2018-12-20

An Open Geospatial Internet of Things Cloud Service Architecture Based on the Big Data Lambda Architecture

Khalafbeigi, Tania

UNIVERSITY OF CALGARY

An Open Geospatial Internet of Things Cloud Service Architecture Based on the Big Data Lambda Architecture

by

Tania Khalafbeigi

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN GEOMATICS ENGINEERING

CALGARY, ALBERTA

DECEMBER, 2018

Abstract

The Internet of Things (IoT) consists of sensors and actuators embedded in everyday devices interconnecting and communicating through interoperable information and communication technologies. The real potential of IoT is in creating innovative applications by integrating and repurposing IoT sensing and controlling capabilities from different sources. However, proprietary IoT systems now create silos that make the IoT goal almost unreachable as the applications need to deal with heterogeneous data from different systems. In addition to the problem of heterogeneity, big data is a challenge for all technologies in the modern world. As predicted by CISCO and IDC, the number of internet-connected objects will reach at least 50 billion by 2020. As a result, IoT is facing heterogeneity and big data challenges including volume and velocity. We have proposed an architecture for IoT with the focus on data management challenges in this dissertation. The proposed architecture merges the Lambda architecture with the SensorThings API. The SensorThings API is used as a solution for the heterogeneity problem. One of the solutions for data heterogeneity or so-called interoperability in IoT is the use of a standard API. SensorThings API has been proven to be a mature, open geospatial standard for IoT by various literature, implementations, and its widespread adoption. Moreover, the Lambda architecture addresses big data volume and velocity challenges through the use of three layers architecture: batch, serving, and speed. We implemented a case study of our proposed architecture with real air quality data. For our implementation, we used Hadoop and Azure technologies. Our case study showed that our proposed architecture significantly improves the performance of IoT service on real-world big open data.

Keywords: Internet of Things, Big Data, SensorThings API, Lambda Architecture

Acknowledgements

I would like to thank the following individuals who supported me and did not leave me alone in all aspects of my research:

- Dr Steve Liang for his guidance, financial support, and excellent supervision.
- Dr Xin Wang, Dr Reda Alhajj, Dr Mea Wang, and Dr Kyle O'Keefe for providing useful comments about my research during my candidacy exam.
- Charlene K for helping me polishing the English of this dissertation
- Mahdi, my husband, and also my parents for their spiritual support.

I would also want to thank my colleagues in GeoSensorWeb lab for all their help when we worked on SensorThings API design, especially Alec Huang and Mohammad Jazayeri.

Last, but not least, I would like to thank SensorUp Inc. for sponsoring my implementation on Microsoft Azure.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Abbreviation	Definition
ACID	Atomicity, Consistency, Isolation, and Durability
API	Application Programming Interface
CRDT	Conflict-free Replicated Data Type
CRUD	create, read, update, and delete
DAG	Directed Acyclic Graph
DBMS	Database Management System
DDS	Distributed Data System
DHS	Department of Homeland Security
DSS	Decision Support System
DW	Data warehouse
ETL	Extract Transform Load
HDFS	Hadoop Distributed File System
HiveQL	Hive Query Language
ICT	Information and Communication Technology
ID	identifier
IDC	International Data Corporation
IoT	Internet of Things
IoV	Internet of Vehicles
ITU	International Telecommunication Union
JSON	JavaScript Object Notation
LASS	Location Aware Sensing System
MAC	Media Access Control
MDSD	Model Driven Software Development
MPP	Massively Parallel Processing
MQTT	Message Queuing Telemetry Transport
NoSQL	Not only SQL
O&M	Observation and Measurement
OGC	Open Geospatial Consortium
PaaS	platform as a service
PDBMS	Parallel Database Management System
RDBMS	Relational Database Management System
REST	Representational State Transfer
SCN	Service-Controlled Networking
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SOS	Sensor Observation Service
SQL	Structured Query Language
SSN	Semantic Sensor Network
SWE	Sensor Web Enablement
XML	Extensible Markup Language

Introduction

Billions of small sensors and actuators will be embedded in everyday objects and connected to the Internet forming a concept called the Internet of Things (IoT) (Arlitt et al., 2015; Evans, 2011). In the concept of IoT, there are a great many "things" that are connected to each other using wireless or wired connections with unique addressing schemas. With the help of these connections, "things" can interact and cooperate with each other in order to create new applications (Vermesan & Friess, 2013).

The International Telecommunication Union (ITU) (International Telecommunication Union, 2012) defines IoT as "*a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies*". IoT-enabled objects can sense their environment, collect information, and communicate and interact with each other. In the context of IoT, the "things" in the physical world are identifiable and capable of being integrated into communication networks. By populating our environment with real-world sensor-based devices, the IoT is opening the door to a variety of application domains, such as environmental monitoring, transportation and logistics, urban informatics, smart cities, as well as personal and social applications (S. Liang, Bermudez, Huang, Jazayeri, & Khalafbeigi, 2013).

The observation data collected by "things" in IoT are different depending on the sensors that produce them (e.g. temperature, humidity, sound, light, etc.). Along with these differences in types, the data is collected using different structures (can also be unstructured). Furthermore, the

high frequency of producing data by a large amount of "things" results in IoT data deluge. The heterogeneity, ubiquity and streaming nature of IoT data make data management a challenging task in IoT (Barnaghi, Sheth, & Henson, 2013). Data management is a key research topic for IoT and it plays a crucial role in the effective operation of IoT (M. Ma, Wang, & Chu, 2013). Data management in IoT includes the tasks of data collection, integration, cleaning, storage, processing, analysis, and visualization (Mishra, Lin, & Chang, 2015). For this thesis, I focused on data storage, processing and analysis in IoT; other data management tasks are out of the scope of this dissertation.

For this dissertation, I proposed an architecture for IoT with a focus on data storage, processing and analysis challenges. I divide the solution into two sub-solutions:

- We (GeoSensorWeb Laboratory[1]) propose using the SensorThings API (S. Liang, Huang, & Khalafbeigi, 2016) for IoT services in order to address the challenge of heterogeneous data for IoT.
- I propose applying the Lambda architecture (Marz & Warren, 2015) over the SensorThings API in order to overcome storage, processing and analysis challenges for IoT.

[1] http://sensorweb.geomatics.ucalgary.ca/

The next subsection explains IoT data management challenges that are what this dissertation aims to solve. Then, the objectives are explained in detail followed by a brief discussion as to how our proposed solution fits the objectives.

Challenges of Data Management in IoT

The nature of IoT data causes different challenges for IoT data management. IoT data are inaccurate, heterogeneous, massive, real-time, and also have implicit semantics (Ding, Yang, & Wu, 2011; M. Ma et al., 2013). Also, IoT data has spatiotemporal attributes (Ding et al., 2011). Based on these data characteristics, data management in IoT can be categorized as data cleaning; data (pre)processing; data storage and analysis; and handling security and privacy (Ding et al., 2011; T. Fan & Chen, 2010; M. Ma et al., 2013; Mishra et al., 2015). For this dissertation, IoT data management is discussed from a big data perspective which covers data preprocessing, and data storage and analysis. In the following subsection, big data management for IoT is elaborated on.

IoT Big Data Management

The term big data in the context of information technology is often thought of as managing large datasets. However, although managing large volumes is an important challenge in the big data world, big data does not only mean big data volume but also big data velocity and variety. Volume, Velocity and Variety are called the three “V”s of big data management challenges (Russom, 2011) as shown in Figure 1.

With the rapid development of IoT and the increasing number of IoT-enabled objects, it has been observed that IoT also encounters big data challenges. As such, IoT data has been widely used in literature as a typical example of big data challenges (W. Fan & Bifet, 2013). An explanation of the basic concept of the three "V"s big data model and how IoT fits into this model is given below. (C.-Y. Huang, 2013)

Data Volume: The most obvious characteristic of big data is the large data volume, which refers to large datasets in terms of size or number of data records. As predicted by CISCO (Evans, 2011) and the International Data Corporation (IDC) (Arlitt et al., 2015), the number of internet-connected objects will reach at least 50 billion by 2020. Each of these sensors registers its observations frequently and even managing some of the data from the 50 billion devices is challenging with today's data management systems. It is foreseeable that with the increasing number of IoT-enabled "things", IoT will be generating more and more data every day. As a result, storing and retrieving the large volume of data will be a major challenge for IoT.

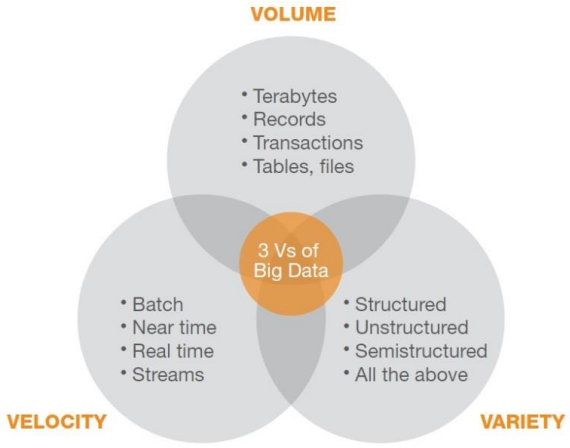

Figure 1 Big Data Three "V"s (Russom, 2011)

Data Velocity: Data velocity refers to the high rate of data production. In the context of IoT, sensors are used to detect interesting events and higher sampling frequency will reduce the chance of missing important events. As a result, billions of future IoT sensors will produce data at high frequencies. Sound pressure sensors are examples that typically register their readings every second. Another example is the Boeing jet engines that produce ten terabytes of sensor data every 30 minutes during flights (C.-Y. Huang, 2013). In addition, an effective IoT data management system not only needs to store and index the continuous sensor data streams flowing into the system but also needs to continuously answer queries about these data streams. As a result, efficiently processing high-velocity data streams is another big data challenge faced by IoT.

Data Variety: Data variety refers to managing different datasets with a large variety of characteristics. As shown in Figure 1, the different characteristics can be in terms of data structures, incompatible data formats, and different data interfaces (Russom, 2011). Sensor data is relatively structured in comparison to social media data. However, sensor data varies greatly in terms of hardware, data types, observed phenomena, communication protocols, data encodings, semantics, and syntaxes (C.-Y. Huang, 2013). For example, there are different kinds of temperature sensors from different vendors and their observations need to be organized in a consistent fashion by IoT. Each of these sensors has their own IoT service provider that develops and uses its own proprietary software interface, encodings, and ontologies. This means that the

number of proprietary interfaces are growing as the number of IoT devices increases. As a result, the IoT sensor data may come from different sources with different interfaces and structures (or may even be unstructured). Thus, effectively integrating heterogeneous data in order to provide a coherent view for innovative applications is another challenge for the IoT world.

Objectives and Proposed Solutions

In this subsection, I first clarify the objectives before explaining the proposed solutions to achieve the objectives.

High Level Objective: To propose an architecture that can address data management challenges for the Internet of Things from the big data perspective.

My overall objective is to address data management challenges by proposing an architecture for IoT. To better explain the proposed solution, I divide the overall objective into the big data three "Vs" and then explain the proposed solution for each objective.

Objective 1: To Propose a Solution for the Data Variety Challenge for IoT

OGC SensorThings API is an OGC standard that proposes an API for IoT services to monitor and control IoT devices (i.e. sensors and actuators). The SensorThings API applies Representational State Transfer (REST) -like architecture and uses HTTP protocols for its communication. The overall objective of the API is to make IoT services interoperable so that the real value of IoT can be achieved by creating innovative applications for these services. In other words, the data variety problem for IoT can be overcome by using this API.

The SensorThings API proposes a standard solution for IoT-enabled devices to interact with each other and through the Web using the JSON data format. The JSON format is preferable to the Extensible Markup Language (XML) format, because it is lightweight, simple, and efficient for presenting data in the server. In other words, it is easy to use (S. Liang et al., 2016).

For the SensorThings API, a REST-like service interface is used for accessing the resources. With this API, each resource is identified using a unique identifier. As a result, each resource can be accessed uniquely without the need to know its related resources. The SensorThings API supports four basic operations for all the resources: create, read, update, and delete (CRUD). Since the API uses HTTP protocol, it uses HTTP POST for create; GET for read; PUT and PATCH for update; and finally DELETE for delete.

The SensorThings API consists of three major parts which are: Sensing, Tasking, and Rules engine parts. The Sensing part defines an interoperable framework to manage and access sensors and observations. The Tasking part defines an interoperable framework for managing the actuators and submitting tasks to them. Finally, the Rules engine part defines events as the connection point between the Sensing and Tasking parts. The Sensing part was published in July 2016 and the Tasking part will be published in late 2018. The Rules engine part is a work in progress and is planned for 2019. The focus of this dissertation is on the Sensing part.

The Sensing part contains following resources: Things, Location, HistoricalLocation, Datastreams, Sensors, ObservedProperties, Observations, and FeaturesOfInterest. A Thing is an IoT-enabled object and has a current Location and may have some previous locations in

HistoricalLocations. It can also have one or more Datastreams. A Datastream is a mechanism for grouping Observations with the same Observed Property and Sensor. The Sensors of the IoT-enabled object produce results (or readings) with values that are an estimate of an ObservedProperty of the FeatureOfInterest. These readings are called Observations in the SensorThings API.

The SensorThings API provides a standard framework for IoT services. As a result, even if each IoT-enabled object has its own implementation of this API as a software interface, integrating the data from these services to create innovative applications is straightforward. In summary, we believe that SensorThings API provides sematic interoperability through its data model and syntactic interoperability with its REST API. Thus, we believe that by using the SensorThings API for IoT services we can address the data variety challenge for IoT.

I propose using Lambda architecture (Marz & Warren, 2015) on top of of the SensorThings API to address the data volume and velocity challenges. Lambda architecture is a generic, scalable and fault-tolerant data processing architecture that can be used for real-time data processing. It contains three layers. The batch layer manages the master datasets and creates batch views. The serving layer indexes the batch views and prepares them for querying. Finally, the speed layer deals with real-time data processing and query answering. The following explains how using the different layers of Lambda architecture can address data volume and velocity challenges for IoT.

Objective 2: To Propose a Solution for the Data Volume Challenge for IoT

I propose using the batch and serving layer to address the data volume challenge for IoT. The batch layer responsibilities are: 1) storing an immutable, constantly growing master dataset, and 2) computing arbitrary functions on that dataset and creating batch views. The serving layer indexes these precomputed batch views so that they can be efficiently queried. In other words, the serving layer makes batch views queryable. The serving layer continuously swaps in new versions of batch views that are periodically computed by the batch layer. Since the batch layer processing takes at least a few hours, the serving layer is updated at most once every few hours. (Marz & Warren, 2015)

I propose to use the batch layer together with the serving layer to address the data volume challenge. Different kinds of technology can be used to organize the master dataset and batch views which contain large amounts of data from sensors and actuators. NoSQL (Not Only SQL) data stores and Apache Hadoop[2] are the canonical example of batch processing systems (Marz & Warren, 2015).

Objective 3: To Propose a Solution for the Data Velocity Challenge for IoT

I propose using the speed layer in Lambda architecture to address the data velocity challenge for IoT. The serving layer updates whenever the batch layer finishes precomputing

[2] http://hadoop.apache.org/

batch views. As a result, data that is received during batch views precomputation is not represented in batch views. If the service answers the queries only based on the serving layer, the response would not contain real-time data. To overcome this problem, Lambda architecture has a dedicated real-time data system (arbitrary functions computed with arbitrary data in real-time) named speed layer.

One of the strengths of the Lambda Architecture is that once data makes it through the batch views and is loaded onto the serving layer, the corresponding results for the real-time views will be removed from the speed layer. This means real-time views that are no longer needed can be discarded from the speed layer frequently. It makes the architecture more fault-tolerant, since the speed layer is much more complex than the batch and serving layers and the probability of faults for real-time view is more than for batch view.

The speed layer is more complex than the batch layer because the whole master dataset is processed each time to create batch views for the batch layer, but with the speed layer, real-time views are created using incremental computation which is much more complex. Moreover, since calculation performance is a major factor for the speed layer, heuristic methods may be used to calculate the view approximation. As a result, fault occurring is more probable for the speed layer than batch layer. However, even if a fault occurs for the speed layer, it will soon be replaced with the correct information in the batch views. To address the data velocity challenge, the speed layer is used to provide real-time sensor and actuator data processing and query answering.

I propose using the SensorThings API in order to solve the data variety challenge. To complete our solution for data management, I propose using the Lambda architecture (Marz & Warren, 2015) on top of the SensorThings API to address both data volume and velocity challenges.

I believe that an IoT system that is implemented based on the SensorThings API and using the Lambda architecture can address data challenges as a whole, i.e. variety, volume, and velocity.

In the second chapter, we explain the SensorThings API in detail along with its related work. Then in the third chapter, we discuss big data management for IoT as well as Lambda architecture. We also review the related literature for big data management for IoT. The fourth chapter discusses how to merge the SensorThings API with Lambda architecture to create an open geospatial architecture for addressing big data challenges for IoT. We discuss a case study implementation of the proposed architecture in chapter five, as well as the experiments to show that the proposed architecture is suitable for addressing the big data three Vs challenges for IoT. Finally, chapter six provides the conclusion and discusses future work.

SensorThings API, Details and Design Decisions

We use the SensorThings API in our proposed architecture to address the big data variety challenge or so-called interoperability. In this chapter, we will talk about the OGC SensorThings API in detail. The first section is an introduction to the requirements behind the SensorThings API and where the need for the standard arose. Then, we discuss the place of the SensorThings API in the IoT reference model. That is followed by a discussion on work related to the SensorThings API. Finally, we explore details about SensorThings from its data model to its flexible REST-like API and different extensions.

Introduction

Similar to Web 2.0 (Hinchcliffe, 2006), the real potential of IoT is in creating innovative applications by repurposing and assembling the IoT sensing and controlling capabilities from different sources in novel, effective and sometimes unexpected ways. However, today's IoT service providers are developing and using their own proprietary software interfaces. Proprietary systems are called stove pipes or vertical silos and cannot be combined or extended easily (Ahlgren, Hidell, & Ngai, 2016). Using these proprietary systems for IoT results in a vendor locked in problem and makes creating integrated innovative applications for IoT very difficult or almost impossible (Ahlgren et al., 2016).

An example of proprietary IoT systems is a smart lighting system that only works with light bulbs from the same vendor. These systems are usually designed as end-to-cloud-to-end systems and the cloud is controlled by the vendor. Using these end-to-cloud-to-end systems for IoT

introduces interoperability issues between different IoT systems (Ahlgren et al., 2016). Lack of interoperability limits the growth of marketing (Fältström, 2016) and IoT innovative applications.

The number of proprietary interfaces are growing as the number of IoT devices increases. Consequently, the effort required to interconnect different IoT devices for innovative applications is growing exponentially. A standardized interface for IoT sensors is a key solution to this problem. There is a need for an open standards-based interoperable Web Application Programming Interface (API) that allows different IoT sensing devices and applications to interoperate. In other words, in order to address the interoperability issue, we need horizontally designed IoT systems with well-defined open standard APIs instead of vertical silos (Ahlgren et al., 2016).

Our GeoSensorWeb Laboratory proposed an Open Geospatial Consortium (OGC) standard called “SensorThings API”. I was responsible for designing the data model and API (together with other members) and also for implementing the world’s first prototype system. We worked on the design and development of a REST-like API that can overcome the above-mentioned IoT interoperability issues. The goal was to capture the observations and controlling capabilities from IoT devices and make them easily available through data aggregation portals (e.g., cloud-based IoT platforms).

SensorThings API can be the building block to achieving IoT interoperability, enabling us to address the IoT data variety challenge. IoT devices can simply connect to such a service and

sensing devices can upload their observations to the service by simply using the HTTP POST request. Moreover, IoT controlling devices also can be controlled and tasked by users through the service. Such a service has the flexibility of updating and deleting the uploaded data using simple HTTP PUT/PATCH and DELETE requests. The service implemented based on SensorThings API accepts the JavaScript Object Notation[3] (JSON) format for the input data and also GeoJSON[4] for uploading location information. In addition, the retrieved data from the service also uses the IoT data model encoded in JSON encodings.

The service implemented based on the SensorThings API supports the use case as shown in Figure 2. The use case starts with some IoT devices registering themselves to the service. For the sensing devices, registration information contains the phenomenon observed by one or many sensors. As the IoT devices can accept tasks and be controlled, they can also register and publish their tasking capabilities to the service. After registration, sensing devices can start uploading their observations to the service. Users can then access those observations and send controlling tasks to the controlling devices through the service. All the scenario functionalities follow the REST-like architecture, i.e. using the HTTP verbs (i.e., GET, POST, UPDATE, and DELETE).

[3] http://www.json.org/
[4] http://geojson.org/

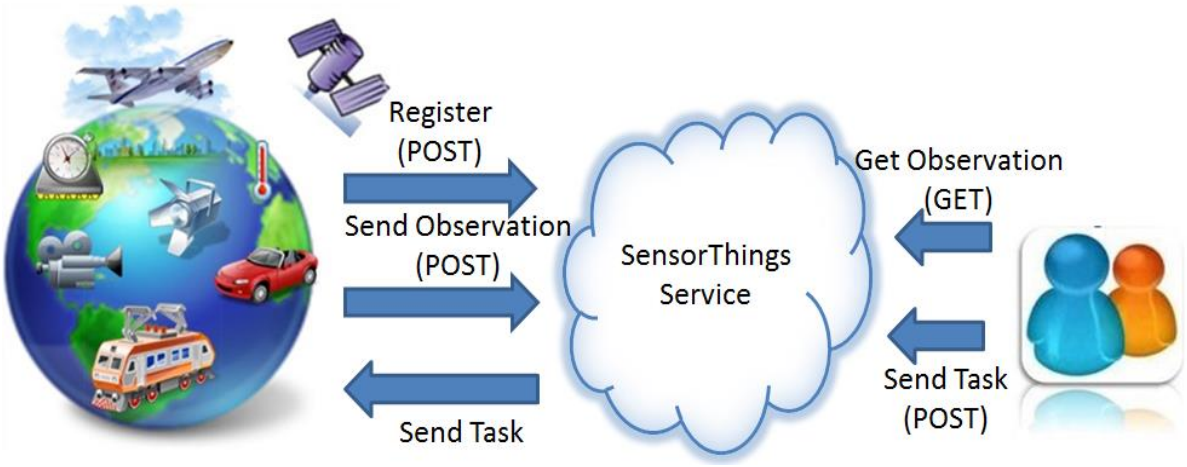

Figure 2 SensorThings Service Sample Use Case

Based on standard interfaces for IoT sensors, such as SensorThings API, a Web of Things vision can be realized and cloud services can be built to serve a wide variety of sensors all over the world. Such services need to handle a very high throughput rate (i.e. a very large number of requests per second), since many sensors will frequently register their observations and many requests will also be sent from applications using their sensor data for different applications. Moreover, the service needs to scale with the growing amount of data it stores without sacrificing performance. To sum up, IoT faces data volume and velocity challenges. Thus, we need to design an architecture for global IoT devices that can efficiently store and retrieve a large number of sensor data and also handle a large number of queries with different types in a very efficient manner.

OData and SensorThings

OData (Chappell, 2011) defines an abstract data model (Handl, Pizzo, & Biamonte, 2014) and a protocol (Pizzo, Handl, & Zurmuehl, 2014) that lets any client access information exposed by any data source. OData is a general-purpose data access mechanism. This means that it provides a simple and easy way for different types of clients to access a wide variety of data. The OData protocol is based on REST and HTTP. It also defines the method for modelling the data as well as how to query them.

SensorThings API is based on OData in general. It defines the IoT data model for OData and adds geospatial queries as well as extended functionalities such as Message Queuing Telemetry Transport (MQTT) and Data Array to the OData basics. To sum up, SensorThings uses the OData proven data access mechanism, while customizing and extending it for IoT systems.

IoT Reference Model and Place of SensorThings

In this Section, we first overview the ITU-T IoT reference model (illustrated in Figure 3) (International Telecommunication Union, 2012) before discussing where the place of SensorThings would be in this architecture.

ITU-T defines the IoT reference model as having four layers: Application; Service and Application Support; Network; and Device. There are also management capabilities and security capabilities associated with the four layers.

The Application layer is where IoT applications reside. Physical and virtual sensor devices as well as their gateways are part of the Device layer. The Network layer consists of the medium for the sensor data to be transferred over the Internet or Network. There are different protocols that are popular for the IoT Network layer including, but not limited to, HTTP, MQTT, and CoAP.

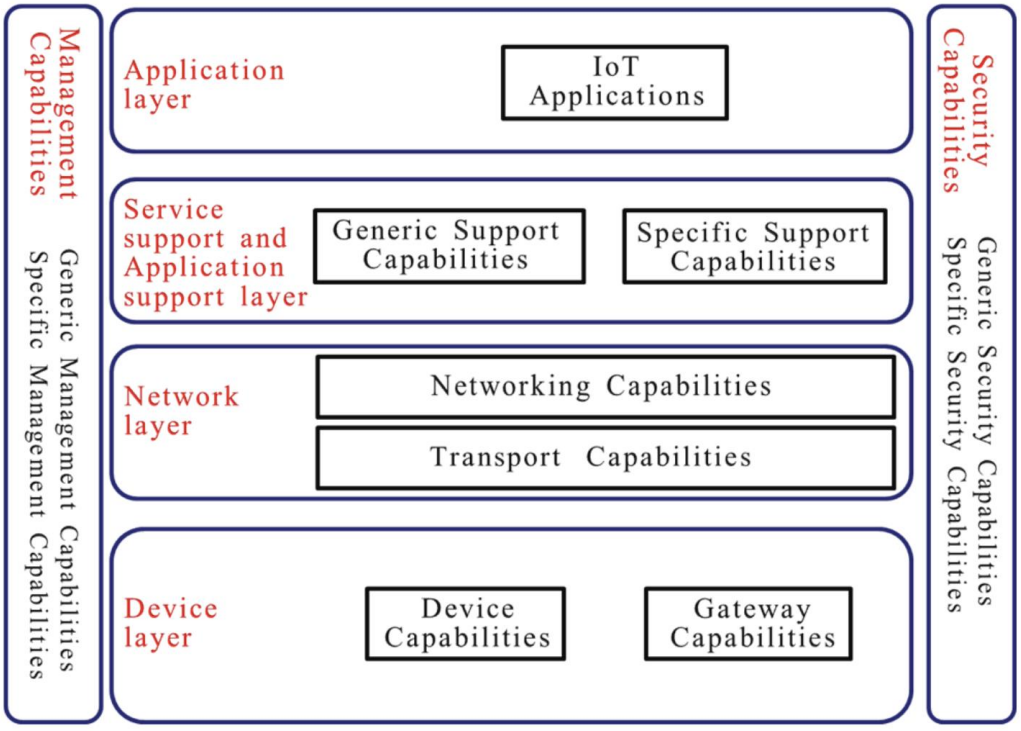

Figure 3 IoT Reference Model (Adapted from (International Telecommunication Union, 2012)**)**

The Service and Application Support layer is where the interfaces for devices and applications reside. IoT data processing and storage is also part of this layer. We can see that

SensorThings is a standard for the Service and Application Support layer, consisting of a standard data model for data storage, a standard REST-like interface for IoT applications, and standard HTTP and MQTT interface for devices.

Related Work

The OGC SensorThings API was approved on February 2016 and published in July 2016. The SensorThings API has been very well-received and has become popular in a short period of time due to its powerful data model and interface. In two years, there have been multiple implementations of the API for clients to use. In the GeoSensorWeb lab, we implemented the first prototype implementation for SensorThings API. SensorUp Inc., a Calgary-based startup implemented the first reference implementation of the SensorThings API as it was being approved. Mozilla has started an IoT project based on the SensorThings API and they are developing their implementation of the API. CGI is developing a SensorThings API implementation called Kinota in conjunction with the University of Lafayette. Fraunhofer developed an open source reference implementation and Geodan is also implementing a SensorThings API called GOST that is written in Go language. Other companies such as Compusult and 52North are also working on their SensorThings implementation. These different implementations not only represent the maturity of the standard, but also illustrate the absence of the vendor locked in issue when using the SensorThings API standard for IoT.

In order to review the literature, we divided them based on their context. First, we discuss other solutions proposed for the IoT interoperability challenge. We further divide this category

into two subcategories: adaptor/translator solutions and standard platform solutions. Then, we review literature that used and reviewed SensorThings.

IoT Interoperability Challenge

In this subsection, we discuss different solutions proposed for the IoT interoperability challenge in available literature. The solutions can be categorized into two subcategories: adaptor/translator solutions, and platforms.

Adaptor/Translator Solutions

An IoT challenge in the modern world is the increasing number of proprietary systems with their own protocols and structures. One category of solutions tries to provide an adaptor between all the proprietary solutions and connect all the different types together by defining an adaptor for each of the proprietary systems. IFTTT[5] (If This Then That) is an example of such a solution. IFTTT defines different connectors. For example, it has a connector that acts on Philips Hue if a certain event happens from Nest. These solutions work, but they are not optimal solutions as the effort to develop such solutions increases exponentially with the increase of the number of IoT systems. We call these solutions adaptor/translator solutions in this dissertation.

[5] https://ifttt.com/

Apple HomeKit[6], Google Home[7], and Amazon Alexa[8] are other examples in this category that attempt to provide many adaptors/connectors for different IoT systems in order to provide a unified system in which several IoT systems can work together. Ninja Blocks[9] can also be categorized in this category, although it has a very limited number of system adaptors.

The problem with all the solutions in this category is that as the number of proprietary IoT systems grows, the effort required for creating adapters grows exponentially. As a result, although these solutions might seem to be working well now, they are not the ultimate solution for the interoperability challenge. Moreover, the geospatial aspect of the SensorThings API is a unique feature that other solutions lack. In addition, it is worth mentioning the vendor locked in challenge that happens when using these adaptor systems. If the vendor shuts down their adaptor service for any reason, moving to another system could be extremely expensive, if not impossible.

[6] https://www.apple.com/ca/ios/home/
[7] https://store.google.com/product/google_home
[8] https://developer.amazon.com/alexa
[9] https://ninjablocks.com/

IoT Platforms and Standards

The other category of solutions emphasizes the standard approach. These solutions define standards and try to define a unified way that can be used by different IoT Systems. Xively[1], ThingSpeak[1] and ThingWorx[1] are IoT platforms that can be used for IoT systems. However, these are not standards and they provide platforms which can result in a vendor locked in problem.

