UNIVERSITY OF SOUTHAMPTON

Faculty of Physical Sciences and Engineering
Electronics and Computer Science
Web and Internet Science

Efficient Querying for Analytics on Internet of Things Databases and Streams

by

Eugene Siow

Thesis for the degree of Doctor of Philosophy

February 2018

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

Web and Internet Science

Doctor of Philosophy

Efficient Querying for Analytics on Internet of Things Databases and Streams

by Eugene Siow

This thesis is concerned with the development of efficient methods for managing contextualised time-series data and event streams produced by the Internet of Things (IoT) so that both historical and real-time information can be utilised to generate value within analytical applications.

From a database systems perspective, two conflicting challenges motivate this research, interoperability and performance. IoT applications integrating streams of time-series data from heterogeneous IoT agents require a level of semantic interoperability. This semantic interoperability can be achieved with a common flexible data model that represents both metadata and data. However, applications might also have time constraints or require processing to be performed on large volumes of historical and streaming time-series data, possibly on resource-constrained platforms, without significant delay. Obtaining good performance is complicated by the complexity of the data model.

In the first part of the thesis, a graph data model is shown to support the representation of metadata and data that various research and standard bodies are working towards, while the 'volume' of IoT data is shown to exhibit flat, wide and numerical characteristics. A three step abstraction is defined to reconcile queries on the graph model with efficient underlying storage by query translation. This storage is iteratively improved to exploit the character of time-series IoT data, achieving orders of magnitude performance improvement over state-of-the-art commercial, open-source and research databases.

The second part of the thesis extends this abstraction to efficiently process real-time IoT streams continuously and proposes an infrastructure for fog computing that shows how resource-constrained platforms close to source IoT agents can co-operatively orchestrate stream processing. The main contributions of this thesis are therefore, i) a novel interoperable and performant abstraction for querying IoT graph representations, ii) high performance historical, streaming and fog computing time-series database implementations and iii) analytical applications and platforms built on this abstraction that act as practical models for the socio-technical development of the IoT.

Contents

Nomenclature

Common Abbreviations

AoT	Array of Things
API	Application Programming Interface
ARIMA	Auto-RegressIve Moving Average
BGP	Basic Graph Pattern
CEP	Complex Event Processing
CoAP	Constrained Application Protocol
COW	Copy-On-Write
CPU	Central Processing Unit
CQELS	Continuous Query Evaluation over Linked Streams
DBMS	DataBase Management System
EPL	Event Processing Language
ES	Elastic Search
FPC	Floating Point Compression
H2M	Human-to-Machine
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IQR	Inter-Quartile Range
IRI	Internationalised Resource Identifier
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LSD	Linked Sensor Data
LSM-Tree	Log-Structured Merge-Tree
M2M	Machine-to-Machine
MAD	Median Absolute Deviation
MQTT	Message Queuing Telemetry Transport
OH	Open Humans
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
OS	Operating System

OWL	Web Ontology Language
PIOTRe	Personal Internet Of Things Repository
QRB	Quantum Re-ordering Buffer
RAM	Random Access Memory
RDF	Resource Description Framework
REST	REpresentational State Transfer
RLE	Run Length Encoding
RPi	Raspberry Pi
RSP	RDF Stream Processing
RSP-QL	RSP Query Language
SAO	Stream Annotation Ontology
S2S	SPARQL-to-SQL or SPARQL-to-Stream
S2SML	S2S Mapping Language
SAREF	Smart Appliances REFerence Ontology
SIoT	Social Internet of Things
SMA	Simple Moving Average
SPARQL	SPARQL Protocol and RDF Query Language
SPO	Subject, Predicate, Object
SQL	Structured Query Language
SSN	Semantic Sensor Network
SSTable	Sorted String Table
SSW	Semantic Sensor Web
SWIBRE	SWappable Interface for BGP Resolution
SWIPE	SWappable Iterator for oPErations
SWoT	Social Web of Things
TDB	Tuple DataBase
TP	Time-Partitioned
TritanDB	Time-series Rapid Internet of Things ANalytics DataBase
TrTable	TritanDB Table Data Structure
UUID	Universally Unique IDentifier
W3C	World Wide Web Consortium
WoT	Web of Things
XML	eXtensible Markup Language

Common Symbols

.	Append Operator
α	Distribution Function, Decision
B	Blank Node
b	Binding, Block, Broker
$\mathbb{B}$	Set of Bindings

C	Set of Columns, Connections, Consensus
c	Column, Connection, Evaluation
Δ	Difference Between Values, Time Bounds
δ	Difference Between Values, Discovery Function
E	Set of Edges
ϵ	Eywa Network
F	Faux Node
G	BGP Operator, Graph
g	Gizmo2 Setup
Γ	Set of Streams
γ	Group/Aggregation Operator, Stream
I	IRI Node
$\bowtie$	Join Operator
$\ltimes$	Left Join Operator
L	Literal Node, Liking
λ	Conversion Function, Connection
μ	Map-Match-Operate Function, Topic
M	Model Tier, Set of Mappings, Topics
m	Mapping, Message, Model
N	Nodes
o	Object
ω, ϖ	Work Function, Task
p	Publish Function, Predicate, Pi Setup
Π	Projection Operator
Q	Quantum Re-ordering Buffer, Set of Queries
q	Query, Quotient, Quantum, Sub-query
R	Set of Relations, Result Set
r	Result, Remainder, Ratio
ρ	Extend Operator, Run-length, Relation Function
S	Shelburne Dataset
s	Source, Subscribe Function, Subject, Server Setup
σ	Selection/Filter Operator
T	Database Tier, Sink, Taxi Dataset
t	Time-series
τ	Set of Timestamps, Client, Time Horizon, Transmission
θ	Threshold
U_{id}	UUID Placeholder in Faux Node
$\cup$	Union Operator
υ	Set of Variables
V	Set of Vertices
W	Window

List of Figures

List of Tables

Listings

Declaration of Authorship

I, Eugene Siow, declare that this thesis titled, 'Efficient Querying for Analytics on Internet of Things Databases and Streams' and the work presented in it are my own and have been generated by me as a result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: Siow et al. (2016a), Siow et al. (2016b), Siow et al. (2016c) and Siow et al. (2017).

Signed: ____________________________________

Date: ____________________________________

Acknowledgements

For their inspirational mentoring, help and support through the course of my PhD, I would like to thank my supervisors, Dr Thanassis Tiropanis and Professor Dame Wendy Hall. The advice, feedback and learning opportunities they have afforded me have been invaluable in leading towards the production of this thesis and I am extremely grateful. Additionally, I would like to thank Professor Kirk Martinez and Dr Nicholas Harris for their constructive feedback during my first year and upgrade vivas which has helped in the writing of a more precise and refined thesis.

This thesis has been made possible by the support of my wife, family and friends who have constantly encouraged and affirmed my efforts. I am also extremely thankful to the University of Southampton and the EU VOICE project for funding my studentship. My employer DSO National Laboratories has also been exceptionally understanding in granting me time off from work to explore my research directions within this doctorate.

Finally, I would like to reserve special thanks for my colleagues Dr Xin Wang, Dr Aastha Madaan and Dr Ramine Tinati for all the thought-provoking exchanges we have had in the lab and on conference travel which has helped mould me into a better researcher.

> “It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment."
>
> — *Carl Friedrich Gauss*

Chapter 1

Introduction

> "Who controls the past controls the future. Who controls the present controls the past."
>
> — *George Orwell*

A smart home. A robot vacuum cleaner whirrs to a stop as a car pulls up the driveway. Lights in the hallway brighten as a man enters the house. The temperature has been subtly adjusted prior to his arrival. The man issues a voice command to a home assistant console to order a pizza for dinner. The assistant acknowledges the request and reminds him that the milk in the fridge is expiring soon. Behind the scenes, the robot vacuum, lights, motion sensors, heating, fridge and home assistant have all been exchanging signals over a wireless network and cooperating to understand the man's intentions and respond to them.

A factory like many others. A product travels down a production line and interacts with equipment at various stages. Each piece of equipment has been adjusted based on ambient conditions. A maintenance robot predictively replaces a piece of equipment reaching the end of its life. The product reaches an automatic quality control machine that ensures the product meets a set of criteria based on a specification. A self-directed vehicle picks up this batch of products at the end of the production line and delivers them to the nearby warehouse.

A smart city. An autonomous car moves as part of a non-stop stream of vehicles on the main road of the city. There are no control signals as vehicles work together efficiently to maintain the continuous flow of traffic. The car branches off into a parking lot and enters the empty designated parking space.

Each of these three situations have been inspired by the Internet of Things (IoT) applications described by Manyika et al. (2015). Each situation, has in common, a large ensemble of independent agents exchanging local information, both current and historical, to seamlessly produce a global effect creating value for individuals and society. This interconnected network of agents forms part of what has come under the umbrella of the IoT. Despite the wide range of applicable domains and interest from both academia and industry, there has been little convergence when interoperability between these agents and the integration of information produced is concerned.

Amidst the lack of consensus on standard formats and protocols however, information can be represented in a form whose meaning is independent of the agent generating it or the application using it, sometimes termed as semantic interoperability (Tolk et al., 2007). This in turn stands to facilitate the development of new types of applications and services within the IoT that have a more comprehensive understanding and are able to derive quantitative insights from the physical world across application domains.

This thesis is concerned with how information produced by IoT agents, from a database systems perspective, can be i) *represented,* ii) *stored and* iii) *utilised* in applications. These concerns are systematically addressed. Firstly, an *information representation for semantic interoperability* in the IoT, the graph model, is determined. A survey of research and standard bodies' proposals and literature for semantic interoperability showed that each representation could be abstracted as a graph model that allows for both contextual metadata and data to be represented. Next, an efficient means to *store and retrieve information* with this representation, an abstraction of a query translation process, is iteratively improved for databases then streams. This abstraction allows graph metadata and time-series data to each be stored efficiently, exploiting their physical characteristics and the nature of hardware, yet be seamlessly queried together as a graph. Finally, *IoT applications and platforms that build* on this efficient infrastructure are realised. In the process of developing these solutions, a deeper understanding of IoT systems and their information characteristics is obtained through a series of scientific studies.

The contribution of this thesis is grounded in the formal definition of solutions that inform practical database implementations which are then experimented on using published public datasets and benchmarks to compare against state-of-the-art alternatives.

In order to contextualise this thesis the next three sections follow a flow of information in the IoT from its production by agents (Section 1.1) to storage, retrieval and processing within databases (Section 1.2) and finally to its usage within applications (Section 1.3). This follows closely the phases of the IoT data lifecycle proposed by Abu-Elkheir et al. (2013), namely, collection, management and processing. The concerns of this thesis are then formulated as research questions in Section 1.4, followed by Section 1.5 which highlights specific intellectual contributions. Then, an outline of the remainder of the thesis is presented in Section 1.6 followed by a list of publications and software produced for this thesis in Section 1.7.

1.1 Information in the IoT

Data is the product of observation. IoT agents possess sensors that observe 'properties of objects, events and their environment' and produce symbols called data in the information hierarchy (Rowley, 2007).

Information is data with a context. The context of data is represented in metadata, data that describes this data. IoT sensor data is often only considered to be useful in the presence of metadata like a description of the IoT agent, sensor location, sensor type and unit of measurement. The readings produced, as Milenkovic (2015) argued, would otherwise be a meaningless set of values to another agent or application.

Data collected at points in time from a sensor of an IoT agent, form a sequence called a time-series.

1.2 Database and Stream Processing in the IoT

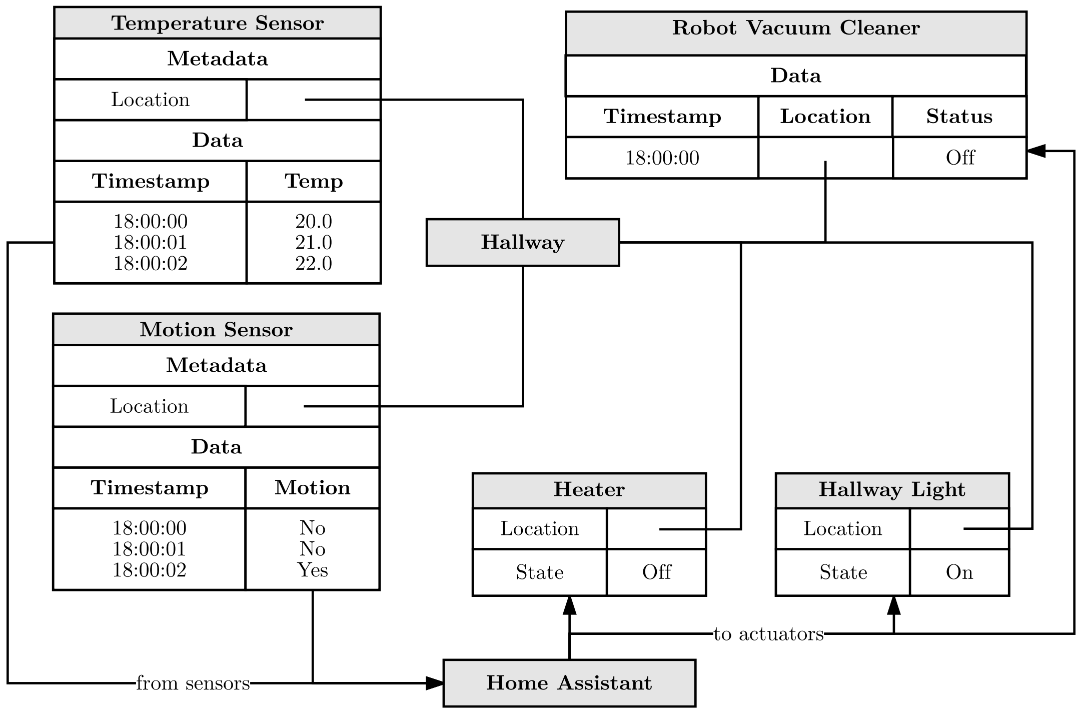


Figure 1.1: Sensors, Actuators, Data and Metadata in a Smart Home Scenario

In the earlier smart home scenario, the motion sensor and temperature sensor in a hallway each produced a time-series stream of sensor data as shown in Figure 1.1. A home assistant application processed each stream and actuated the lights when motion was detected two seconds after six in the evening and shut off the heater when an optimal temperature of 22.0 degrees celsius was reached. All the sensors and actuators in this example were located in the hallway and the ‘location’ metadata property gave context to the data recorded. Historical time-series data from the motion sensor within a database was retrieved by the home assistant to create a schedule for the robot vacuum cleaner to finish cleaning the house before six in the evening to avoid getting in the way of the home owner.

This scenario illustrates two requirements at this point in the IoT information flow. The first of which is that both historical time-series data and real-time streaming data need to be efficiently stored, retrieved and processed so that applications can perform automation tasks with minimal delay. The second requirement is the provision of storage and retrieval for a representation of both metadata and data for use in IoT applications with the eventual goal being that of semantic interoperability.

1.3 Applications and Analytics in the IoT

Applications have been known to drive the adoption of technology, just as the iPhone was the catalyst for smart phone and data service adoption (West and Mace, 2010). In fact, applications in the IoT have even greater potential. A chief executive of Twitter once described it as, a reinvention of the town square but with television, due to its utility as a real-time news source quickly disseminating information to a network. Hermida (2010) called this broad, asynchronous, always-on awareness system, ambient journalism. While studies have shown 317 million active users of Twitter monthly in 2016 (Statista, 2017b), they have put estimates of the number of connected IoT agents in the same year at 22.9 billion (Statista, 2017a). IoT agents with the capability to be always-on, deployed in the harshest conditions and most far flung regions certainly make a case for a wider reach and larger spatio-temporal observation space than Twitter’s ambient journalism. Hence, the collective understanding of the physical world, quantitative insights that can be derived and range of use cases is limited only by the applications realised for the IoT.

Analytics is the process of deriving knowledge, know-how that ‘makes possible the transformation of information into instructions’ (Rowley, 2007). Combining the wide observation space of the IoT with the output of analytics in the form of insights drawn from information, helps empower a class of applications that can optimise and automate tasks which are time-consuming, repetitive or difficult for individuals, businesses and the government. Hence, the responsibility of database systems and stream processing at this stage is to ensure that analytical applications can perform analytics on the interoperable representation of information efficiently.

An emerging paradigm that presents a new set of requirements for databases, streams and applications in the IoT is that of fog computing. Chiang et al. (2017) described fog computing as an ‘end-to-end horizontal architecture’ that distributes all of storage, processing and networking functions closer to users, in a continuum from the cloud to

IoT agents called Things. Bonomi et al. (2012) argued that the wide-spread distribution of Things and run-time bounds or limited connectivity in certain use cases make such an architecture necessary. Database systems supporting applications across more diverse hardware specifications is therefore also a point of consideration.

1.4 Research Questions

The flow of information from production to usage in the IoT, as described in the previous sections, forms the premise for an overall database systems research question:

> "How can database systems support the efficient storage and retrieval of a semantically interoperable representation of data and metadata, both historical and real-time, from the Internet of Things, for analytical applications?"

The literature surrounding database systems contains support for many different models of representation and optimisations for storing and retrieving such models. IoT research is multi-faceted and spreads out in many directions, one of which is the collecting and processing of big data for the creation of knowledge which a semantically interoperable representation is essential for (Stankovic, 2014). There has however only been some tentative effort to bring this research together which is elaborated on in Chapter 2 that sets out the background literature in detail. To pursue the original research question, three additional sub-questions, developed in the following paragraph, are asked.

The first two sub-questions are "Are there unique characteristics in IoT time-series data?" and "Can a representation of IoT metadata and time-series data that enables interoperability be found?". These sub-questions motivate the study of IoT data, semantic interoperability in the IoT and other IoT literature. Having established that there exists a set of characteristics and a semantically interoperable representation for the IoT, the next question is then "How can database and stream processing systems utilise these characteristics to efficiently store and query this representation model for analytics across diverse hardware?". The answering of each sub-question proceeds the argument of the original research question, the output of which is shown in the next section listing the main research contributions.

1.5 Research Contributions

This thesis focuses on optimising database systems for the efficient and interoperable querying of information from the IoT for analytical usage within applications. A summary of the intellectual research contributions are as follows:

- A study of IoT survey literature that advises a categorisation of IoT research with follow-up studies on advanced applications, architectures, technologies, database and stream processing systems and analytics (Chapter 2).
- A study of public IoT time-series data across application domains, showing the flat, wide and numerical nature of data streams that also display varying periodicity (Chapter 3).
- A study of metadata models for the IoT from different domains and perspectives showing there is a common underlying graph or tree representation (Chapter 3).
- A study of metadata and data ratios in graph model IoT data of the Resource Description Framework (RDF) format, highlighting the issue of metadata expansion (Chapter 3).
- An abstraction for query translation enabling graph queries to be executed across graph model metadata and time-series data. An implementation using in-memory RDF storage with a relational database demonstrates comparatively better aggregation support, fewer joins and greater storage and retrieval efficency than a range of RDF stores and translation engines (Chapter 4).
- A new compressed, time-reordering, performance-optimised data structure for storing IoT time-series data, TrTables (Chapter 5).
- An evaluation of compression algorithms and data structures for time-series data advising the design of a time-series database, TritanDB (Chapter 5).
- An evaluation of multiple time-series, relational and NoSQL databases including TritanDB for IoT time-series data across cloud and Thing hardware (Chapter 5).
- A method for translating and executing continuous queries on a stream of RDF graph model data and an infrastructure for distributing this over a fog computing network (Chapter 6).
- A set of applications and analytics applying the methods in this thesis, for performant and interoperable IoT storage and querying, to personal repositories, the Social Web of Things and time-series analytics (Chapter 7).

1.6 Thesis Outline

This introduction chapter has provided the context from which this research originated and its direction and contribution. This section explains the remainder of the thesis which is divided into seven components. The first component (Chapter 2) covers a study of the background literature within the fields of IoT and database systems research relevant to this thesis.

The next four components address the research questions proposed in this thesis. Chapter 3 investigates the characteristics of IoT time-series and seeks a representation to model IoT metadata and IoT time-series data through a series of studies. Chapter 4 moves on to propose and realise a solution to efficiently store and query both time-series data and metadata within a flexible representation model, targeting greater semantic interoperability. Chapter 5 then iterates on the storage and querying of time-series data by an understanding and sensitivity towards physical hardware characteristics and IoT data characteristics. Next, Chapter 6 extends optimisations to stream processing, and stream processing within distributed fog computing systems. Each of these components follow a common method whereby understanding is achieved through studies, which then advises the formalisation of solutions guided by the IoT use case. Design and implementation follows which is benchmarked against the state-of-the-art within IoT scenarios and utilising public data for reproducibility.

Finally, Chapter 7 describes IoT applications and analytics that reflect and demonstrate the impact of the database systems research within this thesis on wider IoT themes. Chapter 8 concludes by analysing the limitations of this work and the potential for further study, before final remarks present a summary of this thesis.

1.7 Publications and Software

The work presented in this thesis has resulted in several publications, datasets, submissions under review and software projects presented as follows:

Published work

Eugene Siow, Thanassis Tiropanis, and Wendy Hall (2016a). Interoperable & Efficient: Linked Data for the Internet of Things. In *Proceedings of the 3rd International Conference on Internet Science*, 2016.

Eugene Siow, Thanassis Tiropanis, and Wendy Hall (2016b). PIOTRe: Personal Internet of Things Repository. In *Proceedings of the 15th International Semantic Web Conference P&D*, 2016.

Eugene Siow, Thanassis Tiropanis, and Wendy Hall (2016c). SPARQL-to-SQL on Internet of Things Databases and Streams. In *Proceedings of the 15th International Semantic Web Conference*, 2016.

Eugene Siow, Thanassis Tiropanis, and Wendy Hall (2017). Ewya: An Interoperable Fog Computing Infrastructure with RDF Stream Processing. In *Proceedings of the 4th International Conference on Internet Science*, 2017.

A note is made at the start of each chapter in this thesis of the sections which contain the above published work and the relevant citations.

Published Datasets

Eugene Siow (2017). Cross-IoT Study of the Characteristics of IoT Schemata and Data Streams. *doi:10.5258/SOTON/D0202.*

Eugene Siow (2016). Dweet.io IoT Device Schema. *doi:10.5258/SOTON/D0076.*

Submissions Under Review or Revision

Eugene Siow, Thanassis Tiropanis, Wendy Hall (2017). Analytics for the Internet of Things: A Survey. *ACM Computing Surveys*, Submitted may 2016, Revised sep 2017.

Eugene Siow, Thanassis Tiropanis, Xin Wang, Wendy Hall (2017). TritanDB: Time-series Rapid Internet of Things Analytics. *ACM Transactions on Database Systems*, Submitted sep 2017.

Eugene Siow, Thanassis Tiropanis, Wendy Hall (2017). A Decentralised Social Web of Things. *IEEE Pervasive Computing*, Submitted nov 2017.

Software Projects

- **DistributedFog.com** (Chapter 2): An educational resource describing the various technologies within the IoT landscape (`http://distributedfog.com`).
- **S2S** (Sections 4.2.4 and 6.1.1): An implementation of the Map-Match-Operate abstraction proposed in this thesis with support for graph query translation from SPARQL queries to execution on relational databases or streams using the S2SML data model mapping language (`https://github.com/eugenesiow/sparql2sql`).
- **TritanDB** (Chapter 5): A time-series database for IoT analytics, implementing a rich graph data model with underlying time-partitioned block compression using TrTables and utilising Map-Match-Operate (`http://tritandb.com`).
- **Eywa** (Section 6.2): A fog computing infrastructure that provides control and data planes for RDF stream processing to be performed on resource-constrained client, source and broker nodes (`https://github.com/eugenesiow/eywa`).
- **PIOTRe** (Section 7.1): A Personal IoT Repository based on S2S that consists of a sink for IoT data streams and historical time-series that allows applications to be built with it (`https://github.com/eugenesiow/PIOTRe-web`).
- **Hubber.space** (Section 7.3.4): An experimental sandbox application developed to realise the core social network functions of a decentralised Social Web of Things which integrates a lightweight and scalable publish-subscribe broker to control information flow among PIOTRe devices (`http://hubber.space`).

Chapter 2

Background

"If I have seen further than others, it is by standing upon the shoulders of giants."

— *Sir Isaac Newton*

This chapter details an overview of the literature surrounding both Internet of Things (IoT) research and that of database systems which have increasingly been driven by the need to store and process big data. Each research area has grown tangentially without much overlap and hence this chapter seeks to establish common ground, acknowledge the tentative efforts taken to integrate research and surface the gaps that motivated the research within this thesis. The chapter starts by defining the IoT and introducing a means of categorising the broad, multi-directional research through an analysis of various survey papers, building up to studies of IoT architectures, the layers within those architectures and the technology building blocks of interest within those layers in Section 2.1. The chapter then moves on to examine database and stream processing system literature in the IoT context in Section 2.2. The goal of efficient and interoperable data management is to act as an enabler for advanced applications and services through analytics, which is the focus of Section 2.3. Finally, Section 2.4 summarises literature involving the emerging IoT research area of fog computing.

2.1 Definition and Categorisation the IoT

The vision of the IoT was defined by the International Telecommunication Union (2012) as a 'global infrastructure' that interconnects Things with 'interoperable information and communication technologies' to enable 'advanced services' that have 'technological and societal' impact. The report from the United Kingdom Government Office for Science agreed and added that the purpose of interconnecting these 'everyday objects' is so that data can be shared (Walport, 2014). The same points were also observed in the definition by Vermesan and Friess (2014) and can be summarised as follows. The IoT is:

- an infrastructure that exists at a global scale,
- made up of IoT agents called Things,
- interconnected so that data can be shared and
- has the potential for technological and societal impact through advanced applications and services.

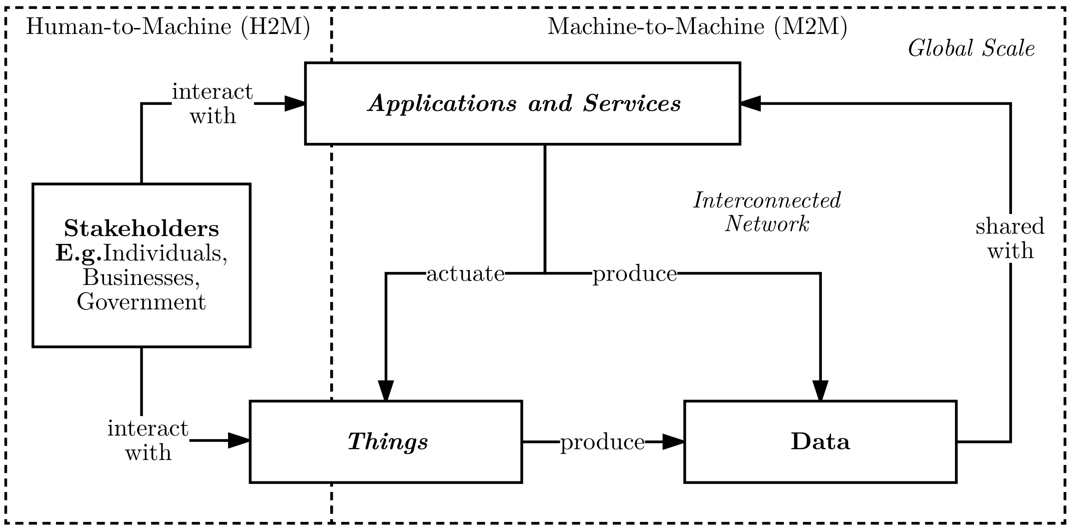


FIGURE 2.1: Visualisation of the IoT Definition

Figure 2.1 visualises the definition of the IoT and the interactions between its components. Various stakeholders in the IoT, for example individuals, businesses or the government, interact with IoT agents called Things, or applications and services through human-to-machine (H2M) interfaces. Both Things and applications produce data that is shared through the interconnected network and this exchange is termed Machine-to-Machine (M2M) interaction. Applications and services, possibly across multiple application domains, act on data to actuate Things or produce more data which could contain insights that lead to various positive societal impacts, such as greater productivity with the smart automation of time-consuming tasks. All of these interactions can occur on a global scale. The flow of information in the IoT, introduced in Chapter 1, describes the M2M flow of data produced by Things to applications and services.

This thesis is particularly concerned with the M2M interaction of IoT Things, data and applications with the goal of improving how the data produced can be represented, stored and used by applications and services in both an interoperable and efficient manner. As such, this section moves forward to cover the broad spectrum of IoT research through a sensible means of categorising the literature that will be introduced in Section 2.1.1.

2.1.1 IoT Applications, Platforms and Building Blocks Categorisation

Given the broadness of the IoT landscape, a sensible way of categorising literature is the one used by IoT venture capitalist, Turck (2016), who categorised IoT stakeholders (businesses and organisations) into those involved with verticals, IoT applications, those involved with horizontals, IoT platforms, and those involved with the building blocks that make up these. Research literature can be categorised similarly as verticals, horizontals and building blocks but with a focus on the underlying technologies, original research and state-of-the-art applications instead of stakeholders.

Year	Survey Reference	A	P	B	Description
2010	Atzori et al. (2010)	✓	✓	✓	IoT Survey
2011	Bandyopadhyay et al. (2011)		✓	✓	Middleware for IoT
2012	Miorandi et al. (2012)	✓	✓	✓	IoT Vision, Apps, Challenges
	Barnaghi et al. (2012)		✓	✓	Semantic Interoperability
2013	Vermesan and Friess (2013)	✓	✓	✓	IoT Ecosystem
2014	Perera et al. (2014)		✓	✓	Context Aware Computing
	Zanella et al. (2014)	✓	✓	✓	IoT for Smart Cities
	Xu et al. (2014)	✓	✓	✓	IoT in Industries
	Stankovic (2014)		✓		IoT Challenges
2015	Al-Fuqaha et al. (2015)	✓	✓	✓	Protocols, Challenges, Apps
	Granjal et al. (2015)		✓	✓	Security Protocols, Challenges
2016	Ray (2016)	✓	✓	✓	IoT Architectures
	Razzaque et al. (2016)		✓		Middleware for IoT
2017	Lin et al. (2017)	✓	✓	✓	Fog Computing
	Farahzadia et al. (2017)		✓		Middleware for Cloud IoT
	Sethi and Sarangi (2017)	✓	✓	✓	IoT Architectures, Apps

Legend: A = Applications, P = Platforms, B = Building Blocks

Table 2.1: Chronological Summary of Previous IoT Surveys Categorised by Applications, Platforms and Building Blocks Research

Table 2.1 presents a sequence of IoT surveys in chronological order and categorised according to whether they studied IoT applications, platforms or building blocks or a combination of the three. The progression of surveys showed an increasing amount of interest in a variety of topics related to the IoT since its inception when the term was coined by Ashton (2009).

The surveys also covered a range of research topics from defining the general vision, challenges and ecosystem of the IoT (Atzori et al., 2010; Miorandi et al., 2012; Vermesan and Friess, 2013; Stankovic, 2014; Al-Fuqaha et al., 2015; Ray, 2016; Sethi and Sarangi, 2017) to tackling specific domains like Smart Cities (Zanella et al., 2014) or industries (Xu et al., 2014) to surveying specific technologies and platforms like middleware (Bandyopadhyay et al., 2011; Razzaque et al., 2016; Farahzadia et al., 2017), semantic interoperability technology (Barnaghi et al., 2012), context aware computing (Perera et al., 2014) or fog computing (Lin et al., 2017), to addressing specific issues like security (Granjal et al., 2015). They also showed a sustained amount of interest and activity in each segment of the categorisation.

2.1.2 IoT Applications: Application Areas, Domains and Themes

A range of application areas were first elicited from all the surveys describing IoT applications. To better understand the 'technological and societal impact' of each application area, they were categorised according to their impact towards the three drivers of sustainable development, namely, society, economy and environment. The goal of sustainable development, which is used for analysing medium to long term development issues at a large scale, is usually defined within literature as the intersection of the society, the economy and the environment (Giddings et al., 2002).

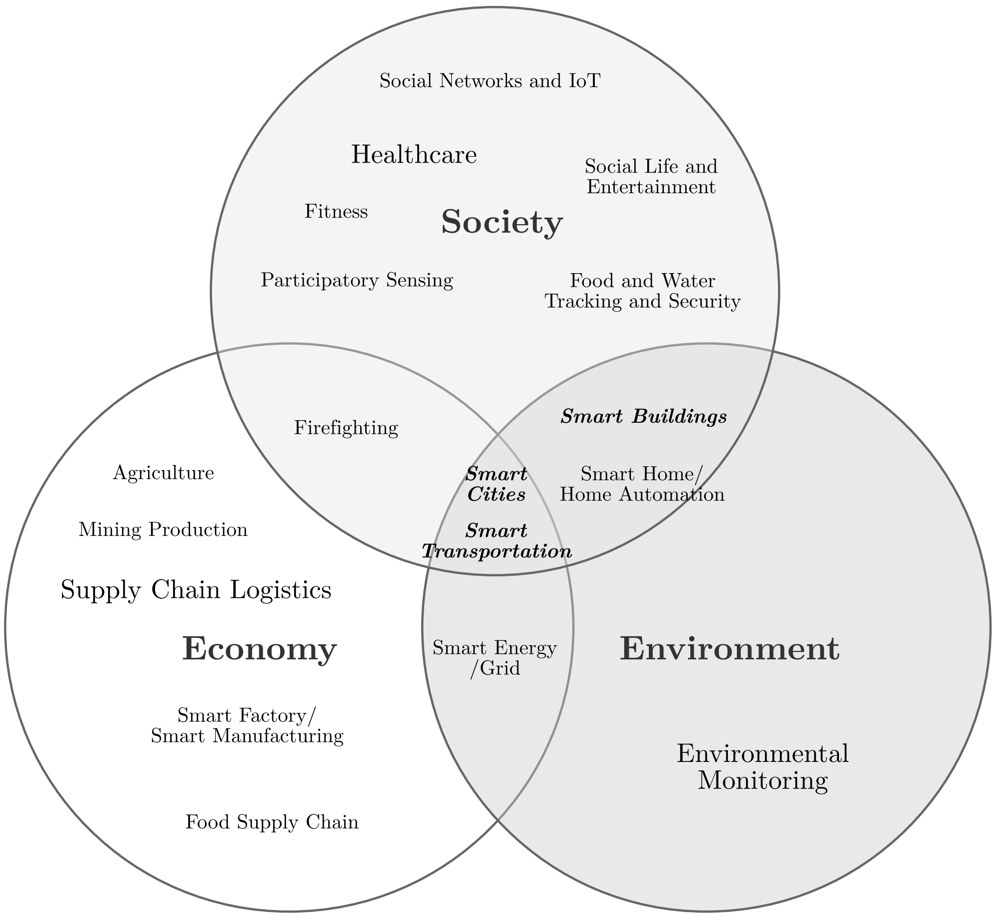


FIGURE 2.2: Application Areas from Surveys Categorised by Impact to Society, Economy and Environment

Figure 2.2 shows the categorisation of the various application areas elicited from the surveys. The categorisation was based on the most direct impact of the output each application area had, for example, the improvement that IoT **healthcare** brought in the ambient-assisted-living use case (Dohr et al., 2010) benefited a group of people within the society directly, while any economic impact was indirect. Hence, healthcare is categorised under society only. Most application areas can be seen to impact the society and economy, as explained by Giddings et al. (2002), because of the political and material reality within countries, constructed by businesses and governments, that prioritises these two areas over the environment.

All of the surveys, except the one by Xu et al. (2014) on industries, have listed smart cities as an IoT application area. **Smart cities** whose vision, as presented by Zanella et al. (2014), is to use advanced technologies to 'support added-value services for the administration of the city and for the citizens', have an impact on the environment, society and economy. In the 101 scenarios within a smart city listed by Presser et al. (2016), there are scenarios with environmental impact like air pollution monitoring and counter measures, those with societal impact like public parking space availability prediction, and those with economic impact like efficient lighting to reduce energy costs.

Smart transportation systems also span smart cities and other application areas like supply chain logistics (Atzori et al., 2010; Xu et al., 2014), can intelligently manage traffic, parking and minimise accidents (Sethi and Sarangi, 2017), or even provide assisted driving or vehicle control and management (Vermesan and Friess, 2013) and impact the economy, society and environment.

Some of the scenarios for the smart city (Presser et al., 2016) also overlap other application areas mentioned like **environmental monitoring** covered by Miorandi et al. (2012) which also extends monitoring to wide areas outside cities for anomalies and early-warning for disasters. The **smart grid**, smart energy (Vermesan and Friess, 2013; Lin et al., 2017) and energy conservation systems (Sethi and Sarangi, 2017) are flexible electrical serving and storage infrastructure utilising renewable sources of energy. The smart grid is able to react to power fluctuations through reconfiguration and optimisation, providing a positive economic and environmental impact.

The application areas of the IoT with an economic impact within industries are also very wide and include monitoring and controlling the **food supply chain** and safer **mining production**, all of which were described in the survey by Xu et al. (2014). Smart inventory and product management as described by Miorandi et al. (2012) helps improve the **supply chain logistics** throughput. **Smart factories** that allow efficient production and optimisation of the manufacturing process are another industrial use case (Vermesan and Friess, 2013). The automatic control of environmental parameters such as temperature and humidity for **agriculture** is yet another industrial IoT application area described by Sethi and Sarangi (2017). **Firefighting** which minimises property damage has both economic and social impact (Xu et al., 2014).

Smart buildings as described by Miorandi et al. (2012) help to reduce the consumption of resources, benefiting the environment, and improve the 'satisfaction level of humans populating it', benefiting the society. **Smart homes** are a particularly popular IoT application area, sometimes considered a specific type of smart building, and have been described in quite a few surveys (Atzori et al., 2010; Miorandi et al., 2012; Vermesan and Friess, 2013; Sethi and Sarangi, 2017). This popularity was attributed by Sethi and Sarangi (2017) to two reasons, the increasing ubiquity of sensors, actuators and wireless networks within home settings, and the increased trust in technology to improve 'quality of life and security' within homes. Other application areas with societal impact include **participatory sensing**, **social networks and IoT**, that involve community information and knowledge, and **food and water tracking and security**, each described by Vermesan and Friess (2013), **social life and entertainment** as well as **fitness** described by Sethi and Sarangi (2017) with facilities like the smart gym for fitness and smart museum for education and entertainment described by Atzori et al. (2010).

Hence this categorisation of application areas helps introduce the full range of use cases presented in all the surveys studied while establishing an understanding of the direct impact of an application area on the development of the society, economy and environment. It is then possible to further distil and group the application areas to derive a set of general domains, namely, health, industry, environment and living, and another set of broad application themes that cut across domains, namely, smart cities, smart transportation and smart buildings. The grouping is shown in Figure 2.3.

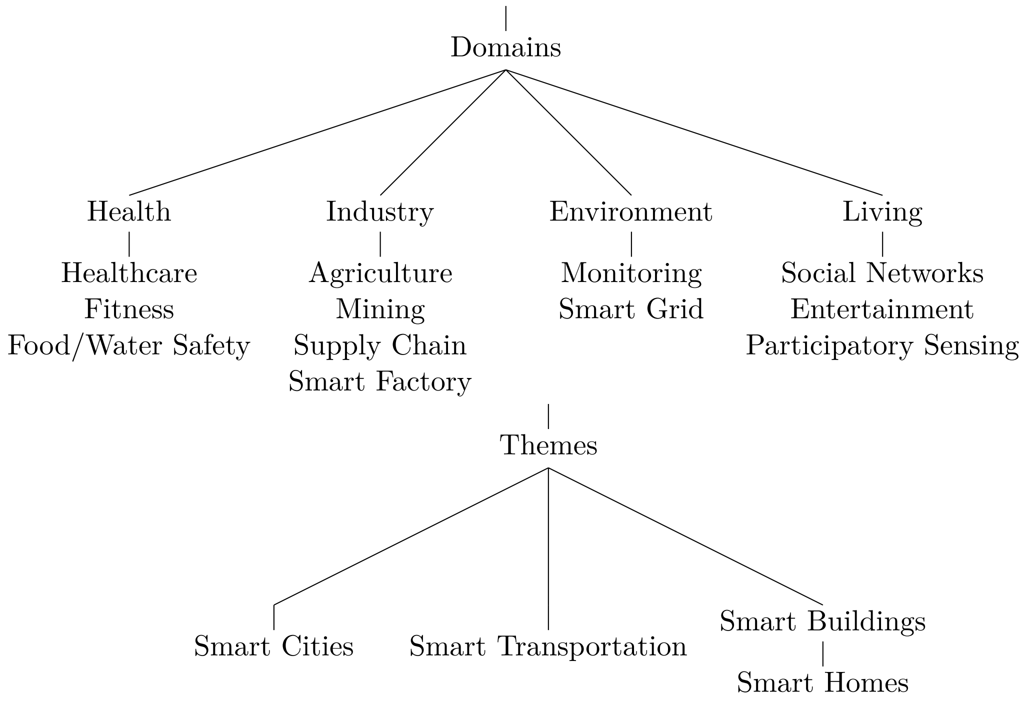


Figure 2.3: Application Domains and Themes Based on Application Areas

This structure of application domains and themes is then helpful for conducting a systematic review to address one of the gaps in the literature present in the reviewed surveys, that is, by producing a summary of IoT research involving actual advanced applications and services, how these applications derive insights and knowledge from data and the kind of data used, streaming or historical. This is covered in the next section.

2.1.3 IoT Applications: A Survey of Techniques and Data Currency

To further the understanding of IoT verticals within applications for this thesis, a survey of the research literature detailing actual advanced applications and services developed for each application domain and theme was conducted. This helps to fill a gap in the literature as the purpose of each of the surveys from Table 2.1 was to introduce application use cases rather than review applications and in particular, the techniques and approaches the applications took to produce smart or intelligent outputs or perform automation. Another goal is also to identify in literature whether IoT applications acted on historical data in databases or processed real-time streaming data.

The methodology for this survey follows that of an evidence-based systematic review (Khan et al., 2003). A total of 460 articles were retrieved from 2011 to 2015, and another 311 articles from 2016 and 2017 were added as the survey was revised and updated. The articles were identified from the Web of Science platform[1], that indexed an extensive list of multi-disciplinary journals and conferences across research databases, with a search criteria that included the keywords `'big data'` or `'analytics'` and filtered by `'internet of things'`. The articles were screened manually with an inclusion criteria that mandated they 1) were original research, 2) described actual designs, implementations and results, 3) had value-added output and 4) served IoT use-cases, bringing the total to a set of 32 articles. These articles were further classified into the four application domains and three application themes derived in the previous section with each domain and theme well-represented as shown in Table 2.2. In addition, the table also states the respective techniques used to derive insights and the currency of the data used, whether it was historical data from a database or real-time streaming data.

Appendix A provides more detailed descriptions of the individual papers for each application domain and theme. A point to note from an analysis of the survey of application papers was that both historical data and real-time streaming data was used in IoT applications across domains which justifies the need to support both databases and stream processing in this thesis' work, as highlighted in Chapter 1. Another point to note was that although the papers covered a broad range of techniques, there was little mention or usage of any common IoT platforms, and each application used their own stack of technologies while data produced was placed in a vertical silo, hence, there was little cross-domain integration of data within the applications, even within application themes like smart cities and smart transportation, at the point of this survey.

The next section continues the study of the IoT by looking at the next category, platforms, backed by the research summarised in the set of IoT surveys from Table 2.1.

[1]https://webofknowledge.com/

Reference	Application	Technique	CC
	Health		
Mukherjee et al. (2012)	Ambient-Assisted-Living	Rules, Ontologies	R
Henson et al. (2012)	Medical Diagnosis	Abductive Logic	H
Rodríguez-González et al. (2012)	Medical Diagnosis	Descriptive Logic	H
Hunink et al. (2014)	Prognosis	Data mining	H
Li et al. (2013)	Healthcare Costing	Random Forests	H
	Industry		
Vargheese and Dahir (2014)	On Shelf Availability	Video Analytics	R
Nechifor et al. (2014)	SCM Environment Control	CEP	R
Verdouw et al. (2013)	Agriculture SCM	CEP	R
Robak et al. (2013)	SCM	Ontologies	R
Engel and Etzion (2011)	SCM Critical Shipments	MDP	R
	Environment		
Yue et al. (2014)	Haze Early Warning	Data mining	R
Schnizler et al. (2014)	Disaster Warning	Anomaly Detection	R
Mukherjee et al. (2013)	Wind Forecasting	ANN	H
Alonso et al. (2013)	Recommend Energy Usage	ML + Rules	H
Ahmed (2014)	Energy Policy Planning	Classifier + Models	H
Ghosh et al. (2013)	Smart Energy System	Data mining	H
	Living		
Gimenez et al. (2012)	Security Monitoring	Video Analytics	R
Mukherjee and Chatterjee (2014)	Wearable Lifestyle Monitor	Anomaly Detection	R
McCue (2006)	Police Deployment	Data mining	H
McCue (2005)	Suspicious Activity Detection	Visualisation	H
Razip et al. (2014)	Situational Awareness	Visualisation	H
Ramakrishnan et al. (2014)	Civil Unrest Prediction	Multimodal	H
Guo et al. (2011)	Memory Augmentation	Data mining	H
	Smart Cities		
Zheng et al. (2014)	Smart Parking	Data mining	R
He et al. (2014)	Smart Parking	Data mining	R
Strohbach et al. (2015)	Smart City Grid	CEP + Models	R
	Smart Transportation		
Mak and Fan (2006)	Traffic Control	Video Analytics	R
Liebig et al. (2014)	Travel Routing	CRF, A*Search	R
Wang and Cao (2014)	Travel Routing	B_n & MDP	R
	Smart Buildings		
Ploennigs et al. (2014)	Smart Building Heating	Anomaly Detection	R
Makonin et al. (2013)	Smart Home Energy	Rules	R
Weiss et al. (2012)	Smart Meter Patterns	Signature Detection	H

Legend: ANN=Artificial Neural Network, B_n=Bayesian Network, **CC = Data Currency**, CEP=Complex Event Processing, CRF=Conditional Random Fields, **H=Historical**, MDP=Markov Decision Process, ML=Machine Learning, **R=Real-time**, SCM=Supply Chain Management

TABLE 2.2: Summary of IoT Applications by Domains and Themes with the Techniques to Derive Insights and the Data Currency of Each Application

2.1.4 IoT Platforms: Architectures, Layers, Middleware

The literature on the IoT has proposed a number of general reference architectures which are summarised in Figure 2.4. The three-layer architecture initially proposed by Yun and Yuxin (2010) drew from service-oriented paradigms in industrial informatics literature and defined an application, network and perception layer. The perception layer for the IoT consists of Things, while the network layer connects applications and services in the application layer to these Things. A similar proposal by Ning and Wang (2011) was inspired by the human nervous system and models the three layers as the brain, spinal cord and nerves. Wu et al. (2010) expanded the application layer to business, application and processing layers to form a five-layer architecture. The business layer contains systems that manage applications, business models and privacy within organisations. The processing layer, also called the middleware layer by Mashal et al. (2015), was introduced to store and process data, delivered by the transport layer and created by Things from the perception layer, for applications. The transport layer, which handles flow control and consistency, and the network layer, which provides addressing and routing, are often combined in IoT literature and referred to by either name.

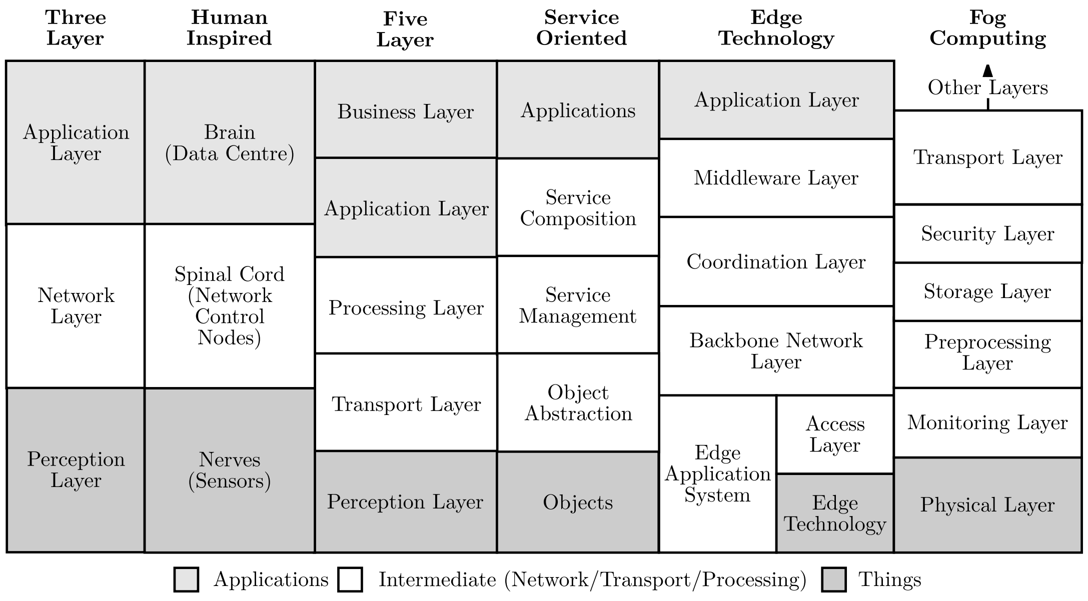


FIGURE 2.4: IoT Architectures

The service-oriented architecture proposed by Atzori et al. (2010) provides well-defined components to describe services for a middleware software layer between the application and perception layer of the IoT, which can also be seen as a sub-layer architecture for the intermediate processing layer.

Tan and Wang (2010) proposed an 'edge technology' architectural design that extends the processing, transport and perception layers to cater for applications that run on

IoT Related Technology	Reference/Link
Application Layer: App Protocols	
Advanced Message Queuing Protocol (AMQP)	http://www.amqp.org
Constrained Application Protocol (CoAP)	http://coap.technology
Data Distribution Service (DDS)	http://portals.omg.org/dds
Hypertext Transfer Protocol (HTTP)	http://http2.github.io
MQ Telemetry Transport (MQTT)	http://mqtt.org
WebSockets	Fette and Melnikov (2011)
XMPP[a]	http://xmpp.org
Application Layer: Others	
Analytics	Section 2.3
Cloud of Things	Aazam et al. (2014)
Social IoT	Atzori et al. (2012)
Social Web of Things	Section 7.3
Web of Things	https://www.w3.org/WoT
Processing Layer	
Big Data Processing	Chen et al. (2014)
Databases and Stream Processing	Section 2.2
Middleware	Razzaque et al. (2016)

[a]Extensible Messaging and Presence Protocol (XMPP)

Table 2.3: References for Application and Processing Layers Technologies

Things themselves or on gateways deployed within localised 'edge networks'. This particular architecture has started to gain popularity recently with the emergence of a new paradigm, fog computing, that provides compute services for 'customers or applications in the space between networking central servers and end-users' (Lin et al., 2017).

Aazam and Huh (2014) proposed an architecture for a fog computing node like a smart gateway, that adds four layers, security, temporary storage, preprocessing and monitoring, between the physical Thing layer and the transport layer. According to this architecture, the security layer provides protection for private and sensitive data, the storage layer stores data on fog resources, the preprocessing layer manages, aggregates and filters data, and the monitoring layer monitors the activities of the underlying nodes.

Each of the surveys with literature on platforms and building blocks was reviewed, with the set of proposed architectures from Figure 2.4 in mind, to determine components within each of the layers. A summary of the findings is presented in Figure 2.5 that shows architectural layers made up of components from the literature. Platforms are horizontals of one or more architectural layers made up of building block components (each a box in the figure) that contain lists of technologies, standards or requirements. The areas of particular research within this thesis, which is concerned with efficient and interoperable data representations within database and stream processing systems for analytical applications, are also highlighted (shaded) in Figure 2.5. The business and

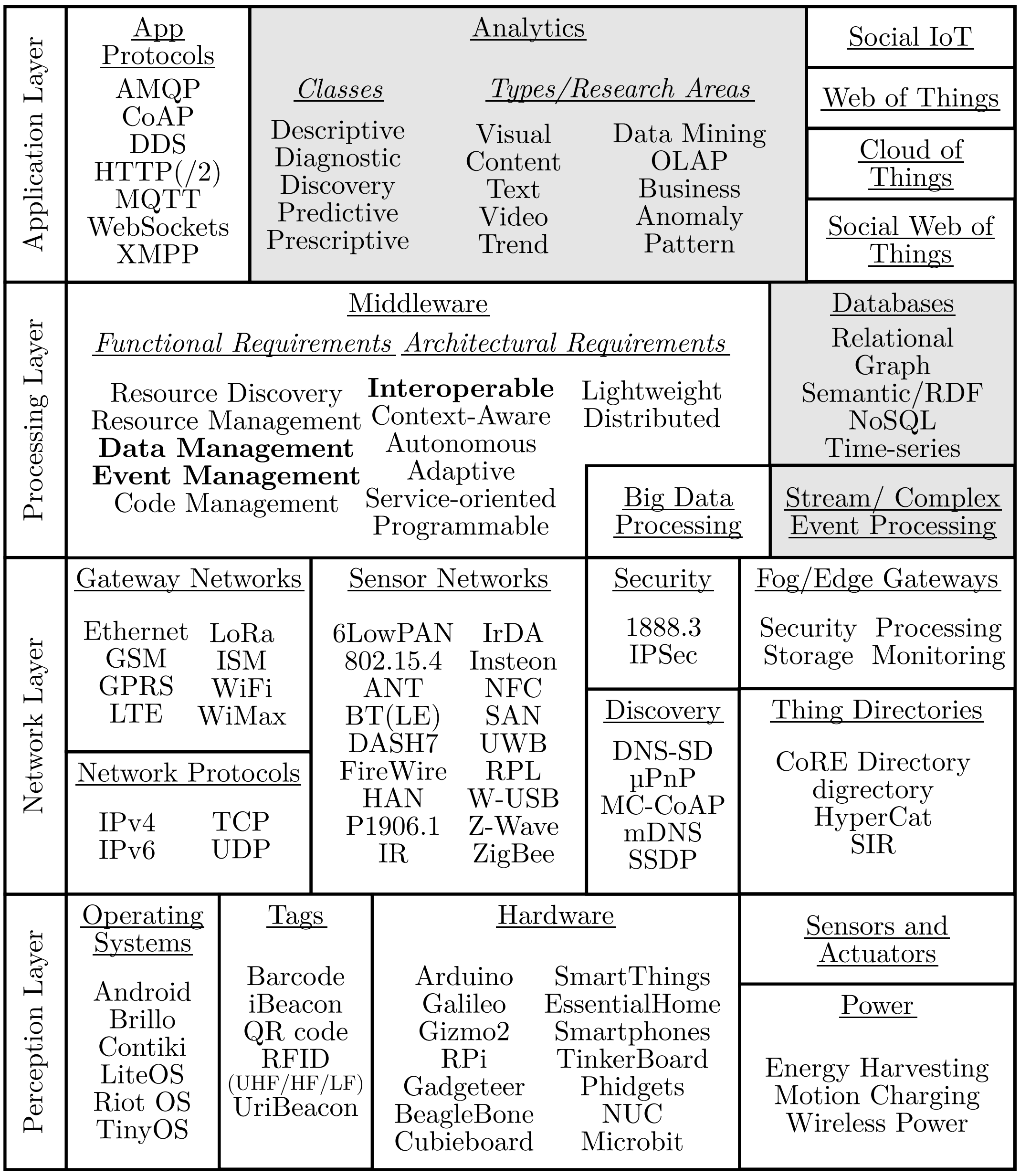


Figure 2.5: IoT Architectural Layers Forming Platform Horizontals with Building Blocks as Components

application layers are combined to form a single layer in this summary, while the network layer contains data-link, network and transport layer functions. Table 2.3 provides references or links to the literature for the application and processing layers while Table 2.4 provides references for the network layer and Table 2.5 for the perception layer.

Middleware for the IoT provides a software layer between the application and the network communications layer. The processing layer, sometimes called the middleware layer, contains IoT middleware which has received much attention within literature (Bandyopadhyay et al., 2011; Farahzadia et al., 2017). In the survey paper on IoT

IoT Related Technology	Reference/Link
Network Layer: Gateway Networks	
Ethernet	IEEE 802.3
GSM and GPRS[a]	Hillebrand (2013)
Long-Term Evolution (LTE)	`http://www.3gpp.org/article/lte`
Long Range (LoRa) Wide Area Network	`http://www.lora-alliance.org`
WaveLAN-II using ISM[b]	Kamerman and Monteban (1997)
WiFi	IEEE 802.11
WiMax[c]	IEEE 802.16
Network Layer: Sensor Networks	
6LowPAN, IEEE 802.15.4	RFC 4944, IEEE 802.15.4
ANT/ANT+	`http://www.thisisant.com`
Bluetooth (BT) Low Energy (LE)	`http://www.bluetooth.com`
DASH7	`http://www.dash7-alliance.org`
IEEE 1394 FireWire	http://1394ta.org
Home Area Network (HAN)	Huq and Islam (2010)
IEEE P1906.1 Nanoscale Communication	IEEE 1906.1-2015
Infrared Protocols (IrDA)	`http://www.irda.org`
Insteon Home Automation Protocol	`http://www.insteon.com`
Near Field Communication (NFC)	ISO/IEC 18000-3
Storage Area Network (SAN)	O'Connor (2003)
Ultra-wideband (UWB), Impulse Radio (IR)	Akyildiz et al. (2002)
Wireless USB (W-USB)	`http://usb.org/developers/wusb`
Z-Wave	`http://www.z-wave.com`
ZigBee	`http://www.zigbee.org`
Network Layer: Security	
IEEE 1888.3 Network Security	IEEE 1888.3-2013
IPSec	RFC 4301, RFC 4309
Network Layer: Discovery	
DNS Service Discovery (DNS-SD)	http://www.dns-sd.org
Micro Plug and Play (μPnP)	Yang et al. (2015)
Multicast CoAP (MC-CoAP)	RFC 7252
Multicast DNS (mDNS)	RFC 6762
Simple Service Discovery Protocol (SSDP)	IETF Draft Revision 3
Network Layer: Thing Directories	
CoRE Directory	IETF Draft Revision 11
digrectory	Jara et al. (2013)
HyperCat	`http://www.hypercat.io`
Sensor Instance Repository (SIR)	Jirka and Nüst (2010)
Network Layer: Others	
Fog/Edge Gateways	Aazam and Huh (2014)
Internet Protocol, Version 6 (IPv6)	RFC 2460

[a]Global System for Mobile Communications (GSM), General Packet Radio Service (GPRS)
[b]Industrial, Scientific and Medical (ISM) Radio Bands
[c]Worldwide Interoperability for Microwave Access (WiMax)

Table 2.4: References for Network Layer Technologies

IoT Related Technology	Reference/Link
Sensors and Actuators	http://distributedfog.com
Perception Layer: Operating Systems	
Android and Brillo	http://developer.android.com/things
Contiki	http://www.contiki-os.org
LiteOS	Cao et al. (2008)
Riot OS	https://riot-os.org
TinyOS	Levis et al. (2005)
Perception Layer: Tags	
iBeacon	http://developer.apple.com/ibeacon
Quick Response (QR) Code	ISO/IEC 18004:2015
Radio-frequency identification (RFID)	http://www.ieee-rfid.org
UriBeacon	http://github.com/google/eddystone
Perception Layer: Hardware	
Arduino	http://www.arduino.cc
BeagleBone Boards	http://beagleboard.org/bone
Cubieboard	http://cubieboard.org
EssentialHome	http://www.essential.com/home
Galileo	http://software.intel.com/iot/
Gizmo2 x86	http://www.gizmosphere.org
Microbit	http://microbit.org
.NET Gadgeteer	Villar et al. (2012)
Next Unit of Computing (NUC)	http://www.intel.com/nuc
Phidgets	http://www.phidgets.com
Raspberry Pi (RPi)	https://www.raspberrypi.org
SmartThings	http://www.smartthings.com
TinkerBoard	https://tinkerboarding.co.uk
Perception Layer: Power	
Energy Harvesting	Wolf (2017)
Motion Charging: Ampy	http://www.getampy.com
Wireless Power: ubeam	https://ubeam.com

TABLE 2.5: References for Perception Layer Technologies

middleware, Razzaque et al. (2016) defined a set of functional and architectural requirements for middleware. A key architectural requirement is that middleware should be interoperable, working with heterogeneous Things and technologies without any 'additional effort from the application or service developer'. Two functional requirements that middleware should service are data management and event management, in other words, the management of historical databases and real-time event streams of IoT data. These requirements are highlighted among others in Figure 2.5. Hence, there is motivation to study databases and stream processing systems as building blocks for the processing layer platforms to be utilised by analytics within the application layer. Background literature on databases and stream processing systems are covered in detail in Section

2.2 while that on analytics for the IoT is covered in Section 2.3.

2.2 Databases and Stream Processing Systems

Two functional requirements of IoT software within the processing layer according to Razzaque et al. (2016) are to provide services that manage data and manage events. As Wu et al. (2011) pointed out, the sheer multitude of diverse Things periodically sending observations about phenomena, or reporting events or anomalies when detected, produce what Abu-Elkheir et al. (2013) described as a massive volume of ‘heterogeneous, streaming and geographically-dispersed real-time data’, which present a different set of requirements and challenges than traditional database management scenarios.

Efficient data and event management for databases and stream processing systems are one requirement. The motivation behind this requirement is the volume of ‘streaming and continuous’ observation data together with an ‘urgency of the response’ within certain applications, like traffic control or early warning systems (Abu-Elkheir et al., 2013). Observation and event data is also of the time-series form, with certain applications demanding fast response times from database systems on queries ‘across time scales from years to milliseconds’, requiring what Andersen and Culler (2016) termed as a ‘new class’ of systems, to meet the ‘unique storage’ demands of the high volume of time-series data.

Interoperability is another key requirement within IoT data management systems, both streaming and historical. Abu-Elkheir et al. (2013), Milenkovic (2015) and Manyika et al. (2015) all have pointed out the importance of integrating metadata, that includes Thing ‘identification, location, processes and services provided’, with the data produced, for interoperability. Hence, flexible data representations that enable the modelling and storage of metadata and data, which provide a means of semantic interoperability, are also desirable.

With these two requirements in mind, this thesis studies the literature behind specialised time-series databases for storing time-series data in Section 2.2.1, flexible graph-based databases like Resource Description Framework (RDF) stores in Section 2.2.2, and stream processing engines in Section 2.2.5.

2.2.1 Time-series Databases

In recent years, large amounts of telemetry data from monitoring various metrics of operational web systems, have provided a strong use case for research involving time-series databases. This research has been driven most recently by the industry, especially technology companies that utilise and manage large web-scale systems. Sensor networks

Time-series Database	Storage Engine		Read	Query Support[a]			
				Π	σ	$\bowtie$	γ
Heroic		Cassandra	HQL	✓	✓	×	✓
KairosDB		Cassandra	JSON	×	t	×	✓
Hawkular		Cassandra	REST	×	t	×	✓
Blueflood		Cassandra	REST	×	t	×	×
InfluxDB	Native	LSM-based	InfluxQL	✓	✓	t	✓
Vulcan/Prometheus	Native	Chunks-on-FS	PromQL	✓	✓	×	✓
Gorilla/Beringei	Native	In-memory	REST	×	t	×	×
BTrDb	Native	COW-tree	REST	×	t	×	✓
Akumuli	Native	Numeric-B+-tree	REST	×	t	×	✓
OpenTSDB	HBase		REST	×	✓	×	✓
Cube	MongoDB		REST	×	t	×	✓
DalmatinerDB	Riak		DQL	✓	✓	×	✓
Riak-TS	Riak		SQL-like	✓	✓	×	✓
Tgres		Postgres	SQL	✓	✓	✓	✓
Timescale		Postgres	SQL	✓	✓	✓	✓

[a]Π = Projection, σ = Selection, where t is selection by time only, $\bowtie$ = Join, where t is joins by time only, γ = Aggregation functions

Table 2.6: Summary of Time-series Databases Storage Engines and Query Support

and financial systems are also sources of time-series data, while the IoT presents a rapidly growing source and sink of such data. Time-series databases from research publications, the industry and open source projects were reviewed and are summarised in Table 2.6 describing the storage engine utilised, the means that applications can query or read data from that particular database and the types of queries supported. The types of queries analysed were guided by basic relational algebra with projections (Π), selections (σ), joins ($\bowtie$) and aggregate functions (γ) essential for time-series data listed. *Projection* refers to selecting a subset of columns or attributes of a time-series, *selection* refers to filtering specified columns or attributes of a time-series by a set of conditions, *joins* refer to joining two or more time-series based on a set of conditions and *aggregate functions* are a set of common functions like mean, summation or finding the maximum of a grouping of values within specified columns. The following sections, Sections 2.2.1.1 to 2.2.1.4, introduce the time-series databases categorised by their underlying storage engines in more detail.

2.2.1.1 Cassandra-based Storage

Cassandra is a distributed NoSQL database initially developed by Lakshman and Malik (2010) and has a hybrid key-value and columnar (or tabular) data model supporting a SQL-like query language without support for table joins. The storage engine architecture consists of an append-only commit log which then writes to an in-memory memtable for buffering writes which are eventually flushed onto disk as immutable Sorted String Tables (SSTables). This design improves write-performance. Cassandra periodically merges SSTables together into larger SSTables using a process called compaction which improves read-performance. The storage engine is a particular implementation of a Log-Structured Merge-tree (LSM-tree) described by O'Neil et al. (1996).

Spotify (2017), a digital music streaming service, developed the time-series database, Heroic, that builds a monitoring system on top of Cassandra for time-series data storage and also uses the distributed full-text indexing search engine Elasticsearch by Elastic (2017) for metadata storage and indexing. Time-series databases Hawkular (Redhat, 2017), KairosDB (2015) and Blueflood from cloud computing provider, Rackspace (2013), were all built on top of Cassandra.

KairosDB provides a JSON-based querying interface, while Heroic has a functional querying language HQL. Hawkular and Blueflood have Representational State Transfer (REST) interfaces that allow query parameters to be passed through Hypertext Transfer Protocol (HTTP) requests for the results to be returned in the subsequent replies.

2.2.1.2 Native Time-series Storage

Gorilla (Pelkonen et al., 2015) is a fast, in-memory time-series database that was developed by researchers at Facebook for their particular monitoring use case to act as an Operational Data Store (ODS) for time-series data. Gorilla uses a novel time-series compression method adapted specifically to the timestamps and values of time-series data within their ODS. Gorilla was open-sourced in part as the time-series database Beringei[2].

InfluxDB by Influx Data (2017) is another native time-series database that allows metadata to be stored as a set of tags for each event. It utilises an LSM-tree (O'Neil et al., 1996) inspired design for its storage engine, similar to Cassandra, but applies Gorilla's compression method for time-series storage and stores metadata tags in-memory.

Andersen and Culler (2016) introduced a time-partitioned, version-annotated, Copy-On-Write (COW) tree data structure for their time-series database, BTrDb, specifically to support high throughput time-series data from metering devices, called microsynchophasors, deployed within an electrical grid. Akumuli by Lazin (2017) is a similar time-series database built for high throughput writes using a Numeric-B+-tree (a log-structured, append-only B+-tree) as its storage data structure. Timestamps within both databases use delta-of-delta compression similar to Gorilla's compression algorithm. Akumuli uses FPC (Burtscher and Ratanaworabhan, 2009), a compression for floating-point values while BTrDb uses delta-of-delta compression on the mantissa and exponent from each

[2]https://github.com/facebookincubator/beringei

floating point number within the sequence. Both databases also utilise the tree structures to store aggregate data to speed up specific aggregate queries (like averages).

Vulcan[3] from cloud-hosting provider, DigitalOcean, added horizontal scalability to the Prometheus (2016) time-series database, originally built by online audio distribution platform, SoundCloud. Prometheus has a native storage layer, which organises time-series data in compressed chunks of a constant size on the filesystem (FS) on disk. Indexes of the time-series metadata and aggregations are stored in the LSM-tree structured key-value store, LevelDB[4].

Prometheus supports a limited, quite terse, functional query language, PromQL, while Gorilla, BTrDb and Akumuli all use REST interfaces and HTTP request-reply querying. InfluxDB features an SQL-like query language, InfluxQL, that supports a limited form of internal joins, only by time or an approximation of time, of two or more time-series.

All the time-series databases that implemented native storage also implemented a means of compression for the individual points within the time-series. Each of the time-series compression algorithms and specialised native storage data structures are explained and analysed in greater detail in Section 5.2.

2.2.1.3 HBase, MongoDB and Riak-based NoSQL Storage

OpenTSDB was originally developed by web content recommendation and discovery engine StumbleUpon (2017) on top of a distributed NoSQL column-oriented key-value store, HBase, described by Wlodarczyk (2012). DalmatinerDB by Project FiFo (2014) and Riak-TS by Basho (2017) were both developed on top of Riak, originally a distributed NoSQL key-value database, while Cube by Square (2012) was developed on MongoDB (Banker, 2011), a NoSQL document-oriented database that stores collections of structured JSON-like (JavaScript Object Notation) documents.

HBase, like Cassandra, has its roots in the Chang et al. (2008) paper on Google's 'sparse, distributed, persistent multidimensional map' design called BigTable. The conceptual data model, like Cassandra's, is tabular, however, the physical view is that of key-value pairs grouped by 'column families' where the key is the row identifier and value is the individual field or attribute. Zhang (2009) provided a description of the underlying HFile format, similar to the SSTable proposed in BigTable and used in Cassandra, explaining the actual key-value storage format in more detail. Internally, OpenTSDB stores all data points in a single table within HBase. All individual values are stored within a column family, and row keys are byte arrays that store the identifier of the time-series, timestamps and metadata tags in such a way as to group data together within time

[3]https://github.com/digitalocean/vulcan
[4]https://github.com/google/leveldb

buckets, by utilising HBase's sorted key-value organisation, for more efficient querying as detailed in their design documentation[5].

