Access Control and Storage of Distributed IoT data

by

Diana Mends

Thesis submitted in partial fulfillment
of the requirements for the
Master of Computer Science degree

School of Electrical Engineering and Computer Science
Faculty of Graduate and Postdoctoral Studies
University of Ottawa

Abstract

There has been a growth of a class of databases known as the Not only SQL (NoSQL) databases in recent years. Its quick growth has been fueled by a high demand by businesses as it offers a convenient way to store data and is significantly different from our traditional relational databases. It is easy to process unstructured data, offers a cloud-friendly approach and grows through the distribution of data over lots of commodity computers. Most of these NoSQL databases are distributed in several different locations, spanning countries and are known as geo-distributed cloud datastores.

We work to customize one of these known as Cassandra. Given the size of the database and the size of applications accessing the data stored, it has been challenging to customize it to meet existing application Service Level Agreement (SLAs). We live in an era of data breaches and even though some types of information are stripped of all sensitive data, there are ways to easily identify and link it to data of real persons or government. Data saved in different countries are subject to the rules and regulations of that specific country and security measures employed to safeguard consumer data.

In this thesis, we describe mechanisms for selectively replicating data in a large scale NoSQL datastore in respect of privacy and legal regulations. We introduce an easily extensible constraint language to implement these policy constraints through the creation of a pluggable topology provider in the configuration files of Cassandra. Experiments using the modified Cassandra trunk demonstrate that our techniques work well, respect response times and improves read and write latencies.

Acknowledgements

To my husband Patrick who has supported and encouraged me throughtout this process. Cheers to my one year old son who has been with me and has offered to help me type a number of times. You are indeed worth more than 2 sons, James Ahimah Aboagye. Thank you to my father, Mr Anthony Mends who taught me the value of education and my mother, Mrs Anna Mends who has been a calm voice when things got chaotic. To my mother-in-law, Sophia Attuah, thank you for always urging me on and the words of encouragement. Thank you Kukua for your babysitting and to all my friends and family, your support was very much appreciated.

Prof Iluju Kiringa and Prof Tet Yeap, I will be eternally grateful for your support and guidance. You were my rock when I did not think this work will ever be completed.

I thank God all mighty who gave me the strength, wisdom and knowledge. His love has never failed me.

Dedication

To my God Almighty. He has been with me every step of the way! I thank Him for His strength and wisdom, I could not have come this far without Him. Amen.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The term Internet of Things (IoT) is a ubiquitous network of everyday objects known as things that is able to sense and collect information without human intervention using existing internet standards to provide services for information transfer, analytics, data mining, applications and communications [44]. This enables the collection and exchange of information between humans and machines. IoT was first related to Radio Frequency Identification (RFID) in a presentation made at Proctor & Gamble (P&G) in the context of supply chain management in 1999 by Kevin Ashton. He forsaw that RFID will lead to a complete automation of data collection and stated that we needed an internet of things i.e. a standardized way for computers to understand the real world [1]. "Things" are an abstraction used to denote devices that are equipped with embedded network connectivity at a minimum, along with sensors and (possibly) actuators to interact with the environment. Todays definition of "Things" in Internet of Things(IoT) has expanded from the initial idea of RFIDs to include other sensor nodes and actuators. RFID is a sensor-based technology used primarily to identify and track products or living organisms [46].

The prevalence of IoT is estimated by [3] to consist of 26 billion units connected to the internet by the year 2020 with an incremental revenue generation in excess of $300 billion, primarily in services. Furthermore, $1.9 trillion in global economic value will be added through sales into diverse end markets. This puts into perspective the enormous amount of data that will be generated and architectures that would have to be put in place to store, process, manage and analyze this data. It also adds to the importance of building an architecture that can support this amount of data as IoT will be an increasingly larger part of Information Technology (IT) in years to come. The International Data Corporation (IDC) notes that the data just from embedded systems i.e. sensors and physical systems capturing data from the physical universe, will constitute 10% of the digital universe by

the year 2020 (this currently stands at 2%) and represent a higher percentage of target-rich data [5].

Things are sources of enormous amounts of data that can be mined to produce information or knowledge. Data mining is the extraction of patterns and knowledge from large amounts of data for the use in decision making. Currently, different machine learning methods have been used to analyze big data and with some modification in the algorithms and parameters, these algorithms are also used on data generated from IoT devices.

This thesis proposes a cloud-based architecture specifically designed for fleet management to enable the selective replication of vehicle parameters for monitoring and performance analysis. We describe a model that applies different policy constraints for selective replication of IoT data. Fleet management is a function that allows companies relying on transportation for their business to remove or minimize the risks associated with vehicle investments. This function includes vehicle financing, vehicle insurance, vehicle maintenance, vehicle telematics (tracking and diagnosis), driver management, speed management, fuel management and health and safety management [12].

In recent years, the use of cloud services is more ubiquitous than it was a decade ago. A lot more companies are moving their IT and data management services to the cloud as it provides scalable computing power and tangible cost savings. Cloud computing technology allows access to a shared pool of scalable and configurable computing resources. This gives companies access to huge amount of computing power when it is needed and without the added cost of setting up their own data centers and paying for its management.

In the year 2000, Eric Brewer proposed the CAP theorem also called the Brewer's theorem which states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees: Consistency, Availability and Partition tolerance (CAP). All databases therefore make a choice to implement two out of these three. Structured Query Language(SQL) databases support consistency and availability with little tolerance for network partitions which makes their horizontal scaling (i.e. adding more capacity through the addition of processing and storage power to a single machine) difficult. NoSQL databases try to favour horizontal scaling by relaxing either consistency or availability. A typical example is the provision of weak or eventual consistency guarantees. By doing this, these databases become tolerant to network partitions, making it possible to add servers to the setup to increase computing power especially when the number of sensors or clients increases instead of adding more capacity to a single server[59]. In addition to horizontal scalability, NoSQL databases provide advantages of memory and horizontal index and dynamically modifying data schema which are not provided by relational

databases [43].

In our architecture, Cassandra, a NoSQL database, is hosted on some high performance servers in the cloud to support data management. It is built for big data scale and is able to persist, manage and quickly query big data. Cassandra was chosen for this architecture because it is an open source and distributed database system that can be used to store and manage massive fleet data. Cassandra's data model offers the column indexes with the performance of log-structured updates, strong support for denormalization and materialized views and powerful built-in caching. There is no single point of failure as it offers a decentralised architecture which implies no single primary replica. Cassandra also offers a relaxed consistency model known as eventual consistency which guarantees that updates will eventually be applied to all replicas in the different nodes of the system. It also makes room for some nodes not having updated information and goes on to perform a read repair [4].

Using IoT to solve fleet management challenges presents an interesting system building problem in terms of design and implementation of data storage, update and transaction models for managing the huge amount of data generated and analysis of the data to make informed decisions. Real time information relevant to the fleet presents an avenue to understand what is going on in the field, prevent accidents, understand the trend of these incidents and protect vehicle and driver insurance.

1.1 Motivation

IoT applications are well suited in various application domains like agriculture (in the area of smart farming), transportation (fleet management), housing (smart building or cities) and infrastructure (smart power and grids). It has the potential to transform each of these sectors and manage some of their risks effectively. The amount of data generated along with its characteristics brings up interesting research questions that are worth investigating. How can IoT data be replicated only in areas where the data is heavily accessed to reduce interdatacenter and forwarding bandwidth? Can an architecture be built to incorporate elements of data causality where inferences can be made based on previously stored data? How do we process the multiple input data streams that come with managing IoT data? Will data colocation work? What constraints can be applied uniquely to IoT data which are different from other types of data? Is it possible to respect legal and policy constraints governing data transmission and personally identifiable information?

Several challenges arise in the architecture definition and the actual implementation of a

working IoT system especially in fleet management and all of these make this an interesting research topic. There is also a practicality to this topic as some companies whose primary aim is not transportation are looking for ways to effectively manage their transportation needs and raise their bottom line. Thus, the IoT field is currently attracting a lot of investment from major fortune 500 technology and information systems companies like General Electric, Boeing, IBM, Amazon, Microsoft, Intel, CISCO and BlackBerry who have identified the need in building a working architecture to be used.

1.2 State of the Art

There has been a lot of recent developments in the IoT field in both the industry and academia. Many industry leaders are investing a lot of resources into the development of the sector. The concept of a data stream network for IoT is a reality as currently there is infrastructure and bandwidth that allows developers to build real time applications in a secure and reliable manner, connect devices from all over the world, generate a huge amount of data and store it in the cloud. The "loop-back" has been implemented through the use of Wi-Fi and Bluetooth sensors: commands for actions can be sent to devices in the same way data can be received and processed from devices.

It is a straight forward concept to build a network of sensors in a specific domain to collect information and based on this improve the quality of service offered. Due to the versatile nature of these sensors, IoT has been used in the development of applications in different domains such as home and industrial automation, medical aids, mobile healthcare, elderly assistance, intelligent energy management and smart grids, transportation, automotive, traffic management, smart cities and many others [62].

In the field of transportation, Wind River provides a Wind River Intelligent Device Platform which enables developers of transportation applications to jump-start development with pre-configured software components that leverage the Wind River Linux platform and incorporate security features [33]. The company describes a use case where real-time equipment tracking, inter-equipment communication, predictive maintenance, remote upgrades, fuel management, improved passenger comfort and convenience are made possible with IoT.

In the field of e-health, IoT devices have made massive improvements in the lives of elderly and disabled patients. It has provided medical monitoring through the implantation of sensing devices in and around their bodies to gather and transmit physiological data to

qualified medical personnel for analysis and action [58]. This has indeed helped to improve the quality of lives and reduce the number of medical emergencies.

Presently, RFIDs are used in many sectors as electronic barcodes. They are designed in the form of tiny microchips called tags and appended to objects. Data is stored in these tags and used to identify these objects and also store and extract useful information. An everyday example of the use of RFIDs is the retail sector where store owners tag their wares to track and also control shoplifting. They are also used in bank cards and road toll tags as an access control means [58].

In the area of smart agriculture, [14] describes how wireless sensors were installed throughout a nut plantation to measure variables such as moisture, nutrient content, health threats and temperature of the soil. These measurements are periodically sent to dedicated third party cloud servers which compute the appropriate levels of water and fertilizer that should be applied to the plantation. The results are then used to automatically drive the farm's irrigation system by delivering precise amounts of water and fertilizers to the nut groves. This IoT solution aims to reduce operating costs by improving the overall utilization of water and fertilizers needed to grow almond nuts which are notoriously water-thirsty.

There have been various initiatives done in academia. Authors in [49] developed a storage management solution called IoTMDB based on NoSQL to solve the storage and management problems of massive centralized form of IoT data. Their architecture includes describing the data objects, considering the relationships and interoperability between them and proposing a data sharing mechanism based on ontology i.e. defining a shared and common domain knowledge to go beyond syntax level interaction to semantic exchange.

[62] presents a proof-of-concept deployment of an IoT island in the city of Padova, Italy. The authors describe a framework that has been successfully applied to a number of different use cases in the context of IoT systems. The target application consists of a system for collecting environmental data and monitoring the public street lighting by means of wireless nodes, equipped with different kinds of sensors placed on street light poles and connected to the internet through a gateway unit. Environmental parameters such as carbon dioxide(CO) level, air temperature and humidity, vibrations and noise were measured. This framework provided a mechanism to check the correct operation of the public lighting system by measuring the light intensity at each post. Components used in the architecture included wireless sensor network (WSN) gateway, street light, Hyper Text Transfer Protocol-Constrained Application Protocol (HTTP-CoAP) proxy, constrained link layer technologies, database server and an operator mobile device.

[34] describes an open source hybrid cloud approach to agriculture analytics for enabling

sustainable farming practices. Their architecture known as the UCSB SmartFarm, combines data aggregation for disparate agriculture related data sensors and sources, integrates multiple analytics technologies to facilitate a wide range of data inference, prediction, and visualization. To give farmers or growers full control over the privacy, security and sharing of their own data, the authors implemented an on-farm private cloud software infrastructure. Coupled with data from external cloud sources including weather, satellite imagery and other national datasets, the authors provided an interface into which custom analytics applications can be plugged.

Using sensors in the biometric and medical fields is more commonplace now than before. [17] describes an eHealth and medical development product called MySignals which is specifically oriented to researchers, developers and makers. The MySignals device can be used to develop new eHealth web, android or iOS applications or new sensors can be added to build new medical devices. It uses sixteen sensors to measure snore, spirometer, blood pressure (BLE), SPO2(BLE), glucometer (BLE) and body scale (weight, bone mass), body fat, muscle mass, body water, visceral fat, basal metabolic rate and body mass index. Features include amongst others, cloud storage, native android and iOS apps, wifi, bluetooth and BLE radios, a graphic system to see incoming data from sensors realtime.

1.3 The Problem

In this era of advancement in IT, keeping multiple copies of the same data in different locations is important for business continuity and ensures a company's data is always safe. Companies are now moving their data to globally distributed datacenters on the cloud to be able to achieve this. However, it is not necessary to replicate all of a company's data to all locations as it comes with issues such as increase in communication costs, decrease in system performance for updates and this may violate some constraints such as legal, regulatory, compliance or privacy. In the case of Cassandra, there is an increase in forwarded reads resulting in latencies and higher communication costs when a read request is sent to a node that does not have the data stored locally. Replicating data to datacenters where it will never be assessed adds on to the cost of updating this data during write requests. In replicating data, Cassandra only adheres to the replication factor that is set at the data center. This means, it writes at least n replicas of the data but there is no way for an adminstrator to specify where these replicas should be placed.

Data stored in the cloud typically traverses many different data centers and geographic regions and may be hosted in multiple places simultaneously or be dynamically relocated

as needed. However, taking a look at for example the General Data Protection Regulation (GDPR) (Regulation (European Union(EU)) 2016/679) which regulates the export of personal data collected on EU citizens and residents outside of the EU, foreign businesses will not be able to save this data in datacenters located outside of the EU. It therefore behooves on the business to selectively replicate its data to meet this directive.

We therefore investigate a model to perform a complete selective replication in Cassandra and also develop an optimal static placement scheme with respect to policy constraints.

1.4 Solution

We consider how to implement a restricted storage and replication model for a globally distributed database. We have extended our Cassandra system, a popular NoSQL database which is a highly scalable, high-performance distributed database designed to handle large amounts of data across many servers with high availability and no single point of failure, to test our model [4].

In the production version of Cassandra one of the biggest obstacles to managing hundreds, or even thousands, of remote datacenters, is that all datacenters communicate with each other and are connected all the time through a mechanism called gossip. Our modifications to Cassandra gives the ability to customize the way gossiping is done between datacenters. This makes it possible to blacklist and whitelist datacenters for a more flexible gossip. This becomes extremely useful in settings where edge datacenters are supposed to collect information and send to a central datacenter for processing and do not need to communicate with other edge datacenters forming a hub-and-spoke database connectivity; or in instances where due to legal requirements data originating from certain locations are not permitted to be stored in some other locations.

We have designed a constraint language that allows Cassandra administrators to specify policy constraints of how each datacenter should interact with other datacenters. These constraints are specified in a file and forms part of the Cassandra configuration. In designing this language, we have made it extensible so that new constraint parameters can be added in the future without breaking our system. This makes our approach novel as constraint language can be extended and is "future proof".

1.5 Outline

The thesis is organised into 5 chapters. Chapter 2 presents a detailed overview of NoSQL databases, management of IoT data and different architectures. The different components of the architecture are discussed to give the reader an in-depth insight into the mechanisms of the fleet management system adapted for IoT. In chapter 3, we talk about the access control and describe our model for optimal selective replication. We also describe the language and the algorithms used to implement the policy constraints specified by the administrator. In Chapter 4, we present a summary of implementation and the experimental setup. In Chapter 5, we conclude by summarizing the contributions made in this research paper and areas of application of our work in both academia and the industry. We also touch on future areas of research, the issues associated with our work and the way forward.

Chapter 2

Background and Related Work

Various methods have been proposed for storing data generated by heterogeneous IoT devices in the cloud. Some of these architectures cover specific issues like security and privacy, platforms for developing IoT applications for the cloud, data collection and retrieval, data storage and data mining and analytics. A few of the implemented architectures will be discussed.

We start with the work in [51] which addresses the suitability of different types of databases for storing and accessing IoT data in the cloud. They consider the performance of MySQL (a Relational Database Management System) and MongoDB, CouchDB and Redis (all three are NoSql databases) in a cloud environment by using different types of IoT data namely sensor readings and multimedia. In summary, the MongoDB performed well in write-intensive systems for the use of bulk inserts followed by MySQl, CouchDB and Redis. For multimedia data, MySQL using blob storage performed better than MongoDBs GridFS when it came to inserting multimedia files. In this regard, choosing the right kind of database to store IoT data depends on the type of data being stored and speed in retrieval.

As noted, query performance is usually affected by the size of each data item, total size of data in the database and the number of concurrent querying threads. Impact of file size on read latency between MySQL and MongoDB was not significant. MongoDB was slightly faster when serving more clients simultaneously and its GridFS made it easier to shard the database across several machines. The authors however concluded that it was hard to specifically determine the best cloud database for IoT since the data types vary and the scope of relevant use cases is vast. Again since each type of database has its own pros and cons as well as area of application, different databases do better or worse in different application domains. A choice is made based on the properties and requirements of the

specific system. For the scenarios they studied however, there was more potential in using NoSQL databases than traditional relational database systems.

[39] proposes a cloud based IoT data gathering and processing platform. The authors recognise that little effort has been focused on Cloud-based interoperable smart device data gathering and processing to enable informed decision making. Their architecture provides a standard and open mechanism for collecting diverse remote sensor data and processing them irrespective of the sensor device or usage platform. Their design was driven by ease of use, efficient communication, portability, openness and interoperability. Their platform architecture provides dual interactions, data gathering and control of actuators. Sensors and actuators are directly attached to a server or middleware layer through which they receive configuration information, deliver monitored data and forward monitored data from the sensors to their interface and receive control information to be passed on to the actuators. The data is then stored in the cloud to make it public for applications to use or use the analysed data results to derive control decisions for actuators.

[55] presents a data processing framework based on MapReduce processing. It enables data at IoTs to be processed close to the sources of data i.e. at the sensor nodes and on the devices to reduce the amount of data transmission at IoTs, implying that less data is processed in the cloud. Data processing is deployed at the nodes that has the target data and the results are aggregated rather than transmitted in its raw form to the servers on the cloud. Their implementation of MapReduce defines the management system and data processing tasks as mobile agents and map and reduce processing in MapReduce are provided by migrating mobile agents with results of their processing.

[45] implements a selective replication mechanism for PNUTS system, a globally replicated, scalable web database that is used in production at Yahoo. The goal of their mechanism was to minimize replication cost taking into consideration cross data center replication and optimization of bandwidth usage using a dynamic placement method which dynamically migrate replicas away from locations where they are not often accessed to locations where the read rate is higher, subject to any specified constraints. Their mechanism reduces the read rate as data is closer to where they are accessed. They designed a constraint language that allows web developers to specify policy constraints (such as legal constraints or minimum replication levels) as a function of the content of the record.

2.1 Preliminaries

A distributed system is a collection of independent computing processes or processors often referred to as nodes that communicate with each other through a communication network using message passing. The processes on nodes do not share any memory, have independent failure nodes, and share no common clock. The system may be divided into several sub-partitions where nodes in a single partition can communicate with each other but there is usually no communication occurring across partitions [41]. A distributed database is a collection of multiple databases that are usually interconnected and could be spread physically across multiple locations and like a distributed system, communicate via a communication network. A distributed database management system (DDBMS) is simply a software system that manages a distributed database as if all the data was stored in one single location. It synchronizes all the data periodically and ensures that all updates and deletions are applied at all the multiple locations so that users have the most up to date information as stored in the databases.