The Xively platform does not have a concrete data model. It allows users to define their device template and feed sensor data into that template. This approach can be useful for device owners to record and analyze the data from their device. However, when it comes to aggregating data from multiple systems and creating integrated applications, the lack of a coherent data model creates problems. Users may not be expert enough to define the device template and may find out in future that there is more information that they need to gather. Xively can be useful as the IoT middleware platform for individual businesses but the open data model would still result in silos of data. In other words, it does not address the data variety issue properly. However, the SensorThings API has a well-defined data model. Moreover, as it is an API and has multiple

---

[1] https://xively.com/

[1] https://thingspeak.com/

[1] https://www.ptc.com/en/products/iot

different implementations, all tested and compliant to the standard, there will not be any vendor locked in issue.

ThingSpeak defines a data model and does not have the Xively problem. However, its data model is very simple and may not be comprehensive enough. One of the problems is the lack of a unit of measurement. Without a unit of measurement, the data is only useful for its owner who already knows the unit of measurement of his/her device. However, the data cannot be aggregated with data from other devices for creating integrated applications. We can say that the sensor reading is meaningless without its unit of measurement. Also for the geospatial feature, it does not use the standard way and only keeps latitude and longitude without the information about projection. All in all, it may be simple and easy and might be useful for device owner not to worry about the data management platform, but it does not solve the interoperability issue for IoT. On the other hand, the SensorThings API focuses on the interoperability and comprehensive data model as well as simplicity of use. Moreover, for the geospatial feature, the SensorThings API is flexible enough to use different standards. The current suggestion is GeoJSON which is standard JSON presentation for geospatial information. However, the data model is defined in such a way that it can be used with other standards as well, e.g. OGC IndoorGML (K.-J. Li et al., 2015).

In comparison with Xively and ThingSpeak, ThingWorx is better in term of data model as it has some predefined templates that can be inherited by users. However, as it is visible from the website, it is a total industrial platform, and focuses on a solution rather than a standard. Vendor

locked in issue is definite problem in their case. As discussed before, in comparison to ThingWorx, the SensorThings API has a well-defined data model and different implementations to choose from to prevent the vendor locked in problem.

The OGC Sensor Observations Service (SOS) (Bröring, Stasch, & Echterhoff, 2012) is a standard API for sensor networks that can be used for IoT. It is the closest in comparison to the SensorThings API. Basically, SOS and the feedback from its users was one of the motivations for starting the work on the SensorThings API. There are two major issues with SOS. Firstly, it is complex and not easy to use. Secondly, the interface is not flexible in terms of accessing the data. SensorUp Inc. compared SOS to SensorThings API (SensorUp Inc., 2016) and the comparison is shown in Table 1.

Table 1 SOS and SensorThings Comparison (Adapted from (SensorUp Inc., 2016)**)**

	SensorThings API	**SOS**
Encoding	JSON	XML
Binding	REST	SOAP
Inserting New Sensors or Observations	HTTP POST or MQTT	Using SOS specific interfaces, *e.g.*, RegisterSensor(), InsertObservation()
Deleting Existing Sensors	HTTP DELETE	Using SOS specific interfaces, i.e., DeleteSensor()

Pagination	$top, $skip, $@iot.nextLink	Not Supported
Pub-Sub Support	MQTT	Not Supported
Updating Properties of Existing Sensors or Observations	HTTP PATCH	Not Supported
Deleting Observations	HTTP DELETE	Not Supported
Return Only the Properties Selected by the Client	$select	Not Supported
Return Multiple O&M Entities (e.g., FeatureOfInterest and Observation) in One Request/Response	$expand	Not Supported

As we can see from Table 1, there are multiple added functionalities for the SensorThings API as compared to SOS, e.g. CRUD operations on all entities. SensorThings API focuses on being easy-to-use and lightweight. As a result, it adopts JSON instead of XML and also includes support for MQTT. SensorThings also adds support for pagination which increases the server performance dramatically when the data size is large. In summary, SensorThings improves SOS by making it simpler and easier to use and by adding more functionality.

We looked at different adaptor/translator systems, as well as IoT platforms and standards in this section. In summary, SensorThings is superior in multiple ways compared to them. Firstly, it has a well-defined and comprehensive data model as well as a flexible interface for accessing data. Secondly, it is a standard API and multiple implementations are available for users to adopt and they can implement the API themselves. As a result, there is no vendor locked in issue. Also, SensorThings API focuses on being lightweight for resource constrained devices and supports JSON and MQTT for that purpose. SensorThings API keeps geospatial information for its Observations which is lacking in some of the comparative systems. Moreover, SensorThings keeps the geospatial information in a standard way by using GeoJSON and is flexible for the use of other standards such as IndoorGML.

SensorThings API in Literature

After SensorThings API was published as an OGC standard in 2016, it has been studied and discussed in various literature. Even before publication when SensorThing API was just a candidate standard, it was studied and compared in different literature (Gómez Maureira, Oldenhof, & Teernstra, 2014; Jazayeri, Liang, & Huang, 2015; Kotsev, Pantisano, Schade, & Jirka, 2015). The API was found to be compatible with any open or custom hardware (Gómez Maureira et al., 2014) and also easy to use (Kotsev et al., 2015). Moreover, the use of JSON encoding and REST-like API are practical and make the API lightweight (Gómez Maureira et al., 2014; Jazayeri et al., 2015; Kotsev et al., 2015).

There has been an attempt to map the SensorThings API to OpenIoT (van der Schaaf & Herzog, 2015). OpenIoT is an open source middleware implementation for supporting IoT applications. Semantic Sensor Network (SSN) Ontology is the core of OpenIoT which is influenced by Sensor Web Enablement (SWE) which is the base for the SensorThings API. As a result, the core concept of OpenIoT and SensorThings are related and this paper attempted to map their two data models. This paper states that the SensorThings API is easy to use and provides simple abstraction of IoT resources, despite SSN-ontology which is complicated. As a result, finding a bridge between OpenIoT to SensorThings would lead to simplicity and ease of use. This paper shows the value of the SensorThings API as it tries to map another concept, OpenIoT, to the simple and easy to use standard, SensorThings.

After the SensorThings Sensing part was published, work was started on the Tasking part. There was an attempt to extend the API with Tasking in some literature (C. Y. Huang & Wu, 2016b, 2016a) as well. The SensorThings API Tasking part has been approved and will be published soon.

There is recent work on mapping INSPIRE to the SensorThings API (Kotsev et al., 2018). These mappings show the value and adoption of SensorThings API as a standard, as other systems start to move to this standard by mapping their current models. The INSPIRE Directive is a European Union spatial data infrastructure that enables the sharing of environmental spatial information across Europe and mapping this infrastructure to SensorThings API has proven valuable.

There is research that was very recently published about the interoperability of the SensorThings API (Teixeira, 2018). This master thesis developed an application based on multiple implementations of SensorThings API as a proof of concept about the interoperability of the SensorThings API. This work considers the SensorThings API as easy-to-use and a good fit for addressing the heterogeneity challenge for IoT.

Moßgraber et al. in (Moßgraber, Hilbring, van der Schaaf, et al., 2018) focused on crisis management and how data from different sources need to be aggregated in Decision Support Systems (DSS) for managing a disaster which leads to a data heterogeneity issue. They came to the conclusion that the SensorThings API is a helpful solution that is harmonizing heterogeneous IoT data for different processing services for crisis management. There is also other research (Moßgraber, Hilbring, Pouli, & Padeletti, 2018) for creating a knowledge base for cultural heritage protection against climate change in which the ontology is defined on top of the SensorThings API. These two research studies are part of the European Horizon 2020 project. The smart emission (Grothe, Carton, Van Den Broecke, Volten, & Kieboom, 2016), mySMARTLife (mySMARTLIfe Consortium Partners, 2017), and analysis of sensor signals for monitoring of heritage buildings (Watson, Kunz, van der Schaaf, & Ubertini, 2018) projects are also part of the Horizon 2020 project that uses the SensorThings API as the standard for IoT.

Hussain and Wu in (Hussain & Wu, 2018) designed the Information and Communication Technology (ICT) framework for sustaining the interoperability by applying the Model Driven Software Development (MDSD) paradigm and ontology. In order to validate their framework,

they chose to use the SensorThings API and apply their ontology framework on top of it – which shows the value of the SensorThings API in addressing the interoperability challenge.

Trilles et al. in (Trilles et al., 2017) used the SensorThings API as the interoperable IoT service for developing their Sense Our Environment (SEnviro) platform. This project is a smart city system which develops the entire IoT stack from device to application and the SensorThings API is used as the IoT standard for their service layer.

Lue et al. in (Luo, Saeedi, Badger, & Liang, 2018) used the SensorThings API as the IoT platform for monitoring human and animal use of industrial linear features. They created some devices for counting animals and humans and for collecting the data. In order to persist and access their sensor data, they chose to use the SensorThings API as an interoperable IoT standard.

In summary, we can see that the SensorThings API appears in various literature even before it was officially published and it shows that the standard was well-received by the IoT community in a short amount of time. Apart from literature, the SensorThings API has been implemented many times thus far, as mentioned before, and it has been adopted for multiple industry and research IoT projects – showing the maturity of the standard. These implementations and adoption will be discussed in more detail in the Results and Discussion chapter. The next sections of this chapter elaborate on the SensorThings data model and interface in detail.

Data Model

As mentioned before, the SensorThings API has three different parts – the Sensing part, Tasking part, and Rules engine part. Only the Sensing part has been officially published. In this section, we will discuss details of the Sensing part which is the focus of this dissertation.

As we have seen in previous sections, a standard and comprehensive IoT data model, together with geospatial enablement, differentiates the SensorThings API from other literature and research work on IoT. We believe that IoT systems with a variety of requirements can be easily modeled with SensorThings API for different use cases. In this section, we will focus on the SensorThings API data model and design decisions for each of the data model entities.

The SensorThings API follows the OGC Observation and Measurement (O&M) specification (Cox, 2011) in general. Figure 4 shows the SensorThings Sensing part data model. The data model for the SensorThings Sensing part contains eight entities. There are two common properties for all of these entities (except Observation as we will see later): description and name. Description is a short description of the entity and name is a descriptive label for the entity. Observation does not have these properties as it is the reading of the sensor and all its details can be found from related entities.

The main entity, which is also the connection part between the Sensing and Tasking parts, is a *Thing*. A *Thing* for SensorThings has the same definition as Thing for IoT. Put simply, the SensorThings *Thing* is a physical or virtual sensing and tasking device, which means that it can have multiple sensors and actuators. A *Thing* can be different for the same IoT system based on

use cases and application requirements. As an example, consider a smart scale. For the scale manufacturer, the *Thing* is the scale. But from the perspective of a user who uses this scale together with other health and tracking devices for tracking his/her health, the *Thing* is the user and the scale is just one of the sensors measuring information for that user. As a result, the most important factor for modeling the *Thing* is the application requirements and use cases.

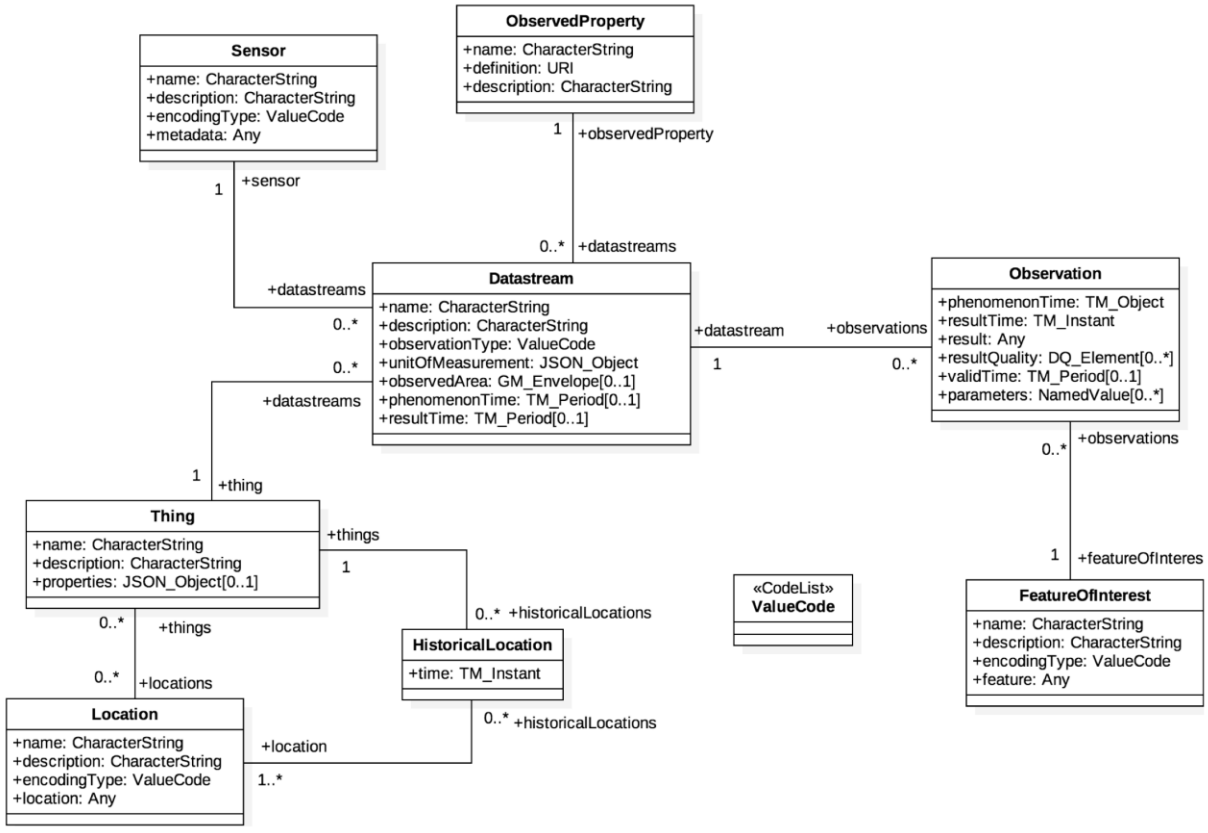

Figure 4 SensorThings Data Model (Adapted from the Standard Specification (S. Liang et al., 2016)**)**

Aside from the name and description that are common in most of the entities, *Thing* has a properties property. Properties is a JSON and can contain arbitrary information about the *Thing*. As a result, it may not be searchable amongst all Things of the system.

A *Thing* can have a *Location*. The *Location* is the current physical position of the *Thing*. As the SensorThings is a standard, it uses a standard way for storing *Location* which is GeoJSON. GeoJSON (Butler et al., 2016) is a geospatial standard for recording location. However, there are other standards for more sophisticated situations such as indoor positioning. OGC IndoorGML (K.-J. Li et al., 2015) and CityGML (Kolbe et al., 2012) is one of the examples. In order to make the system unified and queryable across the whole system, as well as across multiple systems, GeoJSON is used as a primary type for *Location*. But if you need to use other standards such as IndoorGML, you are allowed to link more than one *Location* entity to a *Thing*. In other words, a *Thing* can have more than one *Location*, and they are all different representations of its current location.

SensorThings is flexible with its use of the *Location* entity. If two *Things* have the same *Location*, a user can either reuse the *Location* entity or create a new one with the same information. However, there are best practices in order for the system to be the most useful from an application point of view. For the SensorThings server developers it is best practice to implement the system in a way that reuses the *Location* entities in case there are two identical *Location* entities. For SensorThings users, if the server does not support the automatic reuse of *Location*, it is best practice to reuse *Location* entities by searching and, if possible, linking the

existing *Location* to their *Thing*. This way it will be easier for applications to find all the *Things* in a specific *Location* by accessing *Locations*({id})/*Things*. Even if these best practices are not followed, with the SensorThings rich query options, performing such a query is possible. It may only need more processing on the server to retrieve the result, which makes using this best practice even more important for server developers. The query will be explained in the Data Retrieval section.

Apart from name and description, *Location* has encodingType and location. EncodingType specifies which standard is used for representing *Location*'s location. Location is the position information encoded in encodingType. The major standard used for encoding location is GeoJSON.

There are two types of *Things* with regards to their *Location*: static and moving. Static *Things* are those that stay in the same position either for its whole life or for a long period of time. Examples of these *Things* are sensors and devices that are used at home. Moving *Things* are those that change their *Location* frequently. Sensors and devices connected to cars and public transit are examples of these *Things*.

For static *Things*, the current *Location* is all the information that we need. However, for moving *Things*, we need to keep track of previous *Locations*. For this purpose, SensorThings has an entity called *HistoricalLocation*. *HistoricalLocation* specifies which *Thing* was at which *Location* at what time. The only data it keeps is time and links to *Thing* and *Location*. Time shows the first time the *Thing* was seen at the *Location*. *HistoricalLocation* is one of the entities

that does not have name and description as they are not required. We emphasize that since *Location* is the current position of the *Thing*, we cannot keep track of moving *Things* without *HistoricalLocation*.

Thus far, we have looked at how to model a *Thing* together with its *Location* depending on whether it is moving or static. Now we will examine how to store readings/observations of the sensing devices. Each *Thing* can sense multiple phenomena. For example, a weather station can sense temperature, humidity, wind direction, wind speed, rain precipitation, and Particulate Matters 2.5mm (PM2.5). For this example, *Thing* is the weather station and *Location* is the position of the weather station. As the *Thing* is most probably static, we don't need to worry about moving and tracking the *HistoricalLocations*.

Datastream groups *Observations* of the same type for a *Thing*. *Observations* of a *Datastream* are recorded by the same *Sensor* and are all observing the same phenomenon. In the example of the weather station, we have six *Datastreams* for grouping each type of readings, temperature, humidity, wind direction, wind speed, rain precipitation, and PM2.5. Each *Datastream*'s *Observations* have the same type, observe the same phenomenon, and are read by one *Sensor*. Aside from name and description, a *Datastream* has observationType, and unitOfmeasurement. ObservationType specifies the type of *Observations* grouped by that *Datastream*. For example, it can be numeric, true/false, categories, etc. In order to define these observationTypes in a standard way, SensorThings uses O&M (Cox, 2011) conceptual model. It has defined category, count, measurement, truth, and complex observationTypes.

The UnitOfMeasurement specifies the unit for *Datastream*'s *Observations*. This property is mostly useful for numeric/measurement *Observations*. It is a JSON object with three fields: name, symbol and definition. The definition is a URI for defining the meaning of the measurement. The definition is the means of unification between different unitOfMeasurement. For example, people may use C or degC or degreeC or °C for the Celsius' symbol or something else in another language. However, using the same definition URI shows that all these unitOfMeasurements are the same.

There are three optional properties for *Datastream*: observedArea, phenomenonTime, and resultTime. ObservedArea specifies the area that is observed by all the *Observations* of that *Datastream*. PhenomenonTime is a time period and it specifies the time range this *Datastream* has the *Observations* for. ResultTime is like phenomenonTime except that it shows the range of resultTimes for *Observation*.

Each *Datastream* has a *Sensor* and an *ObservedProperty*. A *Sensor* entity keeps information about the sensor device that is recording the *Observations*. Other than description and name, it has a metadata and encodingType. EncodingType defines the type of metadata. For example, it can be SensorML (Botts, Robin, Greenwood, & Wesloh, 2014), PDF, or the html site. Metadata contains detailed information about the Sensor. It is recommended as the Sensor data sheet or SensorML.

Each *Sensor* can be connected to more than one *Datastream*. SensorThings is flexible about how to use the *Sensor* entity. Each *Datastream* can have its own unique *Sensor* entity and detail

information of the sensor such as, the serial number or Media Access Control (MAC) address, can be persisted in *Sensor* entities. Another way is reusing *Sensor* entities when the same sensor type is used in different *Datastream*s. In this case, it is possible to see all the *Datastream*s that are recorded by the same *Sensor*. For example, SH13[1] is a temperature-humidity sensor. You can check all the *Datastream*s and *Thing*s that are using this *Sensor* by checking *Datastream*s that are connected to that *Sensor* with *Sensors*({id})/*Datastreams*. As a result, depending on the use case, *Sensor* can be modeled and used differently. However, the second approach may be more useful from an application point of view. Same as with *Location*, even if *Sensor* entities are not reused, there might be other ways to get all the *Datastream*s of the same *Sensor* type with the rich query options of SensorThings.

As mentioned before, each *Datastream* has one *ObservedProperty*. *ObservedProperty* is the phenomenon observed by *Datastream*'s *Observation*s. Examples of this phenomena are weather temperature, or dust particulates in the air. Other than name and description, *ObservedProperty* has definition property. Similar to *Location* and *Sensor* entities, although SensorThings is flexible and does not force the reusing of entities, it is best practice to reuse *ObservedProperty* entities in case *Datastream*s are observing the same phenomenon. However, in cases such as when using different languages, definition property is the joint point between

[1] http://wiki.seeedstudio.com/Grove-TempAndHumi_Sensor-SHT31/

ObservedProperties. Definition is a URI containing the definition of the phenomenon and it is used for differentiation purposes between same phenomenon that use different name and description, and can be in different languages. With the reuse approach, we can access all of the *Datastream*s observing the same phenomenon with *ObservedProperties*({id})/*Datastreams*. For the second approach, with the SensorThings rich query option, we can retrieve *ObservedProperties* that have the same definition and then get all of their *Datastreams*.

As mentioned before, each *Datastream* has many *Observation*s. *Observation* is the entity that records sensor readings. The type of reading, the sensor that takes the reading, and the phenomenon that the reading is observing can be found from the *Datastream* attached to the *Observation*, and the *Sensor* and *ObservedProperty* that are attached to the *Datastream*. *Observation* does not have name and description. It has result, phenomenonTime, and resultTime. Result records the reading from the sensor. PhenomenonTime is the time when the Observation happened. ResultTime is the time that the result is recorded in the system. Result and phenomenonTime cannot be null as they are the core information required for an *Observation*. Result's type should match the observationType specified in the *Datastream*.

ResultTime is a time instance and it specifies the time the result is recorded. However, phenomenonTime is a time object. It can be time instance or time interval. Having the flexibility of recording phenomenonTime as instance or interval can be useful in different use cases. Using time instance for phenomenonTime is commonly used for *Observation*. For the example of the weather station, keeping time instance can be efficient. However, for video *Observations*, time

instance is not sufficient. Video *Observation* is the use case for which time interval is the perfect match. In that case, phenomenonTime has the starting time of when that video is taken and also the time that it finishes.

The other use case for which the time interval can be useful for phenomenonTime is when certain granularity for phenomenonTime is not important. For example, consider a case when the seconds that the *Observation* happens is not important. In that case, all the *Observation*s happening within a certain minute of an hour can have a time interval with the start and end of that minute.

It also can be very useful in the case of sampling and aggregation. Raw *Observation*s can be taken and then aggregated for every minute and persisted in another aggregated *Datastream*. In this case, *Observation*s of such a *Datastream* has the aggregated value as their result and the phenomenonTime is the time range for which the *Observation*s are aggregated.

Each *Observation*, in addition to having a linked *Datastream*, has a *FeatureOfInterest*. *FeatureOfInterest* is the feature of the phenomenon observed by the *Observation*. The *FeatureOfInterest* entity has encodingType and feature, as well as name and description. EncodingType specifies which standard is used to represent the feature. Feature is the detailed description of *FeatureOfInterest* with the encoding specified in encodingType. As with *Location*, we recommend using the well-known geospatial standard, GeoJSON.

In a lot of situations, the *FeatureOfInterest* is identical to the *Thing*'s *Location*. For example, for a thermostat that is in the living room, the *Location* and *FeatureOfInterest* are both the living

room's position/area. However, in the case of remote sensing, *Location* and *FeatureOfInterest* are different. A simple example is a camera or satellite. Their position is different from the area they are observing. For the camera, this area is close to the camera's position, whereas it is quite far for the satellite.

Since most of the use cases have the same *Location* and *FeatureOfInterest*, SensorThings has the automatic option for creating the *FeatureOfInterest* from the *Thing*'s *Location*, if the *FeatureOfInterest* is not specified during the creation of the *Observation*. In other words, if you create an *Observation* and you don't link the *Observation* to a *FeatureOfInterest* in the request, SensorThings will create or reuse a *FeatureOfInterest* based on the information from the *Thing*'s *Location*. Note that since the relationship between *Thing* and *Location* is not mandatory, the request of creating *Observation* without linking the *FeatureOfInterest* will fail, if a *Thing* does not have a linked *Location*. Every time a SensorThings server needs to automatically link the new *Observation* to a *FeatureOfInterest*, it first checks if the *Thing* has a *Location*. If not, the request fails. Then, the service checks if there is an existing *FeatureOfInterest* with that *Location* information. If it exists, the service links the new *Observation* to the existing *FeatureOfInterest*. Otherwise, the service will create a new *FeatureOfInterest* and link it to the new *Observation*.

SensorThings forces the reusing of FeaturesOfInterest when it is created and linked automatically by the service. However, to make it simpler for the clients, it is not mandatory to reuse *FeatureOfInterest* when clients are creating and linking *FeatureOfInterest* themselves. Best practice for the service developer is to catch these cases and handles reusing the

FeatureOfInterest in the background so that it does not affect client experience. It is also always best practice for clients to try to reuse *FeatureOfInterest* whenever possible and more importantly to let the server handle *FeatureOfInterest* in case it is the same as the *Thing*'s *Location*.

When *FeaturesOfInterest* are reused throughout the system, different *Observations* for a specific *FeatureOfInterest* can be accessed with *FeaturesOfInterest*({id})/*Observations*. With SensorThings query options, it can be further filtered to access the readings for a specific time interval. Same as with other entities, even if *FeaturesOfInterest* are not reused, those queries are possible with SensorThings rich geospatial query options. However, it may put more load on the service, which makes it important for the service developers to follow the best practices.

SensorThings Application Interface

The previous section has explained the how-tos of modeling IoT systems with the SensorThings API. In this section, we discuss the SensorThings API application interface by exploring the requests and protocol for interacting with the API. We start with data retrieval before moving onto how to post data to SensorThings and we link this section with the device interface.

Data Retrieval

As mentioned before, SensorThings is customizing and extending OData for IoT. For data retrieval, SensorThings has a REST-like interface as well as a set of query options and functions for filtering data. To access all the data, i.e. historical data as well as recent data, HTTP protocol

and GET request are used. The following explains the URL convention and query options that can be used in GET requests.

SensorThings is following OData URL conventions for the requests. The only difference between this convention and regular RESTful is using @iot.id in parenthesis in order to refer to a specific entity. Related entities can be accessed following the OData convention as /Entities({id})/RelatedEntity(ies).

SensorThings also makes access to the entities easier by providing @iot.selfLink and @iot.navigationLink. Accessing the root URL of SensorThings returns the link to access each entity. For each entity, the @iot.selfLink provides the URL to access that specific entity. Moreover, each entity @iot.navigationLink provides a link to its related entities. Knowing the request URL, entities can be accessed by sending HTTP GET request to the SensorThings service.

Furthermore, SensorThings API provides a wide variety of query options for accessing the data. Using these query options, the service response can be customized in terms of number of returned results, content, and order.

$expand and $select query options are used for controlling the level of detail in the SensorThings service response. With $expand, information from related entities can be embedded in the response for a request to read the entity(ies). $select limits the properties that will be returned for each entity. For example, with $select it can be specified for the query to

return only result property for *Observation*s in the read request. As a result, the response will not contain other properties such as phenomenonTime or resultTime.

The $orderby query option is used for sorting the SensorThings response based on one or more properties. The $top query option can limit the number of entities that will be returned in the SensorThings response. $skip is also used to skip a specific number of entities before returning the result from the SensorThings service. $top, $skip, and $orderby are meant to be used together for pagination. Using a specific order, $top and $skip can be used to return entities in pages. Pagination is a use case of these query options. However, they can be used individually and for other purposes as well. For example, getting the latest *Observation* is one of the use cases which can be achieved by ordering the response with phenomenonTime and then asking for the top one.

With the $count query option set to true, the service returns the total number of entities found for the request. If pagination is enabled on the service, the response may only contain a limited number of entities but with $count=true, the service also returns the total number which could be useful for clients.

One of the most useful query options in SensorThings is $filter. With $filter, clients can filter the results from the SensorThings service in a very flexible manner. SensorThings supports arithmetic operators as well as logical operators. There is also a wide variety of functions that can be used for filtering the results. Moreover, spatial functions are one of the things which differentiate SensorThings from the other IoT platforms and proposed standards. For example,

finding the closest sensor to a specific location or another sensor; finding all the sensors in a bounding box; and finding sensors that are observing the same area are easy using $filter in SensorThings.

The query options can also be used to query expanded entities. Clients can filter the expanded entities in the request by specifying the filters in parenthesis in front of the $expand query. This capability makes SensorThings more flexible and easy to use for clients.

SensorThings uses @iot.nextLink as a means of pagination. Clients can manually apply pagination for the service by using $top and $skip or using the pagination that is provided by the service. SensorThings developers are all encouraged to implement pagination by limiting the number of results returned and providing the @iot.nextLink to access the next page of results. Providing pagination boosts the performance of the SensorThings service for most use cases and is strongly recommended for SensorThings service developers.

MQTT

The SensorThings MQTT extension provides access to real-time data using the MQTT protocol. This is another way of accessing SensorThings data in real-time in addition to the HTTP protocol. One of the most important use cases for MQTT in SensorThings is receiving Observations in real-time. The topic that is used for subscribing to MQTT is a collection or navigation collection URL pattern without the service address, i.e. the path starts from v1.0/. After subscribing to a topic, each time an insertion/modification happens to that collection, the

client will receive a MQTT notification with the JSON representation of the added/modified entity.

DataArray

*Observation*s can be retrieved in SWE common DataArray format as well. This feature is added to SensorThings to make it interoperable with SWE common compatible services as well as for efficient retrieval of *Observation*s as it removes redundant information in the response. $format=DataArray can be used to retrieve Observations in DataArray format. In this case, *Observation*s will be grouped by their *Datastream* and response will be formatted to DataArray format.

Data Insertion and Modification

HTTP requests for the insertion and modification of data for SensorThings. POST request is used for creating entities; PATCH for updating an entity; and DELETE for deleting an entity. Also whilst PUT can be used for updating or resetting an entity, a SensorThings service may or may not implement this functionality.

In order to create an entity, the POST request should be sent to the entity collection URL or relative navigation collection URL with a valid entity JSON as the body. For updates, the PATCH request should be sent to the @iot.selfLink of the specific entity. The PATCH request body contains a JSON with the properties of the entity that are modified. Sending a DELETE request to @iot.selfLink results in deleting that specific entity from the collection. If the server also supports PUT for updates, the entity will be replaced completely with the JSON entity in the

PUT body. Thus, the JSON entity in the PUT request body must have all the mandatory fields even if the values are unchanged.