Both Riak-TS and DalmatinerDB, built on the distributed infrastructure of Riak, offer specialised time-series storage. Riak-TS as described by Sicular (2016), introduces the need for structuring data according to quanta, single ranges of time, so that logical keys can be stored together internally, thereby achieving 'data co-location' and making queries more efficient. DalmatinerDB was developed on top of Riak and utilises specific features of ZFS, originally called the Zettabyte File System (Bonwick et al., 2003), like disk caching mechanisms ARC and ZIL, and native data compression.

DalmatinerDB and Riak-TS implement SQL-like syntaxes, DQL and a subset of SQL respectively, without support for join operations. OpenTSDB and Cube have REST interfaces allowing query parameters to be passed via HTTP requests and returning results via HTTP replies.

2.2.1.4 Relational-database-based Storage

Tgres by Trubetskoy (2017b) and Timescale (2017) are time-series databases developed on top of or as an extension to PostgreSQL[6], an advanced open-source relational database system. Both Tgres and Timescale have full SQL query support as a result.

Tgres builds a database system on top of PostgreSQL and at the time of writing, featured a write-optimised storage design for time-series data at regularly-spaced intervals. This design avoided the inefficiency of appending each time-series point to a table, which in relational databases involves a look up of the new identifier, locking of the table or part of the table and blocking until data has been synchronised to disk in most cases. This is avoided by implementing a round-robin database design, introduced by Oetiker (1998), with PostgreSQL arrays, where a whole series or even multiple series for a period of time, would be stored in a single row (Trubetskoy, 2017a). An SQL view is created to increase the efficiency of queries explained by Trubetskoy (2016). The drawback of this approach though is that points have to be evenly-spaced and if multiple series are stored in a row, there is no way to read a single series in isolation. Even though individual array elements can be selected in a view, 'under the hood Postgres reads the whole row'.

Timescale implements an extension within the reliable and mature Postgres ecosystem, that optimises the storage of time-series data chunks, partitioning across machines and query planning to access this data. To support the high data write rates of time-series streams, Timescale uses batched commits and in-memory indexes during ingestion.

[5]http://opentsdb.net/docs/build/html/user_guide/backends/hbase.html
[6]https://www.postgresql.org/

2.2.2 Graph and RDF Databases

A graph database stores a finite edge-labelled graph (Angles and Gutierrez, 2008; Cruz et al., 1987). This graph, G, where $G = (V, E)$, has a set of vertices V and edges E, in which $E \subseteq V \times \Sigma \times V$, using the notation from Reutter (2013). Σ is a finite alphabet and V, a countably infinite set of vertices.

Extending on this definition, Libkin et al. (2016) introduced graph databases storing data graphs, which extend the graph to be $G = (V, E, \rho)$, where ρ is a function that assigns data to each vertex in V, such that $V \to D$. This data graph model is useful for representing IoT data and metadata within a flexible, extensible model. This model also has the potential for semantic interoperability and data integration as edges can be made between metadata, for example, definitions of locations, Things descriptions and provenance information.

Some common data graph types that have emerged from the survey literature on graph databases (Buerli, 2012; Kumar Kaliyar, 2015) are the property graph and the Resource Description Framework (RDF) graph. Vrgoc (2014) pointed out that RDF graphs have the added advantage of edge labels being objects themselves, which 'becomes increasingly important in areas such as data integration, provenance tracking, or querying and maintaining clustered data'. There has also been a large body of work on RDF graphs even in the context of the IoT that can be drawn from, hence, the focus of this background chapter is on RDF graph databases. Section 2.2.2.1 introduces RDF and related terminology while Section 2.2.2.2 introduces its relation to the IoT, and Section 2.2.2.3 surveys RDF stores. This is then followed by Section 2.2.3 that details SPARQL-to-relational query translation research and Section 2.2.4 that introduces federated SPARQL research.

2.2.2.1 Introduction to RDF and Related Terminology

The Resource Description Framework (RDF)[7] is a Linked Data format for representing data as a directed, labelled graph on the web. It defines an abstract syntax which is then implemented or serialised as various languages like RDF/Turtle[8] or RDF/XML. RDF documents consist of two key data structures: i) RDF graphs, which are sets of subject-predicate-object triples, where the elements may be Internationalised Resource Identifiers (IRIs), blank nodes, or data-typed literals and ii) RDF datasets, which are used to organise collections of RDF graphs, and comprise a default graph and zero or more named graphs.

Formally, adopting the notation introduced by Chebotko et al. (2009), an RDF graph is a type of graph data model that consists of a set of triple patterns, $tp = (s, p, o)$,

[7]https://www.w3.org/TR/rdf11-concepts
[8]https://www.w3.org/TR/turtle/

whereby each triple pattern has a subject, s, predicate, p, and an object, o. A triple pattern describes a relationship where a vertex, s, is connected to a vertex, o, via an edge, p and $(s, p, o) \in (IB) \times I \times (IBL)$ where I, B and L are pairwise disjoint infinite sets of IRIs, blank nodes and literal values respectively.

SPARQL[9] is a query language for RDF that supports queries defining triple patterns, conjunctions, disjunctions and optional patterns for retrieval from a graph. It supports many features of modern query languages like aggregations, filters, limits and federation. The ability to perform such operations is essential in queries for analytical applications.

RDF stores are purpose-built databases for the storage and retrieval of triples or quads, triples with a context, through semantic queries, like those written in SPARQL. Quads are triples with an additional graph label or context as a fourth element. This graph label refers to an IRI for a named RDF graph or if not specified, the default graph.

Linked Data[10] is defined as a set of best practices for publishing data on the web so that distributed, structured data can be interconnected and made more useful by semantic queries (Bizer et al., 2009). RDF has been described[11] as one of the key ingredients of Linked Data that provides the means for building a graph data model to represent concepts and their relationships with other concepts.

2.2.2.2 RDF, Linked Data and the IoT

Linked Data has demonstrated its feasibility as a means of connecting and integrating web data using current infrastructure (Heath and Bizer, 2011). This has been extended to represent sensor data in RDF with the Semantic Sensor Network (SSN) ontology (Compton et al., 2012), Semantic Sensor Web (SSW) (Sheth et al., 2008) and Linked Sensor Data (LSD) (Patni et al., 2010). Sensors and devices can be described and linked to sensor observations and measured values as Linked Data through these ontologies.

Furthermore, Linked Data has proved successful in various IoT deployments. The Smart-Santander project (Sanchez et al., 2011) is a city-scale experimental research facility and test bed which allows IoT technologies to be deployed and validated. In the context of this project, Turchi et al. (2014) leveraged Linked Data and Representational State Transfer (REST) paradigms to model and create a Web of Things with InterDataNet (Pettenati et al., 2011) middleware, which wrapped sensor data and exposed it as structured documents.

Barnaghi and Presser (2010) also took a similar approach in using Linked Data for integration. In their Web of Things work, Linked Data was used as a standards-based means of connecting devices and abstracting the underlying heterogenous sensor data.

[9] https://www.w3.org/TR/sparql11-query
[10] http://www.w3.org/DesignIssues/LinkedData.html
[11] http://linkeddata.org/faq

Part of the data transformation process was pushed to either the sensor itself or a gateway connected to the sensor so that at query level, data formats were homogeneous. Pfisterer et al. (2011) added to this technique by providing a machine-learning-enhanced, semi-automated approach to annotating sensors with RDF and a prediction model to optimise returning the top-N queries of the current state of all sensors. Kolozali et al. (2014) also proposed an RDF annotation framework for stream data from sensors and Ganz et al. (2013) applied analysis and reasoning on IoT data to integrate and compress heterogenous streams through abstraction.

Barnaghi et al. (2012) summarised that semantics can serve to facilitate interoperability, data abstraction, access and integration with other cyber, social or physical world data in the IoT. In particular, Linked Data helps with interoperability in the IoT through:

1. The use and referencing of common identifiers, Internationalised Resource Identifiers (IRIs), and ontologies, like the SSN and SSW, to help establish common data structures and types for data integration.
2. The provision of rich machine-interpretable descriptions, content metadata, within Linked Data to describe what data represents, where it originates from, how it can be related to its surroundings, who provides it, and what its attributes are.
3. A common, flexible graph data model with RDF to flexibly represent metadata and data which can be extended for provenance tracking and other structural metadata.

However, Buil-Aranda et al. (2013) have examined RDF store endpoints on the web and shown that performance for generic queries can vary by up to 3 to 4 orders of magnitude and stores generally limit or have worsened reliability when issued with a series of non-trivial queries, putting the performance of RDF stores in doubt.

Database researchers like Abadi et al. (2009) have also remarked that although the RDF data representation is flexible, it also ‘has the potential for serious performance issues’ because ‘almost all interesting queries’ involve many self-joins over the set of triple or quad tuples stored by the RDF store. The next section covers the literature on RDF stores and how they try to mitigate performance challenges.

2.2.2.3 RDF Stores

One of the ways that RDF stores overcome query performance issues is by the use of indexes. Indexes, like the 6 SPO (Subject-Predicate-Object) permutations that Neumann and Weikum (2010) proposed in RDF-3X and Weiss et al. (2008) described in Hexastore, are often used in RDF stores which use relational tables or triple tuple-like structures for reducing scans. Ideally, small indexes should also be stored in-memory.

Virtuoso is one such RDF store based on an Object-Relational database optimised for RDF storage by using indexes (Erling, 2001). Virtuoso creates 5 quad indexes, with the combinations PSOG, POGS, SP, OP and GS, where G is the graph label in a quad. The order of the letters denotes how each index has been organised, for example, the SPO index contains all of the triples sorted first by subject, then by predicate, then by object. The open-source Jena Tuple Database (TDB)[12] is a native RDF store, that implements its own underlying storage engine, which uses a single table to store triples/quads. TDB creates 3 triple indexes (OSP, POS, SPO) and 6 quad indexes. The means of organising and storing triples within a table was established by Harris and Shadbolt (2005), who engineered a hashing function for IRIs and literals to become Resource IDentifiers (RIDs) in 3Store, so that triples could be stored in a table with fixed length columns, separated from a symbols table that served as a 'reverse lookup from the hash to the hashed value'. RDFox (Nenov et al., 2015) is a recent in-memory RDF store that uses a more compact incremental index for RIDs to form a dictionary. This is then used in tandem with a six column triple list that stores subject, object, predicate RIDs and pointers to linked list indexes 'grouped by subject, predicate, and object, respectively'.

Commercial stores like GraphDB[13], formerly OWLIM, have also shown to perform well on some benchmarks (Bishop et al., 2011) using 6 indexes (PSO, POS, entities, classes, predicates, literals). Indexing, however, increases the storage size and more memory is required to load indexes. If indexes are larger than memory, either paging or out-of-memory errors might occur. Stardog[14] and AllegroGraph[15] are other commercial RDF stores that integrate other features like full-text search indexing and geospatial queries although their internal design and architecture are proprietary. Both the commercial NoSQL document store, MarkLogic[16], and Oracle's 11g[17] relational database, implement a layer for storing RDF on top of their underlying storage engines.

Another means of scaling of RDF stores that has been deliberated in research is the partitioning of RDF data on distributed clusters. Ladwig and Harth (2011) developed CumulusRDF to store RDF data on the cloud across a distributed Cassandra storage backend. Similarly, clustered and distributed stores were also researched in work by Harris et al. (2009) with 4Store, Thompson (2013) with Blazegraph, formerly known as Bigdata, and Owens et al. (2008) with a clustered version of TDB.

One other bottleneck for RDF stores is scan-intensive tasks which are common in certain analytical queries like data mining for correlations and calculating statistics of a dataset. Hagedorn and Sattler (2013) proposed a solution of an in-memory, scan-optimised, cache-aware engine that exploits parallel processing through the awareness of partitions in

[12]http://jena.apache.org
[13]http://ontotext.com/products/graphdb
[14]http://www.stardog.com/
[15]https://franz.com/agraph/allegrograph/
[16]http://www.marklogic.com/
[17]http://www.oracle.com/technetwork/database/

scans and employs work-stealing or work-sharing depending on the query, joining or filtering for better utilisation of multi-core systems.

Concerned with the heavy load and reduced availability of SPARQL endpoints, Verborgh et al. (2014) proposed an alternative query method using Linked Data Fragments (LDF), to increase the availability of endpoints. Part of the workload of SPARQL queries is shifted to the client side. The server side preserves resources to handle more clients by serving Triple Pattern Fragments, while leaving joins and aggregations to the client.

2.2.3 SPARQL-to-Relational Query Translation

SPARQL queries can be performed on data stored in relational databases and queried with SQL without transformation to RDF with the help of mappings and SPARQL-to-SQL query translation. The RDB to RDF Mapping Language (R2RML)[18] is a W3C recommendation based on the concept of mapping logical tables in relational databases to RDF with Triples Maps. Triple Maps specify how a set of triples, with their subject, predicate and object are each mapped to columns in a table. The RDF Mapping language (RML) by Dimou et al. (2014) extended R2RML beyond relational data sources. These input sources could be tabular data within CSV files, hierarchical data within XML, HTML documents or web Application Programming Interfaces (APIs).

Efficient SPARQL-to-SQL translation that supports R2RML has been investigated by Priyatna et al. (2014) and Rodriguez-Muro and Rezk (2015) with state-of-the-art translation engines Ontop and Morph respectively. Both engines optimise query translation to remove redundant self-joins, improving on work in D2RQ (Bizer and Cyganiak, 2007), work by Elliott et al. (2009) who proposed an augmentation-based approach that highlighted required variables to minimise self-joins and by Chebotko et al. (2009) who proposed a set of six semantics preserving simplifications to a formal method of SPARQL-to-SQL translation for relational databases storing triples.

Ontop, which translates mappings and queries to a set of Datalog rules, applies query containment and semantic query optimisation to create efficient SQL queries. The optimiser can chase equality generating dependencies for semantic query optimisation, for example, in the case of a self-join on the same column, if the column is a primary key, chasing dependencies generates duplicated atoms which can be safely eliminated, removing the redundant self-join.

Both Morph, which extended the approach of Chebotko et al. (2009) for R2RML-mapped relational databases, and Ontop, have been shown to perform better on various benchmarks like the Berlin SPARQL Benchmark (Bizer and Schultz, 2001) and FishMark (Bail et al., 2012), as well as various project deployments, when compared to traditional RDF

[18]http://www.w3.org/TR/r2rml/

stores. Efficient SPARQL-to-relational translation for the IoT is studied in Chapter 4 with comparisons to Morph and Ontop.

2.2.4 Federated SPARQL

Federated SPARQL queries are queries executed on different, distributed SPARQL endpoints, aggregating the results. They can be seen as a means of gathering Linked Data from a set of disparate sources, hence, the performance of federated queries is almost entirely dependant on the underlying endpoint performance. However, there is an area of work in improving the performance of federated engines which is surveyed as follows. DARQ (Quilitz and Leser, 2008), ADERIS (Lynden et al., 2011), FedX (Schwarte et al., 2011), SPLENDID (Gorlitz and Staab, 2011) and LHD (Wang et al., 2013) are all engines that optimise federated query performance. While FedX uses a heuristic to optimise the join order, all the other engines use statistical means of optimising the federated join order of queries. These systems have been evaluated and reviewed by Schmidt et al. (2011), Hartig (2013) and Saleem et al. (2014), with benchmarks tuned for federated queries. In the evaluation by Saleem et al. (2014), a cached version of FedX performed the best over the range of queries. However, it is inconclusive whether the use of iterative dynamic programming or greedy algorithms perform better in general scenarios and also depend on the availability of source metadata (data access statistics).

More specific performance bottlenecks like data distribution (Rakhmawati and Hausenblas, 2012) and other challenges (Rakhmawati et al., 2013) for federated engines have been identified. Further work by Konrath et al. (2012) in designing a Linked Data spider, that crawls and streams data back to index and identify sources of specific data types, data types (clusters) or equivalence classes, could be useful in providing source information, metadata and indexes for federated systems.

2.2.5 Stream Processing Engines

Stankovic (2014) explained that the IoT will increasingly be composed of ‘a very large number of real-time sensor data streams’ and that a ‘given stream of data will be used in many different ways’. The large volume of data within streams and the need to process these streams quickly and in complex ways, interacting with historical data and other information sources, has led to the development of stream processing systems. Stream processing systems are sometimes called event processing systems or Complex Event Processing (CEP) systems.

As in previous sections, the focus for this thesis is on interoperable models for metadata and data storage, hence, stream processing engines that support the flexible RDF graph model are considered. RDF Stream Processing (RSP) is the area of work which enables

Linked Data as RDF to be produced, transmitted and continuously queried as streams[19]. RSP takes advantage of the semantic interoperability of RDF graphs, while preserving the power of continuously running queries on high volume streams of time-series IoT data.

Zhang et al. (2012) evaluated the functionality of three streaming engines that respectively represent pull-based querying on streams, C-SPARQL (Barbieri et al., 2010), query translation to process streaming data, Morph-streams (Calbimonte et al., 2010), and a native stream processing engine, CQELS (Le-Phuoc et al., 2011). The evaluation, though, focused on the functionality rather than the performance of RSP engines.

The C-SPARQL engine supports continuous pull-based SPARQL queries over RDF data streams by using Esper[20], an open-source complex event processing engine, to form windows in which SPARQL queries can be executed on an in-memory RDF model.

Morph-streams, previously $SPARQL_{stream}$, supports query translation and rewriting from SPARQL to event processing languages with R2RML mappings and execution on the corresponding event processing engine e.g. Esper.

CQELS is a native RDF stream engine, supporting push and pull queries, that takes a ‘white-box’ approach for full control over query optimisation and execution. The authors argue that the need for query translation and data transformation in other engines, as well as the lack of full control over the query execution, pose major drawbacks in terms of efficiency. Therefore, CQELS natively implements the required query operators and optimises query execution by continuously re-ordering operators according to heuristics and caching intermediate results. Efficient RSP for the IoT is studied in Chapter 6 with comparisons to CQELS and C-SPARQL.

2.3 Analytics for the IoT

Analytics is the science or method of using analysis to examine something complex (Oxford English Dictionary, 2017a). When applied to data, analytics is the process of deriving (the analysis step) knowledge and insights from data (something complex). The evolution to the concept of analytics today can be traced back to 1962.

Tukey (1962) first defined data analysis as procedures for analysing data, techniques for interpreting the results, data gathering that makes analysis easier, more precise and accurate and finally, all the related machinery and statistical methods used. In 1996, Fayyad et al. (1996) published an article explaining Knowledge Discovery in Databases (KDD) as the overall process of ‘discovering useful knowledge from data’. Data mining

[19]https://www.w3.org/community/rsp/
[20]http://www.espertech.com/products/esper.php

serves as a step in this process - the application of 'specific algorithms for extracting patterns from data'.

In 2006, Davenport (2006) introduced analytics as quantitative, statistical or predictive models to analyse business problems like financial performance or supply chains and stressed its emergence as a fact-based decision-making tool in businesses. In 2009, Varian (2009) highlighted the ability to take data and 'understand it, process it, extract value from it, visualise it and communicate it', as a hugely important skill in the coming decade. In 2013, Davenport (2013) introduced the concepts of Analytics 1.0, traditional analytics, Analytics 2.0, the development of big data technology, and Analytics 3.0, where this big data technology is integrated agilely with analytics, yielding rapid insights and business impact.

2.3.1 Five Categories of Analytics Capabilities

To start to understand analytics, a classification of analytics according to its capabilities was derived from literature. Bertolucci (2013) proposed descriptive, predictive and prescriptive categories while Gartner (Chandler et al., 2011; Kart, 2012) proposed the extra category of diagnostic analytics. Finally, Corcoran (2012) introduced the additional category of discovery. This thesis adopts the latter, the most comprehensive categorisation of analytic capabilities: descriptive, diagnostic, discovery, predictive and prescriptive analytics. These capabilities are discussed in the next few sub-sections.

2.3.1.1 Descriptive Analytics

Descriptive Analytics helps to answer the question: "What happened?". It can take the form of describing, summarising or presenting raw IoT data that has been gathered. Data is decoded, interpreted, fused and then presented so that it can be understood and might take the form of a chart, a report, statistics or some aggregation of information.

2.3.1.2 Diagnostic Analytics

Diagnostic Analytics is the process of understanding why something has happened. This goes one step deeper than descriptive analytics in that the root cause and explanations for the IoT data are diagnosed. Both descriptive and diagnostic analytics give hindsight on what and why things have happened.

2.3.1.3 Discovery in Analytics

Through the application of inference, reasoning or detecting non-trivial information from raw IoT data, the capability of Discovery in Analytics is obtained. Given the acute problem of volume that big data presents, Discovery in Analytics is also very valuable in narrowing down the search space of analytics applications. Discovery in Analytics on data tries to answer the question of what happened that was not previously known and the outcome is insight into what happened. What differentiates this from the previous types of analytics is using the data to detect something new, novel or different, for example, trends, exceptions or clusters, rather than simply describing or explaining it.

2.3.1.4 Predictive Analytics

For the final two categories of analytics, the focus is shifted from hindsight and insight to foresight. Predictive Analytics tries to answer the question: "What is likely to happen?". It uses past data and knowledge to predict future outcomes (Hair Jr, 2007) and provides methods to assess the quality of these predictions (Shmueli and Koppiu, 2010).

2.3.1.5 Prescriptive Analytics

Prescriptive Analytics looks at the question of what to do about what has happened or is likely to happen. It enables decision-makers to not only look into the future about potential opportunities (and issues), but also presents the best course of action to act on foresight in a timely manner (Basu, 2013) with the consideration of uncertainty. This form of analytical capability is closely coupled with optimisation, answering 'what if' questions so as to evaluate and present the best solution.

2.3.2 Knowledge Hierarchy and IoT App Classification

Figure 2.6 shows how each of the five analytical capabilities proposed in the previous section fits in with the Knowledge Hierarchy (Rowley, 2007; Bernstein, 2011), which is a framework used in the Knowledge Management domain. The value of each capability, is also highlighted in the figure. The knowledge hierarchy starts with data at the base, examples of which are facts, figures and observations (e.g. the raw data produced by Things). Information is interpreted data with context, for example, temperature for a particular location represented by descriptive analytics: an average over a month or a categorical description of the day being sunny and warm. Knowledge is information within a context with added understanding and meaning, for example, possible reasons for the high average temperature this month. Finally, wisdom is knowledge with insight, for example, discovering a particular trend in temperature and projecting it across future

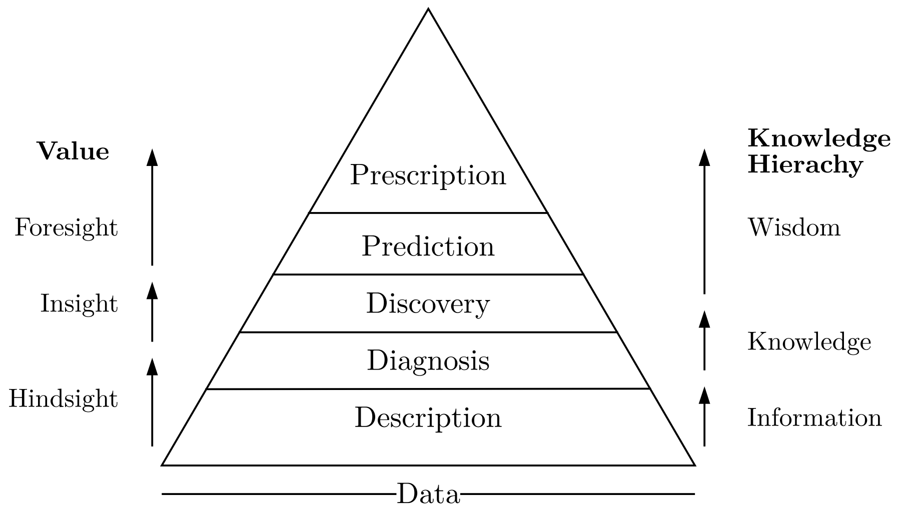


FIGURE 2.6: Analytics and the Knowledge and Value Hierarchies

months while providing cost saving energy management solutions for heating a smart home based on these predictions.

Each component of the knowledge hierarchy builds on the previous tier and the same applies to analytical capabilities. To add a practical view from business management literature to our discussion, a review of organisations adopting analytics (Lavalle et al., 2010) categorised them as Aspirational, Experienced and Transformed. Aspirational organisations were seen to use analytics in hindsight as a justification for actions, utilising the data, information and knowledge tiers in the process. Experienced organisations utilised insights to guide decisions and transformed organisations were characterised by their ability to use analytics to prescribe their actions, effectively applying foresight in their decision making process.

The IoT applications surveyed in Table 2.2 can then be classified according to each of the five analytical capabilities they possess. Table 2.7 shows each application, their domain and the capabilities they achieve. It can be seen that there is a good representation of capabilities across IoT applications and that support for processing each type of analytics is required. Discovery and predictive analytics were the most common in applications, while prescriptive analytics was the hardest to achieve in most applications.

2.4 Fog Computing

Chiang et al. (2017) defined fog computing as an ‘end-to-end horizontal architecture’ for the IoT that distributes the communication, compute, storage and control planes nearer to users ‘along the cloud-to-thing continuum’. This is a wider definition than the one by Bonomi et al. (2012) who initially introduced fog computing as extending the cloud

Reference	Describe	Diagnose	Discover	Predict	Prescribe
Health					
Mukherjee et al. (2012)	✓		✓		
Henson et al. (2012)		✓			
Rodríguez-González et al. (2012)		✓			
Hunink et al. (2014)				✓	
Li et al. (2013)				✓	
Industry					
Vargheese and Dahir (2014)				✓	
Nechifor et al. (2014)				✓	
Verdouw et al. (2013)				✓	
Robak et al. (2013)	✓				
Engel and Etzion (2011)					✓
Environment					
Yue et al. (2014)			✓		
Schnizler et al. (2014)			✓		
Mukherjee et al. (2013)				✓	
Alonso et al. (2013)				✓	
Ahmed (2014)				✓	
Ghosh et al. (2013)			✓		
Living					
Gimenez et al. (2012)			✓		
Mukherjee and Chatterjee (2014)			✓		
McCue (2006)			✓		
McCue (2005)			✓		
Razip et al. (2014)	✓				
Ramakrishnan et al. (2014)				✓	
Guo et al. (2011)			✓		
Smart Cities					
Zheng et al. (2014)			✓		
He et al. (2014)			✓		
Strohbach et al. (2015)	✓			✓	
Smart Transportation					
Mak and Fan (2006)	✓				
Liebig et al. (2014)				✓	
Wang and Cao (2014)					✓
Smart Buildings					
Ploennigs et al. (2014)		✓			
Makonin et al. (2013)		✓	✓	✓	
Weiss et al. (2012)	✓		✓		

Table 2.7: Summary of Applications, Domains and their Analytical Capabilities by Reference

computing paradigm 'to the edge of the network'. Some authors like Lin et al. (2017) have also used the terms fog and edge computing interchangeably.

Edge computing is commonly defined in IoT literature as performing compute and storage on edge devices like smart devices, routers, base stations or home gateways. For example, a smart security camera system might be capable of 'Edge Analytics' by performing face recognition and signature generation on video content before sending just the signature to the cloud for database matching (Intel, 2016).

Across the literature though, common characteristics of fog computing systems can be determined and a set of common planes for fog computing networks, largely borrowed from routing literature (Yang et al., 2004), can be determined. These are presented in the next two sections (Sections 2.4.1 and 2.4.2) with example applications in Section 2.4.2.1.

2.4.1 Characteristics of Fog Computing Systems

Fog computing systems have the advantage of lower latency, greater efficiency and can support performance-critical, 'time-sensitive control functions' as they are in closer proximity and can be more closely integrated with local IoT systems (Chiang et al., 2017). Bonomi et al. (2014) also noted that this is an advantage fog computing systems have over the cloud for applications which require 'very low and predictable latency'.

There are also security advantages in fog computing systems as the distance that data is sent is minimised, reducing the vulnerability to eavesdropping. Proximity-based authentication also becomes a possibility. However, unique security challenges also exist like node capture attacks and sleep deprivation attacks of low-powered devices (Lin et al., 2017). Byers and Swanson (2017) argued that end-to-end security is critical to the success of all fog computing deployment scenarios.

Another characteristic is the large number of nodes. This could be a result of wide geographical distribution and large-scale sensor networks like Smart Grids (Bonomi et al., 2012). Chiang et al. (2017) suggested that this brings about a certain agility, allowing rapid innovation and scaling on affordable nodes. However, this also means that fog nodes will likely be heterogeneous (Vaquero and Rodero-Merino, 2014) making semantic interoperability an important concern.

Finally, localised data analytics coupled with control (Byers and Swanson, 2017) should also work with support for online analytics and interplay with the cloud (Bonomi et al., 2012) achieving the vision of a continuum.

2.4.2 Fog Computing Data and Control Planes

In Section 2.1.4, it was highlighted that Aazam and Huh (2014) proposed an architecture for a fog computing node that contained security, storage, processing (compute) and monitoring layers. Fog computing literature borrows from routing literature (Yang et al., 2004) in defining common data and control planes. From this definition, the data plane can be seen to encompass storage and processing of data while the control plane can be seen to handle security and monitoring tasks.

Another alternative way of envisioning fog computing is by dividing it into architectures for compute, control and communication (Chiang et al., 2017).

2.4.2.1 Example Fog Computing Applications and Infrastructure

An example fog computing application given by Bonomi et al. (2014) is a Smart Traffic Light system that has local subsystem latency requirements in the order of less than 10ms, while requiring deep analytics over long-periods in the cloud.

Lin et al. (2017) described smart grid, smart transportation and smart city examples of integrating fog computing. Each example highlights how fog computing nodes can collect data in a distributed manner and advise distributed energy generators, smart vehicles and smart city sub-applications respectively.

Dastjerdi and Buyya (2016) also described healthcare IoT applications utilising fog computing, giving the example of fall-detection, prediction and prevention. Experiments implementing the proposed fog computing solution had lower latency and consumed less energy than ‘cloud-only approaches’.

Fog computing technologies like ANGELS (Mukherjee et al., 2014) and the scheduler designed by Dey et al. (2013), proposed utilising the idle computing resources of smartphones serving as fog computing nodes ‘on the edge’ through a scheduler in cloud. The nodes themselves keep track of their resource usage states, which are formed based on user behavioural patterns, and advertise free slot availability. The cloud servers receive analytics jobs and advertisements from nodes and then schedule subtasks to them. Cloudlets allow a mobile user to instantiate virtualised compute tasks on physically proximate cloudlet hardware (Satyanarayanan et al., 2009) while IOx by Cisco (2015) is another platform that consists of a fog director, application host and management components, allowing tasks to be virtualised and executed on fog computing nodes.

Machine learning on resource-constrained fog computing nodes has also been demonstrated in the context of a variety of supervised learning and classification tasks (Gupta et al., 2017) and on prediction using a tree model (Kumar et al., 2017).

2.5 Conclusions

This chapter presented the background literature from IoT and database systems research. A study of IoT surveys helped advise a categorisation of the IoT according to applications and platforms with building blocks. Application domains and themes were determined and a survey of IoT applications was done. IoT architectures and platforms were also explored with a layered architecture used to group building blocks of IoT technology. This understanding of the IoT literature, helped to focus the next few sections studying database systems literature on time-series databases and graph databases, especially RDF stores and graph stream processing. This was followed by a study of the history of analytics and analytical capabilities from the literature. IoT applications surveyed were also classified according to how they realised analytical capabilities. Finally, common characteristics that defined fog computing across the literature were identified while example applications and infrastructure were reviewed briefly.

Hence, equipped with a better understanding of the IoT, database systems and specific focus areas of fog computing, time-series databases, RDF databases and stream processing, the next chapter looks at the structure and nature of public IoT data to try and determine if an interoperable data representation can be found and how this representation can be optimised for performance.

Chapter 3

Analysis of the Characteristics of Internet of Things Data

"Life can only be understood backwards; but it must be lived forwards."

— *Soren Kierkegaard*

This chapter describes several studies performed on publicly available sources of data to analyse the characteristics of Internet of Things (IoT) data. The first study, in Section 3.1, establishes an understanding of the structure and four characteristics that can be elicited from IoT data - flatness, wideness, numerical value types and a mixture of periodicity. A cross-IoT study in Section 3.2 then investigates a more diverse sample of sources across IoT application domains for evidence of the characteristics. The chapter moves on to study proposals for metadata representation that enhances semantic interoperability in the IoT in Section 3.3. Finally, Section 3.4 details a study of the characteristics, from a database systems perspective, of a particular rich graph model, the Resource Description Format (RDF), which was found to be more flexible than tree-based models and hence appropriate for modelling metadata and data in the IoT and Section 3.5 concludes with a summary of findings.

3.1 Study of Public IoT Schemata

To investigate the characteristics of data from Internet of Things (IoT) sensors, the public schemata of 11,560 unique Things from data streams on Dweet.io[1] for a month in 2016 were analysed. These were drawn from a larger collected set of 22,662 schemata of which 1,541 empty and 9,561 non-IoT schemata were filtered away. The non-IoT schemata were largely from the use of Dweet.io for rooms in a relay chat stream.

[1]http://dweet.io/see

Dweet.io is a cloud-based platform that supports the publishing of sensor data from Things in JavaScript Object Notation (JSON). JSON is a data-interchange format that uses human-readable text to transmit data objects consisting of attribute-value pairs, array data types and nested data objects. From the public JSON harvested from Dweet.io Things, schemata were derived, classified and analysed for the following characteristics: 1) flat or complex, 2) width and 3) types of values. These characteristics, from a database systems perspective, help to determine how IoT time-series data should be represented for better read, write and storage performance. For example, they represent design decisions for databases on the conceptual organisation (e.g. table or document tree organisation), physical organisation (e.g. rows or columns) and compression support (e.g. variable-length string compression or fixed-length floating point compression) respectively. Additionally, streams of JSON data objects from each Thing were analysed and the periodicity of data was determined. Each of these characteristics is explained and the results are presented in the following sub-sections.

3.1.1 Comparison of Flat and Complex Schemata

```
{
    timestamp: 1501513833,
    temperature: 32.0,
    humidity: 10.5,
    light: 120.0,
    pressure: 28.0,
    wind_speed: 10.5,
    wind_direction: 0
}
```

Listing 3.1: Flat Schema

```
{
    timestamp: 1501513833,
    temperature: {
       max: 22.0,
       current: 17.0,
       error : {
          percent: 5.0
       }
    }
}
```

Listing 3.2: Complex Schema

A flat schema in JSON, as shown in Listing 3.1, is a schema with only a single level of attribute-value pairs, while a complex schema in JSON, as shown in Listing 3.2, has multiple nested levels of data within a tree-like, hierarchical structure.

It was observed that of the schemata sampled, 99.2% (11,468) were flat while only 0.8% (92) were complex.

3.1.2 Width of Schemata

The width of a schema refers to the number of attributes it contains besides the timestamp attribute. In Listing 3.1, the flat JSON schema has a width of 6 because it has a set containing 6 attributes besides the timestamp: (*temperature, humidity, light, pressure, wind_speed, wind_direction*). A schema is considered wide if the width is 2 or more.

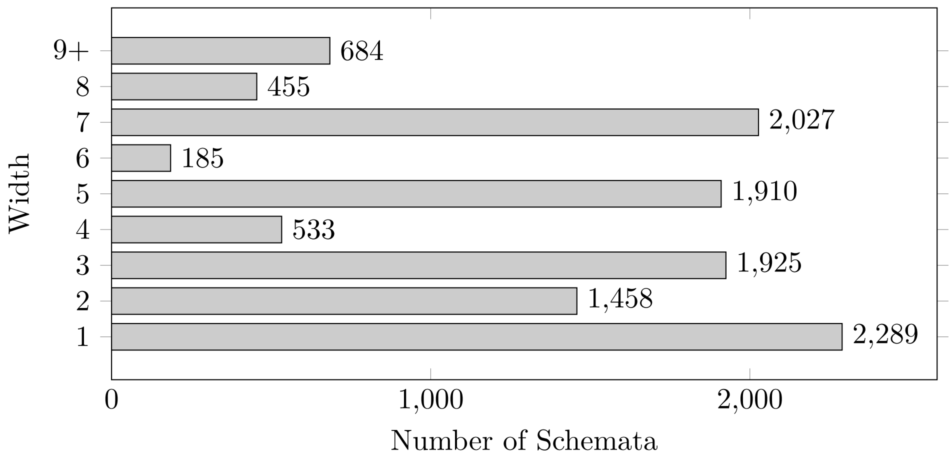


FIGURE 3.1: Width of Schema Histogram from Dweet.io Schemata

This definition borrows from the *tall versus wide* schema design philosophy in big data key-value stores for time-series such as in BigTable and HBase[2].

In the schemata sampled, it was found that 80.0% of Things had a schema that was wide while the majority, 57.3%, had 5 or more attributes related to each timestamp. However, only about 6% had more than 8 attributes. Results are presented in Figure 3.1. The most common set of attributes was from inertial sensors (gyroscope and accelerometer), (*tilt_x, tilt_y, tilt_z*), at 31.3% and networking metrics, (*memfree, avgrtt, cpu, hsdisabled, users, ploss, uptime*), at 9.8%. 1.1% (122) of the schemata were environment sensors with (*temperature, humidity*) attributes.

3.1.3 Data Types of Values within Schemata

The value from an attribute-value pair within a schema can assume varying data types. To determine the distribution of data types of values across all attribute-value pairs, the sampled data was first classified based on basic JSON data types, implicit within flat schema data objects: *Number, String* and *Boolean.* The *String* attribute-value pairs were then further analysed to properly categorise numerical strings, boolean strings, identifiers with a Universally Unique Identifier (UUID) format, categorical strings and test data within their respective categories. A semi-automated process that used human-curated regular expression filters on attribute-values and a dictionary of common attribute-keys was applied to the dataset for this further categorisation. The results were manually verified for accuracy. A similar process was done on numbers to categorise categorical numbers (numerical fields that exhibited qualitative properties so that they took on one of a fixed number of possible values) and identifiers as the appropriate data type. The source code of the classifier and the dictionaries created are available from a public repository[3].

[2]http://hbase.apache.org/book.html#casestudies.schema
[3]https://github.com/eugenesiow/iotdevices

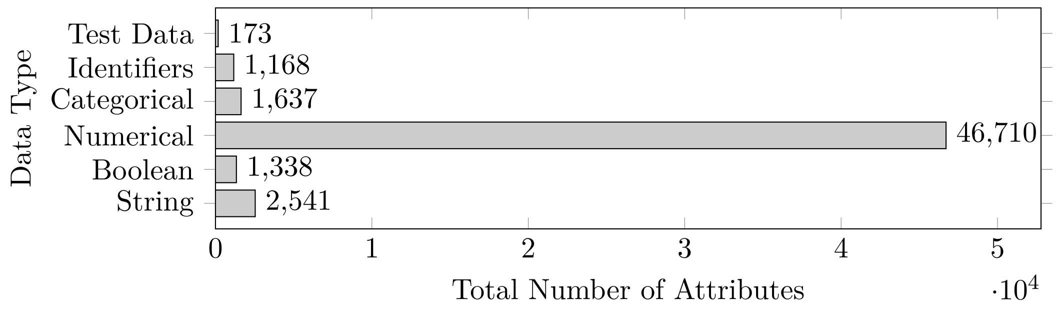


FIGURE 3.2: Number of Attributes of Types of Values in Dweet.io Schemata

It was observed that a majority of attributes besides the timestamp were numerical (87.2%) as shown in Figure 3.2. Numerical attributes include integers, floating point numbers and time values. Identifiers (2.2%), categorical attributes (3.1%) that take on only a limited number of possible values, e.g. a country field, and Boolean fields (2.5%) occupied a small percentage each. Some test data (0.3%) like *'hello world'* and *'foo bar'* were also discovered within string attributes and separately categorised by the classification process. String attributes occupied 4.7%. Of the 2,541 string attribute keys, 13.7% were '*name*', 8.1% were '*message*' and 3.2% were '*raw*' attribute keys.

3.1.4 Periodicity of Data Streams

One of the fundamental characteristics of data streams that form a time-series is how frequently and evenly the observations are spaced in time. The definition of periodicity adopted in this thesis is from the Oxford English Dictionary (2017b) and refers to 'the quality of regular recurrence', in particular, the tendency for data observations in a stream to 'recur at regular intervals'. This should not be confused with the definition of periodicity as seasonality in some data mining literature. A period is the interval between subsequent observations in a time-series and a periodic stream is defined as one with exactly regular timestamp intervals or in other words, a constant period, while a non-periodic stream has unevenly-spaced intervals or a varying period. A periodic stream has maximum periodicity while a non-periodic stream could have high or low periodicity.

One of the differences between the Internet of Things and traditional wireless sensor networks is the potentially advantageous deployment of event-driven sensors within Things instead of sensors that record measurements at regular time intervals, determined by a constant sampling rate. For example, a smart light bulb that measures when a light is on can either send the signal only when the light changes state, i.e. is switched on or off, or send its state regularly every second. The former type of event-driven sensor gives rise to a non-periodic time series as shown in Figure 3.3.

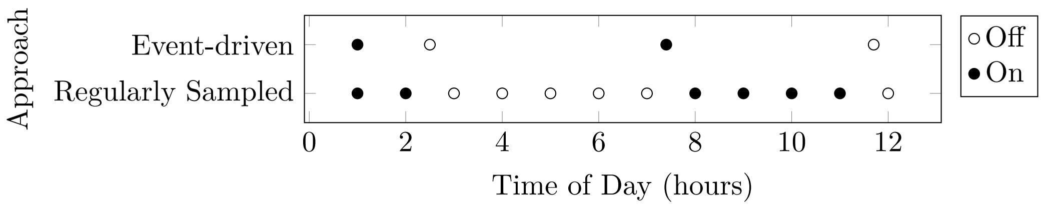


FIGURE 3.3: Non-Periodic Event-driven VS Periodic Regularly Sampled Smart Light Bulb Time-series

Wolf (2017) provided evidence that self-powered, energy-scavenging Things can only run and communicate at 'very low' sample rates to 'live within the power generation capabilities of even fairly robust energy-scavenging mechanisms'. By taking an event-driven approach where data is only transmitted when an important feature is found or a state changed, Things can take advantage of the power consumption characteristics of a lower cost of local computation as opposed to the higher cost of communication.

Hence, event-driven sensing has the advantages of 1) more efficient energy usage preserving the battery as long as events occur less often than regular updates, 2) better time resolution as timestamps of precisely when the state changed or event occurred are known without needing to implement buffering logic on the sensor itself between regular signals and 3) less redundancy in sensor data storage. Both data redundancy and higher precision timestamps are particularly pertinent to this thesis, which is concerned with handling redundant values and high precision timestamps, so as to best store and retrieve IoT time-series. There is a potential disadvantage of event-driven sensing though, in that missing a signal can cause errors, although this can be addressed by an infrequent 'heartbeat' signal to avoid large measurement errors.

To measure the periodicity of each unique Thing sampled, the timestamps of the last 5 observations, 'dweets', from each Thing were retrieved over a 24 hour period (this was the limit of the Dweet.io Application Programming Interface (API)). It was observed from the available streams of Things that 62.1% were non-periodic while 37.9% were periodic.

However, to take into account slight fluctuations in the period that could be a result of transmission delays caused by the communication medium, processing delays or highly precise timestamps, an alternative statistical measure besides the percentages of periodic and non-periodic schemata, the Median Absolute Deviation (MAD) of periods was employed. The MAD of periods, the median of the absolute deviations from the median of periods, is more robust to fluctuations and outliers than the mean or standard deviation. Given the set of periods (the difference between subsequent timestamps) is X, then equation 3.1 defines the MAD of the set X.

$$\text{MAD}(X) = \text{median}(|X - \text{median}(X)|) \tag{3.1}$$

A smaller MAD reflects a greater periodicity and in particular, the percentage of data streams with an MAD of zero reflects a measure of the percentage of approximately periodic IoT time-series in that study. 51.58% of the streams sampled, roughly half the streams, had an MAD of zero and were approximately periodic.

3.1.5 Summary of Dweet.io IoT Data Characteristics

From the sample of 11,560 public IoT schemata from Dweet.io, the following characteristics of IoT data were observed. IoT data tends to be flat and wide. The width of data though, is seldom very wide and is mostly less than 8. The data type of values is largely numerical and roughly half the data streams exhibited non-periodic characteristics while the other half exhibited approximately periodic character. All schemata and collected streams have been published as research data within an open repository[4].

To verify the results of this study and to investigate a range of real-world data across the IoT, so that any possible effects of hobbyist data within the Dweet.io dataset are reduced, a cross-IoT study was performed and is elaborated on in the following section.

3.2 Study of Cross-domain IoT Data and Schemata

The purpose of performing a series of cross-IoT studies was to confirm the characteristics of IoT data across application domains by investigating a broad range of IoT platforms, many of which contain data from numerous real-world application deployments. The study was not intended to be exhaustive by any means but to be representative of IoT data across the range of application themes and domains from Section 2.1.2 with smart cities, smart buildings, environmental and disaster monitoring, smart living, health and general public data streams. Industrial and transportation IoT streams are also studied in Section 5.2.3. All the streams studied could be identified as time-series.

The results of the studies are summarised in Table 3.1 and described in detail in the following sub-sections. Each row of the table presents the details of a study in this order: the name, the reference to the relevant sub-section and the total number of schemata. This is followed by the percentage of flat schemata, the percentage of wide schemata, the percentage of numerical attributes, the percentage of periodic data streams and the percentage of approximately periodic streams with a Median Absolute Deviation (MAD) of periods of zero as explained previously in Section 3.1.4. This cross-IoT dataset was published as research data supporting this thesis through an open repository[5].

[4]http://dx.doi.org/10.5258/SOTON/D0076
[5]http://doi.org/10.5258/SOTON/D0202

Study Details			Characteristics (%)				
	Section	#[a]	Flat	Wide	Num	Periodic	0_{MAD}[b]
SparkFun	3.2.1	614	100.0	76.3	93.5	0.0	27.6
Array of Things	3.2.2	18	100.0	50.0	100.0	0.0	100.0
LSD Blizzard	3.2.3	4702	100.0	98.8	97.0	0.004	91.8
OpenEnergy Monitor	3.2.4	9033	100.0	52.5	100.0	-	-
ThingSpeak	3.2.5	9007	100.0	84.1	83.2	0.004	46.9
OH: Fitbit	3.2.6	19	94.7	10.5	92.0	100.0	100.0
OH: RunKeeper	3.2.6	1	100.0	100.0	75.0	0.0	0.0[c]
OH: Moves	3.2.6	1	0.0	-	-	-	-
OH: HealthKit	3.2.6	15	86.7	92.3	52.0	0.0	15.1

[a]Number of unique schemata
[b]Percentage with Median Absolute Deviation (MAD) of zero (approximately periodic time-series)
[c]Although there was a single schema for RunKeeper, this percentage was derived by looking across the data streams of the 56 members who shared data publicly.

TABLE 3.1: Summary of the Cross-IoT Study on Characteristics of IoT Data

3.2.1 SparkFun: Public Streams from Arduino Devices

SparkFun is a manufacturer and retailer of micro-controller boards especially Arduino devices which are popular for prototyping and developing IoT projects and Things. SparkFun provides a SparkFun Data Service[6] that supports the publishing of observations from Things to public data streams in the cloud.

A sample of 614 unique Things schemata was obtained in January 2016 (over the same period as the Dweet.io dataset) from SparkFun. Only flat schemata were supported while most schemata sampled were wide (76.3%) and 93.5% of the attributes were numerical while only 4.5% were string attributes.

SparkFun allowed the storage and retrieval of much longer time-series sequences than Dweet.io. A history of up to 50MB was allowed, though inserts were limited in quantity to a 100 pushes within a 15 minute window, which together with the greater length of collected streams, hence a greater window for delays, were possible causes for none of the streams being periodic. However, 27.6% were approximately periodic with a Median Absolute Deviation of zero.

3.2.2 Array of Things: A "Fitness Tracker" for Smart Cities

The Array of Things (AoT)[7] is an urban sensing project in which a network of nodes, each containing a modular array of sensors, were deployed around Chicago City in the United States, as part of a smart city development. The nodes measure various factors

[6]https://data.sparkfun.com/streams
[7]https://arrayofthings.github.io/

including the climate, traffic, air quality and noise that impact the efficiency of and how liveable the city is.

A sample of 18 unique sensor schemata from nodes that made up an AoT sensor network were obtained in July 2017. All the schemata sampled were flat while half the schemata sampled were wide. All the attributes from the sampled sensor schemata were numerical. Although 0% of the data streams were periodic, this was a result of slight fluctuations in the period of observations and 100% of the streams were approximately periodic with a Median Absolute Deviation (MAD) of zero.

Each AoT node contained an array of sensor modules[8] and each produced a unique flat schema and data stream with a fixed sampling interval. Delays in transmission caused slight fluctuations in the recorded intervals. Some sensors like the Honeywell HIH4030 humidity sensor measured a single quantity, the relative humidity. Other sensors measured more attributes, like the Honeywell HMC5883L magnetic field sensor to detect heavy vehicle flow that measured three attributes corresponding to the three dimensional axes. Finally, the ChemSense sensor module aggregated five individual air quality sensors data into a single stream forming a schema with a width of 7 (additional attributes containing oxidising and reducing gas concentrations were calculated). Half the sensors produced more than a single attribute and hence only half the schemata were wide. If properties like temperature and light intensity were aggregated among the sensors though, all the schemata would be wide.

3.2.3 Linked Sensor Data: Inclement Weather Measurements

Patni et al. (2010) published the Linked Sensor Data (LSD) dataset that describes sensor data from weather stations across the United States with recorded observations from periods of bad weather. This study analysed schemata from 4702 weather stations during the Nevada Blizzard that were collected from the 1st to the 6th of April 2003. Data streams harvested included more than 647,000 rows of observations with over 4.3 million attribute-value pairs from the stations.

Each of the weather stations recorded a specific set of measurements and produced a unique schema. All of the schemata were flat and most of the schemata, 98.8%, were wide with two or more attributes measured. The stations had measurements that produced either numerical or boolean types of values of which 97.0% of the attributes measured had numerical data types. Although only 0.004% of the stations produced periodic streams, a majority of 91.8% of the stations had fixed sampling rates and recorded approximately periodic observations with Median Absolute Deviation of zero.

[8] https://arrayofthings.github.io/node.html

3.2.4 OpenEnergyMonitor: Home Energy Consumption

The OpenEnergyMonitor system consists of open-hardware electricity, temperature and humidity monitoring units for analysing home energy consumption and environmental data and can be deployed as part of smart homes or smart energy grid systems. EmonCms.org[9] is the companion cloud-based data portal for the OpenEnergyMonitor project that supports applications, dashboards and analytics on the data streams. A sample of 9,033 schemata were retrieved from 2,529 users who published public data feeds of their monitoring systems in July 2017.

The schemata were reconstructed from feeds according to the following methodology. Each of the users sampled had published a list of public feeds, whereby each feed had a tag, a timestamp and a value. Feeds from the same user under the same tag and with the same timestamp were merged as individual attributes of a single schema.

All schemata were flat and a slight majority of 52.5% were wide. If each user's feeds were merged to form a single schemata per user though, 85.2% of the schemata would be wide. All attributes were numerical. It was also not possible to determine the periodicity of streams as EmonCms restricted the format of data retrieved to have a fixed, user-specified period as this was the main use case for energy monitoring dashboards.

3.2.5 ThingSpeak: A MatLab-based IoT Analytics Platform

ThingSpeak is a cloud-based IoT platform service developed by the company behind MatLab, a technical software used in scientific and engineering communities. ThingSpeak allows the aggregation, visualisation and analysis of IoT data streams through integration with MatLab and other applications. A sample of 9,007 unique schemata were collected from public channels on ThingSpeak[10] in July 2017. Each channel on ThingSpeak also provides more metadata than on Dweet.io like the latitude and longitude, a Uniform Resource Locator (URL) and a description.

There was only support for flat schemata on ThingSpeak while a majority of 84.1% of schemata were wide. Most of the attributes (83.2%) were numerical. Although the percentage of periodic streams was low at only 0.004%, a larger percentage of about half (46.9%) of the streams recorded approximately periodic observations with a Median Absolute Deviation of periods equal to zero.

[9] https://emoncms.org/
[10] https://thingspeak.com/channels/public

3.2.6 Open Humans: E-Health/Smart-Living with Wearable Data

Open Humans[11] is a research and citizen science initiative in which members gather and share data about themselves. Public smart living datasets from four wearable and smartphone applications, namely the Fitbit wearable[12], Apple's Health Kit[13], RunKeeper[14] and Moves[15] were collected, totalling 278 entries.

With the Fitbit data, 19 schemata were derived from each member's data. These included activities, sleep, heart rate and additional body weight measures from user input and the Aria Smart Bathroom Scale[16]. 94.7% of the schemata were flat with only the heart rate measures having a complex schema. The schemata for activity tracking and sleep were designed such that each attribute was represented in a separate schema, hence, only 10.5% of schemata were wide. If all the activities and sleep schemata were merged, all schemata would be wide and there would be a total of five schemata. Attributes were mainly numerical (92.0%) while the data retrieved was already aggregated by the application and hence periodic, with a period of one day.

Each fitness activity tracked by RunKeeper contained a path made up of a time-series with flat, wide schema that contained three numerical attributes of four attributes besides the timestamp, namely the latitude, longitude, altitude and a type string-attribute. The time-series of the activities, made up primarily of Global Positioning System (GPS) points, produced by 56 members who shared public data, were all non-periodic and none were approximately periodic with a Median Absolute Deviation of zero.

The Moves App produced a single complex schema recording data within a hierarchy of a storyline, segments, activities and individual tracking points. Certain activities produced a time-series of individual geo-spatial tracking points that was flat, wide, entirely numerical and non-periodic, quite similar to a RunKeeper activity.

15 unique schemata were determined from HealthKit data which were from four groups of base schema: Category, Correlation, Quantity and Workout. Correlation and Workout types of schemata were complex while Category and Quantity types were flat. As most schemata from Open Humans public data were the Quantity type of schema (e.g. heart rate, step count or blood glucose), 86.7% of HealthKit schemata were flat. 92.3% of the flat schemata were wide, while about half the attributes (52.0%) were numerical as schemata contained value and unit attributes, with unit attributes being string-attributes. None of the data streams were exactly periodic while 15.1% were approximately periodic (e.g. heart rate monitoring).

[11]https://www.openhumans.org/
[12]https://www.fitbit.com/
[13]https://developer.apple.com/healthkit/
[14]https://runkeeper.com/
[15]https://moves-app.com/
[16]https://www.fitbit.com/aria

3.2.7 A Summary of the Cross-IoT Study Data Characteristics

The cross-IoT study of a more diverse sample of IoT data confirmed that IoT data exhibited certain characteristics initially informed by the Dweet.io study. There was strong cross-source support that IoT data tended to be flat rather than complex. IoT schemata also tended to be wide, even more so if there was aggregation or grouping of multiple sensors. Numerical data types were also the most common. Finally, IoT database management systems have to account for both evenly-spaced, approximately-periodic and unevenly-spaced, non-periodic data as both were present in the study results.

3.3 Data Models for Interoperability in the IoT

In McKinsey's report on unlocking the potential of the IoT, Manyika et al. (2015) analysed over 150 use cases across domains and concluded with estimates that interoperability will unlock an additional 40 to 60 percent of the total projected IoT market value. Milenkovic (2015) argued that metadata, which provides context to the data reported by sensors, is essential for achieving this interoperability in the IoT. Efforts like that led by the W3C's Web of Things Working Group also proposed to utilise 'metadata that enables easy integration across IoT platforms' as a means for countering fragmentation and promoting interoperability in the IoT[17].

This section seeks to study how metadata is represented in a range of data models proposed across application domains and organisations for the IoT. In particular, this section seeks to understand if there is a common structure underlying the metamodels produced to represent metadata and data by the IoT community. A metamodel can be defined as the model of the data model, an abstraction highlighting properties of the data model itself.

3.3.1 Study of the Structure of IoT Metamodels

Looking across the spectrum of IoT standardisation efforts, three categories of thinking about models for data and metadata were determined based on the three perspectives of the IoT introduced in the pioneering and most widely referenced IoT survey by Atzori et al. (2010) and adapted to use the terminology of the IoT semantic interoperability community. The first was an entity-centric mentality, which corresponds to the internet-oriented perspective, where the relations between various entities, both physical and virtual, were modelled in a bottom-up approach with reference to their application domain. Examples are presented in Section 3.3.1.1. Section 3.3.1.2 presents examples from object-centric models, which corresponds to the Things-oriented perspective, where

[17]http://w3c.github.io/wot/charters/wot-white-paper-2016.html

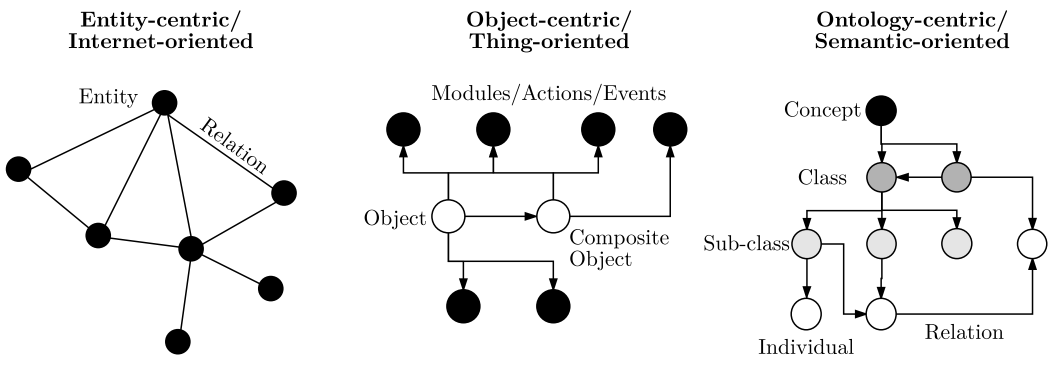

Entity-centric/ Internet-oriented
Entity
Relation
Object-centric/ Thing-oriented
Modules/Actions/Events
Object
Composite Object
Ontology-centric/ Semantic-oriented
Concept
Class
Sub-class
Individual
Relation

FIGURE 3.4: Categories of Metamodel Structures Visualised

Model Name	Domain Specified	Section	Structure	Metamodel
Haystack	Energy & Buildings	3.3.1.1	Graph	Tags
OGC SensorThings	Geospatial Data	3.3.1.1	Graph	Entities
IPSO Objects	Smart Objects	3.3.1.2	Tree	Objects
oneM2M SDT	Home Appliances	3.3.1.2	Tree	Devices
Thing Description	Web of Things	3.3.1.2	Graph	RDF
ETSI SAREF	Smart Appliances	3.3.1.3	Graph	RDF
W3C SSN	Sensors & Actuators	3.3.1.3	Graph	RDF
IoT-O	6 Core Domains	3.3.1.3	Graph	RDF
LOV4IoT	Cross Domain	3.3.1.3	Graph	RDF

TABLE 3.2: Interoperability Models, their Metamodels and Structure across the IoT

the object, perhaps a device or sensor, is the centre of a model which expands and forms composites with other objects. Finally, the ontology-centric approach in Section 3.3.1.3, which corresponds to the semantic-oriented perspective, draws from a top-down modelling of an information model to represent IoT-related concepts and their relations. Table 3.2 shows a summary of the example models, their metamodels and structure.

Figure 3.4 shows a visualisation of the three models. The entity-centric model is made up of a network of entities and relations while the object-centric model is focused on objects and their related actions and events. The ontology-centric model shows how a class structure of concepts helps to define an information model for IoT ‘individuals’, a term used to represent instances of classes in ontology design[18], and their relations.

3.3.1.1 Entity-Centric Models

Project Haystack[19] from the building management systems application domain uses a model of entities, abstractions for physical objects in the real world, and tags, key-value

[18]https://www.w3.org/TR/owl-ref/#Individual
[19]http://project-haystack.org/

pairs applied to an entity, to build a metadata model. Entities have multiple relationships defined using reference tags forming a graph structure. The basic entities defined in the model are from the building management domain and include *site*, *equipment* and *point*. *Point* is the actual sensor or actuator associated with the equipment.

The Open Geospatial Consortium's (OGC) SensorThings standard[20] defines entities like a Thing, Sensor or Location, each containing properties and connected together by relations taking the form of a navigation link. By traversing links and retrieving entities from Representational State Transfer (REST) endpoints, a graph of metadata and data can be formed.

3.3.1.2 Object-Centric Models

Internet Protocol for Smart Objects (IPSO Objects) aims to create common object model for IoT devices and applications[21] by building on the Open Mobile Alliance's Lightweight Specification (OMA LWM2M)[22] to create a repository of generic smart object definitions for interoperability. Examples from the starter pack of IPSO object definitions include a humidity sensor, a light control and an accelerometer among others, each including provision for optional metadata like units. Composite objects, consisting of uni-directional object links from one object to another, form a tree structure within the model.

The Smart Device Template (SDT) schema adopted in a technical specification for home appliances by the OneM2M (2016) standards body allows templates for devices to be defined for different domains. A device is then associated with modules that are associated with properties, events, actions and data points. A tree hierarchy from the domain and device nodes are thus formed.

The Web of Things (WoT) Thing Description defines a model for representing the metadata and a functional description of the interface of a Thing[23]. The common vocabulary contains classes like 'Thing', 'Action' and 'Event' and relies on an underlying Resource Description Framework (RDF) graph model to represent these.

3.3.1.3 Ontology-Centric Models

The Smart Appliances Reference Ontology (SAREF) is an ontology for IoT smart appliances published as a standard by the SmartM2M (2017) group of the European Telecommunications Standards Institute and extended for the energy, building and environment domains. The ontology models concepts of devices, services, tasks and properties. These

[20]https://github.com/opengeospatial/sensorthings
[21]https://github.com/IPSO-Alliance/pub
[22]http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html
[23]https://www.w3.org/TR/wot-thing-description/

concepts are related to each other by predicates within a Resource Description Framework (RDF) graph. The Semantic Sensor Network (SSN) Ontology is a similar ontology with an underlying RDF model for describing sensors, observations, actuators and sampling[24] by the W3C.

The IoT-O ontology introduced in the paper by Seydoux et al. (2016) drew from and referenced many existing IoT ontologies including the SSN and SAREF ontologies to build a modular ontology where each module covers a specific core-domain. These domains included sensing, actuation, lifecycles, services and energy. The design of the ontology followed a known methodology (Suárez-Figueroa, 2010), where a requirements gathering phase drew conceptual (the concepts) and functional (the structure and design) requirements from 8 different IoT ontologies which also included the SPITFIRE ontology by Pfisterer et al. (2011), the oneM2M base ontology[25] and the IoT-Lite ontology by Bermudez-Edo et al. (2016). The underlying model is also an RDF graph.

The Linked Open Vocabularies for Internet of Things (LOV4IoT) dataset published by Gyrard et al. (2016) which comes with a catalogue of 354 ontologies relevant to the IoT[26] helps power a framework that aids the design of cross-domain IoT applications by reusing domain knowledge included in ontologies. The base ontology that connects other ontologies extends concepts from the SSN ontology. The catalogue and output from the framework are RDF graphs.

There is emerging work by Koster (2017) as part of a schema.org working group to provide IoT definitions and a vocabulary that will help reconcile ontology-centric information models and object-centric models like the Thing Description through semantic annotation. The vocabulary used in the parent schema.org project is largely derived from RDF Schema and the RDF graph as a knowledge representation data model.

3.3.2 A Common Structure: The Graph Model

The structure of IoT models observed from the examples in Table 3.2 were either a graph or a tree. A tree is a restricted, hierarchical form of a directed graph without cycles where a child node can have only one parent. Formats like YANG (Schonwalder et al., 2010), Document Type Definition (DTD)[27], JSON Schema[28] and JSON Content Rules (Newton and Cordell, 2017) allow tree model schema to be defined and validated as building blocks for metadata interoperability within models such as those in Section 3.3.1. JSON, the Concise Binary Object Representation (Bormann and Hoffman, 2013) and the eXtensible Markup Language (XML) are some examples of how tree models can be serialised for exchange in the IoT.

[24]https://www.w3.org/TR/vocab-ssn/
[25]http://www.onem2m.org/ontology/Base_Ontology/
[26]http://www.sensormeasurement.appspot.com/?p=ontologies
[27]https://www.w3.org/TR/xml/
[28]http://json-schema.org/

The graph model can be realised either as a property graph, where nodes can also store properties (key-value pairs), or as a general graph like a Resource Description Framework (RDF) graph where all properties are first-class nodes as well. RDF Schema provides a data-modelling vocabulary and simple semantics for the RDF graph[29] while the Web Ontology Language (OWL) allows more complex modelling and non-trivial automated reasoning[30]. The ontologies in Section 3.3.1.3 were built on RDF schema, providing a basis for metadata interoperability in RDF graphs. RDF can also be serialised as a similar variety of formats like JSON-LD[31], XML and RDF Binary using Thrift[32].

Hence, both tree and graph models offer support for rich data and metadata modelling in the IoT with support for a range of schema definition, validation and serialisation formats. As an abstraction of a common structure underlying many standardisations across the IoT for the representation of metadata, the graph model, which can also represent tree structures, is less restrictive. Entities, objects and concepts can be represented as vertices in the graph while relations between them can be represented as edges of the graph. Hence, each of the three entity-centric, object-centric and ontology-centric models can be represented within a graph structure. The next section analyses, from a database systems perspective, the characteristics of RDF graph data and model design in the IoT to establish an understanding of graphs for IoT information representation.

3.4 Characteristics of IoT RDF Graph Data

As established in Section 3.3.2, the graph model is a good abstraction of a common structure for rich metadata and data representation in the IoT. It was also observed from the examples in Sections 3.3.1.2 and 3.3.1.3 that both object-centric and ontology-centric IoT models have used Resource Description Framework (RDF) graphs as the underlying data model, indicating an availability of IoT-related RDF work.

However, the graph model adds a layer of complexity on top of IoT data which was observed from the studies in Section 3.1 to be largely flat. From a database systems perspective, the graph which can integrate both IoT metadata and data within the same structure presents with it certain inefficiencies, the chief of them being the disproportionate expansion of metadata, explained through the study that follows.

3.4.1 Categories of IoT RDF Data from Ontology Sample

Analysing the schemata that a sample of IoT ontologies proposed led to the observation that the RDF graphs produced contained 3 main categories of data:

[29]https://www.w3.org/TR/rdf-schema/
[30]https://www.w3.org/OWL/
[31]https://json-ld.org/
[32]https://afs.github.io/rdf-thrift/

Data Category	Sample RDF Triples

Domain/Device metadata	`sensor1 ssw:processLocation point1`
	`point1 wgs:lat "40.82944"`

Observation metadata	`obs1 a weather:RainfallObservation`
	`obs1 ssw:result data1`
	`obs1 ssw:samplingTime time1`
	`data1 ssw:uom weather:centimeters`

Observation data	`data1 ssw:floatValue "0.1"`
	`time1 time:inXSDDateTime "2017-06-01T15:46:08"`

Table 3.3: Data Categories of a Sample of RDF Triples from Rainfall Observations

- *domain and device metadata* like the location and specifications of sensors,
- *observation metadata* like the units of measure and types of observation and
- *observation data* like timestamps and actual readings.

The Semantic Sensor Web (SSW) project (Sheth et al., 2008), for example, uses an ontology built on the Open Geospatial Consortium's (OGC) observations and measurements model that consists of RDF data of these three categories. Table 3.3 shows a sample of RDF triples from a rainfall observation in the Linked Sensor Data (LSD) dataset recording weather station readings (Patni et al., 2010), adopting the SSW ontology.

Other IoT ontologies from standards bodies, like SAREF and SSN described in Section 3.3.1.3, have a similar structure. SAREF has the concept of devices related to measurements which are directly related to actual data values, while the SSN has systems related to observations and results related to actual data values. Streaming ontologies like the Stream Annotation Ontology (SAO), described by Kolozali et al. (2014), also have a similar structure of a sensor related to a piece of stream data, either a point or a segment, which then relate to actual data values.

Systems can then build on these ontologies to include domain-specific metadata and knowledge. Gray et al. (2011), for example, described the SemSorGrid4Env system used within a flood emergency planning scenario that references a network of ontologies including the SSN to describe its sensor network and upper ontologies, like the Semantic Web for Earth and Environmental Terminology (SWEET) ontologies, to represent domain-specific infrastructural services and datasets.

The graph structure from the design of ontologies presented above is advantageous from a database systems perspective in that rich device and domain metadata that encourages interoperability can be stored just once within the graph and linked to. Observation metadata, however, grows with the volume of observation data within the graph. If the proportion of observation metadata to observation data is large, then there is an undesirable expansion of metadata as the volume of IoT data increases.

3.4.2 Metadata-to-Data Study on IoT RDF Data

In RDF graphs, observation metadata serves the purpose of connecting observation data and hence, has to be repeated with each attribute from each row of observation data. Figure 3.5 expands on the example from Table 3.3 to show how another row of observation data will produce another observation, '`obs2`', with additional observation metadata ('`obs2`','`data2`','`time2`') to connect the graph. Therefore, observation metadata grows with the volume of observation data within the graph.