Cloud computing currently offers a pay-as-you-go or pay-per-use pricing model where an application or service can reach a global user community and the cloud infrastructure dynamically provisions or removes servers depending on the load [41]. The infrastructure risks of managing the network of remote servers that stores, manages and process application data is thus transferred to the cloud service providers (CSP) and gives application developers the focus on their application itself.

Spatial-temporal attributes are intrinsic for IoT data and as explained in [38], in an IoT system, every sampling value corresponds to a location and a time instant describing where and when the value is sampled. For objects or sensors that change location, i.e. can move for example GPS sensors and video-based traffic sensors, their location is dynamic.

EPC (Electronic Product Code) is designed as a universal identifier that provides a unique identity for every physical object anywhere in the word for all time. The canonical representation of an EPC is a URI, namely the 'pure-identity URI' representation that is intended for use when referring to a specific physical object in communication about EPCs among information systems and business application software. It was designed by Massachusetts Institute of Technology's Auto-ID center [11].

Sensors provide information about the physical entity they monitor. This information ranges from the identity of the physical entity to measures of the physical state of the physical entity. Sensors can be attached or embedded in the physical structure of the physical entity or placed in the environment and indirectly monitor entities [32].

Actuators can modify the physical state of a physical entity like changing the state (translate, rotate, stir, inflate, switch on or off) of physical entities or activating or deactivating functionalities of more complex ones [32].

A multi master replication strategy is used where write transactions accessing a partition or node can execute on independent replicas each acting as the master. This offers a weaker consistency guarantee. A primary copy replication executes update transactions at a primary replica; the updates called the downstream updates are then replicated to the secondaries in the order they were executed at the primary [27].

ACID stands for Atomicity, Consistency, Isolation, Durability. Transaction execution must be atomic, meaning either all of its operations are executed or none at all. There is also no interference among transactions. Consistency implies that each transaction is a consistent unit of execution. Transactions are also isolated from the side-effects of other transactions. Lastly durability means the effects of transactions are persistent forever [27].

2.2 Cloud Data Management

Cloud data management has emerged in recent years as a more efficient way to manage massive datasets in different geographical regions. It gives access to a pool of computing network resources and gives the application developer as little infrastructure management as possible. The benefits of using a cloud computing model includes a lower up-front investment, lower operating costs, higher scalability, elasticity, easy access through the web and reduced business risks and maintenance expenses [41]. Amazon [22] defines cloud computing as an on-demand delivery of compute power, database storage, applications and other IT resources through a cloud services platform via the internet with a pay-as-you-go pricing. The pay-as-you-go pricing model allows customers to pay for the use of computing resources on a short-term basis as needed. For example, processors by the hour and storage by the day and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful [28].

Storing data in the cloud comes with its own specifications that should be met. A cloud data management system needs to have:

- Scalability and high performance. Applications are currently dealing with a lot of data. This data needs to be collected, managed, stored and retrieved. The cloud data management system should first be able to handle the growing amount of data and secondly have the ability to accommodate further growth.

- Elasticity. In cloud computing, elasticity is defined as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner such that at each point in time the available resources match the current demand as closely as possible [10]. Cloud applications are usually subjected to enormous fluctuations in their access patterns.

- Heterogeneous servers. Most cloud environments run on commodity heterogeneous servers. Commodity servers are usually low-end servers that are disposed of instead of repaired.

- Fault tolerance. Since commodity servers are low end servers, they are more prone to failures as compared to high-end servers but a cloud data management system must ensure that systems continue to run normally in the midst of failures.

- Security and privacy features. Currently, there are third party companies that offer the cloud computing services. This means one cloud computing company may handle data from different tenants on the same premises and security and privacy is extremely important. Each tenant should only be allowed to access their data and the system should be set up in such a way that tenants are not aware of other tenants or their data.

- Availability. With critical applications being run from the cloud, it is important that services are always available with very limited periods of downtime [41].

There are three main models in cloud computing. These are Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS contains the basic building blocks for cloud IT and typically provide access to networking features, virtual or dedicated hardware for computing resources and data storage space. With storage provided by the IaaS service, the user of this service simply handles the data which is made universally accessible over the internet. IaaS provides the highest level of flexibility and management control over IT resources. PaaS provides a platform to build applications and services thereby removing the need for organizations to manage the underlying infrastructure and allow them to focus on the deployment and management of their applications. SaaS provides the user with a completed product that is run and managed by the service provider. User does not manage the service or the underlying infrastructure but only uses the particular piece of software by connecting via the internet [22, 24]

On-demand cloud computing services are being offered by companies like Amazon, General Electric and Salesforce. Amazon Web Services (AWS) offers a broad set of global cloud-based products including compute, storage, databases, analytics, networking, mobile, developer tools, management tools, IoT, security and enterprise applications. It allows subscribers to have at their disposal a full-fledged virtual cluster of computers available 24 hours a day, 7 days a week and 365 days in a year through the internet. AWS' version of virtual computers have most of the attributes of a real computer including hardware (CPU(s) and GPU(s) for processing, local/RAM memory, hard disk or SSD storage); a choice of operating systems; networking; and pre-loaded application software such as web servers, databases and customer relationship management (CRM) applications. There is a virtualization of input output devices such as the keyboard, display and mouse and allows subscribers to login and configure or setup using a browser [15]. Experimental setup for our work was done on AWS.

General Electric's(GE) Predix Cloud was designed specifically for industrial data and analytics to capture and analyze the unique volume, velocity and variety of machine data within a highly secure, industrial strength cloud environment. A use case involving predix cloud is using sensors to prevent costly flight delays due to problems with aircraft landing gear. Thirty-four sensors such as the hydraulic pressure and brake temperature sensors were used to detect pressure and temperature wear on brakes. The data generated was analyzed and used to create a cyber model of each aircraft's physical landing gear. Using this model, current landing gear issues can be diagnosed and the remaining useful life can be predicted based on historical data [7, 18].

Salesforce is a company also offering cloud computing services. They have gone beyond simply offering shared infrastructure and have added the power of artificial intelligence to build predictive features into customer applications, anticipate next steps and automate tasks [20].

2.2.1 Architecture of NoSQL databases

NoSQL databases are categorized according to the way they store data and are divided into four: Key Value Stores, Document Stores, Column Family Stores and Graph Databases.

The NoSql data models along with some characteristics are summarized in the figure below:

Making references from [42, 41], "Key Value Stores are similar to maps or dictionaries where data is addressed by a unique key and the keys are the only way to retrieve stored data. Values are opaque to the data store as they are seen as uninterpreted byte arrays. A value

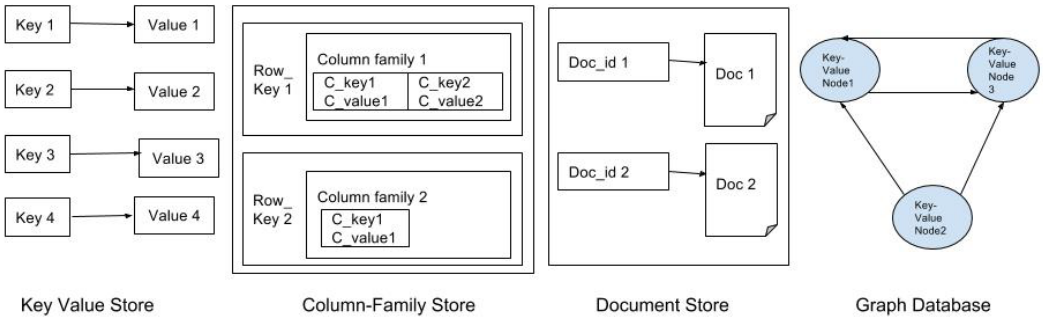


Figure 2.1: NoSQL data models [27]

can be any arbitrary data including an integer, a string, an array or an object. This means that these data stores cannot handle data level querying and indexing and can perform queries only through keys. Each value is isolated and independent from each other and relationships between them must be handled in the application logic. Key value stores are completely schema free and new values associated with their keys can be added at runtime without conflicting any other stored data and without influencing system availability. Key value pairs are grouped into a collection. Key value stores are very efficient in storing distributed data but are not suitable for scenarios requiring relations or structures. Any functionality requiring relations or structures or both must be implemented in the client application interacting with the key value store. Key value stores can be further classified as in-memory key value stores which keep the data in memory. Examples of this are Redis, Memcached and Membase and persistent key-value stores which maintain data on disk such as BerkeleyDB, Voldemort and Riak.

Document Stores encapsulate key value pairs in JSON or JSON (JavaScript Object Notation) like documents. Every document contains a unique key "ID" which is also unique within a collection of documents and therefore identifies a document explicitly. In contrast to key value stores, values are not opaque and can be queried making handling of complex data structures like nested objects very convenient. Some more advantages of using a document store are support of data types, no schema restrictions and ability to add new attributes to existing documents at runtime. Each document within a document store can have a different structure. Document stores provide the capability of indexing documents based on primary key as well as on the contents of the documents. Like key-value stores, they are inefficient in multiple-key transactions involving cross-document operations. Typical examples include CouchDB, MongoDB (uses Binary JSON) and Couchbase server.

Column Family stores were inspired by Google Bigtable which is a distributed storage

system for managing structured data that is designed to scale to a very large size. They are also known as column oriented stores, extensible record stores and wide columnar stores. In Bigtable, the dataset consists of several rows, each of which is addressed by a unique row key known as a primary key. Each row is composed of a set of column families and different rows can have different column families. Similar to key value stores, the row key resembles the key and the set of column families resembles the value represented by the row key. However, each column family further acts as a key for the one or more columns that it holds where each column consists of a name-value pair. This data model was built for high throughput and latency-sensitive data serving. It is described as sparse, distributed and persistent multidimensional sorted map. In this map, an arbitrary number of key value pairs can be stored within rows. Like key value stores, values cannot be interpreted by the system and relationships between datasets and any other data types than strings are not supported natively. Thus these features have to be implemented in the application logic. In general, column family stores provide more powerful indexing and querying than key value stores because they are based on column families and columns in addition to row keys. Column family stores support data versioning, grouping of columns into column families and data partitioning. Examples of column family implementations are HBase, Hypertable and Cassandra (with the inclusion of another dimension called super columns).

Lastly Graph Databases has as its specialty, efficient management of heavily linked or highly interconnected data. It is able to effectively store data that has many relationships and uses efficient traversals between different entities. They originated from graph theory and use graphs as their data model. A graph is a mathematical concept used to represent a set of objects known as vertices or nodes and the links or edge that interconnect these vertices. Nodes and edges consist of objects with embedded key value pairs. The range of keys and values can be defined in a schema. They are suitable in scenarios such as social networking applications, pattern recognition, dependency analysis, recommendation systems and solving path finding problems raised in navigation systems. Graph databases have an upper hand in handling graphs and relationships but are not efficient as other NoSQL data stores in other scenarios. Examples are Neo4J (fully ACID-compliant) and GraphDB which are both based on directed and multi relational property graphs."

2.2.2 Comparision of NoSQL databases

In the figure below, some comparisons are made between some NoSQL databases:

In this section, we discuss four of the popular types of NoSQL databases showing how data

NoSQL Data Model	NoSQL DB	Querying	Map reduce	Partitioning	Replication	Consistency
Key-Value stores	Redis	Does not provide SQL like querying	No	Not available	Master-Slave, asynchronous replication	Eventual consistency. Strong consistency if slave replicas are solely for failover
	Voldemort	No	Yes	Consistent hashing	Masterless, asynchronous replication	Configuration, based on quorum read and write requests
Column family Store	Cassandra	Cassandra query language	Yes	Consistent hashing and range partitioning	Masterless, asynchronous replication	Eventual consistency
	DynamoDB	Proprietary	Amazon Elastic MapReduce	Consistent hashing	Three-way replication across multiple zones in a region	Configurable
Document Stores	MongoDB	Proprietary	Yes	Range partitioning based on a shared key	Master-slave, asynchronous replication	Configurable
	CouchDB	UnQL (SQL like)	Yes	Consistent hashing	Multi-master, asynchronous replication	Eventual consistency
Graph Database	Neo4J	Cypher, Gremlin, SparQL	No	No partitioning (cache sharding only)	Master-slave, but can handle write requests on all server nodes	Eventual consistency
	HyperGraphDB	SQL like querying	No	Graph parts can reside in different P2P nodes	Multi-master, asynchronous replication. Extensible Messaging and Presence Protocol (XMPP)	Eventual consistency

Figure 2.2: NoSQL database comparisons [27]

is managed in the cloud. We will specifically look at BigTable, Cassandra, Dynamo and MongoDB.

Bigtable Making references from [35, 27], "Bigtable is a distributed storage system for managing structured data that is designed to scale to petabytes of data across thousands of commodity servers. It was designed by Google to support its crawl and indexing infrastructure and is used by numerous google products and projects including Google Analytics, Google Finance, Orkut, Personalized search, Writely and Google Earth. it provides wide applicability, scalability, high performance and high availability. Bigtable does not support a full relational data model; instead it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage.

The Bigtable data models deals with clusters. Each cluster stores a number of tables and data is organised into three dimensions: rows, columns and timestamps. A cluster here is a set of processes that run the Bigtable software and it serves a set of tables. A table is a sparse, distributed, persistent multidimensional sorted map. It is logically

represented as a key range and physically represented as a set of SSTables. Rows with consecutive keys are grouped together into a tablet which form the unit of load balancing and distribution while columns grouped into column families form the unit of access control and resource accounting. A row is the unit of transactional consistency in Bigtable and does not currently support transactions across rows. This results in reads of short row ranges being efficient and typically requiring communication with only a small number of machines. Each tablet server has unique read-write access to a given tablet. Data from the tables are persistently stored in the Google File System (GFS) that provides the abstraction of a scalable, consistent, fault tolerant storage. Multiple versions of the same data are indexed by timestamps which are 64-bit integers. These can be assigned implicitly by Bigtable or by client applications. A garbage-collection mechanism is implemented where the client can indicate which versioned data to be garbage collected.

Bigtable API is available for creating and deleting tables and column families and changing cluster, table and column family metadata such as access control rights. In addition to the API, there are several other features that allow the user to manipulate data and these include support for single-row transactions to perform atomic read-modify-write sequences on data stored under a single row key, allowing cells to be used as integer counters and the execution of client-supplied scripts in the address spaces of the servers. The structure of Bigtable includes the Google File System (GFS) to store log and data files, the Google SSTable immutable-file format to internally store data files and a Chubby Lock service which is a highly available and persistent distributed lock service using the Paxos algorithm for replica consistency in the face of failures. There is no replication of user data inside Bigtable, all replication is handled by the underlying GFS layer. A chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests.

The Bigtable implementation has three major components: a library that is linked into every client, one master server and many tablet servers. Clients communicate directly with tablet servers for reads and writes and usually never communicate with the master. The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, garbage collecting files in GFS and handling schema changes such as table and column family creations and deletions. Each tablet server manages a set of tablets which includes read and write requests to the tablets that it has loaded and also splits tablets that have grown too large.

We will now concentrate on details of the tablet as it pertains to read and writing of data in the Bigtable structure. The GFS plays the role of storing the persistent state of a tablet. Updates are committed to a commit log that stores redo records will the memtable stores

recently committed updates. A memtable maintains the updates on row-by-row basis where each row is copy-on-write to maintain row-level consistency. Older updates are however stored in a sequence of SSTables which are immutable. When a write operation arrives at a tablet server, the server checks that is well formed and that the sender is authorized to perform the mutation. This authorization is done by reading the list of permitted writers from a chubby file. If this write is valid, an update is made to the commit log.

Similarly, read requests arriving at the tablet server is checked for well-formedness and proper authorization. A valid read operation is executed on a merged view of the sequence of SSTables and memtable. Incoming reads and writes are not affected while the tablets are being split, merged or compacted. Compaction is in relation to the conversion of a frozen memtable to an SSTable. As is expected, when write operations execute, the size of the memtable increases. When it reaches a threshold, the memtable is frozen and converted to a new SSTable and written to GFS. This shrinks the memory usage of the tablet server and reduces the amount of data that has to be read from the commit log during recovery in the case of a server failure. The process described above is known as a minor compaction. A major compaction involves executing a merging compaction which reads the contents of a few SSTables and the memtable and writes out a new SSTable.

The concept of replication of data is implemented through the GFS. When files are written, the GFS attempts to place one replica of the data on the same machine as the writer. When GFS files are read, the reads are served from the nearest available replica. Chubby is used to manage and store schemas. It provides atomic whole-files writes and consistent caching of small files. Tablet servers simply read the appropriate schema file from Chubby to determine what column families exist. There has been a number of refinements to achieve the current performance, availability and reliability of the Bigtable database. Some of these include the use of locality groups, compression, caching for read performance, bloom filters, commit-log implementation, speeding up tablet recovery and exploitation of immutability.

We move on to the Amazon side of things with Dynamo, Amazon's highly available key-value store [37]. With Amazon running a world-wide e-commerce platform serving tens of millions of customers at peak times using tens of thousands of servers located in many data centers around the world, a reliable, highly scalable, efficient database is a must. Failure of some components is seen as normal and not expected to impact availability and performance.

In terms of the CAP theorem, Dynamo sacrifices consistency for availability and partition tolerance. It was developed by Amazon to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability,

consistency, cost-effectiveness and performance. The architecture consists of partitioned and replicated data using consistent hashing, facilitation of consistency through the use of object versioning and the use of a quorum-like technique and a decentralised replica synchronization protocol to maintain consistency among replicas during updates. Further, a gossip based distributed failure detection and membership protocol is employed. As mentioned by the authors, Dynamo is a completely decentralised system with minimal need for manual administration. It has been designed to be an eventually consistent data store; that is all updates reach all replicas eventually.

There are four main characteristics of Dynamo that is worth mentioning. First, it is targeted at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. secondly, it is for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. This implies that it does not focus on the problem of data integrity and security and is built for a trusted environment. Thirdly, applications that use Dynamo do not require support for hierarchical namespaces or complex relational schema. Finally, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write applications to be performed within a few hundred milliseconds. This latency requirement, was achieved a zero-hop DHT where each node maintains enough routing information locally to route a request to the appropriate node directly. We will go on further to talk about the partitioning algorithm, replication, data versioning, membership and failure handling and scaling before we close the chapter on Dynamo.

Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. It consistent hashing, the output range of a hash function is treated as a fixed circular space or ring. In this case, the largest hash value wraps around the smallest hash value. It implements a variant of consistent hashing where each node gets assigned to multiple points in the ring. It uses the concept of virtual nodes where a virtual node looks like a single node in the system but each node can be responsible for more than one virtual nodes. This offers a number of advantages including load being evenly dispersed across remaining available nodes when a node becomes unavailable; a newly available node or a new node, being added to the system is able to accept a roughly equivalent amount of load from each of the other available nodes; and lastly, the number of virtual nodes that a node is responsible for can be decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

Replication is very important to maintain high availability and durability. In Dynamo, data is replicated on multiple hosts. Each data item is replicated at N hosts where N is a

parameter configured "per-instance". Each key, k, is assigned to a coordinator node which is charge of replication of the data items that fall within its range, storing each key within its range and replicating these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its Nth predecessor. There is the existence of a preference list which stores the list of nodes that is responsible for storing a particular key. A node handling a read or write operation is known as the coordinator and is typically the first among the top N nodes in the preference list.

To maintain consistency among its replicas, Dynamo uses a consistency protocol that has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that R + W ¿ N yields a quorum-like system. The latency of a get or put operation is dictated by the slowest of the R or W replicas. Thus R and W are usually configured to be less than N to provide better latency.