For insertion and deletion, some integrity constraints apply. Insertion integrity constraints are applied to *Datastream* and *Observation*. For creating a *Datastream*, it must be linked to a *Thing*, a *Sensor*, and an *ObservedProperty*. Similarly, in order to create an *Observation*, it must be linked to a *Datastream* and a *FeatureOfInterest*. There is an exception for this constraint for creating an *Observation*. If the *FeatureOfInterest* is the same as the *Thing*'s *Location*, there is no need for the POST request to contain the link from *Observation* to *FeatureOfInterest*. In this case, the SensorThings service must automatically create or reuse a *FeatureOfInterest* with the corresponding *Thing*'s *Location* information. Note that the link between *Observation* and *FeatureOfInterest* is mandatory and this exception only makes the client request easier.

There are also integrity constraints that are applied on most of the entities when an entity is deleted. When a *Thing* is deleted, all of its corresponding *Datastream*s and *HistoricalLocation*s will be deleted automatically by the SensorThings service. Also, deleting a *Datastream* or a *FeatureOfInterest* results in the deletion of all the linked *Observation*s. By deleting *Sensors* or *ObservedProperties*, all of the corresponding Datastreams will be deleted by the SensorThings service. Finally, when a *Location* is deleted, the SensorThings service will delete the linked *HistoricalLocation*s automatically.

SensorThings API provides the capability for embedding entities inside the creation request. In other words, an entity can be created together with its related entities in one request. This

capability is called deep insert. In order to deep insert, the POST request body should contain the valid JSON for each of the related entities that are supposed to be created inline. The POST request body can have some inline entities and some links to existing entities. The JSON body looks like the GET response when the related entity is expanded using $expand.

MQTT

In addition to HTTP POST, the MQTT protocol can be used for creating *Observation*s. To this end, the valid JSON for creating an *Observation* should be published to the *Observation*s topics, i.e. the topics used for subscription to *Observation*s. These topics include collection and navigation URL paths to *Observation*s without the service address. Insertion integrity constraints for *Observation* apply for MQTT creation as well.

DataArray

Similar to retrieving *Observations* in DataArray format, *Observation*s can be created using DataArray. To this end, the JSON request body contains multiple *Observation*s grouped by their *Datastream* and in DataArray format. The POST request should be sent to a special URL path, /v1.0/CreateObservations. This capability provides an efficient way of creating multiple *Observation*s at the same time from the sensor devices.

Batch Requests Extension

SensorThings API supports batch processing in order to provide an efficient way for resource-constrained IoT devices to send multiple requests in one communication to the server. Multiple CRUD requests can be sent to the SensorThings server as one HTTP request using

batch request. SensorThings API is following OData by supporting batch request with the exception that the OData header should be removed from the request. Just as with OData, SensorThings is using Multipart MIME v1.0 message as a standard for representing batch request and response. Batch request is considered an extension for SensorThings. Thus, server support for batch request is optional. However, it is highly recommended as it is useful for effective communication between SensorThings with resource-constrained IoT devices.

MultiDatastream Extension

SensorThings API provides *MultiDatastream* extension to support complex *Observations*. For most of the use cases, *Observation*s have simple types such as number, categories, etc. However, there are use cases in which *Observation*s have a complex type or there is an array of *Observation*s. One example of this situation is when more than one parameter is important for the *Observation*. For example, when a sensor reading depends on the temperature of the unit and the sensor reading should be interpreted differently with respect to the temperature unit. In this case, the SensorThings *MultiDatastream* extension can be used. A *MultiDatastream* observes more than one phenomenon and its *Observation*s are in forms of array. For the previous example, each *Observation* of the *MultiDatastream* is an array with two elements, the first is the sensor reading and the second is the temperature of the unit.

MultiDatastream's properties are slightly different from *Datastream*. First of all, the observationType is always O&M ComplexObservation as all other observationTypes can be modeled using *Datastream*. In order to specify the observationType for each element of the

Observation array, *MultiDatastream* has a property called multiObservationDataType which is an array of O&M observationTypes. Moreover, unitOfMeasurement is a JSON array as it needs to define the unit of measurement for each of the elements in the *Observation* array. All of the other properties are the same as *Datastream*.

As mentioned above, a *MultiDatastream* can observe multiple phenomena. As a result, each *MultiDatastream* can be related to more than one *ObservedProperty*. In other words, a *MultiDatastream* has an *ObservedProperty* for each element of its *Observation* array. The rest of the relations are the same as *Datastream* and *MultiDatastream* has one *Thing* and one *Sensor* and multiple *Observation*s.

MultiDatastream is an extension to SensorThings and supporting that is optional. Even in the case of complex *Observation*s, the system can be modeled without the *MultiDatastream* extension. With multiple *Datastream*s, the system can correlate their *Observation*s at accessing time with queries. However, using *MultiDatastream* makes it easier to use and easier to understand for clients. In the case of the server needing to support complex *Observation*s, this extension provides developers with guidance on how to add the capability to the server. Support for that is highly recommended.

Summary

SensorThings API is an open geospatial standard for IoT that is approved by OGC. SensorThings defines the data model and the retrieval API for managing IoT data. The SensorThings data model is comprehensive and can address most of the IoT use cases. There is a

Thing which is the sensor system. A *Thing* can have *Locations* as its current location and *HistoricalLocation*s as the previous locations. A *Thing* has multiple *Datastream*s. A *Datastream* groups *Observation*s observing the same phenomenon, called *ObservedProperty*, and are generated by the same *Sensor*. Finally, *Observation*s are where sensor readings are stored and *FeatureOfInterest* records the feature observed by each *Observation*.

SensorThings API adopted OData and a REST-like interface for interacting with IoT data. It also has a MQTT extension, a solution for resource-constrained IoT devices. Moreover, the API provides *MultiDatastream* and Batch Requests extensions for more complicated use cases in IoT.

SensorThings API is a solution for homogenizing the heterogeneous IoT data and provides interoperability between different IoT systems. The sensing part was published in July 2016 and in the two years since then, it has been mentioned in different literature, research, and industry projects. Furthermore, there are multiple implementations of the API available, as well as, some in progress which prevent the vendor locked in problem. The evidence shows the maturity of the standard and make it a good fit for addressing the data variety challenge for IoT.

Future Work

The SensorThings API has a second part called the Tasking part. The Tasking part defines the data model and interface for controlling devices or so-called actuators. The SensorThings Tasking part is going to be published in late 2018 as an OGC standard. The Sensing part together with the Tasking part make SensorThings a comprehensive standard for all aspects of IoT. It fills

the gap between sensing and controlling devices and provides a standard API for IoT devices to talk to each other.

Furthermore, there is a work in progress for defining events for the SensorThings API called the Rules engine part. This part connects SensorThings Sensing and Tasking parts together with the definition of an event. For example, the event defines a situation for sensor readings in the Sensing part and if the event occurs different tasks can be generated in the Tasking part.

SensorThings can also be merged with JSON-LD to become linked data ready. Providing linked data helps machines to understand SensorThings better and SensorThings can use that to be linked to other system sources. Also, other systems can digest SensorThings resources smoothly. JSON-LD provides a standard lightweight linked data format. SensorThings can easily be integrated with JSON-LD. For that end, @context needs to be defined for each of the SensorThings entities and their properties. Since SensorThings is also using JSON format and the entities and their properties are well defined, @context can be defined for them easily and the integration will be smooth. Providing comprehensive JSON-LD context is the future work for the SensorThings API.

Data Management for Internet of Things and Lambda Architecture

The main objective of this dissertation is to propose an architecture for IoT that overcomes big data management and geospatial challenges. In this chapter, we explore the big data challenge in IoT as well as Lambda architecture as a potential solution for the big data challenge in general. Then, we discuss the proposed architecture and how it meets the big data compatibility requirements in the next chapter.

Big Data and Internet of Things

One of the most important areas of future information technology is big data and its emerging technological development. In 2011, the big data concept and its potential was introduced in an EMC/IDC research report (Gantz & Reinsel, 2011). It was stated in the 2012 World Economic Forum in Davos, Switzerland, that big data has become a strategic economic resource and has high value that is similar to currency and gold in terms of significance and liquidity (Alharthi, Krotov, & Bowman, 2017; Johnson, 2012). In recent years, many enterprise organizations have started investing in different solutions to cope with big data challenges (IDG, 2015). They realized that an important source of competitive advantage in future is big data analytics (Alharthi et al., 2017).

The total amount of generated data has increased nine times in a five year period, according to the International Data Corporation (IDC) report (Gantz & Reinsel, 2011). It is also predicted that this amount will double every two years (Chen, Mao, Zhang, & Leung, 2014; Yaqoob et al., 2016) . As one of the buzzwords for the past decade, big data refers to large data sets that need

technologies beyond traditional data management tools to manage and process (Akoka, Comyn-Wattiau, & Laoufi, 2017).

IoT and social media are considered the most important drivers for rapid progress on big data technologies and applications (Lee, 2017). IoT devices such as wearables, environmental sensors and smart home appliances are generating a large portion of worldwide data (Lee, 2017). For IoT a great many sensors are embedded into different devices, collecting various types of data such as environmental data, geographical data, astronomical data, and logistic data. As Cisco reported, the number of internet-connected devices has already exceeded the world population (Evans, 2011). The number of internet-connected devices is expected to double in size from 22.9 billion in 2016 to 50 billion by 2020 (Figure 5) (Ahmed et al., 2017). These internet-connected devices include, but are not limited to, Wi-Fi enabled sensors, smart home appliances, and wearable technologies. As IoT is formed by these devices (Figure 6), it shows how IoT is responsible for the data deluge (Ahmed et al., 2017). Currently, IoT data is not the dominant part of big data. However, by 2030, as predicted by HP, there will be a trillion sensors generating data and IoT can be the most important part of big data (Chen et al., 2014).

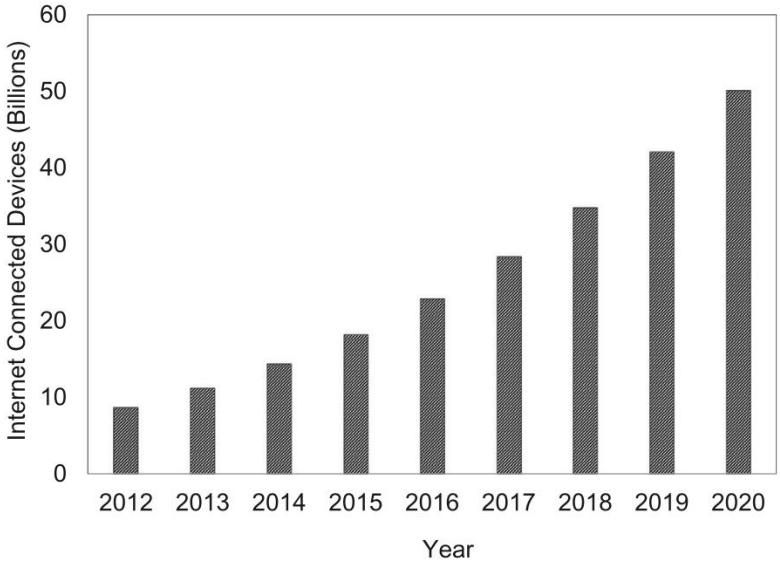

Figure 5 Trend in the Number of Internet-Connected Devices (Adapted from (Ahmed et al., 2017)**)**

Figure 6 Internet-Connected Devices Forming IoT (Adapted from (Ahmed et al., 2017)**)**

Akoka et al in (Akoka et al., 2017) surveyed the literature on big data and did a statistical study on publication trends and subject, application and focus of the papers. The publication trend showed that there is an exponential growth in number of research papers in the area of big data. They found out that Cloud and Analytics were the topics for most of the published papers, whilst in spite of the fact that IoT is one of the main markets for big data application, IoT is one of the least addressed topics in the papers. They expect that IoT will become one of the hottest areas in big data research with an increase in the number of research publications very soon. (Akoka et al., 2017)

IoT and big data research are two technologies that are interdependent. With the growth of IoT applications, more and more data will be generated both in terms of quantity and heterogeneity. As a result, it provides opportunity for the application and development of big data. On the other hand, the research advances and business models of IoT can be accelerated by the application of big data technology.

As we saw, data management in IoT is a hot topic and needs improvement to cope with the data deluge that has already started. In this thesis, we propose an architecture for data management for IoT that addresses big data management challenges.

Since 2001, big data management challenges have been defined using the 3Vs model, Volume, Variety, and Velocity. The 3Vs model was first introduced in a META report (Laney, 2001) and it has been used in different literature and by different organizations such as IBM and

Microsoft to define big data and its challenges (Yaqoob et al., 2016). Through the years more Vs were added to this model such as, Value (Mishra et al., 2015), Veracity (Normandeau, 2013; Ward & Barker, 2013), Validity (Normandeau, 2013), and Volatility (Klarity, 2015). However, since the 3Vs are the main challenges, in the latest definition by Gartner (Gartner, 2018), big data is still defined with the three characteristics of high-volume, high-velocity, and/or high-variety. Thus, in this dissertation, our focus will be on variety, volume, and velocity, as main challenges we face in big data.

In the 3Vs model, volume refers to the massive amount of data that is difficult to collect, manage, and analyze with current infrastructures and tools. Variety refers to heterogeneous data that is generated by different sources with different structures or may even be unstructured. And finally, velocity refers to real-time data streams that are continuously generated with high frequency. The big data generated in IoT has the same characteristics: heterogeneity, large size, and high rate of data streams.

In the next section, we review the literature and also off-the-shelf solutions for big data management for IoT. Then, we explore Lambda architecture as a potential solution for addressing big data volume and velocity challenges. We explain our proposed architecture in the next chapter and elaborate on how it can address all 3Vs challenges of big data.

Related Work

We categorize the related works into two categories: the research and literature, and the platforms and technologies available for big data management.

Related Literature

In this section, literature about IoT and its data management is reviewed. IoT is a wide research area and also an emerging technology. Different surveys about IoT research challenges, prospective applications, and architectural elements have been conducted in recent years (Al-Fuqaha, Guizani, Mohammadi, Aledhari, & Ayyash, 2015; Atzori, Iera, & Morabito, 2010; Gubbi, Buyya, Marusic, & Palaniswami, 2013; S. Li, Xu, & Zhao, 2014; Miorandi, Sicari, De Pellegrini, & Chlamtac, 2012; Zeng, Guo, & Cheng, 2011). These surveys are mostly focused on high level issues and architecture of IoT.

Atzori et al. (Atzori et al., 2010) presented different visions for IoT (things-oriented, internet-oriented, and semantic-oriented). They also surveyed enabling technologies, open issues, and some applications for IoT. Zeng et al. (Zeng et al., 2011) focused on web-oriented architecture for IoT and its open issues in their survey. Miorandi et al. (Miorandi et al., 2012) categorized and explained research areas and ongoing initiatives for IoT.

Gubbia et al. (Gubbi et al., 2013) surveyed IoT with cloud-centric vision. They proposed the use of a specific framework such as Aneka instead of using Cloud storage and MapReduce. Aneka is a platform as a service (PaaS) that can be integrated with Microsoft Azure or Amazon EC2. The proposed solution in this dissertation is the architecture and it is one level of abstraction higher than the proposed Cloud system in (Gubbi et al., 2013), since they used technologies and platforms in their proposed solution. I propose the architecture that allows the user to choose the technology to fulfil that.

Li et al. (S. Li et al., 2014) concentrated on service-oriented architecture (SOA) for IoT as well as open issues in IoT. The SOA they proposed for IoT contains four layers: sensing layer, network layer, service layer, and interface layer. Since they follow SOA, each of these layers is independent and can be exchanged with other services with the same functionalities. I propose an architecture for the service layer in the overall SOA for IoT with the main objective of addressing data management issues.

Al-Fuqaha et al. (Al-Fuqaha et al., 2015) surveyed IoT for its enabling technologies, protocols, and applications. There are two interesting observations in their survey which is related to our work. Firstly, whilst they surveyed application protocols, it shows that REST and MQTT are two of the most used protocols. They are what is used by the SensorThings API as well. Secondly, scalability and interoperability are named as two of the main challenges for IoT which is the focus of this thesis. They concluded that there is a need for platforms that support IoT big data and analytics. They also noted that security is another hot topic for IoT. In this dissertation, the focus is on presenting an architecture for supporting big data analytics for IoT and security is out of the scope of this thesis.

Amongst these surveys there is one with a data-centric view (Qin et al., 2016). In this survey, data management challenges are present in terms of managing data streams, heterogeneous data streams, and large volumes of data. These challenges remain the same as this dissertation challenges the data velocity, variety, volume respectively.

Whitmore et al. did a comprehensive survey on the literature for IoT in 2015 (Whitmore, Agarwal, & Da Xu, 2015). They did a classification of the literature related to IoT and found out that most of the IoT literature has focused on the technology part of IoT and that hardware was usually the topic of interest. It also stated that big data and how IoT would fit into the big data movement is one of the important future steps for IoT.

Arasteh et al. (Arasteh et al., 2016) surveyed components and features of IoT-based smart cities. Heterogeneity, large scale, and big data are listed amongst the challenges of IoT-based smart city applications. Smart city is a spreading use case for IoT and it suffers from the same challenges as IoT.

Ray (Ray, 2017) did a survey on IoT cloud platforms. He compared 26 different platforms and visualization tools from different aspects. It shows the silos for IoT by showing the options that are available out there. He found that heterogeneity management, analytics, visualization, and research-centric clouds were missing on the surveyed platforms, although data management was the goal for most of these platforms. One important finding from this literature is that the platforms that are evaluated either do not have big data capability, or their capability is limited to using NoSQL databases. We can see that they are relying on tools rather than architecture for big data and the only tool used is the NoSQL databases. As concluded in this paper also, big data analytics is the lacking capability on these platforms and needs to be studied. In this thesis, we focus on architecture for big data rather than tools. When the architecture is set, it can be implemented with different tools.

There are several attempts to develop (web-based) interfaces for IoT in order to overcome the heterogeneity challenge (Guinard, Trifa, Mattern, & Wilde, 2011; Leong & Choo, 2014; LogMeIn Inc., 2015). To this end, REST-based approaches and the use of the JSON format are adopted by some of these literature. The reason for the popularity of REST-based architecture is that HTTP transport protocol is widely available and easy to use. Moreover, JSON encoding is adopted since compared to its alternatives such as CSV and XML, it is more lightweight as well as more expressive. Unlike SensorThings API, (Guinard et al., 2011; Leong & Choo, 2014; LogMeIn Inc., 2015) focused on developing a platform rather than providing an API that everyone can customize according to their needs. Furthermore, SensorThings API provides an international and open standard solution for overcoming heterogeneity issues unlike (Guinard et al., 2011; Leong & Choo, 2014; LogMeIn Inc., 2015) as they only try to provide a platform solution that can accept heterogeneous data.

Some literature on IoT architecture focuses on the big data challenge (S. Li et al., 2014; Mishra et al., 2015; Sowe, Kimata, Dong, & Zettsu, 2014; Tracey & Sreenan, 2013). The literature proposed the overall architecture of IoT with regards to the huge number of devices that are connected to IoT and the large volume of data that is created. Tracey et al. in (Tracey & Sreenan, 2013) proposed a holistic architecture for IoT. In their architecture, they only mentioned the large volume of data out of three big data challenges and for that they proposed using HBase for storing the data. Mishra et al. in (Mishra et al., 2015) proposed a cognitive oriented IoT big data framework (COIB framework). In their IoT overall architecture, they used

HBase for large volumes of data. They also add a layer to their architecture for grouping homogenous data out of heterogeneous IoT data using classification. However, real-time data management is not considered in their architecture. Sowe et al. in (Sowe et al., 2014) focused on the heterogeneity challenge in IoT data. They use a Service-Controlled Networking (SCN) middleware in their architecture to manage the heterogonous data. In this middleware, they used different APIs for different types of data that goes to the system with different protocol. However, this approach may not scale well, since the number of these APIs can grow exponentially in time if there is no standard API. Li et al. in (S. Li et al., 2014) proposed using SOA for IoT as mentioned before. This architecture is for the overall IoT and my proposed architecture is for the service layer out of this overall architecture. To summarize, all this work focused on the overall architecture and the data volume and variety challenge. However, they did not elaborate on the data management in detail, since their concentration is on the overall architecture.

There is some work on data storage and indexing for IoT (T. Li, Liu, Tian, Shen, & Mao, 2012; Y. Ma et al., 2012). Ma et al. in (Y. Ma et al., 2012) proposed an update and query efficient index (UQE-Index) framework for IoT. They proposed having two different clusters, one for storage and one for indexing. The data is also classified as current data and historical data. They used B+-tree indexing for time index and R-tree for space index. Their idea can be used for indexing data in the serving layer in our proposed solution.

Li et al. in (T. Li et al., 2012) proposed a storage design for IoT massive data based on NoSQL named IOTMDB. In their proposed design, data storage and data management are separated and the responsibilities are assigned to different machines in a cluster. Master nodes are responsible for data management and slave nodes are only for storing the data. There are also standby nodes working as a secondary option if any damage occurs to the primary nodes. They also proposed performing preprocessing on data and retrieving the *interest* and *digest value* in order to make future queries more efficient. For the data management part, they only focused on historical, tracking and preference queries. They claimed that they chose those queries based on the demand of IoT applications. However, it limits their system. Compared to our proposed architecture, this literature can be used as the master data set for the batch layer. It provides a way for storing and managing the massive data for IoT, but their solution lacks a method for big data analytics in an efficient manner, as also mentioned in their future work. Moreover, no mechanism is proposed for managing IoT real-time data.

Lambda architecture, proposed by Marz (Marz & Warren, 2015), is seen to be the future of big data analytics because of its ability to deal with both historical and real-time data (W. Fan & Bifet, 2013; H. H. Huang & Liu, 2014). Batch processing and stream processing are two categories of big data analytics (H. H. Huang & Liu, 2014). The strength of Lambda architecture is that it combines batch and stream processing together and constructs a general framework for analysing big data.

Villari et al. (Villari, Celesti, Fazio, & Puliafito, 2015) proposed AllJoyn Lambda. They claimed that AllJoyn Lambda is a software solution integrating AllJoyn in the Lambda architecture used for Big Data storage and analytics in IoT. Using Lambda architecture is what makes our work similar to this work. However, there are multiple differentiators between our work. AllJoyn is a software that is used for IoT from the physical to application layer. It claims that it is a solution for the interoperability challenge. However, the interoperability in their case is in sensor discovery and operating systems and does not discuss data management. Also, AllJoyn does not cover the geospatial use cases in IoT. The Lambda architecture is a well-established solution for addressing the big data challenge. But this work is totally different from what we are proposing, as our work is an architecture rather than a software. Also, our work focuses on the service layer and data management whereas their paper vaguely claimed to have a software for the whole IoT stack. Furthermore, we use an open geospatial standard as a solid solution for the IoT interoperability challenge.

Huacarpuma et al in (Cruz Huacarpuma et al., 2017) proposed a Distributed Data Service (DDS) to collect and process data for IoT environments with the goal of enabling multiple IoT middleware systems to share common data services. They used a distributed architecture for storing and processing the data. Their architecture has two parts, data collection and data aggregation. In order to gather the data from different sources and overcome the heterogeneity issue, they used a communications interface that converts different data formats and also a metadata creation module that manages different metadata characteristics. The data collection

part gets data from different IoT middleware and generate metadata and store it in the NoSQL database. They used JSON format for receiving data from IoT middleware. The data aggregation part aggregates and summarizes the data. They claimed that their work was similar to the Lambda architecture batch and serving layer but did not provide details. For example, there is no immutability in their data set. There are similarities between this work and ours. Both try to solve the big data volume, velocity, and variety challenge. Other similarities include using batch processing and real-time processing; using a publish-subscribe paradigm for collecting the data; and using the JSON format. However, one important differentiator for our work is using the SensorThings API. They used some generated metadata for solving the heterogeneity issue. The most important metadata that they mentioned are timestamp and location. We can see in the example they provided, that they used latitude and longitude for storing location information which makes the system limited to points and cannot store other geometries such as polygons. The difference here is the rich and mature data model of SensorThings API compared to their metadata. Also, our proposed architecture uses the Lambda architecture and its best practices for addressing big data volume and velocity simply and seamlessly.

Big data analytics is considered an important next step for IoT health care systems (Firouzi et al., 2018). Manogaran et al in (Manogaran et al., 2018) proposed IoT architecture for smart healthcare monitoring and alert systems to address big data and security challenges. Their proposed architecture has two subsystems, Meta Fog-Redirection (MF-R) and Grouping and Choosing (GC) architecture. MF-R has three phases: data collection, transfer and storage. In the

data collection phase, data is gathered from medical devices. Through fog computing, alerts are generated if the sensor readings are out of their normal healthy range. The data transfer phase moves the data from local databases to Amazon Elastic MapReduce to enable big data MapReduce processing. The storage phase stores the data in Hbase. Security is placed between fog and cloud layer in GC. The nature of this work is for health monitoring and alerting systems. As a result, it makes real-time processing much more important and they add fog computing for what is near the edge. For data management, they do not provide any specific architecture and they merely mentioned different Amazon technologies used such as Amazon EMR and S3. Our work focuses on data management. Fog computing can be added in another layer for any alert needed.

Marjani et al (Marjani et al., 2017) did a survey for big data analytics in IoT. This work surveyed big data, IoT, and big data analytics separately and tried to show the relationship between big data analytics and IoT. The paper proposed that in the IoT architecture layer, big data analytics should be a layer on top of cloud storage and should be implemented using Hadoop technologies. What they proposed as architecture is very high level and without detail as the paper's focus is on the survey part. Our work is a detailed architecture that combines their cloud storage and big data analytics layers. Our SensorThings master dataset can be the cloud storage and the rest of batch layer and serving and speed layers will be in the big data analytics layer.

Darwish and Abu Bakar in (Darwish & Abu Bakar, 2018) explored big data analytics for the Internet of Vehicles (IoV). IoV could be considered a category in IoT. They proposed an architecture which uses Lambda architecture for processing or, as they called it, artificial intelligence layer. Their proposed architecture is for the full stack of IoV including the physical layer and application layer. They proposed using fog computing in the edge layer for transportation sensor devices. This work shows how fog computing can be used alongside Lambda architecture in another layer for application specific use cases. They did not add any details about the possible data model and listed data heterogeneity as one of the challenges needing to be fixed – which was something that we explored in our work.

To the best of our knowledge, the proposed research in this dissertation is the first attempt to integrate open standards and Lambda architecture in the context of IoT to address the big data management challenges.

Big Data Platforms and Technologies

In this section, we explore current technologies and off-the-shelf solutions for big data management for IoT.

Hadoop

Apache Hadoop[1] is one of the most well-known big data technologies now. It is used by many major software organizations including, but not limited to, Amazon, Facebook, Google, IBM, and Yahoo! as well as numerous other companies. Apache Hadoop consists of open-source software for reliable, scalable, distributed computing. The goal of Hadoop is to build a framework for distributed processing of large data sets across clusters of computers. Its main focus is on software rather than hardware, for the availability and seamless handling of failures. Hadoop Common, Hadoop Distributed File System (HDFS), Yarn, and MapReduce are main modules of the Apache Hadoop. There are numerous other projects related to Apache Hadoop including, Hbase, Hive, and Spark.

As two major modules of Hadoop, HDFS and MapReduce are designed for big data storage and parallel processing respectively. One of the strategies that Hadoop uses for big data processing is to move processing and computation close to the data site instead of traditional in-memory processing (Oussous, Benjelloun, Ait Lahcen, & Belfkih, 2017). It reduces the communication load and results in better performance for processing and computation (Oussous et al., 2017).

---

[1] http://hadoop.apache.org [4]

One of the major factors for the success of Apache Hadoop is that it has an important support community and the framework is growing bigger everyday with new components developed or improved and added to the Hadoop framework.

Although the Hadoop framework is improving every day, it is not the ultimate solution for addressing big data challenges in IoT. First of all, we need to find out how to use Hadoop components together in order to reach our big data goals for IoT. It consists of making decisions about the components and distribution, and how to connect them together. For this thesis, we are not focusing on the technologies, but on the architecture for using the technologies together. Moreover, the interoperability or big data variety issue is out of the scope of Hadoop technologies. In summary, the goal of this thesis is to provide an architecture for IoT and then the components can be implemented using off-the-shelf big data technologies such as Apache Hadoop.

Hortonworks

Hortonworks[1] focus is on providing enterprise-ready scalable open solutions and they are using technologies such as Hadoop and Spark. One of their products is called Hortonworks Data Platform (HDP) and it has enterprise ready Apache Hadoop distribution and delivers big data analytics. HDP works on improving Apache Hive (Ahmed et al., 2017). One of the advantages of

[1] https://hortonworks.com

Hortonworks is that it makes using Apache Hadoop easier as it removes all the installation, preparation, and configuration complexity.

However, just like Apache Hadoop itself, Hortonworks cannot be the ultimate solution for addressing big data challenges in IoT and it differs from my proposed solution in two ways. Firstly, it does not address interoperability issues for IoT directly. All these frameworks can process structured, semi-structures, and also unstructured data and they claim that they address the big data variety issue through this. However, analyzing all the data together with different structures would be difficult, if not impossible. Secondly, our proposed solution is an architecture and each part of it can be implemented using Apache Hadoop or Hortonworks or other big data technologies. In other words, our solution defines how the bits and pieces of big data technologies and tools, including Hortonworks platforms, can work together to form a comprehensive solution for addressing big data challenges for IoT.

Pivotal Big Data Suite

The Pivotal Big Data Suite[1] provides open source scalable database technology. Two of the main parts of the Pivotal Big Data Suite are the Pivotal Greenplum and Pivotal GemFire. Pivotal GreenPlum is the analytics module that provides high performance massively parallel processing

[1] https://pivotal.io/

(MPP) analytics. Pivotal GemFire is a combined data and compute in-memory grid that is designed to be fast, scalable, and available (Stolz, 2018).

The Pivotal Big Data Suite still needs time to be adopted (Ahmed et al., 2017). The focus is performance and high data volume. It can be a good solution for addressing bid data volume. However, to address big data challenges we need other technologies for big data variety and velocity. Similar to other technologies, Pivotal Big Data can be the implementation that is used as part of our proposed architecture.

Cloudera Enterprise Data Hub

Cloudera Enterprise is based on the Apache Hadoop distribution and provides a platform that is fast, easy to manage, and secure so that users will not need to focus on the technology and can focus instead on the analytics result that they want. It uses Apache Spark for in-memory data processing as well as Apache Impala, Solr, Kudu, and Hadoop for analytics, search, storage, and optimization. (Cloudera, 2017)

Cloudera Enterprise is similar to our solution in that it is using multiple technologies to provide a platform for big data analytics. However, first of all, it is a general solution and is not finetuned for IoT. Moreover, if users find a more suitable technology for one part of the platform they cannot easily replace and customize it. As discussed, for this thesis we provide the architecture and users can implement our architecture by using different technologies.