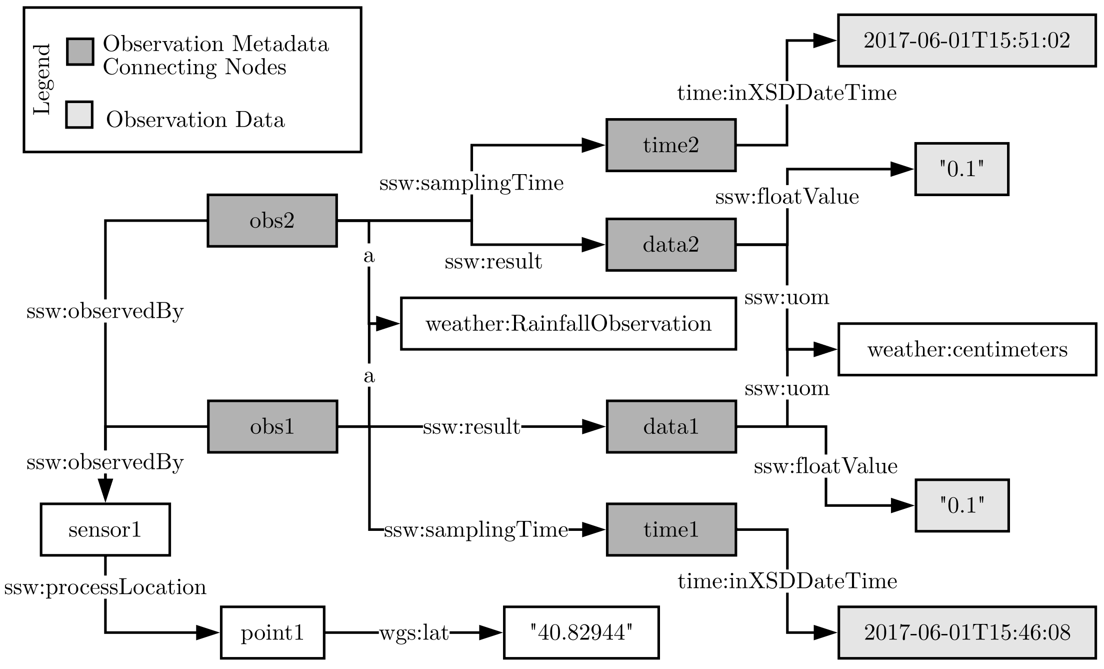


Figure 3.5: An Example IoT Graph from Table 3.3 Showing Metadata Expansion

This section investigates IoT RDF data from different domains and utilising different ontologies to determine the ratio of metadata-to-data. Table 3.4 summarises the results from each dataset showing the domain, ontology, total number of triples and ratio of metadata-to-data triples as categorised in Section 3.4.1.

The Linked Sensor Data (LSD) dataset consisted of a period where there was a blizzard in Nevada, United States, where 108,830,518 triples were recorded from weather stations around the country and another period during Hurricane Ike produced a dataset of 534,469,024 triples. Only 12.5% of each dataset using the same base SSW ontology were observation data, while 0.17% were device metadata and the remaining majority of 87.3% each were observation metadata.

Dataset	Domain	Ontology[a]	Triples (10^6)	Ratio[b]
LSD Blizzard	Environment	SSW	108.8	7:1
LSD Hurricane Ike	Environment	SSW	534.5	7:1
Smart Home Analytics	Smart Home	SSN	11.2	4.5:1
CityPulse Parking	Smart City	SAO	0.7	5.9:1
CityPulse Weather	Smart City	SAO	0.2	2:1
CityPulse Traffic	Smart City	SAO + CT	488.8	3.5:1
CityPulse Pollution	Smart City	SAO	552.2	6:1
CityPulse Events	Smart City	SAO + CT	0.02	1.8:1

[a]SSW = Semantic Sensor Web, SAO = Stream Annotation Ontology, CT = City Traffic
[b]Ratio of Metadata-to-Data, where metadata includes domain/device and observation metadata

Table 3.4: Metadata-to-Data Ratio across RDF Datasets

The Smart Home Analytics Benchmark dataset, described in Appendix B, was based on a different ontology, the SSN ontology, but exhibited a similarly large percentage, 81.7%, of observation metadata. There were 11,157,281 triples in total.

The CityPulse Project produced a set of semantically-annotated RDF graphs from smart city streams of data described in the publication by Bischof et al. (2014). The Stream Annotation Ontology (SAO) (Kolozali et al., 2014) and the City Traffic (CT) ontology[33] were used to describe parking, traffic, weather, pollution and event data from Aarhus in Denmark. A parking stream from May to November 2014 had 759,891 triples of which 14.5% were observation data while 85.5% were metadata. A weather stream from Aarhus in February to June and August to September 2014 had 220,066 triples, 33.5% of which were observation data and 66.5% metadata. The traffic streams from February to June 2014 contained 488,778,997 triples of which 22.2% were observation data and the rest metadata. The pollution streams contained 552,169,405 triples over the period of August to October 2014 whereby 14.3% were observation data triples while 85.7% were metadata. Finally, there were events streams from Surrey, United Kingdom and from Aarhus indicating events in their respective cities counting 23,359 triples in total. 35.6% of the triples were event data including the geospatial position, description and a link to the event's webpage, while 64.4% were metadata triples.

An observation made across all the datasets was that observation metadata, which connected domain metadata and device metadata with time and measurement data, consisted of identifiers. In the example in Figure 3.3, these identifiers are simplified as `obs1`, `data1` and `time1`. However, in practice as observed from this set of datasets, publishers of RDF IoT data often realise these identifiers by generating 128-bit Universally Unique Identifiers (UUIDs) and appending these at the end of Uniform Resource Identifiers (URIs) namespaces. This observation is in agreement with the results in Section 3.1.3 which show that the majority of IoT data attributes sampled, some 97.8%, did not

[33]http://iot.ee.surrey.ac.uk:8080/info.html

contain identifiers. These additional structures within the RDF graph contributed to metadata expansion.

Hence, as seen from the metadata-to-data ratios across RDF datasets, representing IoT data within RDF graphs causes metadata expansion based on the ontology guiding the design of its schema. As the volume of data increases, the ratio remains constant as can be seen from weather observations in the LSD datasets, however, the size of metadata is still many times larger, presenting problems with storing and processing the data.

3.5 Conclusions

From several studies on public, in-use sources of Internet of Things (IoT) data and an analysis of schemata, the current prevalent characteristics of data that emerged were that they are flat, wide and numerical. Periodicity was also shown to vary in streams of time-series data. However, a study of data models in the IoT, driven by the goal of greater interoperability, showed that a common structure underlying many standards was a graph model that could represent rich metadata and data. The Resource Description Framework (RDF) graph is an example of such a implementation. If a graph model were to be used in the IoT for greater interoperability though, an efficient means of storing and querying data needs to be developed, taking advantage of the characteristics of IoT data and mitigating the issue of metadata expansion. The work in the following chapters seek to address this.

Chapter 4

Map-Match-Operate Abstract Graph Querying Framework

"Everything is expressed through relationship. Colour can exist only through other colours, dimension through other dimensions, position through other positions that oppose them. That is why I regard relationship as the principal thing."

— *Piet Mondrian*

The premise of this chapter's work is to examine the appropriateness of the graph model for Internet of Things (IoT) metadata and data in terms of performance. A solution to address the mismatch between the characteristics of IoT datasets and the characteristics of graph models that store rich metadata for interoperability in the IoT, both established in Chapter 3, is proposed. Section 4.1 identifies and introduces the mismatch and proposes a solution to translate graph queries for the efficient execution on time-series IoT data. Section 4.2 then formally defines the Map-Match-Operate solution proposed and discusses its application on an example Resource Description Framework (RDF) graph model, accepting SPARQL graph queries while utilising a relational database for storing time-series IoT data. Section 4.3 describes experiments to benchmark the storage size and performance of this query translation method against RDF databases and state-of-the-art RDF translation methods. Section 4.4 discusses the results of benchmarks while Section 4.5 concludes with a summary of findings in this chapter.

Section 4.2.1.1 on formalising a data model mapping language was published in the Siow et al. (2016a) paper. Sections 4.3 and 4.4 on comparing storage and query performance were published in the Siow et al. (2016c) paper.

4.1 Introduction

The graph is a flexible means of expressing concepts and relationships that allows metadata and data to be modelled, stored and queried together in the IoT. Intuitively, this seems to be a boon for interoperability between devices and applications from different vendors across different domains, while studies in the previous chapter showed that data models by standards bodies seemed to concur and tended to converge on an underlying graph model. However, actual implementations of the graph model as Resource Description Framework (RDF) graphs suffered from metadata expansion that resulted in observation metadata occupying a majority of the graph, from about 2 to 7 times more than actual data, as shown in Section 3.4.2.

On the other hand, current public IoT data from streams were established to be largely made up of observations within a time-series, whereby each observation point of data exhibited flat, wide and numerical characteristics. Conceptually, a compact way of structuring such flat and wide data in two dimensions would be with one dimension representing a row or array of timestamp and connected measurements while the other dimension representing these rows of measurements made over time.

Therefore, there is a mismatch between the graph structure for interoperability, that allows the integration of rich metadata with data, and the conceptual compact structure suitable for IoT time-series data. Furthermore, the RDF graph model presents inefficiencies due to metadata expansion caused by generating identifiers as observation metadata to connect the graph. This larger data size can adversely impact the storage and query performance, especially on resource-constrained IoT hardware.

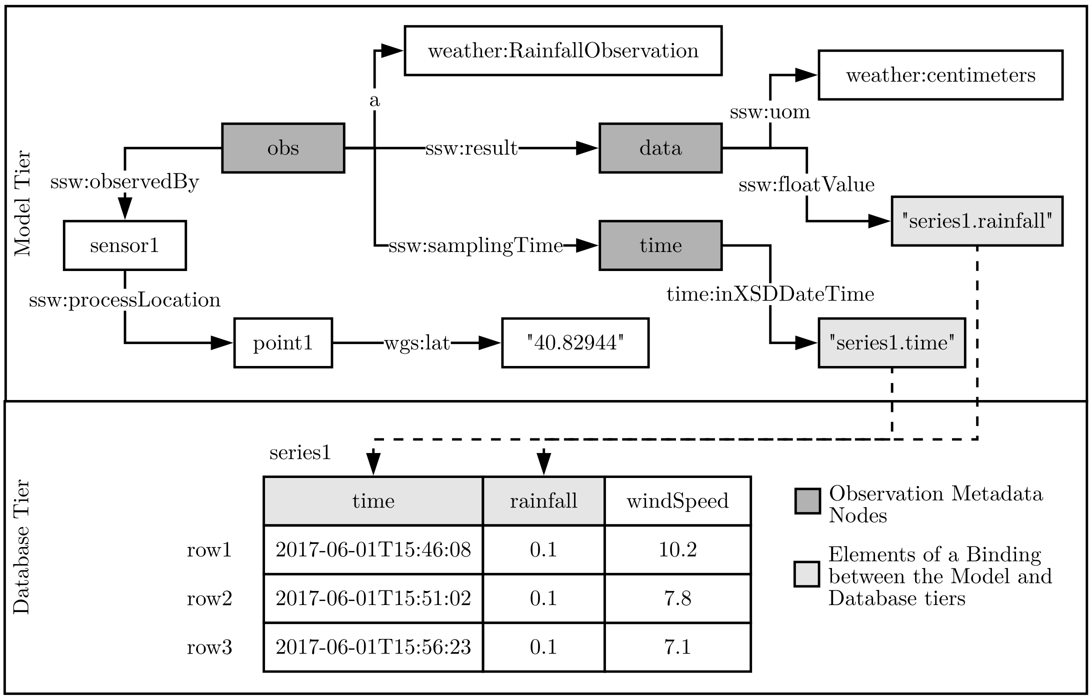
This thesis proposes a solution for this mismatch that structures metadata and data into two tiers, a model tier and a database tier, with a binary relation over these two tiers to connect them. An issued graph query can then be translated by *Map-Match-Operate*, described in the next section, to retrieve results from both tiers. Example application to the RDF graph model helps to explain the abstract *Map-Match-Operate* solution.

4.2 Definition of Map-Match-Operate: A Graph Query Execution Abstraction

A graph query is a means of retrieving content from a graph data structure. In particular, this thesis refers to the specific functionality of subgraph matching to retrieve content from a graph, among those defined by Wood (2012) in a survey of graph query languages.

Map-Match-Operate is an abstraction for graph query execution on data structures used to represent IoT metadata and data. This abstraction allows a graph query, by means of translating the query, to be executed over two-tiers: a rich graph model and a two-dimensional database of IoT data. One dimension within the database consists of rows of measurements made at points in time and another dimension corresponds to the attributes of each row of measurements. Hence, the database tier could also be understood as a set of IoT time-series with flat and wide characteristics.

Figure 4.1 shows how the RDF graph of IoT device metadata, observation metadata and observation data from the example in Table 3.3 and Figure 3.5 can be represented across model and database tiers. Time-series observation data is represented in the database tier in rows, where each row consists of rainfall and wind speed measurements at a particular instant in time. The model tier consists of the graph structure, device and observation metadata and bindings from particular nodes in the graph, for example ‘`"series1.rainfall"`’, to corresponding columns in the database tier, for example ‘`rainfall`’.



	time	rainfall	windSpeed
row1	2017-06-01T15:46:08	0.1	10.2
row2	2017-06-01T15:51:02	0.1	7.8
row3	2017-06-01T15:56:23	0.1	7.1

FIGURE 4.1: An Example IoT Graph from Table 3.3 Across Model and Database Tiers

Observation metadata nodes in the model tier, for example ‘`obs`’, are stored only once for each type of observation within the model tier rather than once for every row, which solves the problem of metadata expansion required for connecting the graph in RDF.

Definition 4.1 formally describes *Map-Match-Operate* while each individual step, map, match and operate, are defined in the Sections 4.2.1, 4.2.2 and 4.2.3 respectively, with the nomenclature of symbols used presented in Table 4.1 for reference. Each step also includes an example application to the RDF graph and the associated SPARQL query language for querying RDF graphs. In these examples, the model tier consists of an RDF graph while the database tier consists of a relational database. The justification for using a relational database is that the tabular data representation in relational databases, consisting of rows and columns, fits the requirement of a compact two-dimensional storage model for flat and wide IoT time-series data. Furthermore, existing literature describing various SPARQL-to-relational query translation methods also provides a platform for comparing the query performance of Map-Match-Operate against the state-of-the-art. Section 4.2.4 describes the details of the S2S engine that implements Map-Match-Operate for SPARQL-to-relational translation and execution.

Definition 4.1 (Map-Match-Operate, μ)**.** Given a database tier, T, which stores a set of time-series, $t \in T$, a graph data model for each time-series, $m \in M$ where M is the model tier, and a query, q, whose intention is to extract data from T and M, the Map-Match-Operate function, $\mu(q, M, T)$, returns an appropriate result set, R, as an answer to the query, q.

Symbol(s)	Description	Definition(s)
μ	Map-Match-Operate Function	4.1
M, m	Model Tier, Individual Graph Model	4.1,4.2
T, t	Database Tier, Individual Time-series	4.1,4.2
C, c	All Columns, Individual Column	4.2
V, E	Vertices of a Model, Edges of a Model	4.2
τ	Set of Timestamps in Time-series	4.2
$\mathbb{B}$	Set of Bindings	4.2
b, b_{RDF}	Binding, Binding for RDF Model	4.2,4.4,4.5,4.6
μ_{map}, μ_{match}, $\mu_{operate}$	Map, Match, Operate Functions	4.2,4.9,4.10
m_{map}	Model Mapping	4.2
m^{s2s}_{map}	S2SML Mapping for RDF Graph	4.3
I_{map}, L_{map}	IRI Map, Literal Map	4.4,4.5
I_C, I_c	Binding Strings in IRI, Binding String	4.4,4.5
I_p	IRI String Parts	4.4,4.6,4.8
F	Faux Node	4.6
U_{id}	Unique Identifier Placeholder	4.6
M^C_{map}	Mapping Closure	4.7,4.10
q, q_{graph}	Graph Query, Graph Subset of Query	4.9,4.10
$\mathbb{B}_{match}, b_{match}$	Set, Elements of Subgraph Matches	4.9,4.10
V_m, E_m	Vertices, Edges within a M^C_{map}	4.9
υ_g	Set of Variables in q_{graph}	4.9
q_{op}	Sequence of Query Operations	4.10
O, O_G	Operation Function, Graph Operator	4.10
$\Pi, \sigma, \cup$	Operators (Özsu and Valduriez, 2011)	-

Table 4.1: Definitions of Symbols in Map-Match-Operate

4.2.1 Map: Binding M to T

A rich graph data model, $m = (V, E)$, consists of a set of vertices, V, and edges, E. A time-series t, consists of a set of timestamps, τ and a set of all columns C where each individual column $c \in C$. Definition 4.2 describes the *map* step on m and t, which are elements of M and T respectively.

Definition 4.2 (Map, μ_{map})**.** The map function, $\mu_{map}(M \times T) \rightarrow \mathbb{B}$, produces a binary relation, $\mathbb{B}$, between the set of vertices, V, and the set of timestamps and columns $(\tau \times C)$. Each element, $b \in \mathbb{B}$, is called a binding and $b = (x, y)$, where $x \in V$ and $y \in (\tau \times C)$.

A data model mapping, m_{map}, is a means of representing the set of bindings, $\mathbb{B}$, within each data model, m, in the model tier, M.

An RDF graph, m_{RDF} is a specific type of graph data model (an instantiation of the abstract m) that consists of a set of triple patterns, $tp = (s, p, o)$, whereby each triple pattern has a subject, s, predicate, p, and an object, o. A triple pattern describes a relationship where a vertex, s, is connected to a vertex, o, via an edge, p. Adopting the notation introduced by Chebotko et al. (2009), $(s, p, o) \in (IB) \times I \times (IBL)$ where I, B and L are pairwise disjoint infinite sets of Internationalised Resource Identifiers (IRIs), blank nodes and literal values respectively. A binding $b_{RDF} = (x_{RDF}, y)$ is an element of $\mathbb{B}_{RDF}$ where $x_{RDF} = (I \times L)$. The data model mapping for RDF, m^{RDF}_{map} (an instantiation of the abstract m_{map}), can itself be expressed as an RDF graph with the S2SML data model mapping language formalised in the next section, Section 4.2.1.1. Two features of S2SML mappings, logical groupings called mapping closures for bottom-up IoT systems and implicit join condition detection within mapping closures are covered in Sections 4.2.1.2 and 4.2.1.3. Finally, S2SML's compatibility with other mapping languages and a comparison of the different syntaxes are discussed in Sections 4.2.1.4 and 4.2.1.5.

4.2.1.1 S2SML: A m^{RDF}_{map} Data Model Mapping Language for RDF

S2SML (Semantic-to-Structured Mapping Language) is a mapping language, developed as part of this thesis, that allows a data model mapping, m^{RDF}_{map}, to be defined as an RDF graph. An S2SML mapping represents the structure and content from the model tier in conjunction with bindings to the database tier. The language was designed to be compatible, through automated translation described in Section 4.2.1.4, with the W3C-recommended Relational Database to RDF Mapping Language (R2RML)[1] and the RDF Mapping language (RML) by Dimou et al. (2014), that extends R2RML beyond relational data sources. S2SML, however, provides additional IoT-schema-specific features for preventing metadata expansion and improving query performance like faux nodes, which contain identifiers only created on projection, and a less verbose syntax.

Hence, this section proceeds to define S2SML. Expanding on the notation that defines I, B, L as sets of Internationalised Resource Identifiers (IRIs), blank nodes and literals,

[1]https://www.w3.org/TR/r2rml/

Symbol	Name	Example
I	IRI	<http://knoesis.wright.edu/ssw/ont/weather.owl#degrees>
I_{map}	IRI Map	<http://knoesis.wright.edu/ssw/{sensors.sensorName}>
B	Blank Node	_:bNodeId
L	Literal	"-111.88222"^^<xsd:float>
L_{map}	Literal Map	"readings.temperature"^^<s2s:literalMap>
F	Faux Node	<http://knoesis.wright.edu/ssw/obs/{readings._uuid}>

Table 4.2: Examples of Elements in (s, p, o) Sets

I_{map} (Definition 4.4), L_{map} (Definition 4.5) and F (Definition 4.6) are pairwise disjoint infinite sets of IRI Maps, Literal Maps and Faux Nodes (nodes specifically designed to represent identifiers that connect observation metadata in IoT RDF graphs, that only materialise as IRIs on projection in queries, and mitigate the metadata expansion problem) respectively. Examples of the types of elements within these sets from Linked Sensor Data (LSD) modelled with the Semantic Sensor Web (SSW) ontology can be found in Table 4.2. Combinations of these terms, for example $I_{map}IBF$, denote the union of their component sets, for example $I_{map} \cup I \cup B \cup F$. Definition 4.3 describes an S2SML mapping, m^{s2s}_{map}, formally. An example of S2SML and a comparison to R2RML is given in Section 4.2.1.5.

Definition 4.3 (S2SML Mapping, m^{s2s}_{map})**.** Given a set of all possible S2SML mappings, M^{s2s}_{map}, an S2SML mapping, $m^{s2s}_{map} \in M^{s2s}_{map}$, is a set of triple tuples, $(s, p, o) \in (I_{map}IBF) \times I \times (I_{map}IBL_{map}LF)$ where s, p and o are the subject, predicate and object respectively.

Using the examples from Table 4.2 as reference, I_{map} elements, as defined in Definition 4.4, form templates for an IRI that consist of the concatenation of an ordered list of strings with two types of elements. These elements are either parts of the IRI string, like '`http://knoesis.wright.edu/ssw/`', or binding strings to data in the database tier, for example, '`{timeSeries.colName}`', which refers to a particular column within a relational database table storing a time-series in the database tier. L_{map} elements, defined in Definition 4.5, are RDF literals with a specific datatype of '`<s2s:literalMap>`' whose value contains a binding to the database tier, in this example, to the column of the table representing a time-series in the format '`timeSeries.colName`'.

Definition 4.4 (IRI Map, I_{map})**.** An I_{map} is a string that forms a template for an IRI which consists of the concatenation of all n elements in an ordered list, $(a_k)^n_{k=1}$, where $n > 1$. Each element, a_k in the list is either part of an IRI string, I_p, or a binding string, I_c, that references an individual column in the database tier, c. A binding $b_{RDF} = (I_{map}, c)$ is formed for each I_c.

Definition 4.5 (Literal Map, L_{map})**.** An L_{map} is defined as an RDF literal that consists of a binding string, I_c, as its value and a datatype IRI of `<s2s:literalMap>` identifying it as an S2SML literal map. A binding $b_{RDF} = (L_{map}, c)$ is formed from the I_c value where c is an individual column in the database tier.

The example of a faux node, F, in Table 4.2 also shows how a placeholder is defined in the format of '`{tableName._uuid}`' with the keyword '`_uuid`' identifying this as a Faux node and this part of the IRI string as a placeholder for a set of generated Universally Unique Identifiers (UUIDs) within the referenced time-series table in the database tier, '`readings`' in this example. Definition 4.6 formally expresses a faux node.

Definition 4.6 (Faux Node, F)**.** A faux node, F, is defined as a string that forms a template for an IRI that consists of the concatenation of all n elements in an ordered list, $(a_k)_{k=1}^{n}$, where $n > 1$. Each element, a_k in the list is either part of an IRI string, I_p, or a placeholder string for a generated identifier, U_{id}, referencing a time-series t in the database tier. If F is projected in a query on a m_{map}^{s2s}, a new column c_{uuid} is generated in the time-series t referenced by U_{id} and a binding $b_{RDF} = (F, c_{uuid})$ is formed in m_{map}^{s2s}.

From the example in Figure 4.1, observation metadata nodes like '`obs`' can be modelled as faux nodes. The advantage of doing so are that they are only created (materialised) as a column within the database tier at query time, when projected within a query. In certain IoT benchmarks like SRBench by Zhang et al. (2012), none of the queries project the observation metadata nodes, hence, the storage space and query execution time can be efficiently reduced with the use of faux nodes. Furthermore, unlike blank nodes, faux nodes, once projected within a query, produce IRIs which are stable and can be exchanged across linked data endpoints. However, in a benchmark like CityBench by Ali et al. (2015), most of the queries project observation metadata nodes and there is no additional benefit in using faux nodes for these queries.

Therefore, each of I_{map}, L_{map} and F in S2SML forms a binding, $b_{RDF} = (x_{RDF}, y)$, from the model tier to the database tier for m_{map}^{s2s} which specifies a m_{map}^{RDF} instantiation of the abstract m_{map}. For I_{map} and F, x_{RDF} is an IRI formed from a string template, while in L_{map}, x_{RDF} is a literal. y for all three are references to a column, c, in the database tier (a new identifier column in the case of F).

4.2.1.2 Mapping Closures: Logical Groupings of Mappings

A mapping closure is a logical group of mappings which can be understood as a single graph consisting of one or more individual mappings. In RDF, each mapping closure is a named graph formed from S2SML mappings. This concept is useful for capturing the use case where several devices work within an IoT system and each publishes individual mappings in a bottom-up fashion. Mappings overlap on common nodes, IRIs in RDF, representing common infrastructure and forming a graph of the entire system within the mapping closure. Definition 4.7 explains a mapping closure.

Definition 4.7 (Mapping Closure, M^C_{map})**.** A mapping closure, M^C_{map}, is composed of individual mappings, m_{map}, from a particular IoT system that produces a set of mappings $M^{system}_{map} = \{m_{map} | m_{map} \in M_{map}\}$, where M_{map} is a set of all possible mappings. The mapping closure is formed from the union of all elements in M^{system}_{map}, so $M^C_{map} = \bigcup_{m \in M^{system}_{map}} m$.

For example, Figure 4.2 shows a graph representation of a mapping of sensors and a rainfall observation mapping. There can also be observation mappings for wind speed and temperature besides this one for rainfall, all connected to a single sensors mapping. Together, this set of mappings forms the mapping closure of a weather station.

4.2.1.3 Implicit Join Conditions by Template Matching

Multiple mappings within a mapping closure might contain a set of bindings to multiple time-series in the database tier. A graph query that retrieves a graph from across multiples mappings might require that columns from multiple time-series be joined to form a result set. In other mapping languages like R2RML and RML, one or more join conditions (‘`rr:joinCondition`’[2]) may be specified between triple maps of different logical tables. In RML, this join condition is between logical sources instead of tables[3].

In S2SML, however, these join conditions are automatically discovered as they are implicit within mapping closures involving mappings with similar IRI maps, I_{map}, which refer to two or more time-series references in the database tier. This is made possible by a process called IRI template matching, formalised in Definition 4.8 as follows.

Definition 4.8 (IRI Template Matching)**.** Given that I_P is the concatenated sequence of strings from the ordered list of only the I_p elements in I_{map} from Definition 4.4. Two I_{map} elements, I_{map_1} and I_{map_2}, with the concatenated sequence of strings I_{P1} and I_{P2} respectively, fulfil an IRI template matching if $I_{P1} = I_{P1}$.

Hence, matching I_{map} elements from different mappings that are bound to different time-series, result in inferred join conditions. This occurs automatically when mappings are added to a mapping closure forming a graph. Figure 4.2 shows a mapping closure with a sensors mapping and a rainfall observation mapping. An element of I_{map} in each of the mappings, ‘`sen:system_{sensors.name}`’ in the sensors mapping, I_{map_1}, and ‘`sen:system_{readings.sensor}`’ in the observation mapping, I_{map_2}, fulfil a template matching as both their I_P are ‘`sen:system_`’. A join condition is inferred between the columns *‘sensors.name’* and *‘readings.sensor’* as a result. When a graph query retrieves this part of the model, with the node in question, a join operation can then be performed on the underlying time-series in the database tier.

[2]https://www.w3.org/TR/r2rml/#foreign-key
[3]http://rml.io/spec.html#logical-join

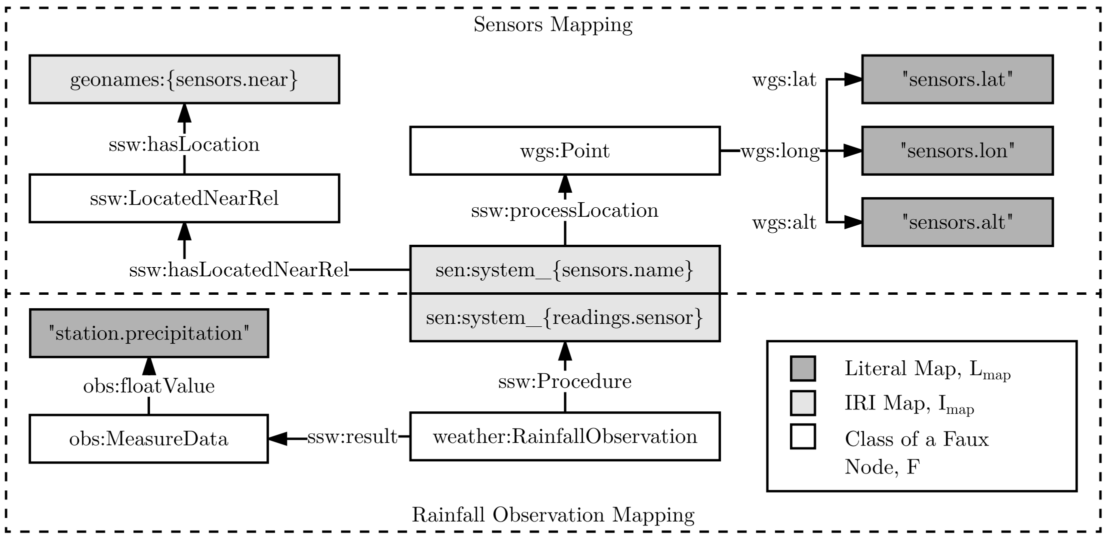


Figure 4.2: Graph Representation of an Implicit Join within a Mapping Closure

4.2.1.4 Compatibility with R2RML and RML

S2SML was designed to be compatible with R2RML and RML through a machine translation service. Triple maps in R2RML and RML are translated to triples in S2SML based on the elements in Table 4.2. For example, an ‘`rr:SubjectMap`’ with an ‘`rr:template`’ is automatically translated as an element of I_{map}. Each subject map in R2RML together with the predicate map and object map within a triple map are translated to a triple in S2SML and vice versa.

Listing 4.1 shows the pseudo-code of an algorithm to perform a basic conversion of S2SML to R2RML. The S2SML data model mapping is already in a global in-memory object (‘`s2sml`’) while the R2RML produced is written to a file (specified by ‘`filename`’). Listing 4.2 shows the pseudo-code of an algorithm to perform the opposite conversion from R2RML (‘`r2rml`’ object) to an S2SML file (specified by ‘`filename`’). Implementations in Java to convert from S2SML to R2RML and vice versa are available online[4],[5].

```
fun GenerateR2RML(filename) { //Converts S2SML to an R2RML file
  for((s,p,o) in s2sml.triples) { //Loops through S2SML triples
    tpMap = GetTripleMapReference(s) //References the relevant triple map
    PutPredicateObjectMap(tpMap,p,o) //Adds p and o maps to the triple map
    r2rml.writeTo(filename)
  }}
fun GetTripleMapReference(node) : TpMapRef {
  var ref = tpMapSet.get(node)
  if(ref == null) {
    val tpMapId = r2rml.generateTpMapId()
    val subjectMapId = r2rml.generateSubjectMapId()
    r2rml.addTurtle("
      $tpMapId a rr:TriplesMap;
```

[4]https://github.com/eugenesiow/lsd-ETL/blob/master/src/main/java/S2SMLtoR2RML.java
[5]https://github.com/eugenesiow/lsd-ETL/blob/master/src/main/java/R2RMLtoS2SML.java

```
        rr:subjectMap $subjectMapId.
      $subjectMapId a rr:Subject;
        ${ProcessNode(node)}.") //Adds R2RML triple map with subject
    ref = TpMapRef(id=tpMapId,subject=subjectMapId)
    tpMapSet.put(ref) }
  return ref }
fun PutPredicateObjectMap(tpMap,p,o) { //Adds a PO Map to the triple map
  if(p.val == "a") {
    r2rml.addTurtle("
      ${tpMap.subjectMapId} rr:class <${o.val}>.")
    return }
  r2rml.addTurtle("
    ${tpMap.id} rr:predicateObjectMap [
        rr:predicateMap [ ${ProcessNode(p)} ];
        rr:objectMap    [ ${ProcessNode(o)} ]; ].")
  if(o.type == "literalMap" || o.type == "IRIMap") { //Adds the logical table
    r2rml.addTurtle("
      ${tpMap.id} rr:logicalTable [ rr:tableName \"${node.tableVal}\" ]") } }
fun ProcessNode(node) : String {
  if(node.type == "IRI") {
    return "rr:constant <${node.val}>"
  } else if(node.type == "literal") {
    return "rr:constant \"${node.val}\""
  } else if(node.type == "literalMap") {
    return "rr:termType rr:Literal; rr:column \"${node.colVal}\";"
  } else if(node.type == "IRIMap") {
    return "rr:termType rr:IRI;
      rr:template \"${node.IRIValPre}${node.colVal}${node.IRIValPost}\";"
  } else if(node.type == "blank") {
    return "rr:parentTriplesMap <${GetTripleMapReference(node).id}>"
  } else if(node.type == "faux") {
    return "rr:termType rr:IRI;
      rr:template \"${node.IRIValPre}${node.uuidCol}${node.IRIValPost}\";"
  } }
```

Listing 4.1: Basic Pseudo-code to Convert from S2SML to R2RML

```
fun GenerateS2SML(filename) { //Converts R2RML to an S2SML file
  results = r2rml.query("PREFIX rr: <http://www.w3.org/ns/r2rml#>
    SELECT * { ?tpMap a rr:TriplesMap;
      rr:subjectMap ?sMap. } ")
  for((tpMap,sMap) in results) { //Loops through R2RML triple maps
    tableName = GetTable() //Gets the logical table
    subject = ProcessMap(tableName,sMap)
    tpMapRef.put(tpMap,subject)
    classStmt = r2rml.getStatement(sMap,"rr:class",null) //Get class
    if(classStmt) s2sml.add(subject,"a",classStmt.object) //Set class
    poResults = r2rml.query("PREFIX rr: <http://www.w3.org/ns/r2rml#>
      SELECT * { <$tpMap> rr:predicateObjectMap ?po.;
        ?po rr:predicateMap ?pMap;
          rr:objectMap ?oMap. } ")
    for((tpMap,sMap) in poResults) { //Convert PO Maps
      predicate = ProcessMap(tableName,pMap)
      object = ProcessMap(tableName,oMap)
      s2sml.add(subject,predicate,object) }
    s2sml.writeTo(filename) }}
fun ProcessMap(table,node) : Node { //Process triples in map
  if(r2rml.contains(node,"rr:termType","rr:Literal")) {
    colStmt = r2rml.getStatement(node,"rr:column",null)
```

R2RML predicate	S2SML example
rr:language	"literal"@en
rr:datatype	"literal"^^<xsd:float>
rr:inverseExpression	"{COL1} = SUBSTRING({{COL2}}, 3)"^^<s2s:inverse>
rr:class	a <ont:class>.
rr:sqlQuery	<context1> {<sen:sys_{table.col}> ?p ?o.} <context1> s2s:sqlQuery "query".

Table 4.3: Other R2RML Predicates and the Corresponding S2SML Construct

```
    res = Literal("$tableName.${colStmt.object}")
  } else if(r2rml.contains(node,"rr:termType","rr:IRI")) {
    tStmt = r2rml.getStatement(node,"rr:template",null)
    res = Resource(tStmt.object.replace("{","{$tableName"))
  } else if(r2rml.contains(node,"rr:constant")) {
    res = r2rml.getStatement(node,"rr:constant",null).object
  } else if(r2rml.contains(node,"rr: parentTriplesMap")) {
    ptStmt = r2rml.getStatement(node,"rr:parentTriplesMap",null)
    res = tpMapRef.get(ptStmt.object) //triple map to subject
  } return res }
```

Listing 4.2: Basic Pseudo-code to Convert from R2RML to S2SML

Table 4.3 defines additional advanced R2RML predicates and the corresponding S2SML construct. ‘rr:inverseExpression’, for example, is encoded within a literal, L_{iv}, with a datatype of ‘<s2s:inverse>’ and the ‘rr:column’ denoted with double braces ‘{{COL2}}’. ‘rr:sqlQuery’ is encoded by generating a context (named graph) to group triples produced from that Triple Map and the query is stored in a literal object with the generated context as the subject and ‘<s2s:sqlQuery>’ as predicate. Faux nodes in S2SML have no related concept in R2RML and by default are translated as IRI templates related to a column named ‘_uuid’. This Universally Unique Identifier (UUID) column needs to be manually generated or can be implemented in the application layer for engines accepting R2RML. The S2SML specification is published online[6].

4.2.1.5 S2SML Example and a Comparison of Mappings

Listings 4.3 and 4.4 show examples of a rainfall observation from a station in the Linked Sensor Data (LSD) dataset modelled as mappings in S2SML and R2RML respectively. Both RDF graphs are written in the compact text form of the Terse RDF Triple Language 1.1 (RDF Turtle[7]). An example instance of a rainfall observation which both mappings model is shown as a graph in Figure 4.3. An RML mapping has the same syntax as R2RML except the ‘rr:LogicalTable’ predicate is replaced by an ‘rml:logicalSource’ predicate to support data sources other than relational tables.

[6]https://github.com/eugenesiow/sparql2sql/wiki/S2SML
[7]https://www.w3.org/TR/turtle/

```
@prefix : <http://example.com/ns#>.
@prefix s2s: <http://iot.soton.ac.uk/s2s/s2sml#>.
@prefix weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>.
@prefix ssw: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>.
:obs{station._uuid} a weather:RainfallObservation;
    ssw:result :res{station._uuid}.
:res{station._uuid} a ssw:MeasureData;
    ssw:floatValue "station.Rainfall"^^<s2s:LiteralMap>.
```

Listing 4.3: S2SML Mapping of a Rainfall Observation using Faux Nodes

```
@prefix : <http://example.com/ns#>.
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>.
@prefix ssw: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>.
:t1 a rr:TriplesMap;
    rr:logicalTable [ rr:tableName "station" ];
    rr:subjectMap [
        rr:template "http://example.com/ns#obs{time}";
        rr:class weather:RainfallObservation ];
    rr:predicateObjectMap [
        rr:predicate ssw:result;
        rr:objectMap [ rr:parentTriplesMap :t2 ]].
:t2 a rr:TriplesMap;
    rr:logicalTable [ rr:tableName "station" ];
    rr:subjectMap [
        rr:template "http://example.com/ns#res{time}";
        rr:class ssw:MeasureData ];
    rr:predicateObjectMap [
        rr:predicate ssw:floatValue;
        rr:objectMap [
            rr:column "Rainfall" ]].
```

Listing 4.4: R2RML Mapping of a Rainfall Observation

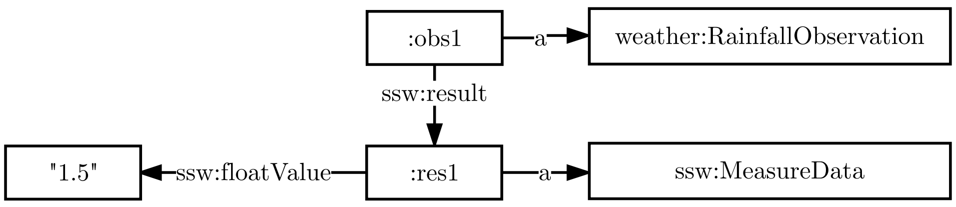


Figure 4.3: Graph Representation of an Example Instance of a Rainfall Observation

The examples show that both mapping languages can be used as a m_{map}^{RDF} data modelling language for the map step in Map-Match-Operate to represent LSD observations. However, the examples also highlight certain differences between S2SML and R2RML. R2RML is more verbose as it uses an RDF graph to model how each triple in the resulting graph model is mapped. S2SML syntax is more succinct as it uses the RDF graph to represent the resulting graph model itself. S2SML also uses faux nodes to model observation metadata like '`:obs{station._uuid}`' while R2RML, in this instance, can either generate a unique identifier column or use an existing column like '`{time}`' within its IRI template. This column effectively acts as a primary key and should be unique, hence, if there are duplicate timestamps, the ‘time’ column cannot be used.

The binding from the model to the database tier is made for the value of rainfall measured in each rainfall observation in S2SML by the ‘station.Rainfall’ literal map. In R2RML, the binding to a relational table is made from the second triple map ‘:t2’ through a chain of predicates from ‘rr:logicalTable’ to ‘rr:tableName’ to the literal value and table name: ‘station’ . The binding to the column in that table is made from the predicate object map with the ‘rr:column’ predicate to the literal ‘Rainfall’. The binding from RML is similar to R2RML with the exception that ‘rr:logicalTable’ is replaced by ‘rml:logicalSource’ and ‘rr:tableName’ by ‘rml:source’.

4.2.2 Match: Retrieving $\mathbb{B}_{match}$ by Matching q_{graph} to M_{map}

The goal of the *match* operation is to perform a subgraph matching of the structure and content within a graph query, q and the model represented by a data model mapping, M_{map}. The result is a set of matched bindings and nodes, $\mathbb{B}_{match}$, from the model subgraph to variables within the query graph.

The M_{map} used by the *match* operation is a union of data model mappings known as a mapping closure, M^C_{map}, from Definition 4.7. Each m_{map} in the mapping closure relates to a subset of time-series in T, the database tier.

The formalisation of the *match* operation is expressed in Definition 4.9. q_{graph} is given as a subset of query, q, that consists of a graph with a set of vertices, V_g, and edges, E_g. v_g is given as the subset of variable vertices and edges from V_g and E_g.

Definition 4.9 (Match, μ_{match})**.** The match function, $\mu_{match}(q_{graph}, M^C_{map})$, performs subgraph matching of q_{graph} to M^C_{map}. Given $q_{graph} = (V_g, E_g)$ and $M^C_{map} = (V_m, E_m)$ where V_g and V_m are finite sets of vertices and E_g and E_m are finite sets of edges, a subgraph match exists if there is a one-to-one mapping $f : V_g \rightarrow V_m$ such that $(u, v) \in E_g \iff (f(u), f(v)) \in E_m$. If f exists, there also exists a relation, $\mathbb{B}_{match}$, between the set of variables from q_{graph}, v_g, and $V_m \cup E_m$. Each element, $b_{match} \in \mathbb{B}_{match}$, is an ordered pair of variable to either vertex or edge, including bindings from M^C_{map}.

A graph query on an RDF graph can be expressed in the SPARQL Query Language for RDF. Listing 4.5 shows a SPARQL query, an example of q, that retrieves rainfall observations with recorded measurements of over 10cm at a weather station. The structure of the RDF graph for an instance of such an observation can be seen in Figure 4.3. The graph to be matched in the query, q_{graph}, is expressed textually as a Basic Graph Pattern (BGP) from lines 5 to 7 in the listing. A BGP consists of a set of triple patterns, tp. Any of the subject, s_{tp}, predicate, p_{tp} or object, o_{tp} in tp can be a query variable from the set v_g within a BGP. In this example, v_g consists of two variables: '`?obs`' and '`?value`'.

```
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>.
PREFIX ssw: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?value WHERE {
  ?obs a weather:RainfallObservation ;
    ssw:result [
      ssw:floatValue ?value ].
    FILTER ( ?value > "10"^^xsd:float)
}
```

Listing 4.5: SPARQL Query to Retrieve Rainfall Observations >10cm from a Station

Hence, the match function, μ_{match}, returns a set of matches, $\mathbb{B}_{match}$, between the query graph and model mapping subgraph. In this example, the data model mapping is from a mapping closure that includes the mapping shown in Listing 4.3. As such, there are b_{match} elements of $\mathbb{B}_{match}$ that match '`?obs`' to '`:obsstation._uuid`', a vertex of V_m in the model mapping which is a faux node, and variable '`?value`' to the binding with the database tier column referred to as '`station.Rainfall`'.

4.2.3 Operate: Executing q's Operators on $T \cup M$ using $\mathbb{B}_{match}$ Results

The final step in Map-Match-Operate is the operate step in Definition 4.10 that uses results from the match step, $\mathbb{B}_{match}$, to execute a sequence of operations, q_{op}, from the query q, on the database and model tiers, $T \cup M$, to return a result set, R.

Definition 4.10 (Operate, $\mu_{operate}$)**.** The operate function, $\mu_{operate}(q, M^C_{map}, T \cup M)$, produces a result set R from a sequence of operations, q_{op}, from the query q. Each individual operation in the sequence q_{op} is in the form of a function, $O(j) = k$, which takes in an input set j and returns an output set k. A specific graph operation function, O_G, takes in an input set, j_G, of q_{graph}, M^C_{map} and $T \cup M$, performs the match function μ_{match} that returns $\mathbb{B}_{match}$ and generates an output set, k_G, with each variable in $\mathbb{B}_{match}$ related to results retrieved from the database and model tiers, $T \cup M$.

To walkthrough the operate step, $\mu_{operate}$, a graph query expressed in SPARQL is shown in Listing 4.6. The graph query specifies a union of three graphs, a graph for snow observations, one for rainfall observations and one for wind speed observations. The intention of the query is to retrieve all weather stations IRIs which are experiencing bad weather with either snow, heavy rainfall or high wind speed. The query is based on query 6 from the SRBench benchmark by Zhang et al. (2012), which uses the Linked Sensor Data (LSD) dataset.

```
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>.
PREFIX ssw: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?station WHERE {
  {
    ?observation ssw:procedure ?station ;
      a weather:SnowObservation ;
      ssw:result [ ssw:boolValue ?value ] ;
      ssw:samplingTime [ time:inXSDDateTime ?time ] .
    FILTER ( ?value == "true" )
    FILTER ( ?time>="2003-04-03T16:00:00"^^xsd:dateTime &&
      ?time<"2003-04-03T17:00:00"^^xsd:dateTime )
  } UNION {
    ?observation ssw:procedure ?station ;
      a weather:RainfallObservation ;
      ssw:result [ ssw:floatValue ?value ] ;
      ssw:samplingTime [ time:inXSDDateTime ?time ] .
    FILTER ( ?value > "30"^^xsd:float )  # centimeters
    FILTER ( ?time>="2003-04-03T16:00:00"^^xsd:dateTime &&
      ?time<"2003-04-03T17:00:00"^^xsd:dateTime)
  } UNION {
    ?observation ssw:procedure ?station ;
      a weather:WindSpeedObservation ;
      ssw:result [ ssw:floatValue ?value ] ;
      ssw:samplingTime [ time:inXSDDateTime ?time ] .
    FILTER ( ?value > "100"^^xsd:float )  # knots
    FILTER (?time>="2003-04-03T16:00:00"^^xsd:dateTime &&
      ?time<"2003-04-03T17:00:00"^^xsd:dateTime)
  }
}
```

Listing 4.6: SPARQL Query to Identify Stations Experiencing Bad Weather

The query can be decomposed to form a tree of operators as shown in Figure 4.4 representing q_{op}. Each node in the tree is an operation function, O, and the sequence of operations in q_{op} is obtained by traversing the tree from its leaf nodes to its root node. A relational algebra vocabulary and notation as described by Özsu and Valduriez (2011) is utilised to denote each operation function. The O symbol is omitted for brevity, for example, the Π function refers to O_{Π} and the G function refers to O_G.

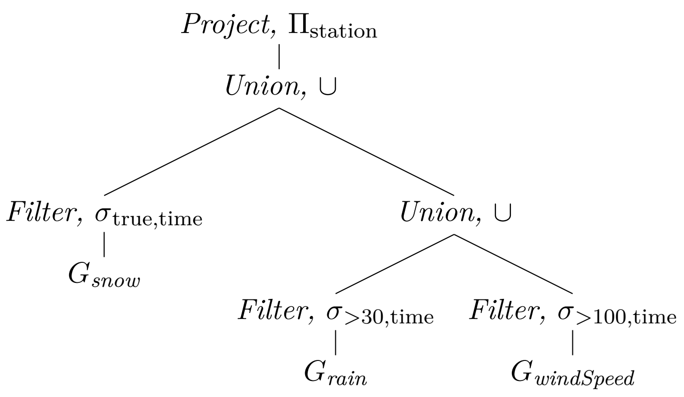

$Project,\ \Pi_{\text{station}}$
$Union,\ \cup$
$Filter,\ \sigma_{\text{true,time}}$
G_{snow}
$Union,\ \cup$
$Filter,\ \sigma_{>30,\text{time}}$
G_{rain}
$Filter,\ \sigma_{>100,\text{time}}$
$G_{windSpeed}$

Figure 4.4: Query Tree of Operators for Listing 4.6's Bad Weather Query

The leaf nodes of the tree are made up of graph operator functions, O_G or simply G, as introduced in Definition 4.10. For example, the G_{rain} operator has a q_{graph} corresponding to lines 15 to 18 in Listing 4.6, which produces a $\mathbb{B}_{match}$ by the μ_{match} function with variables '`?observation`', '`?station`', '`?value`' and '`?time`'. The output set from the function G_{rain} includes each of these variables and the results retrieved from $T \cup M$. A filter, O_σ, then constrains the input results, under the conditions that '`?value`' is more than 30 and that '`?time`' is within a specified bounds, to an output set entering the $O_\cup$ operator. A union is performed at this level by the $O_\cup$ between this input and another from the wind speed observation. Subsequently, another union is performed with the output from the branch of the tree with the snow observation. Finally, a projection operation function, O_Π, outputs a result set, R, containing the values of variable '`?station`' from the output of the union operator. By traversing the tree from leaves to root, a sequence of operations is obtained that can be expressed as equations 4.1 and 4.2 where x and y are the start and end times specified in the filter operations.

$$\cup_1 = \cup(\sigma_{windSpeed>100 \wedge x<time<y}(G_{windSpeed}), \sigma_{rain>30 \wedge x<time<y}(G_{rain})) \tag{4.1}$$

$$R = \Pi_{station}(\cup(\cup_1, \sigma_{snow=true \wedge x<time<y}(G_{snow}))) \tag{4.2}$$

These equations can also be seen as a high-level query execution plan, s_q. By executing each operation in s_q, a final result set, R, can be obtained from this final step of Map-Match-Operate, the $\mu_{operate}$ function.

The next section describes an implementation of Map-Match-Operate that translates SPARQL graph queries with S2SML mapping closures to Structured Query Language (SQL) queries for execution on a database tier with a relational database storing IoT time-series data.

4.2.4 S2S Engine: A Map-Match-Operate Implementation

S2S is an engine that allows graph queries expressed in SPARQL to be executed on a model and database tier consisting of an RDF graph and relational database respectively. The implementation in S2S follows the Map, Match and Operate steps for RDF graphs from Sections 4.2.1, 4.2.2 and 4.2.3 respectively.

The map step supports S2SML, defined in Section 4.2.1.1, as a data model mapping language for building mapping closures with the RDF graph model. Details on how a mapping closure is built in S2S is covered in Section 4.2.4.1. The match step employs a flexible swappable interface for subgraph matching covered in Section 4.2.4.2. Finally, the operate step translates the SPARQL query to a Structured Query Language (SQL) query and executes it on the relational database of which more details can be found in Section 4.2.4.3. S2S source code and documentation is published in an open respository[8].

4.2.4.1 Building a Mapping Closure for Map

In Map-Match-Operate, various data model mappings, m_{map}, can be built up in a bottom-up fashion to form a mapping closure, M^C_{map}. Following from Definition 4.7 of a mapping closure, the S2S engine performs a union of data model mappings within a closure, $\bigcup_{m \in M^C_{map}} m$. In S2S, the mapping closure is implemented as an in-memory RDF graph and as data model mappings are added, template matching as formalised previously in Definition 4.8 is performed. Each I_{map} element within the graph of each mapping m is reduced to I_p, the union of IRI string parts. The union of the set of binding strings from each I_{map}, I_C, as formalised in Definition 4.4, is extracted and stored in a key-value map ,m_{join}, with I_p as the key and the set of I_C bindings to columns from matching templates, within an array as the value.

For example, from the data model mappings in Figure 4.2, the reduced union of string parts, I_p '`<sen:system_>`', will replace both the I_{map} elements from different mappings, '`<sen:system_{sensors.name}>`' and '`<sen:system_{readings.sensor}>`', when the rainfall observation mapping and sensors mapping are added to the in-memory mapping closure graph. Hence, the graph formed from the two mappings is linked at the '`<sen:system_>`' vertex as it should be. m_{join} will store the extracted key-value pair of `(<sen:system_>, [sensors.name, readings.sensor])`. Hence, this implicit join condition between the columns '`sensors.name`' and '`readings.sensor`' is known to S2S whenever the '`<sen:system_>`' vertex is present in a match result set, $\mathbb{B}_{match}$, from the match step.

4.2.4.2 Swappable Interface for Match: SWIBRE

Basic graph patterns (BGPs) are sets of triple patterns within a SPARQL query. The match step of Map-Match-Operate as elaborated previously in Definition 4.9 performs a subgraph matching of the BGPs in a query with any relevant subgraphs from the mapping closure, M^C_{map}. S2S specifies the Swappable Interface for BGP Resolution Engines (SWIBRE) to perform the matching function, μ_{match}. SWIBRE is an interface that takes an input of a BGP, q_{graph} and performs a subgraph matching with the in-memory mapping closure as a RDF graph, M^C_{map}, which is built according to the previous section, Section 4.2.4.1.

[8]https://github.com/eugenesiow/sparql2sql

S2S includes two implementations of the SWIBRE interface specification[9] using open source projects Apache Jena[10] and Eclipse RDF4J[11] respectively. Each implementation is able to load a mapping closure and execute subgraph matching to return an iterator on a result set of matches, $\mathbb{B}_{match}$. A simple means of implementing a SWIBRE interface for any RDF store that supports SPARQL queries is to simply convert each BGP q_{graph} to a SPARQL query, '`select * where { BGP }`', and execute this on the RDF graph of the mapping closure. The result set obtained corresponds to $\mathbb{B}_{match}$.

4.2.4.3 SPARQL-to-SQL Syntax Translation for Operate

S2S parses a SPARQL query to form a query tree of operators like the example in Figure 4.4. By walking the query tree from leaf nodes to the root following the procedure previously defined in the operate function, $\mu_{operate}$ of Definition 4.10, a sequence of operation functions is obtained. This sequence can be translated to an SQL query.

A SPARQL query to retrieve rainfall observations from a station in Listing 4.5, produces a query tree like in Figure 4.5 and a SQL translation as shown in Listing 4.7. In the G_{rain} function, variable '`?value`' has a binding, b_{match}, to the database tier column referred to as '`station.Rainfall`' within the match results set, $\mathbb{B}_{match}$. The table '`station`' is added to the '`FROM`' SQL clause. The filter operation function, σ, adds a restriction to the SQL '`WHERE`' clause '`Rainfall > 10`' for a column using the results in $\mathbb{B}_{match}$. Finally, the project operation function, Π, adds the corresponding '`rainfall`' column for the variable '`?value`' from $\mathbb{B}_{match}$ to the SQL '`SELECT`' clause.

$$\begin{tikzcd}
\textit{Project}, \Pi_{\text{value}} \arrow[d, dash] \\
\textit{Filter}, \sigma_{>10} \arrow[d, dash] \\
G_{rain}
\end{tikzcd}$$

FIGURE 4.5: Query Tree of Operators for Listing 4.5's Rainfall Query

```
SELECT rainfall FROM station WHERE rainfall > 10
```

LISTING 4.7: SQL Query Translation of Listing 4.5

If faux nodes, F, are encountered in the projection operation function, Π, an SQL update statement of the format '`UPDATE table SET col=RANDOM_UUID()`' is run to generate a column of identifiers within the database tier table. The relevant I_{map} element within the mapping closure is updated from '`{table._uuid}`' to '`{table.col}`' so that subsequent queries that refer to the faux node automatically retrieve the identifier column. Therefore, if '`?obs`' was projected in the SPARQL query from Listing 4.5 of

[9]https://github.com/eugenesiow/sparql2sql/wiki/SWIBRE
[10]https://jena.apache.org/
[11]http://rdf4j.org/

rainfall observations, the faux node of `':obsstation._uuid'` would create a identifier column in the station table and the result set would contain the generated identifiers.

```
SELECT station FROM (
  SELECT 'http://knoesis.wright.edu/ssw/system_AROC2' AS station
  FROM station WHERE station.Rainfall>30
    AND station.time>='2003-04-03T16:00:00'
    AND station.time<'2003-04-03T17:00:00'
  UNION
  SELECT 'http://knoesis.wright.edu/ssw/system_AROC2' AS station
  FROM station WHERE station.WindSpeed>100
    AND station.time>='2003-04-03T16:00:00'
    AND station.time<'2003-04-03T17:00:00'
  UNION
  SELECT 'http://knoesis.wright.edu/ssw/system_AROC2' AS station
  FROM station WHERE station.Snowfall=true
    AND station.time>='2003-04-03T16:00:00'
    AND station.time<'2003-04-03T17:00:00'
)
```

Listing 4.8: SQL Query Translation of Listing 4.6

The SPARQL query from Listing 4.6 identifying weather stations experiencing bad weather, when executed on a particular station and mapping closure generated from the LSD dataset, station AROC2 in Salt Lake City, produces the query operator tree in Figure 4.4, that translates to the SQL query in Listing 4.8. There is a union of the three sub-queries for rainfall, snow and wind speed observations and a projection of the `'station'` variable from that union.

Table 4.4 shows a list of common operation functions, O, and their relation to corresponding SQL clauses with descriptions. The SQL syntax used is based on the Backus Normal Form (BNF) notation description of the SQL-2003 standard by Savage (2017a) and specifically realised as SQL grammar for the open source H2 database[12]. Each O in the table also includes a reference to the particular listing in the appendix describing the translation in BNF notation. A special case occurs if there are multiple orphaned tables in the `'FROM'` SQL clause in a translated query, without any join conditions from the operation functions or from the m_{join} key-value map introduced in Section 4.2.4.1, whereby a cartesian product (cross join) of orphaned tables has to be taken.

Examples of queries in SPARQL and the corresponding S2S SQL translation from SR-Bench (Zhang et al., 2012) and the Smart Home Analytics Benchmark in Appendix B are published as documentation online[13,14]. The full set of operator translations from SPARQL algebra is also presented in Appendix C for completeness.

[12]http://www.h2database.com/html/grammar.html
[13]https://github.com/eugenesiow/sparql2sql/wiki/SRBench
[14]https://github.com/eugenesiow/ldanalytics-pismarthome/wiki

O	SQL Clause	Description	BNF[a]
Project Π	Select, From	Subset of relation using $\mathbb{B}_{match}$	C.9
Extend ρ	Select	Alias of variable in $\mathbb{B}_{match}$	C.5
Filter σ	Where, Having, From	Restriction translated using $\mathbb{B}_{match}$	C.3
Union $\cup$	Union	Union between query terms	C.4
Group γ	Group By, From	Aggregation translated using $\mathbb{B}_{match}$	C.7
Slice σ_S	Limit	Add a ‘Limit’ clause	C.11
Distinct σ_D	Select	Add a ‘Distinct’ to ‘Select’ clause	C.10
Left Join $\ltimes$	Left Join..On, Select	If I add to $\mathbb{B}_{match}$, else ‘Left Join’	C.2

[a]Reference to the appendix listing with the translation in BNF using the SQL-2003 standard.

Table 4.4: Operation Functions O and Corresponding SQL Clauses

4.3 Performance and Storage Experimentation

The motivation behind Map-Match-Operate and the S2S engine implementation is to design a solution to address the mismatch between the graph structure for the integration of rich metadata and a compact structure suitable for IoT time-series data, which exacerbated by inefficiencies due to metadata expansion in the RDF model, is thought to detrimentally impact storage and query performance.

As such, this section seeks to introduce the experiments which were used to evaluate the storage and query performance of S2S against a range of 1) graph model RDF stores and 2) state-of-the-art SPARQL-to-relational engines. Two unique IoT scenarios, from different domains producing IoT time-series data, with published datasets are considered in Sections 4.3.1 and 4.3.2. The experimental setup, the stores benchmarked and the metrics compared in these scenarios are described in Section 4.3.3.

4.3.1 Distributed Meteorological System

The first scenario was based on the Linked Sensor Data (LSD) dataset from the Patni et al. (2010) paper, which recorded sensor metadata and observation data from about 20,000 weather stations across the United States. Two known periods of different types of inclement weather were highlighted for performance benchmarks with queries from the SRBench analytics benchmark paper by Zhang et al. (2012). The first period was of the Nevada Blizzard which took place from the 1st to the 6th of April 2003 and the dataset contained 108,830,518 RDF triples from 10,237,791 observations which were used for storage and query experiments. Another period from the 1st to the 13th of September 2008 recorded weather during Hurricane Ike. It was the largest LSD dataset with 534,469,024 triples and 49,487,873 observations and was utilised for storage experiments.

Queries 1 to 10 from SRBench[15] were utilised in this scenario as they were relevant to the scope of the experiment on IoT data from the environmental and disaster management domain. Queries 11 to 17 involved testing federation query performance across multiple SPARQL endpoints of Linked Data datasets on the web like DBpedia or Geonames which was not the aim of the experiment. Although the SRBench queries were written in various streaming SPARQL query languages, modifying them to become SPARQL queries was a trivial process of adding a time constraint. The full set of queries is published online[16].

Each weather station in this scenario stored metadata of its location and its sensors and also recorded observation data from its sensors on a local IoT device. The network of IoT device at stations geographically distributed across the country formed a distributed meteorological system. This scenario models a simple geo-distributed fog computing scenario described by Bonomi et al. (2014) where a large number of sensors that are 'naturally distributed' and have to be 'managed as a coherent whole' maintain a data plane (Yin et al., 2014) where historical data and operational metadata are stored and processed on fog device nodes, while deep analytics over a large territory occurs across the cloud-to-thing continuum.

Hence, at the system level, an application in the cloud executed SRBench queries to detect inclement weather nationally. Each analytical query was broadcasted to all stations in the network. The execution time for each query in the distributed system was measured as the time taken for the slowest station to return results.

4.3.2 Smart Home Analytics Dashboard

The second scenario involved an application dashboard that performed analytics on and visualised smart home data. A variety of data observations from environmental sensors, motion sensors in each room and energy meters from smart home sensors were collected by Barker et al. (2012) over 4 months from April to July in 2012. These observations included 26,192 environment sensor readings with 13 attributes, 748,881 energy meter readings from 53 meters and 232,073 motion sensor readings from six rooms in a smart home. A dashboard and benchmark were developed based on this dataset.

Appendix B describes the dataset, integration as a graph of metadata and data, application dashboard and individual queries that power it in more detail. The set of four queries used in the dashboard also formed an analytics performance benchmark for this scenario. The first three queries were descriptive analytics queries that allowed the examination of data like the temperature and energy usage over time through visualisation on the dashboard. The fourth query allowed anomalies in energy usage to

[15]http://www.w3.org/wiki/SRBench
[16]https://github.com/eugenesiow/sparql2sql/wiki

be detected and diagnosed by identifying unused rooms with appliances running and consuming energy.

The metadata and observation data for this scenario were stored on a local smart home hub, an IoT device that stored data collected from various smart home sensors and was able to deploy various applications like the the smart home analytics dashboard that monitored the conditions in the smart home or a Heating, Ventilation, and Air Conditioning (HVAC) control application that controlled the temperature in the smart home through various actuators like heaters connected to the hub.

4.3.3 Experimental Setup, Stores and Metrics

In both of the IoT scenarios described in the previous two sections, IoT devices were used to store and process metadata and data. Both scenarios utilised datasets that used a rich RDF graph model to represent sensor metadata for interoperability and allowed the Map-Match-Operate S2S engine to be compared against other graph RDF stores and SPARQL-to-relational translation engines. Both scenarios also included a set of SPARQL graph queries relevant for the IoT data involved within specified application use cases of a national inclement weather detection system and a smart home dashboard for monitoring environment conditions and energy usage.

The experimental setup for both scenarios used the compact, affordable and widely-available Raspberry Pi 2 Model B+ (RPi) as IoT devices. For the distributed meteorological system, each IoT device acted as a fog device node deployed in a weather station. The IoT device in the smart home scenario acted as the smart home hub. Each RPi had the following specifications of 1GB of Random-Access Memory (RAM), a 900MHz quad-core ARM Cortex-A7 processor and a Class 10 SD Card for storage. The cloud-tier where the meteorological application executed queries from had a 3.2GHz quad-core processor, 8GB of memory and a hybrid drive. Ethernet connections were used in each network to minimise network latency as a factor within these experiments. The RPi IoT devices ran Raspbian 4.1 based on Debian 7.0 (Wheezy). If a Java Virtual Machine (JVM) was used, the recommended best practice of initially allocating half the system's total physical memory of 512MB was taken (Neat, 2004).

Two RDF stores were included in the benchmark scenarios, the open-source Apache Jena Tuple Database (TDB) and the commercial GraphDB (formerly OWLIM). The other popular commercial RDF store, Virtuoso, was not tested as Virtuoso 7 could not run on the 32-Bit RPi and previous versions did not support SPARQL 1.1 time functions like `'hours'` which were required for most time-series aggregation queries. Version 3.0.1 of TDB and 6.6.3 of GraphDB Standard were tested.

Two state-of-the-art SPARQL-to-relational engines, Ontop by Rodriguez-Muro and Rezk (2015) and Morph by Priyatna et al. (2014), were also tested. A common underlying relational database for these engines and for S2S, the H2 relational database[17], was used. Version 1.6.1 of Ontop, 3.5.16 or Morph and 1.4.187 of H2 were used. Additionally, a quantitative evaluation of the SQL queries produced and the translation time measured was carried out as part of the experiment.

The storage and query performance metrics of storage size of the database for each scenario and the query latency of each query were measured. The query latency was measured by calculating the average query execution time over three runs.

Due to resource constraints for the distributed meteorological scenario, the difference in broadcast and individual connection times for each station was assumed to be minimal, hence, distributed tests for the 4700 plus stations were run in series on a test harness consisting of three RPi devices. The individual times for each station were recorded and averaged over three runs and the maximum among stations for each query was noted.

4.4 Results and Discussion

This section presents the results of the benchmarks from the two IoT scenarios proposed, a distributed meteorological system and a smart home analytics dashboard. Section 4.4.1 analyses the results from the storage performance metric of database size while Section 4.4.2 discusses the query performance metric of query latency.

4.4.1 Storage Performance Metric

Table 4.5 shows the database size of different datasets from the two IoT scenarios proposed on each database benchmarked. Nevada Blizzard and Hurricane Ike are datasets from two periods of observations in the distributed meteorological system while the Smart Home dataset is from the smart home analytics dashboard scenario. The databases compared include the H2 relational database that is used for query translation engines S2S, Ontop and Morph, the open-source TDB RDF store and the commercial GraphDB RDF store. The label S2S is used for the H2 database size in the table to more easily differentiate between the Map-Match-Operate and traditional RDF storage approaches. The number of stores for each dataset is also listed in the table and corresponds to the number of individual weather stations for the distributed meteorological system scenario.

As IoT sensor data benefited from the succinct storage format offered by relational databases for each time-series, S2S using H2 outperformed the RDF stores from about one to three orders of magnitude. The larger database size and worsened storage efficiency in both RDF stores, TDB and GraphDB, was due to metadata expansion and

[17] http://www.h2database.com/

Dataset	#Store(s)	Size (10^2 MB)			Ratio
		S2S	TDB	GraphDB	
Nevada Blizzard	4701	0.90	61.62	1216.94	1 : 68 : 1352
Hurricane Ike	12381	7.61	852.74	3450.04	1 : 112 : 453
Smart Home	1	1.35	21.03	12.21	1 : 15 : 9

TABLE 4.5: Database Size of Each Dataset across Databases

the need for more indices to improve query performance as described by Weiss et al. (2008). TDB creates 3 types of triple indices, *OSP*, *POS* and *SPO*. *S* stands for the subject IRI, *P* for the predicate IRI and *O* for the object IRI or literal within a triple. The order of the letters denotes how each index has been organised, for example, the *SPO* index contains all of the triples sorted first by subject, then by predicate, then by object. GraphDB created 6 indices *PSO*, *POS* and indices listing entities, classes, predicates and literals which explains the larger database size. TDB, however, suffered from greater overheads on each store and a larger number of stores resulted in worsened storage efficiency and a larger size ratio to S2S.

4.4.2 Query Performance Metric

Figure 4.6 shows the query latency of SRBench queries, measured by the average maximum execution time across all the distributed stations, on each of the databases or engines with the LSD Nevada Blizzard dataset. Both versions of Ontop and Morph engines, at the time of experimentation, did not support the aggregation operators required for queries 3 to 9 sans query 6. Additionally, Morph was unable to translate queries 6 and 10 while Ontop's translation of query 10 did not return a result on certain stations (e.g. station BLSC2).

The S2S engine performed better than other databases and engines consistently on all SRBench queries and also had a stable average execution time within the relatively small range of 0.32 to 0.54 seconds. The more compact storage of flat and wide IoT time-series data within relational tables allowed for faster retrieval than with RDF stores. As explained by Abadi et al. (2009), the RDF graph model and data representation as triples, though flexible, has 'the potential for serious performance issues' as a result of requiring self-joins to connect each set of triples.