In order to describe data versioning, the basic explanation get() and put() calls must be stated. The get(key) operation locates the object replicas associated with the key in the storage system and returns a single object or a list of objects with conflicting versions along with a context. The put(key, context, object) operation determines where the replicas of the object should be placed based on the associated key and writes the replicas to disk. The context encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object that is being updated. With Dynamo providing eventual consistency, a put() call may return to its caller before the update has been applied to all the replicas meaning that there can be scenarios where a subsequent get() operation may return an object that does not have the latest updates.

In the context of data versioning, dynamo treats the result of each modification or update as a new and immutable version of the data and therefore allows for multiple versions of an object to be present in the system at the same time. The system does a syntactic reconciliation where it determines the authoritative version. There is also semantic reconciliation which occurs during version branching in the presence of failures combined with concurrent updates resulting in conflicting versions of the object. Here, the client must perform a reconciliation upon read in order to collapse multiple branches of data evolution back into one. Dynamo uses vector clocks in order to capture causality between different versions of the same object. A vector clock is a list of (node, counter) pairs and one is associated with every version of every object. One can determine whether two versions

of an object are on parallel branches or have a causal ordering by examining their vector clocks.

Dynamo implements a "sloppy quorum" in which all read and write operations are performed on the first N healthy nodes from the preference list. This may not always be the first N nodes encountered while walking the consistent hashing ring. Using an example to explain this further, if node A is temporarily down or unreachable during a write operation, then a replica that resides on node A will be sent to another node which is available, e.g. node D. This is done to maintain the desired availability and durability guarantees. The replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica. Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that node A has recovered, node D will attempt to deliver the replica to A. Once the transfer succeeds, node D may delete the object from its local store without decreasing the total number of replicas in the system. This explains the concept of hinted handoffs where neither read or write operations are not failed due to temporary node or network failures. There is however a problem with hinted handoffs where a node that is offline for a long time while have lots of hints being built up on other nodes. When the failed node comes back up, there tends to be a flood of requests from other nodes when it's just about starting up and preparing itself to be fully functional. As a solution, it is now possible to disable hinted handoffs entirely or as a less extreme measure, reduce the priority of hinted handoff messages against new write requests.

Dynamo uses an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. An administrator uses a commandline tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership.

Dynamo uses a failure detection method to avoid attempts to communicate with unreachable peers during get() or put() operations, and when transferring partitions and hinted replicas. Node B is deemed unresponsive by node A if node B fails to respond to node A's messages. Node A then uses alternate nodes to service requests that map to node B's partitions.

The primary advantage of Dynamo is that it provides the necessary knobs using the three parameters of (N, R, W) to tune their instance based on the client applications needs to achieve desired levels of performance, availability and durability. It provides an eventually

consistent database and is built to handle different node failure modes and inconsistencies. It adopts a full membership model where each node has the data hosted by its peers through the gossip mechanism.

We go on to give a brief summary of the architecture of MongoDB using [52, 50] as references. It is a document-orientated database written in the C++ programming language and used for storing content of all types and optimized for speed and scalability. It uses an open data format called Binary-JSON(BSON) to make it easier for a computer to process and search documents and ability to handle binary data. BSON is much easier to traverse and index quickly and convert to a programming language's native data format.

A document represents the unit of storage, its made up of any number of keys and values and has a unique identifier. Thus a key can be used to reference pieces of data inside a document. MongoDB offers a full index support, replication and high availability, auto-sharding enabling horizontal scaling without comprising functionality, Map/Reduce for flexible aggregation and data processing and GridFS for storing files of any size without complicating the users stack.

A collection is a container that stores documents. A collection can be thought of as the schema-free equivalent of a table. Every collection in MongoDB has a unique name which should begin with a letter or optionally an underscore when created using the createCollection function. A single database has a default limit of 24,000 namespaces per database. Each collection accounts for at least two namespaces: one for the collection itself and the other for the first index created in the collection. Each document which consists of key value pairs can be made up of string, integer, boolean, double, min or max keys, arrays, timestamp, object, null, symbol, object id, binary data, regular expression and javascript code. Information is either embedded into a document i.e. placing a certain type of data into the document itself or referenced in another document i.e. creating a reference to another document that contains the specific data. The _id key is a unique identifier which is added to each new document created.

MongoDB uses indexing which is a data structure that collects information about the values of specified fields in the documents of a collection. This data structure is used by MongoDB's query optimizer to quickly sort through and order the documents in a collection. MongoDB automatically creates the database and the underlying collection for a user when they store data in it. A single instance of MongoDB can host multiple independent databases, each of which can have its own collections and permissions. MongoDB uses the concept of auto-sharding where data is split up across machines and rebalanced automatically making it possible to store more data and handle more load without the need

for larger or more powerful machines. MongDB sharding provides automatic balancing for changes in load and data distribution, easy addition of new machines without down time, no single points of failure and automatic failover.

Finally, MongoDB offers the GridFs which is a simple specification used by all of the supported drivers on MongoDB to store large files and enables users to access parts of the file without retrieving the entire file. This is done while maintaining an extremely high performance.

Lastly, we give a brief summary of Cassandra's data model making reference from [**?**]. Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunably consistent, schema-free, column-oriented database that bases its distribution design on Amazons Dynamo and its data model on Googles Bigtable. Cassandra is optimized for excellent throughput on writes.

Cassandra defines a column family to be a container for an ordered collection of rows each of which is itself an ordered collection of columns. It is analogous to a table in relational databases. The basic Cassandra data structure contains a column and a column family. A column is a name/value pair (and a client-supplied timestamp of when it was last updated). Cassandra column families is similar to a four-dimensional hash: [Keyspace][ColumnFamily][Key][Column]. Cassandra consists of super column families which holds named groups of columns known as sub columns where sub columns are named groups of columns, cluster which is where nodes containing data are arranged in a ring, a node holding a replica for different ranges of data, keyspace which is the outermost container for data in Cassandra. Cassandra requires the database administrator to define an outer container called a keyspace that contains column families. The keyspace is a logical namespace to hold column families and certain configuration properties.

As mentioned earlier, Cassandra is a distributed database where data is distributed over several machines operating together as a single instance to the end user. The outermost structure in Cassandra is the cluster which is sometimes called the ring because Cassandra assigns data to nodes in the cluster by arranging them in a ring. A node holds a replica for different ranges of data. The peer-to-peer protocol used in Cassandra allows the data to replicate across nodes in a manner that is transparent to the user and the replication factor is the number of machines in the cluster that will receive copies of the same data. This implies that when a node goes down, all data is not lost as a replica can respond to queries.

In contrast to relational databases where null has to be stored for every column, Cassandra simply does not show null in place of those values, thus saving space. In designing how

data is to be stored in Cassandra, the designer or developer starts with the query model. The question here is what queries will the application need because the queries that will be used will determine the column families or the super column families that the database will have."

2.3 Cassandra

As Cassandra is the database used in this IoT project, we will now go into detail on the data model and some specifics.

Taking some more references from [29, 4], "Cassandra provides a structured key-value store with tunable consistency. Keys map to multiple values, which are grouped into column families. Cassandra provides the flexibility of adding additional columns to specific keys in a column family after the column families have been created. The values from a column family for each key are stored together, making Cassandra a hybrid data management system between a column-oriented DBMS and a row-oriented store.

Every node in the cluster has the same role and there is no single point of failure, giving Cassandra its decentralized feature. Replication strategies are configurable and has been built for multiple-data center deployment, redundancy, failover and disaster recovery. Read and write throughput both increase linearly as machines are added with no downtime or interruption to applications. Cassandra has Hadoop integration with MapReduce, Apache Pig and Apache Hive support.

Cassandra's internal design consists of an internal keyspace called system which is used to store metadata about the cluster to aid in operations and cannot be modified. This metadata includes the node's token, the cluster name, keyspace and schema definitions to support dynamic loading, migration data and details on a node being bootstrapped. Schema definitions are stored in two column families namely the schema and migrations column families. The schema column family holds user keyspace and schema definitions while the migration column family records changes made to a keyspace. One key advantage of using Cassandra is its assurance of availability. It implements this through a peer-to-peer design meaning any given node is structurally identical to any other node. The user states the replication strategy and how many nodes data is to be replicated on. This means that data is available on multiple nodes and data on a failed node will still be available for reads and writes.

The gossip protocol sometimes called the epidemic protocols is used for intra-ring communication so that each node can have state information about other nodes. When a new

node is added to the cluster, it takes time to learn the topology of the ring and accept data that it may also be responsible for. The gossiper runs every second on a timer and triggers the hinted handoffs. When a write request is sent to Cassandra but the node where this write belongs is not available due to whatever reason, the node that "temporarily" keeps this write will create a hint which basically contains information about the write request and the intended node. When the intended node is up, the write request is sent to it. This ensures Cassandra is always available for writes and reduces the time that a failed node will be inconsistent after it comes back online. The gossip protocol is primarily implemented by the org.apache.Cassandra.gms.Gossiper class which is responsible for managing gossip for the local node. When a server node is started, it registers itself with the gossiper to receive end point state information. At the beginning, the gossiper will choose a random node in the ring and initialize a gossip session with it. Each round of gossip requires three messages. The gossip initiator then sends its chosen node a GossipDigestAckMessage. When the initiator receives the ack message from the node, it sends the node a GossipDIgestAck2Message to complete the round of gossip. When the gossiper determines that an endpoint is dead, it "convicts" that end-point by marking it as dead in its local list and logging that fact.

Cassandra uses the Phi Accrual Failure Detection algorithm to detect failure. Accrual failure detection is based on two primary ideas. The first general idea is that failure detection should be flexible which is achieved by decoupling it from the application being monitored. The second idea determines whether a node is dead or not based on whether a heartbeat is received from the node or not. This algorithm implements a suspicion level. This outputs a continuous level of "suspicion" regarding how confident it is that a node has failed.

Anti-entropy, implemented by Cassandra is based on an epidemic theory of computing. It compares all replicas of each piece of data that exist (or are supposed to) and updates each replica to the newest version [2]. This is done using Merkle trees. A merkle tree is a hash tree where leaves are hashes of the values of individual keys. In Cassandra, Merkle tree is a hash representing the data in that column family. If the trees from the different nodes do not match, they have to be reconciled or repaired in order to determine the latest data values they should all be set to. Each column family has its own Merkle tree which is created during a major compaction operation and is kept only as long as is required to send it to the neighboring nodes on the ring. This means that excess data might be sent across the network but it saves local IO and is preferable for very large datasets. Compaction is the process of freeing up space by merging large accumulated data files. After each write update, the anti entropy algorithm performs a checksum against the database and

compares checksums of peers. If the checksums differ, then the data is exchanged. To ensure that this process is relevant, a time window is set to ensure the peers have had a chance to receive the most recent update. To keep the operation fast, nodes internally maintain an inverted index keyed by timestamp and only exchange the most recent updates. Cassandra's implementation of anti entropy is modeled on Dynamos with modifications to support the richer data model.

Cassandra is built to work in multiple nodes and the user is allowed to select the replication strategy. In for example implementing eventual consistency, it takes a bit of time for all the replicas to be updated with a write update. To read data, a client connects to any node in the cluster and based on the consistency level specified by the client, a number of nodes is read. If some reason, some replicas contain some out-of-date value, Cassandra will first return the most recent value to the client and then proceeds to perform a read repair in the background. This will bring the replicas with stale values up to date.

There are two broad consistency levels that can be defined in Cassandra [9]. These are the read and write consistency. In write consistency there are the all, each_quorum, quorum. local_quorum, one, two, three, local_one, any, serial and local_serial. The all consistency level provides the highest consistency and the lowest availability of any other level. In this consistency level, a write must be written to the commit log and memtable on all replica nodes in the cluster for that partition key. The quorum consistency level provides strong consistency if the user or application can tolerate some level of failure. Here, a write must be written to the commit log and memtable on a quorum of replica nodes. In the area of weak consistency, there is the one consistency level where a write must be written to the commit log and memtable of at least one replica node. The difference with the one consistency level and the local_one consistency level is that in the local_one consistency level, a write must be sent to and successfully acknowledged by at least one replica node in the local data center.

The read consistency levels specifies how many replicas must respond to a read request before returning data to the client application. The different levels include all, each_quorum, quorum, local_quorum, one, two, three, local_one, serial and local_serial. Similarly to the write consistency level, the all read consistency level provides the highest consistency of all levels and the lowest availability of all levels. It returns the record after all replicas have responded. The read operation will fail if a replica does not respond. The quorum consistency level ensures strong consistency if the user or application can tolerate some level of failure. Here, it returns the record after a quorum of replicas has responded from any data center. If the user or application can tolerate a high probability of stale data

being read, the one read consistency level can be considered. Here, it returns a response from the closest replica as determined by the snitch.

In general, what guarantees consistency in Cassandra is the minimum Write + Read i.e. (Write Consistency Level + Read Consistency Level) > Replication Factor.

Cassandra uses memtables, sstables and commit logs for write operations. A write is only successful if it is written to the commit log. The commit log is a crash-recovery mechanism that supports Cassandra's durability goals. From the commit logs, the value is written to a memory-resident data structure called the memtable which is implemented by the org.apache.Cassandra.db.Memtable class. If for any reason a write operation does not make it to memtable but is successfully committed in the commit log, it is still possible to recover the data. A threshold is set for each memtable and when the number of objects stored in the memtable reaches the threshold, the contents of the memtable are flushed to disk in a file called an SSTable. A new memtable is then created. All writes are sequential thus no reads or seeks of any kind are required for writing a value to Cassandra since they are all appended operations in the commit logs. Compaction then comes into play to organise the data for better future red performance. For reads, the memtable is first contacted to first find the value.

Explaining the concept of compaction further, data in the SSTables are merged, the keys are merged, columns are combined, tombstones are discarded and a new index is created. One major advantage of this process is improving performance by reducing the number of required seeks. The different types of compaction include a major compaction which is triggered via a node probe or automatically and a read only compaction.

Bloom filters, named after their inventor Burton Bloom are used as performance boosters in Cassandra. They are fast, nondeterministic algorithms for testing whether an element is a member of a set. They are categorised as nondeterministic because it is possible to get a false-positive read from a bloom filter but not a false-negative. The bloom filter is a special kind of cache stored in memory to improve performance by reducing disk access on key lookups. When a query is performed, the bloom filter is first checked before the disk is accessed. If the filter indicates that the element does not exist in the set, it certainly does not because of the possibility of false-negatives. In the case where the filter indicates that the element is in the set, the disk is accessed as confirmation.

Lastly, we look into the concept of tombstones which is similar to a soft delete. When a delete operation is executed, the data is not immediately deleted but instead treated as an update operation that places a tombstone on the value. A tombstone is a deletion marker that is required to suppress older data in SSTables until compaction can run. A garbage

collection grace second is the amount of time that the server will wait to garbage-collect a tombsone. By default, its set to 864,000 seconds, equivalent of 10 days. Cassandra keeps track of tombstone age and once a tombstone is older than GCGraceSeconds, it is garbage-collected. The purpose of this is to give a node that is unavailable time to recover; if a node is down longer than this value, then it is treated as failed and replaced".

2.4 Architecture for Managing IoT Fleet Managment Data

IoT goes beyond interconnection of "things" to provide products and services to new business ideas and revenue models. Different views for IoT have been proposed by a number of authors but we look at the one shared by [26]. Aggrawal et al categorize IoT views into three broad categories: things-oriented vision (focusing on devices), internet oriented vision (focusing on communication and interconnectivity), and semantic oriented vision (focusing on data management and integration). The things-oriented vision includes RFID-based sensor networks and data generated from other kinds of embedded sensor devices, actuators or mobile phones. The internet-oriented vision corresponds to construction of the internet protocol (IP) for enabling smart objects which are internet connected. This vision supports an object which is uniquely identifiable and may have real-time attributes like location which can be continuously tracked, a web of things which re-uses the web-based internet standards and protocols to connect the expanding eco-system of embedded devices built into everyday smart objects. Lastly, the semantic-oriented vision addresses the issues of data management which arise in the context of the vast amounts of information which are exchanged by smart objects and the resources which are available through the web interface. In this section we focus mainly on the internet and semantic oriented visions.

2.4.1 Running example

We ground the methodology and approach of the proposed research topic by considering the following two real-world use cases from the realm of fleet management:

Caravan Transport Group (CTG) [8]: Problem: CTG Inc is an Oakville-based carrier involved in cross-border freight transportation that owns a fleet of 1,500 trailers. CTG intends to reduce operating costs by improving efficiency, overall utilization of the fleet. CTG also wants to put real-time fleet information at the disposal of company decision makers to achieve the desired levels of cost reductions.

Solution: CTG installed BlackBerry (BB) Radar on one third of its fleet. BlackBerry Radar is an Internet of Things Gateway (IoTGW) that is mounted on the rear doors of the trailers; it is made of antennas to connect to a cloud and sensors to monitor temperature, humidity, and the state of the trailer door (closed/open), and the state of the trailer container (empty/loaded). The BB Radar collects readings every five minutes to detect any event of interest and sends reports every 15 minutes via cellular connection to BlackBerrys cloud-based IoT software platform. The later can then be accessed from a wide range of devices including tablets and smartphones.

This is an example of a real world implementation and relevance of IoT. IoT has been used to efficiently solve a real world problem through the collection and analysis of sensor data.

2.4.2 Generic Architecture for IoT systems

The whole idea in building a network of "things" is to have an end-to-end IoT stack. As mentioned by Mohak Shah in [56], the components of this stack will include data acquisition from the Machine-to-Machine(M2M) layer, data processing, data sharing through an interconnected network, insights' discovery and capability to relay results both to devices (for potential actions) as well as to (automatic or manual) decision makers. M2M technology is the technology involving wired or wireless communication of homogenous devices as well as capture and transmission of sensor data for storage and processing by applications.

The main focus in sensor based data management solutions is to harvest real-time data promptly for quick decision making taking into consideration their resource-constrained nature. Below is a generic architecture of a data management framework for IoT.

The proposed architecture is made up of 5 layers: the "things" layer, perception layer, network layer, middleware layer and application layer. The "things" layer consists of IoT sensors, smart objects and actuators. These are put on trucks and can be modified for any type of fleet. The network layer provides communication between the "things" layer and perception layer for transmission of data, results and queries. The perception layer handles the data that is first collected and offers filtering and preprocessing capabilities. This layer is proposed to first clean the data of all noise, fill in missing data, discard inaccurate or redundant data, detect events and store the massive amount of data that is generated from the "things" layer.

