE 2.12: Insert throughput of RDF4Led compared to Jena TDB, RDF4J and Virtuoso on Gallileo Gen II, Raspberry Pi Zero and Raspberry Pi 3. (a) Insert throughput results on Gallileo Gen II; (b) Insert throughput results on Raspberry Pi Zero; (c) Insert throughput results on Raspberry Pi 3.

reimplemented those with our algorithm (see [Sections 2.5–2.7](#)). As a result, the size of RDF4Led is only 4 MB, while the size of Jena TDB is 13 MB; the size of RDF4J is 58 MB; and the size of Virtuoso is 180 MB.

[Figure 2.12](#) reports the result of **Exp1** in which we measure and compare the update throughput of RDF4Led with Jena TDB, RDF4J, and Virtuoso. The description of **Exp1** can be found in [Section 2.3.4](#). Due to the similar result, we omit the experiment results for Raspberry Pi 2 (RPi3), and Beagle Bone Black (BBB), and report the experiment results on Intel Galileo Gen II (GII), Raspberry Pi Zero W (RPi0), and Raspberry Pi 3 (RPi3).

The results show that in comparison to Jena TDB and Virtuoso, RDF4Led can store a larger dataset and has a much higher update throughput. On GII, RDF4Led scaled up to 22 million triples, which is three times larger than Virtuoso and four times larger than Jena TDB. We stopped the experiment of RDF4Led on GII when its speed dropped below 80 triples/second after inserting 22 million triples. The scalability of RDF4Led on Pi0 (see [Figure 2.12\(b\)](#)) and BBB was similar. On both these devices, RDF4Led was able to add the full 50 million triples. The update throughput of RDF4Led also decreased when the number of triples in the store increased. Among the three engines, RDF4Led has the highest inserting throughput on GII and RPi0. On GII, with 5 million triples inserted, the update throughput of RDF4Led was still around 350 triples/s, whereas Virtuoso was only able to insert data with a speed of 125 triples/s. On Pi0, RDF4Led performed update operations two to three times faster than Virtuoso. Even when the size grew to nearly 50 million triples, RDF4Led's speed still stayed at 300–350 triples/s, which was two times faster than the speed of Virtuoso with only 15 million triples. On RPi3, RDF4Led updated faster than Virtuoso in the beginning. When its storage size reached 20 million RDF triples, the speed of RDF4Led became lower than that of Virtuoso. However, RDF4Led still ran faster than RDF4J and Jena TDB.

The **Exp1** results show how the lack of memory negatively influences the scale of standard RDF engines like Jena TDB, RDF4J, and Virtuoso. RDF4Led can insert more data as it has a smaller memory footprint and requires less memory to maintain the indexes. Furthermore, compared to the other engines, RDF4Led inserts data faster as our flash-aware index structure and writing strategy are better compatible with the flash memory's I/O behaviour. Other RDF engines employ B<sup>+</sup> tree to index RDF data in their storage. Their low throughput confirmed the negative influence of flash I/O behaviour on the write performance of such disk-based indexing techniques. Because RDF4Led was designed to save memory on devices with more RAM like RPi2 and RPi3, Virtuoso can run faster than RDF4Led. Virtuoso's algorithms need bigger memory buffers to achieve their best performance.

The results of **Exp2** are shown in [Figure 2.13](#). In this experiment, we compared the query response time of RDF4Led with that of three engines on GII, RPi0 and RPi3 with a dataset size that all engines could handle. The results show that RDF4Led answered all queries considerably faster than Jena TDB and RDF4J on all devices. RDF4Led, RDF4J and JenaTDB follow the nested execution model to compute multiple joins between RDF triples that match triple patterns. However, Jena TDB and RDF4J were implemented with an iterator pattern, while RDF4Led followed the visitor pattern. In general, both algorithms execute lookup operations and index scan operations to extract the compatible triples from the dataset. The performance of these algorithms is mainly influenced by the performance of the lookup and index scan operations on the indexes. The better performance of RDF4Led indicates that our lightweight index structure helps RDF4Led outperform the B<sup>+</sup> tree implemented in Jena TDB.

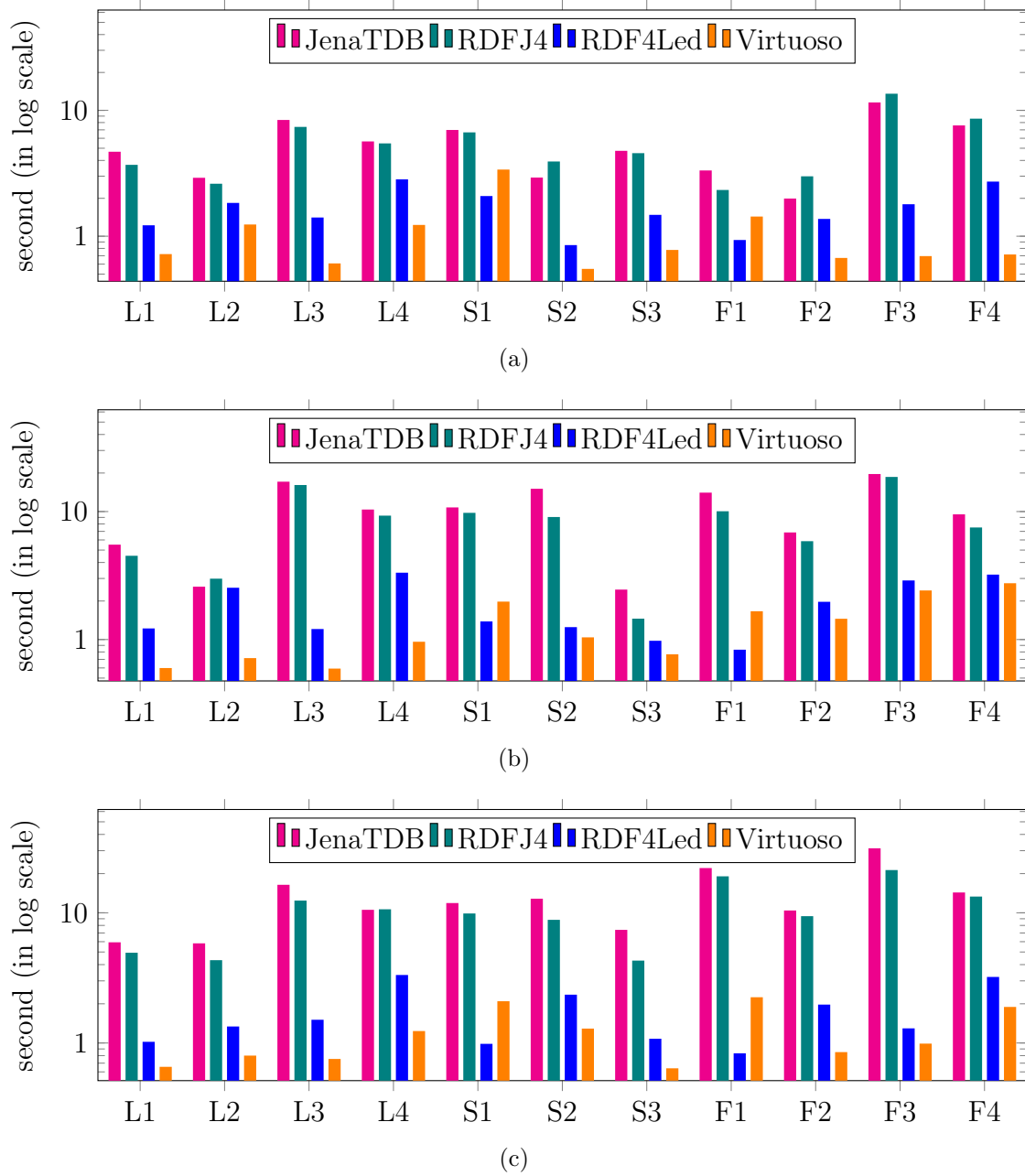
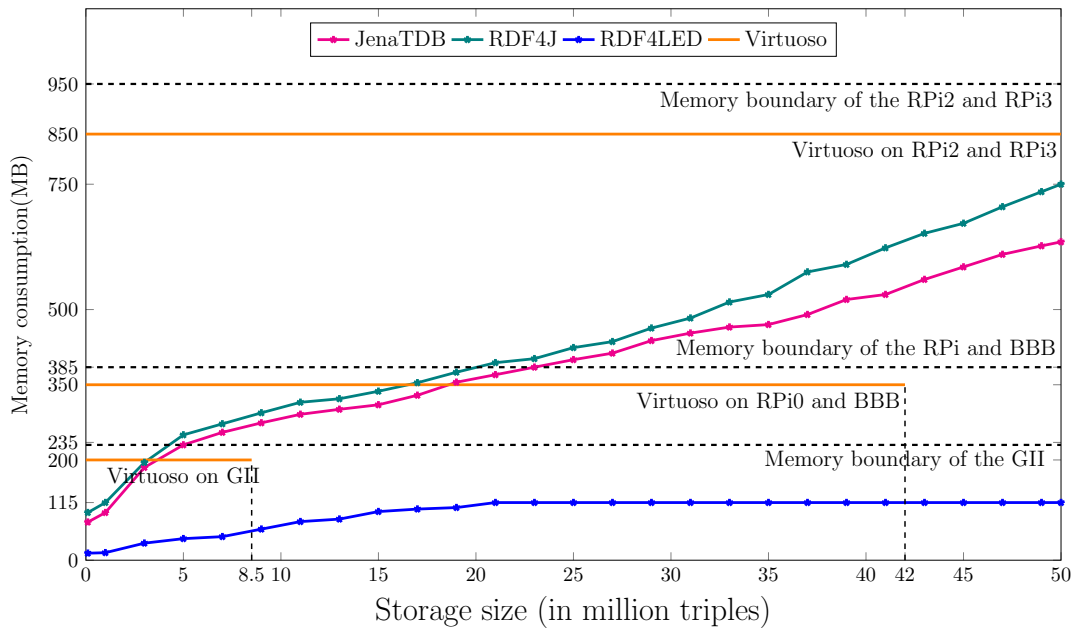
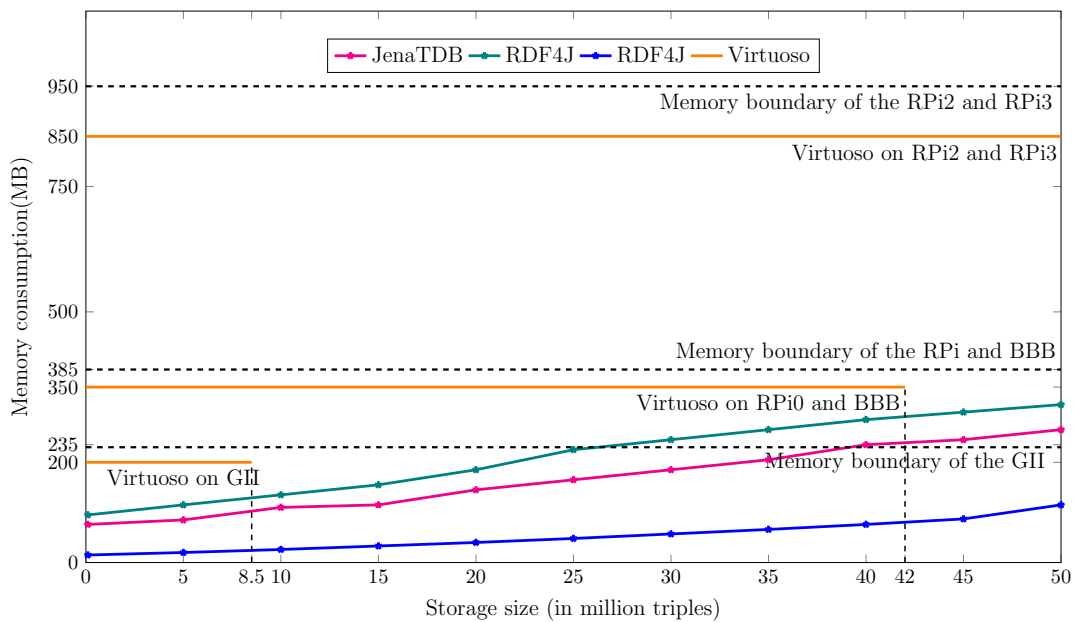


FIGURE 2.13: Query test results of Jena TDB, RDF4J, Virtuoso, and RDF4Led. (a) Query response time against 5 million triple dataset on Gallileo Gen II; (b) Query response time against 20 million triple dataset on Raspberry Pi Zero; (c) Query response time against 50 million triple dataset on Raspberry Pi 3.

On the same dataset and on the same device, RDF4Led only answers the queries generated from templates F2 and S1 faster than Virtuoso does. These queries include star-shaped patterns of more than 6 triple patterns. In other cases, RDF4Led is slower than Virtuoso as it does not aggressively pre-allocate a fixed amount of memory for sophisticated optimisation algorithms (Virtuoso allocates two to three times more than RDF4Led). We see this as an option to improve query performance in our future work. However, at the current stage, RDF4Led can deliver good performance for datasets of up to 50 million



(a)



(b)

FIGURE 2.14: Memory consumption of Jena TDB, RDF4J, Virtuoso, RDF4Led. **(a)** Memory consumption report of update throughput test; **(b)** Memory consumption report of query evaluation test.

triples on these devices, e.g., 5 s at maximum and 1 s on average query execution time. The performance and scalability of RDF4Led can enable these kinds of low-capability devices to handle approximately 600 thousand sensor observations or 6 months worth of data of 25 weather stations in a single active RDF graph.

For **Exp3**, the memory consumption of RDF4Led and other engines is reported in [Figure 2.14](#). In the insertion experiment, RDF4Led consumed only 85 MB of memory even when the storage went up to 50 million triples. In the query evaluation experiment, RDF4Led used less memory than the other engines did in the insertion test. Even with a dataset of 50 million triples, RDF4Led used only 80 MB. This was only half of the memory that Jena TDB used in the query test with 10 million triples and was only 10% of the memory that the Virtuoso occupied constantly.

## 2.9 Conclusions

This paper presents an empirical study of the scalability of RDF engines running on resource-constrained devices, which are representative of typical IoT edge nodes. The results of the study show that these engines do not scale on such devices due to the lack of main memory and the special I/O requirements of flash storage. To address these problems, we proposed RDF4Led, a RISC-style approach to building an RDF engine tailored to resource-constrained edge devices. Our approach includes a flash-memory-aware storage structure, a flash-memory-aware buffer management strategy for RDF data, and a low-memory-footprint join strategy for improved scalability as well as robustness. The RDF4Led engine is significantly smaller in terms of memory footprint than generic RDF engines like Jena TDB, RDF4J, or Virtuoso. We tested RDF4Led on five different types of ARM boards. These experiments showed that RDF4Led can handle 2–5 times more data than its competitors. Moreover, RDF4Led requires only 10%–30% of the memory consumed by Jena TDB, RDF4J and Virtuoso when operating on the same size of the dataset. It can handle up to 50 million triples with approximately 115 MB of memory and can outperform its competitors in updating throughput; it is faster in answering queries than its Java counterpart, Jena TDB. While Virtuoso can deliver faster query processing time, it does so by pre-allocating a fixed amount of memory, which is 3 times more than that required by RDF4Led and with a significantly more complex implementation.

## Chapter 3

# Querying Heterogeneous Personal Information On The Go

The work outlined in this chapter was published in:

Le-Phuoc, D., **Le-Tuan, A.**, Schiele, G. and Hauswirth, M., 2014, October. Querying heterogeneous personal information on the go. In International Semantic Web Conference (pp. 454-469). Springer, Cham.

## Abstract

Mobile devices are becoming a central data integration hub for personal information. Thus, an up-to-date, comprehensive, and consolidated view of this information across heterogeneous personal information spaces is required. Linked Data offers various solutions for integrating personal information, but none of them comprehensively addresses the specific resource constraints of mobile devices. To address this issue, this paper presents a unified data integration framework for resource-constrained mobile devices. Our generic, extensible framework not only provides a unified view of personal data from different personal information data spaces but also can run on a user's mobile device without any external server. To save processing resources, we propose a data normalisation approach that can deal with ID-consolidation and ambiguity issues without complex generic reasoning. This data integration approach is based on a triple storage for Android devices with small memory footprint. We evaluate our framework on a set of experiments on different devices and show that it is able to support complex queries on large personal data sets of more than one million triples on typical mobile devices with very small memory footprint.

**Contributions:** The author of this thesis partly shares with the first author the contribution to the approach for ID-consolidation ([Section 3.2](#)). The author of the thesis works with the third author on designing the system architecture ([Section 3.3.1](#)). In this paper, the main contributions of the author of the thesis are the development of the RDF store ([Section 3.3.2](#)) and the experimental study presented ([Section 3.4](#)).

## 3.1 Introduction

The availability of an up-to-date, comprehensive and consolidated view of a user's social context not only enables novel applications such as distributed social network (Tramp et al. 2011), semantic life (Hoang et al. 2006), or semantic desktop (Weippl et al. 2007) but is increasingly becoming an essential requirement for many mobile applications. As an example, consider a typical mobile user who has access to the contact information of his acquaintances via Facebook, LinkedIn, Google+, and his phonebook. Each of these data sources may contain different types of information about this user, e.g., personal and professional information, phone numbers, or message and call histories, and they may exhibit different levels of quality. Consequently, a contact management application should be able to link and integrate all this information and automatically extract and integrate the right pieces of information from the whole set of data sources available. Similarly, a messaging widget should be able to integrate the messages from different services to track the user's conversations across all messaging service platforms.

However, the creation and continuous maintenance of such a view is a challenging task. The reason for this is twofold: First, despite the steady increase in computation and communication capabilities, mobile devices are battery powered. As a result, developers typically spend a considerable fraction of their development time on minimising the amount of computation and bandwidth utilisation to reduce the impact of their applications on the device's energy profile. Second, despite the popularity of some mainstream social networking services, the creation of a truly comprehensive view on the user's context usually requires the integration of considerable amounts of data from a user-specific set of services. As a result, developers must provide mechanisms to deal with the integration of complementary as well as overlapping and possibly inconsistent data sets.

Existing solutions typically use one of the following two approaches: Either they may use a powerful and well-connected cloud infrastructure to perform the data integration (Quan et al. 2003) or they may focus on the integration of an application-specific set of data types from a (possibly) limited set of services (Rekimoto 1999). The first approach requires the provisioning of access credentials to the centralised/cloud-based data integration infrastructure, which may then access, process and store the user's information. This remotely processing approach for mobile applications raises several privacy and security concerns as security credentials leave the device and privacy is given up (entrusted to the cloud/remote servers without control by the users and without means to enforce it). Therefore, granting access to the mobile device being under the full control of the user is desired in ongoing security and privacy debates in many countries in respect to the cloud. To this end, the second approach is the preferable choice. However, it is not cost-effective as it requires developers to repeatedly make complicated design decisions. Furthermore, it may be also inefficient, especially in cases where multiple applications require access to the same data resulting in duplicate data retrieval and integration.

In this paper, we present an alternative approach for data integration by introducing a comprehensive framework that takes care of data retrieval, identity consolidation, disambiguation, storage, and access locally on mobile devices. To reduce privacy and security concerns, the framework does not require any remote storage and processing. It is solely executed on the mobile device of a user. In contrast to application-specific approaches, our framework is generic with respect to the supported types of data. It is extensible with respect to the supported services and it is open with respect to application support. To achieve this, the framework (1) leverages Linked Data to facilitate the storage of arbitrary types of data, (2) employs a plug-in model to connect to different services, and (3) provides a generic query processor with support for SPARQL to be open with respect to application support. As a validation of the usefulness of the framework and to verify the efficiency of the framework, we present the results of an extensive experimental evaluation.

The remainder of this paper is structured as follows: In the next section, we describe our approach for data consolidation and integration on mobile devices. After that, we present the design and implementation of our framework and evaluate its performance. Finally, we discuss related work and finish the paper with our conclusions and discuss directions for possible future work.

## 3.2 Integration of Heterogeneous Personal Information

To integrate the personal data from different data sources, several approaches proposed a unified data model for transforming heterogeneous data formats to RDF driven by agreed-upon vocabularies (FOAF, SIOC, vCard, etc) (Bojārs et al. 2008). RDF statements are used to link and describe people, their social relationships, the content objects relevant to them, etc. However, a person can have multiple identifiers (IDs) on different data spaces. When they are integrated in a single data space, these IDs have to be interlinked and unified to represent a unique person. To uniquely identify someone across various data spaces, there are some rules that have to be set to infer and ensure the uniqueness of that person. Along with some explicit properties like *owl:sameAs*, there are some implicit rules defined from the properties indicating that two IDs are “talking” about the same person (Bojārs et al. 2008). For instance, in practical, an “*inverse-identification*” property is used as an indirect identifier, e.g., foaf:phone, foaf:mbox\_sha1sum. Therefore, having multiple identifiers poses several challenges for aggregating personal data from heterogeneous data spaces to store in RDF and to make it useful on resource constrained devices.

To demonstrate why it is challenging to enable a unified, integrated view of heterogeneous personal information sources on mobile devices, let us take a closer look on the example depicted in Figure 3.1. The data shown in this figure is acquired and transformed to RDF from Facebook, Google+, LinkedIn and phone contacts of a user’s mobile phone in a similar fashion as proposed in (Tramp et al. 2011, Bojārs et al. 2008).

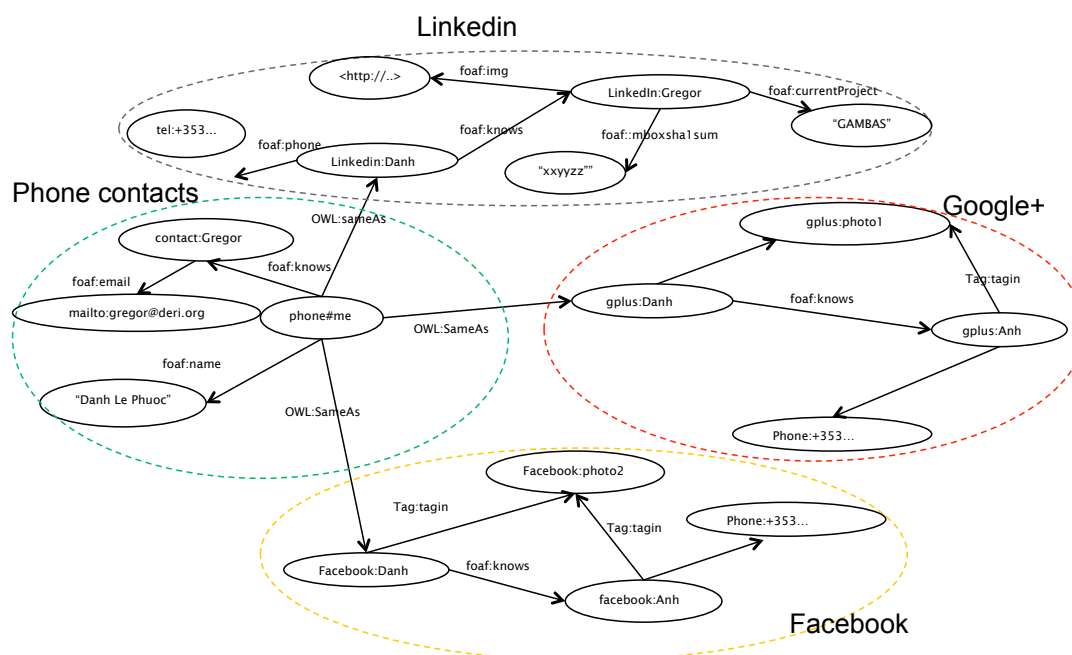


FIGURE 3.1: Simple RDF graph integrated from data silos

The explicit *owl:sameAs* statements are added to link a user’s IDs from different data spaces. In addition, two RDF nodes, *facebook:Anh* (Facebook identifier) and *gplus:Anh* (Google+ identifier), represent one person because they have the same inverse-identification property *foaf:phone* with the same value  $\langle phone : +35389... \rangle$ . Similarly, two RDF nodes, *linkedin:Gregor* (LinkedIn identifier) and *phone:Gregor* (identifier given by the phone’s contact application) also represent one person because the *sha1sum* value of his email has the same value as his *foaf:mbox.sha1sum* in LinkedIn. In essence, this RDF graph implicitly represents “different” pieces of information of three people who have different RDF statements attached to different RDF nodes representing each of them. However, if we store the simple RDF reification<sup>1</sup> of this graph in a standard RDF store, the SPARQL query processor will not be able to return complete information about a person. For instance, the query *”SELECT ?friends WHERE {phone:me foaf:knows ?friend}”* can only return one friend with the identifier *contact:Gregor* from the explicit statement in the phone contacts. Standard SPARQL is not able to infer the implicit statement  $\{phone:me foaf:knows facebook:Anh\}$  because *phone:me* and *facebook:me* is the same person.