Of the RDF stores, GraphDB generally performed better than TDB, especially on query 9. Query 9 shown in Listing 4.9, which gets the daily average wind force and direction observed by the sensor at a given location during a range of time, results in TDB doing a time consuming join operation between two subgraphs, `'WindSpeedObservation'` and

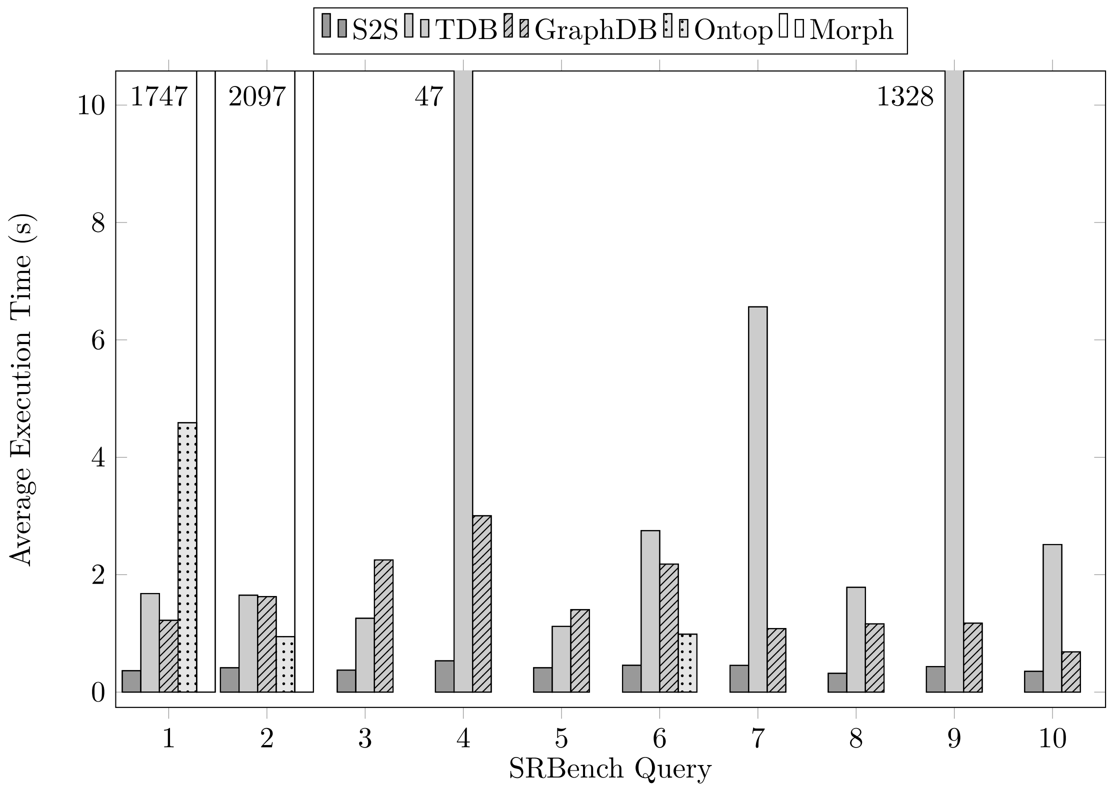


FIGURE 4.6: Average Maximum Execution Time for Distributed SRBench Queries

'`WindDirectionObservation`', within the memory-constrained environment of a distributed fog node. An investigation where the subgraphs in query 9 were retrieved individually in TDB by separate queries showed that each query cost a 100 times less in terms of execution time, confirming that large in-memory joins were an issue. Query 4 in SRBench was similar but with '`TemperatureObservation`' and '`WindSpeedObservation`' instead, and where the slowest station had considerably smaller subgraphs.

```
SELECT ( IF(AVG(?windSpeed) < 1,  0,
          IF(AVG(?windSpeed) < 4,  1,
            IF(AVG(?windSpeed) < 8,  2,
              // ... more wind force cases omitted for brevity
                IF(AVG(?windSpeed) < 73, 11, 12) ))) AS ?windForce )
    ( AVG(?windDirection) AS ?avgWindDirection )
WHERE {
  ?sensor ssw:generatedObservation ?o1 ;
    ssw:generatedObservation ?o2 .
  ?sensor ssw:processLocation ?sensorLocation .
  ?sensorLocation wgs84_pos:lat "{latitude}"^^xsd:float ;
    wgs84_pos:long "{longitude}"^^xsd:float .
  ?o1 a weather:WindDirectionObservation ;
    ssw:observedProperty weather:_WindDirection ;
    ssw:result [ssw:floatValue ?windDirection] ;
    ssw:samplingTime ?instant .
  ?o2 a weather:WindSpeedObservation ;
    ssw:observedProperty weather:_WindSpeed ;
    ssw:result [ssw:floatValue ?windSpeed] ;
    ssw:samplingTime ?instant .
  ?instant time:inXSDDateTime ?time .
  FILTER (?time>"2003-04-01T00:00:00"^^xsd:dateTime &&
    ?time<"2003-04-02T00:00:00"^^xsd:dateTime)
} GROUP BY ?sensor
```

Listing 4.9: SRBench Query 9: Getting the Daily Average Wind Force and Direction Observed by a Sensor from a Particular Location

Of the SPARQL-to-relational engines compared against, Ontop performed better than the RDF stores on queries 2 and 6, although it performed worse on query 1. Morph performed worse on both queries 1 and 2. The queries that were translated by S2S, Ontop and Morph were different although similar mappings (S2SML for S2S and translated to an R2RML equivalent for Ontop and Morph) were used, hence, resulting in differences in query latency. An additional comparison between SPARQL-to-relational engines, including S2S, in terms of the structure of queries generated and translation time was done. Table 4.6 shows the average translation time, t_{trans}, of the 3 engines. The S2S SWIBRE interface used for the match step in Map-Match-Operate, as described in Section 4.2.4.2, was the Jena implementation.

Q[a]	t_{trans} (ms)			Joins			Join Type & Structure	
	S2S	Ontop	Morph	S2S	Ontop	Morph	Ontop (qview)	Morph
1	**16**	702	146	**0**	6	4	implicit	4 inner
2	**17**	703	144	**0**	6	4	5 nested, 1 left outer	4 inner
6	**19**	703	-	**0**	5	-	5 implicit	-
10	**32**	846	-	**0**	6	-	UNION(2x3 implicit)	-

[a]The number of a particular SRBench query (Zhang et al., 2012).

Table 4.6: SPARQL-to-SQL Translation Time and Query Structure

In R2RML, used in Ontop and Morph, as shown by the example in Listing 4.4, the time column of the relational table that stored time-series data was used in IRI templates to generate identifiers for intermediate observation metadata nodes. The time column need not be unique and in the LSD dataset there were occurrences of repeated timestamps, hence, the column was not suited to be and was not listed as a primary key. Therefore, Ontop's semantic query optimisation process to chase equality generating dependencies as elaborated on by Rodriguez-Muro and Rezk (2015) could not be used to reduce joins. Hence, of the queries that could be run by Ontop and Morph as shown in Table 4.6, all had redundant inner joins on the time column as compared to S2S, that used Faux nodes when modelling IoT time-series data, which produced more efficient SQL queries.

S2S had the fastest query translation time, t_{trans}, averaged over 100 tries for each of the queries. Both Ontop and Morph though, supported additional inferencing or reasoning features motivated by semantic web use cases. Ontop, which performed an extra round trip to obtain database metadata from the relational database, had the longest translation times.

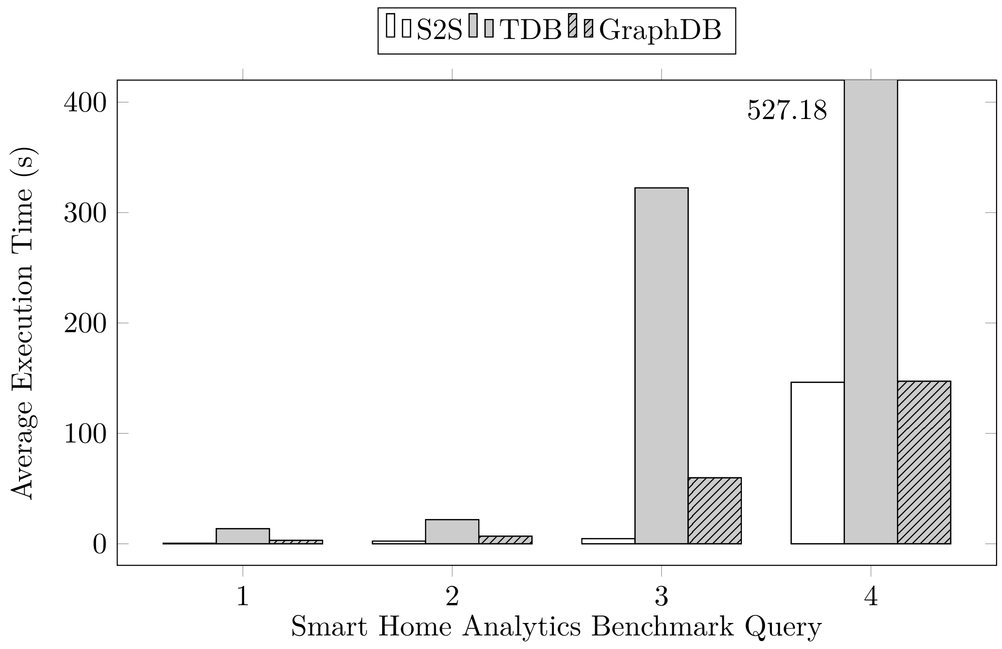


FIGURE 4.7: Average Execution Time for the Smart Home Analytical Queries

Although queries 1 and 2 were similar in purpose, in that both retrieved the rainfall observed from a station in an hour, query 2 had an additional ‘`OPTIONAL`’ SPARQL clause on the unit of measure term, hence, Ontop generated different structures for each query, explaining the discrepancy in time taken in Figure 4.6. The types of join syntax produced are listed in the table. For query 1, Ontop produced a query with 6 joins using the implicit join notation that listed 7 tables in the ‘`FROM`’ clause of the ‘`SELECT`’ statement and 6 filter-predicates in the ‘`WHERE`’ clause connecting the time columns. This was opposed to the inner join notation produced by Morph for the same query. The SQL translations of each query from all the engines are available in the SRBench section of the S2S documentation[18].

In the smart home scenario, S2S query performance on aggregation queries as shown in Figure 4.7 was once again ahead of the RDF stores, from 3 to 70 times faster. The only exception was for query 4 in which S2S and GraphDB had similar query latency. All the queries in the scenario required SPARQL 1.1 space-time aggregations which excluded the other SPARQL-to-relational engines from the benchmark. Of the RDF stores, GraphDB had all-round better performance than TDB for this set of queries.

S2S, TDB and GraphDB all performed faster on queries 1 and 2, which aggregated temperature time-series data over a day in query 1 and a month in query 2, than on queries 3 and 4. The set of about 30 thousand temperature sensor readings was significantly smaller than the set of about 750 thousand energy meter readings accessed in query 3, which aggregated the hourly room-based energy usage. As disk access for all the databases was expensive, especially on the slow SD card medium used on the

[18]https://github.com/eugenesiow/sparql2sql/wiki/SRBench

smart home hub to store data, queries that had less data to read performed faster. More detailed information on the queries and benchmark can be found in Appendix B.

Query 4 which diagnosed unattended energy usage in rooms required access to both the meter and motion sensor data, a comparatively larger set of data. Space and time aggregation, by room and by hour respectively, was performed on both sets of data. A join between these two sets was necessary on S2S which increased the average execution time. Overall, the S2S engine still provided significant query performance improvements through reducing self-joins between columns, for example the timestamp and the internal temperature columns on the same row within the relational table could be read more efficiently, and with reduced metadata expansion.

SR_{Bench}	S2S	TDB	GraphDB	Ontop	Morph	Ratio
1	**0.37**	1.68	1.22	4.59	1747.70	1: 5: 3: 13: $(4 \cdot 10^3)$
2	**0.42**	1.65	1.63	0.95	2097.16	1: 4: 4: 2: $(5 \cdot 10^3)$
3	**0.37**	1.26	2.25	-	-	1: 3: 6
4	**0.53**	47.08	3.00	-	-	1: 88: 6
5	**0.42**	1.12	1.40	-	-	1: 3: 3
6	**0.46**	2.75	2.18	0.99	-	1: 6: 5: 2
7	**0.46**	6.56	1.08	-	-	1: 14: 2
8	**0.32**	1.79	1.16	-	-	1: 6: 4
9	**0.44**	1328.20	1.18	-	-	1: $(3 \cdot 10^3)$: 3
10	**0.35**	2.51	0.69	-	-	1: 7: 2

Smarthome	S2S	TDB	GraphDB	Ratio
1	**0.47**	13.71	3.13	1: 29: 7
2	**2.46**	21.90	6.91	1: 9: 3
3	**4.69**	322.36	59.81	1: 69: 13
4	147.65	527.18	**147.28**	1: 4: 1

Table 4.7: Average Query Execution Times (in s) of IoT Scenarios

Table 4.7 summarises the average query execution times for all the experiments. The ratio of average execution time from each query on each database and engine is compared to S2S. Performance improvements range from two times to three orders of magnitude with the sole exception of query 4 from the Smart Home Analytics Benchmark, where query latency is tied with GraphDB.

4.5 Conclusions

This chapter has provided an IoT-specific solution to reconcile the tension between supporting a rich graph model for the integration and interoperability of metadata with data, and the characteristics of current time-series IoT data through 1) the introduction of a formal Map-Match-Operate method, 2) its realisation within a query translation engine for the RDF graph model and 3) experimentation that substantiated its storage and query performance improvements over other databases and SPARQL-to-relational engines. It was also noted that the state-of-the-art SPARQL-to-relational engines, like Ontop and Morph, that have reported significant performance improvements over RDF stores on various benchmarks and deployments, were not yet equipped to support or optimised for queries and models from the IoT domain. As such, the contribution of Map-Match-Operate, S2S, and S2SML as a data modelling language, fills a gap in the literature for IoT time-series data, IoT scenarios and analytical queries.

Chapter 5

TritanDB: A Rapid Time-series Analytics Database

"I see a picture right now that's not parallel, so I'm going to go straighten it. Things must be in order."

— *Katherine Johnson*

While the previous chapter established the efficiency of the Map-Match-Operate method for the execution of graph queries over a graph model tier and a relational database tier, this chapter seeks to investigate even more efficient and specialised representations of time-series data within the database tier to further enhance storage and query performance. Section 5.1 introduces the gap between the conceptual and physical representations of time-series data in the database tier and the potential opportunity for greater optimisation. This potential drives an investigation of state-of-the-art data structures and compression for time-series data in Section 5.2 that then advises the design of a time-series database for Internet of Things (IoT) data, TritanDB, described in Section 5.3 which replaces the relational database in the database tier. The implementation within a Map-Match-Operate engine is then compared against several state-of-the-art time-series databases and the results are presented and discussed in Section 5.4. Finally, Section 5.5 draws conclusions from the findings in the chapter.

5.1 Introduction

Conceptually, time-series Internet of Things (IoT) data that show flat and wide characteristics can be stored compactly as a two-dimensional relational table. One dimension of the table, rows, correspond to points in time while another dimension, columns, corresponds to various measurements made at those points in time. There is, however, a distinction between the conceptual and physical properties of these database tables. At a physical level, database tables on common storage media like magnetic disks, solid state drives or Random Access Memory (RAM) need to be mapped to one-dimensional structures to be stored and retrieved. Internally, these storage media provide just a one-dimensional interface, for example, reading and writing from a given linear offset.

Therefore, this chapter's work is motivated by the opportunity to combine the flat and wide characteristics of IoT data, with a deeper understanding of how time-series data can best be organised with respect to the underlying physical properties of the storage medium for storage and retrieval. Furthermore, the largely numerical type of data and the presence of both approximately periodic and non-periodic data also suggest that a study of state-of-the-art compression algorithms and data structures will be beneficial for designing a means for storing time-series data efficiently. Fortuitously, the big data era has driven advances in data management and processing technology with new databases emerging for specialised use cases. The large volumes of telemetry data from performance monitoring scenarios within web-scale systems have pushed the research and development of time-series databases to the forefront, providing an array of technologies and designs to study, build upon and compare against. A survey of these time-series databases was presented in Section 2.2.1.

A three-time Formula One champion, Sir Jackie Stewart, was reputed to have coined the term 'mechanical sympathy', also quoted by Robinson et al. (2015) in their unrelated discussion on idiomatic query design for graph databases. It reflects how a race car driver, by understanding the mechanics and engineering behind the vehicle, is able to work harmoniously together with it to achieve the best performance. This idea of 'mechanical sympathy' forms a driving factor behind this chapter's work that seeks to further optimise the performance of Match-Map-Operate from Chapter 4, which already represents significant gains in flexibility and storage and query performance. A series of microbenchmarks helps form a scientific understanding on compression techniques and data structures to achieve greater 'mechanical sympathy'.

5.2 Time-series Compression and Data Structures

In Section 3.1, it was established that a current sample of Internet of Things (IoT) data exhibited numerous characteristics. The time-series data studied tended to be flat, wide and numerical. Furthermore, there was a mixture of periodic, approximately periodic and non-periodic time-series data. Flat, wide and numerical IoT time-series data presents a unique use case for compression algorithms across the dimension of columns, on its timestamp column as well as each column representing an attribute, likely of numerical type data.

Compression is used in relational databases and is especially important in advanced column-stores as a performance optimisation rather than just for reducing storage size. When data is compressed, less time is spent on input-output operations with the physical storage medium interface when data is read from disk into memory (or from memory to CPU). Another motivation of compression is related to CPU speed increasing faster as compared to memory bandwidth, hence, the comparative cost of accessing data as opposed to compressing and decompressing it in terms of CPU cycles has become greater as explained by Abadi et al. (2012). Section 5.2.1 explains timestamp compression, compressing the sequence of timestamps in a time-series, while Section 5.2.2 explains compressing values within columns of each attribute over the range of timestamps.

To evaluate the performance and compression ratios produced by each of these algorithms for timestamp and value compression, Section 5.2.3 describes a set of IoT time-series datasets from different application domains for use in microbenchmarks and Section 5.2.4 presents and discusses the results of microbenchmarks of the compression algorithms on each dataset.

The large volumes of time-series data from the IoT that have been efficiently compressed need to be persisted to disk in a way that they can be efficiently retrieved during queries. This motivates the study of a range of data structures commonly associated with state-of-the-art time-series databases. Hence, Section 5.2.5 describes a number of implementations of data structures for storing time-partitioned, compressed blocks of time-series IoT data. A proposal for a novel data structure called TrTables is made in Section 5.2.5.4 and the performance of each implementation is then evaluated in terms of read, write and space amplification metrics by microbenchmarks in Section 5.2.6.

5.2.1 Timestamp Compression

Timestamps in a series can be compressed to great effect based on the knowledge that in practice, the delta of a timestamp, the difference between this timestamp and the previous, is a fraction of the length of the timestamp itself and can be combined with variable-length encoding to reduce storage size. If the series is somewhat evenly-spaced, Run Length Encoding (RLE) can be applied to further compress the timestamp deltas. As shown in Sections 3.1 and 3.2, a significant proportion of IoT time-series exhibited periodic or approximately periodic, evenly-spaced data streams.

For high precision timestamps (e.g. in nanoseconds), where deltas themselves are large however, delta-of-delta compression that stores the difference between deltas can often be more effective. Figure 5.1 depicts various methods of compressing a series of timestamps with millisecond precision. The following sections go into each of the three

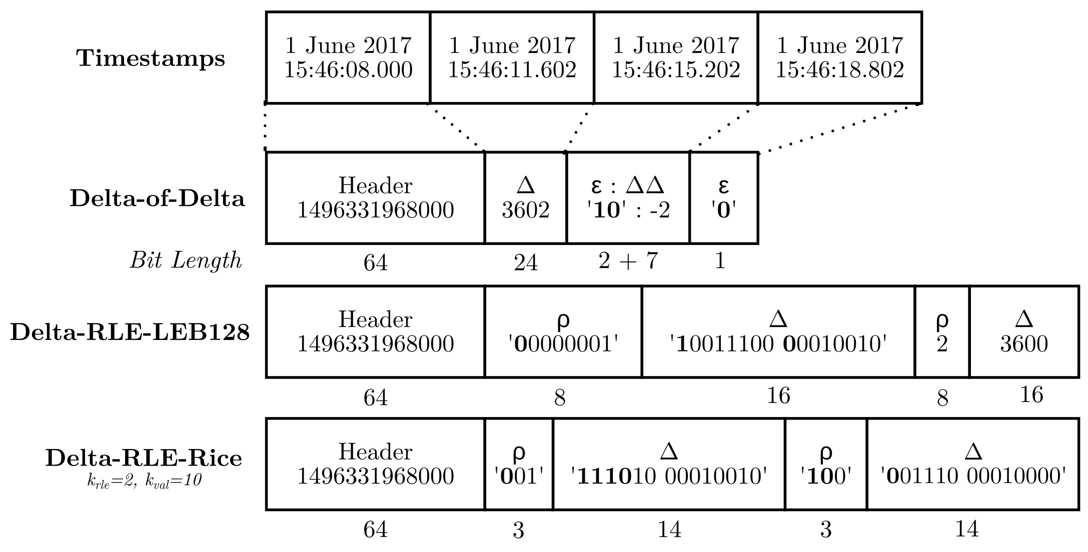


FIGURE 5.1: Visualisation of the Timestamp Compression at Millisecond Precision with Various Delta-based Methods

methods, Delta-of-Delta compression in Section 5.2.1.1, Delta-RLE-LEB128 compression in Section 5.2.1.2 and backwards adaptive Delta-RLE-Rice compression in Section 5.2.1.3.

5.2.1.1 Delta-of-Delta Compression

Delta-of-Delta compression builds on the technique for compressing timestamps introduced by Pelkonen et al. (2015) to support effective compression on varying timestamp precision. The header stores a full starting timestamp for the block in 64 bits and the next length of bits depends on the timespan of a block and the precision of the timestamps. In the example in Figure 5.1, a 24 bit length is used to store the timestamp delta of '3602'. The 24 bit length is derived for a 4 hour block at millisecond precision with the first delta assumed to be positive. For the same 4 hour block, if the precision of timestamps during ingestion is determined to be in seconds a 14 bits length is calculated and used, while this length is 24 bits for millisecond precision and 44 bits for nanoseconds precision.

ϵ is a 1 to 4 bit variable-length binary value that indicates the next number of bits to read for the first delta-of-delta value. '0' means the delta-of-delta ($\Delta\Delta$) is 0, while '10' means read the next 7 bits as the value is between '-63' and '64' (range of 2^7), '110' the next 24 bits, '1110' the next 32 bits. Finally, an ϵ of '1111' means reading a 64 bit $\Delta\Delta$. The example follows with $\Delta\Delta$s of '-2' and '0' stored in just 10 bits which reflect the delta of the next two timestamps is '3600'. These lengths are adapted from the empirical measurement of the distribution of $\Delta\Delta$ values from the Pelkonen et al. (2015) paper.

5.2.1.2 Delta-RLE-LEB128 Compression

The LEB128 encoding format is a variable-length encoding recommended in the DWARF debugging format specification by the DWARF Debugging Information Format Committee (2017) and commonly used in Android's Dalvik executable format. Numerical values like timestamps can be compressed efficiently along byte boundaries (minimum of 1 byte). In the example in Figure 5.1, the header stores a full starting timestamp for the block in 64 bits followed by a run-length value, ρ, of 1 and the actual delta, Δ, of 3602, both compressed with LEB128 to 8 and 16 bits respectively. The first bit in each 8 bits is a control bit that signifies to read another byte for the sequence if '1' or the last byte in the sequence if '0'. The remaining 7 bits are appended with any others in the sequence to form the numerical value. Binary '00001110 00010010' is formed from appending the last 7 bits from each byte of Δ which translates to the value of '3602' in base 10. This is followed by a run-length, ρ, of two Δs of '3600' each in the example.

5.2.1.3 Delta-RLE-Rice Compression

This proposal for Delta-RLE-Rice compression is inspired by the backward adaptation strategy from Malvar (2006) for the run-length encoding method initially proposed by Rice and Plaunt (1971). The backward adaptation strategy succeeds by tuning a k parameter which allows the adaptive compression of timestamps and run-lengths of varying precision and periodicity respectively. Rice coding divides a value, u, into two parts based on k, giving a quotient $q = \lfloor u/2^k \rfloor$ and the remainder, $r = u\%2^k$. The quotient, q is stored in unary coding, for example, the Δ value '3602' with a k of 10 has a quotient of 3 and is stored as '1110'. The remainder, r, is binary coded in k bits. Initial k values of 2 and 10 are used in this example in Figure 5.1 and are adaptively tuned based on the previous value in the sequence so this can be reproduced during decoding. Three rules govern the tuning based on the value of q, allowing quick convergence on good k values.

$$\text{if } q = \begin{cases} 0, & k \to k-1 \\ 1, & \text{no change in } k \\ >1, & k \to k+q \end{cases}$$

This adaptive coding adjusts k based on the actual data to be encoded so no other information needs to be retrieved on the side for decoding. It also has a fast learning rate that chooses good, though not necessarily optimal, k values and does not have the delay of forward adaptation methods. k is adapted from 2 and 10 to 1 and 13 respectively in the Figure 5.1 example.

5.2.2 Value Compression

This section explores two opportunities when considering value compression for time-series IoT data. The first being that most of the time-series values were determined to be of numerical data type, hence, this section focuses on floating point compression algorithms that also support long integers. Secondly, as noted by Abadi et al. (2012) in their work on column-stores, compressing data with a low information entropy and high data value locality has better performance and intuitively, values from the same column tend to have more value locality and less entropy. Therefore, each attribute value from a time-series can be compressed more effectively with the next value in the sequence and this section of value compression studies compressing along the column dimension.

Three state-of-the-art floating point compression algorithms are considered, the fast Floating Point Compression (FPC) algorithm by Burtscher and Ratanaworabhan (2009) is described in Section 5.2.2.1, the simpler block-aligned method used in Facebook's Gorilla by Pelkonen et al. (2015) is studied in Section 5.2.2.2 and the delta-of-delta method used in the time-series database BTrDb by Andersen and Culler (2016) is considered in Section 5.2.2.3.

5.2.2.1 FPC: Fast Floating Point Compression

The FPC algorithm introduced by Burtscher and Ratanaworabhan (2009) uses the more accurate of two value predictors to predict the next value in a double-precision numerical sequence of floating point numbers. These include the Finite Context Method (FCM) value predictor described by Sazeides and Smith (1997) and the Differential Finite Context Method (DFCM) value predictor introduced by Goeman et al. (2001). Accuracy is determined by the number of significant bits shared between the actual value and each predicted value. After an XOR operation between the chosen predicted value and actual value, the leading zeroes are collapsed into a 3 bit value and appended with a single bit indicating which predictor was used and the remaining non-zero bytes. As XOR is reversible and the predictors are effectively hash tables, fast and lossless decompression can thus be performed.

5.2.2.2 Gorilla Value Compression

Gorilla, unlike FPC does not use predictors and instead compares the current value to only the previous value for speed, as detailed in the paper by Pelkonen et al. (2015). After an XOR operation between the values, the result, r, is stored according to the output from a function *gor()* described as follows, where '.' is an operator that appends bits together, p is the previous XOR value, *lead()* and *trail()* return the number of leading and trailing zeroes respectively, *len()* returns the length in bits and n is the number of meaningful bits remaining within the value.

$$gor(r) = \begin{cases} '0', & \text{if } r = 0 \\ '10'.n, & \text{if } lead(r) >= lead(p) \text{ and } trail(r) = trail(p) \\ '11'.l.m.n, & \text{else, where } l = lead(r) \text{ and } m = len(n) \end{cases}$$

The algorithm was designed for the fast encoding and decoding of blocks of time-series data as there is no condition within the $gor(r)$ function to reduce the number of significant bits stored in the sequence but only to increase them. However, each block resets the counter on significant bits and 'localises' the compression. This phenomenon is explained in greater detail in Section 5.2.6.1 on space amplification.

5.2.2.3 Delta-of-Delta Value Compression

Andersen and Culler (2016) suggested the use of a delta-of-delta method for compressing the mantissa and exponent components of floating point numbers within a series separately. The method was not described in detail in the paper but this thesis interprets it as such: an IEEE-754 double precision floating point number defined by the IEEE Standards Association (2008) can be split into sign, exponent and mantissa components. The 1 bit sign is written, followed by at most 11 bits delta-of-delta of the exponent, δ_{exp}, encoded by a function $E_{exp}()$, described as follows, and at most 53 bits delta-of-delta of the mantissa, δ_m, encoded by $E_{mantissa}()$, also described as follows.

$$E_{exp}(\delta_{exp}) = \begin{cases} '0', & \text{if } \delta_{exp} = 0 \\ '1'.e, & \text{else, where } e = \delta_{exp} + (2^{11} - 1) \end{cases}$$

$$E_{mantissa}(\delta_m) = \begin{cases} '0', & \text{if } \delta_m = 0 \\ '10'.m, & \text{if } -2^6 + 1 <= \delta_m <= 2^6,\ m = \delta_m + (2^6 - 1) \\ '110'.m, & \text{if } -2^{31} + 1 <= \delta_m <= 2^{31},\ m = \delta_m + (2^{31} - 1) \\ '1110'.m, & \text{if } -2^{47} + 1 <= \delta_m <= 2^{47},\ m = \delta_m + (2^{47} - 1) \\ '1111'.m, & \text{else, where } m = \delta_m + (2^{53} - 1) \end{cases}$$

The operator '.' appends binary coded values in the above functions. e and m are expressed in binary coding (of base 2). A maximum of 12 and 53 bits are needed for the exponent and mantissa deltas respectively as they could be negative.

Dataset	Metadata			Timestamps			Field Types		
	Domain	Rows	Fields	P	δ_{MAD}	δ_{IQR}	Bool	FP	Int
SRBench	Weather	647k	4.3m	s	0[a]	0	$\tilde{0}$[b]	$\tilde{6}$	0
Shelburne	Agriculture	12.4m	74.7m	ms/ns	0/0	0.29/293k	0	6	0
GreenTaxi	Taxi	4.4m	88.9m	s	1.48	2	1	12	7

[a]As there were 4702 stations, a median of the MAD and IQR of all stations was taken, the means are 538 and 2.23×10^6

[b]A mean across the 4702 stations was taken for each field type in SRBench

Legend: FP = Floating Point Number, Bool = Boolean, Int = Integer, P = Precision, s,ms,ns = seconds,milliseconds,nanoseconds, $k = \times 10^3$ and $m = \times 10^6$.

Table 5.1: Public IoT Datasets used for Microbenchmarks

5.2.3 IoT Datasets for Microbenchmarks

To evaluate the performance of various algorithms and system designs with microbenchmarks, a set of publicly available IoT time-series datasets were collated. The use of public, published data, as opposed to proprietary data, enables reproducible evaluations and a base for new systems and techniques to make fair comparisons.

Table 5.1 summarises the set of datasets collated, describing the precision of timestamps, Median Absolute Deviation (MAD) of deltas, δ_{MAD}, Inter-quartile Range (IQR) of deltas, δ_{IQR}, and the types of fields for each dataset. Each dataset, SRBench, Shelburne and GreenTaxi (when necessary, shortened to Taxi for brevity) are described in more detail in Sections 5.2.3.1, 5.2.3.2 and 5.2.3.3 respectively.

The MAD was defined previously in Section 3.1.4 while the IQR is defined as the difference between the upper quartile, the 75th percentile, and the lower quartile, the 25th percentile and is a measure of the statistical dispersion of data points.

5.2.3.1 SRBench: Weather Sensor Data

SRBench by Zhang et al. (2012) is a benchmark based on the Linked Sensor Data (LSD) dataset by Patni et al. (2010) that described sensor data from weather stations across the United States with recorded observations from periods of bad weather. In particular, the Nevada Blizzard period of data from 1st to 6th April 2003 which included more than 647 thousand rows with over 4.3 million fields of data from 4702 of the stations was used. Stations have timestamp precision in seconds with the median δ_{MAD} and δ_{IQR} across stations both zero, showing regular, periodic intervals of measurement. The main data type is small floating point numbers, mostly up to a decimal place in accuracy.

5.2.3.2 Shelburne: Agricultural Sensor Data

Shelburne is an agriculture dataset aggregating data from a network of wireless sensors obtained from a vineyard planting site in Charlotte, Vermont, United States. Each reading includes a timestamp and fields like solar radiation, soil moisture and leaf wetness among others. The dataset is available on SensorCloud[1] and was collected from April 2010 to July 2014 with 12.4 million rows and 74.7 million fields. Timestamps were recorded up to nanosecond precision. The δ_{MAD} is zero as the aggregator records at regular intervals (median of 10s), however, due to the high precision timestamps and outliers, there is a δ_{IQR} of 2.93×10^5 (in the microsecond range). All the fields recorded are floating point numbers recorded with a high decimal count and accuracy.

5.2.3.3 GreenTaxi: Taxi Trip Data

This dataset includes trip records from green taxis in New York City from January to December 2016. The data was provided by the Taxi and Limousine Commission[2] and consists of 4.4 million rows with 88.9 million fields of data. Timestamp precision is in seconds and is unevenly-spaced as expected from a series of taxi pick-up times within a big city with a δ_{MAD} of 1.48. However, as the time-series also has overlapping values and is very dense, the δ_{MAD} and δ_{IQR} are all within 2 seconds. There is one boolean field type for the store and forward flag which indicates whether the trip record was held in vehicle memory before sending to the vendor because the vehicle did not have a connection to the server at that time. There are 12 floating point field types including latitude and longitude values with high decimal counts and fares with a low decimal count. There are also integer field types including vendor ID, rate code ID and drop off timestamps.

5.2.4 Compression Microbenchmark Results

A series of microbenchmarks were performed to evaluate the performance of the compression algorithms for timestamps and values on each IoT time-series dataset. For timestamp compression, the storage size of the dataset after compression with the specific algorithm is presented in Section 5.2.4.1. For value compression, both the storage size and the average compression and decompression times were recorded as in Section 5.2.4.2. The experimental setup for the microbenchmarks had a 4×3.2GHz CPU, 8 GB memory and average disk data rate of 146.2 MB/s. The source code for running the microbenchmarks is available on Github[3].

[1]https://sensorcloud.microstrain.com/SensorCloud/data/FFFF0015C9281040/
[2]http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
[3]https://github.com/eugenesiow/tritandb-kt

Dataset	Timestamps (MB)[a]				Values (MB)[b]			
	δ_Δ	δ_{leb}	δ_{rice}	$\delta_\varnothing$	C_{gor}	C_{fpc}	C_Δ	$C_\varnothing$
SRBench	0.6	0.5	**0.4**	5.2	**8.2**	23.8	21.9	33.9
Shelburne (ms)	**8.0**	18.3	13.6	99.5	440.8	**419.3**	426.6	597.4
Shelburne (ns)	**35.9**	56.2	44.1	99.5				
GreenTaxi	4.0	6.9	**1.5**	35.5	342.1	**317.1**	318.8	710.9

[a]δ_Δ = Delta-of-Delta, δ_{leb} = Delta-RLE-LEB128, δ_{rice} = Delta-RLE-Rice, $\delta_\varnothing$ = Delta-Uncompressed
[b]C_{gor} = Gorilla, C_{fpc} = FPC, C_Δ = Delta-of-Delta , $C_\varnothing$ = Uncompressed

Table 5.2: Compressed Size of Timestamps and Values in Datasets with Varying Methods

5.2.4.1 Timestamp Compression Microbenchmark Results

Table 5.2 shows the results of running each of the timestamp compression methods including a control uncompressed set for each dataset. Delta-RLE-Rice, δ_{rice}, was observed to perform best for low precision timestamps (to the second) while delta-of-delta compression, δ_Δ, performed well on high precision, milli and nanosecond timestamps. The adaptive δ_{rice} performed exceptionally well on the GreenTaxi dataset which had precision to seconds and small intervals between records and therefore small deltas. δ_Δ performed well on Shelburne due to an approximately periodic time-series but with large deltas due to high precision.

5.2.4.2 Value Compression Microbenchmark Results

Table 5.2 shows the results comparing Gorilla, FPC and delta-of-delta value compression against each of the datasets. Each compression method has advantages, however, in terms of compression and decompression times, Gorilla compression consistently performed best, as shown in Table 5.3, where each dataset was compressed to a file 100 times and the time taken was averaged. Each dataset was then decompressed and the time taken was averaged over a 100 tries. A read and write buffer of 2^{12} bytes was used for all experiments. FPC had the best compression ratio on values with high decimal count in Shelburne and was slightly better on a range of field types in GreenTaxi than delta-of-delta compression, however, even though the hash table prediction had similar speed to the delta-of-delta technique, it was still up to 25% slower on encoding than Gorilla. Gorilla though, expectedly trailed FPC and delta-of-delta in terms of size for the Shelburne and Taxi datasets which had more rows of data. This was to be expected as Gorilla is a block-based compression algorithm optimised for smaller time-partitioned blocks of data which is explained in more detail in Section 5.2.6.1 on Space Amplification.

As can be observed from Table 5.2, even the worse compression method for timestamps, δ_{max}, occupies but a fraction of the total space using the best value compression method, C_{min}, as shown by the percentage of total size calculation, $\delta_{max} \div (\delta_{max} + C_{min}) \times 100\%$,

Dataset	Compression (s)[a]				Decompression (s)			
	C_{gor}	C_{fpc}	C_{Δ}	Top	C_{gor}	C_{fpc}	C_{Δ}	Top
SRBench	**2.10**	3.25	3.10	C_{gor}	**0.97**	1.61	1.36	C_{gor}
Shelburne	**30.68**	42.02	40.91	C_{gor}	**3.80**	4.57	5.42	C_{gor}
GreenTaxi	**28.85**	32.11	32.94	C_{gor}	**2.94**	4.32	5.77	C_{gor}

[a]C_{gor} = Gorilla, C_{fpc} = FPC, C_{Δ} = Delta-of-Delta

Table 5.3: Average Compression/Decompression Time of Datasets (100 Attempts)

which results in percentages of 6.8%, 11.8% and 2.1% for SRBench, Shelburne and Green Taxi respectively. Hence, an effective and fast compression method supporting hard-to-compress numerical values of both floating point numbers and long integers can greatly improve compression ratios and also read and write performance.

5.2.5 Data Structures and Indexing for Time-Partitioned Blocks

In time-series databases, as seen in microbenchmark results in the previous section, Section 5.2.4, data can be effectively compressed in time order. A common way of persisting this to disk is to partition each time-series by time to form time-partitioned blocks that can be aligned on page-sized boundaries or within memory-mapped files. This section studies generalised implementations of data structures used in state-of-the-art time-series databases to store and retrieve time-partitioned blocks: concurrent B+-trees, Log-Structured Merge (LSM) trees and segmented Hash-trees and each is explained in Sections 5.2.5.1, 5.2.5.2 and 5.2.5.3 respectively. Finally, a novel data structure, the TritanDB Table (TrTable) inspired by the Sorted String Table (SSTable) is proposed for IoT time-series block storage in Section 5.2.5.4.

5.2.5.1 B+-Tree-based

A concurrent B+-Tree can be used to store time-partitioned blocks with the keys being block timestamps and the values being the compressed binary blocks. Time-series databases, like Akumuli from Lazin (2017) which implements an LSM-tree with B+-trees instead of SSTables, and BTrDb from Andersen and Culler (2016) which implements an append-only, Copy-On-Write (COW) B+-tree, use variations of this data structure. This thesis implements the general Sagiv (1986) B^{Link} balanced search tree utilising the verified algorithm in the work by da Rocha Pinto et al. (2011) for the microbenchmarks. The leaves of the tree are nodes that contain a fixed size list of key and value-pointer pairs stored in order. The value-pointer points to the actual block location so as to minimise the size of nodes that have to be read during traversal. The final pointer in each node's list, called a link pointer, points to the next node at that level which allows for fast sequential traversal between nodes. A prime block includes pointers to the

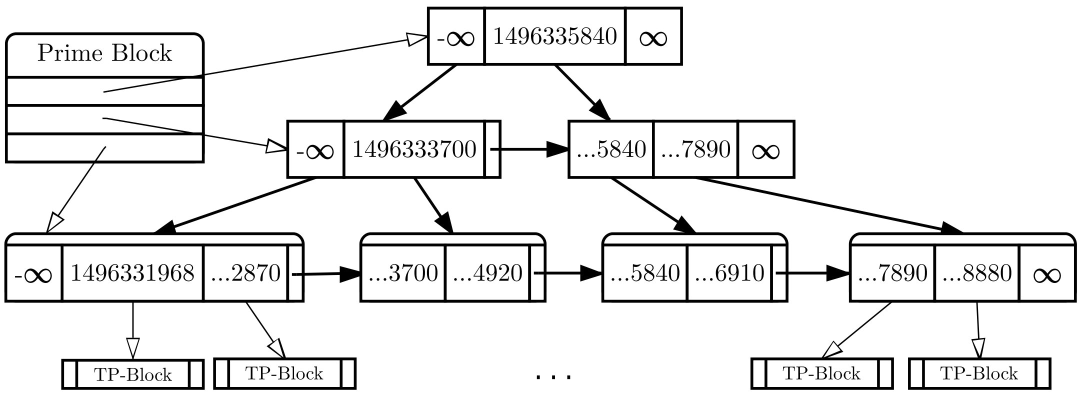


FIGURE 5.2: A B^{Link} Tree with Timestamps as Keys and Time-Partitioned (TP) Blocks Stored Off Node

first node in each level. Figure 5.2 shows this B^{Link} tree with timestamps as keys and time-partitioned (TP) blocks stored off node with a pointer to an offset.

5.2.5.2 Hash-Tree-based

Given that hashing is commonly used in building distributed storage systems and various time-series databases, like Riak-TS by Basho (2017) which uses a hash ring, a generalised concurrent hash map data structure is investigated within these microbenchmarks. A central difficulty of hash table implementations is defining an initial size of the root table especially for streaming time-series of indefinite sizes. Instead of using a fixed-sized hash table that suffers from fragmentation and requires the rehashing of data when it grows, an auto-expanding Hash-tree of hash indexes is used in the implementation for microbenchmarks instead. Leaves of the tree contain expanding nodes with keys and value pointers. Concurrency is supported by implementing a variable segmented Read-Write-Lock approach, the variable of which corresponds to the concurrency factor, that is similar to the one implemented in JDK7's *ConcurrentHashMap* data structure as described by Burnison (2013), and 32-bit hashes are used for block timestamp keys.

5.2.5.3 LSM-Tree-based

The Log-Structured Merge (LSM) tree described by O'Neil et al. (1996) is a write optimised data structure used in time-series databases like InfluxDb by Influx Data (2017), which uses a variation called a Time-Structured Merge (TSM) tree, OpenTSDB on HBase by StumbleUpon (2017), and various Cassandra-based databases that use an LSM-tree implementation described by Lakshman and Malik (2010). High write throughput is achieved by performing sequential writes instead of dispersed, update-in-place operations that most tree-based structures require. This particular implementation of the LSM-tree is based on the bLSM design by Sears and Ramakrishnan (2012) and has an in-memory buffer, a *memtable*, that holds block timestamp keys and time-partitioned blocks as values within a red-black tree to preserve key ordering. When the *memtable* is full, the sorted data is flushed to a new file on disk requiring only a sequential write. Any new blocks or edits simply create successive files which are traversed in order during reads. The system periodically performs a compaction to merge files together, removing duplicates.

5.2.5.4 TrTables: A Novel Time-Partitioned Block Data Structure

TritanDB Tables (TrTables) are a novel IoT time-series optimised storage data structure proposed in this thesis which takes inspiration from the *Sorted String Table (SSTable)* that consists of a persistent, ordered, immutable map from keys to values and the *memtable*, a sorted in-memory buffer used to order random writes of recent commits. Both are used in tandem in big data systems based on BigTable, initially described by Chang et al. (2008).

The main difference between TrTables and the SSTable and memtable combination proposed in BigTable is that no additional background compaction process, which can prove expensive, is required. This is due to the natural temporal ordering (one-way ordering of time) of time-series data and to the introduction of a novel Quantum Re-ordering Buffer (QRB) structure within TrTables. The QRB is an in-memory data structure that supports the re-shuffling of out-of-order timestamps (and corresponding observation data) within a time window. This provides for the scenario that due to best-effort network delivery, IoT time-series sensor data arrives out-of-order on occasion.

To explain the TrTables data structure, three main components, the QRB, the in-memory *memtable* and the on-disk TrTable file are discussed as follows. Definition 5.1 introduces the term quantum, q, and the QRB, Q. This is followed by an explanation of the *memtable* and the TrTable as shown in Figure 5.3.

Definition 5.1 (Quantum Re-ordering Buffer, Q, and Quantum, q)**.** A quantum re-ordering buffer, Q, is a list-like window that contains a number of timestamp-row pairs as elements. A quantum, q, is the amount of elements within Q to cause an *expiration operation* where an insertion sort is performed on the timestamps of q elements and the first $a \times q$ elements are flushed to the *memtable*, where $0 < a \leq 1$. The remaining $(1 - a) \times q$ elements now form the start of the window.

An insertion sort which has a complexity of $O(nk)$ is used to re-order the QRB. k, the furthest distance of an element from its final sorted position, is small if the window is already partially sorted, which is the case unless there are significant network delays. Any timestamp now entering the re-ordering buffer less than the minimum allowed timestamp, t_{minA} (the first timestamp in the buffer) is rejected, marked as ‘late’ and

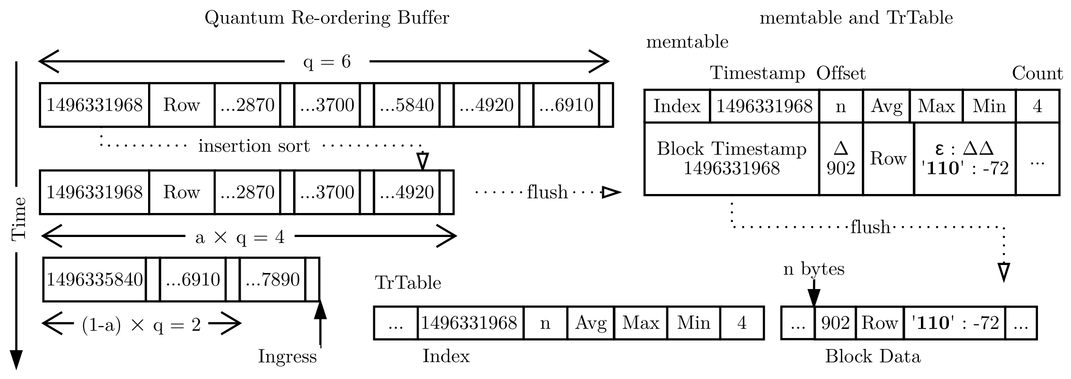


Figure 5.3: Operation of the Quantum Re-ordering Buffer, memtable and TrTable over Time

returned with a warning. Figure 5.3 shows the operation of Q over time (along the y-axis). When Q has 6 elements and $q = 6$, an expiration operation occurs where an insertion sort is performed and the first 4 sorted elements are flushed to the *memtable*. A new element that enters has timestamp, $t = 1496337890$, which is greater than $t_{minA} = 1496335840$ and hence is appended at the end of Q.

The *memtable*, also shown in Figure 5.3, consists of an index entry, i, that stores values of the block timestamp, current TrTable offset and average, maximum, minimum and counts of the row data which are updated when elements from Q are inserted. It also stores a block entry, b, which contains the current compressed time-partitioned block data. The *memtable* gets flushed to a TrTable file once it reaches the time-partitioned block size, b_{size}. Each time-series has a *memtable* and corresponding TrTable file on-disk.

Like the SSTable and memtable combination, operations on TrTables take the form of efficient sequential reads and writes due to it maintaining a sorted order both in-memory with a memtable and on disk with a TrTable file. However, unlike the SSTable and memtable, each key in a TrTable file is a block timestamp while each value is a time-partitioned block, compressed with both state-of-the-art timestamp and value compression. The efficiency of an interface of only sequential reads and writes is magnified by the use of compressed, compact blocks, ordered blocks which require comparatively less disk input-output operations. The block index table which is produced during ingestion and requires no further maintenance as it is immutable once written to disk as a parallel to a TrTable file, boosts the performance of range and aggregation queries.

TrTables also inherit other beneficial characteristics from SSTables, which are fitting for storing time-series IoT data, like simple locking semantics for only the *memtable* with no contention on immutable TrTable files. However, TrTables do not support expensive updates and deletions outside of the time window of the quantum re-ordering buffer as this thesis argues that there is no established use case for individual points within an IoT time-series in the (distant) past to be modified. In the situation that such a use case arises, this can be more efficiently managed by an offline batch process instead.

5.2.6 Data Structure Microbenchmark Results

Microbenchmarks aimed to measure 3 metrics that characterised the performance of each data structure: write performance, read amplification and space amplification. Write performance was measured by the average time taken to ingest each of the datasets over a 100 tries. Borrowing from the definition of Kuszmaul (2014), 'the number of input-output operations required to satisfy a particular query' is the read amplification that was measured here by taking the average execution time of a 100 tries of scanning the whole database (*scan*) and the average execution time of range queries over a 100 pairs of deterministic pseudo-random values with a fixed seed from the entire time range of each dataset (*range*). Space amplification is the 'space required by a data structure that can be inflated by fragmentation or temporary copies of the data' and was measured here by the resulting size of the database after compaction operations. Each time-partitioned block was compressed using δ_{Δ} and C_{gor} compression for timestamps and values respectively. Results for each of the metrics follow in Sections 5.2.6.1, 5.2.6.2 and 5.2.6.3. A similar experimental setup as with the compression microbenchmarks (Section 5.2.4) was used.

5.2.6.1 Space Amplification and the Effect of Block Size, b_{size}

The block size, b_{size}, refers to the maximum size that each time-partitioned block occupies within a data structure. Base 2 multiples of 2^{12}, the typical block size on file systems, were used such that $b_{size} = 2^{12} \times 2^{x}$ and in these experiments $x = \{2..8\}$ was used. Figure 5.4 shows the database size in bytes, which suggests the space amplification, for the Shelburne and Taxi datasets of each data structure at varying b_{size}. Both TrTables-LSM-tree and B+-tree-Hash-tree pairs have database sizes that were almost identical with the maximum difference only about 0.2%, hence, they were grouped together for better visibility in the figure.

A trend where the database size decreased as b_{size} decreased was observed. This is a characteristic of the C_{gor} algorithm used for value compression described in Section 5.2.2.2 as more 'localised' compression occurs. Each new time-partitioned block would trigger the else clause in the $gor(r)$ function to encode the longer $'11'.l.m.n$, however, the subsequent $lead(p)$ and $trail(p)$ were also likely to be smaller and more 'localised' and hence, fewer significant bits needed to be used for values in the block.

TrTables and LSM-tree data structures have smaller database sizes than the B+-tree and Hash-tree data structures for both datasets. As sorted keys and time-partitioned blocks in append-only, immutable structures like TrTables and the LSM-tree after compaction are stored in contiguous blocks on disk, they are expectedly more efficiently packed and stored size-wise. Results from SRBench are omitted as the largest time-partitioned block

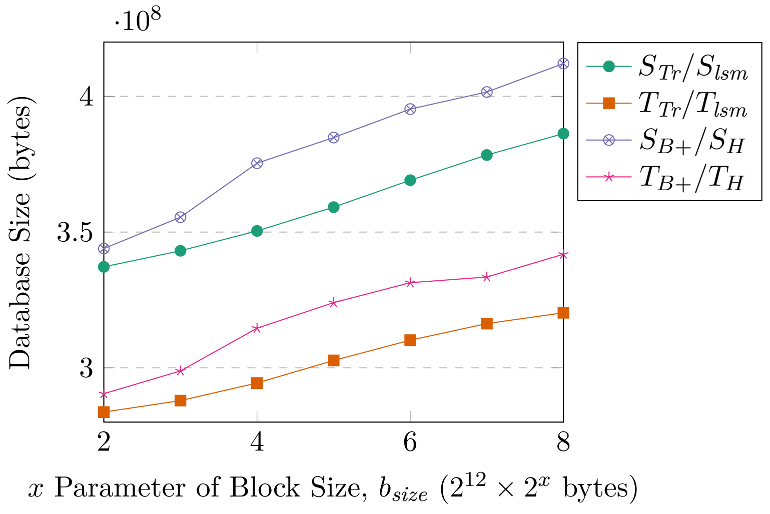


FIGURE 5.4: Database Size of Data Structures of Datasets at Varying b_{size}

Datasets: Shelburne = S, Taxi = T, **Data Structures:** TrTables = Tr, LSM-tree = lsm, B+-tree = B+, Hash-tree = H

across all the stations was smaller than the smallest b_{size} where $x = 2$, hence, there was no variation across different x values and b_{size}.

Key clashing was avoided in tree-based stores for the Taxi dataset, where multiple trip records had the same starting timestamp, by using time-partitioned blocks where $b_{size} > s_{size}$, where s_{size} was the longest compressed sequence with the same timestamp.

5.2.6.2 Write Performance

Figure 5.5 shows the ingestion time in seconds for the Shelburne and Taxi datasets of each data structure when varying b_{size}. Both TrTables and LSM-tree performed consistently across b_{size} due to append-only sequential writes which corresponded to their log-structured nature. TrTables were about 8 and 16 seconds faster on average than LSM-tree for the Taxi and Shelburne datasets respectively due to no overhead of a compaction process. Both the Hash-tree and B+-tree performed much slower (up to 10 times slower on Taxi between the B+-tree and TrTables when $x = 2$) on smaller b_{size} as each of these data structures were comparatively not write-optimised and the trees became expensive to maintain as the amount of keys grew. When $x = 8$, the ingestion time for LSM-tree and Hash-tree converged, while the B+-tree was still slower and TrTables were still about 10s faster for both datasets. At that point, the bottleneck was no longer due to write amplification but rather subject to disk input-output speeds.

For the concurrent B+-tree and and Hash-tree, both parallel and sequential writers were tested and the faster parallel times were recorded in the microbenchmark results. In the parallel implementation, the write and commit operations for each time-partitioned block

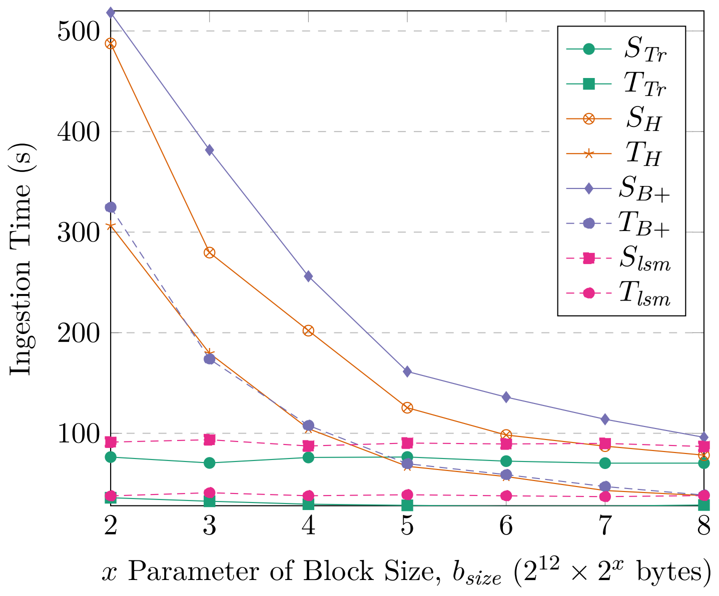


FIGURE 5.5: Ingestion Times of Data Structures of Datasets at Varying b_{size}

Datasets: Shelburne = S, Taxi = T, **Data Structures:** TrTables = Tr, LSM-tree = lsm, B+-tree = B+, Hash-tree = H

(a key-value pair) were handed to worker threads from a common pool using Kotlin's lightweight asynchronous coroutines[4].

5.2.6.3 Read Amplification

Figure 5.6 shows the execution time for a full scan on each data structure while varying b_{size} and Figure 5.7 shows the execution time for range queries. All scans and queries were averaged across 100 tries and for the range queries, the same pseudo-random ranges with a fixed seed were used. The write-optimised LSM-tree performed the worst for full scans. The B+-tree and Hash-tree performed quite similarly while TrTables recorded the fastest execution times. A full scan on a TrTable file was efficient, consisting of a straightforward sequential read of the TrTable file with no intermediate seeks necessary, hence, almost no read amplification.

From the results of the range queries in Figure 5.7, it can be seen that the LSM-tree had the highest read amplification for both datasets as it had to perform a scan of keys across levels while trying to access a sub-range of keys. The Hash-tree had the second highest read amplification, which was expected as it had to perform random input-output operations to retrieve time-partitioned blocks based on the distribution by the hash function. There is the possibility of using an order-preserving minimal perfect hashing function as proposed by Czech et al. (1992) at the expense of hashing performance and space, however, this was out of the scope of the microbenchmarks.

[4]https://github.com/Kotlin/kotlinx.coroutines

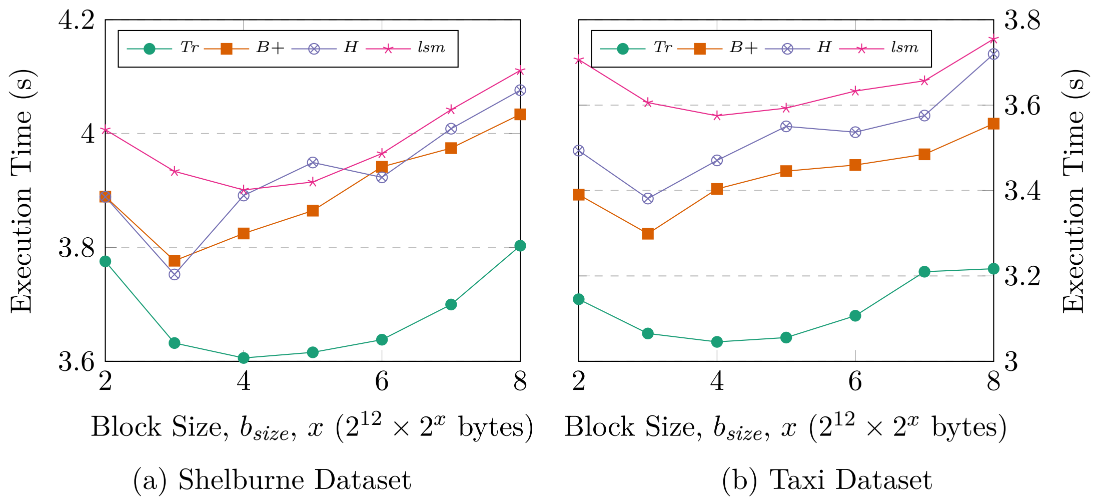


FIGURE 5.6: Full Scan Execution Time per b_{size}

Data Structures: TrTables = Tr, LSM-tree = lsm, B+-tree = B+, Hash-tree = H

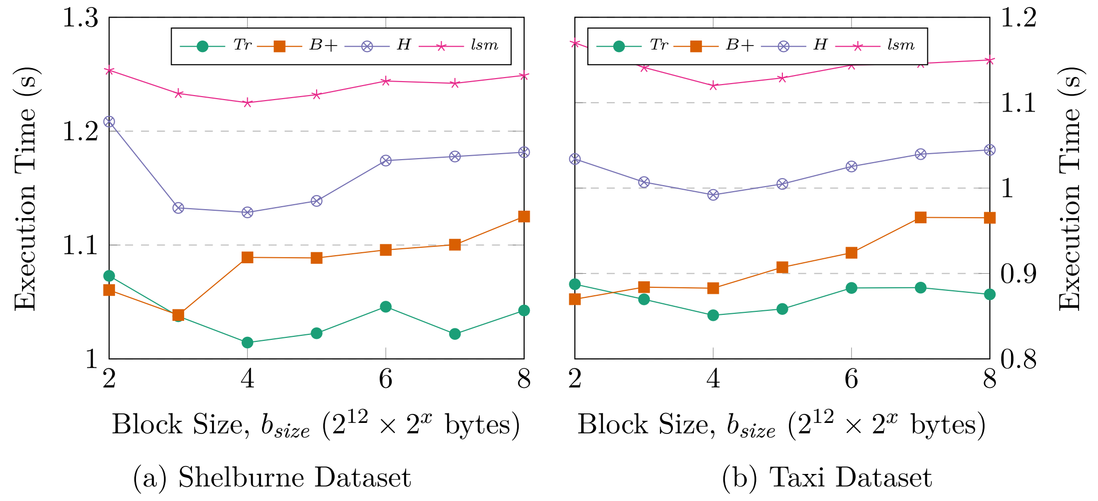


FIGURE 5.7: Range Query Execution Time per b_{size}

Data Structures: TrTables = Tr, LSM-tree = lsm, B+-tree = B+, Hash-tree = H

TrTables also had better performance on both datasets than the read-optimised B+-tree due to its block index that guaranteed a maximum of just one seek operation on the related TrTable file.

From the microbenchmarks on these IoT datasets, $2^{12} \times 2^4$ bytes was observed to be the most suitable b_{size} for reads and TrTables were noted to have the best performance for both full scans and range queries at this particular b_{size}.

5.2.6.4 Data Structure Performance Round-up: TrTables and 64KB

TrTables had the best write performance and storage size due to its immutable, compressed, write-optimised structure that benefited from fast, batched sequential writes.

The in-memory quantum re-ordering buffer and memtable support ingestion of out-of-order, unevenly-spaced data within a window, which was a requirement for IoT time-series data explored previously in Section 3.1. Furthermore, the memtable allowed batched writes and helped amortise the compression time. A b_{size} of 64 KB when $x = 4$ for a size of $2^{12} \times 2^x$ bytes with TrTables also provided the best read performance across both full scans and range queries on the various datasets.

B+-tree and Hash-tree were observed to have higher write amplification, especially for a smaller b_{size} and LSM-tree had higher read amplification.

5.3 Design of TritanDB for IoT Time-series Data

The Time-series Rapid Internet of Things Analytics Database (TritanDB) is a realisation of the Map-Match-Operate abstraction, described previously in Section 4.2, that includes a database tier optimised for time-series data from the Internet of Things. This database tier design follows on from findings made in the previous section, Section 5.2, which evaluated compression and data structures for IoT time-series data. The design of TritanDB is presented in four sections with Section 5.3.1 introducing the input stack, Section 5.3.2 describing the storage engine utilising compression and the TrTables data structure, Section 5.3.3 elaborating on the query engine design that realises Map-Match-Operate and finally, Section 5.3.4 explaining how concurrency is supported.

5.3.1 Input Stack: A Non-Blocking Req-Rep Broker and the Disruptor Pattern

The Constrained Application Protocol (CoAP)[5], MQTT[6] and HTTP are just some of many protocols used to communicate between devices in the IoT. Instead of making choices between these protocols, TritanDB implements a general non-blocking Request-Reply (Req-Rep) broker that exposes a ZeroMQ[7] Application Programming Interface (API) as sockets to utilise the many client libraries provided so that each of the protocols can be implemented on top of it. The broker is divided into a Router frontend component that clients bind to and send requests and a Dealer backend component that binds to a worker to forward requests. Replies are sent through the dealer to the router and then to clients. Figure 5.8 shows the broker design. All messages are serialised as protocol buffer messages, a small, fast, and simple means of binary transport for structured data with minimal overhead.

[5]http://coap.technology/
[6]http://mqtt.org/
[7]http://zeromq.org/

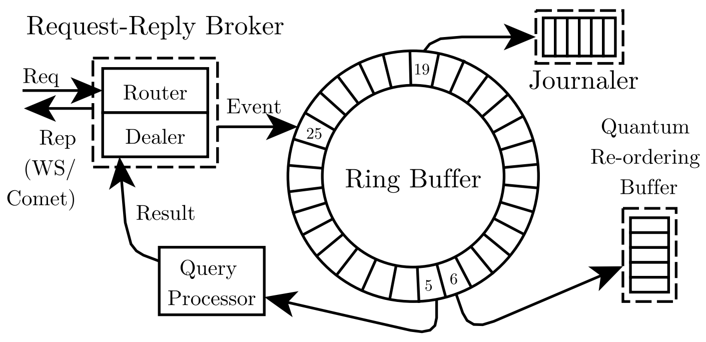


FIGURE 5.8: Request-Reply Broker Design for Ingestion and Querying Utilising the Disruptor Pattern

The worker that the dealer binds to is a high performance queue drawing inspiration from work on the Disruptor pattern[8] used in high frequency trading that reduces both cache misses at the CPU-level and locks requiring kernel arbitration by utilising just a single thread. Data is referenced, as opposed to memory being copied, within a ring buffer structure. Furthermore, multiple processes can read data from the ring buffer without overtaking the head, ensuring consistency in the queue. Figure 5.8 shows the ring buffer with the producer, the dealer component of the broker, writing an entry at slot 25, which it has claimed by reading from a *write counter*. Write contention is avoided as data is owned by only one thread for write access. Once done, the producer updates a *read counter* with slot 25, representing the cursor for the latest entry available to consumers. The pre-allocated ring buffer with pointers to objects has a high chance of being laid out contiguously in main memory, thus supporting cache striding. Cache striding refers to the cache recognising regular stride-based memory access patterns, usually under 2 KB in stride length, and pre-emptively loading these in the cache. Furthermore, garbage collection (in Java) is also avoided with pre-allocation. Consumers wait on the memory barrier and check they never overtake the head with the *read counter*.

A journaler at slot 19 records data on the ring buffer for crash recovery. If two journalers are deployed, one could record even slots while the other odd slots for better concurrent performance. The Quantum Re-ordering Buffer (QRB) reads a row of time-series data to be ingested from slot 6. The memory needs to be copied and deserialised in this step. A Query Processor reads a query request at slot 5, processes it and a reply is sent through the router to the client that contains the result of the query.

The disruptor pattern describes an event-based asynchronous system. Hence, requests are converted to events when the worker bound to a dealer places them on the ring buffer. Replies are independent of the requests although they do contain the address of the client to respond to. Therefore, in an HTTP implementation on top of the broker, replies can

[8]https://lmax-exchange.github.io/disruptor/

be sent chunked via a connection utilising Comet style programming (long polling). In an MQTT implementation, results are pushed to the client through a publish-subscribe system. Other alternatives are duplex communication via protocols like Websockets.

5.3.2 Storage Engine: TrTables and M_{map} Models and Templates

TritanDB Tables (TrTables) form the basis of the storage engine and are a persistent, compressed, ordered, immutable and optimised time-partitioned block data structure. TrTables consist of four major components: a *quantum re-ordering buffer* to support ingestion of out-of-order timestamps within a time quantum, the *memtable*, a sorted in-memory time-partitioned block, and persistent, sorted TrTable files on-disk for each time-series, consisting of *time-partitioned blocks* and a *block and aggregate index*. Section 5.2.5.4 covers the design of TrTables in more detail.

Each time-partitioned block is compressed using the adaptive Delta-RLE-Rice encoding for lower precision timestamps and delta-of-delta compression for higher precision timestamps (milliseconds onwards) as explained in Section 5.2.1. Value Compression uses the Gorilla algorithm explained in Section 5.2.2. Time-partitioned blocks of 64 KB are used, as proposed in Section 5.2.6.4 on evaluating the microbenchmark results.

Additionally, as part of a Map-Match-Operate engine, TritanDB requires data model mappings to execute graph queries on the database tier. To ease the process of designing rich graph models for IoT time-series data, the storage engine includes a templating feature to automatically generate m_{map} model mappings for new time-series. When a time-series is created and a row of data is added to TritanDB, a customisable template based on the Semantic Sensor Network Ontology by Compton et al. (2012) models each column as an observation and automatically generates a m_{map} model map. The m_{map} can subsequently be modified on-the-fly, imported from RDF serialisation formats (XML, JSON, Turtle, etc.) or exported. Internally, TritanDB stores the union of all m_{map}, a mapping closure, M_{map^C}, within an in-memory model. Changes are persisted to disk using an efficient binary format, RDF Thrift[9]. The use of customisable templates helps to realise a reduced configuration philosophy on setup and on the ingestion of time-series data, but still allows the flexibility of evolving the rich graph model and adding additional metadata and data.

5.3.3 Query Engine: Swappable Interfaces

Figure 5.9 shows the modular query engine design in TritanDB that can be extended to support other rich graph models and query languages besides RDF and SPARQL. The query engine builds on the S2S engine work from Section 4.2.4. The impact on runtime

[9]https://afs.github.io/rdf-thrift/

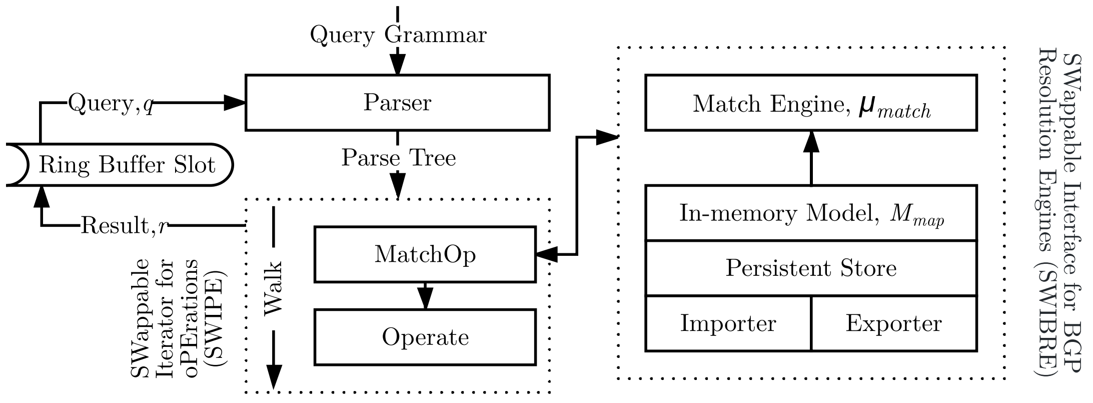

Query Grammar
Query, q
Parser
Ring Buffer Slot
Parse Tree
Result, r
MatchOp
Operate
Walk
SWappable Iterator for oPErations (SWIPE)
Match Engine, μ_{match}
In-memory Model, M_{map}
Persistent Store
Importer
Exporter
SWappable Interface for BGP Resolution Engines (SWIBRE)

FIGURE 5.9: Modular Query Engine with Swappable Interfaces for *Match*, *Operate* and Query Grammar

performance due to providing for modularity is minimised as each module is connected by pre-compiled interfaces and utilises reflection in Kotlin and the performance is empirically justified by the experiments in Section 5.4.

There are three main modular components in the query engine, the *parser*, the *matcher* and the *operator*. The compiled query grammar enables a *parser* to produce a parse tree from an input query, q. The query request is accessed from the input ring buffer described in Section 5.3.1. The parse tree is walked by the *operator* component that sends the G leaves of a parse tree to the *matcher*. The *matcher* performs μ_{match} based on the relevant m_{map} model from the in-memory mapping closure M^C_{map} produced in Section 5.3.2. The $\mathbb{B}_{match}$ is returned to the *operator* which continues walking the parse tree and executing operations till a result, r is returned at the root. This result is sent back to the requesting client through the Request-Reply broker.

The default implementation of modules for TritanDB follows that of the S2S implementation of Map-Match-Operate, covered in Section 4.2.4, with graph queries expressed in SPARQL query grammar, the RDF graph model for the model tier and S2SML data modelling language for mappings. The match engine performing μ_{match}, by default, implements a minimal version of the Jena interface used in S2S that connects to the operate step via the SWappable Interface for BGP Resolution (SWIBRE) described in Section 4.2.4.2. The operate step that plugs in to the SWappable Iterator for oPerations (SWIPE)[10], iterates through the sequence of operations from the SPARQL query to generate a query execution plan for TritanDB. This is similar to the process from the S2S engine described previously except that instead of translating the query to SQL, the query execution plan, like that of the equation 4.2 in Section 4.2.3, is executed on the underlying storage engine from Section 5.3.2. The full set of operators implemented and the mapping to translate from SPARQL algebra is presented in Appendix C.