The network layer also consists of the IoTGW which provides the means to bridge the gap between devices in the field (in this case sensors, data objects and actuators) and the cloud where data is stored and manipulated by enterprise applications. It also offers

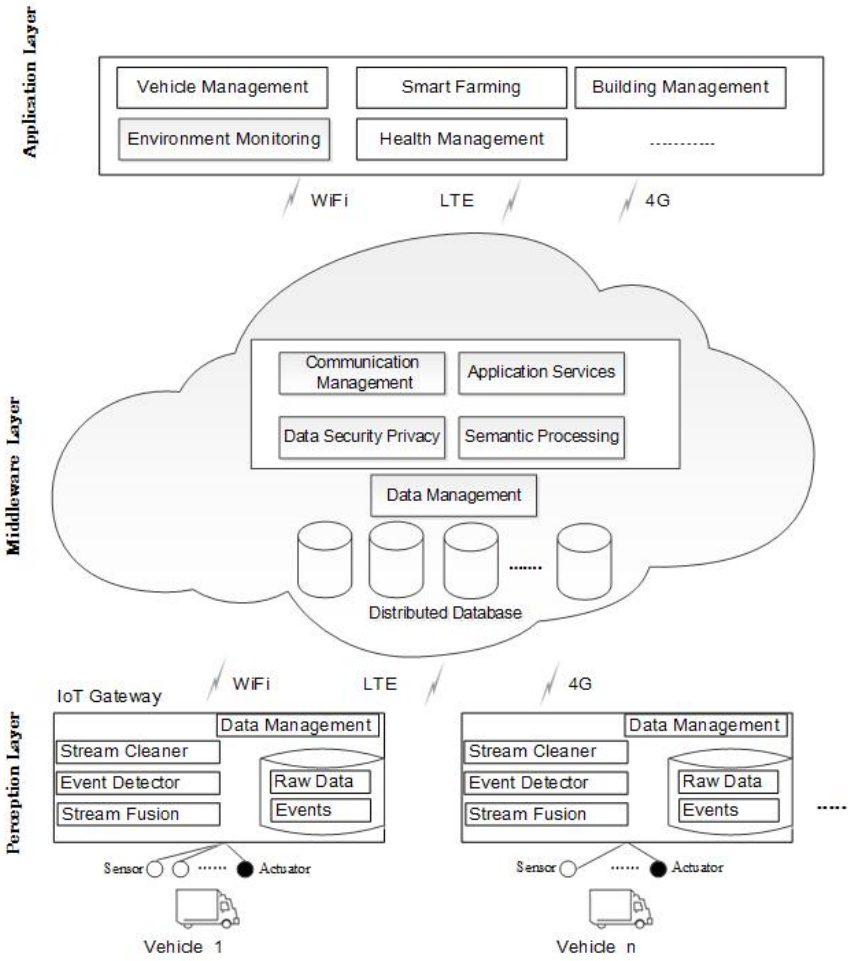


Figure 2.3: Fleet Management Architecture

local processing and storage capabilities to provide offline services and if required real time control over the "things" in the field [13].

The middleware layer consists of a distributed database and some amount of data processing. It consists of the communication management, data security and privacy, application services, semantic processing and data management modules. As has already been mentioned, data protection and privacy measures are a huge concern for IoT. The data security and privacy module has measures in accordance with legal frameworks that are of relevance to both the protected data and to the final users. Privacy plays an important role when "things" owned by private entities or persons such as data generated by medical devices or vehicles generate data and are stored. Various degrees of data privacy requirements can be defined depending on the nature of the data and the entity that generates the data [25].

There is another network layer, connecting the middleware layer to the applications layer.

It has the same characteristics and performs the same functionalities as the one between the perception layer and the middleware layer.

The primary reason for setting up this entire IoT architecture is to be able to make valuable decisions from the data that is generated by the "things". This wealth of data is used by the application layer for decision making. This translates to finding new and improved ways of doing things, new revenue models and increase in revenue and efficiency. The application layer therefore utilizes the information that is generated by the things layer to provide services and analytic capabilities to the end users of the IoT system. Even though this thesis work is focusing on the transportation and logistics application domain, there are other domains like healthcare, smart spaces and personal or social domains that IoT is very relevant.

In processing multiple streams as it happens with IoT data, there are two computational paradigms for IoT big data processing: Data Stream and Map Reduce. In a data stream model, data arrives in a stream and the processing algorithm is tasked with analyzing the data without explicitly storing it. In Map Reduce model, data is distributed in slave machines and computations are performed in the sequence of map and reduce operations. In this section, we would look at data stream for IoT.

Deploying sensor networks in which data is implicitly streamed, implies that there are multiple streams of data that needs to be processed at any particular time. In this section, we will look at streaming, querying and mining IoT stream data and the related challenges.

In the context of IoT, data streaming plays an important role in data processing and analysis due to the amount of data that is generated in the short amount of time. Making reference from [53], one of the typical challenges when it comes to sensor data processing is power constraints. This affects lifetime of the sensor and area of coverage. Sensors are usually powered by small batteries which may be difficult to replace. Theoretically, this puts a limit on how much data can be transmitted by a sensor before it runs out of energy. Data can either be sent by the sensors to a generalized server in which case the power constraints of the sensors becomes a major drawback or some data processing techniques including data aggregation, data compression, modeling and online querying can be done at the sensor network level to reduce the communication costs.

SAS event stream processing is a technology described in [54] that helps to quickly decipher and analyze volumes of continuously flowing events held in streams of data. As sensor data is generated, the technology examines and assesses the events real time before the data is stored. Instead of running queries against stored data, data management and analytical routines are used on event streams to filter, normalize, aggregate and detect patterns in

real time. All this is done before the data is stored to reduce the latency of the insights derived and influence actions in real time.

Query processing for streaming IoT data comes with its own unique challenges. As has already been mentioned, consideration should be made in terms of uncertain, ambiguous, inconsistent and heterogeneous data. Inspite of these, query processing can effectively reduce power consumption as results can be lumped together and sent as one. Aggregate queries which includes the use of MIN, COUNT and AVG operators means that data is preprocessed and the results further used in subsequent queries to produce a final result. In the case of fleet management, join queries are useful in processing data from moving trucks and multiple sensors. Join queries are used for combining data from two or more designated sensor networks to confirm accuracy and consolidate results.

Still under query processing, we look at top-k monitoring queries. This class of queries continuously report the k largest values obtained from distributed data streams. [30] maintains arithmetic constraints at remote stream sources to ensure that the most recently provided top-k answer remains valid to within a user-specified error tolerance. Their approach reduces overall communication cost between different stream sources. Lastly we look at continuous queries. In most application setups that involve sensors, there is the concept of being able to use the data real time. Sensors are thus required to answer queries in a continuous manner based on the data being generated.

Stream mining is extracting useful rules or information from data streams. Typical tasks for stream mining include clustering, classification, outlier and anomaly detection and frequent itemset mining. Clustering is the process of grouping a set of data or objects into a set of meaningful sub-classes. Objects in each sub-class will have more in common with each other as compared to objects in other sub-classes. The authors in [53] go on to mention some requirements to consider when designing algorithms for clustering data streams. These are unique to sensors as they are memory constrained devices with some time limitations. These include providing clustering results via fast and incremental processing of data objects, rapidly detecting new clusters or changes of existing clusters, scaling to the potentially unbounded number of objects in data streams, providing a model representation that is consistently compact regardless of the number of data objects, rapidly detecting the presence of outliers and acting accordingly and lastly, dealing with different data types such as XML, GPS temporal and spatial information.

Classification uses prior knowledge to guide the partitioning process to construct a set of classifiers to represent the possible distribution of patterns. Basically, compared with clustering, classification is a supervised learning process whereas clustering is an unsupervised

learning process. In outlier and anomaly detection, the main task is to find data points that are most different from the remaining points in a given data set. Most existing outlier detection algorithms are based on the distance between every pair of points. The points that are most distant from all other points will be marked as outliers. Frequent itemset mining is to find sets of items or values that co-occur frequently, or in other words, to find co- occurrence relationships in a transactional data set. Here a transactional data set refers to a data set where a set of items appears together in some specified context. Given a predefined support s, the goal in frequent itemset mining is to find all subsets of items that occur at least s number of times, or in other words, that appear in at least s transactional data sets at hand. Frequent itemset mining is both CPU and input-output intensive. Therefore, it is costly to completely re-mine a dynamic data set, which will be a typical case in IoT.

2.4.3 Fleet Management Architecture

There are a lot of advantages of using IoT for fleet management and companies rely on such vehicle fleet for their operations or even entire business models. These include postal and courier services, delivery industries, servicing companies etc. These connected vehicles are meant to optimize different value drivers like gas consumption, route optimization, service time reduction, resource allocation and fleet efficiency [56]. In our work, we propose a fleet management architecture for vehicles which can be adapted for other types of fleet like airplanes, trains etc. This can be seen in the figure below:

As can be seen from the figure 3.2, there are 4 different types of layers-things layer, perception layer, middleware and application layer. The things layer comprises of the entities that generate data for the entire architecture which are the trucks. In real time, this data is transmitted through the network layer to the perception layer. The network layer which is present wherever data is being moved around and between layers, consists of 3G, LTE and 4G networks. In summary, the things layer is the data source and can include a wide range of devices including sensors, RFID tags, mobile devices, laptop, embedded chips and any apparatus that has embedded intelligent devices with communication capabilities [25].

The IoTGW has 4 databases which comprises of a collection of real time data, derived data, inferred data and reconciled data. The authors in [32] explain these different types of data in IoT systems. It is a better design strategy to store real time data close to the source of the data generation.

Real time data is data reflecting the current status of the system. In our architecture, it is

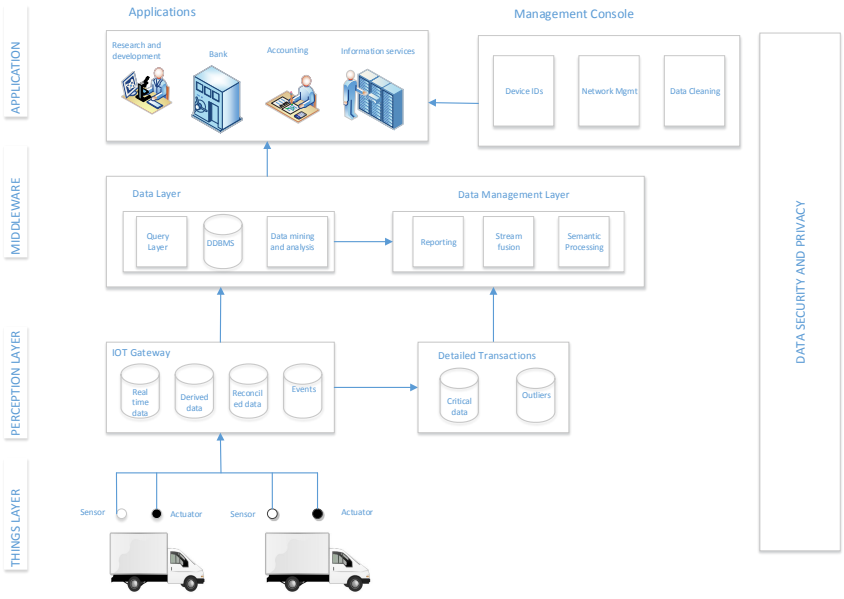


Figure 2.4: Fleet Management Architecture

the data that is being produced directly from the sensors located on the trucks. This data may include but not limited to information on the truck and its content, truck location and cargo, status of the truck, distance travelled, temperature and humidity. Real time data is usually stored close to the object that is generating the data to reduce the cost of data transmission and increases its availability. Derived data is data that has been generated based on some actions being performed on real time data. These actions could be summarizing, averaging or aggregating. It is therefore the conclusion of real time data. Reconciled data is real time data that has been cleansed, adjusted or enhanced to provide an integrated source of quality data that can be used directly by data analysts. Finally, there is an events database on the IoTGW which captures data that is different from the ordinary. A typical example is when brakes are suddenly applied on the truck, the speed may go from 100km/h to 0km/h in a few seconds. So the purpose of the events database is to capture all the data surrounding this brake event and store it.

On the perception layer, there is also a layer called detailed transaction. The main purpose of this is to capture critical data and outliers. Critical data is data that needs to be attended to immediately. Using the same example as before, if the brakes are suddenly applied and the data after this event shows the truck getting back on the road, this is not seen as a critical event. But in the case where the truck comes to a complete stop and does not move again, it is an event that needs to be quickly investigated. Outliers can on one hand

be seen as noise and may include data that has some parts missing thus may throw any form of analysis off board.

The middleware layer includes the data and data management layer. The data layer is the source or feed for the data management layer. We have chosen to use Cassandra as our distributed database management system. This data model has already been explained earlier. The middleware a very important component as it includes both the query and data mining and analysis modules. Both of these modules work on the data stored in the Cassandra database. The query layer is important for query processing and optimization. It is responsible for generating the optimized query, executing the query and storing the results of the query which is subsequently used in the reporting module of the data management layer. The data mining and analysis module use machine learning to discover patterns in the large data generated to be able to make informed decisions.

The data management layer has three modules- reporting, stream fusion and semantic processing. As the name suggests the reporting module is used to produce detailed and summarized reports on the data that is collected at the things layer. We will go on to talk about stream fusion and semantic processing in the later chapters.

The application layer consists of the applications and management console. There are numerous applications that consume the data generated by the sensors. These may include but not limited to transportation or logistics, traffic management, business intelligence applications, accounting applications etc. The management console module consists of the list of device ID's, network management and data cleaning modules. This list can be updated and it essentially contains information in relation to the workings of the entire architecture.

Data security and privacy is a layer that is applied to each of the different layers in the architecture. Movement of the data between the different layers have to be complete, accurate, trusted and secure. Data encryption and authentication plays an important role in this architectue to ensure its privacy. Unauthorized accesses can be flagged and investigated by the module to ensure that the data that is being generated and moved around is a true representation of what is happening on the field.

2.5 Internet of Things Gateways (IoTGW)

The primary purpose of an IoTGW is to settle the heterogeneity between various sensor networks and mobile communication networks or internet. It strengthens the management

of wireless sensor networks and terminal nodes and bridge traditional communication networks with sensor networks to make network communication easier and manage the devices of sensor networks. They are highly configurable, standalone computing systems that are intended to be embedded in devices, vehicles, buildings, etc, in order to serve as the end points on the edge of IoT. Therefore, the key issues of implementing an IoTGW system is to address the heterogeneity of different sensor networks, the diversity of protocols in the wireless sensor network and traditional telecommunication networks and strengthen the management of the endpoint networks. It can also be seen as the core unit of information gathering and control in the IoT architecture [63, 36].

One main characteristics of the IoTGW which is relevant to our work is its manageability feature as mentioned by the authors in [63]. It does both sensor node and gateway device management. Sensor node management aims to acquire the sensor's identification, status and properties and realize remote startup, shutdown, control and analysis. These pieces of information are essential when data is being stored and tagged with a particular sensor. The gateway device management realizes the gateway device's control, diagnosis, configuration, upgrade and maintenance.

The system requirements of an IoTGW are to perform data forwarding, protocol conversion, management and control of sensor nodes and as in our architecture storage of sensor data. An IoTGW has the basic functionality to receive and store data from the things layer and forward it to the upper layer in the architecture. To move this data, the gateway needs some form of wireless communication protocol. [63] mentions the use of Zigbee to acquire the data from the sensor nodes and then 2G/3G/LTE or any other network interface to send the packets to the internet or other layers in the IoT architecture. In this regard, the IoTGW analyzes and re-packages the sensor data based on the wireless sensor network protocols after receiving it and then sends this re-packaged data based on current telecommunication protocols.

2.6 Middle Tier Architecture

The middle tier architecture seeks to mediate interaction between the internal workings of the fleet management architecture and the third party applications that rely on the data produced at the sensor layer. It enables the different components in the entire architecture to interoperate and is the final level of their different functions before being exposed to the outside world. This architecture handles the heterogeneous applications that access the data and provides an interface that is easily accessible. The middle tier layer consisting of

the data and data management layer forms a Peer-to-Peer (P2P) network. A P2P network is a network of peers where devices communicate directly with each other. Files or data can be shared directly between the systems on the network without the need of a central server.

The middle tier architecture is made up of infrastructure which facilitates the creation of business applications and provides core services like transactions, concurrency and messaging. It supports third party application development and delivery as it supplies the data in processed form. It hides the distributed nature of the fleet management architecture and presents a collection of interconnected parts coming together for one purpose. It also provides a high level, uniform interface for application developers and integrators to compose their applications and import relevant data. It supplies common services and functions and avoid duplicating efforts and also enhances collaboration between different applications. This architecture can also be thought of as a business intelligence layer as it provides tools to extract, transform and load data into warehouses. It supplies the query, analysis and reporting tools used in decision support.

2.7 Control Management

Control management is a function that ensures all layers conform to the stated goals and deviations due to errors are seen and minimized. To this end, this is a function that is embedded in all the layers in our architecture. There are two types of control management. The first is putting measures in place to ensure that errors do not occur. This is especially important in an architecture where real time decisions are being made on critical events. Making an informed decision on erroneous data can have some serious implications and should be avoided at all costs. The question here is how the IoT infrastructure incorporates this kind of control management in all of its layers.

Let us take a look at the things layer. Sensors are devices that come with inherent issues like software glitches and may need software updates, bug fixes, hardware failures and repairs. To prevent inaccurate data from being reported, these devices will have to be checked and properly configured. Using the fleet management example, the license plate or VIN number of the truck the sensor is reporting on will have to be configured. Again other settings like the number of seconds between sending data and the type of data should be configured.

The second type of control management involves taking corrective action after the errors have occurred. The objective of this is to be able to point out the mistakes that have

occurred and the measures that are put in place to prevent them from recurring. An example of this is what happens at the IoTGW where missing data is filled in or inaccurate data is either corrected based on previous data or discarded entirely before being sent up the layers. In both of these instances, it is important to be able to maintain access to the devices to be able to maintain security and implement feature enhancements and fix bugs in their firmware.

One area of control management looks into all the different layers working together seamlessly to accomplish the goal of setting up the fleet management system. According to EFL Brech, control is checking current performance against pre-determined standards contained in the plans, with a view to ensure adequate progress and satisfactory performance.

2.8 Data Management

Data management refers to the architecture, best practices and process of managing the data lifecycle in any system. For IoT as with other domains, it shows how data is to be stored, managed and or archived. Archiving refers to the offline long-term storage of data that is not immediately needed for the system's ongoing operations [25]. The data management layer is an intermediary between the "things" generating the data and the applications accessing the data for analytical and reporting purposes. This is seen in figure 2.5 above. In regards to how data is stored, the data management layer is split into two parts and consists of long term and short term storage. One part is on the IoTGW to store data close to where it is produced to reduce data transmission costs. This is a short term storage as data is always moved to the main database at the data management layer for analysis and querying. At any one point, the data layer is a representation of all aggregated data received from various IoTGW sites.

IoT data is quick to become obsolete and irrelevant and due to the volume of data that is generated, it may not make sense to continue to store all that historic data. On one hand this data may become stagnant data which may not be used for any more analytical purposes or it could be further aggregated and still stored in the distributed database. But the question remains- at what point is there too much data and no longer useful?

2.9 Cloud Data Management for Fleet Management

The architecture for managing IoT data for fleet management could be seen as an extension of a Cloud Database Management System (Cloud DBMS). A Cloud DBMS is a DBMS that scales out the storage and replication of data to thousands of geographically dispersed data servers that are highly available. It must be elastic (i.e. the scale adapts dynamically to the workload which typically comes from a variable number of users), reliable (i.e. it should be fault tolerant), and available (i.e. data must be accessible in the presence of temporary node disconnections) while relaxing data consistency to increase performance. Typically, applications that run in the cloud have two characteristics:

- Applications that have a large footprint for data usage. Examples of these include Facebook, Yahoo, Amazon.
- Applications that have a small footprint for data usage. These applications are usually seen in Software-as-a-Service and Platform-as-a-Service solutions. Examples include Salesforce.com and Microsoft Dynamics Customer Relationship Management.

The former class of applications are usually single tenant applications where a single instance of a software or application is running with all of the supporting infrastructure serving a single customer or tenant. Each tenant thus has a dedicated DBMS server and instance of the software without any sharing with other tenants. This is usually important as the data usage is very huge. The advantages of this form of tenancy include enhanced security and reliability, easy backup and restore. The latter class of applications are multi tenant applications where each tenant's data requirements are often small that a lot of tenants come together to share resources in other to reduce the total cost of operation. In effect a lot of single tenants come together to form a multi tenant application. This type of tenancy is also offered on the cloud and is possible for different applications to share all tiers of the cloud software stack: the web or application tier, the caching tier and the database tier.

IoT applications for fleet management can be seen as a hybrid of both single tenant and multi tenant applications because

- Applications have a large footprint for data usage. This is seen in the amount of sensor data that is generated in a short amount of time.
- Applications have unpredictable load characteristics and varying resource requirements. This is a typical characteristic of a multi tenant application. Data is only

generated and sent up the IoT architecture when the fleet is on the move. This means there are varying amounts of data that is generated during a period of time needing different combinations of resources.