<sup>1</sup><http://www.w3.org/TR/rdf-primer/>

The solution for this problem is to use entailment regimes<sup>2</sup> instead of the simple entailment in the above graph. This requires a modification in the SPARQL query processor to employ a reasoner to infer implicit RDF statements for basic graph pattern matching operators. However, this approach is not practical because it needs a considerable amount of memory and a fairly powerful CPU for the reasoning process. Another alternative solution is to use an ID consolidation approach (Quan et al. 2003, Dong & Halevy 2005) to compute all implicit RDF statements, then store them in an RDF store and query it with a standard SPARQL query processor. However, this approach is hard to adopt for resource constrained mobile devices. On top of that, having all possible explicit RDF statements in an RDF store is expensive for both updating and querying the data stored in the storage. It is even more expensive for incrementally updating the RDF store because the data needs to be synchronised with the original data sources (Schandl & Zander 2009, Tummarello et al. 2007).

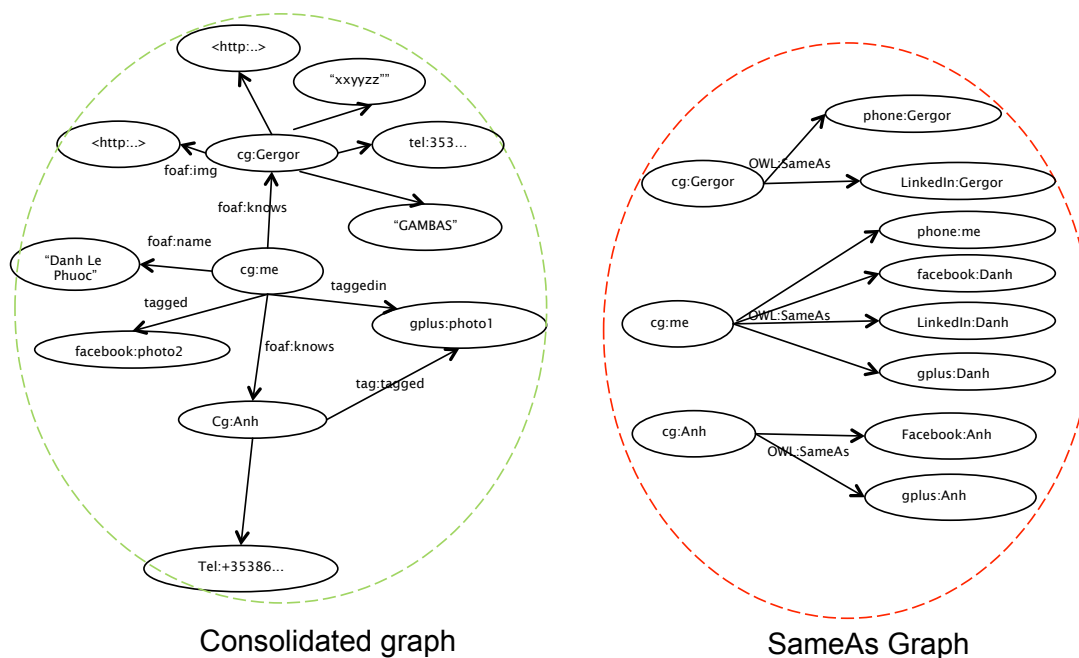


FIGURE 3.2: Consolidated and SameAs graph

To remedy these problems, we propose to create a unified integration view by managing additional graphs to query all personal information desired. Firstly, we manage a “consolidated graph” that contains the aggregated personal information from different data spaces. As illustrated on the left of Figure 3.2, the consolidated graph provides an aggregated view of personal information, so that a standard SPARQL query processor can provide complete answers relevant to a person. Note that this consolidated graph uses only one ID scheme that provides a single ID for one person. However, the integration view also has a *SameAs graph* that links the consolidated IDs with their counterparts given in other data spaces as shown on the right of Figure 3.2.

<sup>2</sup><http://www.w3.org/TR/sparql11-entailment/>

To store and manage the provenance information of the data acquired from difference data spaces, the integration view also stores the data from each data space as a named graph. This enables queries to correlate the consolidated graph with a graph containing information from a particular data space. For instance, the following query is used to query all friends in Facebook that are tagged with “me in a photo” posted in Google+ or Facebook or other data spaces.

```

1 SELECT ?fbfriend
2 FROM NAMED ds:facebook
3 FROM NAMED ds:cg
4 FROM NAMED ds:sameas
5
6 WHERE {
7   GRAPH ds:facebook { Facebook:me foaf:knows ?fbfriend. }
8   GRAPH ds:cg        { ?cgfriend pim:tagged ?photo.
9                       ?cgme     pim:tagged ?photo.}
10  GRAPH ds:sameas    { ?cgfriend owl:sameAs ?fbfriend.
11                       ?cgme     owl:sameAs Facebook:me. }
12 }

```

LISTING 3.1: An example of a SPARQL query to find all friends in Facebook that are tagged with me in a photo posted in Google+ or other data spaces.

To relieve the user of the burden of using the proper identifiers corresponding to the data space and writing such a long query involving the *SameAs* graph, it should be possible to use the user identifiers in queries as all identifiers will be translated to the proper ID scheme based on the context given by the GRAPH keyword. For instance, a Facebook ID *Facebook:me* will be translated to the corresponding one in the consolidated graph by the query processor. The above query could be written in a shorter form as follows:

```

1 SELECT ?fbfriend
2 FROM NAMED ds:facebook
3 FROM NAMED ds:sameas
4
5 WHERE {
6   GRAPH ds:facebook { Facebook:me foaf:knows ?fbfriend. }
7   GRAPH ds:cg        { ?cgfriend  pim:tagged ?photo.
8                       Facebook:me pim:tagged ?photo.}
9 }

```

LISTING 3.2: A short form of a SPARQL query to find all friends in Facebook that are tagged with me in a photo posted in Google+ or other data spaces.

To create and maintain the unified view composed from such graphs, we would need a data integration platform that requires several features specifically designed for mobile devices. The first feature is the data aggregation from heterogeneous data sources. After the data is aggregated, it has to be consolidated to create constituent graphs for the integrated view. To store and query data from these graphs, the platform also needs a fully fledged RDF store tailored to the needs of resource-constrained devices. On top of that, the data in this RDF store has to be accessed in a controlled manner to meet the security and privacy concerns of personal data. These requirements drive the design and implementation decisions of our framework in the following section.

### 3.3 System Design and Implementation

To enable querying heterogeneous personal information with the unified view described in the previous section, we design the system architecture to meet the aforementioned requirements in following. We also describe the implementation of the core component, the RDF store for mobile devices, that dictates the expected performance of the whole system in the context of resource constraints.

#### 3.3.1 System Architecture

Figure 3.3 shows the overall system architecture which we will discuss in the following.

As discussed before, there are different sources for personal information like Facebook or Google Calendar that developers should be able to integrate into our framework. To do so, our framework allows developers to create *Connector* classes and plug them into the framework using the *Connector Manager*. Each Connector is tailored towards a specific information source. So far, we provide a core set of Connectors, namely for Facebook, Google+, Google Calendar, LinkedIn and local mobile phone content. If the developers need to access additional information sources, they can easily implement new connectors for them using the existing ones as a blueprints.

Each connector pulls relevant information from its information source and pushes it towards the *Data Consolidator*. The Data Consolidator consolidates and integrates data from different connectors into the corresponding RDF graphs for each data source. The Data Consolidator also computes the aggregated graph and SameAs graph as described in Section 3.2. These RDF graphs are stored as an integrated view into the RDF Store.

The *RDF Store* is a core component of our system as it manages triple data directly on the mobile phone instead of on an external server. RDF triples can be stored, indexed, and retrieved. The store contains the actual personal data as well as all metadata needed for data consolidation, e.g., user IDs in different data sources and how they relate to each other. The RDF Store has a major influence on our system performance and thus must

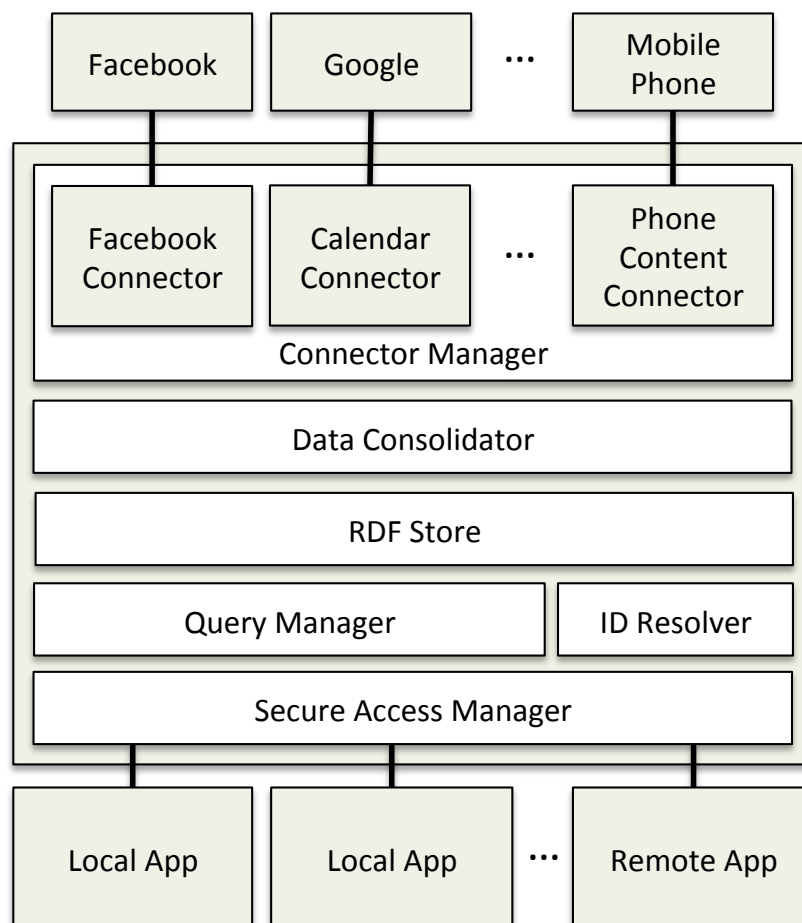


FIGURE 3.3: System architecture overview

be highly efficient both in terms of execution speed and memory usage. We therefore implemented a RDF store that is specifically tailored towards mobile devices instead of using a feature reduced version of a well-known system like Jena. We will discuss the design and implementation of our RDF Store in more detail later in this section.

To access the RDF Store, clients can use two system components, the *Query Manager* and the *ID Resolver*. The Query Manager can handle standard SPARQL queries on the data in the RDF Store. In addition to standard query planning and execution, the Query Manager is also responsible for rewriting queries if necessary. This is the case if the query contains an ID for a user that originates in one of the original data sources, e.g., the ID of a user in Facebook. The Query Manager detects this and rewrites the query such that a consolidated ID is used. This allows clients to place queries without knowing about the data consolidation. From the client's point of view, it can use the data as if all of it was available on Facebook. The ID Resolver offers an alternative way of dealing with multiple IDs. It allows a client to request information about a user's ID from different data sources. As an example, a client can ask for the IDs of a user for which it provides the Facebook ID. The ID Resolver looks up the necessary metadata in the RDF Store

and returns all IDs for this user, the consolidated ID as well as the user's ID in Google+, etc.

Clearly, security and privacy are major factors when designing a system that manages personal data. Therefore, we chose to add an additional system component, the *Secure Access Manager*, which is responsible for ensuring that all client accesses are done in a secure and privacy-preserving manner. The Secure Access Manager receives requests from local as well as remote client applications. It authenticates the requesting clients and checks their authorisation to access private data. Authorisation is given by the local user using so-called privacy policies. If access is granted, the Secure Access Manager forwards the request to either the Query Manager or the ID Resolver. It also forwards any results to the requesting client.

To ensure secure communication with remote clients, the Secure Access Manager uses the PIKE approach for secure peer-to-peer communication establishment for mobile devices (Apolinarski et al. 2013). PIKE includes mechanisms to initiate a secure key exchange and to establish a secure network connection with it. It is also able to set up an ad hoc communication network between mobile devices if necessary.

### 3.3.2 High Performant and Low Memory Consumption RDF Store

As presented in Section 3.2, our solution for maintaining a unified integration view of heterogeneous personal information is to manage a consolidated graph for the aggregated data from different spaces and a sameAs graph to link the consolidated IDs with their counterparts. Applying this approach on mobile devices requires a mobile RDF store component which is designed for update-intensive operations. To this end, we built a native and fully fledged, persistent RDF storage and SPARQL query processor for Android devices, called *RDF On the Go* (RDF-OTG)<sup>3</sup>. RDF-OTG has been extensively used for managing semantic contextual information on mobile devices in PECES<sup>4</sup> and GAMBAS<sup>5</sup> projects. In our implementation, we focused on minimising the memory footprint and designing data structures tightly coupled with the storage mechanism of mobile devices to achieve maximum efficiency in terms of low memory consumption and high update frequency. In the following, we briefly describe our main optimisation to maximise performance and scalability for personal information management applications on mobile devices. A full analysis of the performance gains is given in Section 3.4.

Reducing memory consumption is one of the critical key targets in mobile DBMS design (Nori 2007) since most mobile devices have (relatively) limited memory. To achieve

---

<sup>3</sup>RDF-OTG is open-sourced at <https://code.google.com/p/rdfonthego/>

<sup>4</sup><http://www.nes.uni-due.de/research/projects/peces/>

<sup>5</sup><http://www.gambas-ict.eu/>

that, we reduce the memory footprint of data operations on RDF data by using dictionary encoding, similar to the implementations of JenaTDB or Sesame. Each RDF node is mapped to a compact 32-bit integer with 9 bits to encode the node type and the remaining 23 bits encoding a string identifier which is kept separately on the flash memory instead of in the main memory. Most operations on nodes, e.g., matching during a query execution, can be performed on these node identifiers without accessing the actual string representation. Thus, only one integer must be kept in memory for each node, while string representations can be stored on the flash memory. This leads to a memory footprint of just up to 12 bytes per triple. This is considerably low compared to 450 bytes per triple for the Jena Memory Model as reported by the memory profiler. Note that the compact integer format is used for optimising millions rather than billions of RDF nodes, which we believe this is the common scale of most mobile personal information applications. For instance, 1.5 million triples are required to represent the information of 1200 user profiles (cf. Section 3.4). However, if necessary, this restriction could be easily removed.

Mobile devices are equipped with flash memory as the secondary storage. Flash memory has no mechanical latency, reading is faster than writing, and the storage is organised in memory blocks. Instead of reading or writing individual bytes, the I/O unit always reads/writes the whole block. That leads to its erase-before-write limitation when writing a single byte in a block, i.e., the whole block must be read, modified, and written again. Thus, to achieve the writing requirement, our RDF store needs to optimise writing efficiency rather than reading efficiency. To simplify the writing process, we use the simplest version of the multiple indexing framework of RDF data. It contains only three cyclic orderings of a triple's components with respect to subject(S), predicate(P), and object(O): SPO, POS and OSP. Each indexing order of triples is stored in a separate table.

Due to the impact of flash memory, unmodified versions of traditional data structures do not perform well. On the other hand, flash-aware indexing structures do not work well with “narrow and long” tables as resulting from the above indexing approach. Thus, we use a two-layer indexing approach to manage these tuples of three encoded integers in their corresponding tables. In each table, tuples are lexicographically sorted, partitioned, and compressed into individual fixed-size and same-length blocks to flash I/O block size of the device. The second index layer is a sparse index, small enough to fit into the main memory to enable fast lookup for the triples contained in each block. The index holds the lowest and highest node identifier in each sorted block. We also use an in-memory caching mechanism which maintains a limited number of frequently used index blocks.

If a new triple is added, it must be added to the indexes. To do so, the system loads the required index blocks into the cache. Then the triple must be allocated at the right position in the index. This is trivial if the triple should be added at the end of an existing block that still has open space. Otherwise, we would need to move all triples by one position, resulting in a large number of writes. To further reduce the number of read/write accesses, when we need to remove a block from the cache and write it back to

flash, our strategy chooses a block that has the highest chance of not being changed in the future.

## 3.4 Experimental Evaluation

The approach for data integration presented in Section 3.2 avoids reasoning tasks by modifying RDF triples and then storing them in a unified integration view. This solution is suitable for mobile devices since it does not require much memory for executing the reasoning tasks, but it requires a highly performant mobile RDF store when the graphs have to be modified frequently to maintain a unified view. Thus, performance of the system described in Section 3.3 heavily depends on the performance of the back-end mobile RDF store. In this section, we present a thorough experimental evaluation<sup>6</sup> of our system’s performance and scalability in terms of data updating and querying. The evaluation uses two system configurations with different mobile RDF stores to evaluate its impact on the system’s performance and to measure the efficiency gained through our RDF store. In the following, we first describe the setup of the experiments and then present and discuss the results obtained from the results.

### 3.4.1 Evaluation Setup

Our evaluation setup is as follows: To evaluate the impact of our special triple store on system performance, we compare two different system configurations. The first one uses our system as described in the last section, i.e., it uses our triple store, RDF On The Go (RDF-OTG). The original implementation of RDF-OTG was presented in (Le-Phuoc et al. 2010). Since then, RDF-OTG has been completely redesigned and reimplemented for maximum performance in Section 3.3.2. In the experiments we use the most recent version. In the second configuration, we replaced RDF-OTG by TDBoid<sup>7</sup>, a version of Jena TDB for Android OS. The rest of the system remained unchanged.

To evaluate how different device profiles with different resources and capabilities impact on the performance, we use three classes of Android devices in the experiments: a HTC Desire, a Samsung Galaxy Nexus, and a Nexus 7 Tablet. Their configuration details are described in Table 3.1.

For the evaluation dataset, we use a social network data generator (Pham et al. 2012) to generate three social networks, one for Facebook, one for Google+, and one for LinkedIn. From these, we extract relevant data profiles for a person, i.e., the profiles of that person and his/her friends, and feed them into our system. The data generator generates random inverse identification properties, e.g, mbox\_sha1sum, phone from the same dictionary for

<sup>6</sup>The description of how to reproduce the results can be found at <https://code.google.com/p/rdfonthego/>

<sup>7</sup><https://code.google.com/p/androjena/>

HTC desire	Samsung Galaxy Nexus	Nexus 7 Tablet
AndroidOS 2.3.3	AndroidOS 4.2.2	AndroidOS 4.2.2
998Mhz CPU	1200Mhz CPU	1300Mhz CPU
404MB physical RAM	694MB physical RAM	974MB physical RAM
32MB DVM heap size	96MB of DVM heap size	64MB of DVM heap size

TABLE 3.1: Android devices

three social networks, so the overlaps are random. On this dataset, we conducted the following four experiments:

*Update throughput:* In the first experiment, we tested how much new data that the system can incrementally update with a certain underlying RDF store corresponding to each hardware configuration. We simulated the process of data growing by gradually adding more data to the system. We measured the throughput of inserting data (triples/second) until the system crashed or until we reach 1 million triples (whichever happened first).

*Query processor comparison:* In the second experiment, we tested the performance and functionality of TDBoid and RDF-OTG using 8 typical queries with the maximum data sizes that both TDBoid and RDF-OTG could support. The queries are chosen to cover all query patterns and different complexities. Note that each query accesses to the aggregated view which already involves data from multisources and queries 7 and 8 are to show the ability to refer to the original data sources. The list of queries in SPARQL language is given in Appendix B.

*Memory consumption:* In the third experiment, we measured the memory consumption of two system configurations while performing the queries. The experimental application ran the different queries repeatedly and recorded the maximum memory heap that the operating system allocated for it. To evaluate the impact of the data size on memory consumption, the test was conducted on the Nexus 7 Tablet with five datasets with different sizes. Note that the memory consumption is device-independent. We used the same query sets as in the second experiment.

*Scalability:* In the last experiment, we evaluated the scalability of RDF-OTG by measuring the query response time of the above 8 queries with the maximum data sizes that each of the above devices could store.

### 3.4.2 Evaluation Results

Figure 3.4 shows the results of our first experiment, in which we measured the update performance of RDF-OTG and TDBoid when adding increasingly triples to the store. As we can see, in general, the writing throughput of RDF-OTG is roughly twice as high as TDBoid's. This shows the advantage of our optimisation for flash memory compared to the design used in TDBoid, which was originally designed for normal magnetic disks.

In addition, while throughput decreases for larger data sizes in the store, RDF-OTG is able to add more triples to the store for all scenarios with acceptable rates (approx. 200 triples/sec for the HTC Desire and approx. 500 triples/sec for the two more powerful devices), even if nearly one million triples were already in the store. TDBoid on the other hand, is not only slower, but it also cannot cope at all with such data sizes and reaches its upper capacity limit at 100k triples on the HTC Desire, 220k triples on the Galaxy Nexus and around 200k triples on the Nexus 7.

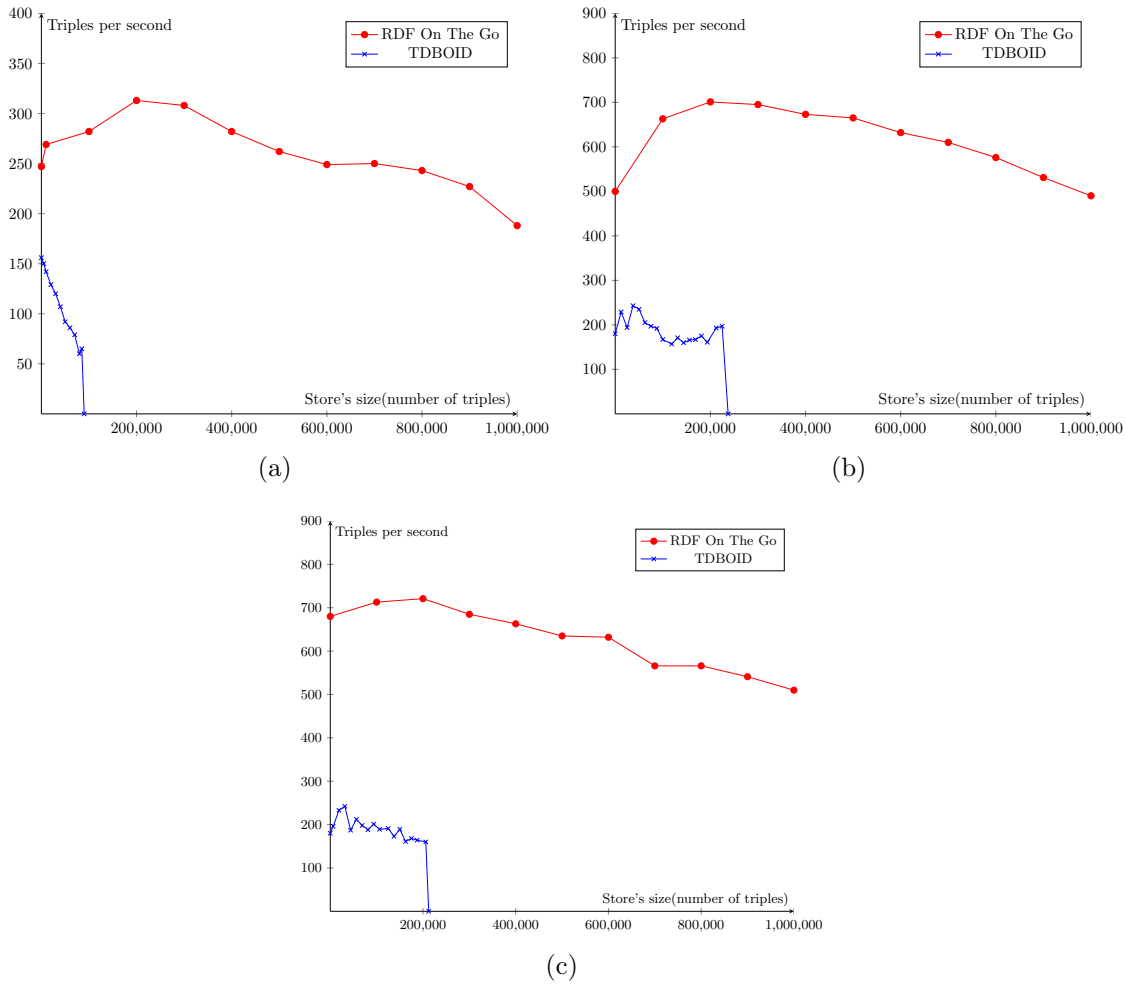


FIGURE 3.4: Update throughputs of RDF-OTG compared to TDBoid on HTC Desire, Samsung Galaxy Nexus and Nexus Tablet 7. (a) Update throughput results on HTC Desire; (b) Update throughput results on Galaxy; (c) Update throughput results on Nexus Tablet 7.