MapR

MapR is an enterprise big data platform that is partly based on Hadoop. One of their products is the converged data platform that is using MapR-DB. It uses different Hadoop and Apache technologies on their big data platform. The main difference of our proposed solution with this platform is that MapR is a platform whereas our work is an architecture. Furthermore, our architecture addresses the interoperability issue and is finetuned for IoT whereas MapR is a general platform. Moreover, using their platform can result in the vendor lock in problem whereas our solution is based on an open standard and does not rely on any proprietary solutions.

Database Management Systems

Database Management Systems (DBMS) are the everyday solutions for data storage. There are different types of DBMS. In this subsection, we discuss RDBMS, PDBMS, and NoSQL and how they differ from our solution.

RDBMS

Relational database management system (RDBMS) is the most mature and widely used data management solution. As a result, RDBMS can be a solution for data management for IoT. RDBMS is transaction oriented; it focuses on atomicity, consistency, isolation, and durability (ACID) of transactions. These characteristics ensure data integrity and stable management of processing results (Choi, Jeon, & Yoon, 2014). In other words, the ACID characteristics refer to “all or nothing”, “the results of each transaction are tables with legal data”, “transactions are independent”, and “database survives system failures” respectively (Pokorny, 2013).

RDBMS is finetuned for high data reliability. However, sometimes achieving high reliability unavoidably introduces overheads and subsequently sacrifices performance. Meanwhile, there are a great many applications in which performance is a more important factor than reliability, such as analytics applications. In this case, RDBMS is not a suitable solution anymore (DataStax Corporation, 2013). IoT applications are such kinds of applications, because as mentioned before, performance in answering queries is one of the most important characteristics that an IoT application must have. Therefore, RDBMS is not an appropriate solution for IoT.

PDBMS

A more scalable solution than RDBMS is the Parallel Database Management Systems (PDBMS). PDBMS uses the full capabilities of multiprocessors in order to achieve high performance and high availability (Valduriez, 1993). In other words, PDBMS combines database management with parallel processing. Like DBMS, PDBMS is write-optimized (Qin et al., 2016) or transaction oriented (Whang, 2011). As a result, PDBMS is designed for homogeneous relational data. Unfortunately, IoT data is heterogeneous and PDBMS is not appropriate. An alternative to PDBMS is NoSQL data stores that are read-optimized (Qin et al., 2016) and suitable for large-scale heterogeneous data (Whang, 2011).

NoSQL

NoSQL stands for Not only SQL (Structured Query Language). The reason is that NoSQL data stores mostly do not use SQL for query processing. One of the reasons for moving from RDBMS to NoSQL is the data model flexibility in NoSQL. In addition, NoSQL data stores can

manage different types of data (i.e. structured, semi-structured, and unstructured) with less constraints than RDBMS. Finally, NoSQL does not have the above-mentioned overheads and performance issues of RDBMS (DataStax Corporation, 2013). As a result, when performance is an important factor in designing an application, using the NoSQL data store is considered a more suitable choice than RDBMS.

In addition, NoSQL data stores are more horizontally scalable than RDBMS. The reason is that these data stores relax and simplify some of the restrictions of RDBMS, i.e. relational data structure and transaction processing overheads. Key-value stores are the simplest NoSQL data stores and their tables contain only key-value pairs that can be accessed only through the primary key (i.e. key) very fast. These NoSQL data stores are also called big hash tables. Compared to RDBMS, key is the same as attribute name in the RDBMS table. There are also more complicated NoSQL data stores that have a collection of key-value pairs. (Pokorny, 2013)

The NoSQL database appears to be a more suitable solution for data management challenges in IoT, but it is not complete. Although we mentioned that the NoSQL database can accept heterogeneous data types, it does not mean that it can understand different semantics in heterogeneous data. As a result, although the NoSQL database helps in addressing the data volume challenge, it cannot address the variety and velocity challenges thoroughly. The IoT data management system requires a high level architecture that has the NoSQL database as a part of it, together with other components that address all the big data challenges.

As we discussed, there are different technologies related to big data and big data analytics, but none of these technologies are the ultimate solution for addressing big data challenges for IoT. However, these technologies can be customized and used together for that purpose. In other words, a big data system needs multiple tools and techniques (Marz & Warren, 2015). For this thesis, we propose an architecture in which these technologies can be used and the big data challenges in IoT addressed. Our architecture is based on the Lambda architecture (Marz & Warren, 2015) together with the SensorThings API (S. Liang et al., 2016) in order to address the big data volume, velocity, and variety challenges for IoT. We will discuss Lambda architecture in next section before moving on to the proposed architecture in the next chapter.

Lambda Architecture

Lambda architecture (Marz & Warren, 2015) is a new paradigm for the big data problem. It focuses not only on the scalability of the paradigm, but also on its ease of use and fault tolerance. Lambda architecture minimizes the complexity so that the paradigm can be developed and implemented easily by a small team and does not need enterprise resources. The main idea behind Lambda architecture is to provide an architecture with the capability of computing any arbitrary function over a large data set in real-time. Also, the main motivation behind it is that there is no single tool or technique that can fulfil the idea. Lambda architecture is based on a layered architecture and provides three layers that together can provide an infrastructure for creating a big data system.

The Lambda architecture consists of three layers: batch layer, serving layer, and speed layer. In the following sections, we will discuss the functionalities of each of these layers.

Batch Layer

Batch layer's main purpose is to hide the computation time from the client. Executing functions and queries on huge amounts of data, e.g. petabytes of data, is extremely time consuming and would not be acceptable to users. Batch layer precomputes the functions and creates batch views. Then instead of running user queries on the fly, they can be answered based on precomputed batch views in an efficient manner. Other than batch computing, the batch layer is responsible for storing all the data as the master dataset of the system. In short, the two functions of the batch layer are (1) storing the immutable master dataset, and (2) running arbitrary functions on the master dataset continuously and creating batch views.

For Lambda architecture, batch layer is proposed to be simple. With regards to that, the batch processing will be simple single threaded functions and the scalability will be handled horizontally. In other words, rather than being worried about complicated multithreaded processes for batch processing, Lambda architecture proposes adding more nodes to the processing system in order to increase performance as needed. The use of the MapReduce paradigm is proposed for batch processing in order to achieve horizontal scalability.

Master dataset is the core and ground truth in Lambda architecture. It is our raw data and it is the only important part of Lambda architecture that is required to be safeguarded from corruption, since all the other parts can be generated from this master dataset in case of loss. The

first suggestion of Lambda architecture for the master dataset is to store the data in its rawest form. The word data refers to information and facts that cannot be derived from anything else and serves as axiom for all other derived information. The master dataset should be a collection of these data. Information is a collection of knowledge about the dataset that can be extracted from the data. Also, queries are the questions that can be asked about our dataset. Moreover, views are information derived from our data in order to help in answering queries. Since information, queries, and views can be derived and answered by the data, data in its rawest format is what we need to store in our master dataset. The raw data will be the material for answering user queries. The rawer the data is, the more questions can be answered based on it.

In addition, Lambda architecture discusses that there is a trade-off between storing the data in a structured or unstructured format. It argues that sometimes unstructured data can provide more information. This happens when the algorithm for deriving structured data from unstructured data has a chance of improving over time, which means that we may be able to derive more information from the unstructured data in future with an improved semantic algorithm. However, if the algorithm for extracting structured data is simple and accurate, structured data should be stored in the master dataset. As a result, this factor should be considered when designing the data model for the master dataset.

In addition to the rawness of data, Lambda architecture defines the master dataset as being immutable and perpetual. It claims that if the big dataset is raw, immutable, and perpetual, the big data system will be more robust. The suggestion of immutability helps the master database to

be human fault-tolerant and also simple. However, for our proposed solution which is finetuned for the Internet of Things, we will use mutable data whilst minimizing this mutability to the least possible. We will discuss this in the next chapter when explaining our proposed architecture.

Lambda architecture suggests that the data should be eternally true. In order to achieve that, together with data immutability, data should be labeled with time. For our proposed architecture, we use the SensorThings API as the data model for the master dataset. The most important and useful entity of SensorThings for IoT is Observation and is time-stamped in two ways, the time that the observation is recorded and the time that the observation actually happened. We will discuss this in more detail in the next chapter as we elaborate on our proposed solution in more detail.

Data Model

Lambda architecture has a fact-based data model, which means that the data consists of a set of facts that are fundamentally immutable, atomic, and time-stamped units. Each fact also needs to be identifiable. Fact-based model is helpful for answering client questions and queries, and can tolerate human errors. Human fault tolerance is the result of immutability. Because with faulty updates to the data, we would lose the data, but with immutability, the update to the data will be stored as a new time-stamped data which can be removed if faulty. As we will discuss in the next chapter, the Observation entity has all the features of a fact-based model. Comparing SensorThings to the recommended fact-based model of Lambda architecture, we can say that

Observation is our fact-based model and the rest of the entities are storing metadata about those facts.

Batch layer has the advantage of both normalized and denormalized data models. The fact-based data model for the master dataset is for suggesting normalized data model for data, whereas the batch views are denormalized and optimized for client queries. Batch views are reconstructed from scratch periodically and they do not need to be updated. As a result, there are no disadvantages from being denormalized.

In Lambda architecture, there should be a schema for the fact-based data model. Lambda architecture suggests using graph schema and enforcing it with the fact-based data model. In our solution, we use the SensorThings API as our data model and the schema is already present.

Master Dataset

Master dataset is where all the data is stored in the batch layer. The characteristic required for the master dataset is defined by the requirements we have for our batch layer. As discussed above, the data in the batch layer should be immutable and eternally true. This means that first the data will be accumulated and the master dataset must be scalable. Secondly, the data will not be updated. As a result, the master dataset only needs to be efficient for appending data and it does not need to be efficient for randomly accessing the data. As a result, the data does not need to be indexed to be performant for updates.

In our proposed solution, Observation is our only immutable entity. However, we can argue that since other entities are only keeping metadata about Observation and therefore they are

much smaller in size, the rare update cost is negligible. As a result, we do not need to design our master dataset for frequent updates, although other entities are mutable.

Batch layer periodically accesses bulk data to compute batch views and we know that the data will be huge. Therefore, the master dataset should have the capability of processing large amounts of data. Finally, the master dataset should have the flexibility of compressing the data. Because the data is huge and can affect the cost, we will need to compress the data in order to reduce the cost at some point. However, the design for compression is different for various use cases and the master dataset should be flexible for that reason. Distributed file systems are one of the best fits for the master dataset in Lambda architecture as they have the required capabilities, whereas they don't have the extra features that lead to higher cost. Key-value data stores are one of the examples of overkill for the master dataset, because they have all sorts of optimization for random access which is not required for the master dataset and result in higher costs.

Batch Views Precomputation

There should be a balance between how much of the computation is done on the fly and how much we precompute into batch views. No precomputation results in very high latency in answering the queries, whilst precomputing everything is infeasible. The idea is to precompute some intermediate results enough for answering queries quickly.

There are two approaches for precomputing batch views: incremental computation and recomputation. Recomputation means that the batch views will be created every time based on the whole dataset. Incremental approach is updating the already computed batch views when new

data is added to the system. There is a trade-off for choosing each approach in terms of performance, human fault tolerance, and generality of the computation algorithm.

Incremental algorithms use less resources during computation, however, their batch view size might be significantly larger. Unfortunately, incremental approach is prone to human error whereas the recomputation approach is not. The reason is that, if there is faulty data in the dataset because of human error, we can delete it and the recomputation approach automatically computes the batch view based on the current dataset which does not have faulty data. But for the incremental approach the effect of that faulty data is in the batch views and will not disappear, since we only consider new data added to system. Moreover, the recomputation approach results in more general algorithms as it uses a simpler structure for batch views as well as simpler on-the-fly calculation for answering queries.

In order to have a more robust system, Lambda architecture suggests using a recomputation approach for computing batch views. However, an extra incremental approach can also be used for increasing the efficiency of batch processing. But a recomputation version of algorithm must be executed on data, maybe with less frequency than incremental algorithms, in order to ensure that batch views are human fault-tolerant.

MapReduce

First introduced by Google (Dean & Ghemawat, 2004), MapReduce is a distributed computing paradigm that parallelizes computation between a cluster of machines. Using MapReduce as the computing paradigm for a system makes the system inherently scalable, as all

that is required for processing more data is more machines in the cluster. As a result, the MapReduce paradigm is a good candidate for batch processing in Lambda architecture.

In the MapReduce paradigm, a computing program consists of *map* and *reduce* functions. Map function runs on parallel data blocks over a cluster of machines and produces some intermediate results, usually in terms of key-value pairs. Then the reduce function merges these intermediate results, usually using the *keys*, and creates the final computation result. The process of sending intermediate results of *map* tasks to *reduce* tasks is called *shuffling*.

MapReduce computation is fault-tolerant as well. The system automatically retries map functions in case there is a failure that is caused by hardware break down, memory overflow, disk storage overflow, or other causes, and tries to overcome the failures using this process. However, the whole program will fail if a failure happens more than a preconfigured number of times. The reason is that in this case the failure is most probably caused by a program bug rather than server issues. As a result, map functions should be deterministic as the system should be able to re-run it, as needed, seamlessly without changing the final result.

MapReduce algorithms for batch processing can get very complicated, as usually a series of continuous MapReduce jobs are required to compute a batch view. Lambda architecture recommends using pipe diagrams as a means for designing batch computation. Pipe diagrams are helpful in terms of designing batch computation at a higher level without the concerns for the details of the complicated MapReduce jobs. Pipe diagrams are also supported by different tools that can be used for batch processing including, but not limited to, Hive, Pig, and Cascalog.

The pipe diagram defines the batch processing in terms of the concepts of SQL queries. Each part has an operation and the input and output is defined concisely. The operation can be processing in terms of tuples, functions, filters, aggregators, joins, and merges. Although the concept for operations are from SQL definitions, they are not limited to predefined functions and can be user-developed programs.

After high level designing with the pipe diagram, each part can be translated into a MapReduce job, so that the whole batch processing will be executed in a scalable manner. Each operation is simple enough so that it can be easily implemented by MapReduce jobs. We will see this step in the next chapter when we discuss our proposed architecture in more details.

We discussed the details about batch layer and batch computing. More details about how these concepts apply to our proposed architecture will be discussed in the next chapter. In the next section, we will discuss the second layer of Lambda architecture, the serving layer.

Serving Layer

Serving layer provides access to batch views in an efficient manner. In other words, the serving layer is where the precomputed batch views are indexed and can be queried efficiently to answer user queries. The serving layer is in close connection with the batch layer, as the batch layer frequently computes and updates batch views. Due to the high latency nature of batch processing, batch views served by the serving layer are not up-to-date with recent data. Dealing with recent data is the responsibility of the speed layer which we will discuss in the next section.

The most important characteristic of the serving layer is to be performant for answering queries. In this case, latency, which is the time that is required by the serving layer to serve one query, becomes an important metric. Indexing is the solution for the serving layer to provide better performance and lower latency for answering user queries.

The indexing of the serving layer is different from indexes in traditional databases as batch views are distributed over a cluster of machines in the serving layer. As a result, how to distribute batch views amongst the cluster is also part of indexing for achieving lower latency. It adds a new rule to our indexing that the fewer machines that need to be accessed for answering a query, the lower the latency, and thus, the better the performance. Moreover, the serving layer can benefit from denormalizing the data for the purpose of improving the performance and hence reducing latency.

Denormalization is the process of storing redundant data for the purpose of minimizing expensive joins in order to achieve better performance. Denormalization has the cost of data consistency verification. However, in Lambda architecture this verification is not important as the main data in the master dataset is stored with the normal structure, and the serving layer in which we use denormalization has been frequently overridden by the batch layer. As a result, even if any inconsistency arises, it will be fixed soon when the next round of batch processing is completed.

Lambda architecture puts some requirements in the serving layer. The serving layer must be scalable, fault-tolerant, batch writable, and support random reads. Being scalable is inevitable as

it needs to query batch views that can be big. Using a distributed cluster of machines can help the serving layer to be scalable. The serving layer should tolerate machine faults whilst serving queries. As the batch views are re-created periodically, the serving layer should have the capability of dumping the current batch views and replacing them with newly computed batch views. And finally, since the serving layer is where the queries are answered, it needs to support random access to batch views as requested by different queries.

The serving layer does not need to support random writes. It is interesting as most of the complexity is introduced by random writes in the database systems. As the batch views are computed periodically, the serving layer only needs to be capable of replacing the bulk of batch views with new ones and there is no need for updating only part of the view. The only place in Lambda architecture which needs to be capable of random writes is the speed layer which handles the real-time data and will be discussed in the next section. The simplicity of the serving layer is a big advantage in Lambda architecture, especially because it is the container of the majority of the queryable data in the architecture.

Speed Layer

The only part left from Lambda architecture is the random write for real-time data processing and it is the responsibility of the speed layer. Unlike the batch layer, the processing in the speed layer needs to be done incrementally. However, having batch and serving layers in Lambda architecture makes the speed layer requirements narrower than a big data system supporting random writes. Firstly, the speed layer is only responsible for the data that hasn’t

been added to the serving layer yet, which makes the data size much smaller than the master dataset. Basically, the speed layer only handles recent data as old as the latency of batch processing which is usually a few hours. Moreover, the speed layer does not need to be as robust as the batch and serving layers, as the data in the speed layer is transient. As a result, even if any error occurs during processing it will soon be corrected as the data is moved to the batch layer for more robust processing.

If we want to compare Lambda architecture with traditional data architecture, we will find that traditional architecture only has speed layers which are implemented with relational databases. As a result, the processing in traditional data architecture needs to be far more complicated. However, since Lambda architecture assigns different roles and responsibilities to different layers, the whole architecture and processing is simpler and more robust.

The speed layer processes the real-time data and creates some real-time views. These real-time views fill the gap of the real-time data, that are not yet in the serving layer, for answering queries. As discussed above, the process for creating these real-time views is different from batch processing as it is incrementally processed because it has to be as efficient as possible. Although the responsibilities for processing and storing batch views are separated between batch and serving layers, for real-time views, the speed layer is responsible for both of the functions for real-time data.

As discussed, the speed layer needs to use incremental algorithms for producing real-time views. Recomputation is not a good approach for the speed layer for two reasons. Firstly, for the

speed layer, latency needs to be decreased as much as possible in order to produce real-time or near real-time results, depending on the application. Although the size of data that needs to be processed is less than the master dataset, the latency that we are looking for here is with an order of magnitude less than batch processing. The latency that results from recomputing real-time views can be in order of minutes. Thus, recomputing real-time views is not an option for the speed layer. Moreover, recomputation is very resource intensive, if not impossible, considering the rate of new data coming to the system. As a result, the incremental approach is considered the best for the speed layer.

Real-time views should be optimized for both random reads and writes. Random writes are required for the use of the incremental approach and random reads are needed because these views are used for answering queries. Furthermore, real-time views should be scalable and also fault-tolerant. This scalability is achieved by using distributed architecture. The speed layer should tolerate machine failures as with the serving layer. All the characteristics we mentioned for the speed layer are features of the NoSQL database. As a result, the use of the NoSQL database for implementing the Lambda architecture speed layer is recommended.

Whether the real-time views structure is the same as batch views structure or not depends on the complexity of the computation. Complex computations can easily be done in batch layer, as high latency is not the issue. However, such computation may not be suitable for the low latency nature of the speed layer. In this case, the computation can be simplified to only calculate approximate answers for the query. The layered nature of Lambda architecture provides the

flexibility for eventual accuracy which means that all the data will eventually end up in batch views which are accurate, and if there is any approximation in real-time view it is only temporary. Using approximation algorithms for real-time views are optional in Lambda architecture, but might be the optimal choice for more complicated computation.

Challenges for Incremental Processing for Speed Layer

Incremental computation for the speed layer brings up some challenges. The most important one is what is called CAP theorem. CAP theorem says that whenever the data system is distributed and partitioned, the data system cannot be consistent and available at the same time. In other words, for distributed data systems there is a trade-off between consistency and availability. This challenge is introduced by real-time updates and random writes to the system. Availability needs replicas of data whilst managing consistency is difficult with replicas especially considering real-time updates. The CAP theorem is also valid for batch and serving layers. However, the different nature of batch and serving layers makes the CAP theorem not an important challenge, as availability is the obvious choice for these layers rather than consistency. The reason is that batch views in the serving layer are always out-of-date because of the high latency of batch processing and as a result the serving layer is never consistent and availability would be the choice.

In contrast, the speed layer needs to be both available and eventually consistent somehow, which brings up the CAP theorem. The eventual consistency supports the theorem as the instant consistency is not an issue and we only need the final processed value to be consistent. Lambda

architecture recommends using conflict-free replicated data types (CRDTs) to assist eventual consistency. CRDTs limit the operations that can be done on different values and thus assist with consistency. For example, if the CRDT only allows addition and we find an inconsistency between two replicas, we know that the maximum value is the correct data and the other just misses some additions.

We emphasize that although the CAP theorem adds more complexity to the speed layer and conflicts and corruption to the processed data is a possibility, the final process that goes through the batch and serving layer will correct all the possible corruptions. In other words, any inconsistency or corruption that may happen in the speed layer is just temporary and will be fixed in the batch and serving layers soon.

One important challenge for the speed layer is the expiration of real-time views. It means removing real-time views as their data is added to the batch views in the serving layer. The simplistic approach is to use the databases that support data expiry and set the expiry time to the expected finishing time for batch processing. However, since it is just an estimated time, if the batch processing takes longer for any reason, for some time some data is neither in batch views nor real-time views. In order to prevent this situation, Lambda architecture recommends another approach. The suggested approach maintains two sets of real-time views and cleans one of them each time that the batch views get ready. It is clear that some redundancy is involved but the maintenance will be very simple. Queries are always answered using the bigger real-time views. Although some redundancy is introduced it is not much of a concern as the real-time views only

contain a very small portion of the data, usually couple of hours, and it is not big enough to be worried about.

Asynchronous vs Synchronous Updates for Real-time Views

There are two approaches in the speed layer for updating real-time views: synchronous and asynchronous. Synchronous updates mean that the database is locked for each request, does the update, and then releases it. As a result, the system halts for each request. On the other hand, asynchronous updates are added to a queue and executed in order in the database. Thus, an update may happen with some delay, which is usually between milliseconds to a few seconds in the speed layer. Meanwhile, it can benefit from batch processing multiple requests in the queue, which is not possible with the synchronous approach. Synchronous updates are good for transactional data, whereas asynchronous updates are good for analytics with better throughput and better management of high loads. Because of the benefits of asynchronous updates, Lambda architecture suggests using asynchronous updates for the speed layer unless there is a good rationale for using synchronous or transactional updates for the designed system.

In order to pursue an asynchronous approach, the speed layer should have the ability for queuing and stream processing. Moreover, fault tolerance is a feature we need to keep in mind for the speed layer. In order to achieve fault tolerance, all the processing should have the ability to re-run without the loss of information, just as for batch processing, in case there is machine failure and the processing is corrupted. This feature should be considered in stream processing and queuing.

Queuing

The queue that is used for speed layer should be persistent. Firstly, as discussed above, we want to be able to re-run the processing. If a machine is broken and there is no persisted queue, the data it was processing will be lost. Also, the data can be lost in an overwhelmed machine that is using most of its memory and computation resources because of multiple simultaneous queries. The persisted queue helps the speed layer to survive such situations, and tolerates those errors.

When using persisted queue, there should be a mechanism to empty the queue as the data is processed with the real-time views so that the processing units will then get the next top element of the queue to process. The simple acknowledgement approach does not work for the speed layer, as the data might need to be processed in multiple machines and consumed for multiple real-time views. Using separate queues for each processing is also not a good option, since the complexity will increase with the number of simultaneous data multiplied by the amount of processing that needs to be done on that data.

As an alternative approach, Lambda architecture suggests letting the applications keep tracking their processed data. In this approach, the persisted queue will contain all the data, and each processing application keeps track of the data it has already consumed and gets the desired data part from the queue. This approach also fulfils the requirement for the capability of re-running or re-playing the data in case of machine failure as the data can be retrieved back from the time the machine crashed.

In order to maintain the queue and prevent it from getting very big over time, there will be a Service Level Agreement (SLA) that the queue always keeps data for a certain amount of time or until the queue reaches a specific size. The time limit or the maximum size can be configured based on the system in such a way that ensures that no data will be lost for real-time views. In the meantime, if for any reason, some data is lost, the system will correct this situation when the data is added to the batch views. As a result, any possible loss will be temporary. This type of queue is called multi-consumers queue and is the recommended queue structure for Lambda architecture.

Stream Processing

The data in the queue is processed using stream processing to create real-time views. The data can be processed one-at-a-time or using micro-batch processing depending on the application. The one-at-a-time approach has lower latency and simpler programming models, whereas micro-batch has higher throughput. With a one-at-a-time approach, the system will process one data tuple at a time. However, this processing can be parallelized in a cluster to gain higher throughput.

One-at-a-Time Stream Processing

For the one-at-a-time processing approach, Lambda architecture recommends using the storm model for the speed layer which has a graph of computation representing the stream processing pipeline. This graph of computation is called a topology. Storm model suggests a single program deployed over the whole cluster for processing the topology rather than having

different processing for each node in the cluster. In this model, the executable uses different nodes to complete each part of processing in the topology.

The storm model contains streams as its main concept. Streams are a sequence of tuples and the tuples are named data values. The Storm model processes the streams and the result of the processing will be new streams. Spout is another concept in the storm model topology. Spouts are sources for the streams. Spouts can be where data comes to the system and then tuples and streams will be created from them. Another concept in the storm model is bolt which is responsible for executing actions on streams. The input and output of a bolt are both streams. Moreover, a bolt can have multiple streams for input and also produce more than one stream as output.

Most of the logics and basic processing of the topology is happening inside the bolts. In other words, the equivalent processing that we do in batch processing with filtering, aggregation, etc. is performed in bolts in the storm model for the speed layer. With these concepts, a topology can be re-defined as a network of spouts and bolts in which the bolts are processing the streams that come from spouts or the output streams of other bolts.

Bolts and spouts are abstract concepts and instances of them are called tasks in the Storm model. Tasks should have the capability of being run in parallel. This feature is similar to MapReduce batch processing in which the map and reduce functions should be inherently parallel. In this model, each bolt task is receiving all the output streams from all the tasks that

generate the input for that bolt. The tasks for each bolt or each spout are spread over a cluster of machines.

Storm model uses stream grouping to define which tasks will receive the tuple that a task emits. Shuffle grouping and fields grouping are two different approaches for stream grouping. Shuffle grouping is basically distributing tuples using a random round-robin algorithm, whilst filed grouping is using hash function to decide which task will receive this tuple. Field grouping is more useful if the tasks are designed to receive some targeted tuples.

Storm model guarantees at-least-once processing for streams and tuples. To achieve this, the system should support retry in case of failure. Storm model proposes retrying from the root instead of retrying from the point of failure. Each spout tuple has a directed acyclic graph (DAG) when processed inside storm model topology. If a failure happens in any part of this DAG, processing of the spout tuples will be re-executed which guarantees at-least-once processing. Tuple DAGs can be big but there are tools that can track the DAG with high efficiency and scalability.

In order to be able to re-try a spout tuple, the processing should be idempotent, i.e. running the process multiple times should not affect the process result. However, the requirement for idempotent processing in the storm model is a soft requirement as the speed layer result can handle some inaccuracy since it will be temporary and will be fixed in the batch layer. Also, the failures are rare and any introduced inaccuracy is small. Using non-idempotent processing can

result in lower latency which will be desirable for the speed layer, whilst any introduced inaccuracy will be minimal and also temporary.

Micro-Batch Stream Processing

Micro-batch processing is another approach for stream processing in the speed layer. It has the advantages of better accuracy and higher throughput compared to one-at-a-time processing. This approach is useful for applications in which any inaccuracy, or even temporary inaccuracies, are unacceptable. However, micro-batch processing has slightly higher latency compared to a one-at-a-time processing approach. In other words, for applications that accuracy is a mandatory requirement, we can gain accuracy with micro-batch processing in the Speed layer, at the cost of higher latency.

Despite the one-at-a-time approach that tracks and processes each tuple DAG individually to guarantee the at-least-once processing, the micro-batch approach tracks a batch of tuples and the processing is executed in a specific order in order to achieve exactly-once processing. As such, each batch of tuples has a unique identifier (ID) and each batch will be processed after the completion of the previous batch.

Each batch of data is partitioned and sent to different tasks for parallel processing. Each intermediate processed result will keep the ID of latest processed batch. This way if processing one batch of tuples fails, the processing on that batch will be replayed and the tasks will update only the intermediate results that has the latest processed ID other than the ID of the current batch. As a result, the processing will not update the results that are already updated in the

previous round and only updates the results that haven't yet been updated during the time of the failure. Storing the ID of the latest processed batch adds idempotence to non-idempotent operations that are done in tasks. Thus, those tasks can be replayed multiple times but the end result is always the same.

Micro-batch processing consists of two parts: batch local processing and stateful computation. Batch local processing refers to computation done inside each batch that is local to that batch and independent from other batches. For example, partitioning the tuples in the batch in order to send to different tasks is one of the batch local computations. Stateful processing is the computation that needs to keep the state of the micro-batch across all batches.

Stateful computation in micro-batch processing results in the need for transactional spouts. Transactional spouts are sources of data that can partition data to micro-batch of tuples and replay exactly the same micro-batch in case a failure happens. For this to occur, the transactional spout should know which offsets from which partition the particular micro-batch can be derived.

As mentioned before, micro-batch processing has higher latency and throughput compared to one-at-a-time processing. The latency is caused by the time needed to coordinate partitions for micro-batch as well as waiting for one batch to complete the processing and then start the next batch. This latency may be as small as milliseconds to a second but it is significant considering the difference in order of magnitude. Moreover, higher throughput is achieved in micro-batch processing when processing batch of tuples all at once.

Pipe diagrams can be used to represent micro-batch processing as it is used for showing batch processing in the batch layer. Pipe diagrams can be used exactly as it is used in batch layer for batch local computations. However, for the stateful computation the pipe diagram needs to be extended as the pipe diagrams are designed to work on one batch at a time without the consideration of any state. In other words, the pipe diagram can show the micro-batch computation, but there is more to micro-batch processing than what can be shown in the pipe diagram. Micro-batch processing needs to keep batch IDs as state and it is hidden inside the pipe diagram abstraction.

As the final word about the speed layer, it is worth mentioning that the processing done in the speed layer is not always real-time processing. For example, finding inactive sensors, which are sensors that have not been active and sending data in the past hour, can be our use case and computing that is not real-time processing, but rather, processing the data from the past. As a result, depending on the use case and the questions that need to be answered, the processing of the speed layer may or may not be real-time processing.