[10]https://eugenesiow.gitbooks.io/tritandb/

5.3.4 Design for Concurrency

Immutable TrTable files simplify the locking semantics to only the quantum re-ordering buffer (QRB) and memtable in TritanDB. Furthermore, reads on time-series data can always be associated with a range of time (if a range is unspecified, then the whole range of time) which simplifies the look up via a block index across the QRB, memtable and TrTable files. The QRB has the additional characteristic of minimising any blocking on the memtable writes as it flushes and writes to disk a TrTable as long as t_q, the time taken for the QRB to reach the next quantum expiration and flush to memtable, is more than t_{write}, the time taken to write the current memtable to disk.

The following listings describe some functions within TritanDB that elaborate on maintaining concurrency during ingestion and queries. The QRB is backed by a concurrent ArrayBlockingQueue in this implementation and inserting is shown in Listing 5.1 where the flush to memtable has to be synchronised. The insertion sort needs to synchronise on the QRB as the remainder $(1-a) \times q$ values are put back in. The '`QRB.min`' is now the maximum of the flushed times. Listing 5.2 shows the synchronised code on the memtable and index to flush to disk and add to the BlockIndex. A synchronisation lock is necessary as time-series data need not be idempotent (i.e. same data in the memtable and TrTable at the same time is incorrect on reads). The memtable stores compressed data to amortise write cost, hence flushing to disk, t_{write}, is kept minimal and the time blocking is reduced as well. Listing 5.3 shows that a range query checks the index to obtain the blocks it needs to read, which can be from the QRB, memtable or TrTable, before it actually retrieves each of these blocks for the relevant time ranges. Listing 5.4 shows the internal get function in the QRB for iterating across rows to retrieve a range.

```
fun insert(row) {
  if(QRB.length >= q) {
    synchronized(QRB) {
      arr = insertionSort(QRB.drain())
      QRB.put(remainder = arr.split(a*q,arr.length))
    } synchronized(memtable) {
      memtable.addAll(flushed = arr.split(0,a*q-1))
      memidx.update()
    }}
  row.time > QRB.min ? QRB.put(row, row.time) }
```

Listing 5.1: Insertion into the Quantum Re-ordering Buffer (QRB)

```
fun flushMemTable() {
  synchronized(memtable) {
    TrTableWriter.flushToDisk(memtable,memidx)
    BlockIndex.add(memidx)
    memidx.clear()
    memtable.clear()
  } }
```

Listing 5.2: Flush memtable and Index to Disk

```
fun query(start,end):Result {
  blocks = BlockIndex.get(start,end)
  for((btype,s,e,o) in blocks) { //relevant blocks
    when(btype) {
      ‘QRB’ -> r += QRB.get(s,e)
      ‘memtable’-> r += memtable.get(s,e)
      ‘trtable’ -> r += trReader.get(s,e,o) //offset
    }}
  return r }
```

Listing 5.3: Query a Range across Memory and Disk

Finally, handling faux nodes (Definition 4.6) in TritanDB requires a specific design for concurrency. The criteria for faux node materialisation in TritanDB is that the identifier produced within the IRI pattern should be stable and unique. This is achieved for the memtable and TrTables, which are immutable once written, by assigning a fixed universally unique identifier (UUID) to each block and concatenating the sequence number of each point in the time-series to this identifier. However, if the range of time specified in a query that has faux nodes (within $\mathbb{B}_{match}$) is unbounded or has a maximum timestamp larger than the minimum QRB timestamp (‘`QRB.min`’), the QRB is immediately flushed to the memtable.

```
fun QRB.get(start,end):Result {
  for(row in this.iterator()) {
    if(row.time in start...end) r.add(row)
    else if(row.time > end) break }
  return r }
```

Listing 5.4: Internal QRB Function to Get

5.4 Experiments, Results and Discussion

This section covers an experimental evaluation of TritanDB with other time-series, relational and NoSQL databases, commonly used to store time-series data. Results are presented and discussed across a range of experimental setups, datasets and metrics for each database. Section 5.4.1 introduces the experimental setup which provides hardware specifications covering a cloud-to-thing continuum and the design of each individual experiment to measure various performance metrics. Section 5.4.2 evaluates the storage and ingestion performance results while Section 5.4.3 evaluates the translation overhead of Map-Match-Operate in TritanDB and the query performance results.

Specification	Server1	Server2	Gizmo2	Pi2 B+
CPU	2×2.6 GHz	4×2.6 GHz	2×1 GHz	4×0.9 GHz
Memory	32 GB	4 GB	1 GB	1 GB
Disk Data Rate	380.7 MB/s	372.9 MB/s	154 MB/s	15.6 MB/s
OS	Ubuntu 14.04 64-bit			Raspbian Jessie 32-bit

Table 5.4: Specifications of Each Experimental Setup

5.4.1 Experimental Setup and Design

Due to the emergence of large volumes of streaming IoT data and a trend towards fog computing networks that Chiang et al. (2017) described as an 'end-to-end horizontal architecture that distributes computing, storage, control, and networking functions closer to users along the cloud-to-thing continuum', there is a case for experimenting on both cloud and Thing setups with diverse hardware specifications.

Table 5.4 summarises the CPU, memory, disk data rate and Operating System (OS) specifications of each experimental setup. The disk data rate was measured by copying a file with random chunks and syncing the filesystem to remove the effect of caching. *Server1* is a high memory setup with high disk data rate but lower compute (less cores). *Server2* on the other hand is a lower memory setup with more CPU cores and a similarly high disk data rate. Both of these setups represent cloud-tier specifications in a fog computing network. The *Pi2 B+* and *Gizmo2* setups represent the Things-tier as compact, lightweight computers with low memory and CPU clock speeds, ARM and x86 processors respectively and a Class 10 SD card and a mSATA SSD drive respectively with relatively lower disk data rates. The Things in these setups perform the role of low-powered, portable base stations or embedded sensor platforms within a fog computing network and are sometimes referred to as fog computing nodes.

The databases tested were drawn from the survey of time-series databases in Section 2.2.1 and included state-of-the-art time-series databases InfluxDB (Influx Data, 2017) and Akumuli (Lazin, 2017) with innovative LSM-tree and B+-tree inspired storage engine designs respectively. Two popular NoSQL, schema-less databases that underly many emerging time-series databases: MongoDb[11] and Cassandra (Lakshman and Malik, 2010) were also compared against. OpenTSDB (StumbleUpon, 2017), an established open-source time-series database that works on HBase, a distributed key-value store, was also tested. Other databases tested against include the fully-featured relational database, H2 SQL[12] and the search-index-based ElasticSearch (Elastic, 2017) which was shown to perform well for time-series monitoring by Mathe et al. (2015).

[11]https://www.mongodb.com/
[12]http://www.h2database.com/

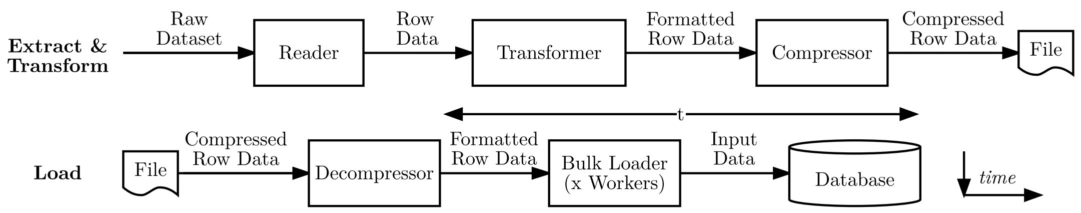


FIGURE 5.10: Extract, Transform and Load Stages of Ingestion Experiment over Time

Experiments on each setup were performed and the design of each experiment is described in Sections 5.4.1.1 and 5.4.1.2 to test the ingestion performance and query performance respectively for IoT data.

5.4.1.1 Ingestion Experiment Design

Figure 5.10 summarises the ingestion experiment process in well-defined Extract, Transform and Load stages. A reader sent the raw dataset as rows to a transformer in the Extract stage. In the Transform stage, the transformer formatted the data according to the intended database's bulk write protocol format and compressed the output using Gzip to a file. In the Load stage, the file was decompressed and the formatted data was sent to the database by a bulk loader which employed x workers, where x corresponded to the number of cores on the experimental setup. The average ingestion time, t, was measured by averaging across 5 runs for each setup, dataset and database. The average rate of ingestion for each setup, s^1, s^2, p and g was calculated by dividing the number of rows of each dataset by the average ingestion time. The storage space required for the database was measured 5 minutes after ingestion to allow for any additional background processes to conclude, as a diverse set of databases were tested. Each database was deployed in a Docker container for uniformity.

The schema design for MongoDB, Cassandra and OpenTSDB were optimised for read performance in ad-hoc querying scenarios and followed the recommendations of Persen and Winslow (2016b) in their series of technical papers on mapping the time-series use case to each of these databases. This approach modelled each row by their individual fields in documents, columns (Persen and Winslow, 2016a) or key-value pairs (Persen and Winslow, 2016c) respectively with the tradeoff of storage space and ingestion performance for better query performance on a subset of queries deemed important.

5.4.1.2 Query Experiment Design

The aim of the query experimentation was to determine the overhead of query translation and the performance of TritanDB against other state-of-the-art stores for IoT data and queries, particularly analytical queries useful for advanced applications. Since, there was

Database	Shelburne	Taxi	SRBench
TritanDB	**0.350**	**0.294**	0.009
InfluxDb	0.595	**0.226**[a]	0.015
Akumuli	0.666	0.637	**0.005**
MongoDb	5.162	6.828	0.581
OpenTSDB	0.742	1.958	0.248
H2 SQL	1.109/2.839[b]	0.579/1.387	0.033
Cassandra	1.088	0.838	0.064
ElasticSearch (ES)	2.225	1.134	- [c]

[a]InfluxDb points with the same timestamp were silently overwritten (due to its log-structured-merge-tree-based design), hence, database size was smaller as there were only 3.2×10^6 unique timestamps of 4.4×10^6 rows.

[b](size without indexes, size with an index on the timestamp column)

[c]As each station was an index, ES on even the high RAM *Server1* setup failed when trying to create 4702 stations.

TABLE 5.5: Storage Space (in GB) Required for Each Dataset on Different Databases

little precedence in terms of time-series benchmarks for the IoT, the types of queries were advised by literature for measuring the characteristics of financial time-series databases, an established use case, from the set of queries proposed by Jacob et al. (1999). Each of the following query types was executed and averaged 100 times using the pseudo-random time ranges generated from a fixed seed:

1. Cross-sectional range queries that access all columns of a dataset.
2. Deep-history range queries that access a random single column of a dataset.
3. Aggregating a subset of columns of a dataset by arithmetic mean (average).

The execution time of each query was measured as the time from sending the query request to when the query results had been completely written to a file on disk. The above queries were measured on the Shelburne and GreenTaxi datasets as the time periods of SRBench from each weather station were too short for effective comparison.

All source code and information on database-specific formats for ingestion and query experiments are available from an open Github repository[13].

5.4.2 Discussion of the Storage and Ingestion Results

Table 5.5 shows the storage space, in gigabytes (GB), required for each dataset with each database. TritanDB, that makes use of time-series compression, time-partitioning blocks and TrTables that have minimal space amplification, had the best storage performance for the Shelburne and GreenTaxi datasets and was second to Akumuli, which had a

[13]https://github.com/eugenesiow/influxdb-comparisons

smaller 4 KB block size, for the SRBench dataset which consisted of many stations each storing a shorter range of time. TritanDB was however still about 4 times more storage efficient than the H2 relational database and 65 times smaller than the read-optimised MongoDb schema design. InfluxDb and Akumuli, that also utilise time-series compression, produced significantly smaller database sizes than the other relational and NoSQL stores.

MongoDb needed the most storage space amongst the databases for the read-optimised schema design chosen, while search index based ElasticSearch (ES) also required more storage. ES also struggled with ingesting the SRBench dataset, when creating many time-series as separate indexes, and failed even on the high RAM *Server1* configuration. In this design, each of the 4702 stations were an index on their own to be consistent with the other NoSQL database schema designs.

As InfluxDb silently overwrote rows with the same timestamp, it produced a smaller database size for the GreenTaxi dataset of trips. Trips for different taxis that started at the same timestamp were overwritten. Only 3.2×10^6 of 4.4×10^6 trips were eventually stored. It was possible to use unique tags to differentiate each taxi in InfluxDb but this was also limited by the fixed maximum tag cardinality of 100,000, hence, not feasible.

TritanDB had from 1.7 times to an order of magnitude better storage efficiency than all other databases for the larger Shelburne and Taxi datasets. It had a similar 1.7 to an order of magnitude advantage over all other databases except Akumuli for SRBench.

Table 5.6 shows the average rate of ingestion, in rows per second, for each dataset with each database, across setups. From the Server1 and Server2 results, it can be observed that TritanDB, InfluxDb, MongoDb, H2 SQL and Cassandra all performed better with more processor cores rather than more memory, while Akumuli and OpenTSDB performed slightly better on the high memory Server2 setup which also had a slightly better disk data rate. For both setups and all datasets, TritanDB had the highest rate of ingestion from 1.5 times to 3 orders of magnitude higher on Server1 and from 2 times to 3 orders of magnitude higher on Server2 due to the ring buffer and sequential write out to TrTable files.

The Class 10 SD card, a Sandisk Extreme with best-of-class advertised write speeds, of the Pi2 B+ setup was an order of magnitude slower than the mSATA SSD of the Gizmo2 setup. Certain databases like Akumuli did not support the 32-bit ARM Pi2 B+ setup at the time of writing, therefore, some experiments listed in Table 5.6 could not be carried out. On the Gizmo2, TritanDB ingestion rates were about 8 to 12 times slower than on Server2 due to a CPU with less cores, however, it still performed the best among the databases and was at least 1.3 times faster than its nearest competitor, H2 SQL.

Database	Server1 (10^3 rows/s)			Server2 (10^3 rows/s)		
	$s^1_{shelburne}$	s^1_{taxi}	$s^1_{srbench}$	$s^2_{shelburne}$	s^2_{taxi}	$s^2_{srbench}$
TritanDB	**173.59**	**68.28**	**94.01**	**252.82**	**110.07**	**180.19**
InfluxDb	1.08	1.05	1.88	1.39	1.34	1.09
Akumuli	49.63	18.96	61.78	46.44	17.71	59.23
MongoDb	1.35	0.39	1.23	1.96	0.58	1.81
OpenTSDB	0.26	0.08	0.24	0.25	0.07	0.22
H2 SQL	80.22	45.23	51.89	84.42	52.67	77.12
Cassandra	0.90	0.25	0.78	1.47	0.45	1.66
ES	0.10	0.09	-	0.11	0.04	-
	Pi2 B+ (10^2 rows/s)[a]			Gizmo2 (10^3 rows/s)		
	$p_{shelburne}$	p_{taxi}	$p_{srbench}$	$g_{shelburne}$	g_{taxi}	$g_{srbench}$
TritanDB	**73.68**	**26.58**	**48.42**	**32.62**	**12.77**	**14.05**
InfluxDb	1.33	1.28	1.43	0.26	0.25	0.28
Akumuli	-[b]	-	-	9.79	3.84	10.48
MongoDb	-[c]	-	1.78	0.22	0.06	0.21
OpenTSDB	0.10	0.03	0.08	0.05	0.02	0.05
H2 SQL	32.11	18.80	34.26	15.13	8.30	10.42
Cassandra	0.67	0.27	0.85	0.16	0.05	0.15
ES	-[d]	-	-	0.03	0.01	-

[a]Note the difference in order of magnitude of 10^2 rather than 10^3
[b]At the time of writing, Akumuli does not support ARM systems.
[c]Ingestion on MongoDb on the 32-bit Pi2 for larger datasets fails due to memory limitations.
[d]Ingestion on ElasticSearch fails due to memory limitations (Java heap space).

TABLE 5.6: Average Rate of Ingestion for Each Dataset on Different Databases

5.4.3 Evaluation of Query Performance and Translation Overhead

TritanDB supports a rich graph model for integrating IoT metadata and data over model and database tiers. This is possible through the use of the Map-Match-Operate method which involves a query translation process. Hence, the query translation overhead is a part of benchmarking query performance against other state-of-the-art time-series databases, relational databases and NoSQL databases used to store time-series data. Section 5.4.3.1 measures the query translation overhead independently, while Section 5.4.3.2 presents the evaluation of query latency for different query types on each database for the Shelburne and GreenTaxi (Taxi) datasets. TritanDB results included the time taken for the query translation process (although not from a cold start of the JVM).

5.4.3.1 Query Translation Overhead

The translation overhead was measured as the time taken to parse the input query, perform the *match* and *operate* steps and produce a query plan for execution. The JVM

Setup	Cross-sectional	Deep-history	Aggregation
Server1	53.99	52.16	53.72
Server2	58.54	53.31	53.99
Pi2 B+	581.99	531.95	537.80
Gizmo2	449.33	380.18	410.58

Table 5.7: Average Query Translation Overhead (in ms) for Different Query Types across Different Setups

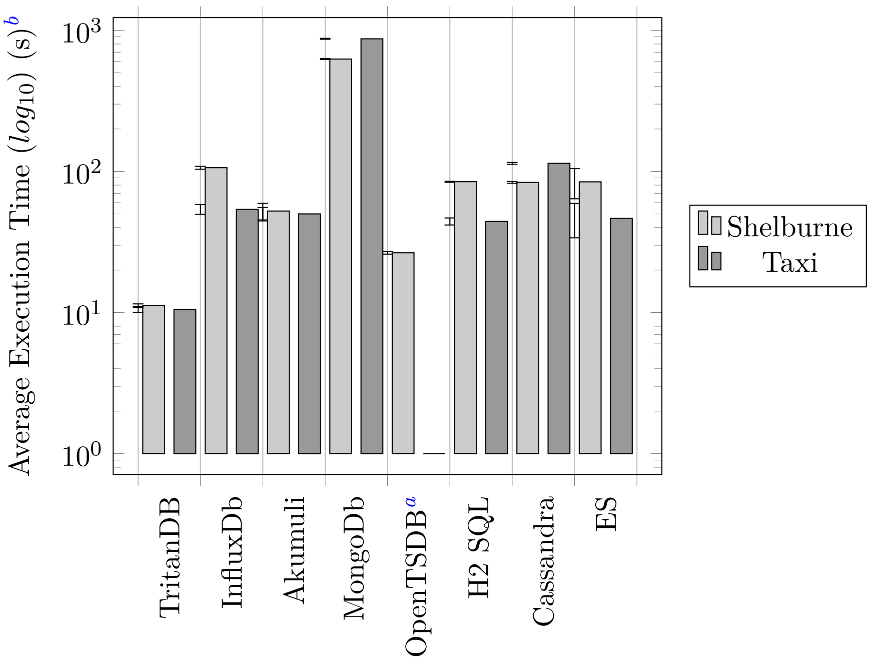
was shutdown after each run and a Gradle[14] compile and execute task started the next run (performing a cold start) to minimise the impact of the previous run's caching on the run time. Time for loading the models in the *map* step was not included as this occurs on startup of TritanDB rather than at query time. Table 5.7 shows the query translation overhead, averaged across a 100 different queries of each type (e.g. cross-sectional, deep-history) and then averaged among datasets, across different setups.

The mean query overhead for all three types of queries were similar, with deep-history queries the simplest in terms of query tree complexity followed by aggregation and then cross-sectional queries, which involved unions between graphs. The results reflected this order. Queries on the Pi2 B+ and Gizmo2 were an order of magnitude slower than those running on the server setups, however, still executed in sub-second times and in practice are much improved with the caching of query trees. When executed in a sequence without restarting the JVM, subsequent query overhead was under 10ms on the Pi2 B+ and Gizmo2 and under 2ms on the server setups. The Gizmo2 was faster than the Pi2 B+ in processing queries and Server2 was slightly faster than Server1.

5.4.3.2 Cross-sectional, Deep-history and Aggregation Queries

Figure 5.11 shows the results of a cross-sectional range query on the server setups Server1 (s^1) and Server2 (s^2). As cross-sectional queries are wide and involve retrieving many fields/columns from each row of data, the columnar schema design in MongoDb, where each document contained a field of a row, had the slowest average execution time. Furthermore, the wider Taxi dataset of 20 columns had longer execution times than the narrower Shelburne dataset of 6 columns. This disparity between the results of each dataset was also observed for Cassandra, where a similar schema design was used. Row-based H2 SQL and ElasticSearch, where each row was a document, showed the inverse phenomena between datasets. Purpose-built time-series databases TritanDB, OpenTSDB and Akumuli performed the best for this type of query. TritanDB had the fastest average query execution time of about 2.4 times better than the next best, OpenTSDB running on HBase, which did not support the Taxi dataset due to multiple duplicate timestamps in

[14]https://gradle.org/



[a]OpenTSDB queries cannot be executed on the Taxi dataset because multiple duplicate timestamps are not supported

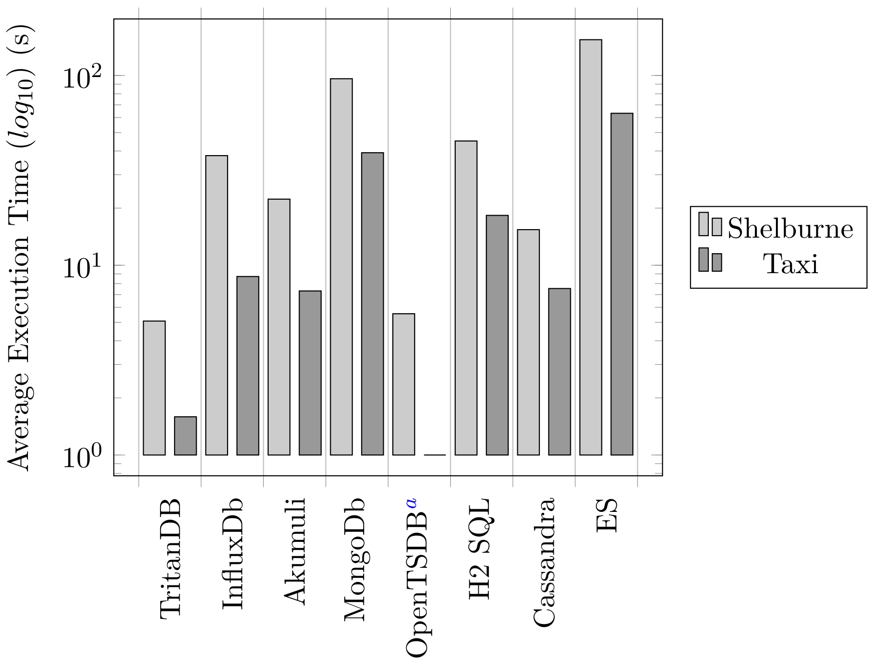
[b]The execution times are the mean of s^1 and s^2 while the confidence interval to the left of each bar indicates the range of execution time.

FIGURE 5.11: Cross-sectional Range Query Average Execution Times for Databases, Mean of s^1 and s^2 Setups

the dataset, and 4.7 times faster than the third best, Akumuli, for cross-sectional range queries on server setups.

Figure 5.12 shows the average execution time for each database taking the mean time from s^1 and s^2 setups for deep-history range queries. It was observed that all databases and not only those that utilised columnar storage design performed better on the Taxi dataset than on Shelburne when retrieving deep-history queries on a single column due to there being less rows of data in the Taxi dataset. TritanDB once again had the fastest query execution times for deep-history queries and was 1.1 times faster than OpenTSDB and 3 times faster than the third best Cassandra. Both OpenTSDB and Cassandra had columnar schema design optimised for retrieving deep history queries which explained the narrower performance gap than for cross-sectional queries. ElasticSearch, which stored rows as documents and required a filter to retrieve a particular field from each document to form the single deep-history column, performed poorly.

Table 5.8 shows the average execution time for various queries on TritanDB on both Things setups. The Gizmo2 was faster than the Pi2 B+ but was from 3 to 13 times slower than the mean of the server setups execution times across various queries. The Pi2 B+ setup was 6 to 36 times slower than the servers. It was observed that there was an inversion of results between the narrow Shelburne and wide Taxi datasets, where Shelburne execution times were lower than those of the Taxi dataset, on the Gizmo2



[a]OpenTSDB queries cannot be executed on the Taxi dataset because multiple duplicate timestamps are not supported

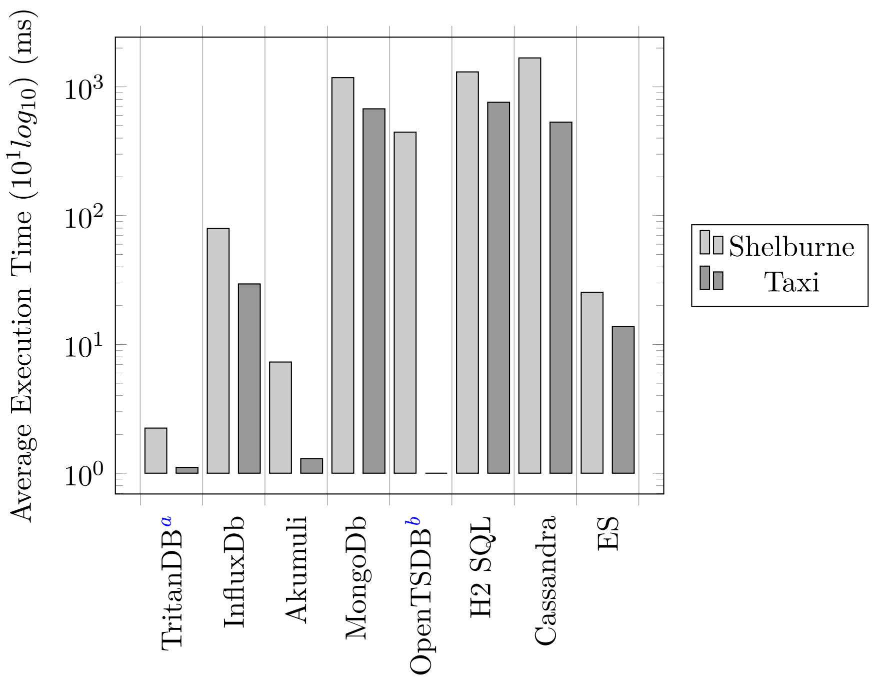
FIGURE 5.12: Deep-history Range Query Average Execution Time Mean of s^1 and s^2 on Each Database

TritanDB	Pi2 B+ (s)		Gizmo2 (s)		Ratio (s:p:g)	
	$p_{shelburne}$	p_{taxi}	$g_{shelburne}$	g_{taxi}	$r_{shelburne}$	r_{taxi}
Cross-Sectional	388.18	375.97	54.44	62.19	1:35:5	1:36:6
Deep-History	111.10	47.95	19.03	21.37	1:22:4	1:30:13
Aggregation	0.13	0.16	0.07	0.06	1:6:3	1:15:6

TABLE 5.8: Average Query Execution Time for Query Types on TritanDB on the Pi2 B+ and Gizmo2

for both the cross-sectional and deep-history queries where the bottleneck was the CPU for reading and decompressing time-partitioned blocks. However, the bottleneck shifted to the slow write speed of the Pi2 B+ to the SD card and so the larger number of rows in the Shelburne dataset took precedence in performance metrics and Shelburne dataset execution times were slower than on the Taxi dataset for cross-sectional and deep-history queries on the Pi2 B+. Aggregation queries on the Thing setups were all in the sub-second range for TritanDB, in the tens of milliseconds range on the Gizmo2 and about a hundred milliseconds on the Pi2 B+.

Figure 5.13 shows the average execution time for each database taking the mean time from s^1 and s^2 setups for aggregation range queries. An average aggregator was used in the queries on a subset of columns and a $10^1 log_{10}$ scale was used to fit the range of execution times in the graph. TritanDB and Akumuli had the fastest execution times, within about 10 to 100 milliseconds, as they both store aggregates for blocks (e.g. sum, max,



[a]As the 100 queries are executed in sequence, the query translation overhead decreases to 0 to 2ms after the initial query

[b]OpenTSDB queries cannot be executed on the Taxi dataset because multiple duplicate timestamps are not supported

FIGURE 5.13: Aggregation Range Query Average Execution Time Mean of s^1 and s^2 on Each Database

min, count) within the in-memory block index and within the B+-tree structure respectively. TritanDB performs a fast lookup of the in-memory index and scanned the first and last blocks if required to obtain the aggregation on the remainder subset of points within those blocks and was 3.3 and 1.2 times faster than Akumuli for the Shelburne and Taxi datasets respectively. Native time-series databases like InfluxDB, TritanDB and Akumuli performed the best for aggregation queries as this is a key optimisation for time-series rollup and resampling operations. ElasticSearch also performed well for aggregation queries with indexing tuned specifically for time-series metrics, agreeing with independent benchmark results by Mathe et al. (2015).

Table 5.9 shows the average cross-sectional query execution time on the Pi2 B+ and the Gizmo2, while Table 5.10 shows the results for deep-history queries, and Table 5.11 shows the results for aggregation. These results are presented as such for completeness. TritanDB as shown in Table 5.8 was faster than all other databases on each type of query on the Thing setups of the Pi2 B+ and Gizmo2 as well.

Database	Pi2 B+ (10^2 s)		Gizmo2 (10^2 s)	
	$p_{shelburne}$	p_{taxi}	$g_{shelburne}$	g_{taxi}
InfluxDb	- [a]	-	-	-
Akumuli	-	-	2.46	2.04
MongoDb	-	-	28.35	41.22
OpenTSDB	- [b]	-	-	-
H2 SQL	21.65	11.49	10.45	3.75
Cassandra	26.44	25.28	6.82	6.15
ElasticSearch	-	-	8.74	5.54

[a]InfluxDB encounters out of memory errors for cross-sectional and deep-history queries on both setups.
[b]OpenTSDB on both setups runs out of memory incurring Java Heap Space errors on all 3 types of queries.

TABLE 5.9: Average Query Execution Time for Cross-sectional Queries on the Pi2 B+ and Gizmo2

Database	Pi2 B+ (10^2 s)		Gizmo2 (10^2 s)	
	$p_{shelburne}$	p_{taxi}	$g_{shelburne}$	g_{taxi}
InfluxDb	-	-	-	-
Akumuli	-	-	1.14	0.40
MongoDb	-	-	4.44	1.88
OpenTSDB	-	-	-	-
H2 SQL	13.52	5.03	3.04	1.25
Cassandra	4.77	1.30	1.10	0.32
ElasticSearch	-	-	8.64	3.53

TABLE 5.10: Average Query Execution Time for Deep-history Queries on the Pi2 B+ and Gizmo2

Database	Pi2 B+ (s)		Gizmo2 (s)	
	$p_{shelburne}$	p_{taxi}	$g_{shelburne}$	g_{taxi}
InfluxDb	7.10	1.76	2.75	0.62
Akumuli	-	-	0.37	0.33
MongoDb	-	-	96.45	57.90
OpenTSDB	-	-	-	-
H2 SQL	400.40	192.54	126.49	54.04
Cassandra	102.67	56.75	35.32	19.89
ElasticSearch	-	-	0.45	0.24

TABLE 5.11: Average Query Execution Time for Aggregation Queries on the Pi2 B+ and Gizmo2

5.5 Conclusions

The results in this chapter showed that TritanDB, as a Map-Match-Operate engine for the execution of graph queries on IoT data, provides better storage and query performance than a diverse set of state-of-the-art time-series, relational and NoSQL databases.

The novel design and implementation of TritanDB was made possible by an understanding of compression algorithms and data structures optimised for time-series IoT data that showed 'mechanical sympathy' to the physical representation of time-series data on storage media. Furthermore, the comparison of TritanDB with other databases showed that this storage and query performance improvement also applied across different hardware devices along a cloud-to-thing continuum.

As TritanDB showed in this chapter, IoT data can be stored and queried efficiently while at the same time maintaining a graph data model for the semantic interoperability of metadata. The next chapter (Chapter 6) introduces another dimension of stream processing on real-time flows of IoT time-series data with continuous streaming queries. Chapter 7 then describes how TritanDB can be used in personal data stores for IoT data (Section 7.1) and some optimisations of TritanDB for application workloads and querying involving time-series analytics (Section 7.2).

Chapter 6

Streaming Queries and A Fog Computing Infrastructure

"No bit is faster than one that is not sent; send fewer bits."

— *Ilya Grigorik*

Stankovic (2014) predicted that the IoT will have 'a very large number of real-time sensor data streams' and those streams of data 'will be used in many different ways'. While Map-Match-Operate with TritanDB represents progress in terms of the efficient querying of historical graph-model time-series IoT data stored on disk, this chapter seeks to continue the train of research by investigating efficient stream processing and interoperable graph-model querying on high velocity real-time streams of IoT data. In Section 6.1, the Map-Match-Operate abstraction is applied to continuous queries on streams of time-series data and the performance and scalability of the resulting engine implementation is evaluated. Section 6.2 then extends this streaming Map-Match-Operate engine to build a fog computing infrastructure for distributed stream processing on resource-constrained fog nodes. The performance within a smart city scenario, which involves many distributed devices, different stakeholders, diverse datasets and streams, and applications benefitting the society, environment and economy, is evaluated in Section 6.3. Section 6.4 then presents the implications of this fog computing research to the vision of the internet of the future and Section 6.5 concludes the chapter.

The work involving the efficient query translation from streaming SPARQL to the Event Processing Language (EPL) in this chapter was published in the Siow et al. (2016c) paper while the work on the fog computing infrastructure, Eywa, was published in the Siow et al. (2017) paper.

6.1 Stream Querying with Map-Match-Operate

In the previous chapters, Map-Match-Operate was applied to a graph data model and an underlying relational model, which was further improved with native time-series storage in TritanDB. Abadi et al. (2003) called this type of Database Management System (DBMS) a Human-Active DBMS-Passive (HADP) model, where the DBMS is a passive repository of historical data and humans initiate queries to retrieve results from the database. They went on to introduce a model for monitoring sensor streams where the role of the database is to actively process data and alert humans when abnormal activity is detected and called it a DBMS-Active Human-Passive (DAHP) model.

Arasu et al. (2006) formalised a language, the Continuous Query Language (CQL), to work on such a DAHP model, that allows queries to be issued once and continuously evaluated over real-time streams and relations. Stonebraker et al. (2005) qualified that integrating both stored relations and streaming data is an essential requirement of stream processing. RDF stream processing (RSP)[1] is the area of work that draws from the work on CQL and enables RDF graph data to be produced, transmitted and continuously queried as streams and relations (static RDF graphs).

Map-Match-Operate for stream processing, similar to Map-Match-Operate on a relational (HADP) database or with TritanDB, allows a graph query, for example an RSP query, to be executed on both a graph data model mapping and time-series data. Instead of storage in a relational database though, the time-series data is represented as an underlying stream of real-time data, where each point in each time-series is an event. Section 6.1.1 describes how the S2S Map-Match-Operate engine from Section 4.2.4 is extended for RSP, by translating continuous graph queries (Section 6.1.2), and is evaluated against a state-of-the-art RSP engine, CQELS, in Section 6.1.3, showing orders of magnitude performance gain.

6.1.1 Extending S2S for Stream Processing

S2S, which was introduced in Section 4.2.4, is an engine for Map-Match-Operate that translates SPARQL queries to SQL queries for execution on relational databases. For processing streams, the relational database is replaced by a Complex Event Processing (CEP) engine which ingests a stream of events. Continuous queries can be registered on the CEP engine that pushes results when they are detected. SPARQL is replaced by a continuous query language drafted by the W3C RSP community group[2], RSP-QL.

There is no standard for RSP languages and different engines like CQELS (Le-Phuoc et al., 2011), C-SPARQL (Barbieri et al., 2010) and $SPARQL_{stream}$ (Calbimonte et al.,

[1] https://www.w3.org/community/rsp/
[2] http://streamreasoning.github.io/RSP-QL/RSP_Requirements_Design_Document/

2010) each implemented different SPARQL-like syntaxes. Dell'Aglio et al. (2014) however, presented a formal unified model for RSP languages that captured a cover set of the operational semantics of each engine's extension to SPARQL for streaming. S2S for stream queries implements a syntax of RSP-QL that is derived from the W3C RSP design document based on this unified model. At the time of writing, this implementation within S2S is the only available RSP engine supporting RSP-QL.

To support continuous stream queries, the SPARQL 1.1 syntax is extended from two clauses, 'FROM' and 'GRAPH', in S2S's implementation of RSP-QL. Listing 6.1 shows the RSP-QL 'FROM' clause specified in Extended Backus Naur Form (EBNF) notation while Listing 6.2 shows the 'GRAPH' clause in EBNF.

```
From = 'FROM NAMED [WINDOW]' <ID> ['ON' <SourceIRI> '[RANGE' WindowSpec']'];
WindowSpec = 'BGP' Count | Time TimeUnit ['[SLIDE' Time TimeUnit']'];
TimeUnit = 'ms' | 's' | 'm' | 'h' | 'd';
```

Listing 6.1: SPARQL 'FROM' Clause EBNF Definition for RSP-QL in S2S

```
Graph = 'GRAPH' <ID> { TriplePattern } | 'WINDOW' <ID> { TriplePattern };
```

Listing 6.2: SPARQL 'GRAPH' Clause EBNF Definition for RSP-QL in S2S

The 'FROM' clause is extended in RSP-QL to support defining both relations, called named graphs, and streams, called named windows, with the omission or addition of the 'WINDOW' keyword. If a stream is defined, the 'SourceIRI' of the stream has to be specified. A certain range for this window of the stream should also be provided.

Three options are supported for this range called 'WindowSpec', a count-based Basic Graph Pattern (BGP) window, a time-based tumbling window and a time-based sliding window. These types of windows were described in detail by Chakravarthy and Jiang (2009), with the count-based window specifying the count of events fulfilling this particular BGP to keep in the window, and the time-based tumbling window specifying the size in terms of a time duration to be kept in the window. The time-based sliding window adds a 'SLIDE' parameter that indicates how often the window specified will be computed (the tumbling window has a slide size equal to the window length).

S2SML, described in Section 4.2.1.1, is supported in S2S for creating data model mappings so that the map step in Map-Match-Operate can be performed. Each mapping closure (Section 4.2.1.2) is associated with a particular IRI. Within each mapping closure, there can be multiple streams (identified by a stream identifier) or relations (specified by their table name). The match step then resolves BGPs within streaming queries registered with S2S, matching them with the related mapping closure.

The operate step in S2S then translates the streaming query in RSP-QL to an Event Processing Language (EPL) query which is registered with the underlying Esper event processing engine[3] that also connects via Java Database Connectivity (JDBC) to any

[3]http://www.espertech.com/products/esper.php

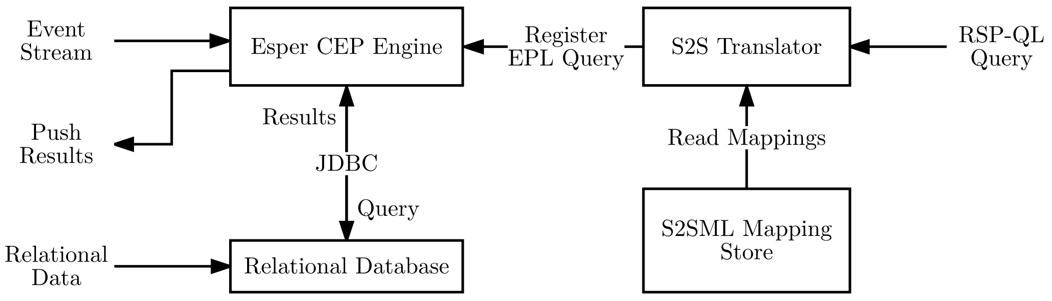


Figure 6.1: Continuous RSP-QL Queries on Streams with S2S and Esper

relational database sources. Esper is a mature, open-source Complex Event Processing (CEP) engine that has shown high throughput and low latency in benchmarks[4], in enterprise systems (San Miguel et al., 2015) and in comparison with state-of-the-art big data streaming systems like Spark Streaming when compared on a single node (Zaharia, 2016). Figure 6.1 shows the entire process from the registering of an RSP-QL query, its translation to an EPL query on Esper by S2S, that also takes in an event stream and is connected to any relational database required. Results for the query are pushed to the requesting client when they are calculated by the Esper engine.

6.1.2 RSP-QL Query Translation in S2S

The SPARQL query in Listing 4.5 can be written as an RSP-QL continuous query on a stream as follows in Listing 6.3. This query pushes results of rainfall observations with a depth of more than 10cm in the last hour.

```
PREFIX weather: <http://knoesis.wright.edu/ssw/ont/weather.owl#>.
PREFIX ssw: <http://knoesis.wright.edu/ssw/ont/sensor-observation.owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?value FROM NAMED WINDOW :obs ON <ssw:observations> [RANGE 1h]
WHERE {
  WINDOW :obs {
    ?obs a weather:RainfallObservation ;
      ssw:result [ ssw:floatValue ?value ].
      FILTER ( ?value > "10"^^xsd:float)
  }
}
```

Listing 6.3: RSP-QL Query Detecting Rainfall Observations >10cm within the Hour

The S2S Translator component in Figure 6.1 produces a query tree, in Figure 6.2, from the RSP-QL in Listing 6.3, that consists of a sequence of BGP match, filter, window and project operators. There is an additional 'WINDOW' operator as compared to the query tree produced from the similar SPARQL query on historical data in Figure 4.5.

[4]http://www.espertech.com/esper/performance.php

$$\begin{tikzcd}
\textit{Project}, \ \Pi_{\text{value}} \arrow[d, no head] \\
\textit{Window}, \ W_{\text{:obs}} \arrow[d, no head] \\
\textit{Filter}, \ \sigma_{>10} \arrow[d, no head] \\
G_{rain}
\end{tikzcd}$$

FIGURE 6.2: Query Tree of Operators for Listing 6.3's Rainfall Query

The process of Map-Match-Operate is similar to that described in Chapter 4 and the G_{rain} operator matches the BGP specified in RSP-QL with the corresponding mapping closure within the S2SML mapping store. The operate step runs through the set of operators in the query tree but produces an EPL query for Esper, shown in Listing 6.4, instead of an SQL query for H2, whereby the differences are merely syntactical.

```
SELECT rainfall FROM station.win:time(1 hour) WHERE rainfall > 10
```

LISTING 6.4: EPL Query Translation of Listing 6.3's Rainfall Query

Queries in RSP-QL and the corresponding S2S EPL translation from SRBench and the Smart Home Analytics Benchmark are published online[5,6]. A full set of definitions for each SPARQL algebra operator and its operate step query translation to EPL are presented in Appendix C for completeness.

6.1.3 Evaluating the Stream Processing Performance of S2S

The same IoT benchmarks in Chapter 4, SRBench (Zhang et al., 2012) and the Smart Home Analytics Benchmark (Appendix B), within the same experimental setup were used to evaluate S2S for streams. The experimental setup details were previously described in Section 4.3.3. S2S for streams was compared against CQELS (Le-Phuoc et al., 2011), a state-of-the-art native RSP engine. Since, these benchmarks compared push-based results reporting, CQELS was the only similar RSP engine to S2S using Esper. As Dell'Aglio et al. (2013) stated in their paper on the correctness in RSP benchmark systems, CQELS reports as soon as the content of the window changes, a push-based results reporting method, while other engines like C-SPARQL only support results delivery when the window closes and there is content in the window.

Each query from both benchmarks were registered on both engines separately using time-based tumbling windows. The time taken from the insertion of an event to the return of a pushed result, the latency of stream query processing, for both engines was measured with 1s delays between sending events, simulating the periodic sampling of observations every second from a set of sensors. An average over a 100 of these results

[5]https://github.com/eugenesiow/sparql2stream/wiki/Q01
[6]https://github.com/eugenesiow/ldanalytics-pismarthome/wiki

was taken. For S2S, the one-off translation time at the start (ranging from 16ms to 32ms) was included in the sum during the average calculation.

Table 6.1 shows the results, for both benchmarks, of the average time taken to evaluate a query from the insertion of an event to the return of a push-based result from S2S, t_{S2S}, and CQELS, t_{CQELS}, with 1s delays in between. Query 6[7] of SRBench, a query to detect weather stations with low visibility by getting the union of visibility, rainfall and snowfall observations, was omitted due to CQELS not supporting the '`UNION`' operator. For all other queries, the S2S engine showed over two orders of magnitude performance improvement over CQELS. Queries 4, 5 and 9 that involved joining subgraphs of observations (e.g. *WindSpeed* and *WindDirection* in Query 9) and aggregations showed larger performance gains. This was similar to the results on historical data in S2S previously reported in Section 4.4.2. It was noted, that although CQELS returned valid results for these queries, they contained an increasing number of duplicates, perhaps due to an issue with the adaptive caching of results within the window, which caused a significant slowdown over a 100 pushes, which is an undesirable behaviour over time. The source code for the experiments are published on Github[8],[9].

A point of note is that the Java function '`System.nanoTime()`' used to measure latency in the benchmarks, relied on a system counter within the CPU which is usually accurate to about 1 microsecond but in the worst case could have only millisecond accuracy. Hence, it was necessary to average the results over at least a 100 measurements.

SR_{Bench}	t_{S2S}	t_{CQELS}	Ratio
1	0.47	138	1 : 294
2	0.46	119	1 : 261
3	0.66	202	1 : 306
4	0.67	$(1.86 \cdot 10^5)$	$1 : (2.77 \cdot 10^5)$
5	0.63	$(1.48 \cdot 10^6)$	$1 : (3.24 \cdot 10^6)$
6	-	-	-
7	0.66	2885	1 : 5245
8	0.67	282	1 : 426
9	0.67	$(1.88 \cdot 10^5)$	$1 : (2.8 \cdot 10^5)$
10	0.73	72	1 : 98

Smarthome	t_{S2S}	t_{CQELS}	Ratio
1	0.64	125	1 : 196
2	0.77	129	1 : 167
3	0.81	-	-
4	3.78	-	-

Table 6.1: Average SRBench and Smart Home Analytics Benchmark Query Run Times (in ms)

To verify that S2S was able to answer SRBench queries close to the rate they were sent, even at high velocity, the delay between insertions was reduced first from 1s to 1ms and then to 0.1ms to produce rates of 1 row/ms and 10 rows/ms respectively. Table 6.2 shows a summary of the average latency, the time from insertion to when query results were returned, of each query for both rates (in ms). It can be observed that the

[7]https://github.com/eugenesiow/sparql2stream/wiki/Q06
[8]https://github.com/eugenesiow/cqels
[9]https://github.com/eugenesiow/sparql2stream

Rate	Latency (in ms) of SRBench Query Number								
(rows/ms)	1	2	3	4	5	7	8	9	10
1	1.300	1.374	1.279	1.303	1.2561	1.268	1.267	1.295	1.255
10	0.155	0.159	0.143	0.161	0.1291	0.137	0.141	0.155	0.129

TABLE 6.2: Average Latency (in ms) at Different Rates for Each SRBench Query

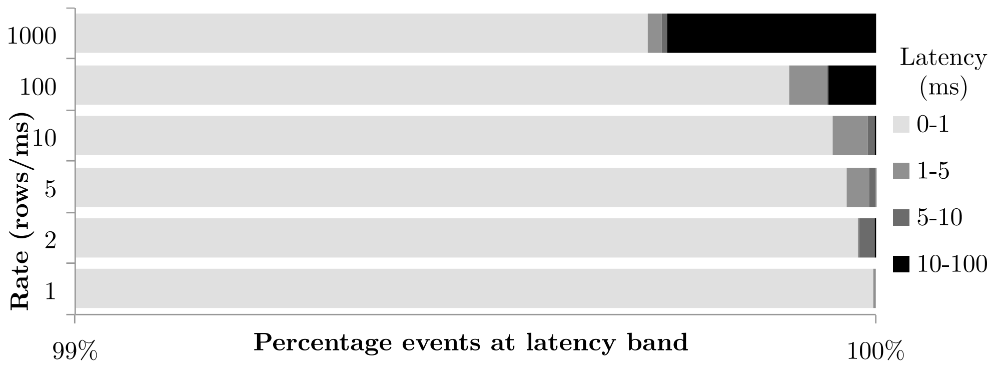


FIGURE 6.3: Percentage Latency at Various Rates for SRBench Query 1 (Focused between 99-100%)

average latency was slightly higher than the inverse of the rate. The underlying stream engine, Esper, maintains context partition states consisting of aggregation values, partial pattern matches and data windows. At high rates, the engine introduces blocking to lock and protect context partition states which results in a slightly higher latency.

To ensure that the impact of this overhead was minimal even at high velocities, a comparison method used by Calbimonte et al. (2012) to compare RSP engines was employed to group messages by latency ranges. Figure 6.3 shows that the effect of Esper's blocking on S2S latency was very minimal as the percentage of high latency query processing time, of more than 1ms, was less than 0.3% (note that the x-axis is from 99% to 100%) even at the highest velocity with a rate of 1000 rows/ms. The longest latency times were also within 100ms and these were only observed at the high rates of 100 and 1000 rows/ms, coinciding with Garbage Collection (GC) pauses of the Java Virtual Machine as a result of a larger number of 'live' objects[10] in memory and greater memory fragmentation. SRBench query 1[11] which involved getting the rainfall observed within an hourly window was used for the latency banding measurement.

Another test was performed to determine the maximum size of data that S2S could process in-memory on the experimental setup of a lightweight IoT device, a Raspberry Pi 2 B+ (RPi) with 1GB of memory. A query with a long *TUMBLING* window, SRBench query 8[12], which measures the daily maximum and minimum air temperature observed by the sensor at a given location, was used. The engine ran out of memory after inserting

[10]https://www.dynatrace.com/resources/ebooks/javabook/impact-of-garbage-collection-on-performance/
[11]https://github.com/eugenesiow/sparql2stream/wiki/Q01
[12]https://github.com/eugenesiow/sparql2stream/wiki/Q08

33.5 million rows, within the window of one day. Each row produced an equivalent of 75 triples at the weather station measured, where each individual observation, that corresponded to a column within the row, produced about 10 triples. By projection, an RDF dataset size of 2.5 billion triples could theoretically fit in the RPi's memory for a single continuous query.

Queries 1 and 2 of the Smart Home Analytics Benchmark, presented in Appendix B, also corroborated the two orders of magnitude performance advantage of S2S over CQELS as shown in Table 6.1. Query 1 measured an average of the internal or external temperature within the smart home over the last hour while query 2 measured the maximum and minimum temperatures over a day. Queries 3 and 4 could not be run on CQELS due to the '`FILTER`' operator throwing an error. Query 4 which detected unattended electrical usage based on metering and motion data involved joining both streams and performing an aggregation for average power. This query saw the average latency of S2S increase, though still stay under 4ms. The latency for this query was measured from the insertion time of the last event involved that produced a result and hence initiated a push, to that of receiving the push result.

Hence, results from the two benchmarks both show that S2S performs better than CQELS for processing IoT time-series data streams with continuous queries. Rows of data ingested by S2S are a more compact representation than sets of triples forming graphs that CQELS processes, which benefits not only push-based results reporting latency but also the maximum size of data that can be streamed in a window. Map-Match-Operate allows S2S to serve as an RSP engine by maintaining the balance of interoperability, with a flexible graph model representation, and efficiency, with query translation.

6.2 Eywa: An Infrastructure for Fog Computing

In the science-fiction motion picture Avatar[13], Eywa was the name of the biological internet on the planet Pandora, made up of trees which were distributed over the surface of the planet. The trees stored memories and processed information. The flora and fauna of the planet then formed an ubiquitous sensor network feeding Eywa.

The Internet of Things (IoT) is growing to become a similar ubiquitous network of sensors and actuators for the planet. Fog computing is an emerging technology which seeks to bridge a gap for the IoT, like the fictional Eywa on Pandora, between the ground, where sensors and actuators are deployed (collect and act on data), and the cloud, where larger amounts of resources for processing and storing data can be provisioned dynamically. Application themes like the smart city from Section 2.1.2 experience the challenges

[13]http://www.avatarmovie.com/

of wide IoT data stream distribution and real-time application query workloads which fit with the concept of Eywa.

The purpose and contribution of this section is to introduce and formally define Eywa, an infrastructural fog computing framework,

- where stream processing can be performed on resource-constrained lightweight fog computing nodes,
- where the processing workload can be distributed among nodes utilising a novel inverse-pub-sub *control plane* and
- each node can maintain independence and control over access, resources, security and privacy.

An Eywa network, that forms the basis for the Eywa fog computing infrastructure, is explained as follows. An Eywa network, ε, consists of a set of nodes, N and connections, C, such that $\varepsilon = (N, C)$. Each node, $n \in N$, can be a source, s, client, τ, or broker, b, node. Each connection, $c \in C$ exists uniquely between two nodes such that $C \subseteq N \times N$.

A source node, $s \in S$, is a node that produces a set of time-series streams, Γ. A client node, $\tau \in T$, is a node with a set of queries Q expecting a set of corresponding results, R. A broker node, $b \in B$, establishes a connection, c, for new source and client nodes to enter the network.

As *source* and *client* nodes form up in an Eywa network, *broker* nodes 1) provide a point of entry for new nodes into the network, 2) do not store or process but forward data, 3) consume minimal resources and 4) employ redundancy through multiple separate instances within the network so as not to become single points of failure. Hence, facilitating both the formation and data flow in Eywa.

The utility of this Eywa network for processing streams is to facilitate 1) *stream query delivery* to relevant nodes, 2) *distributed processing* and 3) *results delivery* to the requesting nodes. To this end, Section 6.2.1 explains query delivery, Section 6.2.2 describes distributed processing and Section 6.2.3 defines results delivery.

6.2.1 Stream Query Delivery by Inverse-Publish-Subscribe

Once an Eywa network has been formed, stream processing can take place as clients, τ, issue stream queries, Q. Traditionally, source nodes, s, publish data while client nodes subscribe to data. In wireless sensor networks literature, these are sometimes referred to as the source and sink respectively (Ciciriello et al., 2007), where both many-to-one (Mainwaring et al., 2002; Intanagonwiwat et al., 2003) and many-to-many (Kim et al., 2003; Akyildiz and Kasimoglu, 2004) paradigms have been studied in the data plane, a fog computing term borrowed from routing literature (Yang et al., 2004). However, in Eywa, it is desirable for clients to collaborate with the sources in a many-to-many fashion to share the workload, hence, an inverse-publish-subscribe mechanism for streaming query delivery in the control plane is proposed.

Each *source* node subscribes to a topic for each of its streams. *Client* nodes then publish queries to the relevant topics. Internationalised Resource Identifiers (IRIs), that work well for the web, are used to provide a means of uniquely identifying and exchanging topics. Definition 6.1 formally describes this mechanism.

Definition 6.1 (Inverse-publish-subscribe)**.** Given the set of topics, M, a source node, s, subscribes to a topic, $\mu \in M$ for each stream within Γ to form $\bigcup_{\mu \in M_\Gamma} sub(\mu)$, where M_Γ is the set of all topics of s and $sub(\mu)$ is a function that produces a subscription to μ. For each query, $q \in Q$, from a client node, τ, a distribution function, α, builds a set of query-topic pairs $\alpha(q) = (Q_\mu, M_q)$ where Q_μ is the set of all sub-queries in q, each referencing a particular topic μ and M_q is the set of all topics referenced in q. Each sub-query, $q_\mu \in Q_\mu$, is published to its particular topic, $\mu_q \in M_q$, by the publish function of query x to topic y, $pub(x, y)$, so all publications from q form $\bigcup pub(q_\mu, \mu_q)$.

6.2.2 Distributed Processing

Source nodes receive the queries, perform part of the processing and deliver the results to *clients* that process the results. This forms the axis of client-source collaboration and distributing processing workload as defined in Definition 6.2. *Source* nodes control their own resources and response to queries. Quality of service is not in the scope of this work and by default is considered best effort within Eywa, however, service-level agreements, trustless networks or consensus protocols can be deployed on top of Eywa which are discussed in the Social Web of Things application in Section 7.3.

Definition 6.2 (Distributed Source Node Processing)**.** A source node, s, receives a query, q, for a topic, μ, and converts it into a work function, ω, with a conversion function, $\lambda(q) \to \omega$. The work function, ω, is applied to the corresponding stream for topic μ, γ_μ, where $\gamma_\mu \in \Gamma$, so that $\omega(\gamma_\mu) = \gamma_r$ and the resulting stream, γ_r, is pushed to the requesting client, τ.

6.2.3 Push Results Delivery and Sequence Diagram

Client nodes receive result streams via a direct push from *source* nodes. Operations involving multiple streams, like aggregations, are performed on the clients and results of the queries are published to topics for applications. Definition 6.3 details the process of results delivery and query output. Unlike in a Wireless Sensor Network (WSN), prolonged operation and the network's survival due to power-constrained nodes take lower precedence than performance and utility; hence, the underlying routing and transport networks are not of concern and decoupled.

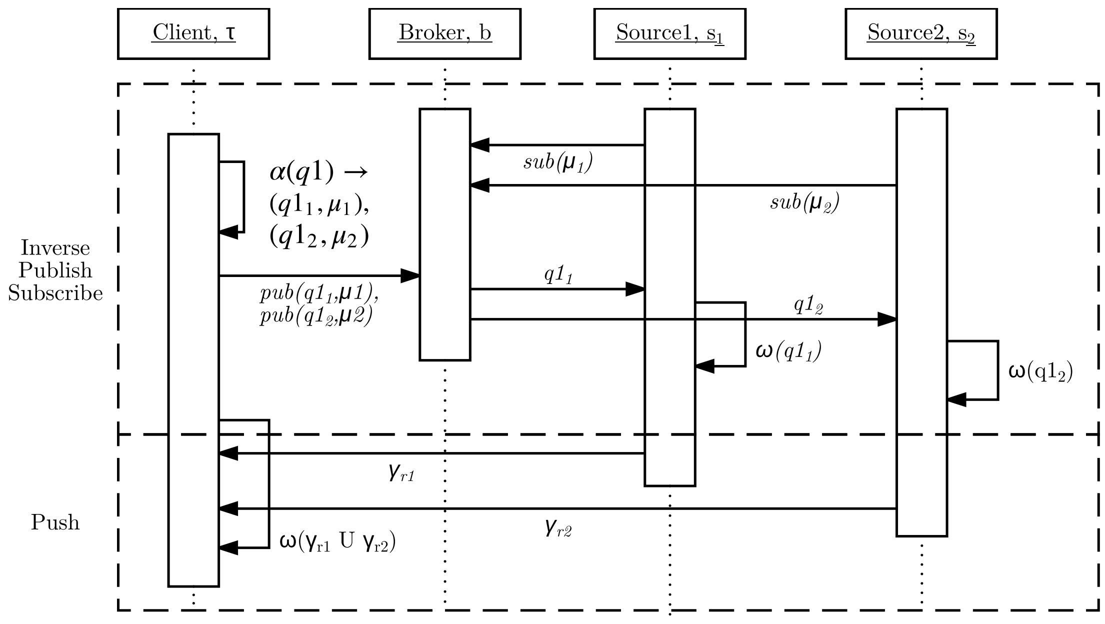


Figure 6.4: Sequence Diagram of Stream Query Processing in Eywa

Definition 6.3 (Push Results Delivery)**.** A client node, τ, receives a set of result streams, Γ_r, by push delivery. For each query, $q \in Q$, in the set of queries for that client, a work function, ω_τ, is produced by $\lambda_\tau(q) \rightarrow \omega_\tau$ and executed on $\bigcup \gamma_r$, where γ_r are all the result streams corresponding to the query, q. The result, γ_q, from $\omega_\tau(\bigcup \gamma_r) = \gamma_q$ is published to a client results topic, μ_τ.

Figure 6.4 shows the full sequence diagram of the query processing process beginning from the *source* nodes subscribing to topic URIs, receiving queries when published by *client* nodes, distributed processing and result delivery.

Source1, s_1, produces a stream named, μ_1, and subscribes to the corresponding topic, μ_1, through broker, b. Source2, s_2, does the same for its μ_2 stream. A query, $q1$, is registered with client, τ, which executes the distribution function, $\alpha()$, which forms pairs of the two sub-queries of $q1$ and their relevant topics. Each sub-query, $q1_1$ and $q1_2$, are published to topics $\mu 1$ and $\mu 2$ respectively. $s1$ and $s2$ receive the sub-queries and register them with a conversion function, $\lambda()$. The work function, $\omega()$, continuously processes the streams produced and registered queries and pushes the results, when available, to τ, which combines the result streams through $\omega()$. Table 6.3 shows a glossary of symbols used and their corresponding definitions for reference.

	Description	Definition(s)
τ	Client node	6.1
μ	Topic (Uniformed Resource Identifier)	6.1
γ, Γ	Stream, Set of streams	6.1, 6.2, 6.3
$\alpha()$	Distribution function, build query-topic pairs	6.2
$\lambda()$	Conversion function, converts query to work function	6.2, 6.3
$\omega()$	Work function, applied to streams	6.2, 6.3

TABLE 6.3: Glossary of Symbols used in the Eywa Definition

6.2.4 S2S RDF Stream Processing (RSP) in Eywa

The work function, $\omega()$, that is applied to streams in Eywa on source nodes during the distributed processing phase, can be implemented with an RDF Stream Processing (RSP) engine. Following the Map-Match-Operate philosophy in Chapter 4 and implementation with S2S for streams in Section 6.1.1, Eywa utilises S2S as an RSP engine and together they are optimised to:

1. Ingest *observation time-series data* that exhibits flat and wide characteristics as succinctly as possible.
2. Abstract the small set of *device metadata* with a flexible graph model to store in-memory as S2SML data model mappings for more semantic interoperability.
3. Distribute parts of the query to be applied on streams on an Eywa source node to share the workload and possibly reduce the bandwidth required by transmitting filtered results rather than raw data (predicate pushdown).

When an RSP-QL query is registered with a client, sub-queries are distributed to the relevant source streams as described previously (Section 6.2.1), translated with Map-Match-Operate and registered as a continuous query on each source node involved. Predicate pushdown is a term used in database literature (Reinwald et al., 1999) and presents certain advantages especially in distributed systems like Eywa. The concept is as such, when a query is issued from a client node in one place to run against a lot of data produced by a set of source nodes in another place, a lot of network traffic could be spawned in transmitting raw data, which could be slow and costly. When certain parts of the query can be 'pushed down' to where the data is actually stored, filtering out most of the data, then network traffic can be greatly reduced.

Examples from CityBench (Ali et al., 2015), a published benchmark of smart city IoT data, are used to elaborate on the translation process. CityBench consists of smart city streams and metadata gathered from IoT sensors deployed within the city of Aarhus in Denmark from February 2014 to November 2015.

Timestamp	Humidity	Temperature	Wind Speed
2014-08-01T00:00:00	56.0	18.0	7.4

Table 6.4: CityBench: Observation from the AarhusWeatherData0 Stream

Table 6.4 shows a weather observation from a CityBench stream. This is a stream of actual flat and wide time-series observation data and consists of humidity, temperature and wind speed readings measured at a particular timestamp. A source node with this stream subscribes to the relevant IRI of the form, '`http://...#AarhusWeatherData0`'.

Listing 6.5 is the corresponding S2SML data model mapping that stores the sensor and observation metadata of the weather data stream. It also contains bindings to fields within the underlying stream, like '`AarhusWeatherData0.tempm`'.

```
@prefix ssn : <http://purl.oclc.org/NET/ssnx/ssn#>
@prefix sao : <http://purl.oclc.org/NET/sao/>
@prefix ct : <http://www.insight-centre.org/citytraffic#>
@prefix ns : <http://www.insight-centre.org/dataset/SampleEventService#>
@prefix s2s : <http://iot.soton.ac.uk/s2s/s2sml#>
ns:obsTemp{AarhusWeatherData0._uuid} a ssn:Observation;
  ssn:observedProperty ns:Property-1;
  sao:hasValue "AarhusWeatherData0.tempm"^^s2s:literalMap;
  ssn:observedBy ns:AarhusWeatherData0.
ns:Property-1 a ct:Temperature.
ns:obsHum{AarhusWeatherData0._uuid} a ssn:Observation;
  ssn:observedProperty ns:Property-2;
  sao:hasValue "AarhusWeatherData0.hum"^^s2s:literalMap;
  ssn:observedBy ns:AarhusWeatherData0.
ns:Property-2 a ct:Humidity.
ns:obsWS{AarhusWeatherData0._uuid} a ssn:Observation;
  ssn:observedProperty ns:Property-3;
  sao:hasValue "AarhusWeatherData0.wspdm"^^s2s:literalMap;
  ssn:observedBy ns:AarhusWeatherData0.
ns:Property-3 a ct:WindSpeed.
```

Listing 6.5: CityBench AarhusWeatherData0 S2SML Mapping (Abbreviated)

This mapping references various ontologies like the Semantic Sensor Network (SSN) Ontology[14] and City Traffic Ontology, providing a common way of describing sensor data, increasing interoperability. A formal definition of the S2SML mapping language is covered in Section 4.2.1.1. Similarly, a mapping is available for the stream of traffic data, with '`avgSpeed`' and '`congestionLevel`' fields, at locations in the city and also other smart city streams like pollution data available from the RSP-QL fork[15] of S2S.

Listing 6.6 shows Query 2 of CityBench expressed in RSP-QL syntax that checks the traffic congestion level and weather conditions on each road of a planned journey. When registered, it processes both traffic and weather streams for observations of traffic congestion level and weather (e.g. temperature), from a particular stretch of road, for the

[14]http://www.w3.org/TR/vocab-ssn/
[15]https://github.com/eugenesiow/sparql2stream/tree/RSP-QL

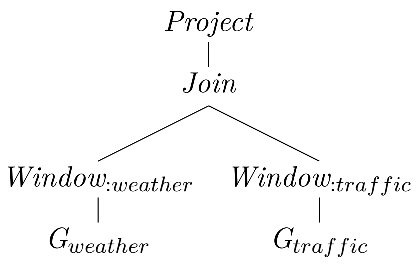


FIGURE 6.5: Query Tree of CityBench Query 2 from Listing 6.6

last 3 seconds. The query is abbreviated for better readability and the weather window actually queries 'v1' to 'v3' for temperature, humidity and wind speed respectively. The graph query provides for interoperability by including metadata in a flexible way while the values 'v1' to 'v4' are efficiently retrieved from fields within the flat streams. 'v1' to 'v3' can be retrieved all at once from the same weather stream event.

```
SELECT ?v1 ?v2 ?v3 ?v4
FROM NAMED WINDOW :traffic ON <http://...#AarhusTrafficData158505> [RANGE 3s]
FROM NAMED WINDOW :weather ON <http://...#AarhusWeatherData0> [RANGE 3s]
WHERE {
  WINDOW :weather {
    ?obId1 a ssn:Observation;
      ssn:observedProperty ?p1;
      sao:hasValue ?v1;
      ssn:observedBy ns:AarhusWeatherData0.
    ?p1 a ct:Temperature.
    #obsId2 for Humidity and obsId3 for WindSpeed abbreviated...  }
  WINDOW :traffic {
    ?obId4 a ssn:Observation;
      ssn:observedProperty ?p4;
      sao:hasValue ?v4;
      ssn:observedBy ns:AarhusTrafficData158505.
    ?p4 a ct:CongestionLevel. }
}
```

LISTING 6.6: CityBench Query 2: Finding the Traffic Congestion Level and Weather Conditions of a Planned Journey (Abbreviated)

A walkthrough of the query translation process for this query is in Section 6.2.5, while the distribution by Eywa's inverse-pub-sub, processing and return to the client for further processing is described in Section 6.2.6. The entire architecture of Eywa's RSP engine is then summarised in Section 6.2.7.

6.2.5 Eywa's RSP-QL Query Translation of CityBench

Eywa uses the S2S engine for stream query translation that was introduced in Section 6.1.2. Firstly, the RSP-QL query was parsed into algebra as shown in Figure 6.5.

The algebra query tree was traversed from leaf to root. The Basic Graph Pattern (BGP), $G_{weather}$, which compromises the portion of the query shown in Listing 6.7 was matched against the *AarhusWeatherData0* S2SML mapping from Listing 6.5 in the match step of Map-Match-Operate that Eywa-RSP's S2S engine follows.

```
?obId1 a ssn:Observation;
  ssn:observedProperty ?p1;
  sao:hasValue ?v1;
  ssn:observedBy ns:AarhusWeatherData0.
?p1 a ct:Temperature. #(...humidity and wind speed parts)
```

Listing 6.7: Basic Graph Pattern, $G_{weather}$, from CityBench Query 2 (Listing 6.6)

The result of the match step was $\mathbb{B}^{weather}_{match}$ that contained bindings like that from variable '`v1`' from $G_{weather}$ to the field '`tempm`' in the stream *'AarhusWeatherData0'*. Similarly, from $G_{traffic}$, $\mathbb{B}^{traffic}_{match}$ contained the binding of variable '`v4`' to '`congestionLevel`' in the stream *'AarhusTrafficData158505'*. The results were passed to the *join* operator.

Since no variables overlapped from the *weather* and *traffic* windows, the result sets were passed upwards, unchanged, to the *project* operator. At the *project* node, variables '`v1`' to '`v4`' were projected. Any variable renaming or aliases (none in this example) would be taken care of in the projection operator as well. The simplified stream query produced, expressed in Event Processing Language (EPL), is shown in Listing 6.8.

```
SELECT AarhusWeatherData0.tempm AS v1 ,
  AarhusWeatherData0.hum AS v2 ,
  AarhusWeatherData0.wspdm AS v3 ,
  AarhusTrafficData158505.congestionLevel AS v4
FROM AarhusWeatherData0.win:time(3 sec) ,
  AarhusTrafficData158505.win:time(3 sec)
```

Listing 6.8: CityBench Query 2 Translated to Event Processing Language (EPL)

CityBench Query 5 in Listing 6.9 which discovers the traffic congestion level on the road where a given cultural event is happening has algebra as shown in Figure 6.6. $Window_{:traffic}$ produced a similar result set as in Query 2. However, $Graph_{sensor}$ and $Graph_{cultural}$ reference relations, historical data sources, instead of streams, hence, SQL instead of EPL queries were produced for each by the S2S translation engine. The translated SQL query syntax from $Graph_{cultural}$ is shown in Listing 6.10.