In our second experiment, we measured the performance of evaluating different queries (as discussed earlier) on existing data sets. The results are shown in Figure 3.5. Unfortunately, TDBoid does not return any results for queries Q7 and Q8 because it does not support queries involving named graphs. Therefore, we omitted these two queries from the graph below. In addition, due to the limitation in the number of triples that TDBoid can handle, we had to reduce the number of profiles contained in the test data for each

of the devices: 45 profiles for the HTC Desire, 180 profiles for the Galaxy Nexus and 112 profiles for the Nexus 7.

The results show that the query performance of RDF-OTG is much higher than TDBoid’s for the Galaxy Nexus and Nexus 7. However, for the HTC Desire, the performance is comparable or even worse for RDF-OTG. The reason for this is that we specifically optimised our system for flash memory. However, the HTC Desire uses an external SD card for storing data instead of internal flash memory. This induces a much higher cost on I/O operations on the HTC Desire. Since TDBoid is originally designed for (relatively slow) magnetic disks, it is able to handle this better than RDF-OTG. However, to do so, TDBoid uses a lot of main memory, which explains its restricted scalability.

The results of the third experiment for measuring the memory consumption for querying are presented in Table 3.2. Due to the limited scalability of TDBoid exhibited in the inserting throughput test, the tests with TDBoid could only be executed on data sizes of 100,000 and 200,000 triples. The results of the experiment demonstrate the great improvement in memory footprint optimisation of our system. On the same dataset, RDF-OTG requires only one-third of the memory that TDBoid needs. For instance, RDF-OTG requires 4MB for the 100,000 triple dataset and 8MB for 200,000 triples to perform the queries, while TDBoid requires 11MB and 26MB for the same setup. The efficiency in memory usage also enables RDF-OTG to support much larger datasets. Even with a dataset of 1.5 million triples, the heap size of a system configured with RDF-OTG is lower than 64MB (the JVM maximum heap size of the Nexus 7 tablet).

	100k	200k	500k	1m	1.5m
RDF-OTG	4MB	9MB	17MB	34MB	46MB
TDBOID	11MB	26MB	N/A	N/A	N/A

TABLE 3.2: Memory consumption of mix queries/size of data

Our last experiment evaluated the scalability of our system. Due to the scalability limitations that we found in earlier experiments, we omitted TDBoid in this experiment and focused on RDF-OTG. Table 3.3 shows the query response times of the 8 queries on the three devices. As we can see, our system is able to handle datasets of 1 million triples (900 profiles) on the HTC Desire, and 1.5 million triples (1200 profiles) on the Galaxy Nexus and Nexus 7 without any problems. For simple queries like Query 1 and Query 3, it takes less than 1 second to answer the query on datasets of more than one million triples on all devices. More complicated queries such as queries Q4, Q5, and Q6, take less than 10 seconds, except for Query 5 on the HTC Desire. For this query, RDF-OTG crashes with an out-of-memory error. The HTC’s maximum heap size of 32MB is not enough for RDF-OTG to handle the large number of intermediate results generated for this query from the one million triple dataset.<sup>8</sup> We plan to look into this matter further in the future to solve this problem. For the rest of the queries, it takes 10-25 seconds to

<sup>8</sup>34MB would be required as shown in our third experiment.

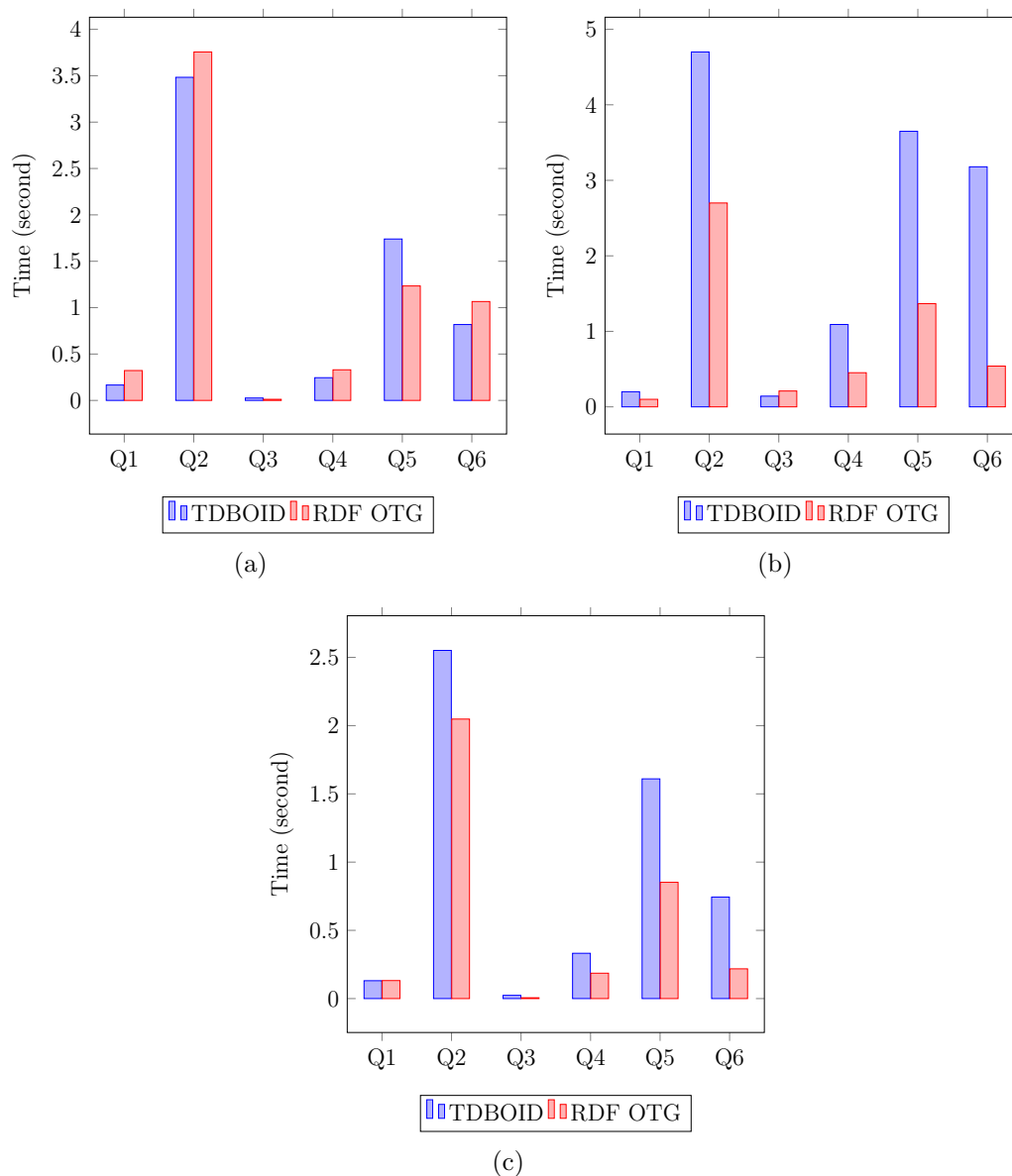


FIGURE 3.5: Query response time of RDF-OTG compared to TDBoid on HTC Desire, Samsung Galaxy Nexus and Nexus Tablet 7. (a) Query response time results on HTC Desire; (b) Query response time results on Galaxy; (c) Update throughput results on Nexus Tablet 7.

answer the query. This is due to the time spent for fetching a big set of output results and is determined by the query, so developers have to be careful to “ask the right queries.”

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q 8
HTC (900 Profiles/1M triples)	0.114	19.035	0.402	3.714	failed	6.093	22.652	24.345
GALAXY (1.2K Profiles/1.5M triples)	0.322	14.705	0.341	3.490	7.858	1.713	16.111	19.223
NEXUS 7 (1.2K Profiles/1.5M triples)	0.113	11.638	0.242	2.458	6.649	1.579	12.769	17.044

TABLE 3.3: Query response time (seconds) on maximum datasets for RDF-OTG

## 3.5 Related Work

Semantic Web and RDF have long been used as a solution for modelling and integrating heterogeneous personal data. Many works have aimed to better allow a user's access to multiple data silos by using Semantic Web technologies to satisfy the requirements of data portability in terms of identification, personal profiles, and friend networks (Bojārs et al. 2008). SemanticLife (Hoang et al. 2006) is one of the early attempts to employ ontologies for modelling personal digital information. Then there is a series of works on the Semantic Desktop, such as the Gnowsis Semantic Desktop (Sauermann 2003) or the Social Semantic Desktop (Decker & Frank 2004) to provide semantic Personal Information Management (PIM) tools. Additionally, other integrated platforms such as Haystack (Quan et al. 2003) and Semex (Dong & Halevy 2005) provide a wide range of tools and functionalities for PIM. However, they all aim at a standard computer environment and do not take into account mobile devices and their specific problems.

Since then, several works have tried to achieve the same functionality on mobile devices. However, the adaptations necessary for the mobile setting proved to be challenging. The first line of work followed the approach of connecting mobile devices to a centralised infrastructure where all processing and storage are delegated to (d'Aquin et al. 2011, David & Euzenat 2010). This line of early work has a lot of security, connection, and performance issues. To address them, there are emerging efforts to ship the processing and storage of personal information on mobile devices. For instance, (Tramp et al. 2011) tries to store personal data retrieved from distributed social networks on the phone. However, most of these works still have certain dependencies and use unsecured data exchanges with intermediate parties.

However, these early works have shown the clear interest of using Semantic Web technologies for integrating personal data on mobile devices, and they also have shown the need for mobile RDF data processing engines. However, these existing works ignore the fact that the existing triple storage technology for normal computers *can not* be directly applied to the mobile setting. For example, the early adoption of Jena to J2ME (Hayun 2009) is micro-Jena<sup>9</sup> which only works on in-memory data on Symbian mobiles. The Android version of Jena, TDBoid, is far better due to newer hardware capabilities, but it has a lot of limitations in respect to performance and scalability as we have shown in our experiments in Section 3.4. SuccinctEdge (Xu et al. 2020) is the recently developed in-memory RDF store targeting small devices. Similarly to our approach, SuccinctEdge aims to minimise memory consumption, thus, it uses *Succinct* data structures (Agarwal et al. 2015) such as bitmaps and wavelet trees to guarantee a low memory footprint for RDF data. SuccinctEdge shows its promising performance on the Raspberry Pi devices which have similar hardware configuration to mobile phones. Unfortunately, SuccinctEdge has not been implemented for mobile phones yet. We believe this paper is the first to

<sup>9</sup>[http://poseidon.ws.dei.polimi.it/ca/?page\\_id=59](http://poseidon.ws.dei.polimi.it/ca/?page_id=59)

systematically investigate and address the issues of security, integration, performance, and scalability of integrating heterogeneous personal information data.

## 3.6 Conclusions

In this paper, we have presented a comprehensive framework for the integration of personal data from heterogeneous data sources, such as different social networks, on mobile devices. Our framework builds upon Linked Data technology to be generic with respect to the supported data types and data requests, offers a plug-in model to be extensible for additional data sources, and relies solely on a user's mobile device, without the need for storing or processing any data on an external, possibly untrusted, server infrastructure. The performance and scalability issues are addressed by our RDF triple store for Android devices, RDF On the Go, which is specifically optimised for mobile devices and flash memory usage. It offers full support for RDF triples and SPARQL queries and is able to handle more than a million triples on typical mobile devices efficiently. Complex queries are supported and can be executed in reasonable time, even for such large data sets but with a very small memory footprint.

## Chapter 4

# Incorporating Blockchain into RDF Store at The Lightweight Edge Devices

The work outlined in this chapter was published in:

**Le-Tuan, A.**, Hingu, D., Hauswirth, M. and Le-Phuoc, D., 2019. “Incorporating blockchain into RDF store at the lightweight edge devices”. In International Conference on Semantic Systems (pp. 369-375). Springer.

## **Abstract**

RDF stores provide a simple abstraction for publishing and querying data, that is becoming a norm in data sharing practice. They also empower the decentralised architecture of data publishing for the Web or IoT-driven systems. Such architecture shares a lot in common with blockchain infrastructure and technology. Therefore, there are emerging interests in marrying RDF stores and blockchain to realise the desirable but speculative benefits of blockchain-powered data sharing. This paper presents the first RDF store with blockchain that enables lightweight edge devices to control the data sharing processes. Our novel approach on the deep integration of the storage design for RDF stores enables the ability to enforce controlling measures on access methods and auditing policies over data elements via smart contracts before they fetch from sources to the consumers. Our experiments show that the prototype system delivers an effective performance for a processing load of 1 billion triples on a small network of lightweight nodes which costs less than a commodity PC.

## 4.1 Introduction

User-generated and sensor data is being widely used by various applications to make them smarter and better. While such applications are using the data for nearly free, such consumer data from both public and private sources is incredibly valuable to corporations, marketers, investors, and individuals. For example, American companies alone are estimated to have spent over \$19 billion in 2018, acquiring and analysing consumer data, according to the Interactive Advertising Bureau <sup>1</sup>. There are recent recurrent questions of how users can take control and make the benefit out of it. It is obvious that to take control of one's data, there must be an ability to accurately account for ownership, and similarly account or keep a record of all transactions, exchanges, and permissions in a secure and tamper-proof manner.

The arrival of blockchain technology gives the promise to get your data out of a corporation's centralised database to store on your own devices or your encrypted storage of your choice at the edges of networks. This will be critical in helping develop transparency and accountability in data sharing as we take ownership of our data. Blockchain provides a number of substantive benefits. It provides a layer of transparency and accountability for data ownership and transactions. It can also minimise the influence of data middlemen in any consumer data transaction while putting the data back in the hands of the consumer. This means users can regain control of who uses our data, when our data is used and our compensation for it.

In parallel, the recently emerging trend of edge computing paradigm for IoT-driven information systems makes it much more feasible to push computation and data management operations closer to the data sources. Being able to store and query data at the edges of networks offers opportunities to improve performance and to reduce network overhead, but also flexibility for the continuous integration of new IoT devices and data sources. This motivates us to build a novel distributed RDF store which leverages blockchain benefits for data publishers at the edges of networks. To the best of our knowledge, our system is the first of this kind. The system design will be presented in [Section 4.2](#), the implementation report will be followed in [Section 4.3](#). Our experimental results in [Section 4.4](#) show that a small cluster of Raspberry Pis (cost less than a commodity workstation) can efficiently handle 1 billion RDF triples of IoT data.

## 4.2 System Overview

To store and to share RDF data on the edge with the guarantee of data ownership and compensation for shared data, we marry a distributed database system and blockchain. Confronting the decentralised edge networks, the distributed database system allows any members to share their data while maintaining their autonomy and independence from

---

<sup>1</sup><https://www.iab.com/news/2018-state-of-data-report/>

centralised servers. A feature of blockchain technology is the smart contract, which allows the data owners to control the access to their data (e.g., who can have the access or how much to pay to gain the access).

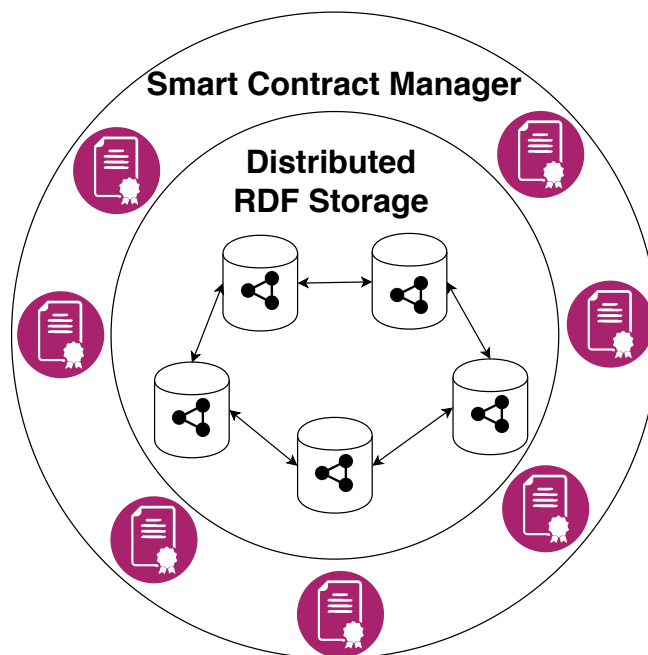


FIGURE 4.1: System overview of an incorporating of distributed RDF storage and Smart contracts.

Our system (Figure 4.1) consists of two subsystems: a *Distributed RDF Storage (DRS)* and a *Smart Contract Manager (SCM)*. The DRS takes responsibility to store RDF data among connected devices in the network physically. Meanwhile, to secure the ownership of the data, the access to the data in the distributed storage is encrypted using smart contracts which are published and managed by the SCM. The basic principle of the system is that when a provider wants to trade his/her data, she/he publishes the data partitions (e.g., a set of sensor readings) associated with smart contracts. A smart contract contains the meta-information of a published data partition, for example, an index key, and a contract that can be used to specify terms like the price scheme and access control policy on data to be fetched to clients.

Considering that to answer a SPARQL query, the SPARQL query processor performs graph pattern matching over RDF datasets. The graph matching operator executes join operations between RDF triples that match the triple query patterns. Therefore, to retrieve the matched data, we organise RDF data in the similar way as RDF4Led (Le-Tuan et al. 2018) does. We store RDF triples in three storage layouts as sorted permutations of triples: SPO (Subject - Predicate - Object), POS and OSP. Each layout is a sorted list and is partitioned into chunks called data blocks. The first triple of each data block and the physical address of the data block are formed an index entry which is kept in the main memory. The triples of the index entries are the keys for searching the data blocks that potentially contain the matched triple of a query pattern. However, instead

of storing the data blocks in the local file system as in RDF4Led, we use distributed file systems to create the DRS subsystem. The distributed file systems provide scalable data distribution and sharding features associated with distributed data structures like DHT (Benet 2014). In such a data structure, each data block that is stored is mapped to a unique identifier. The identifier space is partitioned among the nodes. Each node is responsible for storing all data blocks that are mapped to identifiers in its portion of the space. Hence, the data is distributed and resource consumption is balanced among the edge nodes. Furthermore, instead of keeping the unique access identifiers of data blocks in index entries openly, we allow publishers to encrypt them into smart contracts and keep them in the *Smart Contracts Storage (SCS)* of the SCM (see Figure 4.2). In the Smart Contracts Storage, the smart contracts are kept sorted by the triple, therefore, the corresponding smart contracts that hold the query pattern matched triples can be searched as described in (Le-Tuan et al. 2018). Finally, a data block can be fetched only when the access identifier is revealed, as a consequence, only when the smart contract that holds the identifier of the block is triggered.

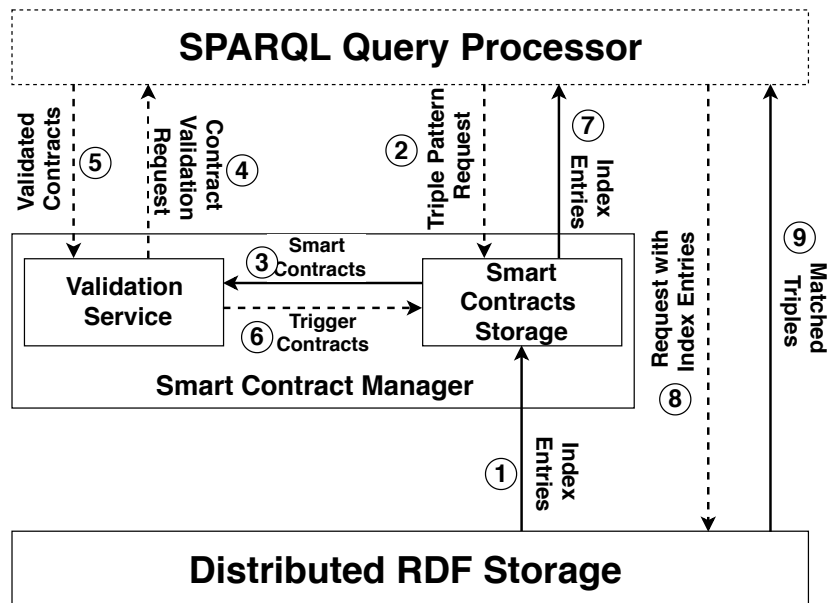


FIGURE 4.2: General procedure

The general workflow of the system is visualised in Figure 4.2. When a client starts sharing her/his RDF dataset, the system indexes the data, partitions the indexes into data chunks, stores the data chunks in the DRS, and sends the index entries to the SCM (1). The SCM encodes the arriving index entries into smart contracts and stores these contracts in the Smart Contracts Storage. From the given triple requests (2), the SCM searches in the SCS for the contracts that hold the accesses to the requested data. Later, these contracts are sent to the *Validation Service* which developed with blockchain technology (3). The Validation Service verifies if the contracts are validated and navigates the validation requests to the SPARQL Query Processor (4). When the contracts are validated (5), the Validation Service triggers these contracts (6) and returns the opened index entries with access identifiers to the SPARQL Query Processor (7). With these index entries and

access identifiers, the SPARQL Query Processor can fetch the requested triples from the DRS for further processes (i.e joining) (8,9).

### 4.3 Implementation and Deployment

To implement the system, we extended the Java code base of RDF4Led (Le-Tuan et al. 2018) that allows RDF data processing tasks (e.g., parsing, indexing) to execute on the edge node. The DRS subsystem is implemented by reengineering the Physical Layer to store the data blocks (which contain RDF molecules) in the p2p file system IPFS (Benet 2014). IPFS is a secure, high-throughput, distributed block storage model with content-addressed hyper-links. It allocates a unique hash for each stored block of data. In IPFS, the data blocks are identified and retrieved by their IPFS hashes. In SCM, the SCS is stacked on top of the Buffer Layer. The smart contract and related features are implemented with Ethereum<sup>2</sup>. However, instead of using an in-memory caching of RDF4Led, the index entries which also keep the address of their corresponding smart contracts are stored in Redis<sup>3</sup>. Redis is a distributed in-memory key-value data that allows data being clustered in the memory of multiple devices. When a device joins the network, it also contributes its computational resources to the whole system. To communicate the Validation Service with the Ethereum’s blockchain network, we use Web3j of Ethereum.