2.10 Conclusion

In this chapter, we started by looking at NoSQL databases and comparing some examples based on specific features. We delved into the data management approaches for IoT in the cloud and its relation to IoT. Processing multiple streams in IoT is an integral concept as in almost all architectures, there may be a lot of sensors with data coming in in short intervals. We explained the concept of both data streaming and stream mining. We talked about some query processing of streaming IoT data and briefly mentioned some types of queries. As has been seen above, not all of these domains is fully developed and there is a lot more work to be done and contributions to be added. This chapter has mainly been about existing research that has been done in the context of storing, processing and analyzing IoT data and the different components of the architecture in relation to fleet management and their seamless interaction with each other.

Chapter 3

Access Control Model

There are two arguments when we look at data access control in cloud storage services. The first is ensuring the security and privacy of information from other users of the cloud service provider (CSP) and the second is ensuring only approved users have access to sensitive or confidential information in the same enterprise.

To ensure a company's data is confidential against other users of the CSP, cryptographic approaches are usually applied where decryption keys are given to only authorized users. We look at work presented in [60] where the authors used different encryption methodologies to provide a fine grained access control for enterprises sharing confidential data on cloud servers. They combined a hierarchical identity-based encryption (HIBE) system and the ciphertext-policy attribute-based encryption (CP-ABE) system to produce a Hierarchical attribute-based encryption (HABE) model. The CP-ABE is an encryption system that allows the encryption of data by specifying an access control policy over attributes enabling only users with the same set of attributes satisfying the policy to decrypt the corresponding data. A HIBE system is a generalization of an identity based encryption (IBE) system that mirrors an organizational hierarchy. In this, an identity at level k of the hierarchy tree can issue private keys to its descendant identities but cannot decrypt messages intended for other identities. An IBE is a public key system where the public key can be an arbitrary string such as an email address [31].

Authors in [47] offer a similar solution for scalable, flexible, and fine-grained access control of outsourced data in clould computing. They proposed a hierarchical attribute-set based encryption (HASBE) by extending CP-ABE with a hierarchical structure of users thus inheriting flexibility and fine-grained access control in supporting compound attributes of CP-ABE. [61, 40, 48] all use an attribute based encryption method to provide data

access control to untrusted or public cloud servers without disclosing the underlying data contents. User privileges, confidentiality and accountability are enhanced in these models.

Our method of access control is the second where we assume that an enterprise data is secure on the cloud servers and we move to determine where data can be stored and eventually accessed by authenticated users.

3.1 Running Example

Consider for example a global delivery company like UPS that makes deliveries all over the world. In each city it works, it uses vehicles moving around the city to deliver packages. Each vehicle is equipped with sensor devices that writes user records generated to a database. The sensor data collected and sent from the vehicle include its GPS location, driver alertness, engine data and speed. This data can be collected in specific timelines, say every 3000 seconds and stored in a distributed database which has replicas all over the world. Business applications read this data to produce reports and make real time decisions. To ensure that these applications have local access to the data and very reduced response times, this sensor data is stored close to where they are produced and or will be frequently accessed. This dictates the access pattern of the sensor data stored.

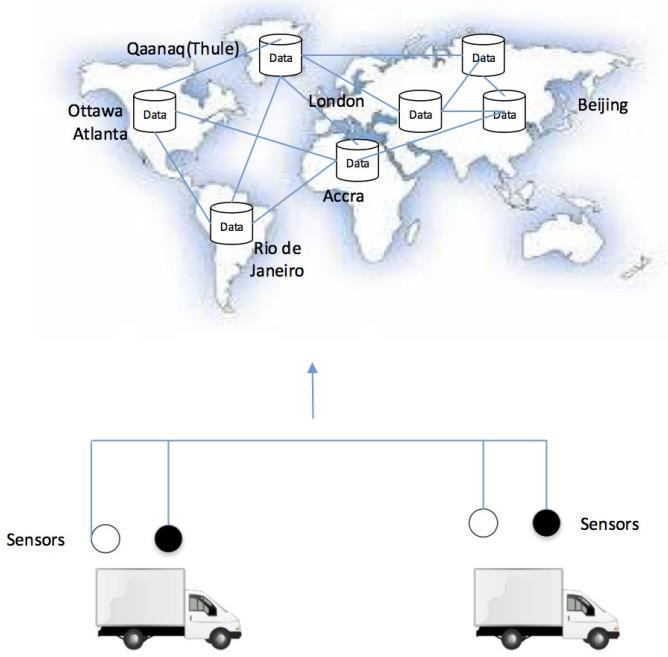


Figure 3.1: Distributed database cluster located in various cities

Each of the vehicles send their sensor data to data centers set up in each city as they move around and it is replicated in data centers in other countries for redundancy. Thus, data produced in Europe can be consumed in Africa. To prevent delays in writing data and to reduce data loss, this sensor data is stored in the closest data center. What happens when the data is being read is a different story. Data is also read from the closest datacenter and if that data center does not have the data requested, the read request is forwarded to next closest data center. In order to reduce the wait time for requested data and reduce the cost of communicating among data centers, it will make sense to replicate the data close to where they will be accessed.

The replication of data in a distributed environment has implications for system performance. Consider for example a data item D. Its replication strategy will be the set of nodes at which D is replicated. Thus the replication strategy determines how many replicas of D are created and to which nodes these replicas are written to. In a read-intensive network, a widely distributed replication of D in order to increase the number of local reads, thus reducing the response time for read requests, increases commnunication cost and decreases the system performance when an update has to made on the network. However, in a write intensive system, writing to a few nodes or updating a few replicas increases system performance and reduces communication costs. In effect the optimal replication strategy depends on the read-write pattern in the cluster.

3.2 Data Co-location for IoT Transactions

In incorporating transactional access into a system, two things must be considered. Can transactional access be limited to a single node or allowed to span multiple nodes? When data is co-located i.e. data items that are frequently accessed together in a single transaction are put on a single node or on nodes that are close to where the data will be accessed, it results in efficient transaction execution. A system built on this type of architecture, allows efficient transaction execution but often at the cost of constraining the application schema or data access patterns.

A transaction access can be limited to a single node. This class of applications allow efficient non-distributed transaction execution, system to scale out through the addition of more servers, the distribution of transaction load across the servers, limit the effect of a failure to only the transactions executing on the failed nodes and minimizes the impact on the remaining nodes, and allows techniques developed for optimizing transaction execution performance to be potentially applied.

Transactions spanning multiple nodes come with the advantage of offering a more flexible transactional interface to applications but at the cost of making transaction execution more expensive, relaxing consistency and isolation guarantees of transactions. Some systems go further to limit the operations supported and leverage application semantics to relax the performance requirements [27].

The main idea in using a distributed database to store data among other things is to ensure that data is replicated on multiple nodes. This increases redundancy of the data and reduces the risk of complete data loss when there are any unforeseen circumstances. However, there should be a strategy in place in the way that data is replicated to ensure that data retrieval is efficient. When a transaction is initiated and data is to be retrieved from multiple nodes to fulfil the transaction, the strength of inter-data center bandwidth and latency plays a major factor in relation to how fast the response is returned. We take a step back and look at inter-datacenter bandwidth for the initial replication of data. This consists of

- Replication bandwidth: The bandwidth required to send updates between datacenters. Replication can either be synchronous or asynchronous; it can be primary copy-based or multi master.
- Forwarding bandwidth: The bandwidth required to forward read requests to remote datacenters.

The idea here is to minimize the sum of replication and forwarding bandwidth [45].

With this background in place, we look at co-location of IoT data. The assumption we are making here is that data is stored in a globally distributed database. IoT data can be stored on specific nodes based on the types of sensors generating the data, the location of the sensors, the severity or priority associated with the data being generated and the origin of the sensors. As mentioned already, there are two design options to consider for data partitioning i.e. the systems that limit transaction execution to a single node and systems that allow transaction execution to span multiple nodes. IoT systems use either of the two.

Using definition of a transaction from database systems [27], a transaction is a set of operations executed in some partial order and is atomic i.e. there is no interference among transactions, and either all its operations are executed or none. In a fleet management system where for example a truck is fitted with a sensor to send in its GPS location, a transaction is a successful generation of the GPS co-ordinates in a pre-defined timeline and

transmitted over a network to the IoTGW. Transactions generated from the same truck are stored at a particular IoTGW based on the IP address. In this regard, these data items are co-located on a single node. It is easier to create summaries from this data and recognise critical events as and when they happen. Co-locating data from sensors located on a same truck will mean reducing the latency required in contacting other IoTGWs for data in relation to the truck and employing techniques to optimize transaction execution performance.

However, in a typical setup, there will be many sensors and multiple IoTGWs and co-locating data from a single truck to one IoTGW will mean that if the IoTGW fails, all these transactions will be lost and further transactions being generated will not be stored. Storing sensor data based on either the closest IoTGW or the IoTGW with the strongest network connection provides another design worth considering. This will mean that data relating to each truck can be aggregated on the various IoTGWs and either stored or discarded based on its content.

3.3 Selective Replication for IoT Data

Selective replication for IoT data is a mechanism to specify which nodes sensor data can be replicated to based on constraints such as legal, privacy or minimum replication levels and also read access pattern. In a distributed database system, the administrator usually specifies the replication level. This will mean that a replication level of 3 would automatically store 3 full copies of the data in different locations or depending on the architecture of the database, it will be stored on different racks in the same data center. Where the replicas are stored are usually determined by the internal mechanics of the database system. To achieve optimal system efficiency subject to any constraints specified, there should be a way where replicas can be stored on specific nodes while respecting the replication levels set by the administrator.

In a globally distributed database, replicas are copied to datacenters distributed all over the world to decrease remote traffic thus encouraging local access to data, increase speed of requests, response and resource efficiency. When a write is made at a node, it is persisted at the local server first and then updates are sent to all the replicas. The write consistency level will determine how many replica nodes must respond with a success acknowledgement in order for a write to be considered successful.

3.3.1 Data Operations: Read and Write

The read and write parts of our architecture are handled by applications and sensors respectively. These are on the two extremes. The sensors are attached to the fleet that writes the data and the applications which include business analytics that reads the data from the discovery platforms.

The first write occurs when sensors are attached to the fleet to collect specified data which is in turn sent to the database stored on the IoTGW. In a single node architecture, all transactions pertaining to a fleet are written to a single IoTGW over a network connection. There are two main processes that happen to data written to the IoTGW. First, data is sorted out to take out the noise and fill in for missing data through the application of machine learning algorithms. Secondly, data is categorized into critical and non-critical events based on the fleet. Non critical events may be discarded while critical events are sent over a network connection to a distributed database system.

When a sensor data is first generated, it is written to a database stored on the IoTGW. Storage limitations on the IoTGW means only a light weight database like sqlite can be installed and used. Machine learning techniques are applied to the data in the sqlite to fill in missing and incomplete data. Non critical data is then discarded and summaries are made of the critical data. The second write occurs when the summary is then written to a distributed database in the cloud. Working on the data in the distributed database is a discovery platform which has a mix of machine learning and analytical tools to make sense of the data for decision making. A read is done by applications which access the data to make reports and make real time decisions.

3.3.2 Optimal Selective Replication

If we had a complete knowledge of where data should be placed to reduce forwarded reads, increase response time and respect constraints, we would have a perfect database system. In this section, we present a model that will help developers optimise the latency seen in distributed databases like Cassandra. In the table below, we define the parameters used in this model.

We consider database settings where a cluster is deployed spanning different geographical locations. Thus, there are datacenters(DC) in different locations which stores sensor data and the applications accessing and producing reports from this data are also stored in different locations.

Table 3.1: Parameters used in model

Term	Meaning
n	node
i	size of record
j	datacenter location
h	application location
r	read
w	write
T	read and write threshold latency
F(T)	Fraction of reads satisfied within threshold
num(n)	total number of nodes
num(i)	total number of records
FW	forwarded reads
num(FW)	total number of forwarded reads
num(r)	total number of reads
num(w)	total number of writes
RF	replication factor in the cluster
obj	objective function

When data is generated and stored in DC1, based on the replication factor, this data may be replicated in DC2 and DC3. The latency here will be the delay in sending the packets to the other data center locations. We assume latencies associated with bandwidth costs between the nodes in the cluster is negligible. Size of the record i, is equivalent to the delay in writing or reading the record from a data center. When data is to be read, the number of pairwise connections or hops in the network is measured.

We define three different types of writes in our model. As sensor data is being generated in real time, this data is initially sent to the IoTGW. If this data goes beyond some predefined thresholds, it is deemed critical and thus must be sent to the cloud immediately. This is the first type of write. The second write is when data is being replicated to other nodes in the cluster as per the replication factor defined. Lastly, at the end of each data, data stored in the IoTGW is written to the cloud in a form of a scheduled download. Even though this type of data does not contain critical data, it is still stored and used for analysis. Each of these different type of write is associated with a network latency which we will define in the equations below.

$$FW = num(n)/RF \tag{1}$$

$$F(T) = num(r) + num(w) \tag{2}$$

$$num(i) <= T^{num(r)num(w)} \tag{3}$$

$$Costofreads = FW * i \tag{4}$$

$$Costofwrites(a) = i * num(i) * a \tag{5}$$

$$Costofwrites(b) = i * num(i) * b \tag{6}$$

$$Costofwrites(c) = i * num(i) * c \tag{7}$$

$$obj = sum(Costofreads + Costofwrites) \tag{8}$$

There are three types of read requests in Cassandra. These are a direct read request, a digest request and a background read repair request. In a direct read request, the coordinator node contacts one replica node for data. A digest request is sent to a number of replicas determined by the consistency level that has been setup by the client to check if the data at the replica nodes are up to date. If any replica node has out of date data, a background read repair request is sent to make the requested row consistent on all replicas involved in a read query [19]. In a multiple datacenter deployment, Cassandra optimizes write performance by using the coordinator node. When there is a write request, from a client application, the coordinator nodes, forwards this request to one replica each in the other datacenters with a tag to forward the write to other local replicas.

If a DC does not hold a data to satisfy a read request, the request will have to be forwarded to the nearest DC. The Cassandra system retrieves the full item from the closest replica and explained earlier, digests from others[57]. We define this forwarded read in equation (1). The default value is 1 and should not go below this. Both forwarded reads and writes are subject to latency threshold which implies the request will have to be satisfied within a predefined threshold in order to satisfy existing SLAs.

Equation (2) defines the total fraction of reads and writes satisfied within the predefined threshold. We make an assumption that communication costs or any form of network latency between nodes in the cloud is negligible. The size of a record is equivalent to the delay in writing the record or fulfilling a read request. This implies, the total size of records must be less than or equal to the total number of read and write thresholds. This is defined in equation (3).

Our objective function will be to minimize the sum of the cost of reads and the cost of writes. There is some amount of delay or latency when a read request has to be forwarded to another node to be fulfilled or another node not necessarily the closest or the local has to fulfil the request. The forwarded request coupled with the size of the record constitute

the cost of the reads and its defined in equation (4).

The three different types of writes defined earlier, represent equations (5), (6) and (7). Each equation differs from the other based on the network latency associated with the write request. The size and the number of records being returned are also considered. The variable 'a' in equation 5 has the highest value among the three equations. This represents the highest form of latency experienced when critical data is being transferred from the IoTGW to the cloud for immediate processing and analysis. In equation 6, the variable b represents the network latency used in replicating data to other nodes in the cluster. We used AWS to implement nodes that were set up in four different regions. There is inter-region latency as is expected and depending on the region where the node is, it either falls below 100ms or above 180ms. For the purpose of this work, we are assuming that most of the writes originate from Canada (i.e. in Ottawa) The value is the lowest as compared to the other forms of writes. Data generated by the sensors on a day-to-day basis may not always contain very important pieces of data but all of it is needed as part of historical data for data analytical purposes. This is the idea behind scheduled writes at the end of each day to transfer all aggregated data generated within the day at the IoTGW to the cloud and is represented by the variable 'c' in equation 7.

Modeling this, we considered three variables- datacenter location, application location and size of data.

Sets:

j datacenter location /Ottawa, Atlanta, Accra, Beijing/

h application location /New-York, Gatineau, Kumasi, Alberta/

Even though data is stored in specific locations, we make the assumption that this data will be read from different locations. We represent this as the application location. In retrieving data between the two locations, the number of hops or the pairwise network latency is measured and is represented in this table as d(h,j):

Table 3.2: Pairwise Network Latency

	Ottawa	Atlanta	Accra	Beijing
NewYork	3	4	2	5
Gatineau	2	6	3	4
Kumasi	5	4	1	12
Alberta	1	2	11	10

Two scalar variables were defined:

f - network latency RF - replication factor

k(j) is defined as the size of data. This is measured in mb. Sample of the data is below

Ottawa 32, Atlanta 10, Accra 11, Beijing 67

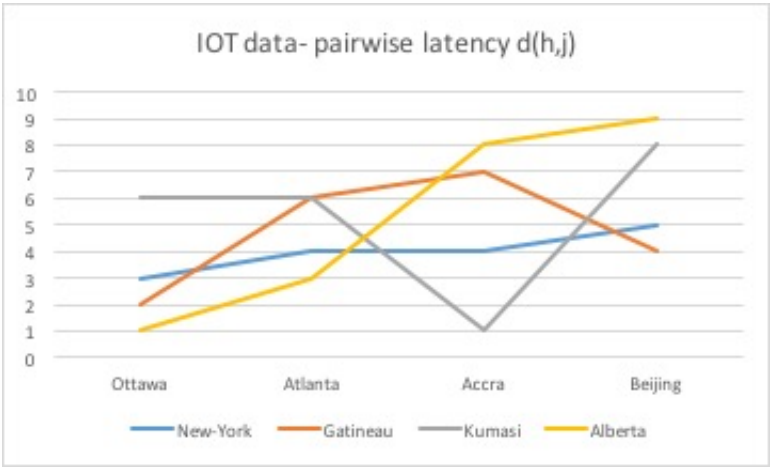


Figure 3.2: Pairwise latency

Cost of writes, cow(h,j) is the cost of writes for critical events where network latency is measured as 1 divided by the linkspeed . cor(h,j) is simply defined as the cost of reads i.e.

cow(h,j) = f * k(j) * (1/linkspeed)

cor(h,j) = d(h,j)/RF * k(j)

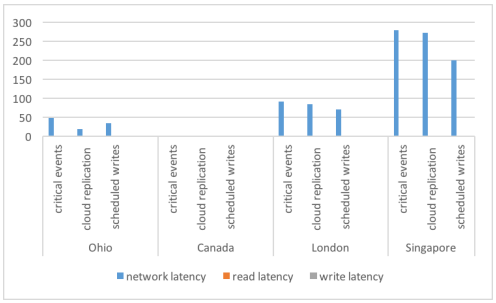
Two objective functions were defined as obj and obj2 i.e.

obj2.. phi2 =e= sum((h,j), cor(h,j) * k(j))

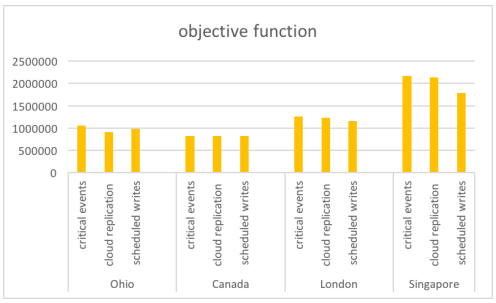
obj.. phi =e= sum((h,j), cow(h,j) +phi2)

Minimizing phi using mixed integer programming(mip) results in an optimal solution.

We run the model defining the network latency from Canada to all other nodes, the read and write latencies and the objective functions.