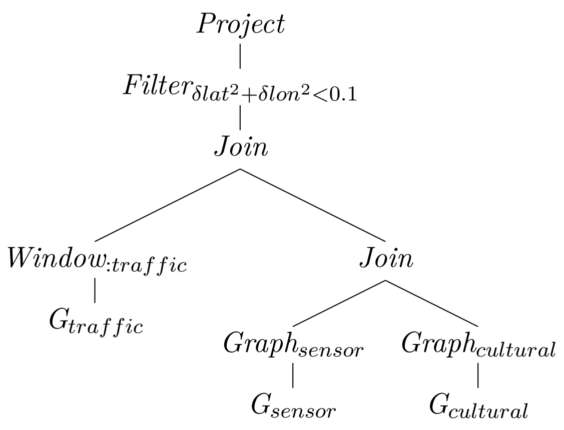


FIGURE 6.6: Query Tree of CityBench Query 5: Traffic Congestion Level at a Cultural Event

```
SELECT ?title ?lat1 ?lon1 ?lat2 ?lon2 ?v2
FROM NAMED WINDOW :traffic ON <http://...#AarhusTrafficData158505> [RANGE 3s]
FROM NAMED <http://...#AarhusCulturalEvents>
FROM NAMED <http://...#SensorRepository>
WHERE {
  GRAPH ns:SensorRepository {
    ?p2 a ct:CongestionLevel;
      ssn:isPropertyOf ?foi2.
    ?foi2 ct:hasStartLatitude ?lat2;
      ct:hasStartLongitude ?lon2.
  }
  GRAPH ns:AarhusCulturalEvents {
    ?evtId a ssn:Observation;
      sao:value ?title;
      ssn:featureOfInterest ?foi.
    ?foi ct:hasFirstNode ?node.
    ?node ct:hasLatitude ?lat1;
      ct:hasLongitude ?lon1.
  }
  WINDOW :traffic {
    ?obId2 a ssn:Observation;
      ssn:observedProperty ?p2;
      sao:hasValue ?v2;
      ssn:observedBy ns:AarhusTrafficData158505.
  }
  FILTER (((?lat2-?lat1)*(?lat2-?lat1)+(?lon2-?lon1)*(?lon2-?lon1))<0.1)
}
```

LISTING 6.9: CityBench Query 5: Traffic Congestion Level on the Road where a Given Cultural Event is Happening

```
SELECT title, lat, lon FROM AarhusCulturalEvents
```

LISTING 6.10: $Graph_{cultural}$ Portion of CityBench Query 5 Translated to SQL

At the first *join* operator from the bottom of the tree, the SQL queries were integrated into the ‘`FROM`’ clause of the EPL statement produced in the S2S translation as shown in Listing 6.11.

```
FROM sql:AarhusCulturalEvents [ ‘SELECT title, lat, lon
  FROM AarhusCulturalEvents’ ] AS AarhusCulturalEsvents ,
sql:SensorRepository [ ‘SELECT lat, lon FROM SensorRepository’ ] AS
    SensorRepository
```

Listing 6.11: ‘FROM’ Clause of Translated EPL Query from CityBench Query 5 (Listing 6.9) after $Graph_{sensor}$-$Graph_{cultural}$ Join

At the next *join* operator, as there was a variable ‘p2’ present in both inbound nodes, a join was performed between the metadata in the model from $Window_{:traffic}$ and the previous *join*. The modified translation to SQL within the ‘FROM’ clause is shown in Listing 6.12.

```
... [ ‘SELECT lat, lon FROM SensorRepository
  WHERE propId=\‘Property-b9f9...\’’ ] AS SensorRepository,
    AarhusTrafficData158505.win:time(3 sec)
```

Listing 6.12: Abbreviated ‘FROM’ Clause of Translated EPL Query from CityBench Query 5 (Listing 6.9) after Join Operator on $Window_{:traffic}$

Finally, a filter on the addition of the square delta of latitude and longitude constrains the traffic sensor and cultural event locations to within 0.1 units of area[16]. All CityBench queries and corresponding S2S translations have been published on the benchmark wiki[17].

6.2.6 Eywa's RSP Query Distribution

Query distribution is the process whereby part of a query workload, ω_τ, is distributed from the client (where the query is registered and the results are expected), to the source node (where the data is produced or stored). The workload is distributed within the Eywa network and a simple form of predicate pushdown is achieved.

For example, for Query 2 in CityBench (Listing 6.6), at the *project* operator at the root of the algebra tree, the S2S engine in Eywa tracks that *hum*, *tempm* and *wspdm* are the fields required from $Window_{:weather}$ while only *congestionLevel* is required from $Window_{:traffic}$. Hence, the projection of these fields is the work function, ω_τ, pushed to each source node producing the traffic and weather streams. These relationships are tracked as a projection tree for the streams as shown in Figure 6.7 for Query 2.

Following RSP Query Translation (Section 6.2.5), the resulting simplified, translated EPL query is registered on the client. The projection tree is distributed through Eywa's inverse-pub-sub mechanism to the subscribing source nodes of the IRIs of $Window_{:weather}$ and $Window_{:traffic}$. Hence, each source node pushes only an effective subset of each stream, γ_q, with only the projected fields to the client which combines and does any

[16]https://github.com/eugenesiow/Benchmark/wiki/Q05
[17]https://github.com/eugenesiow/Benchmark/wiki/

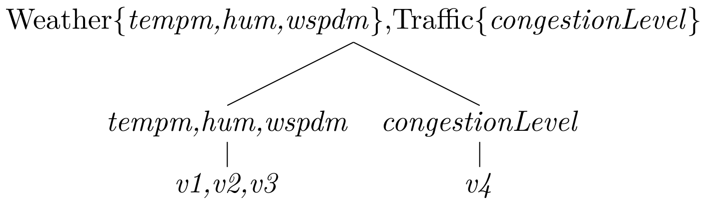


FIGURE 6.7: Projection Tree for Query 2 Streams

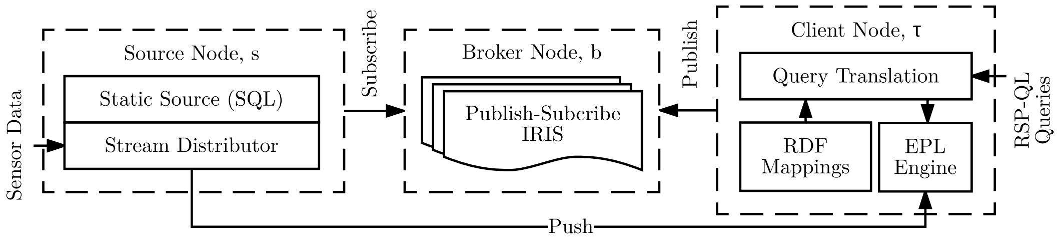


FIGURE 6.8: Architecture of Eywa's RSP Fog Computing Framework

further processing on the received streams. For relations, static sources of historical data, the work function, ω_τ, also comprises SQL queries which are automatically distributed to the source nodes by the underlying Esper engine in S2S through Java Database Connectivity (JDBC) connections. In Query 5 of CityBench as shown in Listing 6.11, an SQL query is executed on the static *SensorRepository* source node.

6.2.7 Eywa's RSP Architecture

Figure 6.8 shows the architecture of Eywa's RSP implementation in this thesis consisting of *client, source* and *broker* nodes. The *brokers* use lightweight ZeroMQ sockets which use a binary transmission protocol that has minimal overhead to support publish-subscribe IRI topics. *Client* nodes receive queries in RSP-QL and projected streams via push from *source* nodes. They process complex events in queries using an EPL engine, Esper, that registers translated queries with data model mappings. *Source* nodes produce streams of time-series IoT data or store static data, e.g. sensor location data, cultural event data. A stream distributor component pushes projected streams, γ_q, to client nodes as required. Static queries utilise JDBC connections from client to source nodes.

The current design of Eywa makes several assumptions. Firstly, the source, broker and client nodes are assumed to be co-operative. At least one broker node is active within Eywa and known by the source and client nodes. Delivery via push from the source to client nodes is considered best effort.

6.3 Evaluation of Eywa with CityBench

The performance and scalability of the Eywa fog computing infrastructure was evaluated on smart city vehicle traffic, parking, cultural events, weather and pollution IoT data streams with CityBench (Ali et al., 2015). As explained earlier in Section 6.2, the smart city scenario, which envisions affordable fog devices owned by people and organisations that are distributed throughout the city, its buildings and infrastructure, is a particular pertinent use case for Eywa. The benchmark featured continuous stream queries that powered smart city applications, for example, a parking space finder and an admin console. Eywa and the underlying S2S RSP engine were compared against CQELS (Le-Phuoc et al., 2011) and C-SPARQL (Barbieri et al., 2010) RSP engines.

The streaming and static source nodes, client nodes and broker nodes within the experimental setup were hosted on resource-constrained lightweight computers, Raspberry Pi (RPi) 2 Model B+s, which were widely available and had good community support. Each RPi had 1GB RAM, a 900MHz quad-core ARM Cortex-A7 CPU and used Class 10 SD Cards. Ethernet connections were used between the nodes for reliable transport during the experiment and each node's clock was synchronised using the Network Time Protocol (NTP).

For each query, experiments were performed for a varying amount of concurrent queries and data streams. Both the latency and memory consumption were measured for each engine. Tests were run for 15 minutes each, based on the methodology from the benchmark paper (Ali et al., 2015), and averaged over 3 runs. The goal was to measure the performance, in terms of the latency of queries, and scalability, which was shown by the memory consumption while varying the number of data streams and concurrent queries. Additionally, to measure the benefits in scalability of Eywa's fog computing infrastructure, Eywa-RSP with S2S was tested and compared with client, source and broker on a single node against when they were distributed across multiple nodes with workload distribution. A client and broker were deployed on one node, while two source and broker nodes producing different streams were deployed within the network.

6.3.1 Latency Evaluation Results

The latency of a query refers to the average time between when a stream observation arrived and when the results from the query output were generated.

Figure 6.9 shows the latency over time of each of the engines for Query 1[18] in CityBench which measured the traffic congestion level on two roads. Eywa-RSP had the lowest latency and best query performance. The average latency over time for queries in CityBench, measurements were made at one minute intervals over the total time of 15

[18]https://github.com/eugenesiow/Benchmark/wiki/Q01

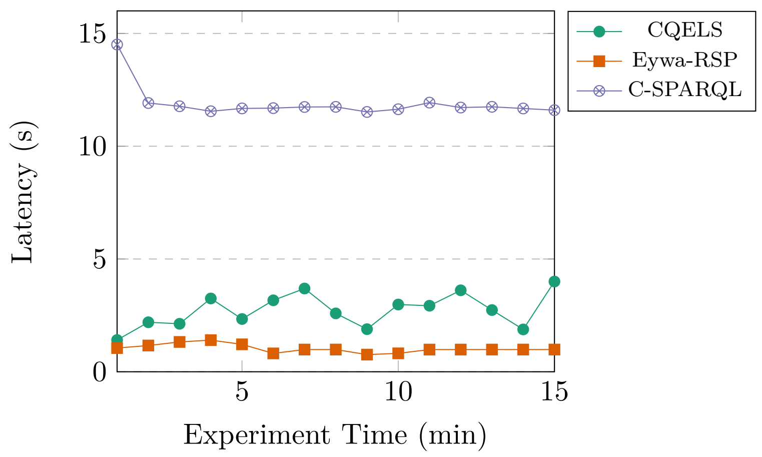


Figure 6.9: Latency of CityBench Q1 across Time

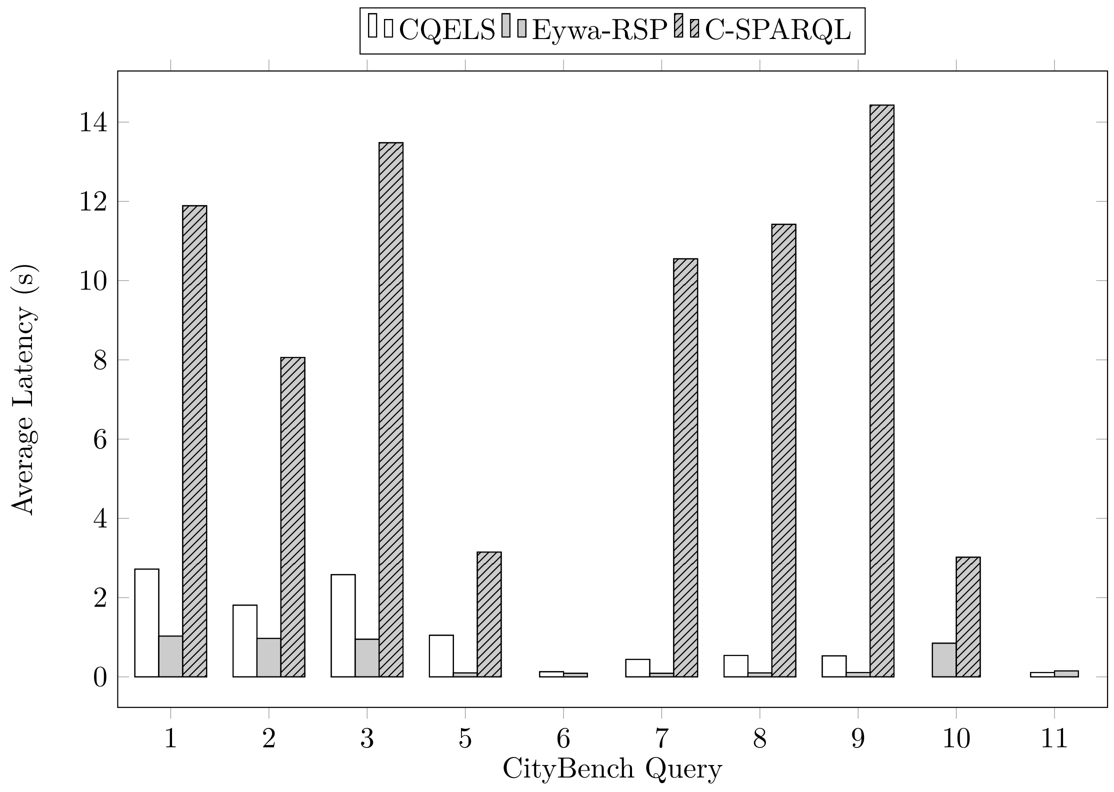
minutes, are summarised in Figure 6.10. Eywa-RSP was the fastest on each of the queries followed by CQELS and then C-SPARQL. By representing metadata triples within data model mappings which are processed in the query translation process as opposed to in the stream itself, Eywa-RSP was able to more efficiently process more concise streams of IoT time-series data than CQELS and C-SPARQL, resulting in better average latency for almost every query. The only exception was for Query 11, which notified when any weather observation was detected, where CQELS was marginally faster. As CQELS processed triples, the CQELS query was actually simpler and was optimised with adaptive caching as it specified a single BGP for a general weather observation, which had one less triple than one which specified the type of observation. In Eywa-RSP, a union of all the '`wspdm`', '`tempm`' and '`hum`' had to be performed on the stream.

Figure 6.11 shows the algebra of the translated Query 1 from CityBench, registered on the client node. Π is the operator for retrieving the projected columns (e.g. congestionLevel) from each event in the stream window from a source node.

The algebra for CQELS and C-SPARQL on the other hand, as shown in Figure 6.12, require the retrieval of, τ_{BGP}, for each event in the stream. The extra triples for each observation in the stream are shown in Listing 6.13, representing the observation metadata (lines 1 to 3) and relation to sensor metadata (line 4). It is more expensive in terms of overall latency when retrieving the extra triples and processing the query without workload distribution at the source.

```
1 ?obId1 a ?ob.
2 ?obId1 ssn:observedProperty ?p1.
3 ?obId1 sao:hasValue ?v1.
4 ?obId1 ssn:observedBy <http://...#AarhusTrafficData182955>.
```

Listing 6.13: Additional Metadata Triples for CQELS and C-SPARQL



Queries for each engine that could not be run were Eywa:4, CQELS:4,10,12, C-SPARQL:4,6,11,12

FIGURE 6.10: Average Latency of CityBench Queries on RSP Engines

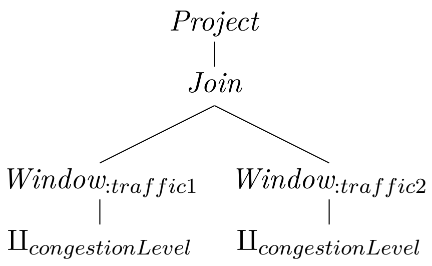


FIGURE 6.11: Algebra of Query 1 for Eywa-RSP

6.3.2 Scalability Evaluation Results

Scalability evaluation looked at the amount of memory resources used by each RSP engine and the results when increasing the number of concurrent queries and increasing the number of data streams. The lower the memory resources used, the more scalable the system was, especially on resource-constrained fog nodes.

Figure 6.13 shows the memory consumption of each engine across time. Eywa-RSP streamed a more concise format of time-series data points as events and only the necessary projected fields from the source node, hence, as compared to the more verbose set of triples shown in Listing 6.14 for other engines, Eywa-RSP used significantly less

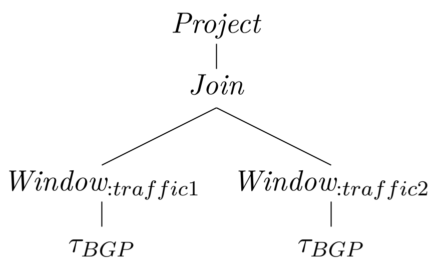


FIGURE 6.12: Algebra of Query 1 for CQELS and C-SPARQL

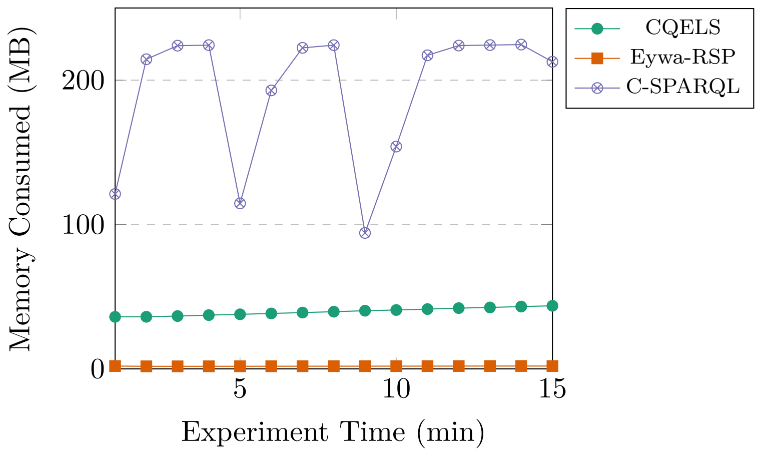


FIGURE 6.13: Memory Consumed by CityBench Query 2 across Time on Client

memory, about 20 times less than CQELS and two orders of magnitude less than C-SPARQL. As expected, the more complex Query 5[19], a filter operation on two graphs and a stream, took the most memory and Query 11[20], checking for observations from weather sensors only on a single stream, took the least on all engines with Eywa-RSP an order of magnitude less than CQELS and two orders of magnitude less than C-SPARQL.

```
:Property1 a ct:CongestionLevel.
:Observation1 a ssn:Observation;
  ssn:observedProperty :Property1;
  sao:hasValue "congestionLevel";
  ssn:observedBy ns:AarhusTrafficData158505;
  time:inXsdDateTime "timestamp".
:Observation2 a ssn:Observation;
  ssn:observedProperty :Property2;
  sao:hasValue "avgSpeed";
  ssn:observedBy ns:AarhusTrafficData158505;
  time:inXsdDateTime "timestamp".
```

LISTING 6.14: Set Of Triples from a Row/Event of the Traffic Stream

When increasing the number of concurrent queries on each engine, Eywa-RSP once again achieved the lowest memory consumption. The memory consumed on Eywa-RSP was also the only one consistent and did not increase over time like C-SPARQL and CQELS.

[19]https://github.com/eugenesiow/Benchmark/wiki/Q05
[20]https://github.com/eugenesiow/Benchmark/wiki/Q11

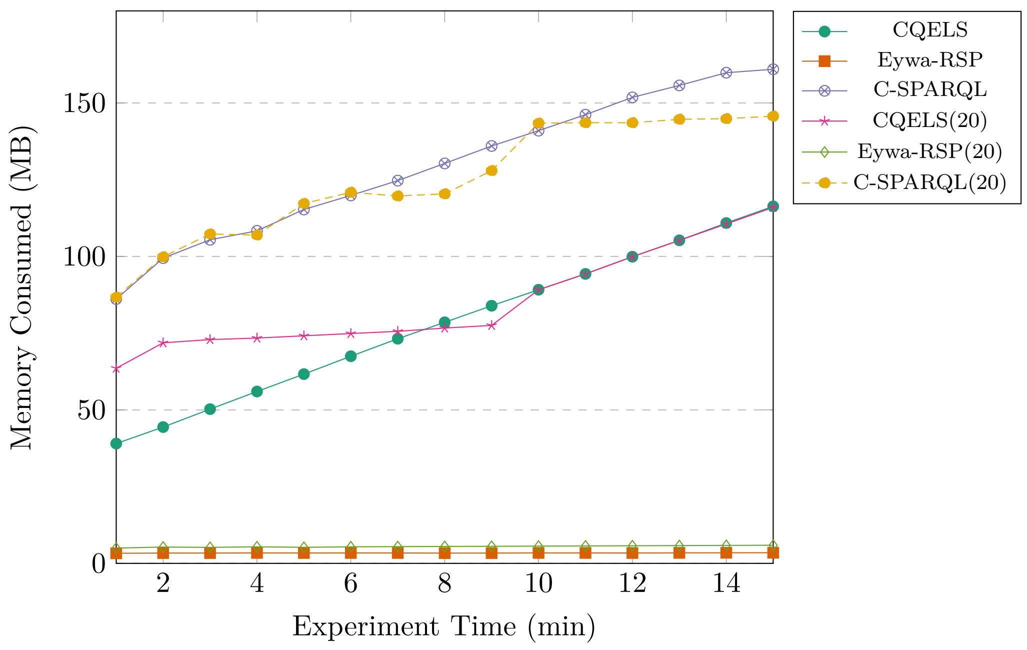


FIGURE 6.14: Memory Consumed when Increasing Number of Concurrent Queries on CityBench Query 5 from 1 to 20

This is summarised in Figure 6.14 which shows the results of Query 5 of CityBench at two different configurations of a single query and 20 concurrent queries. Eywa-RSP had a slight, stable 2MB to 5MB increase in memory consumption on increasing concurrent queries. There were consistent results from these two configurations for other CityBench queries and while Query 2 and Query 8[21], combining two parking streams to find vacancies, took up significantly more memory for C-SPARQL, Query 1 and Query 3[22], combining two traffic streams to find the average congestion level, took up more memory for CQELS. Eywa-RSP took up at least an order of magnitude less memory for each query. The results and experiment source were published online in a similar way as the original paper for comparison[23].

Figure 6.15 shows the memory consumption when increasing the number of pollution data streams from 2 to 5 in CityBench's Query 10 which looks for the most polluted area in the city in real-time[24]. Eywa-RSP once again had the lowest memory consumption and C-SPARQL actually ran out of memory on the resource-constrained client in the 5 stream configuration just before 15 minutes.

Finally, to observe how much the Eywa fog computing framework, through the projection operator on the source node, actually improved the memory consumption and scalability, 5 and 8 pollution streams from CityBench Query 10 were tested on the fog computing setup, which was used in previous experiments, and a single-node setup. The client,

[21]https://github.com/eugenesiow/Benchmark/wiki/Q08
[22]https://github.com/eugenesiow/Benchmark/wiki/Q03
[23]https://github.com/eugenesiow/Benchmark/tree/master/result_log/samples
[24]https://github.com/eugenesiow/Benchmark/wiki/Q10

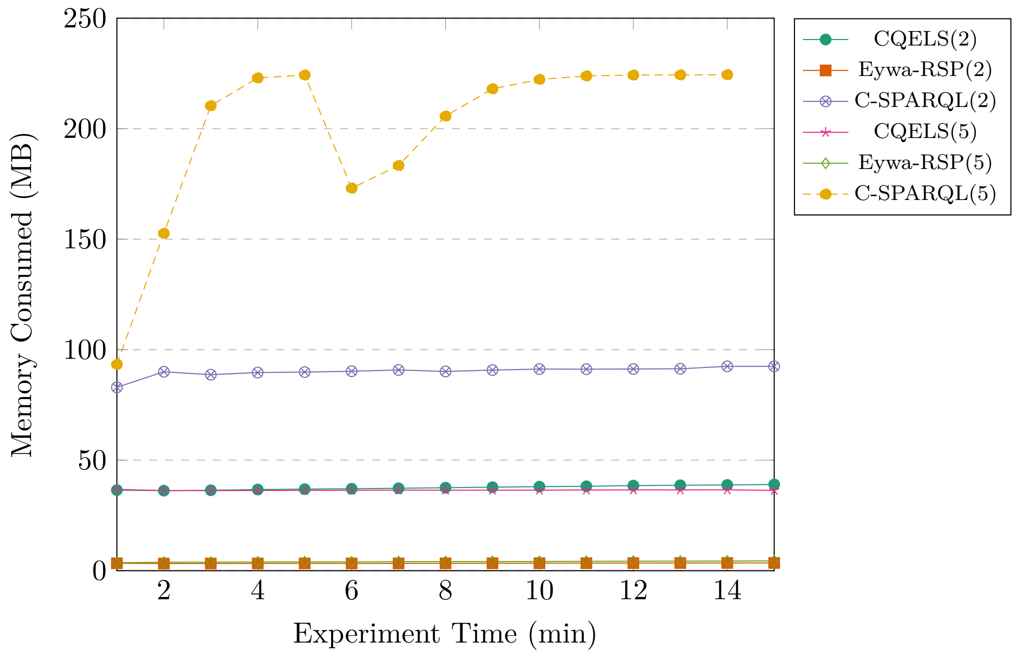


FIGURE 6.15: Memory Consumed by CityBench Query 10 with Varying Data Stream Configurations of 2 and 5

The graph lines formed from the results of Eywa-RSP(2) and Eywa-RSP(5) are very close, as are those from CQELS(2) and CQELS(5).

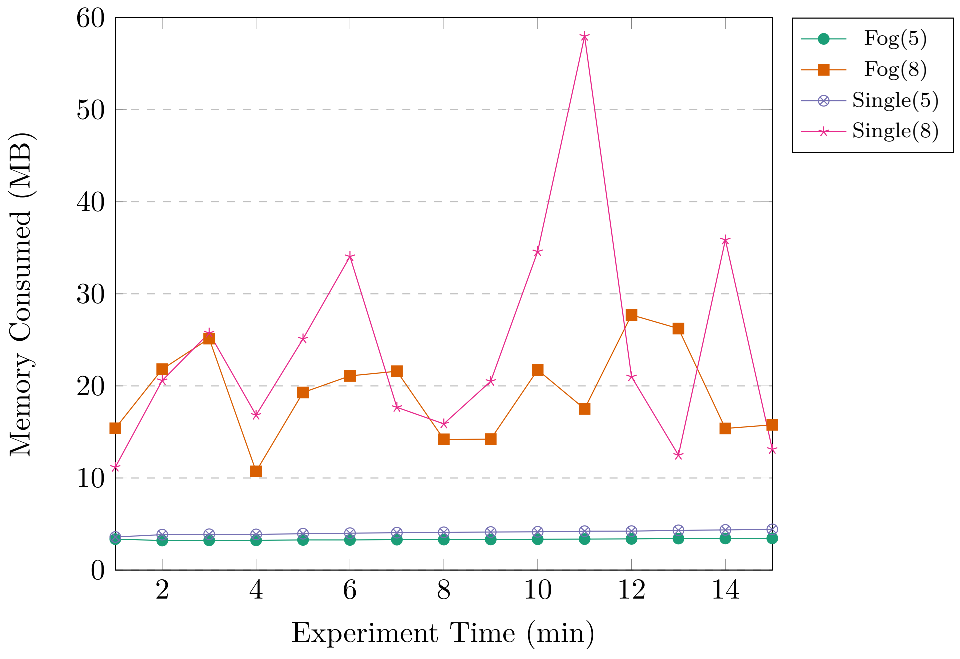


FIGURE 6.16: Fog vs Single-Node Eywa-RSP Networks: Memory Consumed while Increasing Streams on CityBench Query 10 from 5 to 8

broker and source nodes were all run on the same node in the single-node setup. Figure 6.16 shows the amount of memory consumed by the distributed fog execution of the query (on the client) and the single-node execution of the query.

As the projection operator was distributed to the source nodes and only the applicable part of the stream was sent to the client, there was a significant difference in memory consumed when the number of incoming data streams was increased. At 2 and 5 streams for Query 10, the memory consumed was similar for distributed fog and single-node setups, hence the results of 2 streams was omitted from the graph. When there were 8 streams on Query 10 however, the fog computing approach consumed less memory. This was due to the flow of data being significantly large enough that the projection operation passed down to the source node of each stream also had a significant effect on the overall memory consumed. Hence, as the amount of streams increase, Eywa's Fog Infrastructure provides greater scalability.

6.4 Fog Computing within the Internet of the Future

The Internet has been a great motor of socio-economic activity in the past decade and the Internet of the Future has the potential to do even more. Like the fictional Eywa in Avatar, Eywa as a fog computing infrastructure has the potential to make the IoT and the planet more intelligent, more connected and at the same time more engaged and social, while valuing privacy and security.

Learning from the web, that grew as a free, open and standards-based information space on top of the internet (Berners-Lee and Fischetti, 2000), the IoT needs to endeavour to support interoperability. Modelling and publishing metadata as RDF not only encourages interoperation through the use of common identifiers in IRIs and referencing common ontologies, it also represents concepts as machine-readable graphs (Bizer et al., 2009) which goes towards solving challenges like discovery and data integration in the IoT. Fog computing that serves as a layer between sensors and actuators, and the cloud, and possesses comparatively more compute and storage than sensors, serves as the entry level compute and storage platform for interoperability and integration.

The current business model for many web services is that free content is exchanged for our personal data. However, 'as our data is held in proprietary silos, out of sight to us, we lose out on the benefits we could realise if we had direct control over this data and chose when and with whom to share it[25]', argued inventor of the web Sir Tim Berners-Lee. Fog computing on lightweight, distributed computers means that data that is collected from sensors and devices are stored and processed locally. As access control technology evolves, specific privacy policies with additional trust and fault tolerance mechanisms can be created (Roman et al., 2013) to control what goes to the cloud.

[25]http://theguardian.com/technology/2017/mar/11/tim-berners-lee-web-inventor-save-internet

Furthermore, as data is stored and processed by some applications locally, the need to shuttle data to and from the cloud is minimised, providing better guarantees of quality of service for mission-critical IoT apps and there is less dependency on supporting high bandwidth global connections. In disaster management IoT scenarios, where last-mile connectivity is lost, having data locality and offline access is especially valuable. There are also performance benefits over storing encrypted data in traditional clouds as a means to maintain privacy because it is easier to perform processing over data. For example, there is no need for crypto-processors or to apply special encryption functions on data. When access controls permit, data can still be sent to the cloud for big data analytics.

Finally, as the internet and IoT advance, studying the socio-technical aspects of the intersection between intelligent, co-operative, autonomous machines in the IoT and human users is gaining importance. Fog computing, as a distributed, interoperable application layer between the network, autonomous IoT end-devices, applications and human end-users, can serve 1) to build and manage this socio-technical network, 2) to facilitate information flow or sharing, 3) to host applications and 4) as an observation platform for the study of these emerging networks. This application of a decentralised Social Web of Things is covered in greater detail in the next chapter (Chapter 7).

6.5 Conclusions

This chapter highlighted the extension of Map-Match-Operate to efficiently process real-time streams in the IoT and proposed an infrastructure, Eywa, for fog computing that shows how resource-constrained platforms close to the source IoT agents can be utilised for stream processing.

The implementation of S2S, a Map-Match-Operate engine that translates streaming SPARQL graph queries expressed as RSP-QL to the Event Processing Language (EPL), showed greater efficiency than state-of-the-art RSP engine CQELS for the same published benchmarks, within the same experimental setup as that described in Chapter 4.

Eywa, which utilises an inverse-publish-subscribe system to distribute streaming queries across fog nodes running S2S engines, showed improvements in stream processing within a smart city scenario for both scalability and latency metrics as compared to CQELS and C-SPARQL. The current design limitations of Eywa though, are that it makes the assumptions of a co-operative network, with a set of well-known broker nodes and best-effort delivery from source to client nodes. Hence, applications that require strict guarantees on the correctness of results cannot be served by Eywa at present. Mitigating the reliability of Eywa would require implementing a Byzantine Fault Tolerant (Lamport et al., 1982) consensus within the distributed system which is not the focus of this thesis.

The next chapter introduces analytical applications and platforms that put into perspective how the efficient data and event management that Map-Match-Operate brings for historical and streaming data, can motivate the socio-technical development of the IoT especially when considering the emerging distributed fog computing paradigm.

Chapter 7

Applications of Efficient Graph Querying and Analytics

"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."

— *Donald Knuth*

This chapter introduces analytical applications and platform infrastructure that build on the interoperable and efficient data management with Map-Match-Operate for historical and streaming data in the Internet of Things (IoT). Previous chapters in this thesis introduced and have already evaluated Map-Match-Operate as a means of performant and interoperable IoT time-series storage, hence, the main aim of the applications and platforms described in this chapter are not to validate Map-Match-Operate but rather to show how this research can lead to applications in different directions. Section 7.1 describes how Map-Match-Operate can power personal IoT repositories and analytical dashboards running on them while Section 7.2 describes how TritanDB can be used for time-series analytics. Section 7.3 introduces how a Social Web of Things, based on social capital theory, can be built on current infrastructure and be evolved over time, producing greater value for personal repositories. A mechanism for distributing IoT information within resource-constrained decentralised social networks under ten thousand nodes is also devised by comparing the scalability of existing technologies.

The work involving personal IoT repositories in this chapter was published in the Siow et al. (2016b) paper and demonstrated at the International Semantic Web Conference.

7.1 PIOTRe: Personal Internet of Things Repository

Resource-constrained IoT devices like Raspberry Pis, with specific performance optimisation, can serve as interoperable personal repositories that store and process time-series IoT data for analytical applications. This thesis implements PIOTRe, a personal datastore that utilises S2S query translation technology on Pis to process, store and publish historical and streaming IoT time-series data. PIOTRe was demonstrated in a smart home scenario with a real-time dashboard and historical visualisation application that utilised RDF stream processing (RSP), a set of descriptive analytics visualisations on historical data, a framework for registering stream queries within a local fog computing network and a means of sharing metadata globally with Thing directories and catalogues like HyperCat. In particular, the following contributions are presented in this section:

- A real-time smart home dashboard that utilises S2S's RDF stream processing (RSP) engine to push results to descriptive analytics widgets via websockets.
- A smart home descriptive analytics application that uses a set of SPARQL queries with space-time aggregations, translated by S2S, to visualise historical data over time, stored by TritanDB.
- A means of publishing and sharing metadata and mappings with a standardised format called HyperCat.

Section 7.1.1 covers the design and architecture of PIOTRe while Section 7.1.2 introduces a real-time dashboard and a historical visualisation application, both running on PIOTRe. Section 7.1.3 and Section 7.1.4 show how PIOTRe interacts with a fog computing network using Eywa and the wider IoT using HyperCat respectively.

7.1.1 Design and Architecture of PIOTRe

PIOTRe was designed, tested and demonstrated to run on a compact and mobile lightweight computer, like a Raspberry Pi, or an x86 alternative, like the Gizmo2[1]. PIOTRe can also start as a source, client or broker node, or a combination of all three, within an Eywa fog computing network as described in Section 6.2. Internally, PIOTRe consists of a number of components as shown in Figure 7.1 that are described as follows.

Sensors, human input and other event streams take the form of time-series input data streams to the PIOTRe system. Data streams are ingested as an event stream with Esper which underlies the RDF stream processing (RSP) engine and are also simultaneously stored in a historical TritanDB store (Chapter 5). Each stream forms a time-series

[1]http://www.gizmosphere.org/products/gizmo-2/

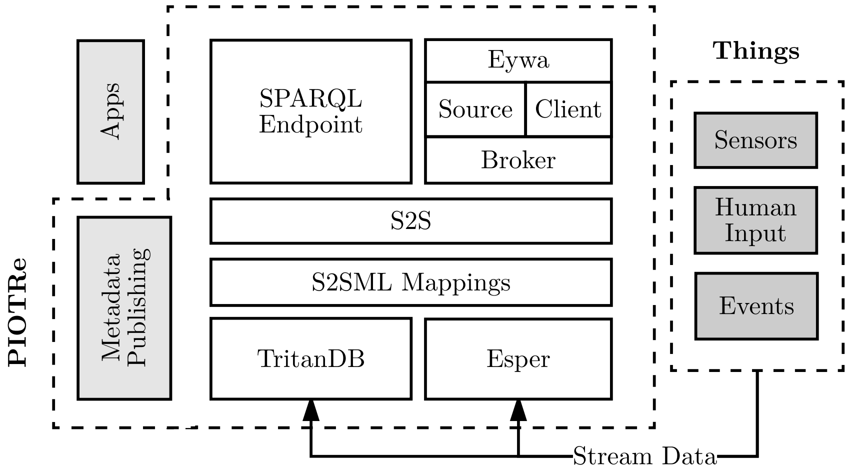


Figure 7.1: Architecture of PIOTRe

within TritanDB. The RSP engine is driven by an S2S engine that works on translating RSP-QL queries to Event Processing Language (EPL) queries for Esper[2], the method of which was described in Section 6.1.

Mappings are shared across the S2S engine for streams and TritanDB and are represented in S2SML, as introduced in Section 4.2.1.1, which is an RDF representation of how a metadata graph maps to fields within a TritanDB time-series or an event within a stream. PIOTRe provides both a visual What You See Is What You Get (WYSIWYG) interface, as shown in Figure 7.2, and a script editor to author S2SML data model mappings.

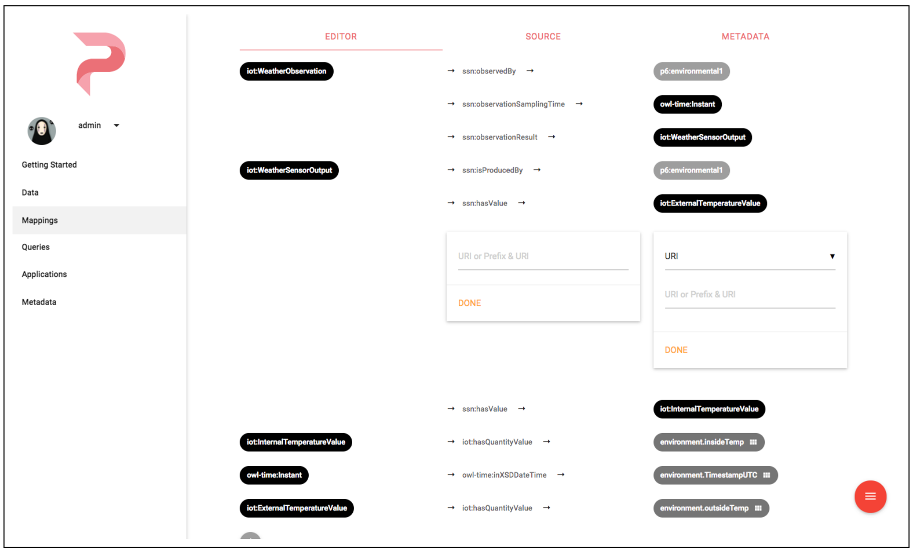
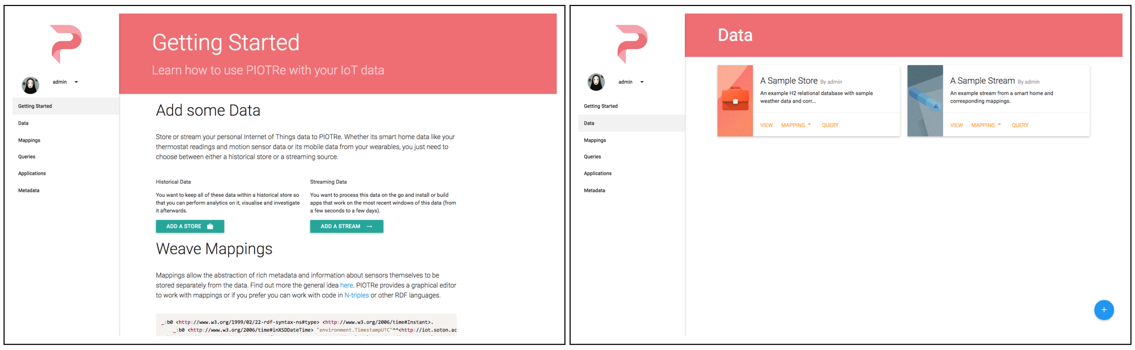


Figure 7.2: What You See Is What You Get (WYSIWYG) S2SML Editor in PIOTRe

Applications in PIOTRe have the flexibility of being written in any language and framework and communicate with the TritanDB through the interface of a SPARQL Endpoint for historical data and with S2S for streams by registering an RSP-QL continuous query

[2]http://www.espertech.com/

with an Eywa client. The metadata publishing component publishes the metadata from mappings, like the sensor descriptions, locations, data fields and formats to support global discovery and interoperability with the wider IoT. Screenshots of the PIOTRe interface, which walks users through adding data and weaving mappings, are shown in Figure 7.3 and the source code is published online on GitHub[3].



Getting Started

Learn how to use PIOTRe with your IoT data

Add some Data

Store or stream your personal Internet of Things data to PIOTRe. Whether its smart home data like your thermostat readings and motion sensor data or its mobile data from your wearables, you just need to choose between either a historical store or a streaming source.

Historical Data

You want to keep all of these data within a historical store so that you can perform analytics on it, visualise and investigate it afterwards.

Streaming Data

You want to process this data on the go and install or build apps that work on the most recent windows of this data (from a few seconds to a few days).

Weave Mappings

Mappings allow the abstraction of rich metadata and information about sensors themselves to be stored separately from the data. Find out more the general idea here. PIOTRe provides a graphical editor to work with mappings or if you prefer you can work with code in N-triples or other RDF languages.

```
_:b0 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.w3.org/2006/time#Instant>.
    _:b0 <http://www.w3.org/2006/time#inXSDDateTime> "environment.TimestampUTC"^^<http://iot.soton.ac
```

Data

A Sample Store By admin

An example H2 relational database with sample weather data and corr...

A Sample Stream By admin

An example stream from a smart home and corresponding mappings.

FIGURE 7.3: Getting Started with PIOTRe and Adding IoT Data Sources

7.1.2 Example Applications using PIOTRe

IoT Freeboard[4] is a real-time dashboard that consists of a simulator that replays a stream of smart home data at a variable rate to PIOTRe and a high customisable web application dashboard that receives the output from the set of registered push-based streaming RSP-QL queries as a websocket events stream. Each result in the stream consists of the query name it was generated by and fields of the result set. An example in Listing 7.1 shows a result of an RSP-QL query that calculates the average power from a particular smart meter every time there is a change in consumption. This result is then visualised as a spark line on the application dashboard that shows the trend of power consumption of this meter. A host of other widgets can be added to the dashboard to visualise events. A demonstration of the dashboard's capabilities was presented at the International Semantic Web Conference and is available online[5].

```
{
    "queryName":"averagePower",
    "averagePower":"10.5",
    "uom":"W"
}
```

LISTING 7.1: JSON Result Streamed through WebSocket from RSP Engine in PIOTRe for a Query Retrieving Average Power from a Smart Home Meter

The Pi Smart Home Dashboard[6] is an application that uses SPARQL queries on historical smart home data to provide visualisations across time and space. The queries

[3]https://github.com/eugenesiow/piotre-web
[4]https://github.com/eugenesiow/iotwo
[5]https://youtu.be/oH0iSWTmKUg
[6]https://github.com/eugenesiow/ldanalytics-PiSmartHome

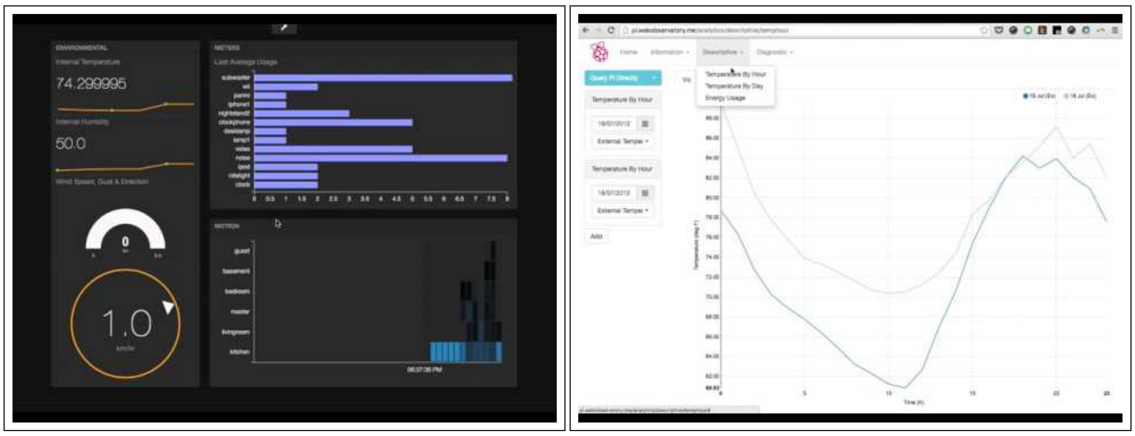


Figure 7.4: Applications using Streaming and Historical Data with PIOTRe

are presented within the Smart Home Analytics Benchmark described in Appendix B. The application allows the user to tweak the time range and sensor types through a user interface that changes parameters to SPARQL queries which generates different resulting graph visualisations. A demonstration video is available online and a live demo was presented at the International Semantic Web Conference[7]. Figure 7.4 shows screenshots of IoT freeboard on the left and the Pi Smart Home on the right.

7.1.3 PIOTRe as an Eywa Fog Computing Node

PIOTRe stores, processes and runs applications on data. As such, PIOTRe can serve as a source, client and broker or a combination of all three within an Eywa fog computing network to perform RDF stream processing (RSP) on streams. In some IoT scenarios like disaster management or environmental monitoring in remote locations, Eywa can be used as a local, offline network for processing data. The Eywa client-broker-server architecture was described previously in Section 6.2 and enables stream queries to be registered within the control plane of a fog computing network using inverse-publish-subscribe and allows results to be delivered by a push-pull mechanism within the data plane of the network. The mechanism is known as inverse-publish-subscribe as source nodes subscribe to IRIs instead of publishing to topics and clients nodes publish queries directed at particular IRIs and this communication is facilitated by brokers with known addresses. An Eywa client uses the S2S engine to translate and execute RSP-QL queries across the distributed network. The entire sequence of events from registering a query to receiving results was described previously with a sequence diagram in Section 6.2.3.

[7]https://youtu.be/g8FLr974v9o

7.1.4 Publishing and Sharing Metadata with HyperCat

HyperCat[8] is a catalogue for exposing and describing collections of IRIs that refer to IoT assets over the web. PIOTRe automatically publishes metadata describing streams and historical data sources of Things within its repository as a HyperCat catalogue when the system is online. PIOTRe acts as a HyperCat server and serves the main catalogue through the '`/cat`' endpoint as a JSON document following the HyperCat 3.00 specification described by Jaffey et al. (2016). Listing 7.2 shows an automatically generated catalogue from PIOTRe which consists of a historical store and a stream, each providing a unique IRI with the '`href`' key (lines 10 and 20) which can be accessed to discover individual metadata catalogues describing the related Things. For example, the historical store consists of data from temperature, wind speed and humidity sensors.

```
 1 {
 2   "catalogue-metadata": [
 3     { "rel": "urn:X-hypercat:rels:isContentType",
 4       "val": "application/vnd.hypercat.catalogue+json"
 5     },
 6     { "rel": "urn:X-hypercat:rels:hasDescription:en",
 7       "val": "My PIOTRe Catalogue"
 8     } ],
 9   "items": [
10     { "href": "/data/e587f08b-dd2a-43bc-87e9-e5560cd9b9c5",
11       "metadata": [
12         {
13           "rel": "urn:X-hypercat:rels:hasDescription:en",
14           "val": "Example TritanDB database with weather data and mappings."
15         },
16         {
17           "rel": "http://purl.org/net/iot#type",
18           "val": "store"
19         } ] },
20     { "href": "/data/f3bc4714-8658-426f-bc71-0b1fe7e41900",
21       "metadata": [
22         {
23           "rel": "urn:X-hypercat:rels:hasDescription:en",
24           "val": "An example stream and mappings from a smart home."
25         },
26         {
27           "rel": "http://purl.org/net/iot#type",
28           "val": "stream"
29         } ] } ]
30 }
```

Listing 7.2: A JSON HyperCat Catalogue Generated by PIOTRe

Listing 7.3 shows an abbreviated HyperCat catalogue, generated for the example historical TritanDB store in Listing 7.2, showing the metadata of the wind direction sensor automatically generated from the S2SML mappings. This is possible as HyperCat

[8]http://www.hypercat.io/

supports the listing of any number of IRIs and any number of RDF-like triple statements related to them. Hence, the unit of measure, ‘`uom`’ and the observed property, ‘`WindDirection`’ are detected from the mapping because they are linked as triples to the binding of the underlying TritanDB time-series field, ‘`WindDirection`’.

```
{
  "item-metadata": [
    { "rel": "urn:X-hypercat:rels:isContentType",
      "val": "application/vnd.hypercat.catalogue+json"
    },
    { "rel": "urn:X-hypercat:rels:hasDescription:en",
      "val": "Example TritanDB database with weather data and mappings."
    } ],
  "items": [
    { "href": "/data/e587f08b-dd2a-43bc-87e9-e5560cd9b9c5/WindDirection",
      "metadata": [
        {
          "rel": "urn:X-hypercat:rels:hasDescription:en",
          "val": "WindDirection"
        },
        {
          "rel": "ssw:uom",
          "val": "weather:degrees"
        },
        {
          "rel": "ssw:observedProperty",
          "val": "weather:WindDirection"
        } ] } ]
}
```

Listing 7.3: Sensor Metadata of the Historical Store HyperCat Catalogue

7.2 Time-series Analytics on TritanDB

Chapter 5 showed that TritanDB, as a Map-Match-Operate engine for interoperable graph-based metadata and time-series IoT data, provided better storage and query performance than a range of time-series, relational and NoSQL databases. This section presents the features of TritanDB that enable applications to utilise queries written in SPARQL, with appropriate extension functions, for time-series analysis.

7.2.1 Resampling and Conversion of Non-Periodic Time-series

In Chapter 3, it was discovered that IoT data collected from across a range of domains in the IoT consisted of both evenly-spaced, periodic or approximately periodic, and non-evenly spaced, non-periodic time-series. While there exists an extensive body of literature on the analysis of evenly-spaced time-series data (Lütkepohl, 2010), few methods exist specifically for unevenly-spaced time series data. As Eckner (2014) explained, this was because the basic theory for time-series analysis was developed 'when limitations in computing resources favoured the analysis of equally spaced data', where efficient linear algebra routines could be used to provide explicit solutions. TritanDB that works across both resource-constrained fog computing platforms and the cloud, provides two efficient methods for dealing with unevenly-spaced data.

One method of transforming unevenly-spaced to evenly-spaced time-series in TritanDB is resampling. This is achieved by splitting the time series into time buckets and applying an aggregation function, such as an average function, to perform linear interpolation on the values in that series. Listing 7.4 shows a SPARQL 1.1 query compliant with the W3C specification that converts an unevenly-spaced time-series to an hourly aggregated time-series of average wind speed values per hour.

```
SELECT (AVG(?wsVal) AS ?val) ?hours WHERE {
  ?sensor isA windSensor;
    has ?obs.
  ?obs hasValue ?wsVal;
    hasTime ?time.
  FILTER (?time>"2003-04-01T01:00:00" && ?time<"2003-04-01T11:00:00"^^xsd:dateTime
    )
} GROUP BY (hours(?time) as ?hours)
```

Listing 7.4: SPARQL Query on TritanDB to Resample the Unevenly-spaced to an Evenly-spaced Wind Speed Time-series

However, as Eckner (2014) summarised from a series of examples, performing the conversion from unevenly-spaced to evenly-spaced time-series results in data loss with dense points and dilution with sparse points, the loss of time information like the frequency of observations, and causality being affected. The linear interpolation used in resampling also 'ignores the stochasticity around the conditional mean' which leads to a difficult-to-quantify but significant bias when various methods of analysing evenly-spaced time-series are applied, as shown in experiments comparing correlation analysis techniques by Rehfeld et al. (2011).

Hence, a more graceful approach to working with unevenly-spaced time-series is to use Simple Moving Averages (SMA). Each successive average is calculated from a moving window of a certain time horizon, τ, over the time-series. An efficient algorithm to do so is from an SMA function defined in Definition 7.1 as proposed by Eckner (2017).

Definition 7.1 (Simple Moving Average, $SMA(X,\tau)_t$)**.** Given an unevenly-spaced time-series, X, the simple moving average, for a time horizon of τ where $\tau > 0$ and $t \in T(X)$, is $SMA(X,\tau)_t = \frac{1}{\tau}\int_0^{\tau} X[t-s]ds$. $T(X)$ is the vector of the observation times within the time-series X, $X[t]$ is the sampled value of time series X at time t and s is the spacing of observation times (between this and the last).

Figure 7.5 shows a visualisation of how SMA is calculated. Each observation is marked by a cross in the figure and this particular time horizon is from $t-\tau$ to t. The area under the graph averaged over τ gives the SMA value for this window. In this case, s is the time interval between the rightmost observation at t and the previous observation.

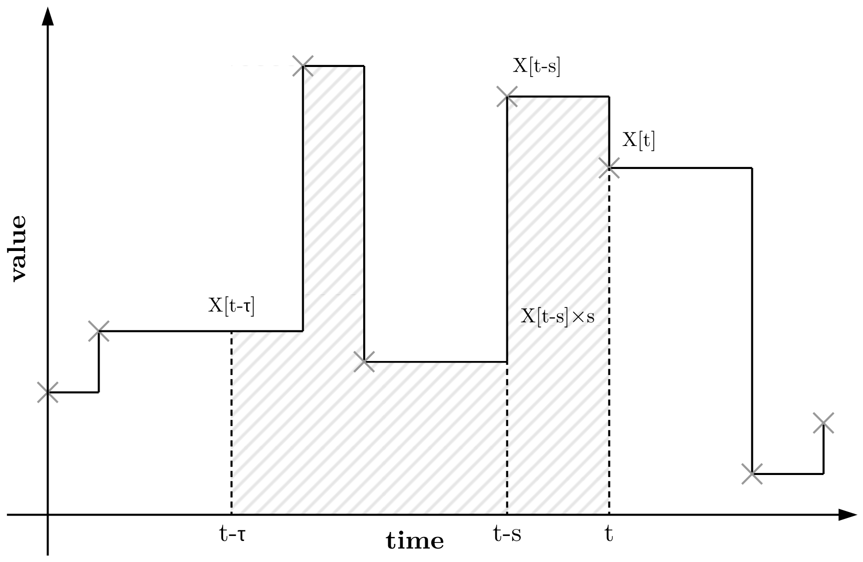


FIGURE 7.5: Visualisation of Simple Moving Average Calculation

The ‘`hours()`’ function in Listing 7.4’s query can be changed to a ‘`sma(?time,tau)`’ function from an extension to SPARQL implemented in TritanDB. This produces an SMA time-series using the efficient algorithm implementing the SMA function by Eckner (2017) and shown in pseudo-code form in Listing 7.5.

```
left = 1; area = left_area = X[1] * tau; SMA[1] = X[1];
for (right in 2:N(X)) {
  // Expand interval on right end
  area = area + X[right-1] * (T[right] - T[right-1]);
  // Remove truncated area on left end
  area = area - left_area;
  // Shrink interval on left end
  t_left_new = T[right] - tau;
  while (T[left] <= t_left_new) {
    area = area - X[left] * (T[left+1] - T[left]);
    left = left + 1;
  }
  // Add truncated area on left end
  left_area = X[max(1, left-1)] * (T[left] - t_left_new)
  area = area + left_area;
  // Save SMA value for current time window
  SMA[right] = area / tau;
}
```

LISTING 7.5: Algorithm to Calculate Simple Moving Average

This algorithm incrementally calculates the SMA for a N(X)-sized moving window, where N(X) is the number of observations in time-series X, reusing the previous calculations. There are four main areas (under the graph) involved for each SMA value calculated, the right area, the central area, the new left area and the left area. The central area is unchanged in each calculation. The algorithm first expands and adds the new right area from T[right] - T[right-1], where `right` is the new rightmost observation. The leftmost area from the previous iteration is removed and any additional area to the left less than $T[right] - \tau$, the time horizon, is also removed. The removed area is the left area. A new left area from $T[right] - \tau$ till the next observation is then calculated and added to the total area. This new total area is then divided by the time horizon value, τ, to obtain the SMA for this particular window.

7.2.2 Custom Models For Forecasting

TritanDB includes an extendable model operator that is added to queries as a function extension in SPARQL. An example is shown in Listing 7.6 which shows how a forecasting of the next months points of evenly-spaced time-series can be made using a moving average function, a seasonal ARIMA model function with a 4-week seasonal cycle and a years worth of time-series data in a SPARQL query.

```
SELECT (FORECAST(?tVal,30) AS ?val) WHERE {
  ?sensor isA tempSensor;
    has ?obs.
  ?obs hasValue ?tVal;
    hasTime ?time.
  FILTER (?time>"2011-04-01T00:00:00" && ?time<"2012-04-01T00:00:00"^^xsd:dateTime
    )
} GROUP BY ARIMA_S(sma(?time,1d),4w)
```

Listing 7.6: Forecasting a Month with a 4-week-cycle Seasonal ARIMA Model on a Year of Time-series

The result set of the query includes a forecast of 30 points representing values of the temperature sensor in the next month. ARIMA and random walk models are also included from an open source time-series analysis library[9].

7.3 Social Web of Things

One of the major advantages of the IoT is that the collective knowledge and increased spatio-temporal observation space enables a level of decision-making that is greater than the sum of its parts. How this flow of information is construed and managed, however, is a challenging problem, especially as the volume of information increases past the human capacity for comprehension.

Atzori et al. (2011) first conceptualised a Social Internet of Things (SIoT) as a 'social network of intelligent objects' and further developed a conceptual three-tiered architecture consisting of application, network and sensing layers centred around SIoT servers,

[9]https://github.com/jrachiele/java-timeseries

gateways and smart objects (Atzori et al., 2012). Establishing and maintaining social relationships for the exchange of information within the SIoT are recognised as essential within their SIoT concept. One possible socio-technical approach to realise the SIoT is to combine an understanding of Social Capital, a resource made available by developing norms of trust and reciprocity through interactions with relations within social networks, with an understanding of technologies that can facilitate a technological transition from human-centric networks to machine-automated ones. The foundations for efficient and interoperable data management developed within this thesis form a basis for this application that proposes a decentralised social network application ecosystem called the Social Web of Things (SWoT) on top of Map-Match-Operate with publish-subscribe technologies for information exchange. Section 7.3.1 formalises the SWoT, drawing inspiration from the dimensions of Social Capital Theory. Section 7.3.2 then describes how a transition can be made from a human-driven SWoT to a machine-automated one, that can handle the increasing volume of data, for the core SWoT functions. Next, Section 7.3.3 describes how experiments advise a scalable publish-subscribe infrastructure for decentralised SWoT fog computing nodes. Finally, Section 7.3.4 describes an experimental SWoT implementation.

7.3.1 Social Capital Theory Inspired SWoT

The Social Web of Things (SWoT) is an application that manages the information flow within the IoT. The SWoT draws from the theory of Social Capital, that identifies resources for the conduct of activities, for example, information access and collaboration, gained from the interaction with relations within a social network, to shape its design.

The sociological origin of the term social network, as explained by Scott (2012), was as a metaphor to relate the social world, the 'intertwined mesh of connections' in which actors were bound, to familiar everyday concepts drawn from textile production. Within this mesh of connections, the interaction between actors creates norms of trust and reciprocity, which generates an abstract resource known as Social Capital. Social Capital then gives access to information, opportunities and collaborative work (Nahapiet and Ghoshal, 1998). Easley and Kleinberg (2010) summed up the notion of Social Capital nicely as providing a framework for considering 'social structures as facilitators of effective action' by actors. The purpose of the SWoT is not to develop a metric for abstract Social Capital measurement but to take inspiration from its theoretical aspects, specifically its three dimensions, structural, cognitive and relational, to design and equip the *social structure*, the network connecting Things, for the *effective action* of disseminating information for the running of applications and collaborating on work.

Nahapiet and Ghoshal (1998) identified the dimensions of Social Capital motivating 'knowledge transfer' as

- structural, how the network is connected and configured,
- cognitive, through the sharing of language and narratives and
- relational, referring to the trust, norms and obligations in the network.

Firstly, Definition 7.2 defines the underlying network called the SWoT core network which consists of a graph of Things and edges. Next, Definition 7.3 introduces the functions that facilitate the establishment and growth of G and the structural dimension of Social Capital in the SWoT, and Definition 7.4 describes, using terminology from information theory by Sanders et al. (2003), the network information flow.

Definition 7.2 (SWoT Core Network)**.** *A SWoT network consists of a finite, directed graph, G, where $G = (V, E)$. V are the vertices of G and E are the ordered edges of G where $E \subseteq V \times V$.*

Each vertex v, where $v \in V$, is a Thing and each edge e, where $e \in E$, refers to a set of relations between Things. A Thing can refer to a 1) physical thing like a sensor, actuator or device, or 2) a virtual thing like an intelligent agent, app or human avatar. A relation set, R, is obtained by a function, ρ, where $\rho(e) = R$. r, where $r \in R$, is a relation between Things, for example, follows or trusts, or any of the relationships defined by Atzori et al. (2012) like a parental object or a co-location object relationship.

Definition 7.3 (SWoT Structural Dimension Functions)**.** *A discovery function, $\delta(G)$, where $\delta(G) \subseteq V$, enables the discovery of a limited (public) subset of vertices within the SWoT. A connection function, $\lambda(G, u, v, r, \pm)$, adds or removes a relation, r, to the graph G for the edge $\{u, v\}$, where $u \in V$ and $u \neq v$.*

Definition 7.4 (SWoT Information Flow Function)**.** *Considering the SWoT graph $G = (V, E)$, a source vertex $s \in V$ and a set of sink vertices $T \subseteq V$, the task of publishing and transmitting information from source to sink set is performed by the function $\tau(s, T, m)$, where m is the message to be transmitted.*

The quality or weight of relations of the structural dimension of Social Capital motivated information flow is reflected by the relational dimension where repeated interactions among vertices over time accumulates trust and mutual obligations, forming 'social solidarity' (Sandefur and Laumann, 1998). Therefore, transmission from a source to a sink vertex is governed by a flow control function specified in Definition 7.5 which can aid the relevance, timeliness and trustworthiness (Nahapiet and Ghoshal, 1998) of messages in Definition 7.4. The evaluation function, c, and threshold, θ are measures of trustworthiness that work on particular trust models. Nitti et al. (2014) suggested subjective and objective trust models for the SIoT where the subjective model reflects trustworthiness calculated through peer-to-peer relationships while the objective model reflects trustworthiness decided by the entire network. Both methods align with the understanding of Social Capital Theory utilised by the SWoT and can be applied.

Definition 7.5 (SWoT Flow Control Function)**.** $\alpha(s,t) = \{0,1\}$, where $t \in T$, governs the boolean decision to transmit a message from source s to sink t vertex and α includes an evaluation function $c(s,t) = [0,1]$ determining if the value of relational dimension of Social Capital is within a threshold, θ, of range [0,1] as expressed by:

$$\alpha(s,t) = \begin{cases} 0, & \text{if } c(s,t) \leq \theta \\ 1, & \text{otherwise} \end{cases}$$

The cognitive dimension involves speaking common languages and sharing narratives and translates to the issues of interoperability and understanding the features and characteristics of other Things within the IoT. As identified previously in this thesis, the heterogeneity of Things is a critical challenge for the IoT. The idea of cognitive Social Capital however, provides added incentive for researchers, governments and the industry to build semantically interoperable Things which publish rich metadata of themselves. Learning from the experience of the Web, Raggett (2016) explains that web developers initially did not benefit enough from semantic annotations and microformats to justify the effort invested in learning and adding them. However, the promise of instantly getting a better presentation and ranking on search engines drove adoption of common schema.org microformat[10], thus gaining a form of cognitive social capital. This cognitive dimension is implicit within the flow of information as per Definition 7.4.

It follows that there are varying types of information flows within the SWoT. A one-to-one form of propagating messages, m, is called direct messaging, described in Definition 7.6 where the cardinality of the sink set, T, is one such that, $|T| = 1$. A one-to-many form is called sharing, introduced in Definition 7.7, where $|T| \geq 1$.

Definition 7.6 (SWoT Direct Messaging Function)**.** Messaging, $\tau_M(m, s, v_r)$, is a function that sends a direct message, m, from a sender vertex, s to a receiver vertex, v_r. This is a special case of the information flow function, $\tau(s, T, m)$, where the receiver vertex is the only sink vertex, such that $v_r \in T$ and $|T| = 1$.

Definition 7.7 (SWoT Sharing Function)**.** Sharing a message is the task of transmitting or retransmitting from the source or receiver vertex, u, to a sink set of vertices, T, by the function $\tau_s(u, T, m)$, where m is the message received and $|T| \geq 1$.

Sharing messages also entails the retransmission of received messages to a set of one or more sink vertices. This allows the propagation of messages through the social network called diffusion in online social network literature. Studies have shown that the content value of the message, rather than in-degree of source (decided by the structural and relational dimensions of the network), contributed to its fast diffusion through sharing (Cha et al., 2010; Kwak et al., 2010). Hence, a means of feedback to evaluate the content value of messages called liking in the SWoT is provided in Definition 7.8.

[10] https://schema.org/docs/gs.html

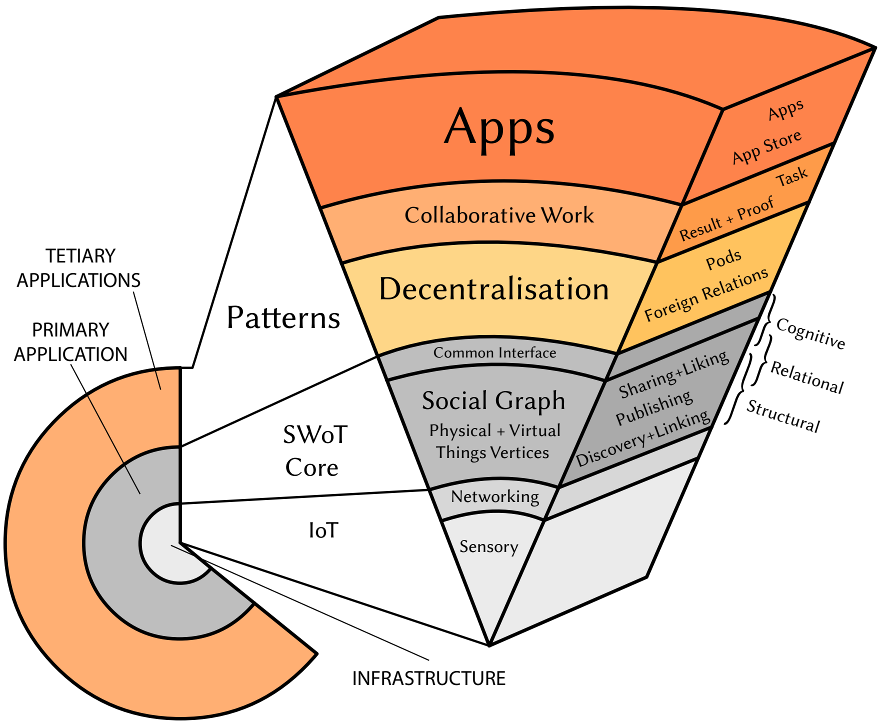


FIGURE 7.6: Layered Design of the SWoT

Definition 7.8 (SWoT Like Scoring Function)**.** Liking is the feedback mechanism of a vertex assigning a like rating, $l = [-1, 1]$, to a message m. A like-score function, $L(m) = \{x \mid -1 \leq x \leq 1\}$, returns the set of likes for a particular message m.

Table 7.1 summarises the SWoT functions presented and related definitions of the social capital inspired design. These can be clearly recognised within steps of building and maintaining the SWoT network, propagating information flow and providing a common interface with feedback mechanisms for information content. Figure 7.6 places the above social capital inspired design of the SWoT core under the segments of the social graph and common interface within a layered SWoT design. This SWoT core is a primary application deployed in the IoT and running on the lower perception and network layers of the IoT infrastructure introduced in the background section, Section 2.1.4. Patterns present general reusable solutions within the tertiary application layer, sometimes called a business layer, working on the core to introduce decentralisation in Section 7.3.1.1, collaborative work in Section 7.3.1.2 and apps in Section 7.3.1.3 for the SWoT.