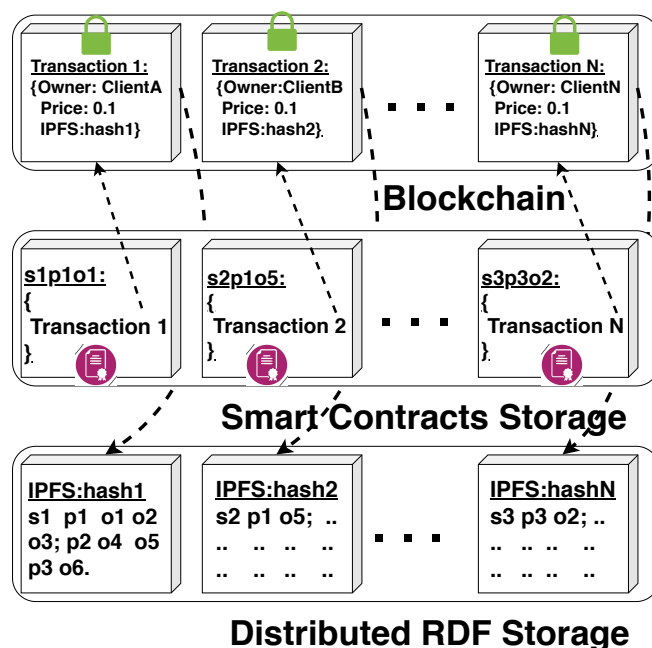


FIGURE 4.3: Physical data organisation

<sup>2</sup><https://www.ethereum.org/>

<sup>3</sup><https://redis.io/>

Figure 4.3 illustrates the physical data organisation of SPO index layout in our system. In this layer, triples are sorted in lexicographical order by S, P, O respectively. In the DRS layer, the sorted list of triples are partitioned, compressed as molecules, and stored in IPFS as byte arrays. The associated IPFS hashes of RDF molecules are packed with the addresses of the smart contracts. A smart contract contains the information of the owner of the RDF molecules and their price scheme. The smart contracts are stored in Ethereum blockchain and are programmed to return the stored IPFS hashes if the transactions are triggered. In the SCS, each entry contains the first triple of each molecule and its smart contract id. These entries are kept in a sorted list of Redis which provides the key-range search that allows index lookup in the same fashion as in RDF4Led.

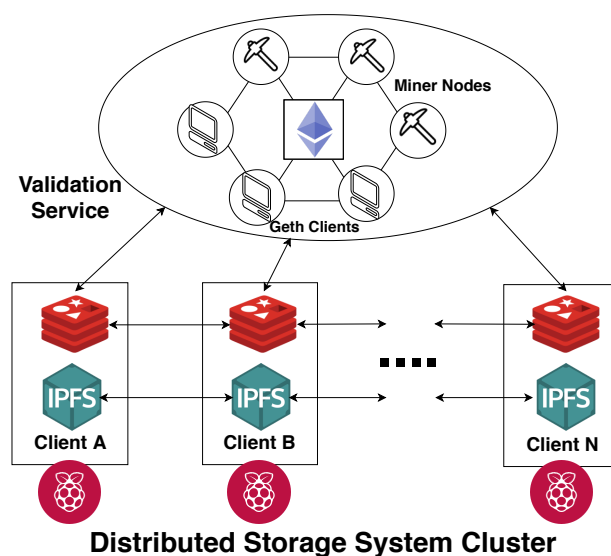


FIGURE 4.4: System deployment schema

Figure 4.4 depicts our system’s deployment strategy. The nodes are installed with IPFS and Redis which make our SCS and DRS layer, respectively. A node’s IPFS and Redis clubs with another node’s IPFS and Redis to form a cluster, respectively, within the network to provide a decentralised distributed data storage. IPFS nodes need to bootstrap using a swarm key of the IPFS cluster to join the cluster and thus can provide additional storage to the DRS layer. A node’s Redis setup needs to be added by Redis Cluster manager to be able to join the Redis Cluster. Thus, every joining node of the system contributes their computation resources. The *Validation Service* Cluster is implemented using Ethereum’s Private Blockchain using Geth, which in turn is configured to use Proof-of-Authority consensus mechanism. It consists of  $(N+1)/2$  Validator nodes to avoid 51% and D-Dos attacks, their task is to mine and commit new blocks of the transaction (SCS’s records). Due to edge nodes, the mining of new blocks is configured only during a transaction is submitted, thus ensuring low computational resource consumption. The remaining nodes are set up as full-node clients, which provide additional storage to the service.

## 4.4 Evaluations

This section presents our evaluation on the deployment as presented in Figure 4.4. We created a network of 15 Raspberry Pi 3 model B (ARMv7 Quad-Core 1.2GHz CPUs, 1GB RAM, 64GB SD card, Raspbian OS), each node costs approximately EUR 50. *Note that the total cost of the whole setup is less than a commodity PC.* The RDF dataset for the evaluation is generated by mapping sensor readings from NOAA <sup>4</sup> dataset to RDF using SSN/SOSA ontology (Haller et al. 2019). The schema of our data set is provided along with the implementation in our Github repository <sup>5</sup>.

We evaluated our system with two experiments. Experiment 1 consists of two tests with two system settings. Firstly, we fixed the cluster size at 10 nodes and measured the average throughput per node when inserting more data. Secondly, we observed the increasing of the throughput when adding more nodes. One billion and 100 million triples dataset was used in the first experiment, respectively. In Experiment 2, we measured the response time for searching the matched triples of single query patterns on a one billion triples dataset.

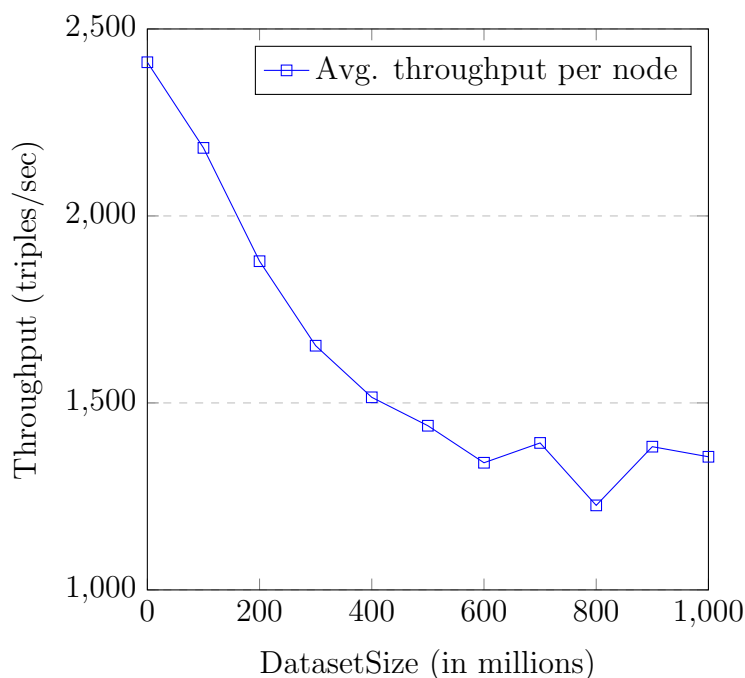


FIGURE 4.5: Acc. Throughput on 10-nodes cluster sizes

Figure 4.5 represents the first setting's result with the accumulated throughput of a fixed number of nodes. As predicted, the throughput gradually decreases when the data stored in the storage increases. After 500 million inserts, we can see a stagnant flow in the graph. The result of the accumulated throughput of a varying number of nodes in the cluster is presented in Figure 4.6. Here, we plot the throughput (triples/seconds) in thousands

<sup>4</sup><https://www.ncdc.noaa.gov/>

<sup>5</sup><https://github.com/anhlt18vn/Semantic2019>

against the number of nodes participating. It appears as the number of processing nodes in the cluster increases, the throughput also increases. After reaching a peak point, the growth has minimised due to the decentralised nature of the system.

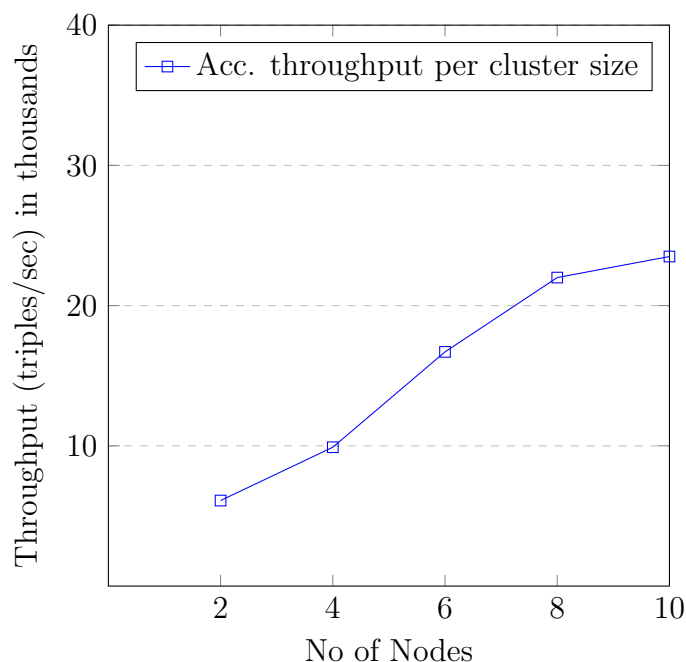


FIGURE 4.6: Acc. Throughput on varying cluster sizes

From both tests, we can observe that the system is competent in handling a large dataset due to its decentralised ecosystem. We have also seen that the system's performance becomes stagnant after attaining peak. It is mainly because the decentralised system adheres to the network latency, which occurs due to its replication and distribution strategies. With the current architecture, the system has proven highly scalable, although it is acknowledged that there has been a trade-off in terms of performance to achieve this nature of the degree of scalability.



FIGURE 4.7: Query response time

[Figure 4.7](#) shows the result of experiment 2. We can see that the system deliver very good performance for the query that needs data from 1-10 blocks on the 1 billion dataset. However, the response time increases linearly to the number of fetched blocks. From our analysis, the delay is mainly contributed by the throughput/delay of processing multiple transactions on Ethereum.

## 4.5 Conclusion and Outlook

This paper presents the first implementation of a novel RDF distributed store with blockchain technology for the decentralised edge network. The experiments proved that the system can be deployed on a network of lightweight edge devices such as Raspberry Pi. On a network of fifteen nodes of Raspberry Pi, the system is able to host a dataset up to a billion RDF triples. The cost for deploying such a system is quite competitive and flexible as a Pi node costs less than EUR 50 and the number of nodes can be elastically increased or decreased at runtime. Next step, we will increase tenfold in the number of nodes to study the scalability and limitations. For the shortcomings shown in [Figure 4.7](#), the processing throughput causing long delays to the queries that use a large number of data blocks can be increased by the new achievement of transaction throughput, e.g. 20k transactions/second in ([Gorenflo et al. 2019](#)).

# Chapter 5

## Autonomous RDF Stream Processing for IoT Edge Devices

The work outlined in this chapter was published in:

Nguyen-Duc, M., **Le-Tuan, A.**, Calbimonte, J.P., Hauswirth, M. and Le-Phuoc, D., 2019, November. Autonomous RDF Stream Processing for IoT Edge Devices. In Joint International Semantic Technology Conference (pp. 304-319). Springer, Cham.

## Abstract

The wide adoption of increasingly cheap and computationally powerful single-board computers has triggered the emergence of new paradigms for collaborative data processing among IoT devices. Motivated by the billions of ARM chips having been shipped as IoT gateways so far, our paper proposes a novel continuous federation approach that uses RDF Stream Processing (RSP) engines as autonomous processing agents. These agents can coordinate their resources to distribute processing pipelines by delegating partial workloads to their peers via subscribing continuous queries. Our empirical study in “cooperative sensing” scenario with resourceful experiments on a cluster of Raspberry Pi nodes shows that the scalability can be significantly improved by adding more autonomous agents to a network of edge devices on demand. The findings open several new interesting follow-up research challenges in enabling semantic interoperability for the edge computing paradigm.

**Contribution:** In this paper, the main contributions of the author of the thesis are the continuous federation approach (Section 5.2), and the design and implementation of Fed4Edge (Section 5.3).

## 5.1 Introduction

Over the last few years, Semantic Web technologies have provided promising solutions for achieving semantic interoperability in the IoT (Internet of Things) domain. Ranging from ontologies for describing streams and devices (Haller et al. 2019, Kaebisch et al. 2019b), to continuous query processors and stream reasoning agents (Dell’Aglia et al. 2017), these efforts constitute important milestones towards the integration of heterogeneous IoT platforms and applications. While these different technologies enable the publication of streams using semantic technologies (e.g., RDF streams), and the querying of streaming data over ontological representations, most of them tend to centralise the processing, relegating interactions among IoT devices simply to data transmission. This approach may be convenient in certain scenarios where the streams, typically time-annotated RDF data, are integrated following a top-down approach, for instance, using cloud-based solutions for RDF Stream Processing (RSP). However, in the context of IoT, decentralised integration paradigms fit better with the distributed nature of autonomous deployment of smart devices (Satyanarayanan 2017). Moreover, moving the computation closer to the edge networks, such as sensor nodes or IoT gateways, will not only create more chances to improve performance and to reduce network overhead/bottlenecks, but also to enable flexible and continuous integration of new IoT devices/data sources (Munir et al. 2017).

Thanks to recent developments in the design of embedded devices, e.g., ARM boards (Smith 2008), single board computers are getting cheaper and smaller while increasing their computational power. For example, a Raspberry computer costs less than 40 EUR and its size is just roughly as big as the size of a credit card. Despite the size, they are powerful enough to run a fully functioning Linux distribution that provides both *operational* and *deployment* advantages. On the one hand, they are both power efficient and cost-effective, while computationally powerful. On the other hand, their small sizes make it easier to embed or bundle them with other IoT devices (e.g., sensors and actuators) as a processing gateway interfacing with outer networks, called *edge devices*.

RDF Stream Processing (RSP) (Sakr et al. 2018) extends the RDF data model, enabling to capture and to process heterogeneous streaming sensor sources under a unified data model. An RSP engine usually supports a continuous query language based on SPARQL, e.g. C-SPARQL (Barbieri et al. 2010) and CQELS-QL (Le-Phuoc et al. 2011). Hence, an edge device equipped with an RSP engine could play the role of an autonomous data processing gateway. Such an autonomous gateway can coordinate the actions with other peers connected to it to execute a data processing pipe in a collaborative fashion. However, to the best of our knowledge, there has not been any in-depth study on how such a decentralised processing paradigm would work with edge devices. In particular, an edge device has 10-100 times less resources than those of a PC counterpart, which is originally the expected execution setting for an RSP engine. Hence, this raises two main questions: how feasible would it be to enable such a paradigm for edge devices,

and how it would affect the performance and scalability. Putting our motivation in the context of 100 billion ARM chips that have been shipped so far (Segars 2017), enabling computational and processing autonomy along with semantic interoperability will have a great impact even for a small fraction of this number of devices (e.g. 0.1% account for 100s millions devices).

To this end, this paper investigates how to realise this edge computing paradigm by extending an RSP engine (i.e., CQELS) as a continuous query federation engine to enable a decentralised computation architecture for edge devices. A prototype engine was implemented to empirically study the performance and scalability aspects on “cooperative sensing” scenarios. Our experiment results on a realistic setup with the biggest network of its kind in Section 5.4 show that our federation engine can considerably scale the processing throughput of a network of edge devices by adding more nodes on demand. We believe this is the largest experiment setup of its kind so far. The main contributions of the paper are summarised below:

1. We propose a novel federation mechanism based on autonomous RSP Engines and distributed continuous queries.
2. We present technical details on how to realise such a federation mechanism by integrating an RSP engine and an RDF Store for edge devices.
3. We carry out an empirical study on performance and scalability on “cooperative sensing” that leads to various quantitative findings and then opens up several interesting follow-up research challenges.

The paper is outlined as follows. The next section presents our approach on continuous federation based on autonomous RSP. Section 5.3 describes the implementation details of our federated RSP engine for edge devices. The setup and results of the experiments is reported in Section 5.4. We summarise related work in Section 5.5 and Section 5.6 concludes the paper.

## 5.2 Continuous Federation with Autonomous RSP

### 5.2.1 Preliminaries: RDF Stream Processing with CQELS-QL

CQELS-QL is a continuous query language for RSP that extends SPARQL 1.1 with sliding windows (Le-Phuoc et al. 2011). As an example, the CQELS-QL query in Listing 5.1 continuously provides “updates for the locations of 10 weather stations which have reported the highest temperatures in the last 5 minutes”. This query then is also used as Query Q3 in the experiments of Section 5.4.

```
1 PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX sosa:<http://www.w3.org/ns/sosa/>
3 PREFIX prov:<http://www.w3.org/ns/prov#>
4 PREFIX wgs84:<http://www.w3.org/2003/01/geo/wgs84_pos>
5
6 SELECT ?sensor ?maxTemp ?lat ?lon
7 SELECT
8 {
9   {
10    SELECT ?sensor (MAX(?temp) as ?maxTemp)
11    STREAM ?streamURI [RANGE 5m ON sosa:resultTime]
12    {
13      ?observation sosa:hasSimpleResult ?temp.
14      ?sensor rdf:type <TempSensor>.
15      ?sensor sosa:madeObservation ?observation.
16    }
17    GROUP BY ?sensor
18  }
19  ?streamURI prov:wasGeneratedBy ?sensor.
20  ?sensor sosa:isHostedBy ?station.
21  ?station wgs84:Point ?loc.
22  ?loc      wgs84:lat   ?lat.
23  ?loc      wgs84:lon   ?lon.
24 }
25 ORDER BY ?maxTemp
26 LIMIT 10
```

LISTING 5.1: Query Q3 - Updates for the locations of 10 weather stations which have reported the highest temperatures in the last 5 minutes

In the original centralised setting, the above query can be subscribed to a CQELS engine installed in one *processing node*. Stream data in different RDF formats (e.g., JSON-LD or Turtle) can be provided to it from data acquisition nodes, called *streaming nodes*. These streaming nodes collect data from sensors of weather stations that can be geographically distributed in different locations. In practice, an edge device can host both a CQELS node and a streaming node, but we can assume they communicate via an internal process. As soon as the data is collected, the sensor data is pushed to the CQELS engine via a streaming protocol such as Websocket or MQTT. The incoming data continuously triggers the processing pipeline compiled from Query Q3. Consequently, the computing node that hosts this CQELS engine needs to have enough resources (bandwidth, CPU, and memory) to deal with the workload regardless of how many stream nodes exist in the network. Hence, if the CQELS engine is only hosted on an edge device, the physical limit of its hardware quickly becomes a bottleneck as shown in Section 5.4. To create a more scalable

processing system, we need to decentralise the processing pipelines of similar queries to a network of edge devices connected to these stream nodes. The following two sections describe our approach to enable this type of network.

## 5.2.2 Dynamic subscription & discovery for autonomous RSP engines

To enable a CQELS engine to work in a decentralised fashion, it would require the capability to operate as an autonomous agent which can collaborate with other peers to execute a distributed processing pipeline specified in CQELS-QL. An autonomous CQELS node can dynamically join a network of existing peers by subscribing itself to an existing node in the network, called a parent node, and it then notifies the parent node about the query service and the streaming service it can provide to the network. These services can be semantically described by using the vocabularies provided by VoCaLS (Tommasini et al. 2018). For instance, VoCaLS allows describing the URIs of the streams and their related metadata (e.g., sensors that generate the streams), which are used in the query patterns of the query Q3. Hence, a subscription can be done by sending a RDF-based message via a REST API or WebSocket channel. Listing 5.2 illustrates a snippet of a subscription message in RDF Turtle that is used in our experiments in Section 5.4.

```
1 <>          rdf:type    vocals:StreamDescriptor;
2             rdf:type    vsd:CatalogService;
3             dcat:dataset :NOAAWeather.
4 :NOAAWeather rdf:type    vocals:RDFStream;
5             prov:wasGeneratedBy :TemperatureSensor;
6             vocals:hasEndpoint :NOAAWeatherEndpoint;
7             dct:title    "Weather stream From Berlin".
8 :NOAAWeatherEndpoint rdf:type    vocals:StreamEndpoint;
9             dct:format    frmt:JSON-LD;
10            dcat:accessURL "ws://192.168.178.5/noaa/berlin".
```

LISTING 5.2: Example of subscription message in RDF

Based on the semantic description provided by the subscribed nodes, the parent node can carry the stream discovery patterns which use a variable in the stream pattern, as shown in line 4 of the query Q3. The variable *?streamURI* then can be matched with other metadata as shown in line 8. In this example, it is used to link to the sensors that generated this stream. Recursively, the subscription process can propagate the stream information upstream hierarchically, and vice versa, the discovery process can be recursively delegated to downstream nodes via sub-queries in CQELS-QL.

To this end, when an autonomous CQELS joins a network, it makes itself and its connected nodes discoverable and queryable to other nodes of the network. Moreover, each

node can share its processing resources by executing a CQELS query on it. This will help us treat the query similar to query Q3 as a query on a sensor network whereby sensor nodes and network gateways collaborate as a single system to answer the query of this kind. Next, we will discuss how to federate such queries in the “cooperative sensing” scenario whereby such a network of autonomous CQELS processing nodes will coordinate each other to answer a CQELS-QL request in a decentralised fashion.

### 5.2.3 Continuous Query Federation Mechanism

With the support of the above subscription and discovery operations, a stream processing pipeline written in CQELS-QL can be deployed across several sites distributed in different locations: e.g., weather stations provide environmental sensory streams in various locations on earth. Each autonomous CQELS node gives access to data streams fed from the streaming nodes connecting to it. Such stream nodes can ingest a range of sensors, such as air temperature, humidity, and carbon monoxide. When the stream data arrives, this CQELS node can partially process the data at its processing site and then forward the results as mapping or RDF stream elements to its parent node.

In this context, when a query is subscribed to the top-most node, called the root node, it will divide this query into subquery fragments and deploy at one or more sites via its subscribed nodes. A query fragment consists of one or more operators, and each fragment of the same query can be deployed on different processing nodes. Recursively, a subquery delegated to a node can be federated to its subscribed nodes. All participant nodes of a processing pipeline can synchronise their processing timeline via a timing stream that is propagated from the root node. The execution process of subquery fragments can use resources, i.e., CPU, memory, disk space, and network bandwidth of participant nodes to process incoming RDF graphs or sets of solution mappings and generate output RDF graphs/sets of solution mappings. Output streams may be further processed by fragments of the same query, until the results are sent to the query issuer at the root node. For example, the subquery of the query Q3 in Listing 5.3 can be sent down to the nodes closer to the streaming nodes, then the result will be recursively sent to the upper nodes to carry out the partial top-k queries in lines 10 and 11 until it reaches the root node to carry out the final computation steps to return the expected results.

```
1 PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX sosa:<http://www.w3.org/ns/sosa/>
3 PREFIX prov:<http://www.w3.org/ns/prov#>
4 PREFIX wgs84:<http://www.w3.org/2003/01/geo/wgs84_pos>
5
6 SELECT ?sensor (MAX(?temp) as ?maxTemp)
7 WHERE
8 {
9   STREAM ?streamURI [RANGE 5m ON sosa:resultTime]
10  {
11    ?observation sosa:hasSimpleResult ?temp.
12    ?sensor      rdf:type                <TempSensor>.
13    ?sensor      made:Observation        ?observation.
14  }
15  ?streamURI   prov:wasGeneratedBy ?sensor.
16 }
17 GROUP BY ?sensor
```

LISTING 5.3: Example of the subquery of Q3

This federation process can be carried out dynamically thanks to the dynamic subscription and discovery capability above. Moreover, the processing topology of such as processing pipelines in our experiment scenarios of Section 5.4 can be dynamically configured by changing where and how participant nodes subscribed themselves to the processing networks. For example, we carried out five different federation topologies in Section 5.4. The biggest advantage of this federation mechanism is the ability to dynamically push some processing operations closer to the streaming nodes to alleviate the network and processing bottlenecks which often happen at edge devices. Moreover, this mechanism can significantly improve the processing throughput by adding more processing nodes on demand as shown in the experiments in Section 5.4.