Summary

In this chapter, we reviewed IoT data management and its challenges as well as the related literature and tools. Then we went over the details of the Lambda architecture which is our candidate for addressing data management challenges in our proposed architecture.

The Lambda architecture is an architecture for big data management and consists of three layers: batch, serving, and speed layers. The batch layer and serving layer are used for storing,

preprocessing and indexing large datasets. In other words, these two layers are designed to address the data volume challenge. The speed layer is designed for processing real-time data and queries. The Lambda architecture is also capable of addressing the data velocity challenge through the use of the speed layer.

We use Figure 7 to summarize the process and show the overall structure of the Lambda architecture (Marz & Warren, 2015).

The journey of data in the Lambda architecture for answering queries is as follows:

1. When new data is added to the system, it is sent to both the batch layer and the speed layer. This new data is added and stored in the master dataset in the batch layer. It is also used for incremental updating of real-time views by the speed layer.
2. For the Lambda architecture, the master dataset is designed to be append-only and contains the raw data that enter the system.
3. Common query functions are precomputed in the batch layer periodically and batch views are created. The preprocessing is performed from scratch each time. This characteristic makes batch views human fault-tolerant. The batch layer can compute any arbitrary function on any arbitrary data to support different applications.
4. Batch views are indexed and prepared for queries in the serving. The batch processing in the batch layer takes time. As a result, each time the serving layer is updated it is out of date compared to all the data that comes to the system at that point in time.

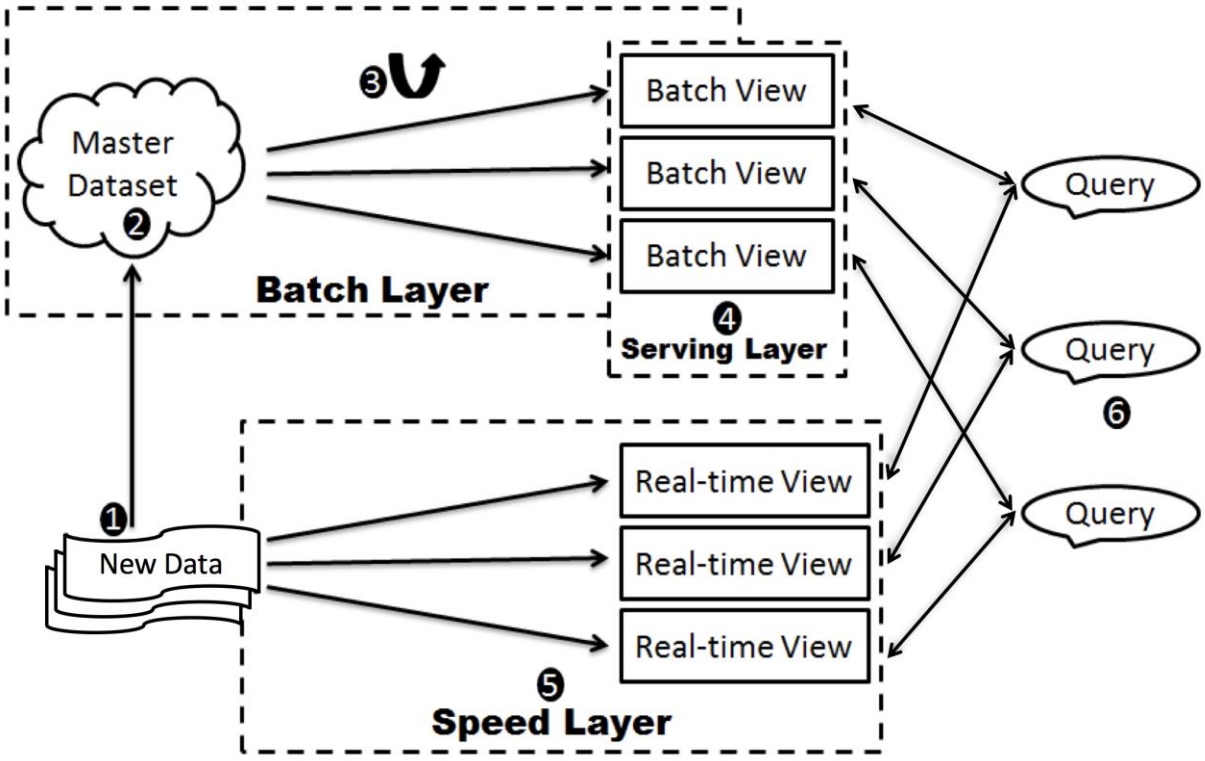

Figure 7 The Overall Process in Lambda Architecture (Adapted from (Marz & Warren, 2015)**)**

5. Because of the latency of batch processing, the serving layer is always out of date. This is when the speed layer comes in to answer the real-time queries for which answers are not present in the serving layer yet. To this end, the speed layer produces real-time views by using fast, incremental update algorithms on new data that enters the system. In this way, the real-time views are always up to date. The real-time views contain the views of data recently entering the system which are not yet absorbed by the batch views. As a result,

both batch views and real-time views are needed for answering queries. However, real-time views should be discarded as soon as they are absorbed into the batch views.

6. The answers for the queries are created by merging both batch and real-time views together. This is the final part of the data journey in the Lambda architecture.

In the following chapter, we explain how we use the Lambda architecture with the SensorThings API to address all of the 3 Vs of big data management challenges in IoT.

Integrating the Lambda Architecture with the SensorThings API

For this dissertation, I am proposing an open geospatial cloud service architecture for the Internet of Things. The focus of the proposed architecture is to solve data management challenges for IoT. We described the OGC SensorThings API and Lambda Architecture in detail in the previous chapters. In this chapter, I explain how using Lambda Architecture together with the SensorThings API result in an architecture that can overcome data management challenges for IoT. We first explain how to use the Lambda architecture with the SensorThings API, before discussing how this architecture overcomes big data managements challenges for IoT.

Proposed Architecture

The proposed architecture uses Lambda architecture together with the SensorThings API. In this section, we discuss how we can merge the Lambda architecture with the SensorThings API in order to create an architecture for solving big data challenges for IoT. For this we will discuss how SensorThings is used in each layer of the Lambda architecture.

Data Model and Master Dataset

As we discussed in the previous chapter, Lambda architecture requires a fact-based data model. The data model for the SensorThings API suits all the requirements for the Lambda architecture. It is normalized as suggested in Lambda architecture. Moreover, the SensorThings API data model satisfies the fact-based data model of Lambda architecture as the Observation entity is the centric entity in SensorThings for which all other entities keep metadata. As a result, Observation can be the fact entity. We propose making the Observation table immutable as

suggested by the Lambda architecture which means that Observation entries can only be added to the system but cannot be updated. Since Observations are sensor readings, they are immutable and time-stamped by nature. If there is an anomaly in the Observation data it can be detected in the application level by using anomaly detection or a cleansing algorithm. Furthermore, erroneous Observations can be deleted later by the user in another process and it will affect the batch views for the next batch processing period. By making the Observation entity immutable, we follow the Lambda architecture recommendation for achieving higher fault tolerance.

Although the FeatureOfInterest keeps the metadata for the Observation entity, we propose making this table immutable as well. In most of the use cases, the FeatureOfInterest is the GPS reading for the sensor and is either the same for all the sensor's Observations in the case of static sensors, or different in case of moving sensors. In either case, there is no need for updating the FeatureOfInterest entity for Observation. If there is faulty data for the FeatureOfInterest it will be corrected in the next Observation reading and thus no problems will arise.

All the other SensorThings entities including, Thing, Location, HistoricalLocation, Datastream, Sensor, and ObservedProperty keep metadata for Observation and they rarely change. We loosen the Lambda architecture requirement of immutability for these entities and give the user the capability to update them in case the wrong data was entered in the first place. As these entities only contain metadata information and not facts, we can still benefit from the fault tolerance that the Lambda architecture offers.

As discussed, the SensorThings data model fits perfectly with the Lambda architecture data model and we can use this data model as our master dataset data model. To sum up, Observation is our fact table and immutable. FeatureOfInterest is also immutable. All other entities are metadata or so-called dimensions for the Observation table and rarely change, except when the user inputs the wrong information.

By using the SensorThings API as the Lambda Architecture data model, we benefit from the fault tolerance from the Lambda architecture as well as the flexibility of SensorThings to get Observations from different points of view, e.g. Sensor view or ObservedProperty view. The data can be queried to get all the Observations of a specific Sensor; all the Observations that observe the same phenomenon (known as ObservedProperty); or all the Observations that observe the same place (known as FeatureOfInterest). In summary, the system is flexible but fault-tolerant when we make Observation and FeatureOfInterest immutable.

Batch Views

When we use the Lambda architecture together with the SensorThings API, our batch views can be created based on the analytics use case on top of SensorThings schema. There is no limitation on what we can define for our batch views. Furthermore, geospatial information in the SensorThing API can enrich our IoT analytics with Lambda architecture and it can facilitate any type of spatiotemporal analytics on Observations whilst the Observations can also be categorized based on the phenomena they are observing. Moving on, we will discuss some popular use cases for SensorThings and IoT.

When using Lambda architecture with SensorThings API, batch views can either be summarized or aggregated values, or preprocessed analytics on our Observations with different dimensions from other SensorThings entities that can be carried out based on the use case. Also, batch views can chunk the data based on popular SensorThings queries for a fast read. An example of the second case is if /Things(id)/Datastreams?$exapnd=ObservedProperty,Observations($orderby=result desc;$top=1) is a very frequent SensorThings path; the response, as raw data can be stored as batch view for fast access. The dimension in this case would be Thing id and the result would be stored as a file and can be accessed every time we want to answer the query. The challenge for this query is multiple database joins needed to answer the query as well as the fact that finding the highest result is an expensive operation. Thus, using batch view and preprocessing can improve the performance of this query. When using Lambda architecture, the file in batch views needs to be compared and merged with the information we have in our real-time views so that our response would contain the highest result of all the time. Another situation is that in which the query is needed for only retrieving the highest Observations results. In that case, a flat structure can be stored as batch views and then queried from the serving layer. The flat structure in this case has the Thing required information, can be only its @iot.id, the Datastream required information, again can be only its @iot.id, the ObservedProperty required information, and the Observation that has highest result. This flat structure can be a table and can be queried in a fast and scalable manner from serving layer.

Another more sophisticated query could be /Observations?$filter=phenomenonTime lt {t1} and phenomenonTime gt {t2} and st_within(FeatureOfInterest/feature,{bbox WKT}) and Datastream/ObservedProperty/name eq {n1}&$orderby=phenomnonTime desc. It requests all the sensor readings that observe a specific phenomenon for a specific time range for a specific area. In this case, our dimensions are ObservedProperty, FeatureOfInterest, and time. The data can be chunked based on these dimensions and stored as batch views. It can be as simple as a file structure for dividing the data. We need to define a folder structure for time and FeatureOfInterest. Different indexing can be used for FeatureOfInterest such as QuadTree, GeoHash, ZXY tile number, etc. In (Khalafbeigi, Huang, Liang, & Wang, 2014) we proposed using Time tags as YYYYMMDDHH and the QuadTree index as the structure for the dimensions. For the query mentioned earlier, we need to add the ObservedProperty id into the structure. To respond to this frequent query with raw data, the tags can be hashed into file names for faster access and the result can be read from the file.

In the above use case, instead of QuadTree we can use the ZXY tile number (Wikipedia, 2018b) if the query is usually made for rendering the data on a map. For this, we duplicate the data for each zoom level in order to achieve a fast read when zooming in and out on the map. ZXY is used to store tile files for fast rendering. Z is the directory and X is the subdirectory and Y is the file name that keeps the tile data. In this format, each zoom has $2^{zoom} \times 2^{zoom}$ tiles. In this use case, we can use ZXY numbers as a way of indexing the data geospatially. Also, using this ZXY information, tiles can be pregenerated for rendering. The following is the formula for

calculating X and Y for each Zoom level based on latitude and longitude information available from the SensorThings API:

$$n = 2^{zoom}$$
$$\mathbf{xtile} = n \times ((\mathbf{longitude} + 180) / 360)$$
$$\mathbf{ytile} = n \times (1 - (\log(\tan(\text{radian}(\mathbf{latitude})) + \sec(\text{radian}(\mathbf{latitude})))) / \pi)) / 2$$

As mentioned, the other use case is when a summarized or aggregated value is needed from our Observations or if some more complex analytics are required for Observations. In this case, we do not keep all the data for answering queries and only the aggregated or analysis value is needed. Flat structures can be used as a way of storing the information after aggregation or other analytics. In the flat structure, each row has all the dimension information needed as well as the summarize, aggregated, or computed value for the fact which is the Observation in the SensorThings API. Basically, the flat structure is created by denormalizing the SensorThings schema in order to achieve higher performance for the specific analytics. The flat structure can be stored as a table in the serving layer and used for answering user queries. Although normalization can help keep the data consistent as well as requires less storage, denormalized data can be accessed faster. Since performance is the primary factor when it comes to batch views, denormalized data is a better choice for batch views.

In summary, using the SensorThings API with Lambda architecture does not limit how to define batch views. From the IoT point of view, since the SensorThings API has a comprehensive data model that can fulfill most of the IoT use cases, different geospatial and

spatiotemporal analytics can be easily done on the data and the preprocessed data can be stored in batch views. Also, in terms of batch processing, there is no limitation on the paradigm that we use for preprocessing the data. MapReduce or any other parallel processing paradigm can be easily adopted for preprocessing the data for increasing the performance of batch processing.

Serving Layer

Using Lambda architecture with the SensorThings API does not put any limitation on the Serving layer. For batch views, we can use file structures if we only want to segmentize our raw data for fast access, or we can denormalize and aggregate data in batch views and index them in the serving layer based on the dimensions chosen for the analytics. For geospatial indexing in the serving layer, we can use the applications that provide geospatial indexing. Also, we can use different types of indexing at the preprocessing level, such as ZXY tile numbers that we used for our case study implementation, so that we can use any tool for indexing in the serving layer.

As the Lambda architecture suggested, the serving layer should be optimized for random reads due to its purpose and this should be taken into account when we use the Lambda architecture together with the SensorThings API.

The rule of thumb here is using denormalization, as suggested by the Lambda architecture, for aggregated batch views. Furthermore, the file name structures/hashes can be used for raw data segmentation, and no constraint is introduced by SensorThings into the Lambda architecture.

Speed Layer

When using Lambda architecture with SensorThings API, the simple SensorThings server can be used for answering real-time queries whereas the batch and serving layer can be used for answering questions about historical data. It is possible because of the rich RESTful API that SensorThings API introduces. Moreover, the data can be added to the system in real-time using the lightweight MQTT protocol. As a result, a service that is compliant to the SensorThings API is ready for real-time processing for most use cases.

However, for more sophisticated analytics, it is recommended that the speed layer be implemented for real-time data which results in real-time views. Basically, we move some of the processing time from the run-time to the speed layer. When using SensorThings API with Lambda architecture, depending on the analytics, the real-time view can have the same structure of batch views or be in a rawer format. As an example, real-time views can be as simple as the denormalized version of the SensorThings schema which reduces join on the run-time. In this case there will be data duplication but the read time can be faster. When we use rawer structure for our real-time views, they can be used for answering more user queries, whereas the more aggregated views result in fewer questions that it can answer. However, there is less run-time processing for more aggregated views and answering the questions based on them is faster. As we can see, there is a trade-off here and a decision needs to be made based on the use cases for the designed system.

We discussed batch views for highest result for each Datastream of a Thing in the previous section. For this use case, we can use the same flat structure for our real-time views as for our batch views. The real-time stream processing that is required in the speed layer is as easy as comparing the new data result with the real-time view record and updating it if the new Observation's result is higher. Micro-batch processing can be very useful in this case as the highest Observation result in the batch can be compared and applied to the real-time views.

Similar to the serving layer, merging the SensorThings API with Lambda architecture does not introduce any constraints on Lambda architecture and all the suggestions and guidelines from Lambda architecture can be used for the speed layer of our proposed architecture.

In summary, the SensorThings API enriches the Lambda architecture with its well-defined schema and geospatial information which makes it suitable for IoT applications and analytics. As the SensorThings data model is comprehensive and can be used for most IoT use cases, the proposed architecture can also be used for most big data IoT use cases. In the next Section, we discuss how using the SensorThings API with Lambda architecture provides an architecture suitable for big data management for IoT.

Discussion

As we can see from previous chapters, big data management challenges can be divided into three categories: variety, volume, and velocity. In this section, we discuss how using Lambda architecture with SensorThings API addresses these challenges. Also, in the next chapter, we will

review the evidence that the proposed architecture is suitable for solving data management challenges for IoT.

Big Data Volume

The Lambda architecture is a solution for big data volume and velocity. Basically, the Lambda architecture batch and serving layers solve big data volume by using multiple tactics. Firstly, the Lambda architecture has a master dataset for storing all the data. The master dataset should be able to store large volumes of data but it is not necessarily optimized for random reads. Furthermore, in the batch layer, the Lambda architecture uses batch processing in order to prepare the data for answering user queries efficiently. The paradigms used here are preprocessing and denormalization which both lead to higher performance for answering user queries. Also, the Lambda architecture has the serving layer for storing and accessing the batch views that are created during batch processing. The serving layer needs to be optimized for random reads which leads to high performance for answering user queries.

It can be seen that by using the batch and serving layer together, Lambda architecture can handle high volumes of data and in doing so, provide a solution for the big data volume challenge. The key here is separation of concerns, as the master data set is optimized for writing high volumes of data and the serving layer is optimized for reading. Also, the batch processing moves some computation from runtime to processing time which increases read performance. There is only one concern left – batch processing can be time consuming and can take as much as a few hours. In this case, if we answer user queries from the serving layer, our response does

not contain the information from the data entering the system in those few hours. That is the reason behind the introduction of the speed layer by the Lambda architecture.

Big Data Velocity

Lambda architecture can solve the big data velocity challenge with its speed layer. The speed layer processes streams of real-time data and creates real-time views. These real-time views together with batch views can answer user queries in a timely manner. Different tools and techniques can be used in the speed layer for the system to be responsive to high volumes of data. Incremental analytics and micro-batch processing are two of the techniques discussed in the previous chapter. As a result, Lambda architecture can overcome the big data velocity challenge with its speed layer.

In conclusion, Lambda architecture can overcome big data volume and velocity challenges.

Big Data Variety

For handling the big data variety challenge, we propose using the SensorThings API. Using a standard is one of the solutions for addressing the interoperability challenge or big data variety for IoT. The OGC SensorThings API is a comprehensive and easy-to-use IoT standard published in 2016. Since then, there have been multiple implementations and the standard is widely adopted in the IoT world. We will discuss different implementations of the SensorThings API in the next chapter. The fact that the standard has been widely adopted in the two years since it was published shows the maturity of the standard and the community around it. Hence, the SensorThings API is a perfect candidate for handling the big data variety issue.

Lambda architecture addresses big data volume and velocity, and SensorThings API overcomes the big data variety challenge. We propose addressing big data 3 Vs challenges by using the SensorThings API together with Lambda architecture. We see in this chapter that the SensorThings API data model can match the Lambda architecture schema well. Also, using the SensorThings API with the Lambda architecture does not introduce any limitations or constraints to the Lambda architecture. As a result, we can see that there is no problem in merging the Lambda architecture with the SensorThings API and the resulting architecture can address big data volume, velocity, and variety challenges for IoT.

Explaining the details of merging the SensorThings API and Lambda architecture is much easier with an example use case and we will discuss it in more detail in the next chapter when we discuss our implementation for an air quality monitoring use case.

Results and Discussion

In this chapter, we are going to illustrate a use case that uses the SensorThings API with the Lambda architecture – which is our proposed architecture for IoT. Firstly, we describe our case study and how we gather our data. Then we discuss the different technologies used to implement our architecture. In the third section, we compare the proposed architecture with the naïve implementation of SensorThings. Finally, we will discuss how the proposed architecture improves the IoT system implementation and how it addresses the big data 3 Vs challenges.

Air Quality Case Study

We use the data from the Location Aware Sensing System (LASS) originating from Taiwan (LASS Community / Academia Sinica, n.d.). It is an open source and public environmental sensor network system. LASS gathers sensor data from different sensors and publishes the readings on MQTT. We subscribe to their MQTT and load the sensor readings onto our SensorThings service in real-time. There are different sensors reporting their readings on LASS. However, we only load the data from the Edimax sensors (EDIMAX Technology Co., 2017) which created around 90% of LASS data, because Edimax sensors are reliable and have well-defined API for easy loading.

The sensor readings from LASS have GPS information to use for FeatureOfInterest. However, although their sensors are static or rarely moving, the sensors report slightly different GPS values for every reading. When we load data from LASS into our service, we use Geohash (Wikipedia, 2018a) level 11 to check if the sensor actually moved or if it is GPS error and then

load the data into our SensorThings API. As a result, if a sensor is moved less than 7.4 centimeters, our loader detects it as a static sensor and uses the old FeatureOfInterest for its Observation.

We use the device's Media Access Control (MAC) address to identify the Thing for the device. We only load data from Edimax sensors as we know the Sensors in their devices and also their ObservedProperties. We create the Thing and corresponding Datastreams when we see the first reading from each MAC address. The other important factor for Edimax sensors is that we know the unitOfMeasurement for its readings which makes the sensor reading meaningful. After the first time that the entities are provisioned for each MAC address, the next readings will only be checked for GPS data and if the sensor moved, before being added as Observations to their corresponding Datastream.

We started gathering LASS data since September 2016 and currently we have around 4000 Things and more than 630 million Observations in our SensorThings so far. 22 phenomena have been observed which led to 22 ObservedProperties and 22 Sensors. After cleaning the GPS data, there are currently around 40,000 FeaturesOfInterest in our SensorThings API. The rate of reading for sensors are different throughout the time, but currently it is around 30 hertz which means there will be around 2.5 million Observations added to our SensorThings API every day.

We gather LASS data as a real-world dataset, load it into the SensorThings model and implement and experiment on our proposed architecture.

Case Study Implementation

We used Apache Hadoop and Azure technologies for implementing a case study for our proposed IoT architecture. SensorThings MQTT is the means of entering the data into our implementation. For our implementation, we used Azure HDInsight clusters which provides clusters using Hortonworks Hadoop distribution. HDInsight provides different cluster types for Hadoop, Storm, etc. All of them use Hortonworks, but they are optimized and finetuned for different use cases, i.e. the storm cluster is optimized for applications that use storm.

The first technology we used to stream the MQTT data throughout our system was Apache Strom. We used an Azure HDInsight Cluster with two namenodes and five supervisors. The cluster has a total of 36 processing cores as well as 200GB memory. We had a storm process for each entity that subscribed to its SensorThings topic and streamed their data through the system.

Master Dataset

We used Apache Hive to implement our master dataset. Apache Hive is an open data warehouse residing on top of the Hadoop Distributed File System (HDFS) and facilitates querying the data using its SQL-like query language, HiveQL (Leverenz, 2018). Hive automatically runs the queries using MapReduce which increases the performance for querying large datasets.

We defined a table for each entity in hive as well as two tables with identical schema to the Observation table for storing real-time Observations. As a result, the table we do the batch processing on is different from the table we add real-time data to as there will be no conflicting

problems, and we know exactly which part of the data our batch views belong to. We used the Azure HDInsight Hadoop cluster for our apache Hive with two namenodes and four workers. The cluster has a total of 24 processing cores as well as 200GB memory.

When new data streams to the system through storm it is saved in the corresponding table in our Hive master datastet. For our implementation, we used apache SQOOP to move our historical data from PostgreSQL to Hive and then the storm process adds all the new data that streams through the system.

Batch Layer

We used the Azure Data Factory to implement our batch processing. The Azure Data Factory is a hybrid Extract-Transform-Load (ETL) service for creating, scheduling, and managing data integration in a scalable manner (Microsoft Azure, 2018a). It is a serverless technology which means that Microsoft Azure automatically manages the backend for the system in the cloud. It supports different connectors and data flow. We used a copy data pipeline in the Azure Data Factory.

The pipeline starts from Hive, creates a summarized view from raw data using the Hive query, and stores it as batch views. We used the Azure SQL Data Warehouse (DW) for storing our batch views as the serving layer. Azure SQL DW is a cloud-based data warehouse that separates computing from storage. As a result each component can scale independently (Microsoft Azure, 2018b). Azure SQL DW supports columnar caching which is caching

frequently used columns and row groups. As it has distributed cloud infrastructure, the system can scale for large datasets very well.

Batch View Structure

For our batch view we chose time, FeatureOfInterest, and ObservedProperty as our dimensions and to keep count of and calculate the sum of the Observation results as facts. We use a flat table structure for our purpose. Since the geospatial aspect of our IoT data is important to us, we used the ZXY tile number for aggregating Observations. And we keep the aggregation for zoom zero to 12. This means that there will be redundancy in our batch views which means that we need more storage. However, this structure requires less run-time computation for answering user queries and leads to faster response.

Together with ZXY we also keep the centroid for latitude and longitude that is the average of latitude and longitude of all FeatureOfInterest in that tile. We can use these centroids for some applications that work better with coordinates and ZXY for other applications.

For the time dimension, we keep the data down to hours, and we aggregate from minutes. In our batch views, we keep the date and the hour as two separate fields so that grouping will be faster for daily run-time aggregation.

We always keep two timelines of batch views, one which is processing now, and one which is ready to use for queries. Once one set is ready to use the other starts processing again. We also have two timelines for our real-time views as described in the previous chapter. We will go over the real-time view timelines in the Speed Layer section.

Batch Processing

As mentioned earlier, we used the Azure Data Factory for our batch processing. Our batch processing has four parts:

1. We copy all the data from Hive real-time Observation table to the main Observation table to get the data ready for processing.
2. We delete all the Observations before this time point from one of our two real-time view timelines. (We will cover more about real-time timelines later)
3. We delete the outdated batch views from our serving layer.
4. We keep track of the batch and real-time views that should be used during the time when the batch processing is happening in SQL DW.
5. We run a hive query grouping the data based on our dimensions and create a view.
6. We save the batch view in Azure SQL DW.
7. We repeat step 3 and 4 for as many zoom levels as desired. We store batch views up to zoom 12.

We process different zoom levels sequentially, rather than in parallel, since our cluster is small in terms of memory and cannot handle them all together. For more powerful clusters it can be done in parallel. However, as we saw in the previous chapter, for the Lambda architecture optimizing the batch process is not critical as the system will not be held back for that processing.

The Query that is run on level 3 for zoom zero is as follows:

```
select count(*) as total_count, sum(result) as total_sum,
   data_stream.observed_property as observed_property_id,
   observed_property.description as observed_property_name,
   to_date(observation.phenomenon_time_start) as aggregation_date,
   HOUR(observation.phenomenon_time_start) as aggregation_hour,
   0 as zoom,
   floor((get_json_object(regexp_replace(feature,\"\\\\|\",\",\"),
      '$.coordinates\\[0]')+180)/360*power(2,0)) as x,
   floor((1 - ln(tan(radians(get_json_object(regexp_replace(
      feature,\"\\\\|\",\",\"),  '$.coordinates\\[1]'))) +
      1 / cos(radians(get_json_object(regexp_replace(
      feature,\"\\\\|\",\",\"), '$.coordinates\\[1]')))) / pi())
      / 2 * power(2,0)) as y,
   round(avg(get_json_object(regexp_replace(feature,\"\\\\|\",\",\"),
      '$.coordinates\\[1]')),5) as lat,
   round(avg(get_json_object(regexp_replace(feature,\"\\\\|\",\",\"),
      '$.coordinates\\[0]')),5) as lon
from sensorthings.observation  as observation join
   sensorthings.feature_of_interest as feature_of_interest
      on (observation.feature_of_interest=feature_of_interest.id)
    join sensorthings.data_stream as data_stream
      on (observation.data_stream=data_stream.datastream_id)
    join sensorthings.observed_property as observed_property
      on (observed_property.obs_property_id=data_stream.observed_property)
group by data_stream.observed_property, observed_property.description,
    to_date(observation.phenomenon_time_start),
    HOUR(observation.phenomenon_time_start),
    floor((get_json_object(regexp_replace(feature,\"\\\\|\",\",\"),
      '$.coordinates\\[0]')+180)/360*power(2,0)),
    floor((1 - ln(tan(radians(get_json_object(regexp_replace(
      feature,\"\\\\|\",\",\"), '$.coordinates\\[1]'))) +
      1 / cos(radians(get_json_object(regexp_replace(
      feature,\"\\\\|\",\",\"), '$.coordinates\\[1]')))) / pi())
      / 2 * power(2,0));
```

The Hive queries will be run using MapReduce automatically. Apache TEZ is used for scheduling and running MapReduce. Apache TEZ is an extensible framework for data processing in Hadoop that improves the MapReduce paradigm by improving its speed and scalability (Hortonworks, 2018).

Serving Layer

We used the Azure SQL Data Warehouse (DW) for storing and indexing our batch views in the serving layer which is a cloud-based data warehouse. The Azure SQL DW separates computing from storage. As a result, computation can be improved in terms of scalability independent from storage. It uses Massive Parallel Processing (MPP) for high performance and scalability. In its cloud-based architecture, there is a control node that prepares the query for parallel processing and then compute nodes run the query in parallel. Also, Azure SQL DW uses adaptive caching for fast response to frequent queries.

Azure uses a metric called performance level which defines how powerful the cluster is. For our implementation, we used the basic level with the least performance level provided, but in the experiment section we can see that our architecture still outperforms other solutions by order of magnitude.

Speed Layer

For implementing the speed layer of our architecture, we used Azure Eventhub and Azure Stream Analytics and our real-time views are stored in Azure SQL DW. Since we used the Azure platform for implementing a case study of our architecture, we chose to route our messages to Eventhub to create real-time views seamlessly in this platform.

When the new data streams into the system through storm it will be sent to Eventhub. Eventhub queues the messages automatically as required and works seamlessly with Azure Stream Analytics to make sure that all the data goes through the stream analytics. The stream

analytics job is to analyze the datastream coming into the system in small batches and storing the result in the SQL DW.

Real-Time View Structure

For our real-time views, we decided to keep our raw data and only add some information to it. Since the data in our real-time view is only there for a few hours and not very big, the computation for the dimension we defined can be done relatively fast in real-time. However, we added the ZXY tile number as well as separate date and hour columns to the raw data to minimize the real-time computation. As we add the ZXY tile number to our data, each raw data is duplicated to 12 datasets for our 12 zoom levels.