(a) Read and write latencies



(b) Objective function

Figure 3.3: AWS summaries

3.3.3 Language for Policy Constraints

Policy constraints are normally determined by legal requirements, characteristics of the underlying cluster architecture and specific application requirements. Cassandra administrators will be able to specify which specific data-centers are visible from a specific datacenter to aid how gossiping is done in the cluster. The rules defined by the administrator consists of a predicate that defines datacenter and its allowed or blocked counterparts. Below settings can be SET for each cluster: BLACKLIST: all datacenters specified under a blacklist are removed from gossiping with the datacenter; the remaining datacenters in the cluster are allowed for data exchange. WHITELIST: In this case, only the datacenters specified in the whitelist are permitted to communicate with this datacenter. All other datacenters in the cluster are blocked from any communication. Note that for each datacenter, only one of above rules can be applied at any one time i.e. you cannot specify a blacklist and whitelist setting for the same datacenter.

We have developed a simple constraints language for specifying constraints. This format is not domain specific and can be applied widely and allows constraints to be specified by database administrators to control access to IoT data. Each line in our constraint file must have below format:

ruleline ::=<datacenter>"=" <json-constraint>

<datacenter>::="" [a-zA-Z_]+ ""

<json-constraint>::="{" <constraint-type>" : "<$datacenter - array$>"}"

<constraint-type>::="whitelist" | "*blacklist*"

<datacenter-array>::="[" <datacenter-affected>∗(","<$datacenter - affected$>)"]"

<datacenter-affected>::="{" "dc: "<*datacenter*>"}"

EXAMPLE: Consider a Cassandra cluster consisting of datacenters named after the respective countries of locations: Canada, Mexico, US, Japan, China and Ghana. Now assume that due to regulatory concerns data originated in Canada must not be stored in China and Mexico then below rule can be a valid policy constraint definition for Canada datacenter: Canada={"whitelist":[{"dc": "US"},{"dc":"Japan"},{"dc":"Ghana"}]}

Above rule specifies that only datacenters Ghana, Japan and US should be allowed to communicate with the Canada datacenter; all other datacenters i.e. Mexico and Japan should be blocked from gossiping with this datacenter

3.3.4 Constraints Validity

Section 4.2.3 describes the format in which a rule must be written for it to be considered valid. We now define validity. Given the set of all possible datacenter locations DC, a constraint rule is defined as R, DC(R) are the set of datacenter locations defined, WL(R) is defined as a whitelist configuration, BL(R) is defined as a blacklist configuration. min(DC) is defined as the minimum number of datacenters that can be used. A constraint rule is valid if

- $\forall R \rightarrow WL(R) \parallel BL(R)$

- $\exists min(DC) \rightarrow 1 \leq min(DC) \leq \mid DC \mid$
- $\exists WL(R) \rightarrow WL(R) \subseteq DC$
- $\exists BL(R) \rightarrow BL(R) \subset DC$
- $\exists BL(R) \wedge \exists WL(C) \rightarrow BL(R) \cap WL(R) = \emptyset$
- $\exists BL(R) \wedge \exists min(DC) \leq \min DC \mid - \mid BL(R) \mid$

An invalid constraint validation will throw an error in the logs and will result in the constraint not being applied to any datacenter. All of these work together and therefore more than one of the validity rules can be applied at any one time.

3.3.5 Access Pattern Based Placement

In this modern era where everything is advertised as being fast, a user having to wait for a page on a website to load may simply close the webpage and move on to something else.

This may be mean a loss in advertising revenue or even a valuable customer. Even though most internet speeds are at their fastest than they have ever been, application developers still have to be careful about response times of their webpages. One school of thought argue that placing frequently accessed data together in a single transaction results in an efficient transaction execution and shorter response times. This is the scenario that was described in the data colocation section. There have been some authors that have described various ways to move data around especially in a globally distributed database after counting how many times this data was requested from a specific location. As described in [45], there is a user record for Alice who lives in Europe and constantly updates her Facebook profile. If Alice never travels to Asia and her profile is rarely accessed from Asia, the network and disk bandwidth to propagate her updates to an Asian datacenter are not justified.

Since this paper uses Cassandra, we will briefly describe how Cassandra attempts to solve access pattern based placement. In a globally distributed Cassandra setup, a user is connected to the closest or local datacenter to receive read and write requests. When a client performs a read or write request, the coordinator node contacts the number of required replicas to satisfy the consistency level included with each request. An example is where a read request is being processed using a quorum consistency, a cluster created using network topology strategy and a replication factor of 3. A quorum is calculated and rounded down to a whole number i.e. (sum of replication factor/2) + 1. Here, 2 of the 3 replicas for the requested data would be contacted and a result sent back to the client.

Cassandra combats cross data center latency by promoting local reads and writes, multiple replicas in each datacenter, ensuring that a client's load balancing policies prefer the local data center, ensuring data is consistent across all data centers through the run of a repair operation more frequently and housing datacenters in the same clusters. With these steps, cross data replication latency is negligible if any at all and thus does not warrant co-locating data.

3.4 Optimal Static Placement

We discuss our novel methodology of implementing constraints in Cassandra. We initially address the grammar used, and explain the algorithms implemented and go on to define the constraint language.

We implement the blacklist and whitelist functionality in Cassandra to provide a more flexible gossip through the use of JavaScript Object Notation (JSON). JSON is a lightweight,

text-based language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. It defines a small set of formatting rules for the portable representation of structured data [6]. We are implementing our methodology using JSON as it is an extensible language and more constraints or attributes can be added in future.

As has been mentioned earlier in Cassandra, a collection of nodes come together to form a datacenter and multiple datacenters come together to form a complete cluster. Cassandra performs its data distribution and replication through consistent hashing which allows the distribution of data across a cluster to minimize reorganization when nodes are added or removed. Consistent hashing partitions data based on a partition key. If data is to be written, a hash value is assigned to each partition key. Each cluster is responsible for a range of data based on the hash value and Cassandra places the data on each node according to the value of the partition key and the range that the node is responsible for [16].

The first assumption we make is that datacenters forming a cluster are found in different countries. The second assumption is that, based on some constraints which could be legal or privacy issues, data generated in one country cannot be stored in a particular country. We have implemented blacklist and whitelist attribute list that allows an administrator to specify specific countries or locations where data can or cannot be stored. Datacenters on the whitelist can have data written to them successfully while datacenters on the blacklist cannot have data written to them and are not available for gossip.

Using a sample configuration for the algorithm described in figure 4.4 below, Canada={"whitelist":[{"dc":"US"},{"dc":"Japan"},{"dc":"Mexico"}]}, var1 =Canada. If the administrator has set a valid whitelist or blacklist configuration then the string is successfully parsed. Since the config is that of a whitelist, StorageService is set to Canada, Japan, Mexico, US and China. Var2 is then set to China as this is the only datacenter not part of the string. Var2 is then passed to the removeDCs variable, hints are disabled and no data written to the Canada DC is replicated to the China DC in respect of the specified constraints. The removing DC algorithm is described in figure 4.5. This algorithm works with two main lists- dcsWithHHDisabled and the restoreHHDCs. The dcsWithHHDisabled are DCs unavailable in the cluster and restoreHHDCs contain DCs that have previously been blacklisted.

The algorithms used are shown below:

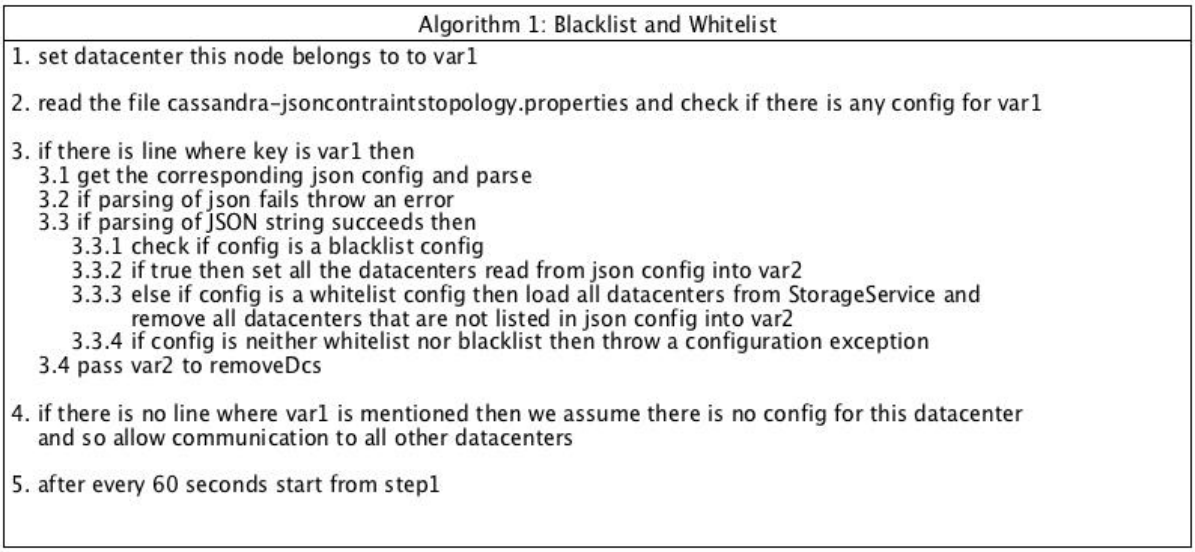


Figure 3.4: Blacklist and Whitelist algorithm

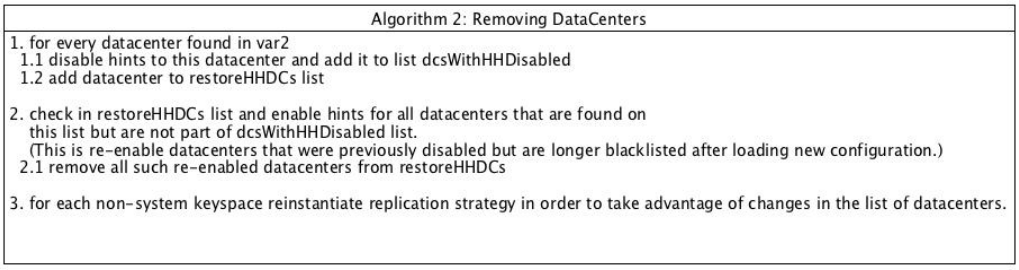


Figure 3.5: Algorithm for Removing DCs

Cassandra uses gossip as its primary way of exchanging information between nodes. It is a peer-to-peer communication protocol in which nodes periodically exchange state information about themselves. A DC found in the excludeDC list implies that all nodes in this DC is not available for communication. The assumptions we make in our model is that a DC cannot be on its own blacklist and thus an exception will be thrown if this is erroneously done in the configuration. A DC is available for gossip and thus the sendGossip method is called if it found in the isGossipableDatacenter list. Below listings shows a summary of how the selective replication is actually achieved using our procedure.

Algorithm 3-Implementing Selective Replication

1. set var1 to datacenter this node belongs to
2. check that the file cassandra-jsoncontraintstopology.properties exists
    2.1 if files does not exist create an empty list of excludeDCs and exit
    2.2 if file exists continue to next step
3. Create an excludeDCs list after parsing the file.
4. check the excludeDCs file, and;
    4.1 if excludeDCs contains var1 throw an error as node cannot communicate with other nodes in own DC
    4.2 if excludeDCs does not contain var1 continue to next step
5. for each gossip to be sent
    5.1 check if excludeDCs is empty
        5.1.1 if true, then sendGossip
        5.1.2 if excludeDCs is not empty
            5.1.2.1 check if the node to gossip with belongs to a whitelisted DC by calling isGossipableDatacenter
                5.1.2.1.1 if isGossipableDatacenter returns true, sendGossip
                5.1.2.1.2 if isGossipableDatacenter returns false, exit

  end loop
6. repeat step 5

Figure 3.6: Selective Replication Algorithm

Method - isGossipableDatacenter (takes DC as input and returns true|false)

1. set var1 to the DC to be checked as gossipable
2. check size of excludeDCs
    2.1 if size is 0, return true

2.2 check if var1 is in excludeDCs
    2.2.1 if true, return true
    2.2.2 if var1 is not in excludeDCs, return false

Figure 3.7: Datacenters available for Gossip

3.5 Centralized Approach of Access Control Model

In this section we present a centralised approach to our policy constraint implementation. A centralised system is simply a system that is located, stored and maintained or operated from a single location. Consider a 6 node Cassandra cluster distributed in different cities in a country. Each of these nodes is configured and set up to respond to both reads and writes from other nodes in the cluster. In order to implement policy constraints, the constraint file on a node is designated as a "master" and contains configurations for each of the different nodes in the cluster. All changes made to the constraint file on the "master" node overrides the constraint file on each of the nodes after a pre-determined delay. During a write request, each node simply uses the constraint file stored in its configuration folder

to confirm which restrictions apply in terms of the number of replicas and the location of these replicas.

The "master" node sends out a constraint broadcast to all nodes in the cluster at pre-configured time intervals. The questions that arise here are what happens when constraints change in the middle of a write? Do old configurations take precedence over current ones when the write is started? Where are these broadcasts stored on each server? What pre-configured intervals will not have too much impact on system performance? How does a centralised system handle errors in configurations?

This centralised approach to implementing policy constraints comes with the advantage of changes being made in only one location and thus control over its contents is maintained as it can easily be analysed or mirrored. There is a single source of failure and management of this list is minimal as it is located in one location. It is a write-once-read-all architecture implying constraints and updates are written once and applied to all existing nodes. Adding more nodes simply means updating the constraint file with whatever conditions pertains to this new node.

Questions that arise from using a "master" node are what happens when this node is unavailable? What goes into demoting a "master" node and setting up a new one? There is an increase in communication costs as all nodes will have to be contacted for an update in the constraints file. In a write-intensive system, a centralized approach may not be feasible as system performance will generally be degraded because of the constant broadcast. However, in a read-intensive system, there may be fewer nodes to contact and thus read latency is reduced. During a read request, no reference is made to the policy constraints. If data is not available locally, the coordinator node will forward the read request to other nodes, return the result and perform a read repair if it is required. The response time is even shorter when data is available locally.

3.6 Distributed Approach of Access Control Model

There are two implementations in a distributed approach for implementing policy constraints in an access control model. In a distributed approach, each node in the cluster has their own copy of the contraint file. There are two ways that this can be achieved. The first is creating a constraint file on each of the nodes in the cluster. Changes in the configuration will have to be applied manually and independently on each node. This affords the flexibility of adjusting the configuration on a single node to achieve a certain desired effect whilst the configuration on other nodes remain the same. It however this comes with

the drawback that even for a small change, an administrator must login to each node and effect the change. This can be a long and tedious process which is prone to mistakes. In a large cluster, the complexity is multiplied.

The second type is creating copies of the constraint file on each node, however, changes made to any of the files are automatically propagated to each of the nodes and at each point in time, all the nodes will have the same configuration. There is no designation of a node as a "master".

In the first design, local files stored on each server may be different as there is no coordination between the nodes on application of constraints. This is a more difficult implementation in especially a cluster with lots of nodes since the creation and updating of the file is a very manual process. The issue of an administrator or an unauthorised user making illegal changes to one of the files is worth considering. Even though securing the configuration files of Cassandra is beyond the scope of this paper, this is worth mentioning. As long as the constraint is valid, our model will go ahead and apply them to restrict where data is stored in the cluster. There is not another level of checks to confirm that contents are authorised. It is easy to log the changes made and by which user but this will be a reactive rather than a proactive approach. Cassandra does provide some security features like authentication based on internally controlled rolenames and passwords, authorization based on object permission management, authentication and authorization based on Java Management Extensions (JMX) usernames and passwords and Secure Socket Layer (SSL) encryption and also some general security measures like closing all non-essential firewall ports [21]. It is interesting to note that the cassandra.yaml file is stored in plain text so these security measures have worked so far.

The second design where copies of the constraint file are found on each node but changes are made on any one node and automatically propagated to each of the nodes is a hybrid of the centralized and distributed approach. Having the files stored locally will mean that requests are fulfilled faster. Administrator will simply have to make changes on any one node and set a broadcast interval for the changes to be sent to all nodes in the cluster. if there is a hardware failure with the node propagating the changes, the nodes will simply work with the version of the constraint file they have stored. Setting up of more nodes can easily be done and configuration changes will be minimal.

In this implementation, each node will have local access to the implementation of constraints, thus overcoming the hurdle of contacting a "master" node for each update of the constraint file. Replication will be dramatically improved using this approach. Each node will have local access to the implementation of constraints. When a Cassandra node is

started, the constraint file is loaded and its contents is applied during a write request. This reduces the communication cost found in the centralized approach and decreases the write latency. Changes to the read-write requests are responded to in a timelier manner. This design holds up very well in relation to hardware failure or a node being unavailable because of its high redundancy structure. Data is easily recovered from other nodes and applied to any new server setup. The system is therefore not severely impacted. There is however the issue of bandwidth cost when updates to constraints are being sent in the cluster. How often can this be done so as not to increase the communication cost and affect the efficiency of this type of approach? Errors in the constraint rules are propagated to each node and can result in degradation of system performance with each node raising an exception in parallel.

Looking at the pros and cons of each of these designs, we have decided to go with the distributed approach. We have implemented our model with each node having the constraint property file as part of their configuration.

3.7 Conclusion

In order to provide quick access to data in a globally distributed database, data must be physically located near the user. However, the cost of replicating data all over the world is very prohibitive. In this chapter we examined options in replicating data to specific places in the midst of respecting constraints which may be due to legal, privacy or preferential reasons. For IoT data, the sensors generate and write the data to a database while applications which generate reports for analysis are the readers of this data. We looked at an optimal selective replication which was based on the size of the data and the assumption that datacenter locations may be different from the application locations introducing the concept of forwarded reads as data may not be available locally. We went ahead to share the syntax for representing policy constraints and stated what exactly makes a constraint valid. We use an easily extensible language to make it easy to add more rules and also shorten the learning curve for new uses.

We share our methodology for implementing constraints in Cassandra using our optimal static placement approach and describe the algorithms used. It should be noted that our strategy can be tweaked and applied to other distributed databases. The concept of centralized or decentralized database systems has been discussed extensively in literature. With this in mind, we examine how best to implement our approach. We describe the design guidelines and its application in our model.

In Chapter 4, we present experiments using a modified version of Cassandra installed in datacenters around the world on Amazon AWS servers. This demonstrates the effectiveness of our solution. We share the results, do some analysis and share our conclusions.

Chapter 4

Performance Evaluation

Our perfomance evaluation for this paper will be focusing on the distributed approach of access control on IoT data. The experiment can be summarized and easily understood based on this simple flowchart below:

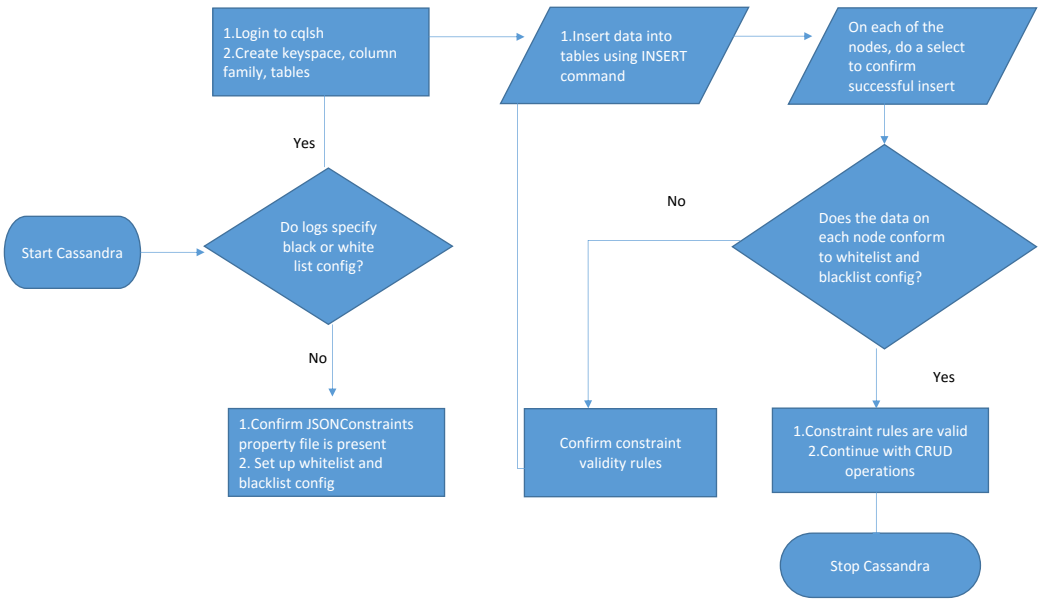


Figure 4.1: Modelling our Solution

4.1 Testbed

A JSONConstraintsDatacenterTopologyProvider.java class is created in Cassandra to add the ability to configure Cassandra instances using constraints written in JSON format. This file is put in the Cassandra path/conf. There are two main methods and a class used to implement our idea. These are the reloadDatacenterTopologyProvider(), reloadConfiguration() and the DatacenterTopologyProperties class.