### 5.3 Design and Implementation

To enable the cooperative federation of RSP engines on edge devices, we built a decentralised version of the CQELS engine, called Fed4Edge. Fed4Edge was implemented by extending the algorithms and Java code base of the original open sourced version of CQELS (Le-Phuoc et al. 2011). Thanks to the platform-agnostic design of its execution framework (Le-Phuoc et al. 2015), the core components are abstract enough to be seamlessly integrated with different RDF libraries to port to different hardware platforms. To tailor the RDF-based data processing operations on edge devices (e.g., ARM CPU, Flash-storage, and likes), we integrated the core components of CQELS with the counterparts

of RDF4Led (Le-Tuan et al. 2018), a RISC style RDF engine for lightweight edge devices. The Fed4Edge system will be open-sourced at <https://github.com/cqels/Fed4Edge>.

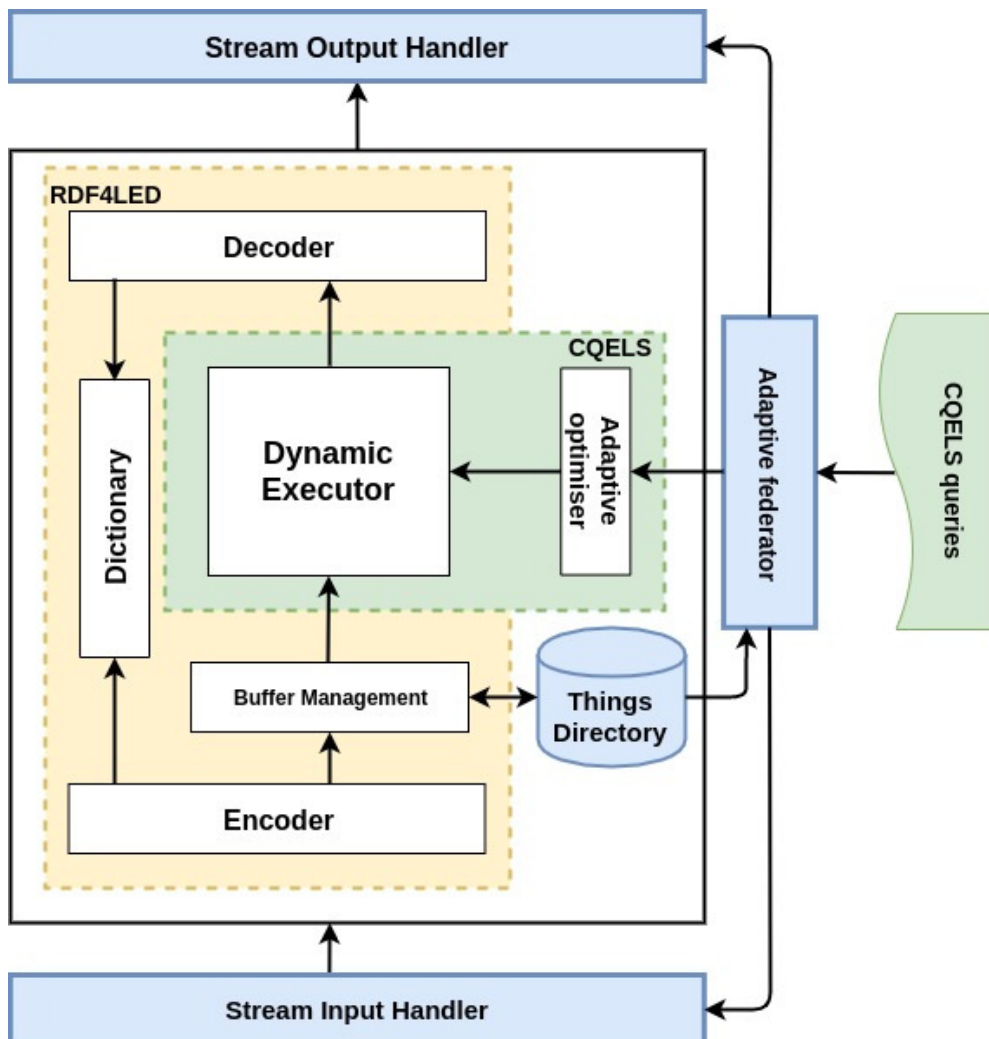


FIGURE 5.1: Overview architecture of Fed4Edge

The architectural overview of the system is depicted in Figure 1. The core components of CQELS and RDF4Led such as the *Dictionary*, *Encoder*, *Decoder*, *Dynamic Execution*, *Adaptive Query Optimiser* and *Buffer Manager* are reused in Fed4Edge implementation. The extension plugins of them such as *Adaptive Federator*, *Thing Directory*, *Stream Input Handler*, and *Stream Output Handler* are built to facilitate the federation mechanism proposed in Section 5.2. The technical details of these components are discussed next.

CQELS and RDF4Led share similar RDF data processing flows due to the fact that both systems apply the same RDF data encoding approach, which normalises RDF nodes into a fixed-size integer. By encoding the RDF nodes, most of the operators on RDF data can be executed on a smaller data structure rather than large variable-length strings. This approach is commonly used in many RDF data processors to reduce memory footprint, I/O time, and improve cache efficiency. The platform-agnostic design of CQELS allows the size of the encoded node to be tuned to adapt to a targeted platform without changing

the implementation of other core components. Therefore, the Encoder, Decoder, and Dictionary of RDF4Led can be easily integrated with CQELS for the RDF normalisation tasks. After receiving RDF data from RDF stream subscriptions via the Stream Input Handler, the data is encoded by the Encoder. The encoded RDF triples are then sent to the Buffer Manager for further processing. The Decoder waits for the output from the Dynamic Executor, transforms the encoded nodes back to a lexical representation before sending them to the Stream Output Handler. The Encoder and Decoder share the Dictionary for encoding and decoding. Instead of using a 64-bit integer for encoding RDF nodes as in the original version of CQELS, the Dictionary of RDF4Led uses 32-bit integers, which entails less memory footprint for cached data. Therefore, backed by RDF4Led, Fed4Edge can process 30 million triples with only 80MB of memory (Le-Tuan et al. 2018) on ARM computing architecture.

The Buffer Manager is responsible for managing the buffered data of windows and then feeding the data to the Dynamic Executor. Furthermore, the Buffer Manager also manages cached data for querying and writing the static data in the Thing Directory. Stream data is evicted from the buffer by the data invalidating policy defined by the window operators (Le-Phuoc et al. 2011, Le-Phuoc 2017). Meanwhile, the flash-aware updating algorithms of RDF4Led are reused to achieve fast updating for static data (Le-Tuan et al. 2018).

The Dynamic Executor employs a routing-based query execution algorithm that provides dynamic execution strategies in each node (Le-Phuoc 2017, Le-Phuoc & Hauswirth 2018). During the lifetime of a continuous query, the query plan can be changed by redirecting the data flow on the routing network. The Adaptive Optimiser continuously adjusts the efficient query plan according to the data distribution in each execution step (Le-Phuoc et al. 2011, Le-Tuan et al. 2018). RDF4Led and CQELS employ a similar query execution paradigm. While CQELS uses routing-based query execution algorithms, RDF4Led executes SPARQL with a one-tuple-at-a-time policy. Therefore, the same cost model of the Adaptive Optimiser can be applied when calculating the best plan for a query that has static data patterns. The Buffer Manager treats the buffer for join results of the static patterns as a window, and depending on the available memory, it will apply the *fresh update* or *incremental update* policy.

The Adaptive Federator acts as the query rewriter, which adaptively divides the input query into subqueries. For the implementation used in our experiments in Section 5.4, the rewriter will push down operators as close to the streaming nodes as possible by following the predicate pushdown practice in common logical optimisation algorithms. The Thing Directory stores the metadata subscribed by the other Fed4Edge engines (cf. Section 5.2) in the default graph. Similar to (Dell’Aglío et al. 2017), such metadata allows endpoint services of the Fed4Edge engines to be discovered via the Adaptive Federator. When the Adaptive Federator sends out a subquery, it notifies the Stream Input Handler to subscribe and listens to the results returning from the subquery. On the other hand, the

Stream Output Handler sends out the subqueries to other nodes or sends back the results to the requester.

## 5.4 Evaluation and Analysis

### 5.4.1 Evaluation Setup

#### Datasets and Queries:

To prepare the RDF Stream dataset for evaluation, we used the SSN/SOSA ontology Haller et al. (2019) to map sensor readings of the NCDC Integrated Surface Database (ISD) dataset<sup>1</sup> to RDF. The ISD dataset is one of the most prominent weather datasets, which contains weather observation data from 1901 to the present, from nearly 20K stations over the world. A weather reading of a station produces an observation that covers measurements of temperature, wind speed, wind gusts, etc. depending on the type of the sensor equipped on that station. Each observation needs approximately 87 RDF triples to map its values and attributes to the schema illustrated in Figure 5.2. The data from different weather stations was split to multiple devices which acted as streaming nodes (i.e., the white nodes in Figure 5.4). Each streaming node hosts a WebSocket server which manages WebSocket stream endpoints. The data is read from CSV files in local storage, then mapped to the RDF data schema in Figure 5.2 before streaming out.

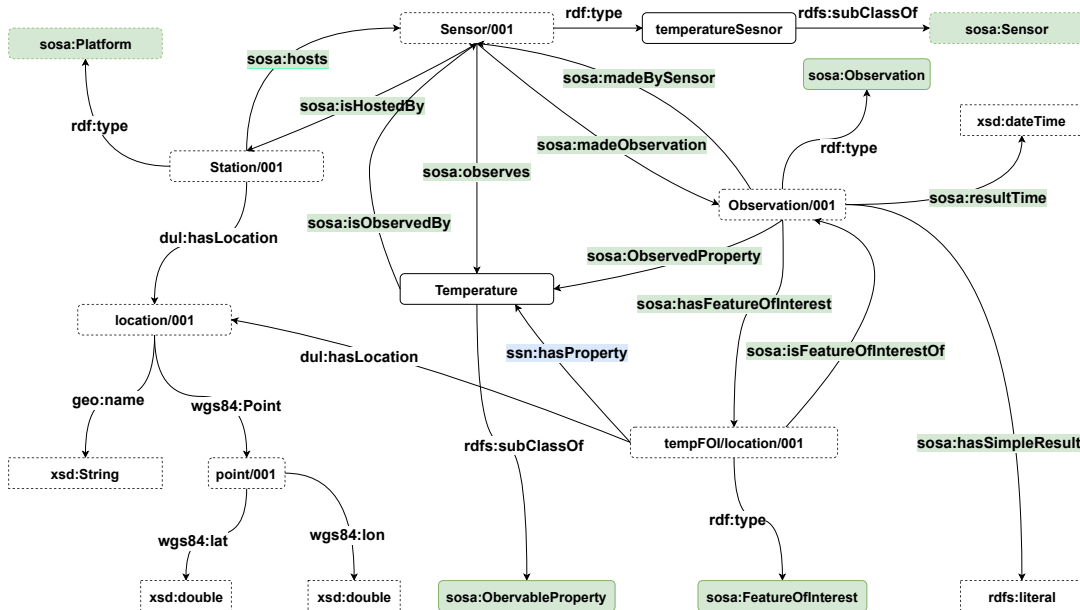


FIGURE 5.2: RDF schema for NCDC weather data

<sup>1</sup><https://www.ncdc.noaa.gov/>

We designed the following queries to show the advantages of cooperative federation for querying streaming data over a network of edge devices. Listings 5.4 and 5.5 respectively present the queries Q1 and Q2 that are used for measuring the improvement of the streaming throughput in simple federation cases. Q1 will return the updated temperature and the corresponding location and Q2 will answer the location where the latest temperature is higher than 30 degrees. The subquery of Q1 and Q2 contains only triple patterns for querying streaming data. With these simple join patterns, the behaviour of the system is mostly influenced by the behaviour of the network. The filter at line 7 of Q2 will reduce the number of intermediate results sent from the lower node, and therefore, it can highlight the benefit of pushing down processing operators closer to the data sources.

For the queries that can show the collaborative behaviour of the participant edge nodes, we used the queries Q3 (as described in the example of Section 5.2) and query Q4 in Listing 5.6. Query Q3 has aggregation and top-k operators and query Q4 includes a complex join across windows.

```
1 PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX sosa:<http://www.w3.org/ns/sosa/>
3 PREFIX prov:<http://www.w3.org/ns/sosa/>
4 PREFIX wgs84:<http://www.w3.org/ns/sosa/>
5
6 SELECT ?temp ?lat ?lon ?resultTime
7 WHERE
8 {
9   STREAM ?streamURI [LATEST ON sosa:resultTime ]
10  {
11    ?obs sosa:hasSimpleResult ?temp.
12    ?obs sosa:resultTime ?resultTime.
13    ?sensor rdf:type iot:TempSensor.
14    ?sensor sosa:madeObservation ?obs.
15  }
16  ?streamURI prov:wasGeneratedBy ?sensor.
17  ?sensor sosa:isHostedBy ?station.
18  ?station wgs84:Point ?loc.
19  ?loc wgs84:lat ?lat
20  ?loc wgs84:lon ?lon.
21 }
```

LISTING 5.4: Query Q1 - Updated the latest temperature and the location

```
1 PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX sosa:<http://www.w3.org/ns/sosa/>
3 PREFIX prov:<http://www.w3.org/ns/prov#>
4 PREFIX wgs84:<http://www.w3.org/2003/01/geo/wgs84_pos>
5
6 SELECT ?lat ?lon
7 WHERE
8 {
9   STREAM ?streamURI [LATEST ON sosa:resultTime]
10  {
11    ?obs sosa:hasSimpleResult ?temp.
12    ?obs sosa:resultTime ?resultTime.
13    ?sensor rdf:type iot:TempSensor.
14    ?sensor made:Observation ?obs.
15    FILTER (?temp > 30)
16  }
17  ?streamURI prov:wasGeneratedBy ?sensor.
18  ?sensor sosa:isHostedBy ?station.
19  ?station wgs84:Point ?loc.
20  ?loc wgs84:lat ?lat
21  ?loc wgs84:lon ?lon.
```

LISTING 5.5: Query Q2 - Return the location where the latest temperature is higher than 30 degree.

```
1 PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX sosa:<http://www.w3.org/ns/sosa/>
3 PREFIX ssn:<http://www.w3.org/ns/ssn/>
4 PREFIX prov:<http://www.w3.org/ns/prov#>
5 PREFIX wgs84:<http://www.w3.org/2003/01/geo/wgs84_pos>
6
7 SELECT ?city ?temp ?windspeed
8 WHERE
9 {
10     STREAM ?streamURI [RANGE 5m ON sosa:resultTime]
11     {
12         ?obs1 sosa:hasSimpleResult ?temp.
13         ?obs1 sosa:resultTime ?resultTime.
14         ?obs1 sosa:hasFeatureOfInterest ?foi1.
15         ?foi1 ssn:hasProperty iot:Temperature.
16         ?foi1 dul:hasLocation ?loc.
17         FILTER (?temp > 30)
18     }
19     STREAM ?streamURI [RANGE 5m ON sosa:resultTime]
20     {
21         ?obs1 sosa:hasSimpleResult ?windSpeed.
22         ?obs1 sosa:resultTime ?resultTime.
23         ?obs1 sosa:hasFeatureOfInterest ?foi1.
24         ?foi1 ssn:hasProperty iot:TempSensor.
25         ?foi1 dul:hasLocation ?loc.
26         FILTER (?windspeed > 15)
27     }
28     ?streamURI prov:wasGeneratedBy ?sensor.
29     ?sensor sosa:isHostedBy ?station.
30     ?station wgs84:Point ?loc.
31     ?loc wgs84:city ?city
32 }
```

LISTING 5.6: Query Q4: Return the city where the temperature is higher than 30 degree and the wind speed is higher than 15km in the last 5 minutes.

### Hardware & Software:

The hardware for the experiment is a cluster of 85 Raspberry Pi model B nodes, each one is equipped with: Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM, and 100 Mbps Ethernet. All nodes connect to five TP-LINK JetStream T2500-28TC switches. Each has twenty-four 100 Mbps Ethernet ports and four 1000Mbps uplinks shown in

Figure 5.3. As to the switching capacity, T2500-28TC has a nonblocking aggregated bandwidth of 12.8Gbps. Four switches for connecting streaming nodes will be connected to the fifth one via the 1000Mps links. This fifth switch is used to connect CQELS processing nodes. Every node uses Raspbian Jessie as the operating system and OpenJDK 1.7 for ARM as the JVM. We set 512MB as the maximum heap size for the Fed4Edge engine.

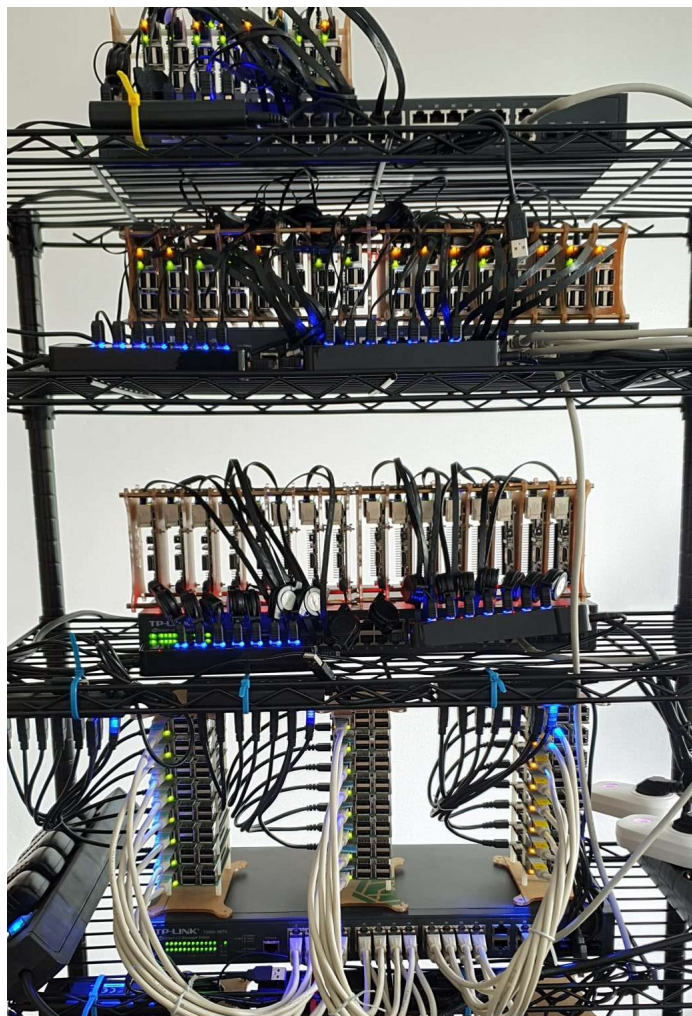


FIGURE 5.3: The evaluation cluster of 85 Raspberry PI nodes

## 5.4.2 Experiments

*Baseline Calibration (Exp1):* In this experiment, we calibrated the maximum processing capability of a processing node as the baseline for the following federation experiment. We increased the number of stream nodes to observe the bottleneck phenomenon whereby increasing more streaming nodes decreases the processing throughput of the network. Each streaming node will stream out the recorded data as its maximum capacity. We will use Query 1 and its two variants as the testing queries. These two variants are made by reducing the four triple patterns into 1 and 2 patterns, respectively. The throughput is

measured by using a timing stream whereby each streaming node will send timing triples indicating when each of them starts and finishes sending their data. In each test, we will equally split 500k-1M observations among streaming nodes and record how much time to process these observations to calculate the average throughput. Note that we separated the streaming and processing processes in different physical devices to avoid performance and bandwidth interference which might have an impact on our measurements.

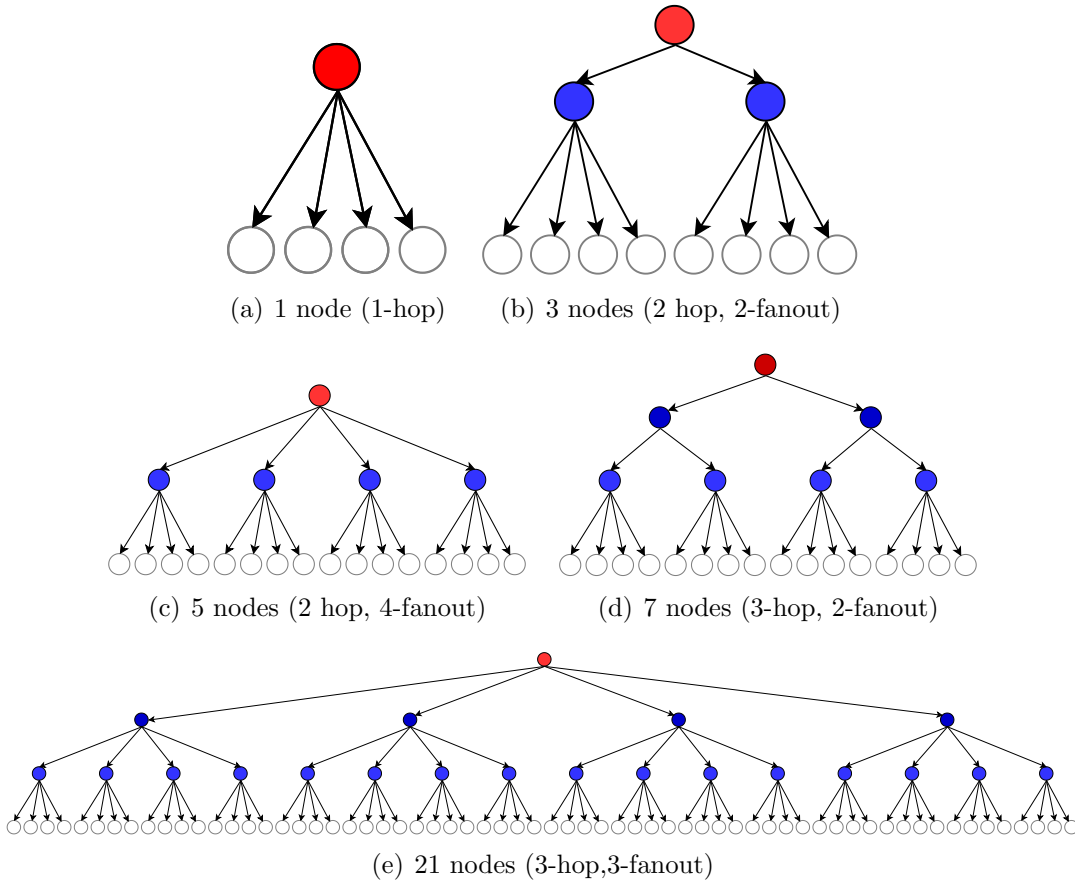


FIGURE 5.4: Federation topologies

#### *Fan-out Federation (Exp2):*

To test the possibility of increasing the processing throughput by increasing more edge nodes as autonomous agents adding to the network, we carried out the tests on five topologies as shown in Figure 5.4. The first topology (1-hop) in Figure 5.4(a) was the configuration that gave the peak throughput in Exp1. Let  $k$  be the number of hops the data has travel to reach to the final destination, we will increase  $k$  to add more intermediate nodes to this topology to create new topologies. As a result, we can recursively add  $n$  nodes to the root node ( $k=2$ , namely 2-hop) and then  $n$  nodes to the root node's children nodes ( $k = 3$ , namely 3-hop) whereby  $n$  is called the fanout factor (denoted as  $n$ -fanout). Then, we have  $\sum_{i=0}^{k-1} n^i$  as the number of nodes of a topology with  $k$  hops and fanout factor  $n$ . We choose  $n = 2$  and  $n = 4$  (corresponding to the number of streaming nodes at the maximum throughput reported in *Exp1* below), thus, we have four new topologies

with 3, 5, 7 and 21 processing nodes in Figure 5.4(b), Figure 5.4(c), Figure 5.4(d), and Figure 5.4(e). In each processing topology, the lowest processing nodes are connected to 4 streaming nodes. We will record the throughput and delay for processing three queries (Q1, Q2, Q3, and Q4) on these five topologies in a similar fashion to Exp1.