Similar to batch views, we keep two timelines of real-time views. One of them is used for answering user queries and the other is used when the current batch processing is finished. We keep two timelines as we want our batch and real-time views to be independent and to avoid any overlap in their data. In order to explain the rationale behind this decision, let's name our timeline view batch1, batch2, real-time1, and real-time2. As we can see in Figure 8, at the starting point of time, t1, we don't have any batch views and batch processing is started to create batch1. At this time real-time1 and real-time2 are identical. At time t2, batch1 is ready and contains all the info about the data before t1 whilst real-time1 has all the data from after t1. As a result, user queries can be answered using batch1 and real-time1 quickly and without any overlapping of data.

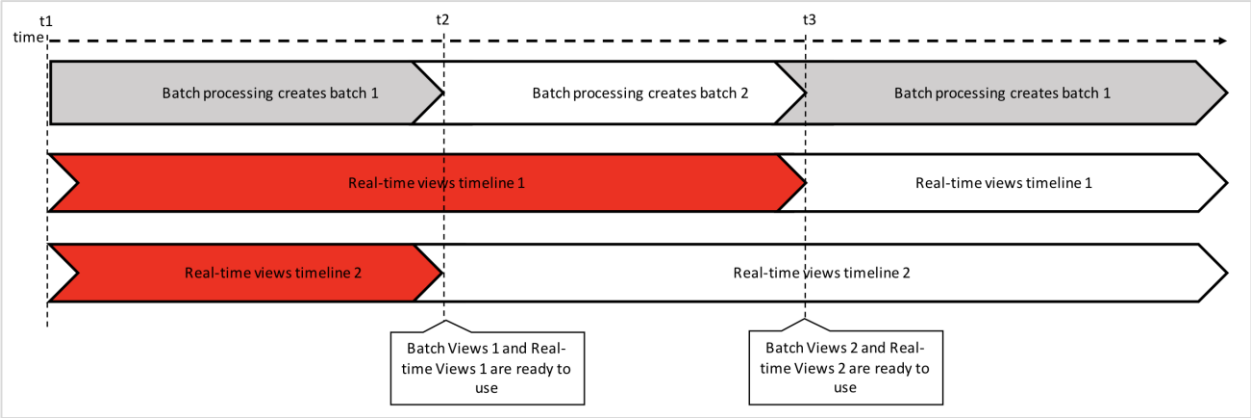

Figure 8 Timeline for Batch and Real-Time Views

At time t2, we start reprocessing our batch views. At this time, we use batch2 to store our result, as batch1 is in use for answering user queries. We delete all the data before time t2 from real-time2, so that after our processing is finished, batch2 and real-time2 won't have any overlap. We cannot do this delete for real-time1 since it is in use for answering user queries and batch1 does not have the info for the data between t1 and t2. When this round of batch processing is finished, batch2 and real-time2 are ready for answering user queries and we repeat the process for batch1 and real-time1 again. Using these two timelines for batch and real-time views helps in keeping batch processing and query answering smooth and quick and without any complications.

Figure 9 shows all the technologies we used for the implementation of our proposed architecture – the Lambda architecture with SensorThings API. Also, Figure 10 shows a screenshot of all the tools and technologies that we used from Azure to implement our proposed architecture.

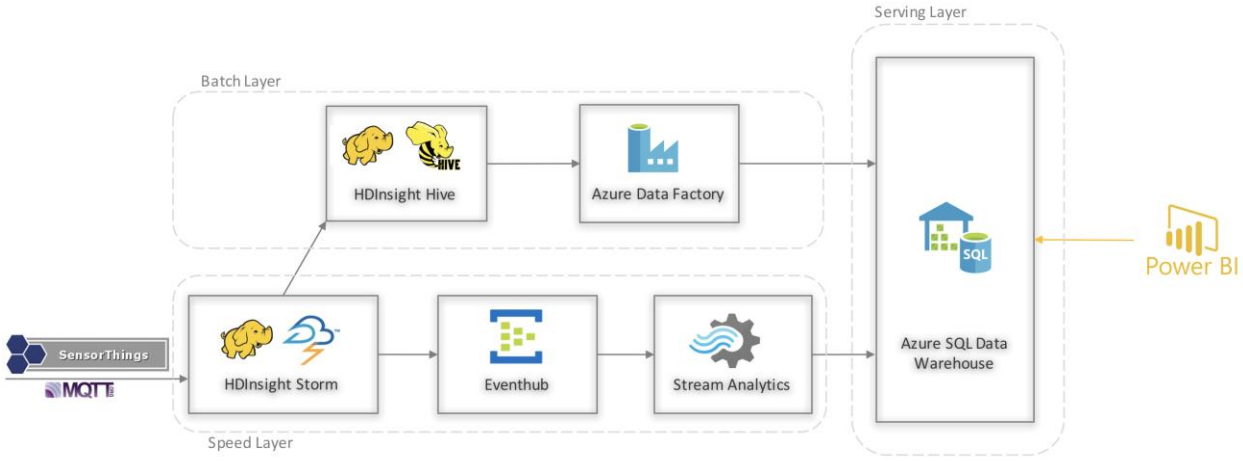

Figure 9 Case Study Implementation of Lambda Architecture with SensorThings API

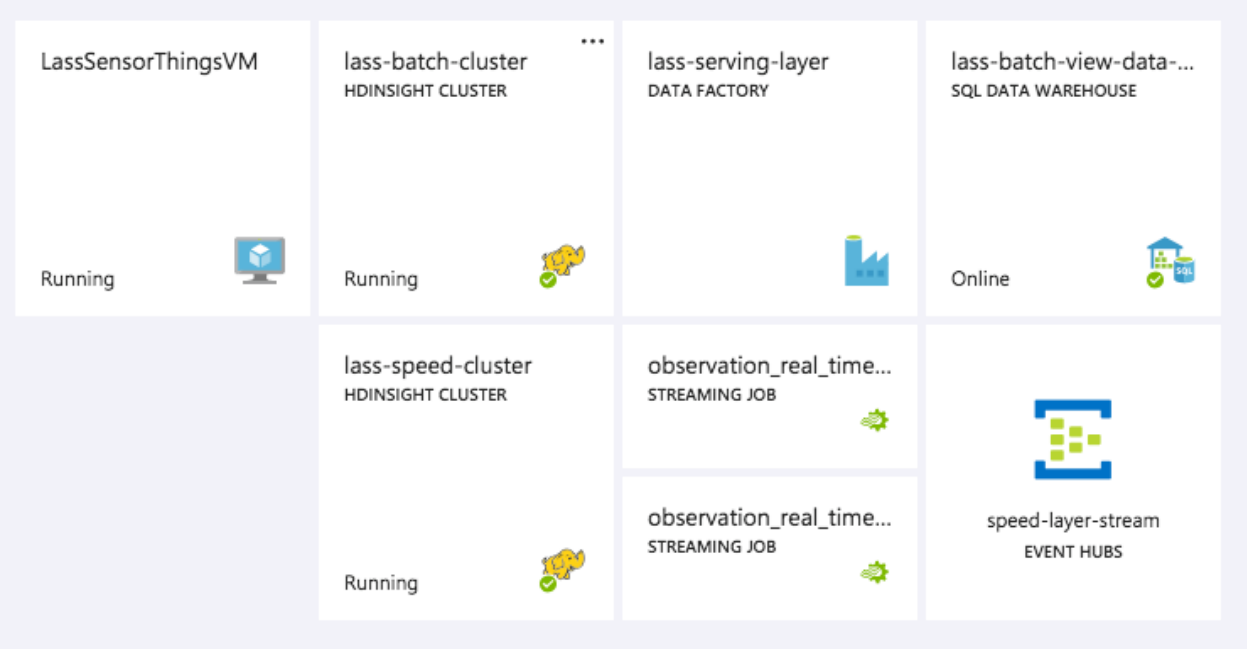

Figure 10 Screenshot of All the Azure Services We Used for Implementing Our Proposed Architecture

Any type of application tool can be used to query our batch and real-time views in SQL DW and port the results to the end user. Visualization is out of the scope of this dissertation. However, as a show case we used Power BI to show a possible application as shown in Figure 11 and Figure 12.

Figure 11 Power BI Screenshot Querying the Data from Our Proposed Architecture, Showing the Average Temperature for Dates Between 29/09/2017-26/04/2018

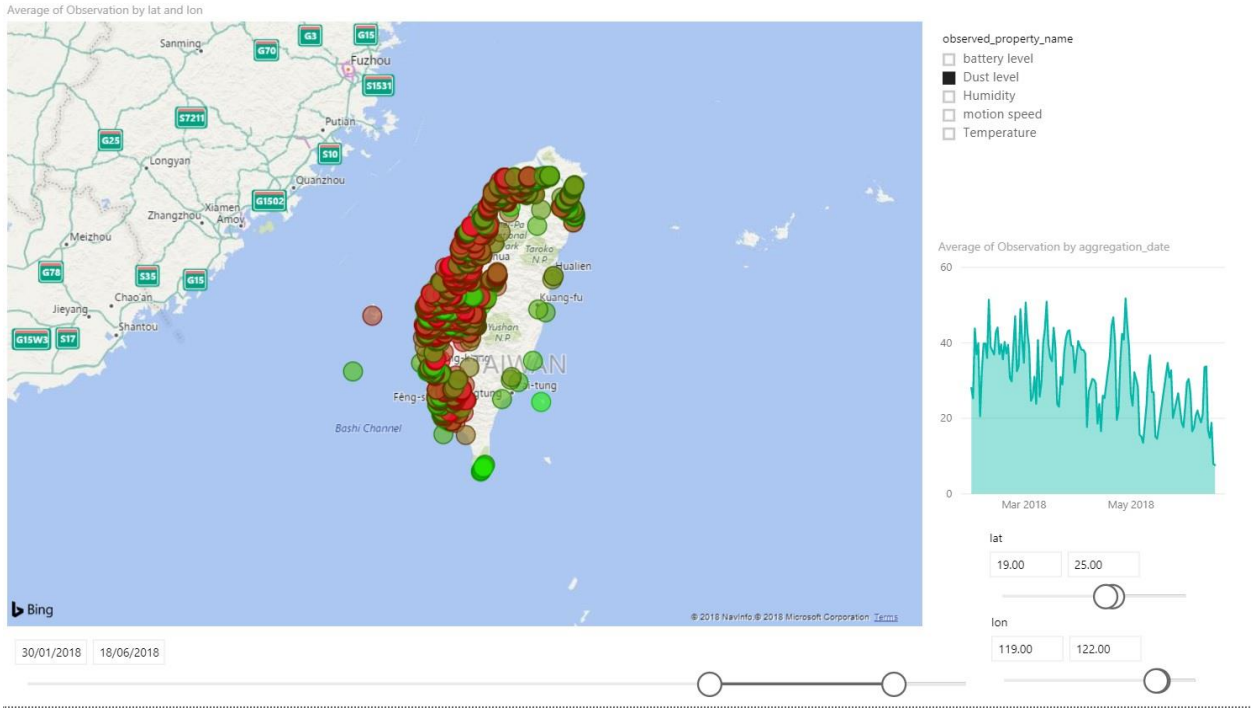

Figure 12 Power BI Screenshot Querying the Data from Our Proposed Architecture, Showing the Average Dust Level (PM2.5) for Dates Between 30/01/2018-18/06/2018 for the Select Area

In the following section, we will explain the experiments done on our implemented case study and discuss the results.

Case Study Experiments

In this section, we discuss the experiments for our implemented case study. For this dissertation, we proposed an architecture suitable for IoT data management and we discuss how it can overcome the big data 3 Vs challenges. In this section, we illustrate how our architecture outperforms other naïve implementations for IoT. In the first subsection, we discuss how our

proposed architecture can overcome the big data variety challenge. Then, we divide our experiments into two categories and each of them provide evidence that our architecture can overcome one of the big data Volume or Velocity challenges. In our final experiment, we illustrate how the proposed architecture performs compared to naïve implementations.

Big Data Variety

As explained before, big data variety refers to different types and structures of data that a big data system is required to deal with. It is also referred to as an interoperability issue for the Internet of Things. The interoperability issue in IoT rises from proprietary IoT systems that result in vertical standalone silos. Horizontally designed IoT systems with well-defined open standard APIs are required to address this interoperability issue. The OGC SensorThings API is a standard that can be used to overcome the interoperability issue. As we described in detail in the "SensorThings API, Details and Design Decisions" chapter, it is a geospatial standard published in 2016, and since then there have been multiple organizations implementing the standard and it is widely adopted by the IoT community. The number of implementations and adoptions show the maturity of the standard and that the SensorThings API is a good fit for solving the interoperability issue for IoT. In the following subsection, we will list different implementations of SensorThings API as well as its adoption.

In order to complete the standard life cycle, a test suite is developed for checking the compliance of different implementations with SensorThings API and ensure the interoperability

between them. I developed a test suite which was published together with the standard in 2016. It is available at the OGC compliance testing website[1] .[7]

Implementations

After the SensorThings API was published in 2016, multiple organizations started working on implementing the standard. My colleagues and I from the GeoSensorWeb lab in the University of Calgary implemented the very first prototype of the standard as a proof of concept whilst the standard was going through review and acceptance.

SensorUp Inc. implemented the first compliant implementation of SensorThings API using Java programming language and PostgreSQL DBMS. SensorUp also provides multiple resources, from webinar to documentation to client libraries, to facilitate the use of the SensorThings API.

In addition, Fraunhofer implemented an open source version of SensorThings API that is available[1] for anyone to fork. This[8] implementation also uses PostgreSQL DBMS for persisting the data. In order to make it easy to use, they provide a Docker container to make it easy to install and run.

---

[1] http://cite.opengeospatial.org/teamengine/[7]

[1] https://github.com/FraunhoferIOSB/FROST-Server[8]

Mozilla also spent some time implementing the SensorThings API using Node.js[1] . The other organization that worked on implementing it is Geodan Inc and they implemented another open source implementation of SensorThings API named GOST. GOST is written using Go language and PostgreSQL. CGI is also developing a SensorThings API named Kinota[2] . Kinota is also an open source implementation and aims to separate different concerns in their implementation (CGI Group Inc, 2017). As an example, they try to modularize persistence level in order to work with different RDBMS and NoSQL databases. Currently they are using Apache Cassandra as their persistence level. There is also an implementation of SensorThings API from a Bosch group named Gossamer[2] .

A quick search on Github[2] shows that there are currently 69 implementations so far for the SensorThings API, either implementing the core or a client for accessing the data. The number of implementations of the standard and the work around it two years from its publication shows it is well-received in the IoT community and is a good solution for solving the IoT interoperability issue.

---

[1] https://github.com/mozilla-sensorweb/sensorthings
[2] https://github.com/kinota/kinota-bigdata
[2] https://github.com/zubairhamed/gossamer
[2] https://github.com/

Standard Adoption

Since the SensorThings API was published in 2016, it was widely adopted for multiple IoT projects. The Department of Homeland Security (DHS) of United States adopted the SensorThings API for their Next Generation First Responders project (Department of Homeland Security, 2018). In addition, INSPIRE, which is Infrastructure for Spatial Information in the European Community, was extended to support the SensorThings API (Kotsev et al., 2018). Eclipse also has a project called Whiskers[2] , providing clients with lightweight gateways for the SensorThings API. IoT Systems adopted the SensorThings API as well [2] and they believe that it is the most robust API available for IoT. Geoconnections Canada is another organization that invested in the SensorThings API for the air quality smart city project around Canada[2] . The Smart Emission Project in Nijmegen was another project that adopted the SensorThings API[2] as a IoT platform.

Horizon 2020[2] is a European Research and Innovation program and Internet of Things is one of their focus topics for the ICT section. There was some effort to integrate the SensorThings API, one of which is INSPIRE (which we talked about). The City of Hamburg is one of the cities

---

[2] https://projects.eclipse.org/projects/iot.whiskers
[2] https://iotsyst.com/sensorthings/
[2] http://smartair.sensorup.com/
[2] http://data.smartemission.nl/
[2] https://ec.europa.eu/programmes/horizon2020/en/what-horizon-2020

that chose the SensorThings API as a IoT standard as part of the Horizion 2020 program (Meiling, Purnomo, Shiraishi, Fischer, & Schmidt, 2018). Furthermore, cities of Hamburg, Nantes, and Helsinki, as part of project mySMARTLife[2], adopted the SensorThings API (mySMARTLIfe Consortium Partners, 2017). The mySMARTLife project focuses on making cities more environmentally friendly by reducing CO2 emissions and increasing the use of renewable energy sources. For this project, the SensorThings API was chosen for the monitoring infrastructure.

What we mentioned above in terms of implementation and adoption of the SensorThings API are all of the work published and we believe that there is even more work in progress around the SensorThings API. The number of implementations and adoptions of the SensorThings API in only two years since it was published show the maturity of the standard and make it a good solution for solving interoperability issues for IoT. As a result, we chose to use the SensorThings API in our proposed architecture for addressing the interoperability challenge also known as, big data variety.

Big Data Volume

Big data volume challenge refers to the large size of data that needs to be maintained in the system. We proposed that by using the Lambda architecture together with SensorThings API, we

[2] https://www.mysmartlife.eu/mysmartlife/

can overcome the big data volume challenge. The techniques used here are first, preprocessing and denormalization, and second, separation of concerns for read and write as well as indexing. The dilemma here is that indexing can make read faster, whilst adding overheads to write. Thus, instead of having a system which is both optimized for read and write and requires maintaining a large dataset, we separate the concerns into different parts of the system and the batch layer is designed for quick write whilst the serving layer uses indexes to be optimized for quick read.

To support our claim about supporting big data volume, we compare the query response time of our implementation with naïve solutions. The first system that we compared is a system that implemented the SensorThings API with the PostgreSQL database. The reason we chose PostgreSQL for our experiment is because of the geospatial nature of the SensorThings API, PostgreSQL is one of the best relational DBMS that can be used and this is proven by the SensorThings implementations that are available.

The second system we compared our architecture with is a SensorThings service that uses the NoSQL database as its data store and we chose Hive which is the tool that we used for implementing our master dataset. NoSQL databases in theory manages large datasets better than RDBMS. The goal here is to show how using the NoSQL database in a well-designed architecture can result in much better performance. In other words, it is not the technology and tools that guarantee the performance, but the architecture and design that can help the system survive large datasets.

We used the real dataset that we gathered from LASS with around 40 thousand FeaturesOfInterest and more than 630 million Observations. In order to show how the system works when the data size gets bigger, we chunked our Observation data from 100 to 500 million Observations and repeated our experiment to find the trend.

The query that we tested the performance for is as follows:

```
select sum(total_count) as count,
   sum(cast(total_sum as float)) as sum,
   sum(cast(total_sum as float))/sum(total_count) as average,
   x,y
from [lass_batch_views].[lass_batch_view_table_2]
where observed_property_id=29014 and
   aggregation_date='2018-08-08'  and
   zoom=8 and
   x>=213 and x<=214 and
   y>=109 and y<=111
group by x,y;
```

This query finds the average dust level for the Taiwan area on August 8th, 2018. We need that information for each tile in zoom level 8. The above query is the query that needs to be sent to our batch views to get the response back. However, using PostgreSQL or Hive with raw data requires a more sophisticated query. The following is the same query for PostgreSQL:

```
select sum(cast(result as double precision)),
   count(*),sum(cast(result as double precision))/count(*),
   floor(((cast((feature::json)->'coordinates'->>0 as double precision))
      +180)/360*power(2,8)),
   floor((1 - ln(tan(radians(cast((feature::json)
      ->'coordinates'->>1 as double precision))) +
      1 / cos(radians(cast((feature::json)->'coordinates'->>1
      as double precision)))) / pi()) / 2 * power(2,8))
from observation  as observation join feature_of_interest as
feature_of_interest
      on (observation.feature_of_interest=feature_of_interest.id)
   join data_stream as data_stream
      on (observation.data_stream=data_stream.datastream_id)
   join observed_property as observed_property
      on (observed_property.obs_property_id=data_stream.observed_property)
where data_stream.observed_property = 29014 and
   observation.phenomenon_time_start>='2018-08-08T00:00:00.000' and
   observation.phenomenon_time_start<'2018-08-09T00:00:00.000' and
   floor(((cast((feature::json)->'coordinates'->>0 as double precision))+180)
      /360*power(2,8))>=213 and
   floor(((cast((feature::json)->'coordinates'->>0 as double precision))+180)
      /360*power(2,8))<=214 and
   floor((1 - ln(tan(radians(cast((feature::json)->'coordinates'->>1
      as double precision))) + 1 / cos(radians(cast((feature::json)
      ->'coordinates'->>1 as double precision)))) / pi()) / 2 *
power(2,8))>=109 and
   floor((1 - ln(tan(radians(cast((feature::json)->'coordinates'->>1
      as double precision))) + 1 / cos(radians(cast((feature::json)
      ->'coordinates'->>1 as double precision)))) / pi()) / 2 *
power(2,8))<=111
group by
   floor(((cast((feature::json)->'coordinates'->>0 as double precision))+180)
      /360*power(2,8)),
   floor((1 - ln(tan(radians(cast((feature::json)->'coordinates'->>1
      as double precision))) + 1 / cos(radians(cast((feature::json)
      ->'coordinates'->>1 as double precision)))) / pi()) / 2 * power(2,8));
```

Also, the query for Hive is as follows:

```
select sum(result) as sum,count(*) as count,
   sum(result)/count(*) as average,
   floor((get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[0]')+180)/360*power(2,8)) as x,
   floor((1 - ln(tan(radians(get_json_object(
      regexp_replace(feature,"\\|",","), '$.coordinates\[1]'))) +
      1 / cos(radians(get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[1]')))) / pi()) / 2 * power(2,8)) as y
from sensorthings.observation  as observation
   join sensorthings.feature_of_interest as feature_of_interest
      on (observation.feature_of_interest=feature_of_interest.id)
   join sensorthings.data_stream as data_stream
      on (observation.data_stream=data_stream.datastream_id)
   join sensorthings.observed_property as observed_property
      on (observed_property.obs_property_id=data_stream.observed_property)
where data_stream.observed_property = 29014 and
   to_date(observation.phenomenon_time_start)='2018-08-08' and
   floor((get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[0]')+180)/360*power(2,8))>=213 and
   floor((get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[0]')+180)/360*power(2,8))<=214 and
   floor((1 - ln(tan(radians(get_json_object(regexp_replace(
      feature,"\\|",","), '$.coordinates\[1]'))) + 1 / cos(radians(
      get_json_object( regexp_replace(feature,"\\|",","),
      '$.coordinates\[1]')))) / pi()) /
      2 * power(2,8))>=109 and
   floor((1 - ln(tan(radians(get_json_object(regexp_replace(
      feature,"\\|",","),'$.coordinates\[1]'))) + 1 / cos(radians(
      get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[1]')))) / pi()) /
      2 * power(2,8))<=111
group by
   floor((get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[0]')+180)/360*power(2,8)),
   floor((1 - ln(tan(radians(get_json_object(regexp_replace(
      feature,"\\|",","),'$.coordinates\[1]'))) + 1 / cos(radians(
      get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[1]')))) / pi()) /
      2 * power(2,8));
```

We measure the response time for all three systems. Figure 13 shows the performance of running the query on batch views from Azure SQL DW. The performance is shown in

milliseconds and the X axis is the number of Observations that are processed in our batch views. As we can see the trend is a constant value for the response time and even with increasing the number of Observations in the system we still achieve a good response time of less than a second from our batch views.

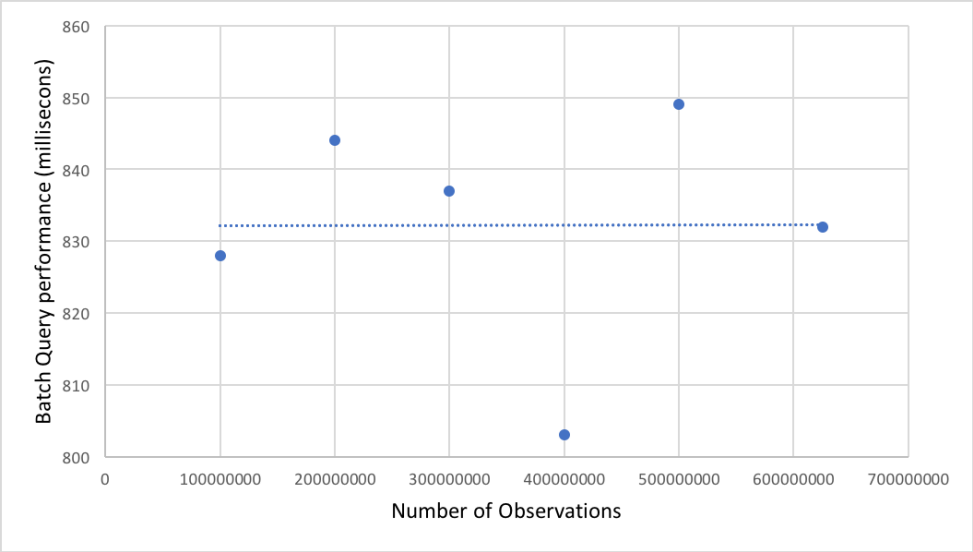

Figure 13 Query Performance on Batch Views in Milliseconds Based on Number of Observations

Figure 14 illustrates the performance of our query running on Hive raw data. As we can see the response time grows linearly with the number of Observations and it is in the order of a few hundred seconds compared with our batch views that respond in milliseconds.

Also, Figure 15 shows the query response time for PostgreSQL. The PostgreSQLtable in the test has multiple indexes on time and the foreign keys. It also has some geospatial index on FeatureOfInterest. However, since we are interested in ZXY, it is not of much help. What we

notice is that PostgreSQL performs better than Hive because of the indexes. However, it shows the polynomial trend and we can predict that when the number of Observations are almost 950 million, Hive will start outperforming PostgreSQL. We believe that the reason we don't see the linear trend here is that for smaller datasets indexes can help the system respond faster. However, as the data size grows the size of the index tree also grows. When the index size is big enough that it cannot fit in the memory it cannot help improve the performance anymore. However, this is not the focus of our experiment. Our experiment shows that our batch views query outperforms PostgreSQL as well.

Finally, Figure 16 shows all the three experiments together and we can see that our batch views architecture by far outperforms both PostgreSQL and Hive. We can say that our architecture improves how the system performs as the data size grows. In other words, the proposed architecture outperforms naïve implementations and it can address the big data volume challenge.

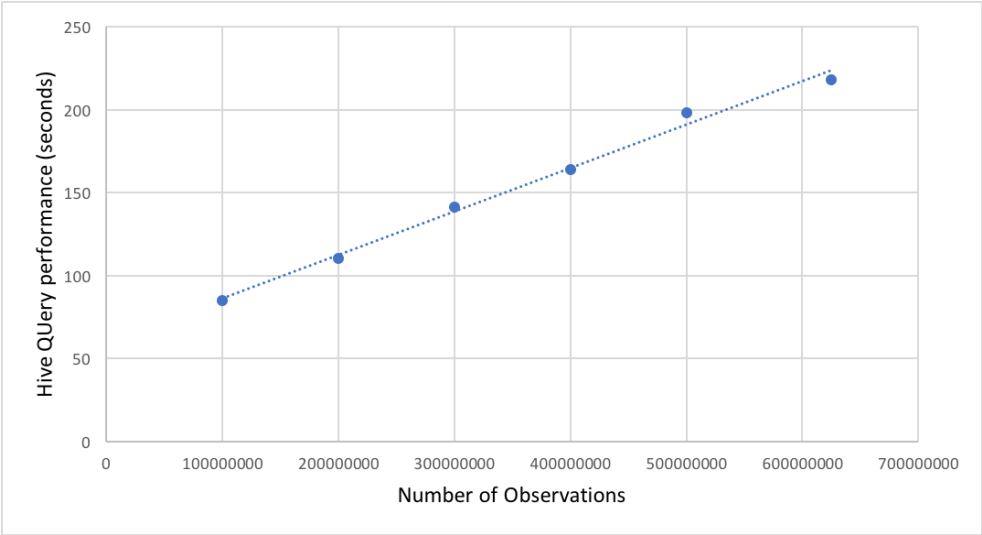

Figure 14 Query Performance on Hive Raw Data in Seconds Based on Number of Observations

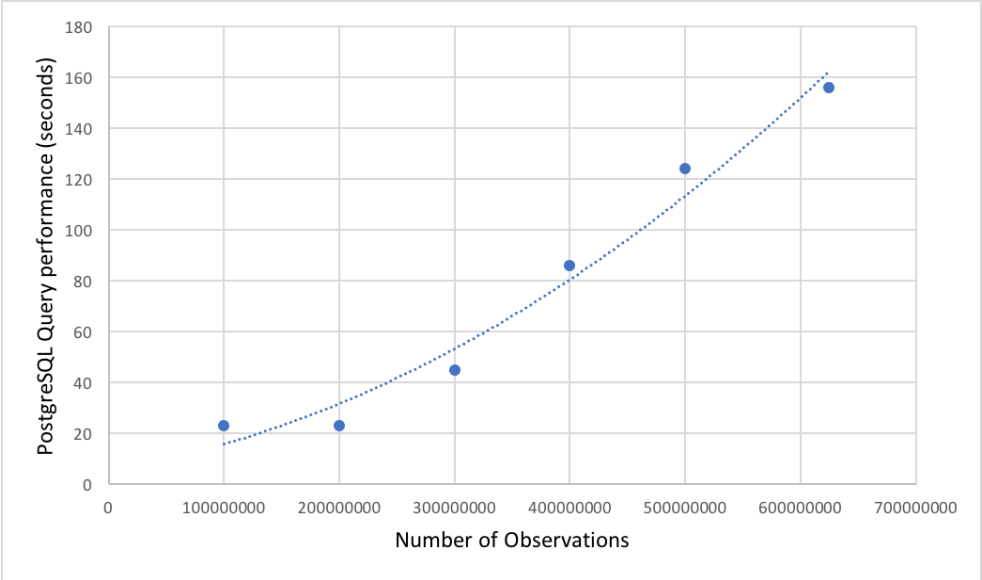

Figure 15 Query Performance on PostgreSQL Raw Data in Seconds Based on Number of Observations

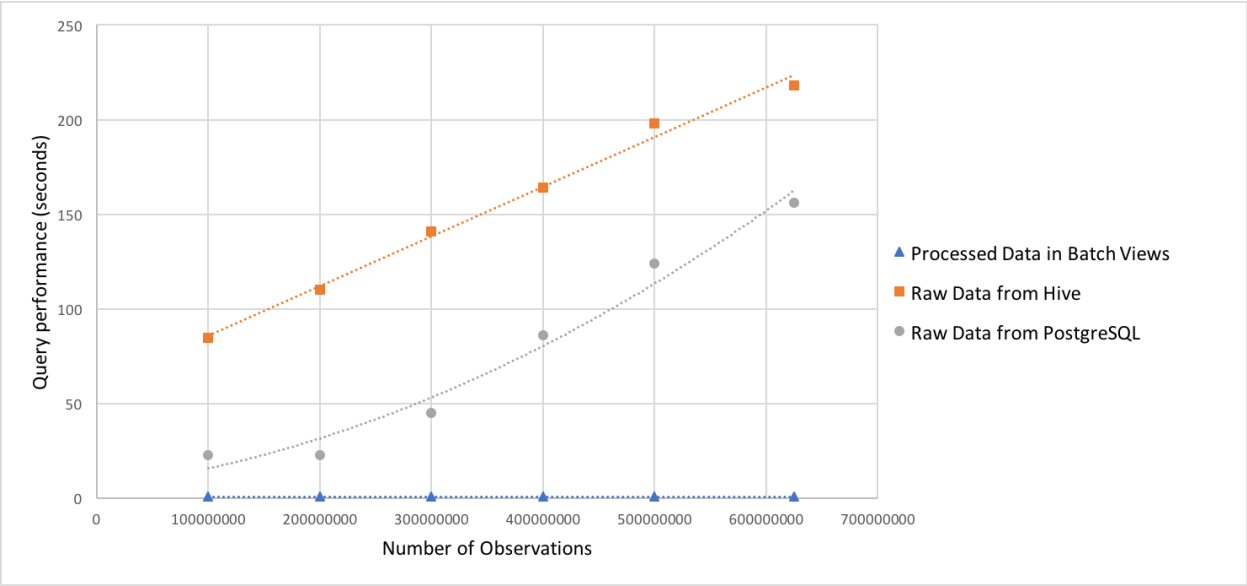

Figure 16 Query Performance in Seconds Based on Number of Observations

Big Data Velocity

Big data velocity focuses on real-time data and refers to streams of data coming to the system at high speed and is one of the challenges in big data. When using the Lambda architecture with SensorThings API, the speed layer is in charge for handling big data velocity. The SensorThings API uses MQTT protocol which is a lightweight publish-subscribe protocol for IoT. For our implementation, we used Apache Storm and also Azure products that are well-known for stream processing. In addition, Apache Storm is the tool that the Lambda architecture suggests for using in the speed layer.