7.3.1.1 Decentralisation Pattern

Yeung et al. (2009) and Datta et al. (2010) argued that the future of online social networks is decentralised with users having greater control over privacy, ownership and dissemination of their personal data. Mastodon[11] in particular, which styles itself as a

[11]https://mastodon.social/about

Functions	Symbol(s) and Names	Definition(s)
	Building and Maintaining a Network (Structural)	
Core Network	G, R, Relation Function (ρ)	7.2
Structural Dimension	Discovery (δ), Connection (λ)	7.3
	Information Flow (Structural + Relational)	
Information Flow	Transmission (τ), Source (s), Sink (T)	7.4
Flow Control	Decision (α), Evaluation (c), Threshold (θ)	7.5
Direct Messaging	Messaging (τ_M)	7.6
Sharing/Re-sharing	Sharing (τ_s)	7.7
	Feedback and Interoperability (Cognitive)	
Liking Feedback	Liking (L)	7.8
Common Interface	Interoperable Message (m) and Metadata	7.3,7.4

TABLE 7.1: Summary of the Functions of the SWoT

federated Twitter-like social network based on open source technology, has gained much popularity in recent times.

When translated to an IoT disaster management scenario, ad-hoc decentralised networks of Things and people might need to be formed to exchange information, an example of which was when last-mile connectivity in Nepal was disrupted after the earthquake (Madory, 2015). Fog computing, that was introduced in the study of background literature in Chapter 2, presents a greater availability of computing resources near to Things that can be utilised for decentralised SWoT needs. Research literature, like Diaspora (Bielenberg et al., 2012), PIOTRe (Siow et al., 2016b) and Solid (Mansour et al., 2016), also have demonstrated how decentralised pods, servers and repositories can manage and store data independently of the social network creating and consuming it.

A pattern for decentralisation can be built on the SWoT core where localised networks can be deployed on decentralised pods. Definition 7.9 describes a pod in reference to the core SWoT terminology presented previously and summarised in Table 7.1. Foreign relations can exist between vertices in different pods connecting a network of decentralised pods.

Definition 7.9. $G_p = (V_p, E_p)$, where $G_p \subseteq G$, is the graph stored in a pod, a decentralised unit server. Let P be the set of all pods, then $G = \bigcup_{G_p \in P} G_p$. Finally, given a vertex $v \in G_{p1}$ and a vertex $q \in G_{p2}$ in another pod, (v, q) or simply vq is a foreign edge and $\rho(vq) = r$ (Definition 7.2) where r is a foreign relation across pods.

7.3.1.2 Collaboration Pattern

This definition of a collaborative work pattern draws from the decomposition of decentralised blockchains, distributed lists of records linked and secured using cryptography, into abstraction layers by Croman et al. (2016). The *network plane* and *storage plane* of any collaborative work in the SWoT is the underlying IoT infrastructure. The SWoT application with its social graph, G, facilitate vertices performing collaborative work, while the *consensus plane* is defined as follows in Definition 7.10.

Definition 7.10. Let ϖ be a task that requires work, p be the proprietor who creates the task, w be the worker accepting the task and Δ the time bounds for the task. r is the result of the task and the consensus function $C(\varpi, r, \Delta)$ returns a boolean value signifying if the task has been completed by w within the time bounds Δ.

This pattern where $p, w \in V$, allows a basis for vertices to share work through units called tasks ϖ. Consensus provides a means of assuring that work has been done, a general form of the proof-of-work concept (Wood, 2014).

7.3.1.3 Apps Pattern

An app, a, builds on the features of the SWoT core and patterns like collaborative work (lower layers of Figure 7.6) to provide utility to Things or users, make decisions and drive actuators in response. An app can possess its own storage, processing and output external to a pod (Definition 7.9), however, is itself a vertex, $v \in V$, as a virtual Thing within a pod (Definition 7.2).

An app store, A, is a set of apps, $a \in A$, made available to the vertices of a pod. The union of app stores form the app ecosystem.

7.3.2 Human-to-Machine Transition

The previous section on a social capital inspired design helped to establish core functions of the SWoT application with patterns for collaboration and decentralisation, all within the specific context of the IoT. This section presents how, with an understanding of current and proposed future technologies, a transition can be made from an initially human-driven and curated SWoT to a machine-automated one for each function. Given the increasing volume of information the IoT promises, this transition is necessary in the long run as information and the complexity of interactions and relationships increases past the human capacity for comprehension. A social network of human-curated and configured Things might seem contrary to the vision of the IoT but provides a realistic starting point to develop and evolve the applications, information flow and collaborative technologies towards machine-automation.

Figure 7.7 summarises the contribution of this section, in which the vertical axis shows functions of the SWoT, for example, managing the SWoT network or information flow, running apps and collaborating on work while the horizontal axis presents the spectrum

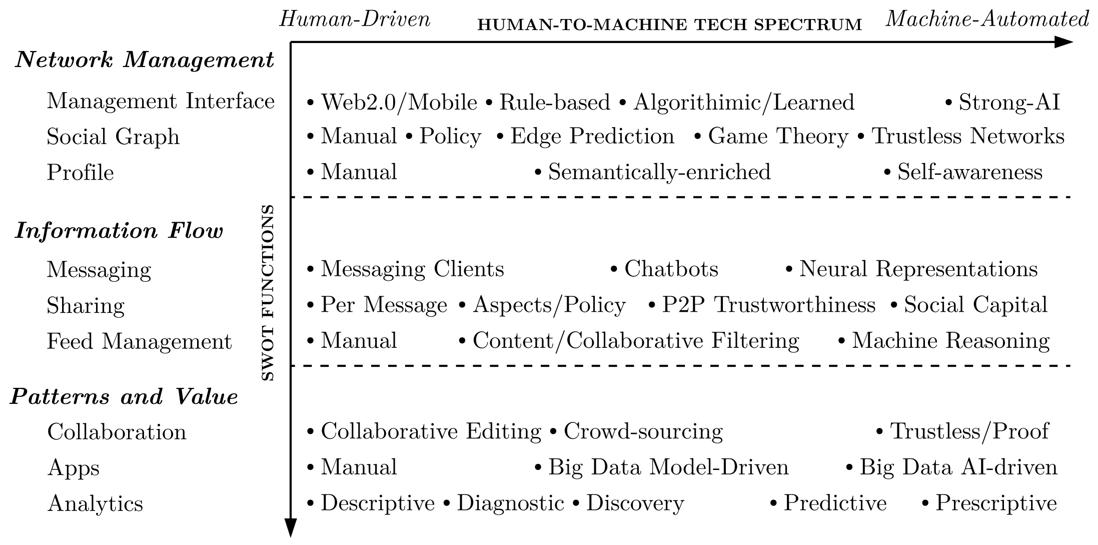

Human-Driven
HUMAN-TO-MACHINE TECH SPECTRUM
Machine-Automated
SWOT FUNCTIONS
Network Management
Management Interface
Social Graph
Profile
• Web2.0/Mobile • Rule-based • Algorithimic/Learned • Strong-AI
• Manual • Policy • Edge Prediction • Game Theory • Trustless Networks
• Manual • Semantically-enriched • Self-awareness
Information Flow
Messaging
Sharing
Feed Management
• Messaging Clients • Chatbots • Neural Representations
• Per Message • Aspects/Policy • P2P Trustworthiness • Social Capital
• Manual • Content/Collaborative Filtering • Machine Reasoning
Patterns and Value
Collaboration
Apps
Analytics
• Collaborative Editing • Crowd-sourcing • Trustless/Proof
• Manual • Big Data Model-Driven • Big Data AI-driven
• Descriptive • Diagnostic • Discovery • Predictive • Prescriptive

FIGURE 7.7: Human-driven to Machine-automated Technology Transition of SWoT Functions

of transition from human-driven to machine-automated technologies. Sections 7.3.2.1 to 7.3.2.3 introduce the technology transition for each group of functions in more detail. This survey of technologies can serve as a research roadmap for various components of the SWoT.

7.3.2.1 Network Management

The interface to manage and interact with the SWoT starts its transition from one heavily dependent on human-computer user interfaces with Web 2.0 and mobile front-ends where users control the management of networks of Things and each interaction between Things, for example the vision presented in Ericsson's UX Lab Demo[12]. This proceeds to semi-automated rule-based approaches (e.g. if-then-else rules[13]) and then to automated agent-based approaches including algorithmic (e.g. sub-gradient methods for multi-agent optimisation (Nedić and Ozdaglar, 2009)) and learning (e.g. adaptive agent (Sayed, 2014)) technologies. Finally, a transition to a strong Artificial Intelligence (AI) approach (Copeland, 2015) to manage all social network interaction.

The Social Graph, G, from Definition 7.2, transitions from relations being formed by manual human-curation (e.g. searching, requesting and accepting friend/follower requests), to those formed by human-defined policy-based approaches (Bonatti et al., 2005), to automated approaches of edge prediction using various machine learning techniques to predict positive and negative links (Leskovec et al., 2010a; Backstrom and Leskovec, 2010). Game theory utilised to model Things as selfish agents can also be used to build or maintain the social graph as a large network at nash equilibrium, a

[12]http://www.ericsson.com/uxblog/2012/04/a-social-web-of-things/
[13]https://ifttt.com/

state from which no agent has an incentive to deviate (Anshelevich et al., 2003). Finally, trustless networks can be a distributed, peer-to-peer means of forming the social graph where nodes trust only contracts, proof-of-work and payments in their interactions (Christidis and Devetsikiotis, 2016).

A Thing's description within its profile transitions from being user-curated to automatically semantically-enriched (e.g. adding a semantic annotation of the location of a new sensor by correlating to an existing sensor with similar output (Pfisterer et al., 2011)). Self-awareness is furthest along the spectrum as it represents the ability of Things to form representations about themselves (Copeland, 2015), learn and modify their profile accordingly and project this understanding to interpret other Thing's profiles.

7.3.2.2 Information Flow

Messaging and communication between Things can initially be driven by administrators of Things through user interfaces like messaging clients before transitioning to Chatbots which utilise a natural language representation and computer vision as a generic interface with increased interoperability. When coupled with a knowledge base, domain-specific Chatbots have been shown, through reinforcement learning, to improve performance within noisy conditions (Dhingra et al., 2017). Finally, results in the domain of end-to-end Neural Machine Translation (Johnson et al., 2016) have hinted that it is possible for machines to internally develop and learn a universal interlingua representation. A transition to such neural representations promises fully machine automated communication between Things.

The propagation of information to other vertices is through a mechanism called Sharing which transitions from each message, m, being individually shared through a human decision, to a semi-automated approach that utilises a human-configured policy to share information to groups of Things (e.g. Aspects[14] within Diaspora). Finally, in a machine-automated environment, it is possible to utilise peer-to-peer trust models (Nitti et al., 2014) or cost-based models to decide how to maximise social capital for each interaction.

The feed is a stream of all information received from the network. Feed management transitions from an unfiltered feed and manual curation of all Thing events through a user interface, to a filtered feed that utilises techniques like content-based recommenders (Lops et al., 2010), that exploit feedback mechanisms like likes to determine value, or collaborative filtering, that exploits a shared understanding of value for recommendations (e.g. it has been shown that *familiarity* rather than *similarity* provides superior recommendations (Guy et al., 2009)). Finally, as machine reasoning develops, Things and apps would be able to identify what events to store and process (Bottou, 2014).

[14]https://diasporafoundation.org/about#aspects

7.3.2.3 Patterns and Value

Collaboration transitions from user interface based, human-driven collaborative editing of Things like bringing together web mashups and dashboards of Things data (Guinard and Trifa, 2009) or creating a consistent environment for co-creating apps (Oster et al., 2006), to crowd sourcing systems that harness the collective intelligence of crowds (Malone et al., 2009) with increasing reliability and resilience to unreliable individual sources (Qi et al., 2013), through forming a probabilistic model of group-level reliability, like within the SWoT social graph, for true value inference. Further down the spectrum, trustless networks that harness concepts like proof-of-work can form a basis for machine-automated collaborative applications running on a decentralised, singleton compute resource, like that envisioned in Ethereum by Wood (2014). However, there is also a need to improve the performance and scalability of trustless networks.

Applications (apps) transition from user input-output driven apps which utilise input from Things, a feedback loop to the user and output to the actuators, to model-driven big data apps (e.g. early warning, real-time monitoring and real-time feedback apps (Chen et al., 2014)) which provide higher value output that can be combined with analytics to automate actuation. Finally, big data-driven artificial intelligence apps can analyse the content of messages, m, and structure of the graph, G, to automatically drive systems like Intelligent Transport Systems and Smart Cities (Al-Fuqaha et al., 2015). Section 2.3 explains the transition of analytics used in apps from lower-value descriptive, diagnostic and discovery analytics to higher-value predictive and prescriptive analytics.

7.3.3 Publish-Subscribe Infrastructure for the SWoT

The SWoT application gives rise to a set of requirements for implementing its core functions. From the building of the social graph, management of the network and information flow, to the functioning of apps, a mechanism for transmitting information between nodes and for storing and processing information is required. This thesis previously discussed Map-Match-Operate that improves the efficiency and interoperability of information storage and processing for IoT sources. However, a mechanism for transmitting information within the SWoT is still required. Hence, different publish-subscribe mechanisms and protocols for the dissemination of information within a network are considered to overlay the Map-Match-Operate data storage and processing technology within the SWoT infrastructure.

Definition 7.4 describes the publishing of a message, m, from a source (publisher) to a set of sink vertices (subscribers). The performance of a publish-subscribe system that allows the transmission of information from the source to sinks can be measured by the average latency from publishing to receiving messages. Two state-of-the-art publish-subscribe application protocol technologies developed for the IoT, MQTT[15] and deepstream μWebSockets[16], were compared for performance in a set of experiments.

MQTT is a publish-subscribe (pub-sub) transport protocol for the IoT and was designed to be lightweight. It is one of the commonly used protocols in the IoT and has been shown to perform better than other popular protocols like CoAP for low packet loss conditions (Thangavel et al., 2014). The MQTT broker used was Mosquitto[17], an open-source Eclipse project. Deepstream is a real-time server for the IoT that supports pub-sub using μWebSockets, shown to significantly outperform[18] popular competitors like Socket.IO and WebSockets and is suggested for IoT usage.

Given the decentralisation pattern described in Section 7.3.1.1, smaller-scale networks with less than 10,000 vertices were considered. In the absence of any published SIoT network structure data, two published social networks of different sizes were considered:

1. arXiv General Relativity, a sparse, undirected collaboration network with 5,242 nodes and 14,496 edges (Leskovec et al., 2007).
2. Wikipedia who-votes-on-whom, a dense, directed network with 7,115 nodes and 103,689 edges (Leskovec et al., 2010b).

As the SWoT social graph and message transmission also involve a relationship logic (RelLogic) component, whereby each transmission is subject to flow control expressed in Definition 7.5, two possible pub-sub system designs are possible: a selective-subscribe and a selective-publish system. Both these systems are explained as follows and were compared within the experiments.

Figure 7.8 shows both pub-sub systems, each with a set of publisher nodes 1, 2 and 3 alongside a set of subscriber nodes 4 and 5. In this example, subscriber 4 has a *follows* relationship with publishers 1 and 2, while subscriber 5 has a *follows* relationship with publishers 2 and 3. In both systems, each publisher, nodes 1, 2 and 3, publishes to its own topic, 1, 2 and 3 respectively. This process where a publisher publishes to a topic is represented in the figure by the $p(x)$ function, where x is a topic number. Subscribers subscribing to a topic is represented by $s(y)$, where y is also a topic number.

In the *selective-subscribe* system, subscribers, after authenticating with the server, obtain a list of topics to subscribe to based on relations they have in the RelLogic component e.g. subscriber 4 which *follows* publishers 1 and 2, therefore, subscribes to topics 1 and 2. The *selective-publish* system consists of publisher nodes publishing to a RelLogic component which then publishes the messages to each of the publisher's relations topics

[15]http://mqtt.org/
[16]https://deepstream.io/
[17]https://mosquitto.org/
[18]https://github.com/uWebSockets/uWebSockets

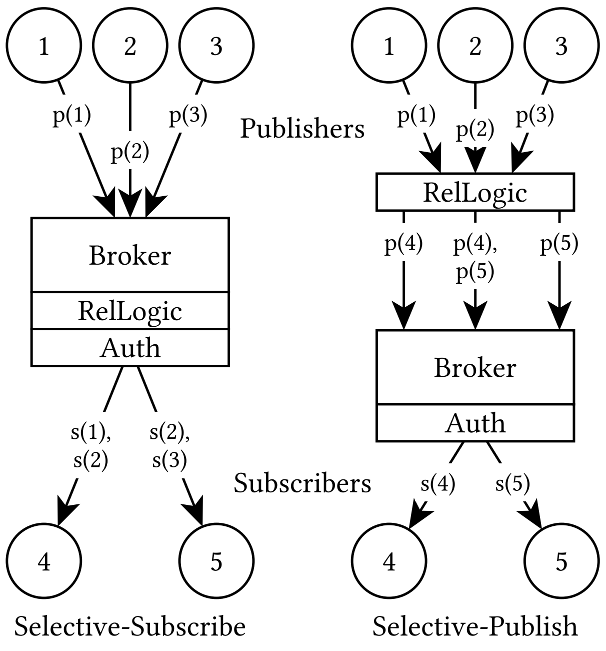


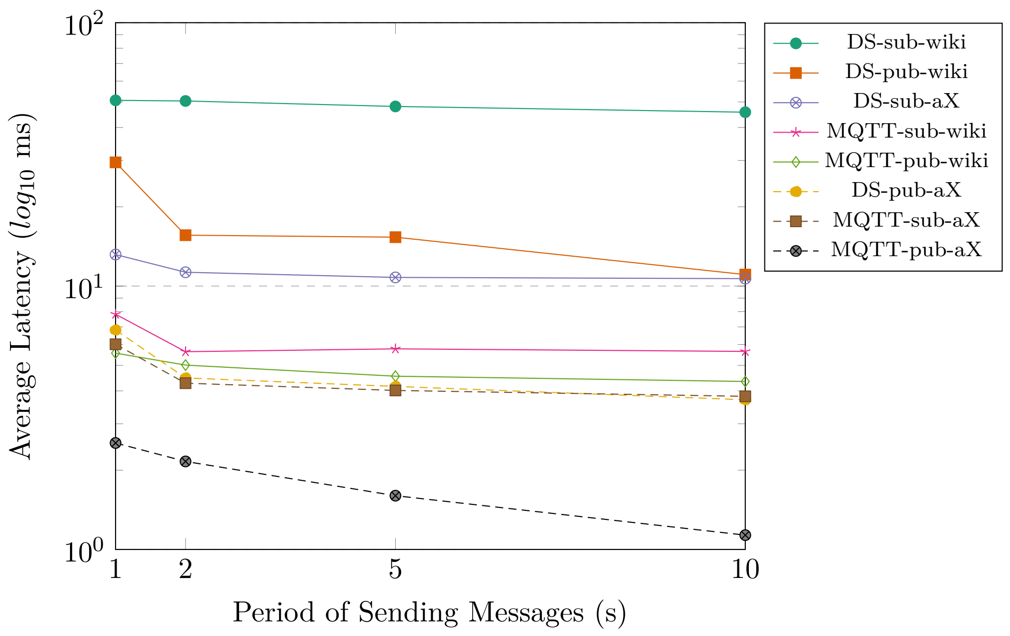
FIGURE 7.8: Selective-Subscribe and Selective-Publish Pub-Sub Systems

e.g. publisher 1 publishes a message to subscriber 4's topic (as subscriber 4 is a *follower*), while publisher 2 publishes to subscriber 4 and 5's topics (its followers). Each subscriber subscribes to its own corresponding topic e.g. subscriber 4 to topic 4. The end result for both systems, in terms of message delivery, is the same. Subscriber 4 receives messages from publishers 1 and 2, while subscriber 5 receives messages from publishers 2 and 3.

The experimental setup included a decentralised server and broker for the SWoT that contained the authentication and RelLogic components. This was hosted on a compact Intel Next Unit of Computing (NUC) platform with dimensions of 4 by 4 by 1.5 inches and contained an i5 processor and 8GB RAM. Publisher and subscriber nodes were run on Raspberry Pi 2 Model B+s (RPi) with 1GB RAM, a 900MHz quad-core ARM Cortex-A7 CPU and Class 10 SD Cards. Ethernet connections were used between the RPi nodes and the server. Due to resource constraints, multiple publishers and subscribers had to be run on the RPi devices as separate threads. Section 7.3.3.1 presents and discusses the results of the experiments, while the design of the RelLogic component is improved in Section 7.3.3.2 by using ideas from TritanDB's ingestion pipeline.

7.3.3.1 Selective Publish-Subscribe Comparison

Figure 7.9 shows the results of the experiments where the vertical axis displays a logarithmic scale of the average latency of receiving fixed size messages with a 32 byte payload across all client nodes while the horizontal axis shows different periods of sending messages e.g. each node sends a message every 5 seconds.



Protocol: DS = Deepstream μWebSockets, **System:** sub = Selective-Subscribe, pub = Selective-Publish, **Network:** wiki = Wikipedia (dense), aX = arXiv (sparse)

FIGURE 7.9: Comparison of Selective-Pub-Sub Systems by Average Latency of Message Transmission

The results can be summarised as such:

- selective-publish performs better than selective-subscribe (has lower average latency from 1.2x to 4.2x) as the connection overhead of many subscribers on a client causes delay and higher latency,
- MQTT performs better (from 2x to 49x) than deepstream μWebSockets with both the sparse arXiv graphs and dense wiki graphs and
- as the period of sending messages increases, as expected, average latency decreases, however, both deepstream and MQTT have sub 50ms average latencies at rates of up to 7,115 messages per second (about 427,000 messages in a minute).

7.3.3.2 Scaling Selective-Publish RelLogic

Although the comparison in the previous section showed that selective-publish systems performed better, they also required that the RelLogic component in Figure 7.8 was able to handle the volume of messages published.

Hence, a high performance queue to process messages being published to RelLogic, lookup the relevant topics to forward to and connect and forward the MQTT messages is required. Hence, three designs are compared. A RelLogic component that implements the disruptor pattern used in high-frequency trading and utilised in TritanDB's ingestion pipeline (Section 5.3.1) is compared to a single threaded and a multi threaded queue.

Experiments are performed at increasing velocities of messages per second, from 1,000 to 10,000. Messages are addressed from a random node to the corresponding set of subscribers from both the arXiv and wikipedia networks. Each experiment is repeated and averaged over 10 runs on the same server as with the previous benchmark. A Java LinkedBlockingQueue is used for the single-threaded implementation and a Java ConcurrentLinkedQueue and AtomicIntegers (for averaging latency) are used for the multi-threaded queue. Figure 7.10 visualises the 3 types of RelLogic queues tested.

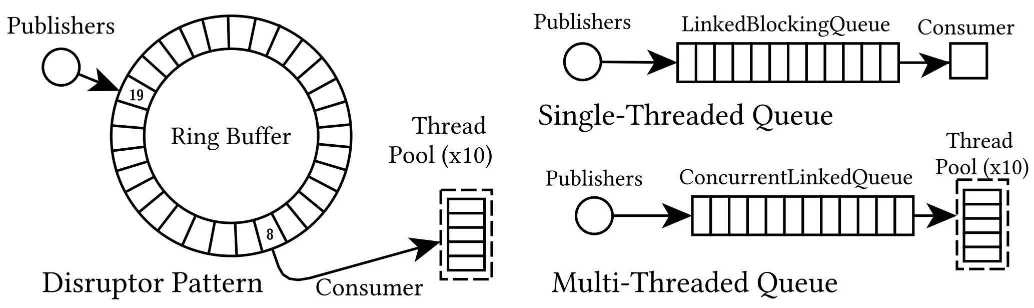


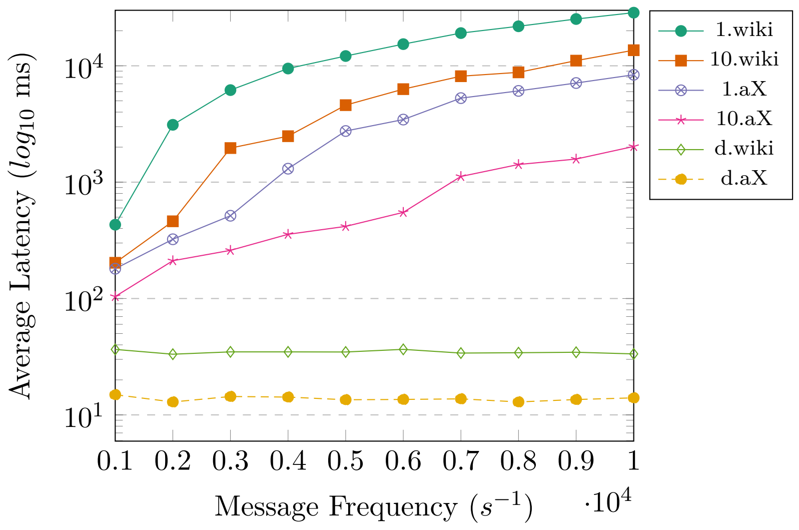
FIGURE 7.10: Visualisation of the RelLogic Queue Implementations

Figure 7.11 shows the results of the wikipedia network (**.wiki*) and arXiv (**.aX*) network benchmarks of the disruptor (*d.**), single threaded (*1.**) and multi-threaded (*10.**) queues. The performance of the disruptor selective-publish servers are from 12 to 800 times faster (have a lower average latency) than the single-threaded queue servers and from 5.5 to 400 times better than the multi-threaded servers. Furthermore, as message frequency increased, average latency remained stable, displaying scalability to increasing message velocity.

7.3.3.3 Lightweight and Scalable Pub-Sub for the Decentralised SWoT

The experimental setup emphasised the use of compact platforms for decentralised deployments of the SWoT which display the lightweight nature of the selective-publish MQTT system, including the RelLogic Disruptor pattern. Even on the more dense wikipedia network with more than 7,000 nodes and 100,000 edges, sub 10ms average latencies could be achieved. These results also show that smaller distributed networks can be supported on fog computing resources like gateway routers and hubs.

Increasing the velocity of messages on both sparse and dense networks up to 10,000 messages per second could also be handled without increasing the average latency with the selective-publish RelLogic component using the Disruptor pattern. Increasing the number of edges about 7 times from sparse to dense networks, results in only a 2.5 times increase in average latency for RelLogic and a maximum of 3.8 times with the selective-publish system. Furthermore, performance can be further enhanced by deploying a



Queue: 1 = Single-threaded, 10 = Multi-threaded, d = Disruptor, **Network:** wiki = Wikipedia, aX = arXiv

FIGURE 7.11: Comparison of Single-threaded, Multi-threaded Queues and Disruptor Pattern RelLogic Servers

Hubber Component	Technology
Broker	Selective-Publish MQTT
RelLogic	Disruptor Pattern
PIOTRe	Map-Match-Operate TritanDB
Auth	OAuth 2.0
GraphDB	Neo4j, W3C Things Description
Foreign Relations	Internationalised Resource Identifiers (IRIs)

TABLE 7.2: Technologies used in Hubber.space

load balancer and bridge or replicating across multiple MQTT brokers to build a high-availability cluster. The experiment source code is available in an online repository[19].

7.3.4 Hubber.space: An Experimental SWoT

Hubber.space[20] is an experimental sandbox application developed to realise the core functions of the SWoT, integrating the selective-publish MQTT broker and RelLogic components in a server, with a data management layer using Map-Match-Operate to store data on nodes. Figure 7.12 shows a component diagram of a decentralised Hubber server, while Table 7.2 shows the technologies employed in each component.

Each decentralised Hubber server contains a selective-publish MQTT broker and a RelLogic component implemented using the disruptor pattern to handle messages, m. The SWoT social network is managed as a graph stored within a graph database like Neo4j (Robinson et al., 2015) and each vertex is described by a W3C Things Description[21]

[19]https://github.com/eugenesiow/hubber-bench-js
[20]http://hubber.space
[21]https://w3c.github.io/wot/current-practices/wot-practices.html#thing-description

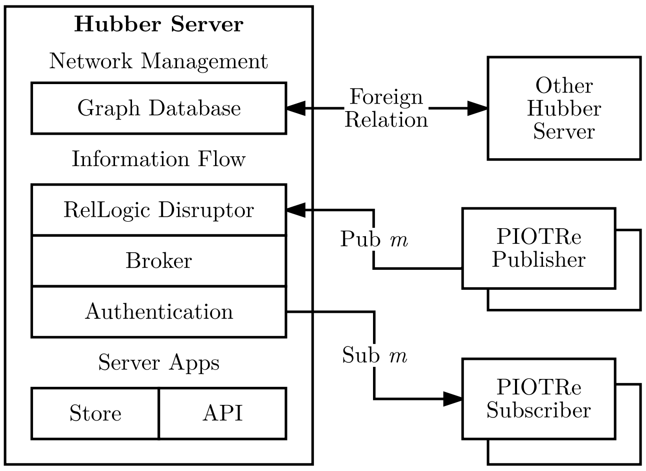
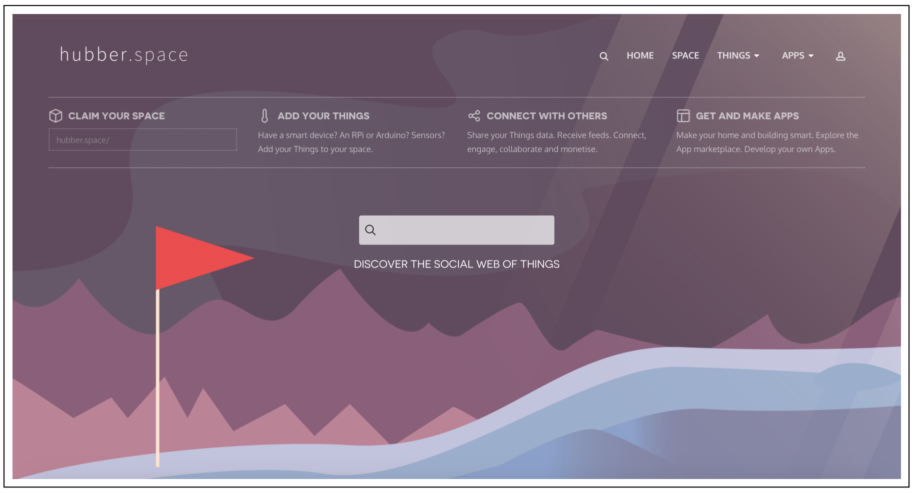


FIGURE 7.12: Component Diagram of a Hubber Server and PIOTRe Nodes



Have a smart device? An RPi or Arduino? Sensors? Add your Things to your space.

Share your Things data. Receive feeds. Connect, engage, collaborate and monetise.

Make your home and building smart. Explore the App marketplace. Develop your own Apps.

FIGURE 7.13: Screenshot of the Hubber.space Landing Page

for interoperability. Foreign relations can be retrieved from other Hubber servers by accessing their IRIs. The information flow components sit behind an OAuth 2.0 authorisation barrier[22]. Apps can be downloaded by nodes from the server App Store. Apps on nodes can also interact with app code running on the server through an API. Nodes can run PIOTRe (Section 7.1) which allows the subscribed messages that include sensor observations and events from the social network to be stored as time-series with a Map-Match-Operate TritanDB engine.

Figure 7.13 shows a screenshot of a decentralised Hubber.space app. The purpose of this deployment was as a seed for users, vendors, developers and machines to co-create the SWoT and start utilising PIOTRe on personal repositories for their IoT data.

[22]https://oauth.net/2/

7.4 Conclusions

This chapter introduced PIOTRe, an application for managing personal IoT repository data and applications, and the SWoT which builds a social network on PIOTRe repositories and other Things through a social capital inspired design. Features like resampling and moving averages that allow TritanDB to perform time-series analytics like predictive forecasting were also introduced. Map-Match-Operate for historical and streaming data management and processing promises greater interoperability and efficiency for the IoT. These applications and platforms go towards promoting adoption and providing a basis to build useful applications which add value to the current and future IoT. Each of these applications was designed and tested on resource-constrained fog computing hardware but also scaled to work in cloud environments, by virtue of TritanDB and S2S, providing better performance and enabling them to handle larger volumes of data.

Chapter 8

Conclusions

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

— *Alan Turing*

This chapter presents a summary and a recap of the contributions of this thesis in Section 8.1. Limitations and opportunities for future work are also discussed in Sections 8.2 and 8.3 before final conclusions are made in Section 8.4.

8.1 Summary

This thesis started by asking the question of how database systems can support the efficient and semantically interoperable storage and retrieval of Internet of Things (IoT) data and metadata, both historical and real-time, for use in analytical applications. The direction taken was to bring together the unique characteristics and requirements of IoT time-series data, established through a series of studies, with an understanding of database and stream processing system design and innovations. The methodology employed throughout was to investigate the state-of-the-art and improve it for the IoT through formal theoretical definition, then practical implementation and finally experimental comparison using well-defined benchmarks, datasets and scenarios.

Chapter 2, a study of background literature of the IoT and database systems, established the need for niche database and stream processing systems within the processing layer of the IoT to support applications and analytics across diverse hardware. The chapter contributed an overall understanding of the IoT landscape divided by applications, platforms and building blocks, from an analysis of survey literature, before focusing on more detailed studies of specific building blocks and technologies like time-series databases, Resource Description Framework (RDF) graph databases, stream processing, fog computing and analytics within IoT applications.

An analysis of public IoT time-series data across application domains in Chapter 3 showed that a large proportion of schemata exhibited the unique characteristics of being flat, wide and numerical. It was also established that both periodic or approximately periodic and non-periodic time-series data were present in streams. These characteristics affect the efficient storage and retrieval of data. In addition, the chapter also contributed a study of metadata models for the IoT showing a diversity taken in approaches to model heterogeneous data from different domains. Standards bodies and research initiatives though, converged on approaches that could be generalised to a graph or tree representation for metadata. However, a follow-up study of metadata and data ratios in graph model IoT data of the Resource Description Framework (RDF) format highlighted the issue of metadata expansion, resulting in inflated data sizes. Efficiency and compactness was traded-off for interoperability and flexibility.

Building on the understanding of the characteristics of IoT time-series data and motivated by the inefficiency of graph model RDF for IoT data, a method for query translation called Map-Match-Operate that abstracted graph queries to be executed across graph model metadata from the data representation of time-series data was devised in Chapter 4. The implementation of this method across RDF graph metadata and time-series data in relational databases showed from two to three orders of magnitude performance improvements, comparatively better aggregation support and fewer joins when compared against a range of databases and translation engines.

The physical interface for storing and retrieving time-series data from hardware is one-dimensional while relational databases present a conceptual two-dimensional interface. Hence, this discrepancy between physical and conceptual representations was identified as an area of further performance optimisation. A new compressed, time-reordering, performance-optimised data structure for the storage of IoT time-series data, TrTables, was proposed in Chapter 5 as a result of an evaluation of time-series database compression algorithms and storage structures. A Map-Match-Operate database system, TritanDB, was implemented to use TrTables and performed better in storage and processing experiments compared against a range of time-series, relational and NoSQL databases on both cloud and Thing hardware.

Chapter 6 investigated stream processing systems and implemented Map-Match-Operate to efficiently translate graph queries for continuous execution on real-time streams of IoT data. Map-Match-Operate for streams was then integrated with a fog computing infrastructure for distributed stream processing, Eywa. Eywa introduced a novel inverse publish-subscribe approach to distribute queries within the control plane and employed predicate pushdown in the data plane, of the projection operator, to distribute the workload among co-operative fog computing nodes.

Finally, Chapter 7 described applications built on Map-Match-Operate's performant and interoperable IoT storage and querying. This included a demonstration of both streaming and historical smart home application dashboards running on PIOTRe, a personal repository for IoT data, deployed on a compact Raspberry Pi device. A social capital theory inspired Social Web of Things application, that provided social network functions to manage the information flow between PIOTRe instances and other Things, was also introduced. Features like resampling and moving averages that allow TritanDB to perform time-series analytics like predictive forecasting were also discussed.

Thus, the contributions presented in this thesis, from identifying that IoT data has a set of unique characteristics, to using these characteristics to improve the performance of graph queries on databases and streams, represent an advance in the state-of-the-art for the efficient storage and retrieval of a semantically interoperable representation of IoT data and metadata for analytical applications. Despite these advances, however, certain limitations exist, while more opportunities have also emerged. The next two sections suggest a number of promising directions for future research.

8.2 Limitations and Future Work

This thesis focused on an understanding of *current* time-series IoT data characteristics, *a particular graph model* for databases and stream processing systems, RDF, where most processing is done on a *single node* whether in cloud or cooperative fog computing scenarios on distributed, resource-constrained fog nodes. This section recognises some limitations in the solutions proposed and suggests extensions and future work.

8.2.1 Multimodal Data

Although a large proportion of current IoT data from sensors conformed to flat, wide and numerical characteristics of time-series, there is a case for providing for a variety of forms of multimodal data in future. Increasingly, images, sounds and unstructured text data might become inputs to the IoT and the storage, integration and processing of such data will gain more importance. More complex hierarchical schemata might also become more common. Research in compression and storage structures can be combined with techniques for extracting feature sets and the machine-understanding of such data.

8.2.2 Other Graph/Tree Models and RDF Property Paths

RDF is considered as the example of a graph data model in this thesis due to the presence of a large body of research work, data availability and its application within the IoT. JSON, however, is increasingly becoming the data-interchange format of choice within the IoT and semantic interoperability discussions within groups like the Internet Engineering Task Force (IETF) are centred around data models that utilise annotations like JSON Schema[1]. Hence, a future direction for work is to extend (or limit, since the tree is a limited form of a graph) Map-Match-Operate to work on JSON-based models.

Another limitation of Map-Match-Operate in this thesis is that path queries[2], queries that specify a chain of predicates and predicate cardinality (arbitrary number of hops), across tiers are not supported at the moment. There is an established theoretical basis for such queries both in SPARQL (Kostylev et al., 2015) and in other graph databases (Reutter, 2013), however, the feasibility of query translation across the model and database tiers is an area of future research.

8.2.3 Horizontal Scalability with Distributed Databases

Horizontal scalability allows a database system to increase capacity by connecting together multiple instances across hardware entities so that the system works as a single logical unit. Many of the NoSQL databases compared against TritanDB like Cassandra and OpenTSDB support horizontal scalability to handle larger volumes of data. Hence, an area for future research includes investigating the feasibility of partitioning time-series data across instances to extend TritanDB for horizontal scalability in the cloud or across fog computing networks with both cooperative and uncooperative nodes.

8.3 Opportunities and Future Work

This section recognises some opportunities for future work that open up due to the contributions of this thesis.

8.3.1 Eywa Fog Computing Operators

Eywa formally defined an infrastructure for *stream query delivery* to source nodes, *distributed processing* and *results delivery* to the client nodes in Section 6.2. Since Eywa provides a means of distributing queries and performing workload coordination in the control plane of a fog network, it opens up the opportunity for research on how various workload operators, besides projections, can be processed in the data plane between the source and client nodes. Query optimisation in the absence of accurate statistics at the client nodes (an issue also experienced in Federated SPARQL, described in Section 2.2.4) and the appropriate division of resources at source nodes are challenges to consider in this future work on operator distribution.

[1]http://json-schema.org/
[2]https://www.w3.org/TR/sparql11-query/#propertypath-syntaxforms

8.3.2 Social Web of Things Knowledge Graph

A knowledge graph that contains entities and relations within the Social Web of Things (SWoT) introduced in Section 7.3 can be formed by integrating mappings of Thing metadata provided as S2SML (Section 4.2.1.1) for Map-Match-Operate. This opens up an area of future work for research on and applications utilising the integrated SWoT knowledge graph.

8.4 Final Remarks and Conclusions

The IoT has huge potential to provide advanced services and applications across many domains and the momentum that it has generated, together with its broad visions, make it an ideal ground for pushing technological innovation and generating value for the society, economy and environment. Data management and processing that supports descriptive, diagnostic, discovery, predictive and prescriptive analytical capabilities will contribute to the growth of the IoT as various stakeholders seek solutions to the problems posed by growing volumes and velocities of information.

This thesis considered the requirements of performance and interoperability when handling the increasing amount of streaming data from the IoT, with an understanding of the unique characteristics of IoT time-series data, while building on advances in RDF graph databases, query translation and time-series databases for telemetry data. The methods and novel designs proposed, Map-Match-Operate, TritanDB and Eywa, all showed improvements in storage and query performance, and scalability when compared with state-of-the-art database and stream processing systems. The results also showed that there is no need to compromise interoperability or performance when using these technologies for the specific use case of IoT time-series data. There was also a focus on benchmarking performance on resource-constrained hardware serving as fog computing nodes in IoT scenarios - base stations, hubs, gateways and routers. In particular, stream processing was shown to be a performant option for analytical querying on resource-constrained nodes, and the performance was further improved by workload distribution within a cooperative fog computing network.

Apart from these, the thesis also looked beyond the immediate performance and interoperability of the solutions proposed, showing how the solutions could lead to applications and platforms in different directions: personal dashboards and repositories, time-series analytics and a Social Web of Things.

Each of the smart home, smart factory and smart city scenarios, presented in the introduction to this thesis, seeks to benefit people by enhancing their daily experiences and optimising the machines and systems that deliver services to them. The work in this thesis is ultimately an endeavour in that direction.

Appendix A

Survey of IoT Applications

Table 2.2 in Chapter 2 summarises 32 IoT application papers by application domain or theme and states the respective techniques used to derive insights and the currency of their data, whether real-time streaming data was used or historical data stored in a database was utilised. More detailed descriptions of the papers are provided in this appendix as follows.

A.1 Health: Ambient Assisted Living, Medical Diagnosis and Prognosis

Mukherjee et al. (2012) surveyed the use of data analytics in healthcare information systems. One area in healthcare where analytics had been applied to was Ambient Assisted Living (Dohr et al., 2010). The solution proposed in the application applied rules to IoT data collected from smart objects in the homes of elderly or chronic disease patients while also considering contextual information and applying inferencing using ontologies to give health advisories to users, update care-givers or contact the hospital in emergencies. In their work, Henson et al. (2012) used abductive logic, 'inference to the best explanation', to provide an intelligent diagnosis for a given set of symptoms. Rodríguez-González et al. (2012), for a similar use case, used descriptive logic combined with knowledge from an ontology to provide their medical diagnosis with inputs from multiple IoT agents. Hunink et al. (2014) went a step further in providing prognosis, the science of predicting the future medical condition of a patient, to help doctors and medical staff make more informed decisions by analysing certain key health indicators of patients, comparing these with records from similar patients and combining this with domain knowledge and medical research to make conjectures about potential medical conditions.

IoT agents like wearables collect a large volume of high dimensional health data. Li et al. (2013) applied the Random Forests algorithm, a data mining technique for predicting classes or dependent variables from high dimensional data (125 risk factors for health expenditure), to health data to provide the input for a healthcare costing model.

A.2 Industry: Product Management and Supply Chain Management

Vargheese and Dahir (2014) proposed a system that improved shoppers' experience by enhancing the 'On the Shelf Availability (OSA)' of products. Furthermore, the system also looked at forecasting demand and providing insights on buyers' behaviour. IoT sensors captured imagery of inventory and these video streams were analysed and cross-verified with other light, infra-red and RFID sensors. This real time data was combined with models from learning systems, data from enterprise Point of Sale (POS) systems and inventory systems to recommend action plans to maintain the OSA of products. The staff of the store were informed and action was taken to restock products. Weather data, local events and promotion details were also analysed with the current OSA and provided demand forecasting and modelled buyers behaviour which was fed back into the system.

Nechifor et al. (2014) described the use of real-time data analytics in their cold chain monitoring process. Trucks which were used for transporting perishable goods and drugs required particular thermal and humidity conditions. Sensors measured the position and conditions in the truck and of each package while actuators - air conditioning and ventilation were controlled automatically to maintain these conditions. On a larger scale, predictions were made on delays in routes and when necessary to satisfy the product condition needs, longer but faster routes with less congestion were selected.

Similarly, Verdouw et al. (2013) examined supply chains - the integrated, physical flow from raw material to end products with a shared objective, and formulated a framework based on their virtualisation in the IoT. These virtual supply chain for the Dutch floricultural sector supported intelligent analysis and reporting. Early warning signals could be sent in case of disruptions or unexpected deviations in the supply chain. In this example and the previous example, Complex Event Processing systems were used to process, analyse and act on the real-time data stream.

Robak et al. (2013) also looked at supply chain management from the perspective of a Fourth-party logistics provider (4PL) that acts as an integrator of resources and technologies to build supply chains. The tracking data gathered by the the 4PL was used to describe what was happening along the transport routes and when combined with information from social media and weather reports, helped users to forecast order and transport volumes, diagnose delays and optimise transport plans.

Engel and Etzion (2011) applied a Markov decision process (MDP) (Bellman, 1957) to an event processing network for supply chain planning and optimisation, to deliver a critical shipment in time, avoiding penalties and handling delays.

A.3 Environment: Disaster Warning, Wind Forecasting and Smart Energy Systems

This combination of sensor and social media data was applied to a haze detection scenario by Yue et al. (2014) in their work that used data mining techniques to detect spikes in social media volume on the haze and processes and combined it with IoT sensor data to provide geospatial haze observations.

Schnizler et al. (2014) described a disaster detection system that worked on heterogenous streams of IoT sensor data. Their method involved Intelligent Sensor Agents (ISAs) that detected anomalies, low level events with location and time information, for example, an abnormal change in mobile phone connections at a ISA in a telecom cell or base station, a sudden decrease in traffic, increase in twitter messages, change in water level or change in the volume of moving objects at a certain location. These anomaly events then entered Round Table (RT) components that fused heterogenous sources together by mapping them to a common incident ontology through feedback loops that involve dcrowdsourcing, human-in-the-loop or adjusting parameters of other ISAs to find matches. The now homogenous incident stream was then be processed by a Complex Event Processing (CEP) engine to complete the situation reconstruction by doing aggregation and clustering with higher-level semantic data, simulation and prediction of outcomes and damage. The resultant incident stream was then used to provide early warning and detection of disaster situations as well as providing better situational awareness for emergency response and relief teams.

Another environmental application was wind forecasting (Mukherjee et al., 2013). Data was collected from wind speed sensors in wind turbines and an Artificial Neural Network (ANN) was used on this historical data over time to perform wind speed forecasting.

Ghosh et al. (2013) implemented a localised smart energy system that used smart plugs and data analysis to actively monitor energy policy and by performing pattern recognition analysis on accumulated data, additional opportunities to save energy were spotted. This resulted in saving on electricity bills especially by reducing the amount of power wasted in non-office hours from appliances.

Similar work by Alonso et al. (2013) worked on using machine learning and an rule-based expert system to provide personalised recommendations, based on energy usage data collected in Smart Homes, that helped a user to more efficiently utilise energy. They went one step further in also providing recommendations through predicting cheaper options by detecting similar patterns in big data collected from a large set of homes. Ahmed (2014) also applied similar analysis on combined consumption data for use in organisations to help in energy policy planning. The model developed helped to classify the energy efficiency of buildings and the seasonal shifts in this classification. By also using more detailed appliance specific data, the model was able to forecast future energy usage.

A.4 Living: Public Safety and Security, Lifestyle Monitoring, and Memory Augmentation

On the issue of public safety and security, McCue (2006), gave examples of how data mining of crime incident data, including spatial-temporal information collected from sensors helped the police to device a targeted deployment strategy which increased the recovery of illegal weapons and decreased incidents of random gunfire while reducing the total number of officers that needed to be deployed. Using visualisation, possibly hostile surveillance activity was detected around a facility and it was noticed that there was a spatial-temporal specificity to it, surveillance was done on a specific day of the week and around a particular part of the building (McCue, 2005).

Visualisation that taps the human cognitive ability to recognise patterns was also employed by Razip et al. (2014) in helping law enforcement officers to have increased situational awareness. Officers were equipped with mobile devices that tapped into crime data and spatial and temporal sensor data to show interactive alerts of hotspots, risk profiles and on demand chemical plume models.

Additionally, there were also public safety and military applications that apply video analytics in detecting movement, intruders or targets. The public safety use case was elaborated on by Gimenez et al. (2012) where they discuss how given the big data problem of having huge amounts of video footage, smart video analytics systems can proactively monitor, automatically recognise and bring to notice situations, flag out suspicious people, trigger alarms and lock down facilities through the recognition of patterns and directional motion, recognising faces and spotting potential problems by tracking, with multiple cameras, how people move in crowded scenes.

Guo et al. (2011) looked at discovering various insights from mining the digital traces left by IoT data from cameras to wearables, to mobile phones and smart appliances. The resulting life logging application helped to augment human memory with recorded data, real world search for objects and interactions with people. The system was also used to improve urban mobility systems by studying large-scale human mobility patterns.

Mukherjee and Chatterjee (2014) presented an application of a fast algorithm for detecting anomalies and for classifying high dimensional data to recognise human activity in real-time and classify activities from sensor data to provide a lifestyle monitor.

The application of analytics to predict civil unrest was shown to be effective in the work by Ramakrishnan et al. (2014) where multiple models (5 different models that include looking at volume, detecting planned protests, using dynamic query expansion to flexibly detect causes and motivations, checking to see if activity cascades through the spread of targeted campaigns to recruit protestors and a baseline model that looks at the likelihood of events occurring based on recent events) were used on a wide range of heterogenous data sources including sensor data to provide robust predictions of future events through corroboration between models. This can also be applied to IoT data including mobile device information, smart appliance and voice controlled device search data, video data and spatial data of crowds, vehicles and security forces to detect civil unrest.

A.5 Smart City: Parking and Big Data Grid Optimisation

Zheng et al. (2014) applied temporal clustering and anomaly detection to IoT parking data in San Francisco to discover extreme car parking behaviour and characterise normal behaviour that helped parking managers to device suitable parking strategies. This type of analysis that helps discover information can be fed back into smart parking systems to optimise the use of limited space in cities. Data mining techniques including a naive Bayes model and a logistic regression model were also applied by He et al. (2014) on sensor data to find suitable car parking spaces in real-time within a smart city scenario.

Strohbach et al. (2015) described a real-time grid monitoring system within a pilot smart city that utilised stream processing with Complex Event Processing (CEP) technology to provide power quality and consumption analysis from a stream of about 360,000 new measurements every second. A forecasting model that balanced the grid not only by adapting the production of energy but also by adapting the consumption through the distributed use of smart meters within smart homes was proposed.

A.6 Smart Transportation: Traffic Control and Routing

Liu et al. (2013) looked at the latest technologies and applications of video analytics and intelligent video systems. The traffic control system described by Mak and Fan (2006) successfully applied video analytics to detecting traffic volume for planning, highlighting incidents and punishing offenders of traffic rules for enhanced safety. Another set of applications was for intelligent vehicles to do in-lane, pedestrian and traffic sign detection that assists the driver.

Work on Intelligent Transport Systems in Dublin city by Liebig et al. (2014) also included prescribing good routes in travel planning using analytical techniques, like a spatiotemporal random field based on conditional random fields research (Pereira et al., 2001), for traffic flow prediction and a gaussian process model to fill in missing values in traffic data, to predict the future traffic flow based on historical sensor data and to estimate traffic flow in areas with limited sensor coverage. These were then used to provide the cost function for the A* search algorithm (Hart et al., 1968) that uses the combination of a search heuristic and cost function to prescribe optimal routes provided the heuristic is admissible and the predicted costs are accurate.

Wang and Cao (2014) also did work on traffic congestion prediction and prescription to optimise traffic flow through a decision-making component. They used a multilayered adaptive dynamic bayesian network (Zhu et al., 2013) to model and predict traffic events and a Markov decision process (MDP) (Bellman, 1957) method to make optimal decisions within the bounds of some uncertainty and some time frame. This was simulated on traffic in Beijing city to show that in a cooperative traffic scenario, which is possible in the context of smart transportation systems, it would reduce congestion.

A.7 Smart Buildings and Smart Homes

Ploennigs et al. (2014) showed how analytics when applied to energy monitoring could help adapt heating for smart buildings. The system was able to diagnose anomalies in the building temperature, for example, break downs of the cooling system, high occupancy of rooms, or open windows causing air exchange with the external surroundings. Using a semantics-based approach, the system could also automatically derive diagnosis rules from sensor network definitions, metadata and behaviour of specific buildings, making it sensitive to new anomalies.

Makonin et al. (2013) described how a rule-based system used in a smart home could help to conserve energy and improve the quality of living through adaptive lighting and adaptive heating. Weiss et al. (2012) similarly leveraged smart meter data in smart homes to raise energy awareness and provide feedback on energy consumption without the need for specialised hardware. This was achieved in their application by developing a signature detection algorithm to identify home appliances by analysing smart meter patterns.

Appendix B

The Smart Home Analytics Benchmark

The aim of the Smart Home Analytics Benchmark was to provide a dataset from the smart home application domain containing an integrated graph of rich metadata and data. The graph is realised as a Resource Description Framework (RDF) graph and the schema designed to integrate the data was based on a reference Internet of Things (IoT) ontology, the Semantic Sensor Network (SSN) ontology[1]. A related set of analytical queries on the graph model that power an application dashboard for visualising smart home data was also devised and form a performance benchmark for graph queries.

The data from smart home sensors was collected by Barker et al. (2012) over 4 months from April to July in 2012. There were a variety of data observations from environmental sensors, motion sensors in each room and energy meters within a smart home. There were 26,192 environment sensor readings with 13 attributes, 748,881 energy meter readings from 53 meters and 232,073 motion sensor readings from 6 rooms.

Sensor metadata, observation metadata and observation data from sensor and meter readings were integrated as an RDF graph utilising a schema that references the SSN ontology. This publicly available schema[2] represents a weather observation from an environmental sensor as shown in Figure B.1. Energy meters produce observations represented by the graph structure in Figure B.2, which were observed by a meter measuring energy usage of a refrigerator in the kitchen described in Figure B.3. Finally, Figure B.4 shows the graph structure of an observation from the motion sensor in the living room.

A set of analytical queries was designed for an application dashboard to allow the monitoring of the smart home environment, visualise trends and detect possible anomalies

[1]http://purl.oclc.org/NET/ssnx/ssn
[2]http://purl.org/net/iot

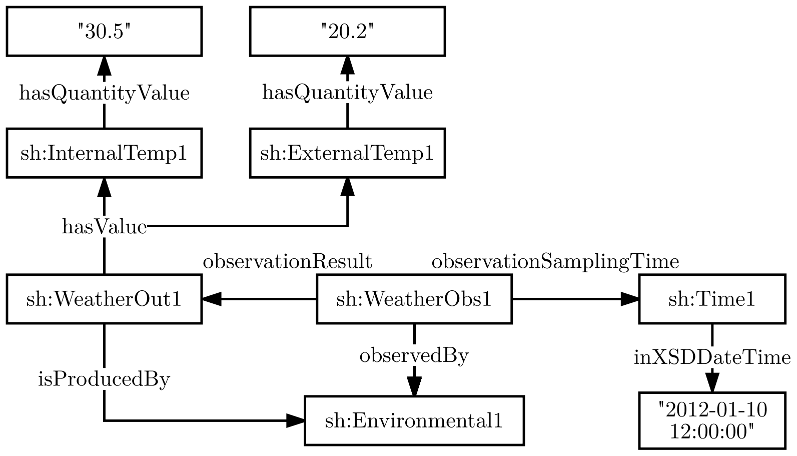


Figure B.1: The Structure of a Weather Observation from the Environmental1 Sensor

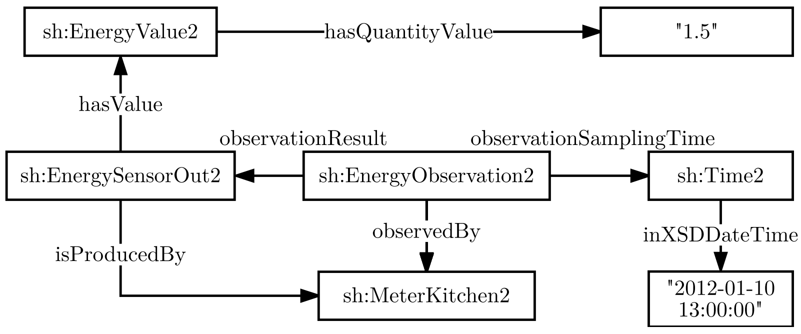


Figure B.2: The Structure of an Energy Observation from the Refrigerator Meter

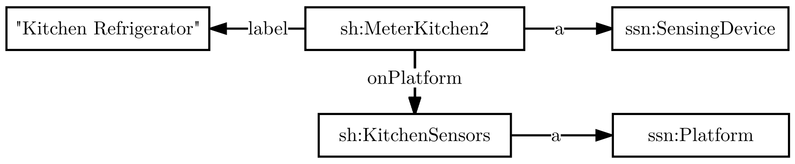


Figure B.3: The Structure of Metadata from the Refrigerator Meter in the Kitchen

over time. Each query requires space-time aggregations on the time-series of observations. The set of queries forms a non-synthetic benchmark using real-world data under an analytical application workload. The set of four graph queries are as follows:

1. the hourly aggregation of internal or external temperature,
2. the daily aggregation of temperature,
3. the hourly, room-based aggregation of energy usage and
4. the diagnosis of unattended energy usage comparing motion detectors and energy meters to determine energy usage in rooms with no activity, aggregating by hour and room.

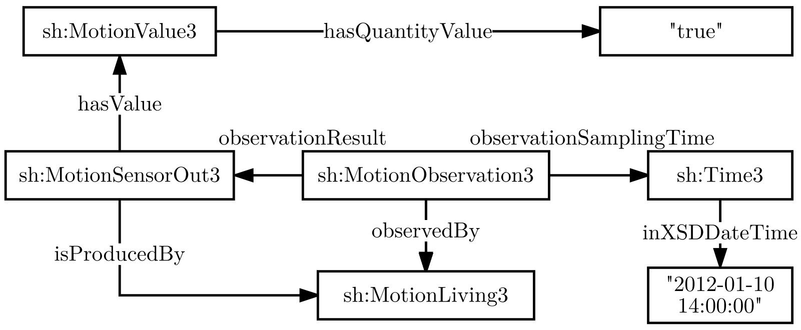


FIGURE B.4: The Structure of a Motion Observation from the Living Room Sensor

The actual query syntax for each query expressed in SPARQL 1.1, a query language for RDF graphs, is as follows in Listings B.1, B.2, B.3 and B.4.

```
PREFIX ssn: <http://purl.oclc.org/NET/ssnx/ssn#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX sh: <http://iot.soton.ac.uk/smarthome/sensor#>
PREFIX iot: <http://purl.oclc.org/NET/iot#>

SELECT (avg(?val) as ?sval) ?hours WHERE {
  ?instant time:inXSDDateTime ?date.
  ?obs ssn:observationSamplingTime ?instant;
    ssn:observedBy sh:environmental1;
    ssn:observationResult ?snout.
  ?snout ssn:hasValue ?obsval.
  ?obsval a iot:{type};
    iot:hasQuantityValue ?val.
  FILTER (?date > "{start}"^^xsd:dateTime && ?date <= "{end}"^^xsd:dateTime)
} GROUP BY (hours(xsd:dateTime(?date)) as ?hours)
```

LISTING B.1: Query 1: Hourly Environmental Observation of Varying Type

```
...
SELECT (max(?val) as ?max) (min(?val) as ?min) ?day WHERE {
  ?instant time:inXSDDateTime ?date.
  ?obs <<ssn:observationSamplingTime ?instant;
    ssn:observedBy sh:environmental1;
    ssn:observationResult ?snout.
  ?snout ssn:hasValue ?obsval.
  ?obsval a iot:{type};
    iot:hasQuantityValue ?val.
  FILTER (?date > "2012-{month}-01T00:00:00"^^xsd:dateTime &&
    ?date < "2012-{month}-30T00:00:00"^^xsd:dateTime)
} GROUP BY (day(xsd:dateTime(?date)) as ?day)
```

LISTING B.2: Query 2: Daily Environmental Observations over a Month

```
...
SELECT ?platform ?dateOnly (sum(?power) as ?totalpower) WHERE {{
    SELECT ?platform ?hours ?dateOnly (avg(?meterval) as ?power) WHERE {
      ?meter ssn:onPlatform ?platform.
      ?meterobs ssn:observedBy ?meter.
      ?meterobs ssn:observationSamplingTime ?meterinstant;
          ssn:observationResult ?metersnout.
      ?meterinstant time:inXSDDateTime ?meterdate.
      ?metersnout ssn:hasValue ?meterobsval.
      ?meterobsval a iot:EnergyValue.
      ?meterobsval iot:hasQuantityValue ?meterval.
      FILTER(?meterval > 0)
      FILTER (?meterdate > "{startDate}"^^xsd:dateTime
        && ?meterdate < "{endDate}"^^xsd:dateTime)
    } GROUP BY ?platform ?meter (hours(?meterdate) as ?hours)
      (xsd:date(?meterdate) as ?dateOnly)
}} GROUP BY ?platform ?dateOnly
```

Listing B.3: Query 3: Hourly, Room-based (Platform) Energy Usage

```
...
SELECT ?motiondate ?motionhours ?motionplatform ?power ?meter ?name WHERE {{
  SELECT (?platform as ?meterplatform) (?hours as ?meterhours)
    (?dateOnly as ?meterdate) (avg(?meterval) as ?power) ?meter
      (sample(?label) as ?name) WHERE {
        ?meter rdfs:label ?label.
        ?meter ssn:onPlatform ?platform.
        ?meterobs ssn:observedBy ?meter.
        ?meterobs ssn:observationSamplingTime ?meterinstant;
          ssn:observationResult ?metersnout.
        ?meterinstant time:inXSDDateTime ?date.
        ?metersnout ssn:hasValue ?meterobsval.
        ?meterobsval a iot:EnergyValue.
        ?meterobsval iot:hasQuantityValue ?meterval.
        FILTER(?meterval > 0)
        FILTER (?date > "{startDate}"^^xsd:dateTime &&
          ?date < "{endDate}"^^xsd:dateTime)
  } GROUP BY ?platform (hours(?date) as ?hours)
    (xsd:date(?date) as ?dateOnly) ?meter }{
  SELECT (?platform as ?motionplatform) (?hours as ?motionhours)
    (?dateOnly as ?motiondate) WHERE {
        ?obsval a iot:MotionValue;
          iot:hasQuantityValue false.
        ?snout ssn:hasValue ?obsval.
        ?obs ssn:observationSamplingTime ?instant;
          ssn:observationResult ?snout.
        ?instant time:inXSDDateTime ?date.
        ?obs ssn:observedBy ?sensor.
        ?sensor ssn:onPlatform ?platform.
        FILTER (?date > "{startDate}"^^xsd:dateTime &&
          ?date < "{endDate}"^^xsd:dateTime)
      } GROUP BY ?platform (hours(?date) as ?hours)
        (xsd:date(?date) as ?dateOnly) }
    FILTER(?motionplatform = ?meterplatform && ?motionhours = ?meterhours
      && ?motiondate = ?meterdate)
}}
```

Listing B.4: Query 4: Device Energy Usage in Rooms with No Activity by Hour

The source code of queries, application dashboard and data integration components are available on an open repository[3].

[3]https://github.com/eugenesiow/ldanalytics-PiSmartHome

Appendix C

Operate Query Translation

In this appendix, the detailed mapping of query translation vocabulary in the Operate step from the Map-Match-Operate abstraction is presented for completeness. The appendix is divided into three sections, Section C.1 elaborates on the translation of SPARQL Algebra from an input graph query to SQL, Section C.2 elaborates on translation to TritanDB operators and Section C.3 on translation to EPL for stream processing.

The set of possible SPARQL algebra operators, obtained from the parsing of an input SPARQL graph query, forms the input that is common across each section. Property path operations, which are not supported, are excluded from this set of algebra. The list of SPARQL algebra operators is obtained from the 'Translation to SPARQL algebra' section in the SPARQL 1.1 specification[1].

C.1 SPARQL Algebra to SQL

The translation from SPARQL algebra operators to SQL in this section uses the Backus Normal Form (BNF) notation description of the SQL-2003 standard by Savage (2017a) and when there are ambiguities, adopts the specific open source H2 database SQL grammar[2]. Table C.1 shows the input parameters, binding output from the $\mathbb{B}_{map}$ and corresponding SQL listing reference for each SPARQL algebra operator expanding on Table 4.4. The translation of each SPARQL algebra operator is elaborated on using the BNF notation presented in Listings C.1 to C.11. Each listing **highlights** the input parameters and binding output from $\mathbb{B}_{map}$ in the context of the translated SQL clauses and syntax. For example, in Listing C.8 for the `'OrderBy'` solution modifier, the SQL `'order by clause'` is represented in BNF notation (SQL-2003 BNF Section 14.1). The input to `'OrderBy'` is a variable, `'var'`, that specifies the result column to order by and whether to sort in ascending or descending order, specified by `'order'`. The binding output from

[1]https://www.w3.org/TR/sparql11-query/#sparqlQuery
[2]http://www.h2database.com/html/grammar.html

SPARQL Algebra	Input	Binding Output	Listing
Graph Pattern			
BGP, G	[triple pattern]	[binding.col], [binding.table]	-
Join, $\bowtie$	[join_condition]	[binding.col], [binding.table]	C.1
LeftJoin, $\ltimes$	[join_condition]	[binding.col], [binding.table]	C.2
Filter, σ	[var, predicate]	[binding.col], [binding.table]	C.3
Union, $\cup$	query_term	-	C.4
Graph	graph	map	-
Extend, ρ	[var, expr], [alias]	[binding.col], [binding.table]	C.5
Minus	query_term	-	C.6
Group/Aggr, γ	[var], [func]	[binding.col], [binding.table]	C.7
Solution Modifiers			
OrderBy	var, order	binding.col, binding.table	C.8
Project, Π	[var]	[binding.col], [binding.table]	C.9
Distinct, σ_D	-	-	C.10
Reduced σ_R	-	-	C.10
Slice, σ_S	offset, fetch	-	C.11

TABLE C.1: Summary of SPARQL Algebra to SQL with Listing References

the match step gives the ‘binding.col’ value from $\mathbb{B}_{map}$, which corresponds to ‘var’ and this is used in ‘sort spec’ as shown in the listing.

```
<joined table> ::= <cross join> | <qualified join> | <natural join> | <union join>
<qualified join> ::= <binding.table> [ <join type> ]
        JOIN <binding.table> <join spec>
<join spec> ::= <join condition> | <named columns join>
<join condition> ::= ON <search condition>
<named columns join> ::= USING <left paren> <join column list> <right paren>
<join type> ::= INNER | <outer join type> [ OUTER ]
<outer join type> ::= LEFT | RIGHT | FULL
<join column list> ::= <column name list>
<column name list> ::= <binding.col> [ { <comma> <binding.col> }... ]
```

LISTING C.1: SQL Translation of the Join Operator, $\bowtie$

Listing C.2 details the differences in SQL translation of the ‘LeftJoin’ operator from the ‘Join’ operator in Listing C.1. The ‘LeftJoin’ operator in SPARQL is used when an optional graph pattern[3] is specified.

```
<join type> ::= <outer join type> [ OUTER ]
<outer join type> ::= LEFT
```

LISTING C.2: SQL Translation of the Left Join Operator, $\ltimes$

Listing C.3 defines the SQL translation in BNF for the ‘Filter’ operator. Both the ‘WHERE’ and ‘HAVING’ clauses specified in SPARQL result in the ‘Filter’ SPARQL algebra operator. These translate directly to SQL-2003 clauses of the same names.