### 5.4.3 Results and Discussions

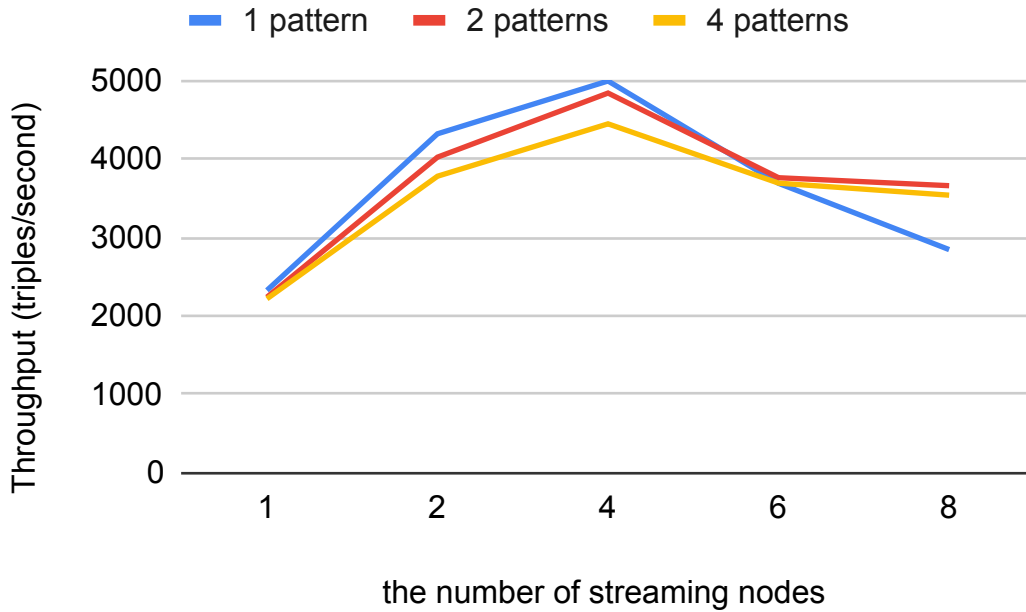


FIGURE 5.5: Baseline calibration

Figure 5.5 reports the results of the experiment Exp1. The maximum processing throughput for three variants of Query 1 on one single edge device is from 4200-5000 triples corresponding to 4 streaming nodes. It is interesting that increasing the number of streaming nodes more than 4 will gradually decrease the overall processing throughput. The results are consistent with different complexities of the variants of Query 1. We observed that the CPU usages were around 60-70% and the memory consumption were around 270-300MB in all tests. Therefore, we can conclude that the bottleneck was caused by the bandwidth limitations. We also carried out a similar test with Q1 on a PC (Intel Core i7 i7-7700K, 4GHz, 1Gb Ethernet, and 16GB RAM) as the root node which has more than 10 times of processing power, memory and network bandwidth than those of a Raspberry Pi model B. As we expected, the PC's maximum throughput is approximately 36k triples/second, around 8-10 times the one with a Raspberry Pi. Note that this PC costs more than the price of 40 Raspberry Pi nodes.

Figure 5.6(a) shows the results of throughput improvement via federating the processing workload on other intermediate nodes in the four proposed topologies. The results show that adding more nodes will increase the processing throughput in general. Most queries

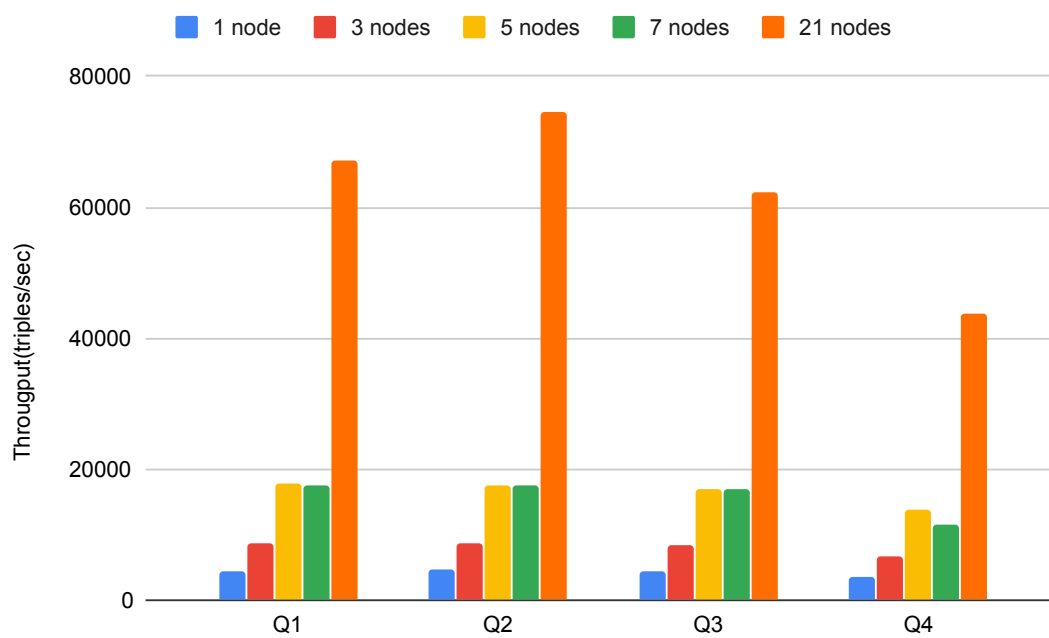
have their processing throughput consistently boosted up as a considerable amount of processing loads were done at the intermediate nodes. However, the increase is not consistently correlated with the total number of processing nodes. The topology with 5 nodes in Figure 5.4(d) gives a slightly higher throughput than that of the topology with 7 nodes in Figure 5.4(c). This can be explained by the fact that both topologies have 4 processing nodes at the lowest level (called leaf processing nodes, i.e., connecting to streaming nodes), but the data in the latter topology has to travel 1 more hop in comparison with the former. Due to our pushing down rewriting strategy presented in Section 5.3, these two upper blue nodes in Figure 5.4(c) did not significantly contribute to the overall throughput but on the other hand cause more communication overhead.

Look closer to the reported figures, we see a high correlation between the number of leaf processing nodes, i.e.,  $n^{k-1}$ , and the processing throughput in all topologies. This shows that our proposed approach is able to linearly scale a network of IoT devices by adding more devices on demand. In particular, a network of 21 Raspberry Pi nodes can collaboratively process up to 74k triples/second or equivalent to roughly 8500 sensor observations/second that are streamed from the other 64 streaming nodes. Hence, the above 20K weather stations across the globe of NCDC can be queried via such a network with the update rate 20-30 observations per minute, which are much faster than the highest update rate currently supported by NCDC<sup>2</sup>, i.e. ASOS 1-minute data. Moreover, the processing capacity of this network is twice more than that of the PC, but it only costs roughly half of the PC. Regarding the energy consumption, each Raspberry Pi only consumes around 2W in comparison of 240W of the above PC.

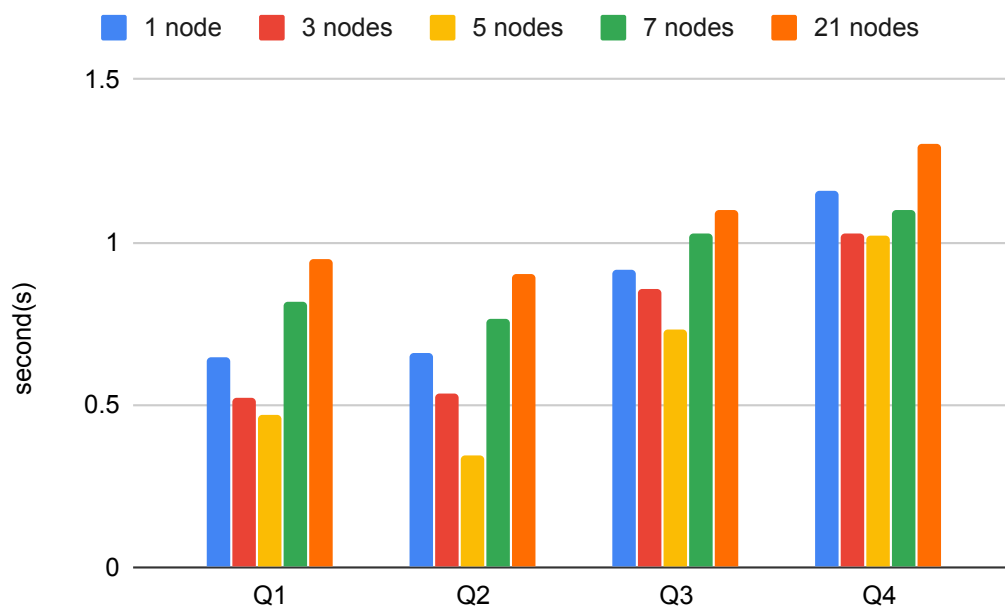
Figure 5.6(b) reports the average time for each observation to travel through a processing pipeline specified by each query with different topologies, i.e., the average processing time. It shows that adding more intermediate nodes for queries Q1 and Q2 can lower the average processing time as it can reduce the queuing time at some nodes. That means communication time might be a dominant factor for the delay in these processing pipelines. In queries Q3 and Q4, we witness a consistent increase in processing time with respect to the number of hops, which explains the nature of query Q3 and Q4 that needs more coordination among nodes. However, it is interesting that increasing 1 hop in the organising network topology just adds 10-15% delay while the maximum throughput gain is linear to  $n^{k-1}$ .

---

<sup>2</sup><https://www.ncdc.noaa.gov/data-access/land-based-station-data>



(a) Processing throughput



(b) Average processing time

FIGURE 5.6: Federation topologies

### 5.4.4 Follow-up Challenges

We observed the CPU, memory consumption and bandwidth in our experiments. It is interesting that all tests used 60-70% of CPU (across 4 cores), 25-30% of physical memory, and 20-40% of Ethernet bandwidth (i.e., 100Mbps). Our reported performance figures show that edge devices have enough resources to enable semantic interoperability for the edge computing paradigm. From our analyses of hardware and software libraries, the most potential suspects for the processing bottleneck are related to the communication among the nodes. Hence, there is a lot of room to make our approach much more efficient and scalable. In this context, to help 100 billions and more edge devices to reach their full potential, we outline some interesting research challenges in following.

The first challenge is how to address the multiple optimisation problems that such a federated processing pipeline entails. The first one is how to optimise an RSP engine for edge devices which have distinctive processing and I/O behaviours from those of PC/workstations due to their own design philosophies. The second challenge is about how to find the optimal operator placements for dynamic execution settings. The subsequent challenge is how to define cost models which are no longer limited to processing time/throughput, but need to cover several cost metrics such as bandwidth, power consumption, and robustness.

Looking beyond database-oriented optimisation goals, another relevant research challenge would be how to model socioeconomic aspects as the control or optimisation scheme for such a cooperative system. In particular, autonomous RSP processing nodes can be operated by different stakeholders which have different utility functions dictating when and how to join a network and to share data and resources. To this end, the coordination strategies become related to the game theory which inspired some relevant proposals in both stream processing and Web communities. For instance, ([Balazinska et al. 2004](#)) proposed a contract-based coordination scheme based on mechanism design, a field in economics and game theory that designs economic mechanisms or incentives toward desired objectives. Similarly, ([Grubenmann et al. 2018](#)) also proposed to use mechanism design for establishing an incentive-driven coordination strategy among SPARQL endpoints. Inspired by this line of work, we also proposed an architecture for in-cooperating blockchain with RDF4Led ([Le-Tuan et al. 2019](#)) to pave the way to incorporate with such incentive and contract-based coordination strategies.

Regarding the cooperation and negotiation among RSP autonomous agents, a potential research challenge is the study and exploration of protocols and strategies that follow the multiagent system paradigm. Although early work on the topic ([Tommasini et al. 2019](#)) pointed at potential opportunities in this area, several aspects have not been studied yet. These include the usage of individual contextual knowledge for local decision making (potentially through reasoning) and for a resource-optimised distribution of tasks among a set of competing/associated nodes. The dynamics of these federated processing networks

would need to adapt to changing conditions of load, membership, throughput, and other criteria, with emerging behaviour patterns on the sensing and processing nodes.

## 5.5 Related work

Semantic interoperability in the IoT domain has gained considerable attention both in the academic and industrial spheres. Beyond syntactic standards such as SensorML, semantically rich ontologies such as SSN-O/SOSA (Haller et al. 2019) have shown a significant impact in different IoT projects and solutions, such as OpenIoT (Soldatos et al. 2015), SymbIoTe (Soursos et al. 2016), or BigIoT (Bröring et al. 2018). Other related vocabularies, such as the ThingsDescription ontology, have also recently gained support from different IoT vendors, aiming at consolidating it as a backbone representation model for generic IoT devices and services. Regarding the representation of data streams themselves, the VoCaLS vocabulary (Tommasini et al. 2018) has been designed as a means for the publication, consumption, and shared processing of streams. Although these ontology resources provide different and complementary ways to represent IoT and streaming data, they require the necessary infrastructure and software components (or agents) able to *interpret* the stream metadata, and apply coordination/cooperation mechanisms for federated/decentralised processing, as shown in this paper.

The processing of continuous streaming data, structured according to Semantic Web standards has been studied in the last decade, generally within the fields of RDF Stream processing (RSP) and Stream Reasoning (Dell’Aglio et al. 2017). A number of RSP engines have been developed in this period, focusing on different aspects including incremental reasoning, continuous querying, and complex event processing, among others (Barbieri et al. 2010, Le-Phuoc et al. 2011, Calbimonte et al. 2010). However, most of these RDF stream processors lack the capability of interconnecting with each other, or to establish cooperation patterns among them. The coordination among RDF stream processing nodes is sometimes delegated to a generic cloud-based stream processing platform such as Apache Storm (e.g. (Le-Phuoc et al. 2013)) or Apache Spark (e.g. Ren & Curé (2017)). In contrast, in this paper, we investigate a more decentralised environment whereby participant nodes can be distributed across different organisations. Moreover, the hardware capabilities of such processing nodes are different from the cloud-based setting, i.e. resource-constraint edge devices.

Regarding the distributed processing and integration of RSP engines on a truly decentralised architecture, different aspects and building blocks have surfaced in the latest years. Initial attempts to provide HTTP-based service interfaces for streaming data were explored in (Barbieri et al. 2010). Other contributions in this line are the RSP Service Interface<sup>3</sup>, and the SLD Revolution framework (Balduini et al. 2017). These propose the establishment of distributed workflows of RSP engines, using lazy-transformation

---

<sup>3</sup><http://streamreasoning.org/resources/rsp-services>

techniques for optimised interactions among the engines. Further conceptualisations of RDF stream processing over decentralised entities have been presented in works such as WeSP (Dell’Aglia et al. 2017)<sup>4</sup>, which advocates for a community-driven definition of stream vocabularies and interoperable interfaces. Cooperation strategy among RDF stream processors, or *stream reasoning agents* is discussed in (Tommasini et al. 2019), introducing potential challenges and opportunities for federated processing through negotiation established across multiagent systems.

## 5.6 Conclusion

This paper has presented a continuous query federation approach that uses RSP engines as autonomous processing agents. The approach enables the coordination of edge devices’ resources to process query processing pipelines by cooperatively delegating partial workloads to their peer agents. We implemented our approach as an open source engine, Fed4Edge, to conduct an empirical study in “cooperative sensing” scenarios. The resourceful experiments of the study show that the scalability can be significantly improved by adding more edge devices to a network of processing nodes on demand. This opens several interesting follow-up research challenges in enabling semantic interoperability for the edge computing paradigm. Our next step will be investigating on how to adaptively optimise the distributed processing pipeline of Fed4Edge. Another interesting step is studying how the communication will effect its performance and scalability on an Internet-scale setting whereby the processing nodes are distributed across different networks and countries.

---

<sup>4</sup><http://w3id.org/wesp/web-data-streams>

# Chapter 6

## Conclusion

### 6.1 Contributions

The significant contributions of the research presented in this thesis can be summarised as follows.

#### **An empirical study of RDF engines running on IoT edge devices**

To address [RQ1](#), we presented an empirical study on the behaviours of RDF engines when they run on IoT edge devices ([Chapter 2, Section 3](#)). In this study, we analysed the performance of the PC-based RDF engines, i.e., Jena TDB, RDF4J, and Virtuoso, when they run on single board computers that are representative of IoT edge devices, i.e., Galileo boards, BeagleBone boards and Raspberry Pi boards. The study was conducted in a simulated IoT scenario of weather data management. To simulate IoT semantic data, we mapped sensor readings from the Integrated Surface Database (ISD) of the National Climate Data Center (NCDC) to RDF using the SSN ontology. The results showed that PC-based RDF engines suffer critical performance issues when they run on such devices due to two major reasons: (i) the lack of main memory and (ii) the specific I/O behaviour of flash-based storage.

#### **A RISC-Style design for a lightweight RDF engine**

The results of [RQ1](#) study indicate that a lightweight RDF engine with a minimum code and memory footprint is required to process RDF data on IoT edge devices. This requirement was briefly presented in [RQ2.1](#). Thus, we proposed a RISC-style approach for designing a lightweight RDF engine ([Chapter 2, Section 4](#)). The RISC-style design philosophy indicates that the necessary features for an RDF engine can be implemented around data access and join operations. The processing load and resource consumption of an RDF engine are mainly caused by these operations. Our approach was to focus the design and optimisation efforts on these operators and use simple implementations for the rest of the engine to reduce the memory and code footprint. As a result, the size of

our prototype, RDF4Led, is only 4 MB, while the size of Jena TDB is 13 MB; the size of RDF4J is 58 MB; and the size of Virtuoso is 180 MB. Moreover, RDF4Led requires only 10%–30% of the memory consumed by Jena, TDB, RDF4J, and Virtuoso when operating on the same size of the dataset (Chapter 2, Section 8).

### A flash-friendly indexing data structure and flash-friendly buffer management for RDF data.

To overcome the research problem presented in RQ2.2, we proposed a lightweight flash-friendly index structure (Chapter 2, Section 5) and a buffer management (Chapter 2, Section 6). We stored RDF triples in a compact format known as “RDF molecule”. To adapt to the specific flash I/O behaviour, we organised the RDF molecules into block units whose size is equal to the flash-erase block size. To reduce the memory required to maintain the indexes of data in the flash storage, we used an alternative index structure that is based on the Block Range Index (BRIN) approach. The basic idea of BRIN is to summarise the information of a data block on persistent storage (e.g., its location) into a small tuple. On top of this index structure, we added an in-memory caching mechanism to cache the atomic data and to cluster the write operations to improve the write performance. The buffer management technique follows the basic principles: (i) minimise the number of physical writes to physical storage; (ii) group multiple updates within one write operation; (iii) keep a relatively high hit ratio for the data in the buffer. Our strategy was to organise the data blocks in two queues, a hot queue and a cold queue. The hot queue keeps the blocks that have been recently accessed. The cold queue contains molecule blocks (blocks in compressed form) that are ready to be written to the flash memory. Data blocks released from the hot queue are compressed and are kept in the cold queue before evicting from the buffer. The data blocks in each queue are released in the order of priorities: (i) clean/unmodified blocks; (ii) higher density blocks, defined by the ratio between the number of triples in a block and the capacity of the block i.e.,  $density_{BlockA} = \frac{\#triples_{BlockA}}{capacity_{BlockA}}$ ; (iii) the least recently used blocks. Consequently, RDF4Led can handle 2–5 times more data than its competitors and can outperform its competitors in updating throughput (Chapter 2, Section 8).

### A low-memory-footprint join algorithm for SPARQL processor.

To address the research question RQ2.3, we proposed a low-memory footprint join algorithm for SPARQL (Chapter 2, Section 7). To minimise the memory required for executing a SPARQL query and making the best use of the indexing scheme, we adopted the *one-tuple-at-a-time* approach to compute the join. This approach can reduce the memory consumption as no virtual temporary memory is required to buffer the intermediate join results. The basic idea of the algorithm to compute the join of a graph pattern is as follows: A mapping solution (mapping for short) is continuously sent to visit each triple pattern of the graph pattern. In each visit, it searches for triples matching the triple pattern. For each matched triple, variables in the triple pattern and the corresponding value in the triple are added to the mapping. The mapping to new values will be sent to visit the next triple pattern, or be returned as a query result when all triple

patterns have been visited. In each run, the algorithm decides which triple pattern the mapping should visit first. Similarly to the routing policy of stream processing engines, the algorithm defines a propagating policy to achieve a certain optimisation purpose. In our case, we attempted to minimise the number of propagations by choosing the shortest index scan in each run. Consequently, RDF4Led consumes only 10%–30% of the memory consumed by its competitors when operating on the same size of dataset and it runs it is faster in answering queries than its Java counterparts, Jena TDB and RDF4J (Chapter 2, Section 8).

### **A framework for integrating heterogeneous personal information on mobile phones.**

To show the feasibility of our RDF engine (RQ3), we proposed a comprehensive framework for the integration of personal data on mobile devices (Chapter 3). The framework is built upon Linked Data technology to be generic with respect to the supported data types and data requests from heterogeneous data sources, such as different social networks. The data integration approach is based on an RDF engine for mobile devices. Furthermore, the framework offers a plug-in model to be extensible for additional data sources and relies solely on a user's mobile device. Additionally, it has a data normalisation approach that can deal with ID-consolidation and ambiguity issues without complex generic reasoning. With the support of RDF On The Go, a version of RDF4Led for mobile phones, our framework can store and query more than a thousand social network profiles (Chapter 3, Section 4).

### **A deep integration of distributed RDF store and blockchain on IoT edge devices.**

To address the (RQ3.2), we presented an architecture and implementation of a novel RDF distributed store with blockchain technology for decentralised IoT edge networks (Chapter 4). The system uses the distributed file system IPFS to enable the share of RDF data across edge nodes. The index of the RDF data in IPFS is securely stored in the smart contracts of a private Ethereum blockchain network. Our experiments proved that the system can be deployed on a network of lightweight edge devices such as Raspberry Pi. On a network of fifteen nodes of Raspberry Pi, the system is able to host a dataset up to a billion RDF triples (Chapter 4, Section 4). The number of nodes can be elastically increased or decreased at runtime.

### **A cooperative federation mechanism to distribute RDF stream processing over IoT edge nodes**

We proposed a continuous query federation approach that uses RDF stream processing (RSP) engines as autonomous processing agents (Chapter 5). The approach enables the coordination of edge devices' resources to process query processing pipelines by cooperatively delegating a partial workload to their peer agents. To enable edge devices to process RDF streams, we implemented a decentralised version of the RDF engine by integrating CQELS and RDF4Led, called Fed4Edge. Our experiments demonstrate the bottleneck

caused by the centralised architecture of RDF stream processing. In our decentralised architecture with the cooperation feature, adding more edge devices to the network of processing nodes significantly improves the RDF stream processing scalability ([Chapter 5, Section 5](#)).

## 6.2 Future Work

The results presented in this thesis pave the way to future research in semantic data processing on the edge of IoT. In the following, we discuss the directions that this work can be extended.