In order to support our claim, we did two experiments. Firstly, we tested the efficiency of our real-time views. For this experiment, we want to show that our proposed architecture

outperforms naïve implementation in terms of answering queries about real-time data that streams to the system. The second experiment tests the latency of the speed layer, which means how long it takes for data to pass through the speed layer in our architecture and reach the real-time views.

For the first experiment, we sent a query to our real-time views as well as to PostgreSQL (which could be an alternative implementation). Basically, as an alternative to our implementation, we keep the batch layer and use a naïve SensorThings implementation with PostgreSQL for speed layer real-time data and only clean its data after each batch processing. We did this test based on the hours of data available in our real-time views. The size of data in our real-time views depends on how long the batch processing takes to complete, because after each batch processing the real-time views will be cleared. Firstly, we did an experiment to show the performance of our batch processing. We shrank our Observation table from 100 to 500 million data and carried out the experiment. Figure 17 shows the result of this experiment. As we can see, it shows a linear trend for batch processing time based on the number of Observations. For our current real system, the batch processing takes 6 hours and 30 minutes.

What we wanted to show in this experiment was that the size of real-time views is relatively small especially when compared to the size of the data as a whole, and it is as big as few hours of data. However, what we show in Figure 17 is not true for all systems. The reason is that first of all, the performance can be improved easily by adding more nodes to the cluster we have for the batch layer. Moreover, with more resources, we can parallelize the computation for each zoom

level which will reduce the processing time by order of magnitude. All we wanted to show was that real-time views will be as small as a few hours of data or a few days at worst.

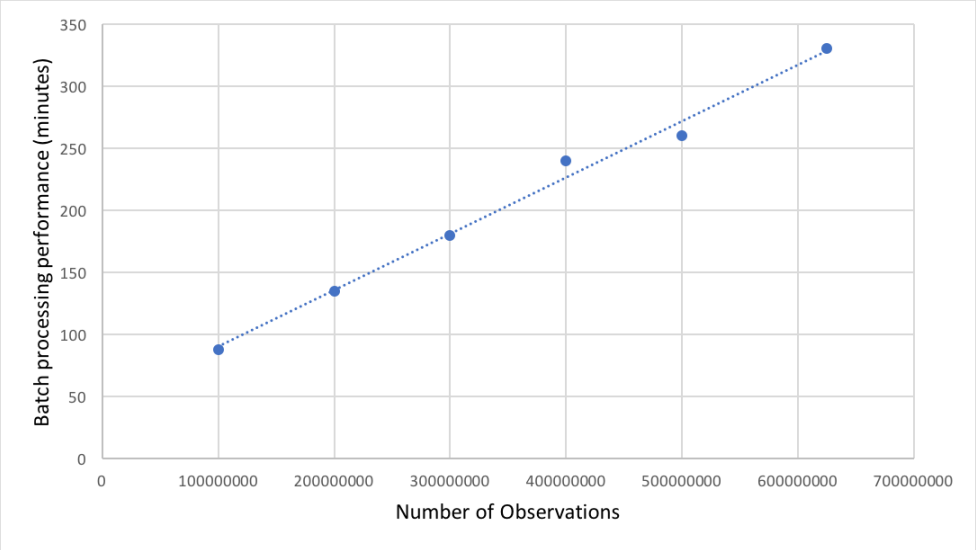

Figure 17 Batch Processing Time in Minutes Based on the Number of Observations

For our experiment, we test the performance of real-time views based on the data for 1, 3, 6, 12, 18, 24, and 48 hours of data. In order to carry out the experiment for PostgreSQL, we chunked the data in our Observation table for 1, 3, 6, 12, 18, 24, and 48 hours and we run the query on them.

The following is the query that we tested on our real-time views:

```
select count(*) as count,
   sum(cast(result as float)) as sum,
   sum(cast(result as float))/count(*) as average,
   x,y
from [lass_real_time_views].[lass_real_time_view_table_2]
where observed_property_id=29014 and
   aggregation_date='2018-08-08' and
   zoom=8 and
   x>=213 and x<=214 and
   y>=109 and y<=111
group by x,y;
```

Also, the query we sent to our PostgreSQL is as follows. It is similar to the query in the previous experiment, but with different data. Also, the constraint on time is not required when the data in the table is equal to, or less than, 24 hours.

```
select sum(cast(result as double precision)),
   count(*),sum(cast(result as double precision))/count(*),
   floor(((cast((feature::json)->'coordinates'->>0 as double precision))
      +180)/360*power(2,8)),
   floor((1 - ln(tan(radians(cast((feature::json)
      ->'coordinates'->>1 as double precision))) +
      1 / cos(radians(cast((feature::json)->'coordinates'->>1
      as double precision)))) / pi()) / 2 * power(2,8))
from observation  as observation join feature_of_interest as
feature_of_interest
      on (observation.feature_of_interest=feature_of_interest.id)
   join data_stream as data_stream
      on (observation.data_stream=data_stream.datastream_id)
   join observed_property as observed_property
      on (observed_property.obs_property_id=data_stream.observed_property)
where data_stream.observed_property = 29014 and
   observation.phenomenon_time_start>='2018-08-08T00:00:00.000' and
   observation.phenomenon_time_start<'2018-08-09T00:00:00.000' and
   floor(((cast((feature::json)->'coordinates'->>0 as double precision))+180)
      /360*power(2,8))>=213 and
   floor(((cast((feature::json)->'coordinates'->>0 as double precision))+180)
      /360*power(2,8))<=214 and
   floor((1 - ln(tan(radians(cast((feature::json)->'coordinates'->>1
      as double precision))) + 1 / cos(radians(cast((feature::json)
      ->'coordinates'->>1 as double precision)))) / pi()) / 2 *
power(2,8))>=109 and
   floor((1 - ln(tan(radians(cast((feature::json)->'coordinates'->>1
      as double precision))) + 1 / cos(radians(cast((feature::json)
      ->'coordinates'->>1 as double precision)))) / pi()) / 2 *
power(2,8))<=111
group by
   floor(((cast((feature::json)->'coordinates'->>0 as double precision))+180)
      /360*power(2,8)),
   floor((1 - ln(tan(radians(cast((feature::json)->'coordinates'->>1
      as double precision))) + 1 / cos(radians(cast((feature::json)
      ->'coordinates'->>1 as double precision)))) / pi()) / 2 * power(2,8));
```

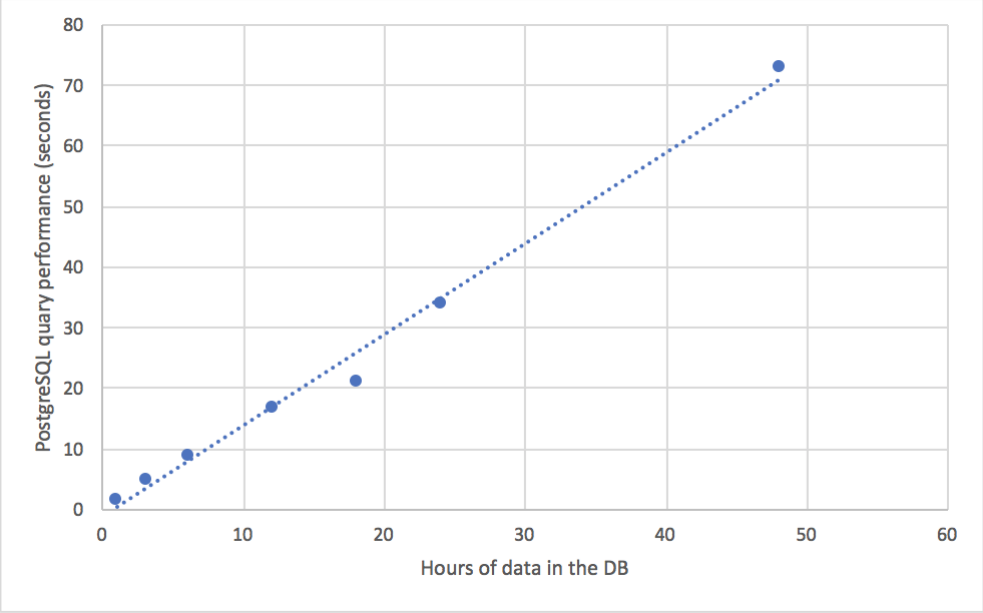

Figure 18 Real-Time Query Performance on PostgreSQL Raw Data in Seconds Based on Hours of Available Observations

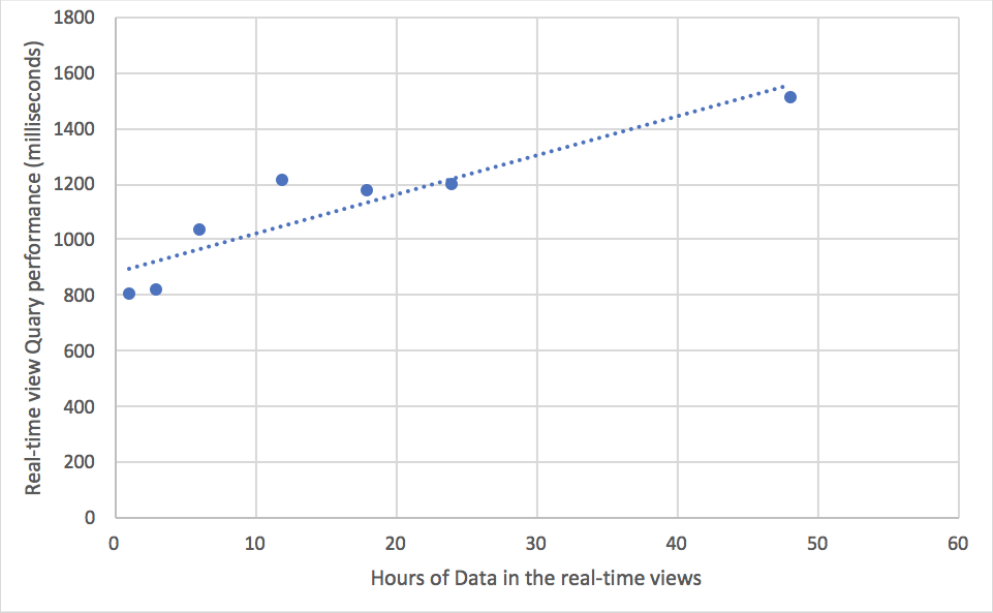

Figure 19 Real-Time Query Performance on Real-Time Views in Milliseconds Based on Hours of Available Observations

Figure 18 and Figure 19 show the performance of our query on PostgreSQL and real-time views respectively. We can see that the difference between the two measurements is not as high as for batch queries because the size of the data is smaller and there is not much preprocessing involved. However, our real-time views still outperform PostgreSQL for two main reasons. Firstly, because we precalculated our ZXY tiles in our real-time views while in PostgreSQL it needs to be calculated on the fly. Secondly, the tool we used for our speed layer, SQL DW, is optimized and finetuned for read, whereas PostgreSQL is optimized for both read and write. Based on the use case and the resources available for batch processing (affects how long the batch processing would take), designers can make the decision to use the SensorThings service itself for the real-time views if the latency is acceptable for their system.

The second experiment we did was testing the latency of our speed layer with different rates of streaming data. The current rate of the system for LASS is around 30 Hertz. We tested for 10 to 1000 Hertz of input data. We generated MQTT Observations using Apache Jmeter and calculated how long it took for the data to go through the speed layer.

Figure 20 illustrates the results of our experiment. What it shows is that increasing the rate of data does not add much to the latency, merely around 0.5% increase when 1000 Hertz is compared to 1 Hertz. This means that the data will not queue up and will not create huge latencies. The latency is roughly five seconds. As such, we can say that our system is working in near real-time and contains all the data up to five seconds ago. The five seconds can definitely be reduced as we used basic capacity for both Azure Eventhub and Azure Stream analytics. Thus,

we show that the architecture can handle big data velocity and still works fine even with a high rate of data and the high rate does not lead to queuing of data and added latency.

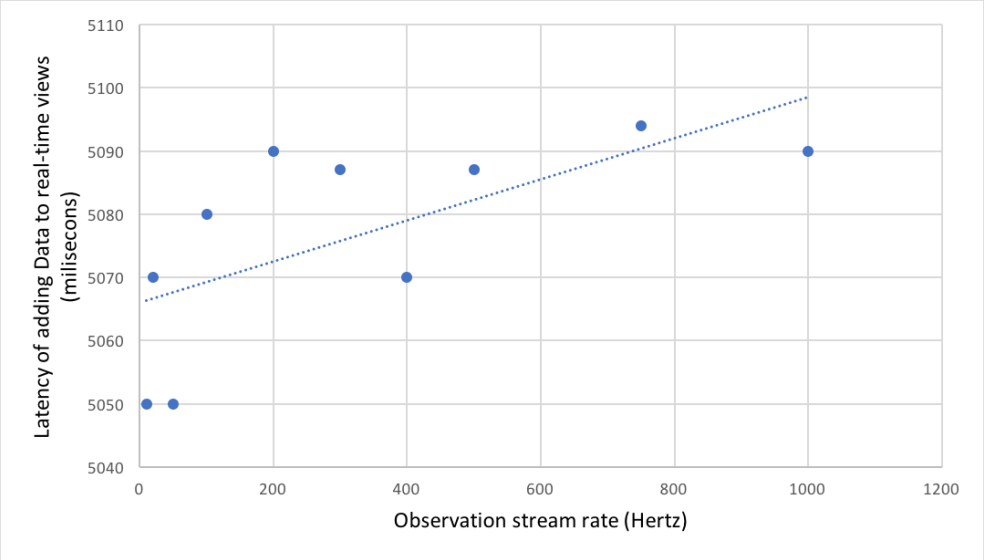

Figure 20 Latency of Adding Data to Real-Time Views in Milliseconds Base of Stream Rate in Hertz

Experiment with Query on All Data

As the final experiment, we tested the performance of our implementation with a query that requires the scanning of all the data. We did this experiment with our batch and real-time views as well as on PostgreSQL and Hive as alternative implementations.

The query we submitted to implementation was as follows:

```
select top(5) sum(count) as count, sum(sum) as sum,
   sum(sum)/sum(count) as average,aggregation_date,
   x , y
from
   ((select top(5) count(*) as count,
      sum(cast(result as float)) as sum,
      sum(cast(result as float))/count(*) as average,
      result_date as aggregation_date, x , y
   from [lass_real_time_views].[lass_real_time_view_table]
   where observed_property_id=29014 and zoom=8
   group by result_date, x, y
   order by average desc)
union
   (select top(5) sum(total_count) as count,
      sum(cast(total_sum as float)) as sum,
      sum(cast(total_sum as float))/sum(total_count) as average,
      aggregation_date, x , y
   from [lass_batch_views].[lass_batch_view_table_2]
   where observed_property_id=29014 and zoom=8
   group by aggregation_date, x, y
   order by average desc)) as tmp
group by aggregation_date, x , y
order by average desc;
```

Here is the same query that was submitted to Hive:

```
select count(*) as total_count,sum(result) as total_sum,
   sum(result)/count(*) as average,
   data_stream.observed_property as observed_property_id,
   observed_property.name as observed_property_name,
   8 as zoom,
   floor((get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[0]')+180)/360*power(2,8)) as x,
   floor((1 - ln(tan(radians(get_json_object(regexp_replace(
      feature,"\\|",","), '$.coordinates\[1]'))) + 1 / cos(radians(
      get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[1]')))) / pi()) /
      2 * power(2,8)) as y
from sensorthings.observation  as observation
   join sensorthings.feature_of_interest as feature_of_interest
      on (observation.feature_of_interest=feature_of_interest.id)
   join sensorthings.data_stream as data_stream
      on (observation.data_stream=data_stream.datastream_id)
   join sensorthings.observed_property as observed_property
      on
(observed_property.obs_property_id=data_stream.observed_property)
where observed_property.obs_property_id=29014
group by data_stream.observed_property, observed_property.name,
   to_date(observation.phenomenon_time_start),
   HOUR(observation.phenomenon_time_start),
   floor((get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[0]')+180)/360*power(2,8)),
   floor((1 - ln(tan(radians(get_json_object(regexp_replace(
      feature,"\\|",","), '$.coordinates\[1]'))) + 1 / cos(radians(
      get_json_object(regexp_replace(feature,"\\|",","),
      '$.coordinates\[1]')))) / pi()) / 2 * power(2,8))
order by average desc
limit 5;
```

Also, this is the query that we submitted to PostgreSQL:

```
select sum(cast(result as double precision)),count(*),
   sum(cast(result as double precision))/count(*) as average,
   floor(((cast((feature::json)->'coordinates'->>0
      as double precision))+180)/360*power(2,8)) as x,
   floor((1 - ln(tan(radians(cast((feature::json)->
      'coordinates'->>1 as double precision))) + 1 / cos(
      radians(cast((feature::json)->'coordinates'->>1
      as double precision)))) / pi()) / 2 * power(2,8)) as y,
   date(observation.phenomenon_time_start)
from observation as observation
   join feature_of_interest as feature_of_interest
      on (observation.feature_of_interest=feature_of_interest.id)
   join data_stream as data_stream
      on (observation.data_stream=data_stream.datastream_id)
   join observed_property as observed_property
      on
(observed_property.obs_property_id=data_stream.observed_property)
where data_stream.observed_property=29014
group by
   floor(((cast((feature::json)->'coordinates'->>0
      as double precision))+180)/360*power(2,8)),
   floor((1 - ln(tan(radians(cast((feature::json)->
      'coordinates'->>1 as double precision))) +
      1 / cos(radians(cast((feature::json)->'coordinates'->>1
      as double precision)))) / pi()) / 2 * power(2,8)),
   date(observation.phenomenon_time_start)
order by average desc
limit 5;
```

These queries try to find the five days and regions that are most polluted (have highest average dust level throughout the day). As a result, it needs to scan all the data and calculate the average and then find the highest five.

Figure 21 shows the performance of our proposed architecture based on the number of Observations. As we can see, there is a constant trend when we increase the number of Observations. We see that the response time is around 1.5 seconds which is more than the time taken for previous queries and the reason is that for this query scanning all the data is necessary.

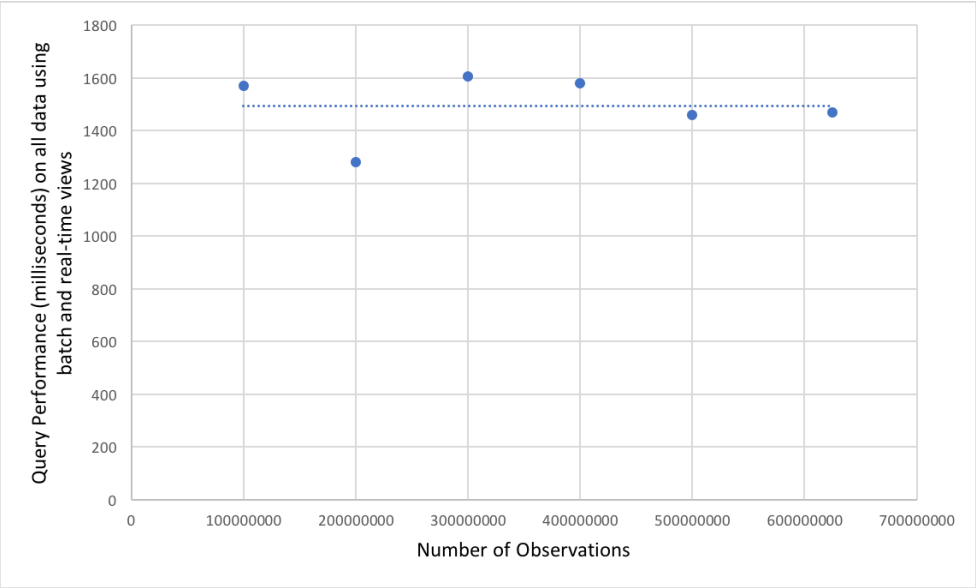

Figure 21 Query Performance in Milliseconds on All Data Using Batch and Real-Time Views Based on Number of Observations

Figure 22 shows the performance of the query on Hive. It shows a linear trend with the increase in the number of Observations, and the response time is a few hundred seconds – which is much more than the 1.5 seconds for our architecture implementation. We see here that our implementation not only performs much better than Hive, but also that its performance does not change with the increase of data.

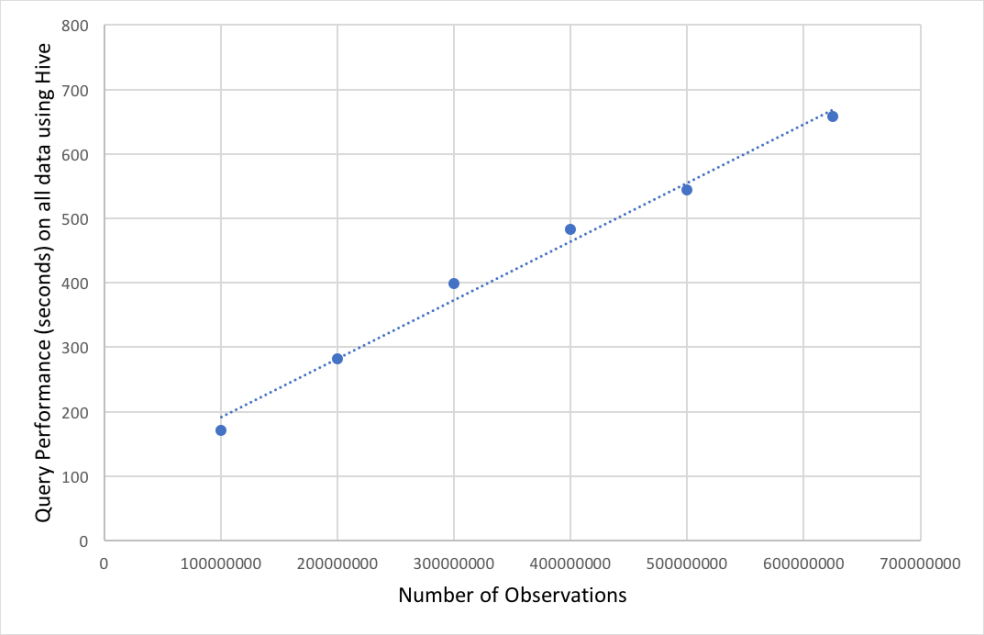

Figure 22 Query Performance in Seconds on All Data Using Hive Based on Number of Observations

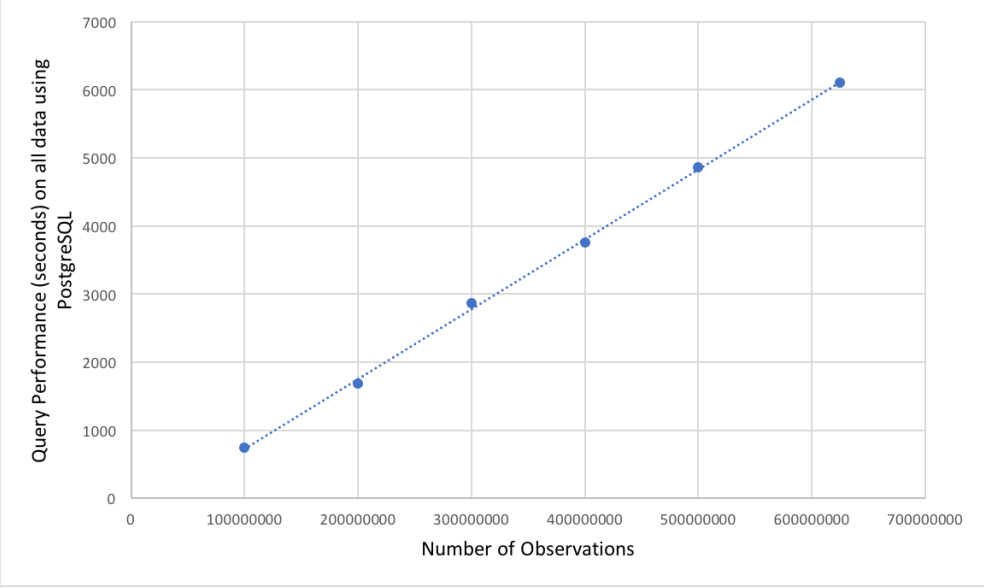

Figure 23 Query Performance in Seconds on All Data Using PostgreSQL Based on Number of Observations

The performance of the query executed by PostgreSQL can be seen in Figure 23. Again, we see a linear trend in performance based on the number of Observations. Also, we can observe that the performance decreases compared to previous queries which results from the need for scanning all the data and not using the indexes.

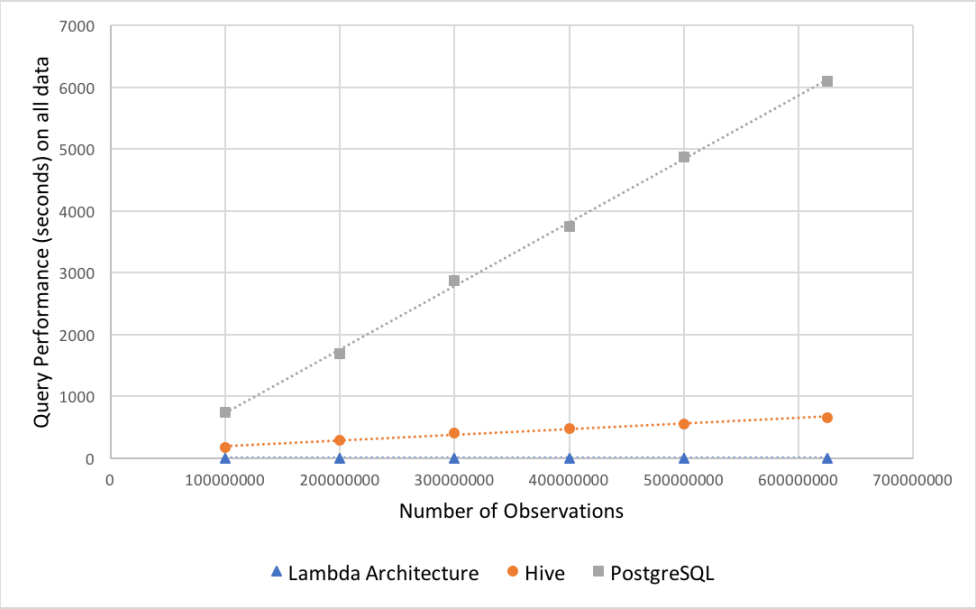

Figure 24 Query Performance in Seconds on All Data Using Our Proposed Architecture, PostgreSQL, and Hive Based on Number of Observations

Figure 24 illustrates the query performance of our proposed architecture compared to Hive and PostgreSQL. As we can see, our architecture outperforms Hive and PostgreSQL significantly. Also, we can see that Hive performs better than PostgreSQL which is expected as Hive is using MapReduce and is faster when whole data scans are required. Also, the indexing will not help in this case because all the data needs to be scanned anyway because of the nature

of the query. Even the trend is better for Hive because of MapReduce. We can see how preprocessing as well as using cloud-based SQL DW for indexing increases the efficiency of question answering in the implementation of our proposed architecture.

Summary and Discussion

In this chapter, we described how we implemented a case study for our proposed architecture and we saw how this implementation outperformed naïve implementations of the SensorThings API. There are couple of points worth highlighting as the result of our experiments.

First of all, as we discussed in the "SensorThings API, Details and Design Decisions" chapter, the SensorThings API is very flexible in terms of how the user can interact with the API in terms of its RESTful API. In order to implement the SensorThings API, the naïve method is relational databased because of the relational nature of the SensorThings data model. Also, PostgreSQL is a good choice for DBMS as it supports the spatial functionality that is required by the SensorThings API. This implementation works pretty well for small datasets. However, as we saw in our experiment results, it is not suitable for big data and we need to find alternative options for implementing the SensorThings API.

One alternative can be using cloud-based NoSQL data stores such as Hadoop technologies or Apache Hive, which was what we used in our experiments. There are two main advantages for Hive for implementing the SensorThings API. Firstly, it has a query language called HiveQL which is very close to SQL. As SensorThings RESTful API mostly follows SQL operations,

Apache Hive can be a natural NoSQL fit. Moreover, Hive supports MapReduce automatically, which means that for each HiveQL query submitted to Hive, it will be run using MapReduce automatically. This is another advantage, as using MapReduce is one of the most popular processing paradigm for large datasets. However, as we saw in our experiment results, Hive cannot be used to address big data challenges by itself. It can be used as part of the architecture, but more technologies are needed to handle all of the big data challenges.

Even when we only talk about big data volume or large datasets, we can see that preprocessing and using a flat structure for data would result in a much faster response to a query. However, there is a trade-off happening between preprocessing and the flexibility of the SensorThings RESTful API. When we change the structure of the data from the SensorThings relational data model into a flat structure, we prepare the data for a specific analytics or query. As a result, we need to prepare multiple flat structures and preprocess them in order to answer different queries.