[3]https://www.w3.org/TR/sparql11-query/#optionals

While ‘HAVING’ specifies a ‘search condition’ for a group or an aggregate function, as it can be written after the SPARQL or SQL ‘SELECT’ clause, the ‘WHERE’ clause is limited to within the ‘SELECT’ statement.

```
<table expression> ::= <from clause> [ <where clause> ]
        [ <group by clause> ] [ <having clause> ] [ <window clause> ]
<where clause> ::= WHERE <search condition>
<having clause> ::= HAVING <search condition>
<search condition> ::= <boolean value expression>
<boolean value expression> ::= <boolean term>
        | <boolean value expression> OR <boolean term>
<boolean term> ::= <boolean factor> | <boolean term> AND <boolean factor>
<boolean factor> ::= [ NOT ] <boolean test>
<boolean test> ::= <boolean primary> [ IS [ NOT ] <truth value> ]
<truth value> ::= TRUE | FALSE | UNKNOWN
<boolean primary> ::= <predicate> | <boolean predicand>
<predicate> ::= <comparison predicate>
     | <in predicate> | <like predicate> | ...
<boolean predicand> ::= <binding.col> | ...
<from clause> ::= FROM <table reference list>
<table reference list> ::= <binding.table> [ { <comma> <binding.table> }... ]
```

Listing C.3: SQL Translation of the Filter Operator, σ

```
<query expr body> ::= <non-join query expr> | <joined table>
<non-join query expr> ::=
        <non-join query term>
        | <query expr body> UNION [ ALL | DISTINCT ] <query term>
        | <query expr body> EXCEPT [ ALL | DISTINCT ] <query term>
```

Listing C.4: SQL Translation of the Union Operator, $\cup$

```
<query spec> ::= SELECT [ <set quantifier> ] <select list> <table expr>
<select list> ::= <select sublist> [ { <comma> <select sublist> }... ]
<select sublist> ::= <derived column> | <qualified asterisk>
<derived column> ::= <binding.col> [ <as clause> ]
<binding.col> ::= <expr>
<as clause> ::= [ AS ] <alias>
```

Listing C.5: SQL Translation of the Extend Operator, ρ

Listing C.6, which details the SQL translation of the ‘minus’ operator, uses the SQL-2003 specification ‘EXCEPT’ clause supported in almost every SQL dialect. In the H2, Oracle and SQL Server grammar, both ‘minus’ and ‘EXCEPT’ can be used interchangeably. MySQL uses ‘NOT IN’, while PostgreSQL only supports ‘EXCEPT’.

```
<non-join query expr> ::= ...
        | <query expr body> EXCEPT [ ALL | DISTINCT ] <query term>
```

Listing C.6: SQL Translation of the Minus Operator

Listing C.7 explains the SQL translation of the ‘group by’ operator and aggregation functions. The aggregate functions[4] defined in SPARQL 1.1 are COUNT, SUM, MIN,

[4]https://www.w3.org/TR/sparql11-query/#aggregates

MAX, AVG, GROUP_CONCAT, and SAMPLE. The SQL-2003 specification equivalents are COUNT, SUM, MIN, MAX, AVG, COLLECT and ANY respectively. 'COLLECT' which creates a multi-set from the value of the argument in each row of a group is an imperfect translation for the 'GROUP_CONCAT' function, which is supported in the H2 and MySQL grammars, as it returns a multi-set rather than a string and needs to be subsequently passed through the 'CAST' function. Oracle uses the 'LISTAGG' function instead while SQL Server from 2017[5], supports 'STRING_AGG'. In PostgreSQL, the 'ARRAY_AGG' function which is then 'CAST' to a string, is an alternative.

```
// group by clause
<group by clause> ::= GROUP BY [ <set quantifier> ] <grouping el list>
<grouping el list> ::= <grouping el> [ { <comma> <grouping element> }... ]
<grouping el> ::= <ordinary grouping set> | ...
<ordinary grouping set> ::= <grouping col ref>
        | <left paren> <grouping col ref list> <right paren>
<grouping col ref list> ::= <grouping col ref> [{ <comma> <grouping col ref> }...]
<grouping col ref> ::= <binding.col> [ <collate clause> ]
// aggregation function
<aggregate function> ::=
        COUNT <left paren> <asterisk> <right paren> [ <filter clause> ]
        | <general set function> [ <filter clause> ] | ...
<general set function> ::= <func>
        <left paren> [ <set quantifier> ] <binding.col> <right paren>
<func> ::= <computational operation>
<computational operation> ::= AVG | MAX | MIN | SUM
        | EVERY | ANY | SOME
        | COUNT | COLLECT ...
```

Listing C.7: SQL Translation of the Group/Aggregation Operator

```
<order by clause> ::= ORDER BY <sort spec list>
<sort spec list> ::= <sort spec> [ { <comma> <sort spec> }... ]
<sort spec> ::= <binding.col> [ <order> ] [ <null ordering> ]
<order> ::= ASC | DESC
```

Listing C.8: SQL Translation of the OrderBy Operator

```
<query spec> ::= SELECT [ <set quantifier> ] <select list> <table expr>
<select list> ::= <select sublist> [ { <comma> <select sublist> }... ]
<select sublist> ::= <derived column> | <qualified asterisk>
<derived column> ::= <binding.col> [ <as clause> ]
<table expr> ::= <from clause> ...
<from clause> ::= FROM <table reference list>
<table reference list> ::= <binding.table> [ { <comma> <binding.table> }... ]
```

Listing C.9: SQL Translation of the Project Operator, Π

```
<query spec> ::= SELECT [ <set quantifier> ] <select list> <table expr>
<set quantifier> ::= DISTINCT | ALL
```

Listing C.10: SQL Translation of the Distinct and Reduced Operators, σ_D and σ_R

Listing C.11 which details the SQL translation of the 'slice' operator uses the H2 Database Grammar as the SQL-2003 specification does not specify a clear 'limit'

[5]https://docs.microsoft.com/en-us/sql/t-sql/functions/string-agg-transact-sql

SPARQL Algebra	TritanDB Operator
Graph Pattern	
BGP, G	match(BGP,map), scan(TS)
Join, $\bowtie$	join(expr...)
LeftJoin, $\ltimes$	semiJoin(expr)
Filter, σ	filter(expr...)
Union, $\cup$	union()
Graph	setMap(map)
Extend, ρ	extend(expr,var)
Minus	minus()
Group/Aggregation, γ	aggregate(groupKey, aggr)
Solution Modifiers	
OrderBy	sort(fieldOrdinal...)
Project, Π	project(exprList [, fieldNames])
Distinct, σ_D	distinct()
Reduced σ_R	distinct()
Slice, σ_S	limit(offset, fetch)

Table C.2: SPARQL Algebra to TritanDB Operator Translation

operator. It is possible to use the '`fetch`' cursor or '`window`' clause (SQL-2003 BNF Sections 14.3 and 7.11), however, H2's '`limit`' clause is well-defined and fulfils the exact purpose of the '`slice`' operator. This is similar in other SQL dialects and implementations like MySQL[6], PostgreSQL[7] and Oracle 12c onwards[8]. SQL Server's TSQL dialect (from 2012 onwards) uses the '`OFFSET-FETCH`' clause instead with similar functionality.

```
<limit clause> ::= LIMIT <fetch> [ OFFSET <offset> ]
```

Listing C.11: SQL Translation of the H2 Slice Operator, σ_S

C.2 SPARQL Algebra to TritanDB Operators

The implementation of the set of TritanDB operators is inspired by the work on an open and common relational algebra specification in Apache Calcite[9]. Table C.2 shows the conversion from a SPARQL algebra operator to the corresponding TritanDB operator. Each TritanDB operator is described as follows.

`match` is described in Definition 4.9 which matches a Basic Graph Pattern (BGP) from a query with a mapping to produce a binding $\mathbb{B}$. A set of time-series are referenced within $\mathbb{B}$. `scan` is an operator that returns an iterator over a time-series `TS`.

[6]https://dev.mysql.com/doc/refman/5.5/en/select.html
[7]https://www.postgresql.org/docs/8.1/static/queries-limit.html
[8]https://oracle-base.com/articles/12c/row-limiting-clause-for-top-n-queries-12cr1
[9]https://calcite.apache.org/docs/algebra.html

`join`, combines two time-series according to conditions specified as `expr` while `semiJoin` joins two time-series according to some condition, but outputs only columns from the left input.

`filter` modifies the input to return an iterator over points for which the conditions specified in `expr` evaluates to true. A common filter condition would a specification of a range of time for a time-series.

`union` returns the union of the input time-series and bindings $\mathbb{B}$. If the same time-series is referenced within inputs, only the bindings need to be merged. If two different time-series are merged, the iterator is formed in linear time by a comparison-based sorting algorithm, for example, the merge step within a merge sort, as the time-series are retrieved in sorted time order.

`setMap` is used to apply the specified mapping to its algebra tree leaf nodes for `match`.

`extend` allows the evaluation of an expression `expr` to be bound to a new variable `var`. This evaluation is performed only if `var` is projected. There are three means in SPARQL to produce the algebra: using `bind`, expressions in the `select` clause or expressions in the `group by` clause.

`minus` returns the iterator of first input excluding points from the second input.

`aggregate` produces an iteration over a set of aggregated results from an input. To calculate aggregate values for an input, the input is first divided into one or more groups by the `groupKey` field and the aggregate value is calculated for the particular `aggr` function for each group. The `aggr` functions supported are `count`, `sum`, `avg`, `min`, `max`, `sample` and `groupconcat`.

`sort` imposes a particular sort order on its input based on a sequence consisting of `fieldOrdinals`, each defining the time-series field index (zero-based) and specifying a positive ordinal for ascending and negative for descending order.

`project` computes the set of chosen variables to 'select' from its input, as specified by `exprList`, and returns an iterator to the result containing only the selected variables. The default name of variables provided can be renamed by specifying the new name within the `fieldNames` argument.

`distinct` eliminates all duplicate records while `reduced`, in the TritanDB implementation, performs the same function. The SPARQL specification defines the difference being that `distinct` ensures duplicate elimination while `reduced` simply permits duplicate elimination. Given that time-series are retrieved in sorted order of time, the `distinct` function works the same for `reduced` as well and eliminates immediately repeated duplicate result rows.

`limit` computes a window over the input returning an iterator over results that are of a maximum size (in rows) of `fetch` and are a distance of `offset` from the start of the result set.

C.3 SPARQL Algebra to Streaming Operators

The translation from SPARQL algebra operators to streaming operators is described with the Event Processing Language (EPL) using the Backus Normal Form (BNF) notation. The EPL grammar is based on an extension to the SQL-92 standard described by Savage (2017b) and defined within the Esper engine documentation[10].

Translation is similar to SQL translation (Section C.1) with the main difference being that the concept of tables is replaced by that of windows which could either be a stream or a static SQL data source connected by JDBC (non-relational data sources are supported in Esper but not in S2S at the moment). Listing C.12 explains the syntax of a window definition in BNF. A '`binding.window`' is the specific '`window def`' produced from the match operation containing either a '`binding.stream`' or '`binding.table`'.

```
<from clause> ::= FROM <window def> [ { <comma> <window def> }... ]
<window def> ::= <window> [ { <hash> <view spec> }... ] [ AS <window alias> ]
<window> ::= <stream def> [ unidirectional ] [ <retain> ] | <sql def>
<retain> ::= retain-union | retain-intersection
<stream def> ::= <binding.stream> <colon> <view> <left paren> <size> <right paren>
<view> ::= length | length_batch | time | time_batch | time_length_batch | ...
<sql def> ::= sql <colon> <database_name> <parameterized_sql_query>
```

Listing C.12: A Window Definition in EPL

Table C.3 shows the input parameters, binding output from the $\mathbb{B}_{map}$ and corresponding EPL listing reference for each SPARQL algebra operator. The translation of each SPARQL algebra operator is elaborated on using the BNF notation presented in Listings C.13 to C.23.

```
<qualified join> ::= <binding.window> [ <join type> ]
        JOIN <binding.window> <join spec>
<join spec> ::= <join condition> | <named columns join>
<join condition> ::= ON <search condition>
<named columns join> ::= USING <left paren> <join column list> <right paren>
<join type> ::= INNER | <outer join type> [ OUTER ]
<outer join type> ::= LEFT | RIGHT | FULL
<join column list> ::= <column name list>
<column name list> ::= <binding.col> [ { <comma> <binding.col> }... ]
```

Listing C.13: EPL Translation of the Join Operator, $\bowtie$

Listing C.14 details the differences in EPL translation of the '`LeftJoin`' operator from the '`Join`' operator in Listing C.13.

[10]http://www.espertech.com/esper/release-7.0.0-beta2/esper-reference/html/epl_clauses.html

SPARQL Algebra	Input	Binding Output	Listing
Graph Pattern			
BGP, G	[triple pattern]	[binding.col], [binding.window]	-
Join, $\bowtie$	[join_condition]	[binding.col], [binding.window]	C.13
LeftJoin, $\ltimes$	[join_condition]	[binding.col], [binding.window]	C.14
Filter, σ	[var, predicate]	[binding.col], [binding.window]	C.15
Union, $\cup$	-	[binding.window]	C.16
Graph	graph	map	-
Extend, ρ	[var, expr], [alias]	[binding.col], [binding.window]	C.17
Minus	-	[binding.window]	C.18
Group/Aggr, γ	[var], [func]	[binding.col], [binding.window]	C.19
Solution Modifiers			
OrderBy	var, order	binding.col, binding.window	C.20
Project, Π	[var]	[binding.col], [binding.window]	C.21
Distinct, σ_D	-	-	C.22
Reduced σ_R	-	-	C.22
Slice, σ_S	offset, fetch	-	C.23

Table C.3: Summary of SPARQL Algebra to EPL with Listing References

```
<join type> ::= <outer join type> [ OUTER ]
<outer join type> ::= LEFT
```

Listing C.14: EPL Translation Difference of the Left Join Operator, $\ltimes$

```
<where clause> ::= WHERE <search condition>
<having clause> ::= HAVING <search condition>
<search condition> ::= <boolean value expression>
<boolean value expression> ::= <boolean term>
        | <boolean value expression> OR <boolean term>
<boolean term> ::= <boolean factor> | <boolean term> AND <boolean factor>
<boolean factor> ::= [ NOT ] <boolean test>
<boolean test> ::= <boolean primary> [ IS [ NOT ] <truth value> ]
<truth value> ::= TRUE | FALSE | UNKNOWN
<boolean primary> ::= <predicate> | <boolean predicand>
<predicate> ::= <comparison predicate> | <like predicate> | ...
<boolean predicand> ::= <binding.col> | ...
<from clause> ::= <binding.window> [ { <comma> <binding.window> }... ]
```

Listing C.15: EPL Translation of the Filter Operator, σ

EPL does not support a ‘Union’ operator between query terms as in SQL (Listing C.4). However, Listing C.16 shows two possible methods in EPL using a full-outer-join or a ‘retain-union’ within the ‘FROM’ clause.

```
// full-outer-join method
<qualified join> ::= <binding.window> [ <join type> ]
        JOIN <binding.window> <join spec>
<join type> ::= <outer join type> [ OUTER ]
<outer join type> ::= FULL
// retain-union method
<from clause> ::= FROM <binding.window> [ { <comma> <binding.window> }... ]
<binding.window> ::= <window> [ { <hash> <view spec> }... ] [ AS <window alias> ]
<window> ::= <stream def> [ unidirectional ] [ <retain> ]
<retain> ::= retain-union
<stream def> ::= <binding.stream> ...
```

Listing C.16: Two Methods of EPL Translation for the Union Operator, ∪

EPL supports an expression alias[11] feature that can be used to reduce translated EPL query length by assigning expression aliases. This is shown in the BNF notation in addition to the translation of 'Extend' within 'SELECT' in Listing C.17.

```
<query spec> ::= SELECT [ <set quantifier> ] <select list> <window expr>
<select list> ::= <select sublist> [ { <comma> <select sublist> }... ]
<select sublist> ::= <derived column> | <qualified asterisk>
<derived column> ::= <binding.col> [ <as clause> ]
<binding.col> ::= <expr>
<as clause> ::= [ AS ] <alias>
// expression alias
<expr alias> ::= EXPRESSION <expr name> ALIAS FOR { <expr> }
```

Listing C.17: EPL Translation of the Extend Operator, ρ and Expression Aliases

EPL does not support the 'MINUS' or 'EXCEPT' operators. A possible alternative is a left-outer-join with a null check on each join column.

```
<qualified join> ::= <binding.window> [ <join type> ]
        JOIN <binding.window> <join spec>
<join type> ::= <outer join type> [ OUTER ]
<outer join type> ::= LEFT
<where clause> ::= WHERE <binding.col> IS NULL
        [ { <comma> <binding.col> IS NULL }... ]
```

Listing C.18: EPL Alternative Translation of the Minus Operator

Listing C.19 explains the EPL translation of the 'group by' operator and aggregation functions. EPL supports the aggregation functions[12] defined in SPARQL 1.1, COUNT, SUM, MIN, MAX and AVG. 'GROUP_CONCAT' is not supported while 'SAMPLE' can be emulated with 'FIRST'.

[11] http://www.espertech.com/esper/release-7.0.0-beta2/esper-reference/html/epl_clauses.html#epl-syntax-expression-alias

[12] http://www.espertech.com/esper/release-7.0.0-beta2/esper-reference/html/functionreference.html#epl-function-aggregation-std

```
// group by clause
<group by clause> ::= GROUP BY [ <set quantifier> ] <grouping el list>
<grouping el list> ::= <grouping el> [ { <comma> <grouping element> }... ]
<grouping el> ::= <ordinary grouping set> | ...
<ordinary grouping set> ::= <grouping col ref>
        | <left paren> <grouping col ref list> <right paren>
<grouping col ref list> ::= <grouping col ref> [{ <comma> <grouping col ref> }...]
<grouping col ref> ::= <binding.col> [ <collate clause> ]
// aggregation functions
<aggregate function> ::=
        COUNT <left paren> <asterisk> <right paren> [ <filter clause> ]
        | <general set function> [ <filter clause> ] | ...
<general set function> ::= <func>
        <left paren> [ <set quantifier> ] <binding.col> <right paren>
<func> ::= <computational operation>
<computational operation> ::= AVG | MAX | MIN | SUM | COUNT | FIRST ...
```

Listing C.19: EPL Translation of the Group/Aggregation Operator

```
<order by clause> ::= ORDER BY <sort spec list>
<sort spec list> ::= <sort spec> [ { <comma> <sort spec> }... ]
<sort spec> ::= <binding.col> [ <order> ] [ <null ordering> ]
<order> ::= ASC | DESC
```

Listing C.20: EPL Translation of the OrderBy Operator

```
<query spec> ::= SELECT [ <set quantifier> ] <select list> <window expr>
<select list> ::= <select sublist> [ { <comma> <select sublist> }... ]
<select sublist> ::= <derived column> | <qualified asterisk>
<derived column> ::= <binding.col> [ <as clause> ]
<window expr> ::= <from clause> ...
<from clause> ::= FROM <binding.window> [ { <comma> <binding.table> }... ]
```

Listing C.21: EPL Translation of the Project Operator, Π

```
<query spec> ::= SELECT [ <set quantifier> ] <select list> <table expr>
<set quantifier> ::= DISTINCT
```

Listing C.22: EPL Translation of the Distinct and Reduced Operators, σ_D and σ_R

```
<limit clause> ::= <limit form 1> | <limit form 2>
<limit form 1> ::= LIMIT <fetch> [ OFFSET <offset> ]
<limit form 2> ::= LIMIT <offset> [ <comma> <fetch> ]
```

Listing C.23: EPL Translation of the Slice Operator, σ_S

SQL and EPL translation from SPARQL using Map-Match-Operate following these rules is implemented within the S2S engine while TritanDB implements the translation from SPARQL to internal operators.

Bibliography

Mohammad Aazam and Eui Nam Huh. Fog Computing and Smart Gateway Based Communication for Cloud of Things. In *Proceedings of the International Conference on Future Internet of Things and Cloud*, 2014.

Mohammad Aazam, Imran Khan, Aymen Abdullah Alsaffar, and Eui Nam Huh. Cloud of Things: Integrating Internet of Things and Cloud Computing and the Issues Involved. In *Proceedings of 11th International Bhurban Conference on Applied Sciences and Technology*, 2014.

Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2012. ISSN 1931-7883.

Daniel Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model And Architecture For Data Stream Management. *The VLDB Journal*, 12(2): 120–139, 2003. ISSN 1066-8888.

Daniel Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 18(2):385–406, apr 2009. ISSN 1066-8888.

Mervat Abu-Elkheir, Mohammad Hayajneh, and Najah Abu Ali. Data Management for the Internet of Things: Design Primitives and Solution. *Sensors*, 13(11):15582–15612, 2013. ISSN 14248220.

Hussnain Ahmed. Applying Big Data Analytics for Energy Efficiency. Masters thesis, Aalto University, 2014.

Ian Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, 2002. ISSN 13891286.

Ian F Akyildiz and Ismail H Kasimoglu. Wireless Sensor And Actor Networks: Research Challenges. *Ad Hoc Networks*, 2(4):351–367, 2004. ISSN 15708705.

Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols and Applications. *IEEE Communications Surveys and Tutorials*, 17(4):2347–2376, 2015. ISSN 1553-877X.

Muhammad Ali, Feng Gao, and Alessandra Mileo. CityBench: A Configurable Benchmark to Evaluate RSP Engines Using Smart City Datasets. In *Proceedings of the 14th International Semantic Web Conference*, 2015.

Ignacio González Alonso, María Rodríguez Fernández, Juan Jacobo Peralta, and Adolfo Cortés García. A Holistic Approach to Energy Efficiency Systems through Consumption Management and Big Data Analytics. *International Journal on Advances in Software*, 6(3):261–271, 2013.

Michael P Andersen and David E Culler. BTrDB : Optimizing Storage System Design for Timeseries Processing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, 2016.

Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Computing Surveys*, 40(1):1–39, feb 2008. ISSN 03600300.

Elliot Anshelevich, Anirban Dasgupta, Eva Tardos, and Tom Wexler. Near-optimal Network Design With Selfish Agents. In *Proceedings of the 35th ACM Symposium on Theory of Computing*, 2003.

Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations And Query Execution. *The VLDB Journal*, 15(2): 121–142, 2006. ISSN 10668888.

Kevin Ashton. That 'Internet of Things' Thing. *RFiD Journal*, 2009.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, oct 2010. ISSN 13891286.

Luigi Atzori, Antonio Iera, and Giacomo Morabito. SIoT: Giving A Social Structure To The Internet Of Things. *IEEE Communications Letters*, 15(11):1193–1195, 2011. ISSN 10897798.

Luigi Atzori, Antonio Iera, Giacomo Morabito, and Michele Nitti. The Social Internet of Things (SIoT) - When Social Networks meet the Internet of Things: Concept, Architecture and Network Characterization. *Computer Networks*, 56(16):3594–3608, 2012. ISSN 13891286.

Lars Backstrom and Jure Leskovec. Supervised Random Walks: Predicting and Recommending Links in Social Networks. In *Proceedings of the 4th ACM International Conference on Web Search and Data Mining*, 2010.

Samantha Bail, Sandra Alkiviadous, Bijan Parsia, David Workman, Mark Van Harmelen, Rafael S. Goncalves, and Cristina Garilao. FishMark: A Linked Data Application Benchmark. In *Proceedings of the 8th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2012.

Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. A Survey of Middleware for Internet of Things. *Communications in Computer and Information Science*, 162:288–296, 2011. ISSN 18650929.

Kyle Banker. MongoDB in Action. Manning Publications, 2011.

Davide F Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Querying RDF streams with C-SPARQL. *ACM SIGMOD Record*, 39 (1):20, 2010. ISSN 01635808.

Sean Barker, Aditya Mishra, David Irwin, and Emmanuel Cecchet. Smart*: An Open Data Set and Tools for Enabling Research in Sustainable Homes. In *Proceedings of the Workshop on Data Mining Applications in Sustainability*, 2012.

Payam Barnaghi and Mirko Presser. Publishing Linked Sensor Data. In *Proceedings of the 3rd International Workshop on Semantic Sensor Networks*, 2010.

Payam Barnaghi, Wei Wang, Cory Henson, and Kerry Taylor. Semantics for the Internet of Things: Early Progress and Back to the Future. *International Journal on Semantic Web and Information Systems*, 8(1):1–21, 2012.

Basho. RiakTS: NoSQL Time-series Database, Retrieved June 2017, from http://basho.com/products/riak-ts/, jun 2017.

Atanu Basu. Five Pillars of Prescriptive Analytics Success. *Analytics Magazine*, pages 8–12, 2013.

Richard Bellman. A Markovian Decision Process. *Journal Of Mathematics And Mechanics*, 6:679–684, 1957. ISSN 01650114.

Maria Bermudez-Edo, Tarek Elsaleh, Payam Barnaghi, and Kerry Taylor. IoT-Lite: A Lightweight Semantic Model for the Internet of Things. In *Proceedings of the IEEE Conference on Ubiquitous Intelligence & Computing*, 2016.

Tim Berners-Lee and Mark Fischetti. Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor. Harper, 2000. ISBN 0062515861.

Jay H Bernstein. The Data-Information-Knowledge-Wisdom Hierarchy and its Antithesis. *NASKO 2.1*, pages 68–75, 2011. ISSN 1931-3896.

Jeff Bertolucci. Big Data Analytics: Descriptive Vs. Predictive Vs. Prescriptive, Retrieved June 2017, from http://goo.gl/dyNDFV, dec 2013.

Ames Bielenberg, Lara Helm, Anthony Gentilucci, Dan Stefanescu, and Honggang Zhang. The Growth Of Diaspora - A Decentralized Online Social Network In The Wild. In *Proceedings of the IEEE INFOCOM Computer Communications Workshop*, 2012.

Stefan Bischof, Athanasios Karapantelakis, Amit Sheth, Alessandra Mileo, and Payam Barnaghi. Semantic Modelling of Smart City Data. In *Proceedings of the W3C Workshop on the Web of Things*, 2014.

Barry Bishop, Atanas Kiryakov, and Damyan Ognyanoff. OWLIM: A Family of Scalable Semantic Repositories. *Semantic Web Journal*, 2(1):33–42, 2011. ISSN 15700844.

Chris Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5:1–22, 2009.

Christian Bizer and Richard Cyganiak. D2RQ - Lessons Learned. In *Proceedings of the W3C Workshop on RDF Access to Relational Databases*, 2007.

Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2001. ISSN 1552-6283.

Piero Bonatti, Claudiu Duma, Daniel Olmedilla, and Nahid Shahmehri. An Integration Of Reputation-based And Policy-based Trust Management. In *Proceedings of Semantic Web Policy Workshop*, 2005.

Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog Computing: A Platform for Internet of Things and Analytics. *Big Data Platforms for the Internet of Things: A Roadmap for Smart Environments*, pages 169–186, 2014. ISSN 1860949X.

Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the 1st Workshop on Mobile Cloud Computing*, 2012.

Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2003.

Carsten Bormann and Paul Hoffman. Concise Binary Object Representation (CBOR). Technical report, Internet Engineering Task Force, oct 2013.

Léon Bottou. From Machine Learning to Machine Reasoning: An Essay. *Machine Learning*, 94(2):133–149, 2014. ISSN 08856125.

Mike Buerli. The Current State of Graph Databases. Technical report, California Polytechnic State University, dec 2012.

Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. SPARQL Web-Querying Infrastructure: Ready for Action? In *Proceedings of the 12th International Semantic Web Conference*, 2013.

Richard Burnison. The Concurrency Of ConcurrentHashMap, Retrieved June 2017, from https://www.burnison.ca/articles/the-concurrency-of-concurrenthashmap, mar 2013.

Martin Burtscher and Paruj Ratanaworabhan. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Transactions on Computers*, 58(1):18–31, jan 2009. ISSN 0018-9340.

Charles Byers and Robert Swanson. The OpenFog Consortium Reference Architecture. Technical report, The OpenFog Consortium, 2017.

Jean-Paul Calbimonte, Óscar Corcho, and Alasdair JG Gray. Enabling Ontology-based Data Access to Streaming Data Sources. In *Proceedings of the 9th International Semantic Web Conference*, 2010.

Jean-Paul Calbimonte, Hoyoung Jeung, Oscar Corcho, and Karl Aberer. Enabling Query Technologies for the Semantic Sensor Web. *International Journal on Semantic Web and Information Systems*, 8(1):43–63, 2012. ISSN 1552-6283.

Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The LiteOS operating system: Towards Unix-like abstractions for wireless sensor networks. In *Proceedings of the International Conference on Information Processing in Sensor Networks*, 2008.

Meeyoung Cha, Hamed Haddai, Fabricio Benevenuto, and Krishna P Gummadi. Measuring User Influence in Twitter : The Million Follower Fallacy. In *Proceedings of the 4th International Conference on Web and Social Media*, 2010.

Sharma Chakravarthy and Qingchun Jiang. Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing. Springer Publishing, 2009. ISBN 9780387710020.

Neil Chandler, Bill Hostmann, Nigel Rayner, and Gareth Herschel. Gartner's Business Analytics Framework. Technical report, Gartner Inc., 2011.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.

Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics Preserving SPARQL-to-SQL Translation. *Data and Knowledge Engineering*, 68(10):973–1000, 2009. ISSN 0169023X.

Min Chen, Shiwen Mao, and Yunhao Liu. Big Data: A Survey. *Mobile Networks and Applications*, 19:171–209, 2014. ISSN 1383469X.

Mung Chiang, Sangtae Ha, Chih-Lin I, Fulvio Risso, and Tao Zhang. Clarifying Fog Computing and Networking: 10 Questions and Answers. *IEEE Communications Magazine*, 55(4):18–20, apr 2017. ISSN 0163-6804.

Konstantinos Christidis and Michael Devetsikiotis. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access*, 4:2292–2303, 2016. ISSN 21693536.

Pietro Ciciriello, Luca Mottola, and Gian Pietro Picco. Efficient Routing From Multiple Sources To Multiple Sinks In Wireless Sensor Networks. *Wireless Sensor Networks*, 2:34–50, 2007. ISSN 03029743.

Cisco. IOX, Retrieved June 2017, from https://developer.cisco.com/site/iox/, 2015.

Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. The SSN Ontology of the W3C Semantic Sensor Network Incubator Group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, dec 2012. ISSN 15708268.

Jack Copeland. Artificial Intelligence: A Philosophical Introduction. John Wiley & Sons, 2015.

Michael Corcoran. The Five Types Of Analytics. Technical report, Information Builders, 2012.

Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On Scaling Decentralized Blockchains. In *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, 2016.

Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. A Graphical Query Language Supporting Recursion. *ACM SIGMOD Record*, 16:323–330, 1987. ISSN 01635808.

Zbigniew Czech, George Havas, and Bohdan Majewski. An Optimal Algorithm for Generating Minimal Perfect Hash Functions. *Information Processing Letters*, 43(5):257–264, 1992. ISSN 00200190.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark Wheelhouse. A Simple Abstraction for Complex Concurrent Indexes. *ACM SIGPLAN Notices*, 46(10):845, 2011. ISSN 03621340.

Amir V Dastjerdi and Rajkumar Buyya. Fog Computing: Helping the Internet of Things Realize Its Potential. *Computer*, 49(8):112–116, 2016. ISSN 00189162.

Anwitaman Datta, Sonja Buchegger, Le-Hung Vu, Thorsten Strufe, and Krzysztof Rzadca. Decentralized Online Social Networks. In *Handbook of Social Network Technologies and Applications*, pages 349–378. Springer, 2010. ISBN 978-1-4419-7142-5.

Thomas Davenport. Competing on Analytics. *Harvard Business Review*, 84(1):98–107, 2006. ISSN 0017-8012.

Thomas Davenport. Analytics 3.0, Retrieved June 2017, from https://hbr.org/2013/12/analytics-30, dec 2013.

Daniele Dell'Aglio, Jean Paul Calbimonte, Marco Balduini, Oscar Corcho, and Emanuele Della Valle. On Correctness In RDF Stream Processor Benchmarking. In *Proceedings of the 12th International Semantic Web Conference*, 2013.

Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. RSP-QL Semantics: A Unifying Query Model To Explain Heterogeneity Of Rdf Stream Processing Systems. *International Journal on Semantic Web and Information Systems*, 10(4):17–44, 2014. ISSN 1552-6283.

Swarnava Dey, Arijit Mukherjee, Himadri Sekhar Paul, and Arpan Pal. Challenges Of Using Edge Devices In IoT Computation Grids. In *Proceedings of the International Conference on Parallel and Distributed Systems*, 2013.

Bhuwan Dhingra, Lihong Li, Xiujun Li, Jianfeng Gao, Yun-Nung Chen, Faisal Ahmed, and Li Deng. End-to-End Reinforcement Learning of Dialogue Agents for Information Access. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2017.

Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van De Walle. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proceedings of the 7th Workshop on Linked Data on the Web*, 2014.

Angelika Dohr, R Modre-Opsrian, Mario Drobics, Dieter Hayn, and Günter Schreier. The Internet of Things for Ambient Assisted Living. In *Proceedings of the 7th International Conference on Information Technology*, pages 804–809, 2010.

DWARF Debugging Information Format Committee. DWARF Debugging Information Format, Version 5. Technical report, Free Standards Group, feb 2017.

David Easley and Jon Kleinberg. Networks, Crowds, and Markets: Reasoning about a Highly Connected World. Cambridge University Press, 2010. ISBN 9780521195331.

Andreas Eckner. A Framework For The Analysis Of Unevenly Spaced Time Series Data, Retrieved June 2017. Working paper, 2014.

Andreas Eckner. Algorithms for Unevenly Spaced Time Series : Moving Averages and Other Rolling Operators, Retrieved June 2017. Working paper, 2017.

Elastic. Elasticsearch: RESTful, Distributed Search & Analytics, Retrieved June 2017, from https://www.elastic.co/products/elasticsearch, jun 2017.

Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, and Z. Meral Ozsoyoglu. A Complete Translation from SPARQL Into Efficient SQL. In *Proceedings of the International Database Engineering & Applications Symposium*, 2009.

Yagil Engel and Opher Etzion. Towards Proactive Event-driven Computing. In *Proceedings of the 5th ACM International Conference on Distributed Event-based Systems*, 2011.

Orri Erling. Implementing a SPARQL Compliant RDF Triple Store Using a SQL-ORDBMS. Technical report, OpenLink Software, 2001.

Amirhossein Farahzadia, Pooyan Shams, Javad Rezazadeh, and Reza Farahbakhsh. Middleware Technologies for Cloud of Things - A Survey. *Digital Communications and Networks*, 3(4):1–13, 2017. ISSN 23528648.

Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, 17(3):37–53, 1996. ISSN 0738-4602.

Ian Fette and Alexey Melnikov. The WebSockets Protocol. Technical report, Internet Engineering Task Force, dec 2011.

Frieder Ganz, Payam Barnaghi, and Francois Carrez. Information Abstraction for Heterogeneous Real World Internet Data. *IEEE Sensors Journal*, 13:3793–3805, 2013. ISSN 1530437X.

Animikh Ghosh, Ketan a. Patil, and Sunil Kumar Vuppala. PLEMS: Plug Load Energy Management Solution for Enterprises. In *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications*, 2013.

Bob Giddings, Bill Hopwood, and Geoff O'Brien. Environment, Economy and Society: Fitting Them Together Into Sustainable Development. *Sustainable Development*, 10: 187–196, 2002. ISSN 09680802.

Roberto Gimenez, Diego Fuentes, Emilio Martin, Diego Gimenez, Judith Pertejo, Sofia Tsekeridou, Roberto Gavazzi, Mario Carabaño, and Sofia Virgos. The Safety Transformation in the Future Internet Domain. *The Future Internet*, pages 190–200, 2012. ISSN 03029743.

Bart Goeman, Hans Vandierendonck, and Koen de Bosschere. Differential FCM: Increasing Value Prediction Accuracy By Improving Table Usage Efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

Olaf Gorlitz and Steffen Staab. SPLENDID : SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data*, 2011.

Jorge Granjal, Edmundo Monteiro, and Jorge Sa Silva. Security for the Internet of Things: A Survey Of Existing Protocols and Open Research Issues. *IEEE Communications Surveys and Tutorials*, 17(3):1294–1312, 2015. ISSN 1553877X.

Alasdair J G Gray, Raúl García-Castro, Kostis Kyzirakos, Manos Karpathiotakis, Jean-Paul Calbimonte, Kevin Page, Jason Sadler, Alex Frazer, Ixent Galpin, Alvaro A A Fernandes, Norman W. Paton, Oscar Corcho, Manolis Koubarakis, David De Roure, Kirk Martinez, and Asunción Gómez-Pérez. A Semantically Enabled Service Architecture for Mashups over Streaming and Stored Data. In *Proceedings of the 8th Extended Semantic Web Conference*, 2011.

Dominique Guinard and Vlad Trifa. Towards the Web of Things : Web Mashups for Embedded Devices. In *Proceedings of the Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*, 2009.

Bin Guo, Daqing Zhang, and Zhu Wang. Living With Internet of Things: The Emergence of Embedded Intelligence. In *Proceedings of the 2011 IEEE International Conferences on Internet of Things and Cyber, Physical and Social Computing*, 2011.

Chirag Gupta, Arun S Suggala, Ankit Goyal, Harsha V Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Ido Guy, Naama Zwerdling, David Carmel, Inbal Ronen, Erel Uziel, Sivan Yogev, and Shila Ofek-Koifman. Personalized Recommendation of Social Software Items Based on Social Relations. In *Proceedings of the 3rd ACM Conference on Recommender Systems*, 2009.

Amelie Gyrard, Christian Bonnet, Karima Boudaoud, and Martin Serrano. LOV4IoT: A Second Life For Ontology-based Domain Knowledge To Build Semantic Web Of Things Applications. In *Proceedings of the 4th IEEE International Conference on Future Internet of Things and Cloud*, 2016.

Stefan Hagedorn and Kai-Uwe Sattler. Efficient Parallel Processing of Analytical Queries on Linked Data. In *Proceedings of the OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, 2013.

Joe F Hair Jr. Knowledge Creation in Marketing: The Role of Predictive Analytics. *European Business Review*, 19:303–315, 2007. ISSN 0955-534X.

Stephen Harris and Nigel Shadbolt. SPARQL Query Processing with Conventional Relational Database Systems. In *Proceedings of the International Conference on Web Information Systems Engineering*, 2005.

Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The Design and Implementation of a Clustered RDF Store. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009.

Peter E Hart, Nils J Nilsson, and Betram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Olaf Hartig. An Overview on Execution Strategies for Linked Data Queries. *Datenbank-Spektrum*, 13(2):89–99, 2013. ISSN 1618-2162.

Wu He, Gongjun Yan, and Li Da Xu. Developing Vehicular Data Cloud Services in the IoT Environment. *IEEE Transactions on Industrial Informatics*, 10(2):1587–1595, 2014. ISSN 15513203.

Tom Heath and Christian Bizer. Linked Data Evolving the Web into a Global Data Space. In *Synthesis Lectures on the Semantic Web: Theory and Technology*. Morgan & Claypool, 2011. ISBN 9781608454303.

Cory Henson, Amit Sheth, and Krishnaprasad Thirunarayan. Semantic Perception: Converting Sensory Observations to Abstractions. *IEEE Internet Computing*, 16(2): 26–34, mar 2012. ISSN 1089-7801.

Alfred Hermida. Twittering the News: The Emergence of Ambient Journalism. *Journalism Practice*, 4(3):297–308, 2010. ISSN 1751-2786.

Friedhelm Hillebrand. The Creation of Standards for Global Mobile Communication: GSM and UMTS Standardization from 1982 to 2000. *IEEE Wireless Communications*, 20(5):24–33, 2013. ISSN 15361284.

Myriam Hunink, Milton Weinstein, Eve Wittenberg, Michael Drummond, Joseph Pliskin, John Wong, and Paul Glasziou. Decision Making in Health and Medicine: Integrating Evidence and Values. Cambridge University Press, jun 2014. ISBN 9781107690479.

Md Zahurul Huq and Sayed Islam. Home Area Network Technology Assessment for Demand Response in Smart Grid Environment. In *Proceedings of the 20th Australasian Universities Power Engineering Conference*, 2010.

IEEE Standards Association. IEEE Standard for Floating-Point Arithmetic 754-2008. Technical report, IEEE Standards Association, aug 2008.

Influx Data. InfluxDB Documentation, Retrieved June 2017, from https://docs.influxdata.com/influxdb/v1.2/, jun 2017.

Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed Diffusion For Wireless Sensor Networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, feb 2003. ISSN 1063-6692.

Intel. Video Analytics at the Edge, Retrieved June 2017, from https://www.intel.com/content/www/us/en/embedded/digital-security-surveillance/video-analytics-at-the-edge-solution-brief.html, 2016.

International Telecommunication Union. Overview of the Internet of Things. Technical report, International Telecommunication Union, jun 2012.

Kaippallimalil J Jacob, Morgan Stanley, Dean Witter, New York, and Dennis Shasha. FinTime - A Financial Time Series Benchmark. *ACM SIGMOD Record*, 28(4):42–48, dec 1999. ISSN 01635808.

Toby Jaffey, John Davies, and Pilgrim Beart. Hypercat 3.00 Specification. Technical report, Hypercat Limited, feb 2016.

Antonio Jara, Pablo Lopez, David Fernandez, Jose Castillo, Miguel Zamora, and Antonio Skarmeta. Mobile digcovery: A Global Service Discovery for the Internet of Things. In *Proceedings of the 27th International Conference on Advanced Information Networking and Applications Workshops*, 2013.

Simon Jirka and Daniel Nüst. OGC Sensor Instance Registry Discussion Paper. Technical report, Open Geospatial Consortium, oct 2010.

Melvin Johnson, Mike Schuster, Quoc V Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation, Retrieved June 2017. Working paper, 2016.

KairosDB. KairosDB: Fast Time Series Database on Cassandra, Retrieved June 2017, from https://kairosdb.github.io/, jun 2015.

Ad Kamerman and Leo Monteban. WaveLAN-II: A High-performance Wireless LAN for the Unlicensed Band. *Bell Labs Technical Journal*, 2(3):118, 1997. ISSN 10897089.

Lisa Kart. Advancing Analytics. Technical report, Gartner Inc., 2012.

Khalid S Khan, Regina Kunz, Jos Kleijnen, and Gerd Antes. Five Steps to Conducting a Systematic Review. *Journal of the Royal Society of Medicine*, 96(3):118–121, 2003.

Sooyeon Kim, Sang H Son, John Stankovic, Shuoqi Li, and Yanghee Choi. SAFE: A Data Dissemination Protocol for Periodic Updates in Sensor Networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, 2003.

Sefki Kolozali, Maria Bermudez-Edo, Daniel Puschmann, Frieder Ganz, and Payam Barnaghi. A Knowledge-Based Approach for Real-Time IoT Data Stream Annotation and Processing. In *Proceedings of the 2014 IEEE International Conference on Internet of Things*, 2014.

Mathias Konrath, Thomas Gottron, Steffen Staab, and Ansgar Scherp. SchemEX - Efficient Construction of a Data Catalogue By Stream-based Indexing of Linked Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 16:52–58, nov 2012. ISSN 1570-8268.

Michael Koster. iot.schema.org. In *Proceedings of the Workshop on IoT Semantic/Hypermedia Interoperability*, 2017.

Egor V Kostylev, Juan L Reutter, and Miguel Romero. SPARQL with Property Paths. In *Proceedings of the 14th International Semantic Web Conference*, 2015.

Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Rohit Kumar Kaliyar. Graph Databases: A Survey. In *Proceedings of the International Conference on Computing, Communication and Automation*, 2015.

Bradley C Kuszmaul. A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. *DZone Refcardz*, pages 1–15, 2014.

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, 2010.

Gunter Ladwig and Andreas Harth. CumulusRDF: Linked Data Management on Nested Key-Value Stores. In *Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2011.

Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010. ISSN 01635980.

Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. ISSN 01640925.

Steve Lavalle, Michael S Hopkins, Eric Lesser, Rebecca Shockley, and Nina Kruschwitz. Analytics : The New Path to Value. *MIT Sloan Management Review*, pages 1–24, 2010. ISSN 15329194.

Eugene Lazin. Akumuli Time-series Database, Retrieved June 2017, from http://akumuli.org/, jun 2017.

Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proceedings of the 10th International Semantic Web Conference*, 2011.

Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting Positive And Negative Links In Online Social Networks. In *Proceedings of the 19th International World Wide Web Conference*, 2010a.

Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed Networks in Social Media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010b.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph Evolution: Densification and Shrinking Diameters. *ACM Transactions on Knowledge Discovery from Data*, 1 (2):1–39, 2007. ISSN 15564681.

Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An Operating System for Wireless Sensor Networks. *Ambient Intelligence*, 8491:115–148, 2005. ISSN 03796566.

Lin Li, Saeed Bagheri, Helena Goote, Asif Hasan, and Gregg Hazard. Risk Adjustment of Patient Expenditures: A Big Data Analytics Approach. In *Proceedings of the 2013 IEEE International Conference on Big Data*, 2013.

Leonid Libkin, Wim Martens, and Domagoj Vrgoc. Querying Graphs with Data. *Journal of the ACM*, 63(2):1–53, 2016. ISSN 1557735X.

Thomas Liebig, Nico Piatkowski, Christian Bockermann, and Katharina Morik. Predictive Trip Planning-Smart Routing in Smart Cities. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference*, 2014.

Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet of Things Journal*, 2017. ISSN 2327-4662.

Honghai Liu, Shengyong Chen, and Naoyuki Kubota. Intelligent Video Systems and Analytics: A Survey. *IEEE Transactions on Industrial Informatics*, 9(3):1222–1233, 2013. ISSN 15513203.

Pasquale Lops, Marco de Gemmis, and Giovanni Semeraro. Content-based Recommender Systems: State of the Art and Trends. Springer, 2010.

Helmut Lütkepohl. New Introduction to Multiple Time Series Analysis. Springer, 2010.

Steven Lynden, Isao Kojima, Akiyoshi Matono, and Yusuke Tanimura. ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints, 2011.

Doug Madory. Earthquake Rocks Internet In Nepal, Retrieved June 2017, from http://dyn.com/blog/earthquake-rocks-internet-in-nepal/, 2015.

Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless Sensor Networks For Habitat Monitoring. In *Proceedings of the 1st ACM International Workshop On Wireless Sensor Networks And Applications*, 2002.

Chin Mak and Henry Fan. Heavy Flow-Based Incident Detection Algorithm Using Information From Two Adjacent Detector Stations. *Journal of Intelligent Transportation Systems*, 10(1):23–31, 2006. ISSN 1547-2450.

Stephen Makonin, Lyn Bartram, and Fred Popowich. A Smarter Smart Home: Case Studies of Ambient Intelligence. *IEEE Pervasive Computing*, 12(1):58–66, 2013. ISSN 1536-1268.

Thomas W Malone, Robert Laubacher, and Chrysanthos Dellarocas. Harnessing Crowds : Mapping The Genome Of Collective Intelligence. *MIT Sloan Research Paper*, 1:1–20, 2009. ISSN 15565068.

Henrique S Malvar. Adaptive Run-Length / Golomb-Rice Encoding of Quantized Generalized Gaussian Sources with Unknown Statistics. In *Proceedings of the Data Compression Conference*, 2006.

Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. A Demonstration of the Solid Platform for Social Web Applications. In *Proceedings of the 25th International Conference on World Wide Web Companion*, 2016.

James Manyika, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. The Internet of Things: Mapping the Value Beyond the Hype. Technical report, McKinsey Global Institute, jun 2015.

Ibrahim Mashal, Osama Alsaryrah, Tein Yaw Chung, Cheng Zen Yang, Wen Hsing Kuo, and Dharma P. Agrawal. Choices for Interaction With Things on Internet and Underlying Issues. *Ad Hoc Networks*, 28:68–90, 2015. ISSN 15708705.

Zoltan Mathe, Adrian Casajus Ramo, Federico Stagni, and Luca Tomassetti. Evaluation of NoSQL Databases for DIRAC Monitoring and Beyond. In *Proceedings of the 21st International Conference on Computing in High Energy and Nuclear Physics*, 2015.

Colleen McCue. Data Mining and Predictive Analytics: Battlespace Awareness for the War on Terrorism. *Defense Intelligence Journal*, 2:47–63, 2005.

Colleen McCue. Data Mining and Predictive Analytics in Public Safety and Security. *IT Professional*, 8(August), 2006. ISSN 1520-9202.

Milan Milenkovic. A Case for Interoperable IoT Sensor Data and Meta-data Formats. *Ubiquity*, 2015(November):1–7, 2015. ISSN 15302180.

Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of Things: Vision, Applications and Research Challenges. *Ad Hoc Networks*, 10 (7):1497–1516, 2012. ISSN 15708705.

Arijit Mukherjee, Swarnava Dey, Himadri Sekhar Paul, and Batsayan Das. Utilising Condor for Data Parallel Analytics in an IoT Context - An Experience Report. In *Proceedings of the 9th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, 2013.

Arijit Mukherjee, Arpan Pal, and Prateep Misra. Data Analytics in Ubiquitous Sensor-based Health Information Systems. In *Proceedings of the 6th International Conference on Next Generation Mobile Applications, Services, and Technologies*, 2012.

Arijit Mukherjee, Himadri Sekhar Paul, Swarnava Dey, and Ansuman Banerjee. ANGELS for Distributed Analytics In IoT. In *Proceedings of IEEE World Forum on Internet of Things*, 2014.

Ujjal Kumar Mukherjee and Snigdhansu Chatterjee. Fast Algorithm for Computing Weighted Projection Quantiles and Data Depth for High-Dimensional Large Data Clouds. In *Proceedings of the 2014 IEEE International Conference on Big Data*, 2014.

Janine Nahapiet and Sumantra Ghoshal. Social Capital, Intellectual Capital, And The Organizational Advantage. *The Academy of Management Review*, 23(2):242–266, 1998. ISSN 03637425.

Adam Neat. Maximizing Performance and Scalability with IBM WebSphere. Apress, 2004. ISBN 978-1-59059-130-7.

Septimiu Nechifor, Anca Petrescu, Dan Puiu, and Bogdan Tarnauca. Predictive Analytics based on CEP for Logistic of Sensitive Goods. In *Proceedings of the International Conference on Optimization of Electrical and Electronic Equipment*, 2014.

Angelia Nedić and Asuman Ozdaglar. Distributed Subgradient Methods For Multi-agent Optimization. *IEEE Transactions On Automatic Control*, 54(1):4711–4716, 2009. ISSN 01912216.

Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A Highly-Scalable RDF Store. In *Proceedings of the 14th International Semantic Web Conference*, 2015.

Thomas Neumann and Gerhard Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *Proceedings of the VLDB Endowment*, 3 (1-2):256–263, 2010. ISSN 21508097.

Andrew Newton and Pete Cordell. A Language for Rules Describing JSON Content. Technical report, Internet Engineering Task Force, mar 2017.

Huansheng Ning and Ziou Wang. Future Internet of Things Architecture: Like Mankind Neural System or Social Organization Framework? *IEEE Communications Letters*, 15(4):461–463, 2011. ISSN 1089-7798.

Michele Nitti, Roberto Girau, and Luigi Atzori. Trustworthiness Management In The Social Internet Of Things. *IEEE Transactions on Knowledge and Data Engineering*, 26(5):1253–1266, 2014. ISSN 10414347.

Michael O'Connor. Method of Enabling Heterogeneous Platforms to Utilize a Universal File System in a Storage Area Network, US Patent 6,564,228, 2003.

Tobias Oetiker. Rrdtool: Round Robin Database Tool, Retreived June 2017, from https://oss.oetiker.ch/rrdtool/, 1998.

Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996. ISSN 0001-5903.

OneM2M. Home Appliances Information Model and Mapping. Technical report, European Telecommunications Standards Institute, sep 2016.

Gerald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *Proceedings of the International Conference on Collaborative Computing*, 2006.

Alisdair Owens, Andy Seaborne, Nick Gibbins, and Mc Schraefel. Clustered TDB: A Clustered Triple Store for Jena. Technical report, University of Southampton, 2008.

Oxford English Dictionary. "analytics, n."., aug 2017a.

Oxford English Dictionary. "periodicity, n."., aug 2017b.

M Tamer Özsu and Patrick Valduriez. Principles of Distributed Database Systems. Springer Science & Business Media, 2011. ISBN 1441988343.

Harshal Patni, Cory Henson, and Amit Sheth. Linked Sensor Data. In *Proceedings of the International Symposium on Collaborative Technologies and Systems*, 2010.

Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A Fast, Scalable, In-Memory Time Series Database. In *Proceedings of the 41st International Conference on Very Large Data Bases*, 2015.

Fernando Pereira, John Lafferty, and Andrew Mccallum. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of 18th International Conference on Machine Learning*, 2001.

Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context Aware Computing for the Internet of Things: A Survey. *IEEE Communications Surveys and Tutorials*, 16(1):414–454, 2014. ISSN 1553877X.

Todd Persen and Robert Winslow. Benchmarking InfluxDB vs. Cassandra for Time-series Data, Metrics & Management. Technical report, InfluxData, nov 2016a.

Todd Persen and Robert Winslow. Benchmarking InfluxDB vs. MongoDB for Time-series Data, Metrics & Management. Technical report, InfluxData, nov 2016b.

Todd Persen and Robert Winslow. Benchmarking InfluxDB Vs. OpenTSDB Time-series Data, Metrics & Management. Technical report, InfluxData, nov 2016c.

Maria Chiara Pettenati, Lucia Ciofi, Franco Pirri, and Dino Giuli. Towards a RESTful Architecture for Managing a Global Distributed Interlinked Data-Content-Information Space. In *Proceedings of the Future Internet Assembly*, 2011.

Dennis Pfisterer, Kay Romer, Daniel Bimschas, Oliver Kleine, Richard Mietz, Cuong Truong, Henning Hasemann, Alexander Kröller, Max Pagel, Manfred Hauswirth, Marcel Karnstedt, Myriam Leggieri, Alexandre Passant, and Ray Richardson. SPITFIRE: Toward a Semantic Web of Things. *IEEE Communications Magazine*, 49(11):40–48, 2011. ISSN 0163-6804.

Joern Ploennigs, Anika Schumann, and Freddy Lécué. Adapting Semantic Sensor Networks for Smart Building Diagnosis. In *Proceedings of the 13th International Semantic Web Conference*, 2014.

Mirko Presser, Lasse Vestergaard, and Sorin Ganea. CityPulse 101 Smart City Scenarios, Retrieved June 2017, from http://www.ict-citypulse.eu/scenarios/, 2016.

Freddy Priyatna, Oscar Corcho, and Juan Sequeda. Formalisation and Experiences of R2RML-based SPARQL to SQL Query Translation using Morph. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 479–489, 2014.

Project FiFo. DalmatinerDb: A Fast, Distributed Metric Store, Retrieved June 2017, from https://dalmatiner.io/, jun 2014.

Prometheus. Prometheus Monitoring System and Time-series Database, Retrieved June 2017, from https://prometheus.io/, jun 2016.

Guo-Jun Qi, Charu C Aggarwal, Jiawei Han, and Thomas Huang. Mining Collective Intelligence in Diverse Groups. In *Proceedings of the 22nd International World Wide Web Conference*, 2013.

Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources With SPARQL. In *Proceedings of the 5th European Semantic Web Conference*, 2008.

Rackspace. Blueflood DB for Metrics, Retrieved June 2017, from http://blueflood.io/, jun 2013.

Dave Raggett. Countering Fragmentation With The Web of Things, Retrieved June 2017, from https://www.w3.org/Talks/2016/04-27-countering-fragmentation.pdf, apr 2016.

Nur Aini Rakhmawati and Michael Hausenblas. On the Impact of Data Distribution in Federated SPARQL Queries. In *Proceedings of 6th IEEE International Conference on Semantic Computing*, 2012.

Nur Aini Rakhmawati, Jürgen Umbrich, Marcel Karnstedt, Ali Hasnain, and Michael Hausenblas. Querying Over Federated SPARQL Endpoints - A State of the Art Survey. Technical report, Digital Enterprise Research Institute, 2013.

Naren Ramakrishnan, Patrick Butler, and Sathappan Muthiah. 'Beating the news' with EMBERS: Forecasting Civil Unrest using Open Source Indicators. In *Proceedings of the 20th International Conference on Knowledge Discovery and Data Mining*, 2014.

Partha Pratim Ray. A Survey on Internet of Things Architectures. *Journal of King Saud University - Computer and Information Sciences*, 2016. ISSN 22131248.

Ahmad Razip, Abish Malik, Shehzad Afzal, Matthew Potrawski, Ross Maciejewski, Yun Jang, Niklas Elmqvist, and David Ebert. A Mobile Visual Analytics Approach for Law Enforcement Situation Awareness. In *Proceedings of the 2014 IEEE Pacific Visualization Symposium*, 2014.

Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Cla. Middleware for Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(1): 70–95, 2016. ISSN 23274662.

Redhat. Hawkular Open Source Monitoring Components, Retrieved June 2017, from http://www.hawkular.org/, jun 2017.

Kira Rehfeld, Norbert Marwan, Jobst Heitzig, and Jürgen Kurths. Comparison Of Correlation Analysis Techniques for Irregularly Sampled Time Series. *Nonlinear Processes in Geophysics*, 18(3):389–404, 2011. ISSN 10235809.

Berthold Reinwald, Hamid Pirahesh, Ganapathy Krishnamoorthy, George Lapis, Brian Tran, and Swati Vora. Heterogeneous Query Processing Through Sql Table Functions. In *Proceedings of the 15th International Conference on Data Engineering*, 1999.

Juan L Reutter. Graph Patterns: Structure , Query Answering and Applications in Schema Mappings and Formal Language Theory. PhD thesis, University of Edinburgh, 2013.

Robert Rice and James Plaunt. Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. *IEEE Transactions on Communications*, 19 (6):889–897, dec 1971. ISSN 0096-2244.

Silva Robak, Bogdan Franczyk, and Marcin Robak. Applying Big Data and Linked Data Concepts in Supply Chains Management. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, 2013.

Ian Robinson, Jim Webber, and Emil Eifrem. Graph Databases: New Opportunities for Connected Data. O'Reilly, 2015. ISBN 1449356265.

Alejandro Rodríguez-González, Javier Torres-Niño, Gandhi Hernández-Chan, Enrique Jiménez-Domingo, and Jose Maria Alvarez-Rodríguez. Using Agents to Parallelize a Medical Reasoning System Based on Ontologies and Description Logics as an Application Case. *Expert Systems with Applications*, 39(18):13085–13092, 2012. ISSN 09574174.

Mariano Rodriguez-Muro and Martin Rezk. Efficient SPARQL-to-SQL with R2RML mappings. *Web Semantics: Science, Services and Agents on the World Wide Web*, 33: 141–169, 2015. ISSN 1570-8268.

Rodrigo Roman, Jianying Zhou, and Javier Lopez. On the Features and Challenges of Security and Privacy in Distributed Internet of Things. *Computer Networks*, 57(10): 2266–2279, 2013. ISSN 13891286.

Jennifer Rowley. The Wisdom Hierarchy: Representations of the DIKW Hierarchy. *Journal of Information Science*, 33(2):163–180, 2007. ISSN 0165-5515.

Yehoshua Sagiv. Concurrent Operations On B-trees With Overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, 1986.

Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. A Fine-Grained Evaluation of SPARQL Endpoint Federation Systems. *Semantic Web Journal*, 1:1–5, 2014.

Beatriz San Miguel, Jose M del Alamo, and Juan C Yelmo. Integrating Business Information Streams in a Core Banking Architecture: A Practical Experience. In *Proceedings of the 16th International Conference on Enterprise Information Systems*, 2015.

Luis Sanchez, Jose Antonio Galache, Veronica Gutierrez, Jose Manuel Hernandez, Jesus Bernat, Alex Gluhak, and Tomas Garcia. SmartSantander: The Meeting Point Between Future Internet Research and Experimentation and the Smart Cities. In *Proceedings of the Future Network & Mobile Summit*, 2011.

Rebecca Sandefur and Edward Laumann. A Paradigm for Social Capital. *Rationality and Society*, 10(4):481–501, 1998.

Peter Sanders, Sebastian Egner, and Ludo Tolhuizen. Polynomial Time Algorithms For Network Information Flow. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, 2003.

Mehadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The Case for VM-Base Cloudlets in Mobile Computing. *Pervasive Computing*, 8:14–23, 2009. ISSN 1536-1268.

Ron Savage. BNF Grammar for ISO/IEC 9075-2:2003 - Database Language SQL (SQL-2003) SQL/Foundation. Technical report, jan 2017a.

Ron Savage. BNF Grammar for ISO/IEC 9075:1992 - Database Language SQL (SQL-92). Technical report, jan 2017b.

Ali H Sayed. Adaptation, Learning, And Optimization Over Networks. *Foundations and Trends in Machine Learning*, 7(4-5):311–801, 2014.

Yiannakis Sazeides and James E Smith. The Predictability of Data Values. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE Computer Society, 1997.

Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. FedBench: A Benchmark Suite for Federated Semantic Data Query Processing. In *Proceedings of the 10th International Semantic Web Conference*, 2011.

Francois Schnizler, Thomas Liebig, Shie Mannor, Gustavo Souto, Sebastian Bothe, and Hendrik Stange. Heterogeneous Stream Processing for Disaster Detection and Alarming. In *Proceedings of the 2014 IEEE International Conference on Big Data*, 2014.

Jurgen Schonwalder, Martin Bjorklund, and Phil Shafer. Network Configuration Management Using NETCONF and YANG. *IEEE Communications Magazine*, 48(9): 166–173, 2010. ISSN 0163-6804.

Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: Optimization Techniques For Federated Query Processing On Linked Data. In *Proceedings of the 10th International Semantic Web Conference*, 2011.

John Scott. The History of Social Network Analysis. In *Social Network Analysis*. Sage, 2012. ISBN 9781473952126.

Russell Sears and Raghu Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

Pallavi Sethi and Smruti R Sarangi. Internet of Things: Architectures, Protocols, and Applications. *Journal of Electrical and Computer Engineering*, 2017, 2017. ISSN 20900155.

Nicolas Seydoux, Khalil Drira, Nathalie Hernandez, and Thierry Monteil. IoT-O, A Core-Domain IoT Ontology to Represent Connected Devices Networks. In *Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management*, 2016.

Amit Sheth, Cory Henson, and Satya S Sahoo. Semantic Sensor Web. *IEEE Internet Computing*, 12(4):78–83, 2008. ISSN 1089-7801.

Galit Shmueli and Otto Koppiu. Predictive Analytics in Information Systems Research. *Robert Smith Research*, pages 06–138, 2010. ISSN 00390895.

Alexander Sicular. Time Series, The New Shiny?, Retreived June 2017, from http://basho.com/posts/technical/time-series-the-new-shiny/, may 2016.

Eugene Siow, Thanassis Tiropanis, and Wendy Hall. Interoperable & Efficient : Linked Data for the Internet of Things. In *Proceedings of the 3rd International Conference on Internet Science*, 2016a.

Eugene Siow, Thanassis Tiropanis, and Wendy Hall. PIOTRe: Personal Internet of Things Repository. In *Proceedings of the 15th International Semantic Web Conference P&D*, 2016b.

Eugene Siow, Thanassis Tiropanis, and Wendy Hall. SPARQL-to-SQL on Internet of Things Databases and Streams. In *Proceedings of the 15th International Semantic Web Conference*, 2016c.

Eugene Siow, Thanassis Tiropanis, and Wendy Hall. Ewya: An Interoperable Fog Computing Infrastructure with RDF Stream Processing. In *Proceedings of the 4th International Conference on Internet Science*, 2017.

SmartM2M. Smart Appliances; SAREF Extension Investigation. Technical report, European Telecommunications Standards Institute, feb 2017.

Spotify. Heroic Time-series Database, Retrieved June 2017, from https://spotify.github.io/heroic/, jun 2017.

Square. Cube Time-series Data Collection and Analysis, Retrieved June 2017, from http://square.github.io/cube/, jun 2012.

John A Stankovic. Research Directions for the Internet of Things. *IEEE Internet of Things Journal*, 1(1):3–9, 2014. ISSN 2327-4662.

Statista. Internet of Things (IoT): Number of Connected Devices Worldwide. Technical report, Statista Inc., jan 2017a.

Statista. Number of Monthly Active Twitter Users Worldwide. Technical report, Statista Inc., jan 2017b.

Michael Stonebraker, Ugur Cetintemel, and Stan Zdonik. The 8 Requirements of Real-time Stream Processing. *ACM SIGMOD Record*, 34(4):42–47, 2005. ISSN 01635808.

Martin Strohbach, Holger Ziekow, Vangelis Gazis, and Navot Akiva. Towards a Big Data Analytics Framework for IoT and Smart City Applications. *Modeling and Optimization in Science and Technologies*, 4(July):257–282, 2015.

StumbleUpon. OpenTSDB - A Scalable, Distributed Monitoring System, Retrieved June 2017, from http://opentsdb.net/, jun 2017.

Mari Carmen Suárez-Figueroa. NeOn Methodology for Building Ontology Networks: Specification, Scheduling and Reuse. PhD thesis, Universidad Politécnica De Madrid, jun 2010.

Lu Tan and Neng Wang. Future Internet: The Internet of Things. In *Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering*, 2010.

Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. Performance Evaluation Of MQTT And CoAP Via A Common Middleware. In *Proceedings of the 9th IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, 2014.

Bryan Thompson. Bigdata Database Architecture. Technical report, Systap, 2013.

Timescale. Timescale: SQL Made Scalable for Time-series Data, Retrieved June 2017, from http://www.timescale.com/, jun 2017.

Andreas Tolk, Saikou Diallo, and Charles Turnitsa. Applying the Levels of Conceptual Interoperability Model in Support of Integratability, Interoperability and Composability for System-of-Systems Engineering. *Journal of Systemics, Cybernetics and Informatics*, 5(5):65–74, 2007. ISSN 1690-4524.

Grisha Trubetskoy. Storing Time Series in PostgreSQL, Retreived June 2017, from https://grisha.org/blog/2016/12/16/storing-time-series-in-postgresql-part-ii/, dec 2016.

Grisha Trubetskoy. Storing Time Series in PostgreSQL - Optimize for Write, Retreived June 2017, from https://grisha.org/blog/2017/01/21/storing-time-seris-in-postgresql-optimize-for-write/, jan 2017a.

Grisha Trubetskoy. Tgres, Retrieved June 2017, from https://github.com/tgres/tgres, jun 2017b.

John W Tukey. The Future of Data Analysis. *Annals of Mathematical Statistics*, 33(1): 1–67, 1962. ISSN 0003-4851.

Stefano Turchi, Federica Paganelli, Lorenzo Bianchi, and Dino Giuli. A Lightweight Linked Data Implementation for Modeling the Web of Things. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communication Workshops*, 2014.

Matt Turck. Internet of Things: Are We There Yet? (The 2016 IoT Landscape), Retrieved June 2017, from http://mattturck.com/2016-iot-landscape/, 2016.

Luis M Vaquero and Luis Rodero-Merino. Finding Your Way In The Fog: Towards a Comprehensive Definition of Fog Computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014. ISSN 01464833.

Rajesh Vargheese and Hazim Dahir. An IoT / IoE Enabled Architecture Framework for Precision On Shelf Availability. In *Proceedings of the IEEE International Conference on Big Data*, 2014.

Hal Varian. How the Web Challenges Managers, Retrieved June 2017, from http://www.mckinsey.com/industries/high-tech/our-insights/hal-varian-on-how-the-web-challenges-managers, jan 2009.

Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Van De Walle. Querying Datasets on the Web with High Availability. In *Proceedings of the 13th International Semantic Web Conference*, 2014.

Cor Verdouw, Adrie Beulens, and Jack van der Vorst. Virtualisation Of Floricultural Supply Chains: A Review From An IoT Perspective. *Computers and Electronics in Agriculture*, 99:160–175, 2013. ISSN 01681699.

Ovidiu Vermesan and Peter Friess. Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems. River Publishers, 2013. ISBN 9788792982735.

Ovidiu Vermesan and Peter Friess. Internet of Things – From Research and Innovation to Market Deployment. River Publishers, 2014. ISBN 9788793102941.

Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. .NET gadgeteer: A Platform for Custom Devices. In *Proceedings of the 10th International Conference on Pervasive Computing*, 2012.

Domagoj Vrgoc. Querying Graphs With Data. PhD thesis, University of Edinburgh, 2014.

Mark Walport. The Internet of Things: Making the Most of the Second Digital Revolution. Technical report, The United Kingdom Government Office for Science, 2014.

Xin Wang, Thanassis Tiropanis, and Hugh C Davis. LHD: Optimising Linked Data Query Processing Using Parallelisation. In *Proceedings of the Workshop on Linked Data on the Web*, 2013.

Yongheng Wang and Kening Cao. A Proactive Complex Event Processing Method for Intelligent Transportation Systems. *International Journal of Distributed Sensor Networks*, 10(3):109–113, 2014. ISSN 23013788.

Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment*, 1 (1):1008–1019, aug 2008.

Markus Weiss, Adrian Helfenstein, Friedemann Mattern, and Thorsten Staake. Leveraging Smart Meter Data to Recognize Home Appliances. In *Proceedings of the 2012 IEEE International Conference on Pervasive Computing and Communications*. IEEE, mar 2012.

Joel West and Michael Mace. Browsing as the Killer App: Explaining the Rapid Success of Apple's iPhone. *Telecommunications Policy*, 34(5-6):270–286, 2010. ISSN 03085961.

Tomasz W Wlodarczyk. Overview of Time Series Storage and Processing in a Cloud Environment. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science*, 2012.

Marilyn Wolf. The Physics of Event-Driven IoT Systems. *IEEE Design and Test*, 34(2): 87–90, 2017. ISSN 21682356.

Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Technical report, Ethereum Project, 2014.

Peter T Wood. Query Languages for Graph Databases. *ACM SIGMOD Record*, 41(1): 50, 2012. ISSN 01635808.

Geng Wu, Shilpa Talwar, Kerstin Johnsson, Nageen Himayat, and Kevin Johnson. M2M: From Mobile to Embedded Internet. *IEEE Communications Magazine*, 49(4):36–43, apr 2011. ISSN 0163-6804.

Miao Wu, Ting Jie Lu, Fei Yang Ling, Jing Sun, and Hui Ying Du. Research on the Architecture of Internet of Things. In *Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering*, 2010.

Lida Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, PP(4):1–11, 2014. ISSN 1551-3203.

Fan Yang, Nelson Matthys, Rafael Bachiller, Sam Michiels, Wouter Joosen, and Danny Hughes. uPnP: Plug and Play Peripherals for the Internet of Things. In *Proceedings of the 10th European Conference on Computer Systems*, 2015.

Lily Yang, Ram Dantu, Terry Anderson, and Ram Gopal. Forwarding and Control Element Separation (ForCES) Framework. Technical report, Internet Engineering Task Force, apr 2004.

Ching-man Au Yeung, Ilaria Liccardi, Kanghao Lu, Oshani Seneviratne, and Tim Berners-Lee. Decentralization : The Future of Online Social Networking. In *Proceedings of the W3C Workshop on the Future of Social Networking*, 2009.

Hao Yin, Yong Jiang, Chuang Lin, Yan Luo, and Yunjie Liu. Big data: Transforming the Design Philosophy of Future Internet. *IEEE Network*, 28(4):14–19, jul 2014. ISSN 08908044.

Peng Yue, Chenxiao Zhang, Mingda Zhang, and Liangcun Jiang. Sensor Web Event Detection and Geoprocessing over Big Data. In *Proceedings of the IEEE International Conference on Geoscience and Remote Sensing Symposium*, 2014.

Miao Yun and Bu Yuxin. Research on the Architecture and Key Technology of Internet of Things (loT) Applied on Smart Grid. In *Proceedings of the International Conference on Advances in Energy Engineering*, 2010.

Matei Zaharia. An Architecture for Fast and General Data Processing on Large Clusters. ACM Books, 2016. ISBN 978-1-97000-157-0.

Andrea Zanella, Nicola Bui, Angelo P Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of Things for Smart Cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014. ISSN 2327-4662.

Schubert Zhang. HFile: A Block-Indexed File Format to Store Sorted Key-Value Pairs, Retreived June 2017, from http://cloudepr.blogspot.com/2009/09/hfile-block-indexed-file-format-to.html, sep 2009.

Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean Paul Calbimonte. SRBench: A streaming RDF/SPARQL benchmark. In *Proceedings of the 11th International Semantic Web Conference*, 2012.

Yanxu Zheng, Sutharshan Rajasegarar, Christopher Leckie, and Marimuthu Palaniswami. Smart Car Parking: Temporal Clustering and Anomaly Detection in Urban Car Parking. In *Proceedings of the 9th IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, 2014.

Xinghui Zhu, Fang Kui, and Yongheng Wang. Predictive Analytics By Using Bayesian Model Averaging for Large-scale Internet of Things. *International Journal of Distributed Sensor Networks*, 2013. ISSN 15501329.