### Self-tuning feature

The evaluation results in ([Chapter 2, Section 8](#)) showed that on the Raspberry Pi 3 (which is equipped with a quad-core CPU), RDF4Led's insertion speed is slower than that of Virtuoso when the storage size is greater than 20 million triples (see [Figure 2.12\(c\)](#)). On a quad-core CPU device, the data structures and algorithms that implement Virtuoso may work better with large size datasets. Hence, it is opening up an opportunity to further optimise RDF4Led to have better performance running on hardware devices similarly to Raspberry Pi 3. Furthermore, it is desirable to develop a mechanism that allows RDF4Led to adaptively select (self-tuning) and employ the optimal data structures and algorithms depending on hardware and data size.

### Support semantic time series data

The adding of semantics to time-series data ([Zhang, Zeng, Yen & Bastani 2019](#)) and offloading of time-series data processing to the edge node ([Yang et al. 2019](#)) are emerging challenges for the IoT. Therefore, supporting semantic time series data ([Zhang, Wang, Li & Cheng 2019](#)) is another desirable feature that we would like to equip in a future version of RDF4Led. The first challenge of this task is to employ a sufficient index data structure to support time-range queries. Furthermore, the incoming time-series data in high rate need to be written to secondary devices as soon as possible, because there is limited memory on edge devices to cache the data. Thus, to process time-series data, an adaptive mechanism is required to tradeoff between read performance, writing rate, and memory available on edge devices. Hence, we are looking for a model to trade-off these performances similar to the RUM overhead model ([Athanassoulis et al. 2016](#)).

### Energy efficiency

Optimising energy consumption is important for IoT edge devices as they often are battery-powered. The energy consumption can be reduced by algorithmic solutions ([Albers 2010](#)). [Demaine et al. \(2016\)](#) have analysed the energy complexity of common algorithms used in database systems (such as searching, sorting). In the next step, we would like to extend the work on energy efficient algorithms to be applicable for RDF4Led.

### Decentralised SPARQL processing for IoT edge networks

Since RDF data can be stored locally on the IoT edge nodes, enabling federated SPARQL queries over these IoT edge nodes is an interesting but challenging research direction. Our study on federation RSP in IoT edge network (see [Chapter 5](#)) showed that the federated system may suffer performance degradation if data is routed to one centralised node where the query is processed. Furthermore, the empirical study in this work also indicated that the network topology might influence the performance of the federation engine. In an IoT edge network, the computation power may be uneven on different IoT edge devices, sending the data to a more powerful node and processing the data on the node (e.g. join) may be faster than executing such operations locally. However, the state-of-the-art federated SPARQL query engines have not taken these factors into account in their design. Therefore, a novel distributed mechanism is required for processing federated SPARQL queries for the IoT edge networks. To develop this mechanism, we propose to extend the linked data fragments framework ([Hartig et al. 2017](#)) to disassemble a federated SPARQL query plan into a processing pipeline. The operators of the pipeline will be placed on different nodes in the network depending on the complexity of the operator and the capability of the processing node. We propose to investigate the challenges addressed in [Chapter 5, Section 4.4](#) to find the optimal operator placement.



# Appendix A

## SPARQL queries used in the experiments of Chapter 2

Linear joins queries:

---

```
#Linear template L1
PREFIX sosa:<http://www.w3.org/ns/sosa/>

SELECT ?sensor ?observation ?simpleResult
WHERE {
  ?sensor      sosa:observes      %ObservableProperty%.
  ?sensor      sosa:madeObservation ?observation.
  ?observation sosa:hasSimpleResult ?simpleResult.
}
```

---

L1 - List all sensors, observations and observation results of a observable property.

---

```
#Linear template L2
PREFIX sosa:<http://www.w3.org/ns/sosa/>

SELECT ?sensor ?observation
WHERE {
  %station%      sosa:hosts      ?sensor.
  ?observation   sosa:madebySensor ?sensor.
  ?observation   sosa:observedProperty iot:Temperature;
}
```

---

L2 - List all temperature sensors and temperature observations at a given station.

---

```
#Linear template L3
PREFIX sosa:<http://www.w3.org/ns/sosa/>

SELECT ?sensor ?observation ?featureOfInterest
WHERE {
  ?observation sosa:hasFeatureOfInterest ?featureOfInterest;
  ?observation sosa:madeBySensor ?sensor.
  %station% sosa:hosts ?sensor.
}
```

---

L3 - List all sensors, observations and features of interest at a location.

---

```
#Linear template L4
PREFIX sosa:<http://www.w3.org/ns/sosa/>

SELECT ?observation ?obsProperty ?simpleResult
WHERE {
  ?observation sosa:observedProperty ?obsProperty.
  ?observation sosa:hasSimpleResult ?simpleResult.
  %sensor% sosa:observes ?obsProperty.
}
```

---

L4 - List all observations, observable properties and results of a sensor.

### Start join queries:

---

```
#Star template S1
PREFIX sosa:<http://www.w3.org/ns/sosa/>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?observation ?time ?result ?simpleResult
       ?featureOfInterest ?obsProperty
WHERE {
  ?observation rdf:type sosa:Observation.
  ?observation sosa:resultTime ?time.
  ?observation sosa:hasResult ?result.
  ?observation sosa:hasSimpleResult ?simpleResult.
  ?observation sosa:hasFeatureOfInterest ?featureOfInterest.
  ?observation sosa:observedProperty ?obsProperty.
  %sensor% sosa:madeObservation ?observation.
}
```

---

S1 - List the information of all observations made by a sensor.

---

```

#Star template S2
PREFIX sosa:<http://www.w3.org/ns/sosa/>

SELECT ?observation ?simpleResult ?time
WHERE {
  ?observation sosa:resultTime           ?time;
  ?observation sosa:hasSimpleResult      ?simpleResult.
  ?observation sosa:hasFeatureOfInterest %featureOfInterest%.
  ?observation sosa:observedProperty    %observableProperty%.
}

```

---

S2 - List all observations, results, and observation timestamps.

---

```

#Star template S3
PREFIX sosa:<http://www.w3.org/ns/sosa/>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?sensor ?observation
WHERE {
  ?sensor      rdf:type          sosa:Sensor;
  ?sensor      sosa:observes     %obsProperty%;
  ?sensor      sosa:isHostedBy   %station%.
  ?observation sosa:madeBySensor ?sensor.
}

```

---

S3 - List all observations of a sensor that observes a specific observable property at a specific station.

### Snowflake joins queries:

---

```

#Snowflake template F1
PREFIX sosa:<http://www.w3.org/ns/sosa/>

SELECT ?sensor ?station ?observation ?time ?simpleResult
WHERE {
  ?sensor      sosa:observes          %ObservableProperty%;
  ?sensor      sosa:isHostedBy        ?station.
  ?observation sosa:madeBySensor      ?sensor;
  ?observation sosa:hasSimpleResult    ?simpleResult;
  ?observation sosa:resultTime        ?time;
  ?observation sosa:hasFeatureOfInterest <:foi/0001>.
}

```

---

F1 - Search for information of observations that observe a specific observable property.

---

```

#Snowflake template F2
PREFIX sosa:<http://www.w3.org/ns/sosa/>
PREFIX qudt11:<http://www.geonames.org/ontology#>

SELECT ?observation ?sensor ?station ?featureOfInterest
       ?time ?simpleResult ?type
WHERE {
  ?observation sosa:observedProperty      %ObservableProperty%.
  ?observation sosa:resultTime             ?time;
  ?observation sosa:madeBySensor           ?sensor;
  ?observation sosa:hasFeatureOfInterest ?featureOfInterest;
  ?observation sosa:hasSimpleResult        ?simpleResult;
  ?observation sosa:hasResult               ?result;
  ?result      qudt11:unit                  ?unit;
  ?result      qudt11:numericValue         ?value.
}

```

---

F2 - Search for information of observations that observe a specific observable property.

---

```

#Snowflake template F3
PREFIX ssn:<http://www.w3.org/ns/ssn/>
PREFIX sosa:<http://www.w3.org/ns/sosa/>
PREFIX wgs84:<http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX dul:<http://www.loa-cnr.it/ontologies/DUL.owl#>

SELECT ?observation ?obsProperty ?time ?simpleResult
WHERE {
  ?featureOfInterest ssn:hasProperty      %ObservableProperty%;
  ?featureOfInterest dul:hasLocation      ?loc.
  ?loc                wgs84:Point         ?point.
  ?point              wgs84:lat           "%lat%";
  ?point              wgs84:lon          "%lon%".
  ?observation        sosa:resultTime     ?time;
  ?observation        sosa:hasSimpleResult ?simpleResult;
  ?observation        sosa:hasFeatureOfInterest ?featureOfInterest.
}

```

---

F3 - List all observation results and observation timestamps of the observations that observe a specific observable property at a specific location.

---

```
#Snowflake template F4
PREFIX sosa:<http://www.w3.org/ns/sosa/>
PREFIX qudt11:<http://www.geonames.org/ontology#>

SELECT ?observation ?time ?unit ?value
WHERE {
  %sensor%      sosa:madeObservation      ?observation.
  ?observation sosa:hasFeatureOfInterest ?featureOfInterest;
  ?observation sosa:resultTime           ?time;
  ?observation sosa:hasResult            ?result.
  ?result      qudt11:unit                ?unit;
  ?result      qudt11:numericValue       ?value.
}
```

---

F4 - List all information of the observation made by a given sensor.



# Appendix B

## SPARQL queries used in the experiments of Chapter 3

---

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

```
SELECT ?person ?property ?info
WHERE {
  ?person foaf:mbox "mailto:Thierry59@gmx.com".
  ?person ?property ?info.
}
```

---

Query 1: Return all information of a person by given mbox

---

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX db:   <http://dbpedia.org/resource/>
```

```
SELECT ?firstname ?lastname ?mbox ?friend ?birthday ?gender
WHERE {
  ?person foaf:based_near db:Bulgaria.
  ?person foaf:firstName ?firstname.
  ?person foaf:lastName ?lastname.
  ?person foaf:mbox ?mbox.
  ?person foaf:birthday ?birthday.
  ?person foaf:gender ?gender.
}
```

---

Query 2: Extract some information of people who are nearby Bulgaria

---

```
PREFIX sioc: <http://rdfs.org/sioc/ns#>
PREFIX sibv: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX dbpo: <http://dbpedia.org/ontology/>
PREFIX fbp: <http://www.facebook.com/person/>

SELECT DISTINCT ?location
WHERE {
  ?user sioc:account_of fbp:p151.
  ?photo sibv:usertag ?user.
  ?photo dbpo:location ?location.
}
```

---

Query 3: Request all the locations that a person has taken a photo

---

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

SELECT DISTINCT ?predicate
WHERE {
  ?person rdf:type foaf:Person.
  ?subject ?predicate ?person.
}
```

---

Query 4: Request incoming property of a person

---

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

SELECT DISTINCT ?predicate
WHERE {
  {
    ?person rdf:type foaf:Person.
    ?subject ?predicate ?person.
  }
  UNION
  {
    ?person rdf:type foaf:Person.
    ?person ?predicate ?object.
  }
}
```

---

Query 5: Request incoming and out coming properties of person

---

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX sibv: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fbp:  <http://www.facebook.com/person/>

SELECT DISTINCT ?photo
WHERE{
  fbp:p39 foaf:knows      ?person.
  ?user  sioc:account_of ?person.
  ?user  sibv:like       ?photo.
  ?photo rdf:type        sibv:Photo.
}
```

---

Query 6: Request all the photos that are liked by a person

---

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sioc: <http://rdfs.org/sioc/ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX fbph: <http://www.facebook.com/photoalbum/>

SELECT ?person {
  GRAPH <facebook> {
    ?person rdf:type foaf:Person.
  }

  GRAPH <master> {
    ?user sioc:account_of ?person.
    ?user sioc:creator_of fbph:pa103.
  }
}
```

---

Query 7: Request the photos on Facebook by LinkedIn account

---

```
PREFIX sibv: <http://www.ins.cwi.nl/sib/vocabulary/>
PREFIX sioc: <http://rdfs.org/sioc/ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?user ?comment
{
  GRAPH <facebook> {
    ?comment rdf:type sibv:Comment.
    ?user sioc:creator_of ?comment.
  }