The reloadDatacenterTopologyProvider() starts off by calling the reloadConfiguration() method to first confirm if datacenter is part of the list of valid datacenters in our cluster. If true, then we confirm if this is either a blacklist or a whitelist configuration. We throw an exception if it is neither and exit the loop. However if a whitelist configuration is found, we put the list of datacenters specified in the JSON config into a variable and then call the StorageService.instance.getAllDatacenters() to get a list of all datacenters that have been configured in the cluster. We compare the contents of the two and set the list of datacenters that are in the StorageService variable but not in the JSON config into a variable called the removeDCs. The removeDCs is essentially a list of datacenters that are on the blacklist. A similar thing happens when the JSON config is a blacklist configuration. Here, the list of datacenters in the JSON config is put in a variable and assigned to the removeDCs which contains a list of datacenters on the blacklist. As datacenters can move from blacklist to whitelist for different datacenters in a cluster, there is a configurable refresh period for this to run and make any new changes to the configuration.

The reloadDatacenterTopologyProvider() contains source code to either disable Hinted Handoffs for datacenters in the removeDCs list or restore hinted handoffs for datacenters no longer on the list to make those datacenters available for write requests and to re-instantiate the replication strategy.

4.2 Configuration

We performed our experiments and validations using an actual wide-area Cassandra cluster deployed across 4 regions (and 4 availability zones) of Amazon Elastic Compute Cloud (Amazon EC2) using test IoT generated data. Each region is a distinct geographical area and each region consists of multiple, isolated locations known as availability zones. This setup enabled us to confirm the benefits of our solution in a real world scenario. Datacenters were set up in US East(Ohio), Canada, EU(London) and Asia Pacific (Singapore) regions. Each was an Amazon EC2 micro size linux (ubuntu) instance. An instance is

a virtual server in the Amazon Web Services (AWS) cloud and is Amazon Elastic Block Store (Amazon EBS) backed. The Amazon EBS provides persistent block storage volumes for use with the Amazon EC2 instances in the AWS cloud. The Amazon EC2 is a web service that provides secure, resizable compute capacity in the cloud for web-scale cloud computing [15].

Inter-DC latencies which occurs during both reads and writes were measured for a period of 12 hours at a time. One cluster consisting of 3 DCs were defined. Cassandra 3.0.14, cqlsh 5.0.1 and python 2.7.12 were installed. The cqlsh utility is a python-based command line client for executing Cassandra Query Language (CQL) commands cql.

Our dataset consists of 100000 IoT records and afforded us the flexibilty of doing muliple operations. The first experiment conducted was the control experiment where aggregated IoT data was sent to the cloud and written to the closest datacenter with data being replicated according to the replication factor. Consistency level was set at level two. Thus a write must be written to the commit log and memtable of at least two replica nodes to be considered a successful write. Modifications in to the Cassandra source code to implement our solution was done using eclipse.

4.3 Experiment

In the first experiment, we created one Cassandra cluster using the four availability zones mentioned. Three datacenters were created - DC1, DC2 and DC3 with DC1 consisting of data stored in Ohio and Canada and DC2 and DC3 containing data stored in London and Singapore respectively. A keyspace called iotkeyspace was created specifying the Network topology stragety with the different replication factors for the different datacenters. DC1 had a replication factor of 3, DC2 and DC3 both had a replication factor of 2. Consistency level was set at two. Based on the queries that were needed to be run, 5 tables were created.

TABLE Vehicle_Data(vehicle_DataID uuid, driver_name text, speed int, gps text, brake_level int, license_Plate text, distance text, duration int, driverID text, trans_date timestamp, PRIMARY KEY ((license_plate), driver_name));

TABLE driver_data(driver_username text PRIMARY KEY, driver_dataID uuid, driver_name text, address text, class_of_license text);

TABLE fleet_data(VIN text PRIMARY KEY, fleet_DataID uuid, fleet_type text, model text, manu_year text, registration text, driver_name text);

TABLE environmental(envi_ID uuid PRIMARY KEY, license_plate text, date timestamp, humidity text, temperature text);

TABLE trip(tripID uuid, license_plate text, driver_username text, start_location text, end_location text, date_of_trip timestamp, PRIMARY KEY (driver_username,date_of_trip));

We started with 500 IoT records and gradually increased while performing some reads and write. We mesured the latencies in milliseconds and displayed averages in graph below:

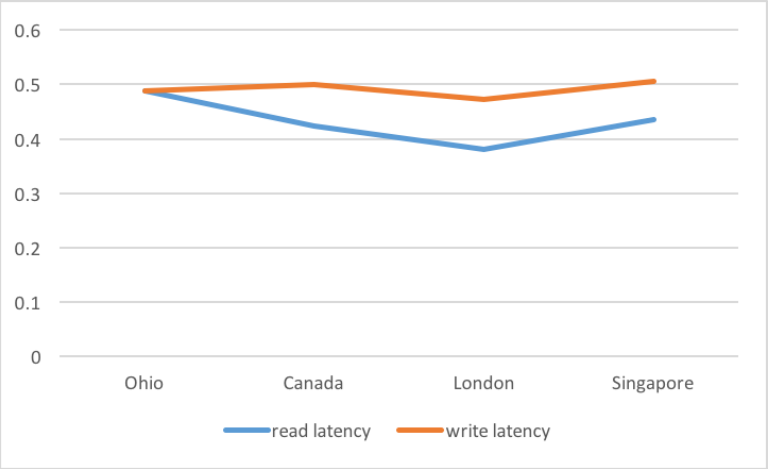


Figure 4.2: Latency(ms)

The modified version of the Cassandra trunck that we worked on was an edited version previously done by Jeff Jira. A Cassandra-jsoncontraintstopology.properties file was created and added to the configuration folder. Please see below a sample of its contents:
DC1={"blacklist":[{"dc":"DC3"},{"dc":DC2"}]}
DC2={"whitelist":[{"dc":"DC2"},{"dc":"DC3"}]}
DC3= {"whitelist":[{"dc":"DC3"}]}

When this configuration file is loaded successfully, the following logs are written to the log files:
Ohio
INFO 07:34:12 blacklist configuration found for DC1
INFO 07:34:12 Disabling communication with dc DC3, DC2 explicitly disabling hints
INFO 07:34:12 Resetting replication strategy of thesiskeyspace

Canada
INFO 07:33:23 blacklist configuration found for DC1
INFO 07:33.23 Disabling communication with dc DC3, DC2 explicitly disabling hints

London

INFO 07:34:51 whitelist configuration found for DC2

INFO 07:34:51 Disabling communication with dc DC1, explicitly disabling hints

Singapore

INFO 07:34:33 whitelist configuration found for DC3

INFO 07:34:33 Disabling communication with dc DC1, explicitly disabling hints

INFO 07:34:33 Disabling communication with dc DC2, explicitly disabling hints

Read and write latencies were measured to validate the modified version of Cassandra

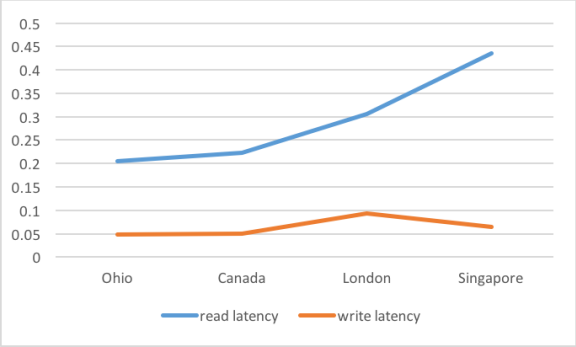
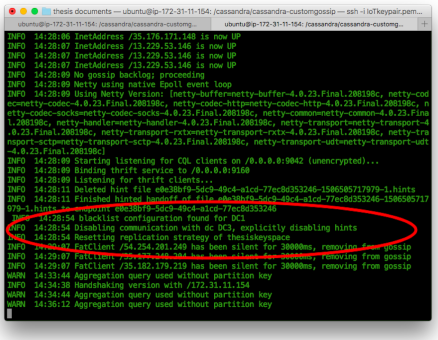
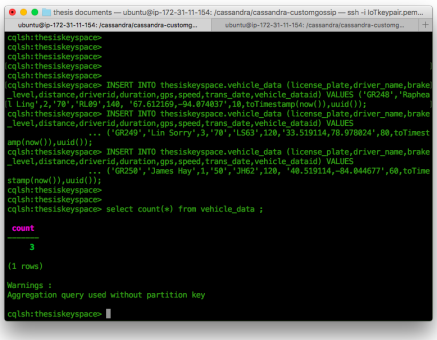


Figure 4.3: Latency(ms)

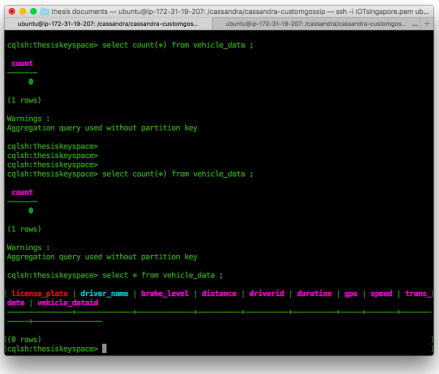
We see results of validation of our model below with the datacenter found in US east:



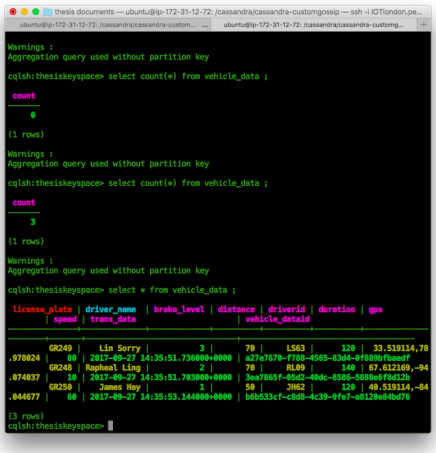
(a) Ohio DC - config file



(b) Ohio DC



(c) London DC

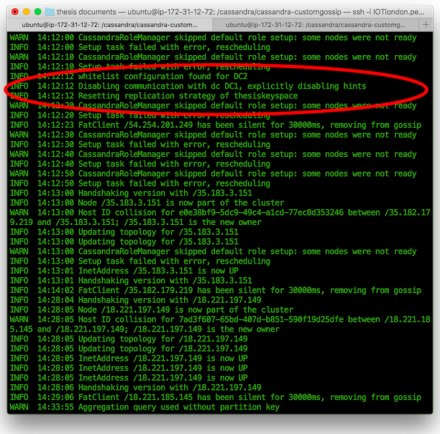


(d) Canada DC

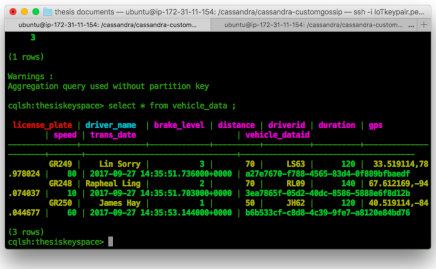
Figure 4.4: Read and write on DC1

When Cassandra is started, it acknowledges which DCs are on the white list and those that are on the blacklist. For DCs on the blacklist, it disables the hints and resets the replication strategy. The default Cassandra replication strategy is overridden to give the administrator control over which DCs store the data. In the figure 5.5, 3 rows of data are inserted in the Ohio DC and replicated to DC2. As has been mentioned, either a blacklist or a whitelist configuration can be specified. In the case of DC1, a blacklist configuration is set with DC3 on it. This means that whenever data is written to DC1, it can be stored anywhere but DC3. Cassandra therefore goes ahead to disable hints on DC3 and updates the replication strategy to implement this change.

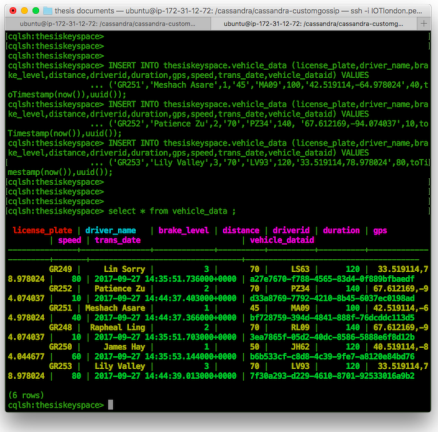
For the setup in London(EU), a whitelist configuration is found for DC2. Nodes on the whitelist are DC2 and DC3, implying DC1 is on the blacklist thus hints are disabled and replication strategy is reset. The insertion of 3 rows done on DC2 is not replicated on DC1 which continues to show previous records.



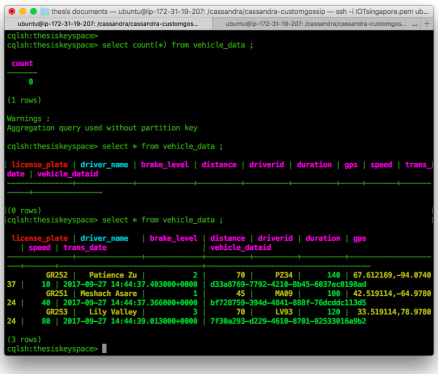
(a) London DC - config file



(b) Ohio DC



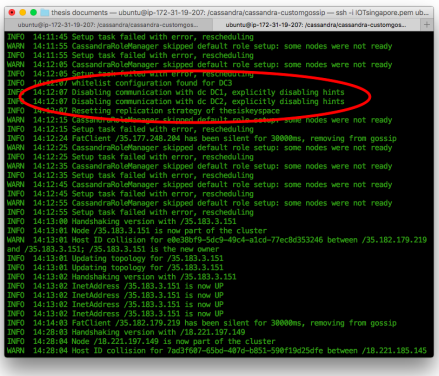
(c) London DC



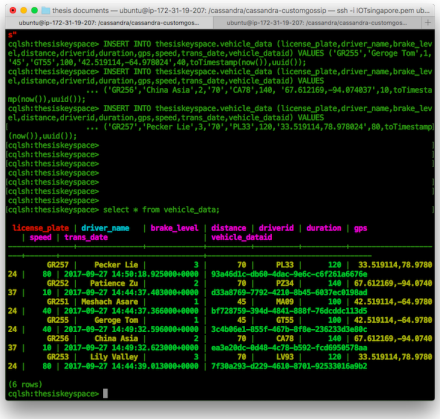
(d) Singapore DC

Figure 4.5: Read and write on DC2

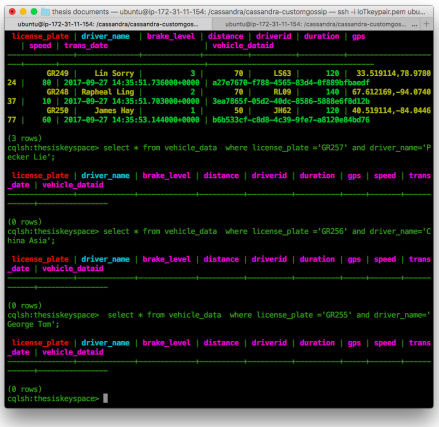
Lastly, we show results from Asia Pacific. DC1 and DC2 are on the blacklist and thus written to DC3 is not replicated on those nodes. After an insertion, we query both DCs to confirm that the data was not replicated there.



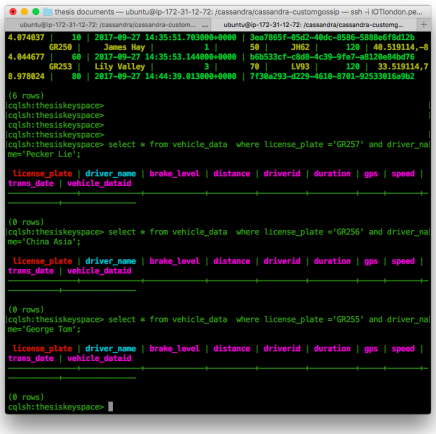
(a) Singapore DC - config file



(b) Singapore DC



(c) Ohio DC



(d) London DC

Figure 4.6: Read and write on DC3

4.3.1 Distributed approach in enforcing contraints

For our new json-constraint configuration to work well, we opted to use lsynced as our replication tool to mirror changes to this file in one node across the entire cluster. Lsynced is a linux-based synching utility that uses rsync for its transport layer. From the manual pages of rsync [23]: Rsync is a fast and extraordinarily versatile file copying tool. It can copy locally, to and from another host over any remote shell, or to and from a remote rsync daemon. It offers a large number of options that control every aspect of its behavior and permit very flexible specification of the set of files to be copied. It is famous for its delta-transfer algorithm, which reduces the amount of data sent over the network by sending only the differences between the source files and the existing files in the destination. Rsync

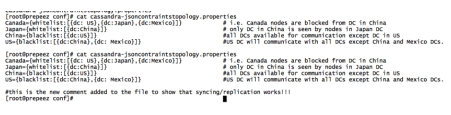
is widely used for backups and mirroring and as an improved copy command for everyday use. It designed to synchronize local changes to a file or directory with a low profile of expected changes to a remote site. The application's core is written in the C programming language and uses Lua scripting to achieve fine grained customizations.

We needed a linux sync utility because all the nodes in our cluster were running Ubuntu Operating System on the Amazon EC2 instance. The reason for choosing lsynced over other synchorinizations options was because it is a light-weight live mirror solution that is relatively easy to install, does not require new filesystems or block devices and it does not hinder local filesystem performance. It can be used for very complex synchronization scenarios but we opted to use it in the fairly straight-forward master-slave scenario. We chose one of our servers as the "master" where all changes will be made and these changes will be propagated to the rest of the cluster after a delay of 30 seconds between each sync. This approach was adopted to steer clear of potential danger we encountered during analysis. We did not want changes to be caught in an endless recursive sync loop which could occur in a peer-to-peer setup. We also added a delay between syncs to slave nodes so that we do not have a situation where the entire cluster was being restarted at the same time since each file update is updated causes an automatic refresh of the Cassandra node. Also in the case where a user makes a mistake we want to give them a small window where they can correct this mistake before the entire cluster goes down.

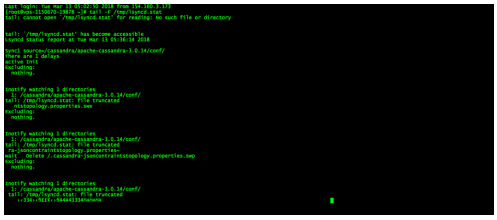
We installed lsynced version 2.1.6 which requires rsync version 3.1 or greater. We setup ssh keys so that rsync can communicate over ssh with all the slave nodes from the "master" node. For below testing we chose to demonstrate lsyncd between two nodes (master IP:168.144.85.12 slave: 136.243.33.155). These nodes were randomly selected to test how automatic syncing can be done in our cluster. Below screenshots show that rsync (version 3.1.1) and lsyncd (version 2.1.5) have been installed on the master node, syncing of files and also updating slave node. SSH-keys are used to properly setup a connection between the master node and the slave node so that passwordless login from master to slave is made possible.