In other words, our architecture needs to be designed in advance for common queries and analytics and then prepare as many flat structures as needed, and add them to our preprocessing pipeline. Although this might seem like a downside of our proposed architecture, we can argue that since our batch processing is always running on all the data and takes a few hours to finish, any new analytics or query we want to add to our system will take a few hours to be preprocessed and then the system would be ready. For our proposed architecture, we chose

higher performance although higher flexibility can still be achieved by adding more pipelines to the batch layer.

The other point worth mentioning, is that naïve SensorThings implementation with PostgreSQL can be an option for our speed layer. In this case, we create a batch layer with preprocessing and batch views and periodically clean the data that we have in our PostgreSQL. In other words, the batch layer would be an archiving place for the data that we have in our PostgreSQL. This option is very useful if most of the queries are about real-time data and the flexibility of SensorThings RESTful API is required.

However, as we observed in our experiments, using a proper speed layer can lead to higher performance for real-time data especially if complex analytics are required. There are two reasons for that. First, part of the processing will be moved to the speed layer rather than being processed at run-time. The second reason is that the place we store our real-time views should be optimized for random read, whilst the SensorThings PostgreSQL needs to be optimized for random read and write. For our implementation of the speed layer, we used SQL DW which is optimized for read and experiments showed that it outperforms PostgreSQL. As a result, a separate implementation of the speed layer can be useful for complex analytics and is a design decision that needs to be made based on the use cases of the system.

In summary, we propose using the SensorThings API which is a comprehensive and easy-to-use open geospatial IoT standard in order to overcome the interoperability issue for IoT. In addition, we propose using the Lambda architecture for implementing the SensorThings API as a

solution for the big data volume and velocity challenge in IoT. We looked at how a case study implementation of our proposed architecture outperformed naïve solutions such as PostgreSQL and Hive.

It is worth mentioning that although our implementation outperforms the alternatives in the experiments, our case study implementation is by no means the best or most optimized implementation and was still only a case study. We faced the limitation of 60 cores for our Hadoop clusters – by using more machines the performance would definitely increase. Moreover, with more powerful clusters, our batch processing can be parallelized which leads to better performance. Also, for Azure Data Factory, Eventhub, Stream Analytics, and SQL DW we use the least performance level possible to reduce costs, and we can achieve better performance by increasing the performance level of each of these technologies.

We used Hadoop technologies on Azure together with other technologies offered by Azure. Amazon AWS cloud technologies are another alternative for implementing our proposed architecture. Also, Apache Spark can be used to implement the whole architecture from scratch and can be run on a Hadoop cluster. These are just some alternatives for implementing our proposed architecture and sum up the main contribution of this dissertation.

Conclusion and Future Work

The Internet of Things consists of sensors and actuators embedded in everyday devices interconnecting and communicating through interoperable information and communication technologies. The real potential of IoT is in creating innovative applications by integrating and repurposing IoT sensing and controlling capabilities from different sources. However, proprietary IoT systems present now create silos that make the IoT goal almost unreachable as the applications need to deal with heterogeneous data from different systems. In addition to the heterogeneity problem, big data is a challenge for all technologies in the modern world. As predicted by CISCO and IDC, the number of internet-connected objects will reach at least 50 billion by 2020. As a result, IoT is facing heterogeneity and big data challenges. In this dissertation, we have proposed an architecture for IoT with the focus on data management challenges.

The proposed architecture merges Lambda architecture with the SensorThings API. The SensorThings API is used as a solution for the heterogeneity problem. One of the solutions for data heterogeneity or so-called interoperability in IoT is using a standard API. The SensorThings API is a mature open geospatial standard for IoT and has been mentioned by various literature. There are multiple compliant implementations for the API which result in no vendor locked in issue. The standard has been adopted by many industry projects since it was published two years ago. Hence, we think that the SensorThings API is a good fit for addressing the interoperability challenge for IoT.

Moreover, the Lambda architecture addresses big data volume and velocity challenges in its three-tier architecture. The Lambda architecture consists of the batch, serving, and speed layers. The batch layer has a master dataset which stores all the data. The key for the master dataset is that it is immutable and as a result, fault-tolerant. There is no need for the master dataset to be optimized for random readings as the user does not deal with the master dataset directly. There is batch processing in the batch layer which creates batch views. Batch views are used for answering user queries. As a result, it should be designed based on system requirements and use cases. Batch views are indexed and stored in the serving layer which is optimized for random reads. User queries are answered by searching batch views from the serving layer. Batch and serving layers address the big data volume challenge with two methods: preprocessing and separation of optimization for random read and write.

However, since batch processing is a time-consuming process and may take up to a few hours, batch views do not reflect all the data in the system and do not include recent data. This is the reason why the Lambda architecture has a third layer named speed layer. The speed layer receives streams of real-time data and creates real-time views. These real-time views together with batch views can answer user queries and the response reflects all the data in the system. The speed layer complements the batch and serving layers. Its purpose is to deal with high frequency real-time streams of data – thus addressing the big data velocity challenge. Different techniques such as micro-batch and incremental processing can be used in this layer to increase the performance for answering user queries.

Since the Lambda architecture is a solution for big data volume and velocity and the SensorThings API addresses the IoT interoperability challenge, we proposed that merging these two results in an architecture that can overcome big data challenges including volume, velocity, and variety (heterogeneity). For integrating SensorThings API with Lambda architecture, the data model from SensorThings will be used as the schema for the master dataset in Lambda architecture. The immutability constraint from Lambda architecture is only applied to Observations and FeaturesOfInterest without any problems as these two entities are immutable by nature. However, this constraint is loosened for other entities as they are rarely changed and we want the benefits from the SensorThings API's flexibility.

Batch views are defined using the SensorThings data model as well as real-time views. The rich data model in SensorThings helps facilitate the analysis with different spatiotemporal dimensions. Moreover, the MQTT protocol used by the SensorThings API can facilitate stream management and processing in the speed layer. There is no constraint introduced to the Lambda architecture by using the SensorThings data model and all the recommendations and guidelines from Lambda architecture can be used for the batch, serving and speed layers.

As a proof of concept, we implemented a case study for air quality data. We used a real online dataset from the Location Aware Sensing System (LASS) originating from Taiwan. The data is gathered using MQTT and stored in the SensorThings service. To design our batch views, we chose three dimensions of time, FeatureOfInterest, and ObservedProperty and summarized our Observations by calculating the average result for the given dimensions. The query that can

be answered by this batch view is “Give me the air quality of this region in this period of time”. We used ZXY tile indexing for locations as one of our query dimensions so that the data can be shown on the map with different zoom levels and granularity efficiently. We also stored the centroid for the locations of all the Observations we summarized into a tile so that it can be queried with them as well.

Hadoop and Azure technologies are used in the implementation of our case study as well. Apache Hive is used for the master dataset in the batch layer. Azure data factory is used for calculating batch views and they are stored in Azure SQL DW as the serving layer. For the speed layer, we used Storm, Azure EventHub, and Azure stream analytics and the resulting real-time views were stored in Azure SQL DW.

In our experiments, we compared our implementation with two systems, one that uses Hive and MapReduce, and the other using PostgreSQL as the data store. We tested three different scenarios: a query that needed only batch views for answering; one that only needed real-time views; and one that needed to use batch and real-time views together and that needed the scanning of all the data. Our experiments showed that our implementation outperforms Hive and PostgreSQL. We observed that if the dataset is small, PostgreSQL outperformed Hive because of its indexing. But for larger datasets, or for cases that needed the scanning of all the data, MapReduce outperformed PostgreSQL queries. In both cases, the case study implementation of our architecture performed better than Hive and PostgreSQL. As expected, we observed that the performance of our implementation has tested constant trends in all the scenarios.

In theory, since the SensorThings API is a solution for interoperability and the Lambda architecture addresses big data volume and velocity challenges, merging them together should result in an architecture that addresses the big data three Vs challenges. With our case study implementation and experiments, we showed that our theoretical hypothesis was true and our proposed architecture can be a solution for overcoming big data volume, velocity, and variety.

Future Work

There are several recommendations for future work in this dissertation. The SensorThings Tasking part will be published soon and the Rules engine part is a work in progress. The Tasking part is for tasking controllable IoT devices. The Rules engine is for processing events on the SensorThings API. As a result, the SensorThings Sensing, Tasking, and Rules engine parts complement each other and provide a standard for the whole IoT ecosystem. How to merge the Tasking and Rules engine parts with the proposed architecture can be valuable future work.

This dissertation proposed an architecture rather than focusing on the optimized implementation. It may also be worth experimenting with different possible implementations. There will not be an optimal solution for all IoT systems and the architecture needs to be implemented based on the requirements of each system. However, a study showing the differences between using different technologies for implementation can help to facilitate the use of this architecture for different industries. In other words, a study showing implementation best practices for different scenarios can be useful.

Moreover, various useful analytics for IoT can be explored to provide recommendations for different applications. This can be merged with the exploration of technology and the results can be used as guidelines for different industries to adopt the architecture.

Finally, the security option for the architecture can be explored. Security is one of the top challenges for the IoT world as noted in various survey literature (Al-Fuqaha et al., 2015; Atzori et al., 2010; Gubbi et al., 2013; S. Li et al., 2014; Miorandi et al., 2012; Puthal, Ranjan, Nepal, & Chen, 2018; Zeng et al., 2011). Security can be implemented as a new layer. However, other options and best practices can also be explored. Providing security options can facilitate the adoption of the architecture in the industry.

In conclusion, we have proposed a geospatial architecture in this dissertation based on the SensorThings API open standard and the Lambda architecture for addressing big data challenges. Implementation technologies, analytics use cases, and security best practices may be useful future work for this dissertation as well as guidelines for merging the SensorThings Tasking and Rules engine parts with the proposed architecture.

References

Ahlgren, B., Hidell, M., & Ngai, E. C. H. E. C.-H. (2016). Internet of Things for Smart Cities: Interoperability and Open Data. *IEEE Internet Computing*, *20*(6), 52–56. https://doi.org/10.1109/MIC.2016.124

Ahmed, E., Yaqoob, I., Hashem, I. A. T., Khan, I., Ahmed, A. I. A., Imran, M., & Vasilakos, A. V. (2017). The role of big data analytics in Internet of Things. *Computer Networks*, *129*, 459–471. https://doi.org/10.1016/j.comnet.2017.06.013

Akoka, J., Comyn-Wattiau, I., & Laoufi, N. (2017). Research on Big Data – A systematic mapping study. *Computer Standards and Interfaces*. https://doi.org/10.1016/j.csi.2017.01.004

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys and Tutorials*. https://doi.org/10.1109/COMST.2015.2444095

Alharthi, A., Krotov, V., & Bowman, M. (2017). Addressing barriers to big data. *Business Horizons*. https://doi.org/10.1016/j.bushor.2017.01.002

Arasteh, H., Hosseinnezhad, V., Loia, V., Tommasetti, A., Troisi, O., Shafie-Khah, M., & Siano, P. (2016). Iot-based smart cities: A survey. In *EEEIC 2016 - International Conference on Environment and Electrical Engineering*. https://doi.org/10.1109/EEEIC.2016.7555867

Arlitt, M., Shah, A., Marwah, M., Bellala, G., Shah, A., Healey, J., & Vandiver, B. (2015). IoTAbench: An Internet of Things Analytics Benchmark. *Proceedings of the 6th*

ACM/SPEC International Conference on Performance Engineering. https://doi.org/10.1145/2668930.2688055

Atzori, L., Iera, A., & Morabito, G. (2010). The Internet of Things: A survey. *Computer Networks*. https://doi.org/10.1016/j.comnet.2010.05.010

Barnaghi, P., Sheth, A., & Henson, C. (2013). From data to actionable knowledge: Big data challenges in the web of things. *IEEE Intelligent Systems*. https://doi.org/10.1109/MIS.2013.142

Botts, M. E., Robin, A., Greenwood, J., & Wesloh, D. (2014). OGC® SensorML: Model and XML Encoding Standard. *OGC®*. https://doi.org/OGC 12-000

Bröring, A., Stasch, C., & Echterhoff, J. (2012). *OGC Sensor Observation Service*. *OGC Implementation Standard*. https://doi.org/OGC 12-006

Butler, H., Daly, M., Doyle, A., Gillies, S., Hagen, H., Schaub, T., & Metacarta, C. S. (2016). The GeoJSON Format. *Internet Engineering Task Force (IETF)*. Retrieved from https://tools.ietf.org/html/rfc7946

CGI Group Inc. (2017). *CGI IoT Data Management for Smart Communities With Kinota™ open source API*.

Chappell, D. (2011). Introducing OData: Data access for the web, the cloud, mobile devices, and more. *Access*, (May).

Chen, M., Mao, S., Zhang, Y., & Leung, V. C. M. (2014). *Big data: related technologies, challenges and future prospects*. SpringerBriefs in Computer Science.

Choi, Y. L., Jeon, W. S., & Yoon, S. H. (2014). Improving database system performance by applying NoSQL. *Journal of Information Processing Systems*. https://doi.org/10.3745/JIPS.04.0006

Cloudera, I. (2017). CLOUDERA ENTERPRISE The Ultimate Data Engine. Retrieved March 23, 2018, from http://kr.cloudera.com/content/dam/www/apac/resources-en/datasheets/cloudera-enterprise-datasheet.pdf

Cox, S. J. D. (2011). *Observations and Measurements - XML Implementation*. *OGC®*. https://doi.org/http://www.opengeospatial.org/

Cruz Huacarpuma, R., de Sousa Junior, R., de Holanda, M., de Oliveira Albuquerque, R., García Villalba, L., & Kim, T.-H. (2017). Distributed Data Service for Data Management in Internet of Things Middleware. *Sensors*, *17*(5), 977. https://doi.org/10.3390/s17050977

Darwish, T. S. J., & Abu Bakar, K. (2018). Fog Based Intelligent Transportation Big Data Analytics in The Internet of Vehicles Environment: Motivations, Architecture, Challenges, and Critical Issues. *IEEE Access*, *6*, 15679–15701. https://doi.org/10.1109/ACCESS.2018.2815989

DataStax Corporation. (2013). *Why NoSQL?*

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplied Data Processing on Large Clusters. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation*. https://doi.org/10.1145/1327452.1327492

Department of Homeland Security. (2018). *Next Generation First Responder Integration*

Handbook, Part 2: Engineering Design.

Ding, Z., Yang, Q., & Wu, H. (2011). Massive Heterogeneous Sensor Data Management in the Internet of Things. In *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*. https://doi.org/10.1109/iThings/CPSCom.2011.6

EDIMAX Technology Co. (2017). AirBox : Smart Air Quality Detector with PM2.5, Temperature and Humidity Sensors. Retrieved September 3, 2018, from https://www.edimax.com/edimax/merchandise/merchandise_detail/data/edimax/global/air_quality_monitoring_semioutdoor/ai-1001w_v2/

Evans, D. (2011). The Internet of Things - How the Next Evolution of the Internet is Changing Everything. *CISCO White Paper*. https://doi.org/10.1109/IEEESTD.2007.373646

Fältström, P. (2016). *Market-driven Challenges to Open Internet Standards*. *Global Commission on Internet Governance Paper Series, Centre for International Governance Innovation (CIGI), paper series no. 33*.

Fan, T., & Chen, Y. (2010). A scheme of data management in the Internet of Things. In *2010 2nd IEEE InternationalConference on Network Infrastructure and Digital Content*. https://doi.org/10.1109/ICNIDC.2010.5657908

Fan, W., & Bifet, A. (2013). Mining Big Data : Current Status , and Forecast to the Future. *ACM SIGKDD Explorations Newsletter*. https://doi.org/10.1145/2481244.2481246

Firouzi, F., Rahmani, A. M., Mankodiya, K., Badaroglu, M., Merrett, G. V., Wong, P., &

Farahani, B. (2018). Internet-of-Things and big data for smarter healthcare: From device to architecture, applications and analytics. *Future Generation Computer Systems*, *78*(2), 583–586. https://doi.org/10.1016/j.future.2017.09.016

Gantz, J., & Reinsel, D. (2011). Extracting Value from Chaos State of the Universe: An Executive Summary. *IDC IView*. https://doi.org/10.1007/s10916-016-0565-7

Gartner. (2018). Big Data.

Gómez Maureira, M. A., Oldenhof, D., & Teernstra, L. (2014). ThingSpeak – an API and Web Service for the Internet of Things. *World Wide Web*.

Grothe, M., Carton, L., Van Den Broecke, J., Volten, H., & Kieboom, R. (2016). Smart emission - Building a spatial data infrastructure for an environmental citizen sensor network. *CEUR Workshop Proceedings*, *1762*. https://doi.org/http://dx.doi.org/urn:nbn:de:0074-1762-0

Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*. https://doi.org/10.1016/j.future.2013.01.010

Guinard, D., Trifa, V., Mattern, F., & Wilde, E. (2011). From the Internet of Things to the Web of Things : Resource Oriented Architecture and Best Practices. *Architecting the Internet of Things*. https://doi.org/10.1007/978-3-642-19157-2

Handl, R., Pizzo, M., & Biamonte, M. (2014). OData JSON Format Version 4.0. *OASIS Standard*.

Hinchcliffe, D. (2006). Enterprise Web 2.0: Creating real business value with Web 2.0.

Retrieved from https://www.zdnet.com/article/creating-real-business-value-with-web-2-0/

Hortonworks. (2018). Apache TEZ. Retrieved August 29, 2018, from https://hortonworks.com/apache/tez/

Huang, C.-Y. (2013). *GeoPubSubHub: A Geospatial Publish/Subscribe Architecture for the World-Wide Sensor Web*. University of Calgary.

Huang, C. Y., & Wu, C. H. (2016a). A web service protocol realizing interoperable internet of things tasking capability. *Sensors (Switzerland)*, *16*(9). https://doi.org/10.3390/s16091395

Huang, C. Y., & Wu, C. H. (2016b). Design and implement an interoperable Internet of Things application based on an extended OGC sensorthings API Standard. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*. https://doi.org/10.5194/isprsarchives-XLI-B4-263-2016

Huang, H. H., & Liu, H. (2014). Big data machine learning and graph analytics: Current state and future challenges. *2014 IEEE International Conference on Big Data (Big Data)*. https://doi.org/10.1109/BigData.2014.7004471

Hussain, A., & Wu, W. (2018). Sustainable interoperability and data integration for the IoT-based information systems. *Proceedings - 2017 IEEE International Conference on Internet of Things, IEEE Green Computing and Communications, IEEE Cyber, Physical and Social Computing, IEEE Smart Data, IThings-GreenCom-CPSCom-SmartData 2017*, *2018–Janua*, 824–829. https://doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData.2017.126

IDG. (2015). Big Data and Analytics: The Big Picture.

International Telecommunication Union. (2012). Overview of the Internet of things. *Series Y: Global Information Infrastructure, Internet Protocol Aspects and next-Generation Networks - Frameworks and Functional Architecture Models*. https://doi.org/10.1109/ESEM.2015.7321184.3.

Jazayeri, M. A., Liang, S. H. L., & Huang, C. Y. (2015). Implementation and evaluation of four interoperable open standards for the internet of things. *Sensors (Switzerland)*. https://doi.org/10.3390/s150924343

Johnson, J. E. (2012). Big Data + Big Analytics + Big Opportunity. *Financlial Executive*. https://doi.org/10.1002/9781118562260

Khalafbeigi, T., Huang, C. Y., Liang, S., & Wang, M. (2014). A hybrid scale-out cloud-based data service for worldwide sensors. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 8505 LNCS, pp. 342–348). https://doi.org/10.1007/978-3-662-43984-5_26

Klarity. (2015). Dimensions of Big Data. Retrieved from http://www.klarity-analytics.com/2015/07/27/dimensions-of-big-data/

Kolbe, T. H., Nagel, C., Lorenz, A., Groger, G., Plumer, L., Czerwinski, A., … Hafele, K.-H. (2012). OGC® CityGML.

Kotsev, A., Pantisano, F., Schade, S., & Jirka, S. (2015). Architecture of a service-enabled sensing platform for the environment. *Sensors (Switzerland)*. https://doi.org/10.3390/s150204470

Kotsev, A., Schleidt, K., Liang, S., van der Schaaf, H., Khalafbeigi, T., Grellet, S., … Beaufils, M. (2018). Extending INSPIRE to the Internet of Things through SensorThings API. *Geosciences*. https://doi.org/10.3390/geosciences8060221

Laney, D. (2001). 3D Data Management: Controlling Data Volume, Variety and Velocity. *Application Delivery Strategies*. https://doi.org/10.1016/j.infsof.2008.09.005

LASS Community / Academia Sinica. (n.d.). LASS PM2.5 Open Data Portal. Retrieved September 3, 2018, from https://lass-net.org/

Lee, I. (2017). Big data: Dimensions, evolution, impacts, and challenges. *Business Horizons*, 293–303. https://doi.org/10.1016/j.bushor.2017.01.004

Leong, P., & Choo, W. (2014). CloudCall: Cloud Database for the Internet of Things. *CloudAsia*.

Leverenz, L. (2018). Apache Hive. Retrieved September 3, 2018, from https://cwiki.apache.org/confluence/display/Hive

Li, K.-J., Lee, J., Zlatanova, S., Kolbe, T. H., Nagel, C., & Becker, T. (2015). OGC® IndoorGML. *Open Geospatial Consortium*. https://doi.org/http://www.opengeospatial.org/

Li, S., Xu, L. Da, & Zhao, S. (2014). The internet of things: a survey. *Information Systems Frontiers*. https://doi.org/10.1007/s10796-014-9492-7

Li, T., Liu, Y., Tian, Y., Shen, S., & Mao, W. (2012). A storage solution for massive IoT data based on NoSQL. In *Proceedings - 2012 IEEE Int. Conf. on Green Computing and Communications, GreenCom 2012, Conf. on Internet of Things, iThings 2012 and Conf. on Cyber, Physical and Social Computing, CPSCom 2012*.

https://doi.org/10.1109/GreenCom.2012.18

Liang, S., Bermudez, L. E., Huang, C., Jazayeri, M., & Khalafbeigi, T. (2013). Advances on Sensor Web for Internet of Things. *American Geophysical Union*.

Liang, S., Huang, C.-Y., & Khalafbeigi, T. (2016). OGC SensorThings API Part 1: Sensing. *Open Geospatial Consortium. Implementation Standard*.

LogMeIn Inc. (2015). Xively by LogMeIn- Business Solutions fo the Intenet of Things. Retrieved from https://xively.com/

Luo, K., Saeedi, S., Badger, J., & Liang, S. (2018). Using the Internet of Things to Monitor Human and Animal Uses of Industrial Linear Features. In *International Symposium on Web and Wireless Geographical Information Systems* (pp. 85–89). Springer.

Ma, M., Wang, P., & Chu, C.-H. (2013). Data Management for Internet of Things: Challenges, Approaches and Opportunities. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. https://doi.org/10.1109/GreenCom-iThings-CPSCom.2013.199

Ma, Y., Rao, J., Hu, W., Meng, X., Han, X., Zhang, Y., … Liu, C. (2012). An efficient index for massive IOT data in cloud environment. In *Proceedings of the 21st ACM international conference on Information and knowledge management - CIKM '12*. https://doi.org/10.1145/2396761.2398587

Manogaran, G., Varatharajan, R., Lopez, D., Kumar, P. M., Sundarasekar, R., & Thota, C. (2018). A new architecture of Internet of Things and big data ecosystem for secured smart

healthcare monitoring and alerting system. *Future Generation Computer Systems*, *82*, 375–387. https://doi.org/10.1016/j.future.2017.10.045

Marjani, M., Nasaruddin, F., Gani, A., Karim, A., Hashem, I. A. T., Siddiqa, A., & Yaqoob, I. (2017). Big IoT Data Analytics: Architecture, Opportunities, and Open Research Challenges. *IEEE Access*, *5*, 5247–5261. https://doi.org/10.1109/ACCESS.2017.2689040

Marz, N., & Warren, J. (2015). *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. *Manning Publications Co.* https://doi.org/10.1073/pnas.0703993104

Meiling, S., Purnomo, D., Shiraishi, J.-A., Fischer, M., & Schmidt, T. C. (2018). MONICA in Hamburg: Towards Large-Scale IoT Deployments in a Smart City. *Proceedings of the European Conference on Networks and Communications, EuCNC*. Retrieved from http://arxiv.org/abs/1803.06854

Microsoft Azure. (2018a). Data Factory. Retrieved September 3, 2018, from https://azure.microsoft.com/en-us/services/data-factory/

Microsoft Azure. (2018b). SQL Data Warehouse. Retrieved September 3, 2018, from https://azure.microsoft.com/en-ca/services/sql-data-warehouse/

Miorandi, D., Sicari, S., De Pellegrini, F., & Chlamtac, I. (2012). Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*. https://doi.org/10.1016/j.adhoc.2012.02.016

Mishra, N., Lin, C. C., & Chang, H. T. (2015). A Cognitive Adopted Framework for IoT Big-Data Management and Knowledge Discovery Prospective. *International Journal of*

Distributed Sensor Networks. https://doi.org/10.1155/2015/718390

Moßgraber, J., Hilbring, D., Pouli, P., & Padeletti, G. (2018). A Knowledge Base for Cultural Heritage Protection against Climate Change. In *15th ESWC Conference*.

Moßgraber, J., Hilbring, D., van der Schaaf, H., Hertweck, P., Kontopoulos, E., Mitzias, P., … Kompatsiaris, I. (2018). The sensor to decision chain in crisis management. In *Universal Design of ICT in Emergency Management Proceedings of the 15th ISCRAM Conference.* Rochester, NY, USA.

mySMARTLIfe Consortium Partners. (2017). *Transition of EU cities towards a new concept of Smart Life and Economy*.

Normandeau, K. (2013). Beyond Volume, Variety and Velocity is the Issue of Big Data Veracity. *Inside Big Data*. Retrieved from https://insidebigdata.com/2013/09/12/beyond-volume-variety-velocity-issue-big-data-veracity/

Oussous, A., Benjelloun, F. Z., Ait Lahcen, A., & Belfkih, S. (2017). Big Data technologies: A survey. *Journal of King Saud University - Computer and Information Sciences*. https://doi.org/10.1016/j.jksuci.2017.06.001

Pizzo, M., Handl, R., & Zurmuehl, M. (2014). OData Version 4.0 Part 1: Protocol. *OASIS Standard*.

Pokorny, J. (2013). NoSQL databases: A step to database scalability in web environment. *International Journal of Web Information Systems*. https://doi.org/10.1108/17440081311316398

Puthal, D., Ranjan, R., Nepal, S., & Chen, J. (2018). *IoT and big data: An architecture with data flow and security issues*. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*. https://doi.org/10.1007/978-3-319-67636-4_25

Qin, Y., Sheng, Q. Z., Falkner, N. J. G., Dustdar, S., Wang, H., & Vasilakos, A. V. (2016). When things matter: A survey on data-centric internet of things. *Journal of Network and Computer Applications*. https://doi.org/10.1016/j.jnca.2015.12.016

Ray, P. P. (2017). A Survey of IoT Cloud Platforms. *Future Computing and Informatics Journal*. https://doi.org/10.1016/j.fcij.2017.02.001

Russom, P. (2011). Big data analytics. *TDWI Best Practices Report*. https://doi.org/10.1109/ICCICT.2012.6398180

SensorUp Inc. (2016). *Comparison of SensorThings API and Sensor Observation Service – Part 1*.

Sowe, S. K., Kimata, T., Dong, M., & Zettsu, K. (2014). Managing heterogeneous sensor data on a big data platform: IoT services for data-intensive science. In *Proceedings - IEEE 38th Annual International Computers, Software and Applications Conference Workshops, COMPSACW 2014*. https://doi.org/10.1109/COMPSACW.2014.52

Stolz, M. (2018). *Scaling Data Services with Pivotal GemFire*. O'REILLY.

Teixeira, J. A. B. S. (2018). *Using SensorThings API to enable a multi-platform IoT environment*. University of Porto.

Tracey, D., & Sreenan, C. (2013). A holistic architecture for the Internet of Things, sensing services and big data. In *Proceedings - 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013*. https://doi.org/10.1109/CCGrid.2013.100

Trilles, S., Calia, A., Belmonte, Ó., Torres-Sospedra, J., Montoliu, R., & Huerta, J. (2017). Deployment of an open sensorized platform in a smart city context. *Future Generation Computer Systems*, *76*, 221–233. https://doi.org/10.1016/j.future.2016.11.005

Valduriez, P. (1993). Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases*. https://doi.org/10.1007/BF01264049

van der Schaaf, H., & Herzog, R. (2015). Mapping the OGC sensorthings API onto the OpenIoT middleware. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-319-16546-2_6

Vermesan, O., & Friess, P. (2013). *The Internet of things: Converging Technologies for Smart Environments and Integrated Ecosystem*. *River Publisher*. https://doi.org/10.1038/scientificamerican1004-76

Villari, M., Celesti, A., Fazio, M., & Puliafito, A. (2015). AllJoyn Lambda: An architecture for the management of smart environments in IoT. *Proceedings of 2014 International Conference on Smart Computing Workshops, SMARTCOMP Workshops 2014*, 9–14. https://doi.org/10.1109/SMARTCOMP-W.2014.7046676

Ward, J. S., & Barker, A. (2013). Undefined By Data: A Survey of Big Data Definitions. *ArXiv Preprint ArXiv:1309.5821*.

Watson, K., Kunz, S., van der Schaaf, H., & Ubertini, F. (2018). Analysis of sensor signals for monitoring of heritage buildings. In *EGU General Assembly Conference Abstracts, Vol. 20*.

Whang, K.-Y. (2011). NoSQL vs. Parallel DBMS for Large-scale Data Management. In *DASFAA 2011 Panel on Challenges in Managing and Mining Large, Heterogeneous Data*.

Whitmore, A., Agarwal, A., & Da Xu, L. (2015). The Internet of Things—A survey of topics and trends. *Information Systems Frontiers*, *17*(2), 261–274. https://doi.org/10.1007/s10796-014-9489-2

Wikipedia. (2018a). Geohash. Retrieved September 3, 2018, from https://en.wikipedia.org/wiki/Geohash

Wikipedia. (2018b). Slippy Map Tilenames. Retrieved September 3, 2018, from http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames

Yaqoob, I., Hashem, I. A. T., Gani, A., Mokhtar, S., Ahmed, E., Anuar, N. B., & Vasilakos, A. V. (2016). Big data: From beginning to future. *International Journal of Information Management*. https://doi.org/10.1016/j.ijinfomgt.2016.07.009

Zeng, D., Guo, S., & Cheng, Z. (2011). The web of things: A survey. *Journal of Communications*. https://doi.org/10.4304/jcm.6.6.424-438