  GRAPH <master> {
    ?user sioc:account_of <http://linkedin.com/person/p174>
  }
}
```

---

Query 8: Request the comments of a friend on Facebook by his Linked account

# Bibliography

- Abadi, D. J., Marcus, A., Madden, S. R. & Hollenbach, K. (2007), Scalable semantic web data management using vertical partitioning, *in* ‘Proceedings of the 33rd International Conference on Very Large Data Bases’, VLDB ’07, VLDB Endowment, pp. 411–422.
- Abadi, D. J., Marcus, A., Madden, S. R. & Hollenbach, K. (2009), ‘Sw-store: a vertically partitioned dbms for semantic web data management’, *The VLDB Journal* **18**(2), 385–406.
- Agarwal, R., Khandelwal, A. & Stoica, I. (2015), Succinct: Enabling queries on compressed data, *in* ‘12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)’, pp. 337–350.
- Aggarwal, C. C., Ashish, N. & Sheth, A. (2013), The internet of things: A survey from the data-centric perspective, *in* ‘Managing and mining sensor data’, Springer, pp. 383–428.
- Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y. & Singh, S. (2009), ‘Lazy-adaptive tree: An optimized index structure for flash devices’, *Proc. VLDB Endow.* **2**(1), 361–372.
- Ajwani, D., Malinger, I., Meyer, U. & Toledo, S. (2008), Characterizing the performance of flash memory storage devices and its impact on algorithm design, *in* C. McGeoch, ed., ‘Experimental Algorithms’, Vol. 5038 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 208–219.
- Akpakwu, G. A., Silva, B. J., Hancke, G. P. & Abu-Mahfouz, A. M. (2017), ‘A survey on 5g networks for the internet of things: Communication technologies and challenges’, *IEEE Access* .
- Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M. & Ayyash, M. (2015), ‘Internet of things: A survey on enabling technologies, protocols, and applications’, *IEEE Communications Surveys Tutorials* **17**(4), 2347–2376.
- Albers, S. (2010), ‘Energy-efficient algorithms’, *Communications of the ACM* **53**(5), 86–96.
- Aluç, G., Hartig, O., Özsu, M. T. & Daudjee, K. (2014), Diversified stress testing of rdf data management systems, *in* ‘The Semantic Web – ISWC 2014’, Springer International Publishing, pp. 197–212.

- Andročec, D., Novak, M. & Oreški, D. (2018), ‘Using semantic web for internet of things interoperability: A systematic review’, *International Journal on Semantic Web and Information Systems (IJSWIS)* **14**(4), 147–171.
- Antonopoulos, A. M. & Wood, G. (2018), *Mastering ethereum: building smart contracts and dapps*, O’reilly Media.
- Apolinarski, W., Handte, M., Iqbal, M. & Marron, P. (2013), Pike: Enabling secure interaction with piggybacked key-exchange, *in* ‘Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on’, pp. 94–102.
- Aranda-Andújar, A., Bugiotti, F., Camacho-Rodríguez, J., Colazzo, D., Goasdoué, F., Kaoudi, Z. & Manolescu, I. (2012), Amada: web data repositories in the amazon cloud, *in* ‘Proceedings of the 21st ACM international conference on Information and knowledge management’, pp. 2749–2751.
- Arasteh, H., Hosseinneshad, V., Loia, V., Tommasetti, A., Troisi, O., Shafie-khah, M. & Siano, P. (2016), Iot-based smart cities: A survey, *in* ‘2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)’, IEEE, pp. 1–6.
- Arenas, M. & Pérez, J. (2011), Querying semantic web data with sparql, *in* ‘Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems’, PODS ’11, Association for Computing Machinery, New York, NY, USA, p. 305–316.
- Ashton, K. (2009), ‘That ‘internet of things’ thing’, *RFID journal* **22**(7), 97–114.
- Athanassoulis, M., Kester, M. S., Maas, L. M., Stoica, R., Idreos, S., Ailamaki, A. & Callaghan, M. (2016), Designing access methods: The rum conjecture., *in* ‘EDBT’, Vol. 2016, pp. 461–466.
- Atre, M., Chaoji, V., Zaki, M. J. & Hendler, J. A. (2010), Matrix ”bit” loaded: A scalable lightweight join query processor for rdf data, *in* ‘Proceedings of the 19th International Conference on World Wide Web’, WWW ’10, Association for Computing Machinery, New York, NY, USA, p. 41–50.
- Atzori, L., Iera, A. & Morabito, G. (2010), ‘The internet of things: A survey’, *Computer Networks* **54**(15), 2787 – 2805.
- Atzori, L., Iera, A. & Morabito, G. (2017), ‘Understanding the internet of things: definition, potentials, and societal role of a fast evolving paradigm’, *Ad Hoc Networks* **56**, 122 – 140.
- Avnur, R. & Hellerstein, J. M. (2000), Eddies: Continuously adaptive query processing, *in* ‘Proceedings of the 2000 ACM SIGMOD international conference on Management of data’, pp. 261–272.

- Baker, S. B., Xiang, W. & Atkinson, I. (2017), ‘Internet of things for smart healthcare: Technologies, challenges, and opportunities’, *IEEE Access* **5**, 26521–26544.
- Balazinska, M., Balakrishnan, H. & Stonebraker, M. (2004), Contract-based load management in federated distributed systems, *in* ‘NSDI’04’.
- Balduini, M., Della Valle, E. & Tommasini, R. (2017), Sld revolution: A cheaper, faster yet more accurate streaming linked data framework, *in* ‘ESWC’.
- Barbieri, D. F., Braga, D., Ceri, S., Della Valle, E. & Grossniklaus, M. (2009), C-sparql: Sparql for continuous querying, *in* ‘Proceedings of the 18th international conference on World wide web’, pp. 1061–1062.
- Barbieri, D. F., Braga, D., Ceri, S. & Grossniklaus, M. (2010), An execution environment for c-sparql queries, *in* ‘Proceedings of the 13th International Conference on Extending Database Technology’, pp. 441–452.
- Barnaghi, P., Wang, W., Henson, C. & Taylor, K. (2012), ‘Semantics for the internet of things: Early progress and back to the future’, *Int. J. Semant. Web Inf. Syst.* **8**(1), 1–21.
- Bazoobandi, H. R., de Rooij, S., Urbani, J., ten Teije, A., van Harmelen, F. & Bal, H. (2015), A compact in-memory dictionary for rdf data, *in* ‘European Semantic Web Conference’, Springer, pp. 205–220.
- Beckett, D. (2002), ‘The design and implementation of the redland rdf application framework’, *Computer Networks* **39**(5), 577 – 588.  
**URL:** <http://www.sciencedirect.com/science/article/pii/S1389128602002219>
- Benet, J. (2014), ‘Ipfs - content addressed, versioned, p2p file system’, *arXiv:1407.3561* .
- Bera, A. (2019), ‘80 Insightful Internet of Things Statistics.’, Available online: <https://safeatlast.co/blog/iot-statistics/>. (accessed on 21 June 2020).
- Berners-Lee, T., Hendler, J., Lassila, O. et al. (2001), ‘The semantic web’, *Scientific american* **284**(5), 28–37.
- Bojārs, U., Passant, A., Breslin, J. & Decker, S. (2008), Social network and data portability using semantic web technologies, *in* ‘Proceedings of the BIS 2008 Workshop on Social Aspects of the Web, Innsbruck, Austria (May 2008)’.
- Bolles, A., Grawunder, M. & Jacobi, J. (2008), Streaming sparql - extending sparql to process data streams, *in* S. Bechhofer, M. Hauswirth, J. Hoffmann & M. Koubarakis, eds, ‘The Semantic Web: Research and Applications’, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 448–462.
- Botta, A., de Donato, W., Persico, V. & Pescapé, A. (2016), ‘Integration of cloud computing and internet of things: A survey’, *Future Generation Computer Systems* **56**, 684 – 700.

- Bouganim, L., Jonsson, B. & Bonnet, P. (2009), ‘uflip: Understanding flash IO patterns’, *CoRR* **abs/0909.1780**.
- Bröcheler, M., Pugliese, A. & Subrahmanian, V. S. (2009), Dogma: A disk-oriented graph matching algorithm for rdf databases, *in* ‘International Semantic Web Conference’, Springer, pp. 97–113.
- Broekstra, J., Kampman, A. & van Harmelen, F. (2002), Sesame: A generic architecture for storing and querying rdf and rdf schema, *in* I. Horrocks & J. Hendler, eds, ‘The Semantic Web — ISWC 2002’, Vol. 2342 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 54–68.
- Bröring, A., Ziller, A., Charpenay, V., Thuluva, A. S., Anicic, D., Schmid, S., Zappa, A., Linares, M. P., Mikkelsen, L. & Seidel, C. (2018), ‘The big iot api-semantically enabling iot interoperability’, *IEEE Pervasive Computing* **17**(4), 41–51.
- Calbimonte, J.-P., Corcho, O. & Gray, A. J. G. (2010), Enabling ontology-based access to streaming data sources, *in* ‘ISWC’, Springer, pp. 96–111.
- Charpenay, V., Käbisch, S. & Kosch, H. (2016), Introducing thing descriptions and interactions: An ontology for the web of things., *in* ‘SR+ SWIT@ ISWC’, pp. 55–66.
- Charpenay, V., Käbisch, S. & Kosch, H. (2017),  $\mu$  rdf store: Towards extending the semantic web to embedded devices, *in* ‘European Semantic Web Conference’, Springer, pp. 76–80.
- Chaudhuri, S. & Weikum, G. (2000), Rethinking database system architecture: Towards a self-tuning risc-style database system., *in* ‘VLDB’, pp. 1–10.
- Chettri, L. & Bera, R. (2019), ‘A comprehensive survey on internet of things (iot) toward 5g wireless systems’, *IEEE Internet of Things Journal* **7**(1), 16–32.
- Chong, E. I., Das, S., Eadon, G. & Srinivasan, J. (2005), An efficient sql-based rdf querying scheme, *in* ‘Proceedings of the 31st International Conference on Very Large Data Bases’, VLDB ’05, VLDB Endowment, pp. 1216–1227.
- Comer, D. (1979), ‘Ubiquitous b-tree’, *ACM Computing Surveys (CSUR)* **11**(2), 121–137.
- Curé, O. & Blin, G. (2014), *RDF database systems: triples storage and SPARQL query processing*, Morgan Kaufmann.
- Daniele, L., den Hartog, F. & Roes, J. (2015), Created in close interaction with the industry: the smart appliances reference (saref) ontology, *in* ‘International Workshop Formal Ontologies Meet Industries’, Springer, pp. 100–112.
- Dannen, C. (2017), *Introducing Ethereum and solidity*, Vol. 318, Springer.
- d’Aquin, M., Nikolov, A. & Motta, E. (2011), Building sparql-enabled applications with android devices, *in* ‘ISWC 2011’.

- David, J. & Euzenat, J. (2010), Linked data from your pocket: The android rdfcontent-provider, *in* 'ISWC 2010'.
- De, S., Barnaghi, P., Bauer, M. & Meissner, S. (2011), Service modelling for the internet of things, *in* '2011 Federated Conference on Computer Science and Information Systems (FedCSIS)', pp. 949–955.
- Decker, S. & Frank, M. R. (2004), The networked semantic desktop, *in* 'WWW Workshop on Application Design, Development and Implementation Issues in the Semantic Web'.
- Dell'Aglio, D., Dao-Tran, M., Calbimonte, J.-P., Le Phuoc, D. & Della Valle, E. (2016), A query model to capture event pattern matching in rdf stream processing query languages, *in* '20th International Conference on Knowledge Engineering and Knowledge Management - Volume 10024', EKAW 2016, Springer-Verlag New York, Inc., New York, NY, USA, pp. 145–162.
- Dell'Aglio, D., Le Phuoc, D., Le-Tuan, A., Intizar Ali, M. & Calbimonte, J.-P. (2017), 'On a web of data streams'.
- Dell'Aglio, D., Della Valle, E., van Harmelen, F. & Bernstein, A. (2017), 'Stream reasoning: A survey and outlook', *Data Science* **1**(1-2), 59–83.
- Demaine, E. D., Lynch, J., Mirano, G. J. & Tyagi, N. (2016), Energy-efficient algorithms, *in* 'Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science', pp. 321–332.
- Desai, P., Sheth, A. & Anantharam, P. (2015), Semantic gateway as a service architecture for iot interoperability, *in* '2015 IEEE International Conference on Mobile Services', pp. 313–319.
- DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R. & Wood, D. A. (1984), Implementation techniques for main memory database systems, *in* 'SIGMOD', pp. 1–8.
- DeWitt, D. J., Naughton, J. F. & Burger, J. (1993), Nested loops revisited, *in* 'Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on', IEEE, pp. 230–242.
- Dong, X. L. & Halevy, A. (2005), A platform for personal information management and integration, *in* 'Proceedings of VLDB 2005 PhD Workshop', Citeseer, p. 26.
- Dunkels, A. & Vasseur, J. (2008), 'Ip for smart objects, internet protocol for smart objects (ipso) alliance, white paper# 1'.
- Erling, O. & Mikhailov, I. (2009), Rdf support in the virtuoso dbms, *in* 'Networked Knowledge - Networked Media'.

- Fletcher, G. H. & Beck, P. W. (2009), Scalable indexing of rdf graphs for efficient join processing, *in* ‘Proceedings of the 18th ACM conference on Information and knowledge management’, pp. 1513–1516.
- Ganzha, M., Paprzycki, M., Pawłowski, W., Szmeja, P. & Wasielewska, K. (2017), ‘Semantic interoperability in the internet of things: An overview from the inter-iot perspective’, *Journal of Network and Computer Applications* **81**, 111–124.
- Gartner (2019), ‘Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020.’, Available online: <https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-iot>. (accessed on 21 June 2020).
- Gershenfeld, N. & Cohen, D. (2006), ‘Internet 0: Interdevice internetworking - end-to-end modulation for embedded networks’, *IEEE Circuits and Devices Magazine* **22**(5), 48–55.
- Gorenflo, C., Lee, S., Golab, L. & Keshav, S. (2019), ‘Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second’, *arXiv:1901.00910* .
- Graefe, G. (1993), ‘Query evaluation techniques for large databases’, *ACM Computing Surveys (CSUR)* **25**(2), 73–169.
- Graefe, G. (2003), Executing nested queries, *in* ‘BTW 2003, Datenbanksysteme für Business, Technologie und Web’.
- Graefe, G. (2007), The five-minute rule twenty years later, and how flash memory changes the rules, *in* ‘Proceedings of the 3rd International Workshop on Data Management on New Hardware’, DaMoN ’07, Association for Computing Machinery, New York, NY, USA.  
**URL:** <https://doi.org/10.1145/1363189.1363198>
- Grubenmann, T., Bernstein, A., Moor, D. & Seuken, S. (2018), Financing the web of data with delayed-answer auctions, *in* ‘WWW ’18’.
- Guinard, D. & Trifa, V. (2009), Towards the web of things: Web mashups for embedded devices, *in* ‘Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain’, Vol. 15, p. 8.
- Gyrard, A., Datta, S. K., Bonnet, C. & Boudaoud, K. (2015), A semantic engine for internet of things: Cloud, mobile devices and gateways, *in* ‘2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing’, IEEE, pp. 336–341.
- Gyrard, A., Patel, P., Datta, S. K. & Ali, M. I. (2017), Semantic web meets internet of things and web of things, *in* ‘Proceedings of the 26th International Conference on World Wide Web Companion’, pp. 917–920.

- Haller, A., Janowicz, K., Cox, S. J. D., Lefrançois, M., Taylor, K., Le-Phuoc, D., Lieberman, J., García-Castro, R., Atkinson, R. & Stadler, C. (2019), ‘The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation’, *Semantic Web* **10**(1), 9–32.
- Harris, S., Lamb, N. & Shadbolt, N. (2009), 4store: The design and implementation of a clustered rdf store, *in* ‘5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)’, pp. 94–109.
- Harth, A. & Decker, S. (2005), Optimized index structures for querying rdf from the web, *in* ‘Proceedings of the Third Latin American Web Congress’, LA-WEB ’05, IEEE Computer Society, Washington, DC, USA, pp. 71–.
- Hartig, O., Letter, I. & Pérez, J. (2017), A formal framework for comparing linked data fragments, *in* ‘International semantic web conference’, Springer, pp. 364–382.
- Hasemann, H., Kroller, A. & Pagel, M. (2014), ‘The wiselib tuplestore: A modular RDF database for the internet’, *CoRR* .
- Hauswirth, M., Wylot, M., Grund, M., Groth, P. & Cudré-Mauroux, P. (2017), *Linked Data Management*, Springer International Publishing, pp. 307–338.
- Hayun, R. B. (2009), *Java ME on Symbian OS: inside the smartphone model*, Vol. 30, John Wiley & Sons.
- Ho, V.-P. & Park, D.-J. (2016), ‘A survey of the-state-of-the-art b-tree index on flash memory’, **10**, 173–188.
- Hoang, H. H., Andjomshoaa, A. & Tjoa, A. M. (2006), Towards a new approach for information retrieval in the semanticle digital memory framework, *in* ‘Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence’, WI ’06, IEEE Computer Society, Washington, DC, USA, pp. 485–488.
- Hossain, M. M., Fotouhi, M. & Hasan, R. (2015), Towards an analysis of security issues, challenges, and open problems in the internet of things, *in* ‘2015 IEEE World Congress on Services’, pp. 21–28.
- Jin, P., Ou, Y., Harder, T. & Li, Z. (2012), ‘Ad-lru: An efficient buffer replacement algorithm for flash-based databases’, *Data & Knowledge Engineering* **72**, 83 – 102.
- Kaebisch, S., Kamiya, T., McCool, M. & Charpenay, V. (2019a), ‘Web of things (wot) thing description’, *W3C, W3C Candidate Recommendation* .
- Kaebisch, S., Kamiya, T., McCool, M. & Charpenay, V. (2019b), ‘Web of things (wot) thing description’, *W3C, W3C Candidate Recommendation* .
- Khadilkar, V., Kantarcioglu, M., Thuraisingham, B. & Castagna, P. (2012), Jena-hbase: A distributed, scalable and efficient rdf triple store, *in* ‘ISWC Posters & Demonstrations Track’, Vol. 12, Citeseer, pp. 85–88.

- Kiljander, J., D'elia, A., Morandi, F., Hyttinen, P., Takalo-Mattila, J., Ylisaukko-Oja, A., Soininen, J. & Cinotti, T. S. (2014), 'Semantic interoperability architecture for pervasive computing and internet of things', *IEEE Access* **2**, 856–873.
- Langley, D. J., van Doorn, J., Ng, I. C., Stieglitz, S., Lazovik, A. & Boonstra, A. (2020), 'The internet of everything: Smart things and their impact on business models', *Journal of Business Research* **122**, 853–863.
- Le-Phuoc, D. (2017), 'Operator-aware approach for boosting performance in RDF stream processing', *J. Web Sem.* **42**, 38–54.
- Le-Phuoc, D., Dao-Tran, M., Le Van, C., Le Tuan, A., Manh Nguyen Duc, T. T. N. & Hauswirth, M. (2015), Platform-agnostic execution framework towards rdf stream processing, in 'RDF Stream Processing Workshop at ESWC2015'.
- Le-Phuoc, D., Dao-Tran, M., Parreira, J. X. & Hauswirth, M. (2011), A native and adaptive approach for unified processing of linked streams and linked data, in 'ISWC'11', pp. 370–388.
- Le-Phuoc, D. & Hauswirth, M. (2018), *Linked Data for Internet of Everything*, Springer International Publishing, Cham, pp. 129–148.
- Le-Phuoc, D., Le-Tuan, A., Schiele, G. & Hauswirth, M. (2014), Querying heterogeneous personal information on the go, in 'International Semantic Web Conference', Springer, pp. 454–469.
- Le-Phuoc, D., Parreira, J. X., Reynolds, V. & Hauswirth, M. (2010), 'Rdf on the go: An rdf storage and query processor for mobile devices', *Demo, ISWC*.
- Le-Phuoc, D., Quoc, H. N. M., Van, C. L. & Hauswirth, M. (2013), Elastic and scalable processing of linked stream data in the cloud, in 'ISWC', pp. 280–297.
- Le-Tuan, A. (2016), Linked data processing for embedded devices, in 'Proceedings of the Doctoral Consortium at the 15th International Semantic Web Conference'.
- Le-Tuan, A., Hayes, C., Hauswirth, M. & Le-Phuoc, D. (2020), 'Pushing the scalability of rdf engines on iot edge devices', *Sensors* **20**(10), 2788.
- Le-Tuan, A., Hayes, C., Wylot, M. & Le-Phuoc, D. (2018), Rdf4led: an rdf engine for lightweight edge devices, in 'Proceedings of the 8th International Conference on the Internet of Things', pp. 1–8.
- Le-Tuan, A., Hingu, D., Hauswirth, M. & Le-Phuoc, D. (2019), Incorporating blockchain into rdf store at the lightweight edge devices, in 'Semantic '19'.
- Li, Y., He, B., Yang, R. J., Luo, Q. & Yi, K. (2010), 'Tree indexing on solid state drives', *Proc. VLDB Endow.* **3**(1-2), 1195–1206.

- Loseto, G., Ieva, S., Gramegna, F., Ruta, M., Scioscia, F. & Di Sciascio, E. (2016), Linked data (in low-resource) platforms: a mapping for constrained application protocol, *in* ‘International Semantic Web Conference’, Springer, pp. 131–139.
- Marr, B. (2018), ‘How much data do we create every day? the mind-blowing stats everyone should read.’, Available online: <https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#434415c260ba>. (accessed on 21 June 2020).
- Mattern, F. & Floerkemeier, C. (2010), *From the Internet of Computers to the Internet of Things*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 242–259.
- Minerva, R., Biru, A. & Rotondi, D. (2015), ‘Towards a definition of the internet of things (iot)’, *IEEE Internet Initiative* **1**, 1–86.
- Miranda, J., Mäkitalo, N., Garcia-Alonso, J., Berrocal, J., Mikkonen, T., Canal, C. & Murillo, J. M. (2015), ‘From the internet of things to the internet of people’, *IEEE Internet Computing* **19**(2), 40–47.
- Munir, A., Kansakar, P. & Khan, S. U. (2017), ‘Ifciot: Integrated fog cloud iot: A novel architectural paradigm for the future internet of things.’, *IEEE Consumer Electronics Magazine* **6**(3), 74–82.
- Nakamoto, S. (2008), ‘Bitcoin: A peer-to-peer electronic cash system’, *Decentralized Business Review* p. 21260.
- Neumann, T. & Weikum, G. (2008), ‘Rdf-3x: a risc-style engine for rdf’, *Proceedings of the VLDB Endowment* **1**(1), 647–659.
- Neumann, T. & Weikum, G. (2010), ‘The rdf-3x engine for scalable management of rdf data’, *The VLDB Journal* **19**(1), 91–113.
- Nguyen-Duc, M., Le-Tuan, A., Calbimonte, J.-P., Hauswirth, M. & Le-Phuoc, D. (2019), Autonomous rdf stream processing for iot edge devices, *in* ‘Joint International Semantic Technology Conference’, Springer, pp. 304–319.
- Nitti, M., Piloni, V., Colistra, G. & Atzori, L. (2015), ‘The virtual object as a major element of the internet of things: a survey’, *IEEE Communications Surveys & Tutorials* **18**(2), 1228–1240.
- Nori, A. (2007), Mobile and embedded databases, *in* ‘Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data’, SIGMOD ’07, ACM, New York, NY, USA, pp. 1175–1177.
- Noura, M., Atiquzzaman, M. & Gaedke, M. (2018), ‘Interoperability in internet of things: Taxonomies and open challenges’, *Mobile Networks and Applications* **24**(3), 796–809.

- Ou, Y., Härder, T. & Jin, P. (2009), Cfdc: A flash-aware replacement policy for database buffer management, *in* 'Proceedings of the Fifth International Workshop on Data Management on New Hardware', DaMoN '09, ACM, New York, NY, USA, pp. 15–20.
- Owens, A. (2011), Using low latency storage to improve RDF store performance, PhD thesis, University of Southampton.
- Owens, A., Seaborne, A., Gibbins, N. et al. (2008), 'Clustered tdb: A clustered triple store for jena'.
- Park, S.-y., Jung, D., Kang, J.-u., Kim, J.-s. & Lee, J. (2006), Cfru: A replacement algorithm for flash memory, *in* 'Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems', CASES '06, ACM, New York, NY, USA, pp. 234–241.
- Pfisterer, D., Romer, K., Bimschas, D., Kleine, O., Mietz, R., Truong, C., Hasemann, H., Kröller, A., Pagel, M., Hauswirth, M. et al. (2011), 'Spitfire: toward a semantic web of things', *IEEE Communications Magazine* **49**(11), 40–48.
- Pham, M.-D., Boncz, P. & Erling, O. (2012), S3g2: A scalable structure-correlated social graph generator, *in* 'Technology Conference on Performance Evaluation and Benchmarking', Springer, pp. 156–172.
- Pilkington, M. (2016), Blockchain technology: principles and applications, *in* 'Research handbook on digital transformations', Edward Elgar Publishing.
- Poslad, S., Middleton, S. E., Chaves, F., Tao, R., Necmioglu, O. & Bügel, U. (2015), 'A semantic iot early warning system for natural environment crisis management', *IEEE Transactions on Emerging Topics in Computing* **3**(2), 246–257.
- Quan, D., Huynh, D. & Karger, D. R. (2003), Haystack: A platform for authoring end user semantic web applications, *in* 'International Semantic Web Conference', Springer, pp. 738–753.
- Rekimoto, J. (1999), Timescape: a time machine for the desktop environment, *in* 'CHI'99 extended abstracts on Human factors in computing systems', pp. 180–181.
- Ren, X. & Curé, O. (2017), Strider: A hybrid adaptive distributed rdf stream processing engine, *in* 'The Semantic Web – ISWC 2017'.
- Rhayem, A., Mhiri, M. B. A. & Gargouri, F. (2020), 'Semantic web technologies for the internet of things: Systematic literature review', *Internet of Things* p. 100206.
- Sakr, S., Wylot, M., Mutharaju, R., Le Phuoc, D. & Fundulaki, I. (2018), *Processing of RDF Stream Data*, Springer International Publishing, Cham, pp. 85–108.
- Satyanarayanan, M. (2017), 'The emergence of edge computing', *Computer* **50**(1), 30–39.

- Sauermann, L. (2003), The gnowsis: using semantic web technologies to build a semantic desktop, PhD thesis, Technical University of Vienna.
- Schandl, B. & Zander, S. (2009), ‘Adaptive rdf graph replication for mobile semantic web applications’, *Ubiquitous Computing and Communication Journal (Special Issue on Managing Data with Mobile Devices)*.
- Seaborne, A. (2010), ‘Jena, a semantic web framework’.
- Segars, S. (2017), ‘Enabling mass IoT connectivity as Arm partners ship 100 billion chips.’, Available online: <https://community.arm.com/iot/b/blog/posts/enablin-g-mass-iot-connectivity-as-arm-partners-ship-100-billion-chips>. (accessed on 21 June 2020).
- Serrano, M., Gyrard, A., Boniface, M., Grace, P., Georgantas, N., Agarwal, R., Barnagu, P., Carrez, F., Almeida, B., Teixeira, T. et al. (2017), ‘Cross-domain interoperability using federated interoperable semantic iot/cloud testbeds and applications: The fiesta-iot approach’.
- Seydoux, N., Drira, K., Hernandez, N. & Monteil, T. (2017), ‘Capturing the contributions of the semantic web to the iot: a unifying vision’, *arXiv preprint arXiv:1709.03576*.
- Shi, F., Li, Q., Zhu, T. & Ning, H. (2018), ‘A survey of data semantization in internet of things’, *Sensors* **18**(1), 313.
- Shi, W., Cao, J., Zhang, Q., Li, Y. & Xu, L. (2016), ‘Edge computing: Vision and challenges’, *IEEE Internet of Things Journal* **3**(5), 637–646.
- Smith, B. (2008), ‘Arm and intel battle over the mobile chip’s future’, *Computer*.
- Soldatos, J. et al. (2015), Openiot: Open source internet-of-things in the cloud, in ‘Interoperability and open-source solutions for the internet of things’, Springer.
- Soursos, S., Žarko, I. P., Zwickl, P., Gojmerac, I., Bianchi, G. & Carrozzo, G. (2016), Towards the cross-domain interoperability of iot platforms, in ‘2016 European conference on networks and communications (EuCNC)’, IEEE, pp. 398–402.
- Stephen, H. & Nicholas, G. (2003), 3store: Efficient bulk rdf storage, in ‘Proceedings of the First International Workshop on Practical and Scalable Semantic Systems’.
- Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C. & Reynolds, D. (2008), Sparql basic graph pattern optimization using selectivity estimation, in ‘Proceedings of the 17th international conference on World Wide Web’, pp. 595–604.
- Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O’Neil, E., O’Neil, P., Rasin, A., Tran, N. & Zdonik, S. (2005), C-store: A column-oriented dbms, in ‘Proceedings of the 31st International Conference on Very Large Data Bases’, VLDB ’05, VLDB Endowment, pp. 553–564.

- Studer, R., Benjamins, V. R. & Fensel, D. (1998), ‘Knowledge engineering: principles and methods’, *Data & knowledge engineering* **25**(1-2), 161–197.
- Su, X., Rieki, J., Nurminen, J. K., Nieminen, J. & Koskimies, M. (2015), ‘Adding semantics to internet of things’, *Concurrency and Computation: Practice and Experience* **27**(8), 1844–1860.
- Swan, M. (2015), *Blockchain: Blueprint for a new economy*, ” O’Reilly Media, Inc.”.
- Szabo, N. (1997), ‘Formalizing and securing relationships on public networks’, *First Monday* .
- Tommasini, R., Calvaresi, D. & Calbimonte, J.-P. (2019), Stream reasoning agents: Blue sky ideas track, in ‘AAMAS’, pp. 1664–1680.
- Tommasini, R., Sedira, Y. A., Dell’Aglia, D., Balduini, M., Ali, M. I., Le Phuoc, D., Della Valle, E. & Calbimonte, J.-P. (2018), Vocals: Vocabulary and catalog of linked streams, in ‘International Semantic Web Conference’.
- Tramp, S., Frischmuth, P., Arndt, N., Ermilov, T. & Auer, S. (2011), Weaving a distributed, semantic social network for mobile users, in ‘Extended Semantic Web Conference’, Springer, pp. 200–214.
- Truong, H. & Dustdar, S. (2015), ‘Principles for engineering iot cloud systems’, *IEEE Cloud Computing* **2**(2), 68–76.
- Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V. & Boncz, P. (2012), Heuristics-based query optimisation for sparql, in ‘Proceedings of the 15th International Conference on Extending Database Technology’, pp. 324–335.
- Tummarello, G., Morbidoni, C., Bachmann-Gmür, R. & Erling, O. (2007), Rdfsync: efficient remote synchronization of rdf models, in ‘The Semantic Web’, Springer, pp. 537–551.
- Ullman, J. D., Garcia-Molina, H. & Widom, J. (2001), *Database systems: the complete book*, Vol. 2, Prentice Hall Upper Saddle River.
- Vermesan, O., Friess, P., Guillemin, P., Gusmeroli, S., Sundmaeker, H., Bassi, A., Jubert, I. S., Mazura, M., Harrison, M., Eisenhauer, M. et al. (2011), ‘Internet of things strategic research roadmap’, *Internet of Things-Global Technological and Societal Trends* **1**, 9–52.
- Walport, M. et al. (2016), ‘Distributed ledger technology: Beyond blockchain’, *UK Government Office for Science* **1**, 1–88.
- Weippl, E., Klemen, M., Fenz, S., Ekelhart, A. & Tjoa, A. (2007), The semantic desktop: A semantic personal information management system based on rdf and topic maps, in M. Collard, ed., ‘Ontologies-Based Databases and Information Systems’, Vol. 4623 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 135–151.

- Weiss, C., Karras, P. & Bernstein, A. (2008), ‘Hexastore: Sextuple indexing for semantic web data management’, *Proc. VLDB Endow.* **1**(1), 1008–1019.
- Wilkinson, K. (2006), ‘Jena property table implementation’.
- Wilkinson, K., Sayers, C., Kuno, H. A., Reynolds, D. et al. (2003), Efficient rdf storage and retrieval in jena2., *in* ‘SWDB’, Vol. 3, Citeseer, pp. 131–150.
- Wood, G. et al. (2014), ‘Ethereum: A secure decentralised generalised transaction ledger’.
- Wylot, M. & Cudré-Mauroux, P. (2015), ‘Diplocloud: Efficient and scalable management of rdf data in the cloud’, *IEEE Transactions on Knowledge and Data Engineering* **28**(3), 659–674.
- Xu, W., Curé, O. & Calvez, P. (2020), ‘Succinctedge: a succinct rdf store for edge computing’, *Proceedings of the VLDB Endowment* **13**(12), 2857–2860.
- Xu, X., Liu, Q., Luo, Y., Peng, K., Zhang, X., Meng, S. & Qi, L. (2019), ‘A computation offloading method over big data for iot-enabled cloud-edge computing’, *Future Generation Computer Systems* **95**, 522–533.
- Yang, X., Wang, T., Ren, X. & Yu, W. (2017), ‘Survey on improving data utility in differentially private sequential data publishing’, *IEEE Transactions on Big Data* pp. 1–1.
- Yang, Y., Cao, Q. & Jiang, H. (2019), ‘Edgedb: An efficient time-series database for edge computing’, *IEEE Access* **7**, 142295–142307.
- Yu, W., Liang, F., He, X., Hatcher, W. G., Lu, C., Lin, J. & Yang, X. (2018), ‘A survey on the edge computing for the internet of things’, *IEEE Access* **6**, 6900–6919.
- Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W. & Liu, L. (2013), ‘Triplebit: a fast and compact system for large scale rdf data’, *Proceedings of the VLDB Endowment* **6**(7), 517–528.
- Zamora-Izquierdo, M. A., Santa, J., Martínez, J. A., Martínez, V. & Skarmeta, A. F. (2019), ‘Smart farming iot platform based on edge and cloud computing’, *Biosystems engineering* **177**, 4–17.
- Zhang, B., Mor, N., Kolb, J., Chan, D. S., Goyal, N., Lutz, K., Allman, E., Wawrzynek, J., Lee, E. & Kubiawicz, J. (2015), The cloud is not enough: Saving iot from the cloud, *in* ‘Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing’, HotCloud’15, USENIX Association, Berkeley, CA, USA, pp. 21–21.
- Zhang, F., Wang, K., Li, Z. & Cheng, J. (2019), ‘Temporal data representation and querying based on rdf’, *IEEE Access* **7**, 85000–85023.

- Zhang, S., Zeng, W., Yen, I.-L. & Bastani, F. B. (2019), Semantically enhanced time series databases in iot-edge-cloud infrastructure, *in* ‘2019 IEEE 19th international symposium on high assurance systems engineering (HASE)’, IEEE, pp. 25–32.
- Zhou, J., Leppanen, T., Harjula, E., Ylianttila, M., Ojala, T., Yu, C., Jin, H. & Yang, L. T. (2013), Cloudthings: A common architecture for integrating the internet of things with cloud computing, *in* ‘Proceedings of the 2013 IEEE 17th International Conference on Computer Supported Cooperative Work in Design (CSCWD)’, pp. 651–657.
- Zou, L., Özsu, M. T., Chen, L., Shen, X., Huang, R. & Zhao, D. (2014), ‘gstore: a graph-based sparql query engine’, *The VLDB journal* **23**(4), 565–590.