Figure 4.7: Seting up rsync between master and slave nodes

The modified file is immediately copied to the slave node. Below shows the same file on the slave node before and after the modification was done on the master node and also the status file on the server node that is recording all activities of lsyncd:



(a) update of constriant file on slave node



(b) Status file on server node

Figure 4.8: Seting up rsync between master and slave nodes

4.4 Conclusion

There are various reasons why the read and write latencies are typically lower than the unmodified Cassandra trunk. Putting DCs on a blacklist means that those DCs are disabled from gossip communication and hints of other DCs. Based on the contents of the JSONConstraintsDatacenterTopologyProvider when data is written to these servers, only whitelisted servers will be able to save a copy of this data. Thus fewer servers are contacted in both read and write operations reducing the latencies. From the above, we confirm that our models work in controlling access to where replicated data is stored in the cluster.

Appendix A

Source code for selective replication stragegy

Please see below the contents of the jsoncontraintstopology.properties

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.apache.cassandra.locator;

import org.apache.cassandra.config.DatabaseDescriptor;
import org.apache.cassandra.config.Schema;
import org.apache.cassandra.db.Keyspace;
```

```
import org.apache.cassandra.exceptions.ConfigurationException;
import org.apache.cassandra.io.util.FileUtils;
import org.apache.cassandra.service.StorageService;
import org.apache.cassandra.utils.FBUtilities;
import org.apache.cassandra.utils.ResourceWatcher;
import org.apache.cassandra.utils.WrappedRunnable;

import java.io.InputStream;
import java.net.URL;

import com.google.common.base.Objects;

import java.util.Iterator;
import java.util.Properties;
import java.util.Set;
import java.util.TreeSet;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.json.simple.JSONArray;
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

/*
 * Cloud Data Management for a Peer-to-Peer Fleet Management IOT Architecture:
 * add ability to configure cassandra instances using constraints written in
    JSON format.
 * This is based on previous work done by Jeff Jirsa
 */
public class JSONConstraintsDatacenterTopologyProvider implements
   IDatacenterTopologyProvider {

   private static final Logger logger =
       LoggerFactory.getLogger(JSONConstraintsDatacenterTopologyProvider.class);

   private static final int REFRESH_PERIOD_IN_SECONDS = 60;
```

```
private Set<String> excludedDcs = new TreeSet<>();
private Set<String> dcsWithHHDisabled =
    DatabaseDescriptor.hintedHandoffDisabledDCs();
private Set<String> restoreHHDCs = new TreeSet<>();

private int lastHashCode = -1;

public JSONConstraintsDatacenterTopologyProvider()
{
    logger.debug("Initializing JSONConstraintsDatacenterTopologyProvider");
    reloadConfiguration();
    try
    {
        FBUtilities.resourceToFile
        (DatacenterTopologyProperties.DATACENTER_TOPOLOGY_PROPERTY_FILE);
        Runnable runnable = new WrappedRunnable()
        {
            protected void runMayThrow() throws ConfigurationException
            {
                reloadDatacenterTopologyProvider();
            }
        };
        ResourceWatcher.watch(DatacenterTopologyProperties.
        DATACENTER_TOPOLOGY_PROPERTY_FILE, runnable,
            REFRESH_PERIOD_IN_SECONDS * 1000);
    }
    catch (ConfigurationException ex)
    {
        logger.error("{} found, but does not conform to json format. Will
            not watch it for changes",
            DatacenterTopologyProperties.DATACENTER_TOPOLOGY_PROPERTY_FILE);
    }
}

public boolean filtersDatacenters()
{
    return true;
}
```

```
public Set<String> filteredDatacenters(Set<String> rawDatacenterSet)
{
    Set<String> filteredDatacenters = new TreeSet<>(rawDatacenterSet);
    for(String dc : rawDatacenterSet)
        if(!isGossipableDatacenter(dc))
        {
            filteredDatacenters.remove(dc);
        }

    return filteredDatacenters;
}

public boolean isGossipableDatacenter(String dcName) throws
    ConfigurationException
{
    if(excludedDcs != null)
    {
        if(excludedDcs.contains(dcName))
            return false;
        else
            return true;
    }
    else
        throw new ConfigurationException("Invalid state:
            JSONConstraintsDatacenterTopologyProvider improperly
            configured");

}

public synchronized void reloadDatacenterTopologyProvider() throws
    ConfigurationException
{
    logger.debug("JSONConstraintsDatacenterTopologyProvider.
    reloadDatacenterTopologyProvider()");
    reloadConfiguration();

    for(String dc : excludedDcs)
```

```
{
    if(!dcsWithHHDisabled.contains(dc) && !restoreHHDCs.contains(dc))
    {
        logger.info("Disabling communication with dc " + dc + ",
            explicitly disabling hints");
        DatabaseDescriptor.disableHintsForDC(dc);
        excludedDcs.add(dc);
        restoreHHDCs.add(dc);
    }
}
// For DCs where we've explicitly disabled HH, restore it once we
    gossip with that DC
for (String dc : restoreHHDCs)
{
    if (! this.excludedDcs.contains(dc))
    {
        logger.info("Re-enabling hints for previously disabled dc " +
            dc);
        DatabaseDescriptor.enableHintsForDC(dc);
        this.restoreHHDCs.remove(dc);
    }
}

if(lastHashCode != hashCode())
{
    lastHashCode = hashCode();
    // For each KS, the replication strategy may need to be
        reinstantiated in order to take advantage of
    // changes in the list of datacenters.
    for(String ksName : Schema.instance.getNonSystemKeyspaces())
    {
        Keyspace k = Keyspace.open(ksName);
        if(k.getReplicationStrategy() instanceof NetworkTopologyStrategy
            )
        {
            logger.info("Resetting replication strategy of {}",
                k.getName());
            k.resetReplicationStrategy();
```

```
            }

        }
    }
    return;
}


/*
 * Reload the configuration file from disk
 */
private void reloadConfiguration() throws ConfigurationException
{
    final DatacenterTopologyProperties properties = new
        DatacenterTopologyProperties();
    final String thisDatacenter = DatabaseDescriptor.getEndpointSnitch().
    getDatacenter(FBUtilities.getBroadcastAddress());
    String thisDatacenterConfig = properties.get(thisDatacenter, null);
    final String invalidFilterMessage = "DatacenterTopologyProvider is set
        to filter local DC " + thisDatacenter + ", this is an invalid
        configuration";
    final String invalidFormatMessage = "Configuration for " +
        thisDatacenter + " is invalid. Unable to parse JSON string";

    if(thisDatacenterConfig == null )
     this.excludedDcs = new TreeSet<>();

    else
    {

        JSONParser parser = new JSONParser();
        try{
           JSONObject jsonConstraintsConfig = (JSONObject)
               parser.parse(thisDatacenterConfig.trim());
           JSONArray constraintString;
           constraintString = (JSONArray)
               jsonConstraintsConfig.get("blacklist");
           if(constraintString==null)
```

```
{
   constraintString = (JSONArray)
       jsonConstraintsConfig.get("whitelist");
   if(constraintString==null)
   {
      logger.info("JSON configuration found for {} is neither
          blacklist or whitelist", thisDatacenter);
      this.excludedDcs = new TreeSet<>();
      //can the constraint type be neither blacklist or whitelist??
      //if yes, uncomment below line
      //throw new ConfigurationException(invalidFormatMessage);
   }
   else
   {
      logger.info("whitelist configuration found for {}",
          thisDatacenter);
      this.excludedDcs=StorageService.instance.getAllDatacenters();
//    constraintType=ConStraintTypes.WHITELIST;
      Iterator<JSONObject> iterator = constraintString.iterator();
         while (iterator.hasNext()) {
            JSONObject dc = (JSONObject) iterator.next();
            String whitelistDc = (String) dc.get("dc");
            if (this.excludedDcs.contains(whitelistDc))
            {
                 logger.debug("Loading " +
                     DatacenterTopologyProperties.DATACENTER_TOPOLOGY_
                 PROPERTY_FILE + ", removing " + whitelistDc + "
                     from set of blacklisted datacenters");
                 this.excludedDcs.remove(whitelistDc.trim());
             }
         }
   }
}
else
{
   logger.info("blacklist configuration found for {}",
       thisDatacenter);
   this.excludedDcs = new TreeSet<>();
```

```
//                constraintType=ConStraintTypes.BLACKLIST;
                  Iterator<JSONObject> iterator = constraintString.iterator();
                    while (iterator.hasNext()) {
                     JSONObject dc = (JSONObject) iterator.next();
                     String blacklistDc = (String) dc.get("dc");
                     if (!this.excludedDcs.contains(blacklistDc))
                     {
                           if (blacklistDc.trim().equals(thisDatacenter))
                             throw new
                                 ConfigurationException(invalidFilterMessage);
                           logger.debug("Loading " +
                              DatacenterTopologyProperties.DATACENTER_TOPOLOGY_PROPERTY_FILE
                              + ", adding " + blacklistDc + " to set of
                              blacklisted datacenters");
                           this.excludedDcs.add(blacklistDc.trim());
                        }
                     }
                  }
               }catch (ParseException e) {
                  throw new ConfigurationException(invalidFormatMessage);
               }
            }
         }

         public int hashCode()
         {
            return Objects.hashCode(excludedDcs);
         }

         static class DatacenterTopologyProperties
         {
            private static final Logger logger =
               LoggerFactory.getLogger(DatacenterTopologyProperties.class);
            public static final String DATACENTER_TOPOLOGY_PROPERTY_FILE =
               "cassandra-jsoncontraintstopology.properties";

            private Properties properties;
```

```
public DatacenterTopologyProperties()
{
    properties = new Properties();
    InputStream stream = null;
    String configURL =
        System.getProperty(DATACENTER_TOPOLOGY_PROPERTY_FILE);
    try
    {
        URL url;
        if (configURL == null)
            url =
                SnitchProperties.class.getClassLoader().getResource(DATACENTER_TOPOLOGY
            _PROPERTY_FILE);
        else
            url = new URL(configURL);

        stream = url.openStream(); // catch block handles potential NPE
        properties.load(stream);
    }
    catch (Exception e)
    {
        // do not throw exception here, just consider this an incomplete
            or an empty property file.
        logger.debug("Unable to read {}", ((configURL != null) ?
            configURL : DATACENTER_TOPOLOGY_PROPERTY_FILE));
    }
    finally
    {
        FileUtils.closeQuietly(stream);
    }
}

/**
 * Get a snitch property value or return defaultValue if not defined.
 */
public String get(String propertyName, String defaultValue)
{
    return properties.getProperty(propertyName, defaultValue);
```

```
        }
    }
}
```

References

[1] The internet of things. Forbes Magazine. "https://www.forbes.com/global/2002/0318/092.html", 2002.

[2] Anti Entropy. "https://wiki.apache.org/cassandra/AntiEntropy", 2013.

[3] The internet of Things, worldwide, 2013. "https://www.gartner.com/doc/2625419/forecast-internet-things-worldwide-", 2013.

[4] *Cassandra*, volume XXXIII. 2014.

[5] The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. "https://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm", 2014.

[6] The JavaScript Object Notation (JSON) Data Interchange Format. "{https://tools.ietf.org/html/rfc7159", 2014.

[7] GE Announces Predix Cloud. "https://www.ge.com/digital/press-releases/ge-announces-predix-cloud-worlds-first-cloud-service-built-industrial-data-analyti 2015.

[8] BlackBerry's Radar trailer tracking-IoT system available; Canada fleet rolls out tech. "http://fleetowner.com/technology/blackberrys-radar-trailer-tracking-iot-system-available-canada-fleet-rolls-out-tec 2016.

[9] Configuring data consistency. "https://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html", 2016.

[10] Elasticity(clould computing). "https://en.wikipedia.org/wiki/Elasticity_(cloud_computing)", 2016.

[11] Electronic Product Code. "https://en.wikipedia.org/wiki/Electronic_Product_Code", 2016.

[12] Fleet management. "https://en.wikipedia.org/wiki/Fleet_management", 2016.

[13] Gateway(telecommunications). "https://en.wikipedia.org/wiki/Gateway_(telecommunications)#IoT_Gateway", 2016.

[14] The future of agriculture. "http://www.economist.com/technology-quarterly/2016-06-09/factory-fresh", 2016.

[15] Amazon Web Services. "https://aws.amazon.com", 2017.

[16] Consistent hashing. "http://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architectureDataDistributeHashing_c.html", 2017.

[17] e-Health Sensor Platform V2.0 for Arduino and Raspberry Pi [Biometric/Medical Applications]. "https://www.cooking-hacks.com/documentation/tutorials/ehealth-biometric-sensor-platform-arduino-raspberry-pi-medical", 2017.

[18] Predix in Action. "https://www.predix.io/community/predix-in-action", 2017.

[19] Read requests. "http://docs.datastax.com/en/cassandra/2.1/cassandra/dml/architectureClientRequestsRead_c.html", 2017.

[20] Salesforce Einstein. "https://www.salesforce.com/products/einstein/overview/", 2017.

[21] Securing Cassandra. "https://docs.datastax.com/en/cassandra/3.0/cassandra/configuration/secureIntro.html", 2017.

[22] Types of Cloud Computing. "https://aws.amazon.com/types-of-cloud-computing/", 2017.

[23] rsync(1)-linux man page. "https://linux.die.net/man/1/rsync", 2018.

[24] Mohammad Aazam, Imran Khan, Aymen Abdullah Alsaffar, and Eui Nam Huh. Cloud of Things: Integrating Internet of Things and cloud computing and the issues involved. *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences and Technology, IBCAST 2014*, pages 414–419, 2014.

[25] Mervat Abu-Elkheir, Mohammad Hayajneh, and Najah Abu Ali. Data management for the Internet of Things: Design primitives and solution. *Sensors (Switzerland)*, 13(11):15582–15612, 2013.

[26] Charu C. Aggarwal. *Managing and mining sensor data*. 2014.

[27] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Data Management in the Cloud: Challenges and Opportunities. *Synthesis Lectures on Data Management*, 4(6):1–138, 2012.

[28] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, 2010.

[29] Prashant Malik Avinash Lakshman. Apache Cassandra 2.2. (July 2008), 2016.

[30] Brian Babcock and Chris Olston. Distributed Top-K Monitoring. *Sigmod 2003*, pages 1–24, 2003.

[31] Dan Boneh, Xavier Boyen, and Eu-jin Goh. Hierarchical Identity Based Encryption with Constant Size Ciphertext. *Advances in CryptologyEURO- CRYPT 2005, Springer, 2005.*, 9666(Lecture Notes in Computer Science):440–456, 2005.

[32] Francois Carrez. Internet of Things Architecture IoT-A Deliverable D1.5-Final architectural reference model for the IoT v3. (257521), 2013.

[33] Transportation U S E Case. INTERNET OF THINGS :.

[34] Nevena Golubovic Benji Lampel Varun Kulkarni Belaji Sethuramasamyraja Bruce Robers Bo liu Chandra Krintz, Rich Wolski. 2016.

[35] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.

[36] Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. An IoT gateway centric architecture to provide novel M2M services. *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 514–519, 2014.

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo. *ACM SIGOPS Operating Systems Review*, 41(6):205, 2007.

[38] Zhiming Ding, Jiajie Xu, and Qi Yang. SeaCloudDM: A database cluster framework for managing and querying massive heterogeneous sensor sampling data. *Journal of Supercomputing*, 66(3):1260–1284, 2013.

[39] Vincent C. Emeakaroha, Neil Cafferkey, Philip Healy, and John P. Morrison. A Cloud-Based IoT Data Gathering and Processing Platform. *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*, pages 50–57, 2015.

[40] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*, page 89, 2006.

[41] Katarina Grolinger, Wilson a Higashino, Abhinav Tiwari, and Miriam Am Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2:22, 2013.

[42] Robin Hecht and Stefan Jablonski. NoSQL evaluation: A use case oriented survey. *Proceedings - 2011 International Conference on Cloud and Service Computing, CSC 2011*, pages 336–341, 2011.

[43] Lihong Jiang, Li Da Xu, Hongming Cai, Zuhai Jiang, Fenglin Bu, and Boyi Xu. An IoT-Oriented Data Storage Framework in Cloud Computing Platform. *Industrial Informatics, IEEE Transactions on*, 10(2):1443–1451, 2014.

[44] Jiong Jin, Jayavardhana Gubbi, Tie Luo, and Marimuthu Palaniswami. Network Architecture and QoS Issues in the Internet of Things for a Smart City. pages 956–961, 2012.

[45] Sudarshan Kadambi, Jianjun Chen, Brian F. Cooper, David Lomax, Raghu Ramakrishnan, Adam Silberstein, Erwin Tam, and Hector Garcia-Molina. Where in the World is My Data. *Proceedings of the 37th International Conference on Very Large Data Bases*, pages 1040–1050, 2011.

[46] Kamesh and N. Sakthi Priya. Security enhancement of authenticated RFID generation. *International Journal of Applied Engineering Research*, 9(22):5968–5974, 2014.

[47] P B R Kavali. Hierarchical Attribute-Based Solution for Flexible and Scalable Access Control in Cloud Computing. 3(2):743–754, 2013.

[48] Ming Li, Shucheng Yu, Kui Ren, and Wenjing Lou. Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner settings. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, 50 LNICST:89–106, 2010.

[49] Tingli Li, Yang Liu, Ye Tian, Shuo Shen, and Wei Mao. A storage solution for massive IoT data based on NoSQL. *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*, pages 50–57, 2012.

[50] Yimeng Liu, Yizhi Wang, and Yi Jin. Research on the improvement of MongoDB Auto-Sharding in cloud environment. *ICCSE 2012 - Proceedings of 2012 7th International Conference on Computer Science and Education*, (Iccse):851–854, 2012.

[51] Phan Thi Anh Mai, Jukka K. Nurminen, and Mario Di Francesco. Cloud databases for internet-of-things data. *Proceedings - 2014 IEEE International Conference on Internet of Things, iThings 2014, 2014 IEEE International Conference on Green Computing and Communications, GreenCom 2014 and 2014 IEEE International Conference on Cyber-Physical-Social Computing, CPS 20*, (iThings):117–124, 2014.

[52] Eelco Plugge, Peter Membrey, and Tim Hawkins. *The Definitive Guide to MongoDB*. 2010.

[53] Yongrui Qin, Quan Z. Sheng, Nickolas J G Falkner, Schahram Dustdar, Hua Wang, and Athanasios V. Vasilakos. When things matter: A survey on data-centric internet of things. *Journal of Network and Computer Applications*, 64:137–153, 2016.

[54] SAS. Understanding Data Streams in IoT Contents. pages 1–6, 2015.

[55] Ichiro Satoh. MapReduce processing on IoT clouds. *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 1:323–330, 2013.

[56] Mohak Shah. Big Data and the Internet of Things. *Big Data and the Internet of Things*, page 33, 2015.

[57] P N Shankaranarayanan, Ashiwan Sivakumar, Sanjay Rao, and Mohit Tawarmalani. Performance sensitive replication in geo-distributed cloud datastores. 2014.

[58] Dieter Uckelmann, Mark Harrison, and Florian Michahelles. *Architecting the Internet of Things*. Number JANUARY. 2011.

[59] Jan Sipke van der Veen, Bram van der Waaij, and Robert J Meijer. Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual. *2012 IEEE Fifth International Conference on Cloud Computing*, pages 431–438, 2012.

[60] Guojun Wang and Qin Liu. Hierarchical Attribute-Based Encryption for Fine-Grained Access Control in Cloud Storage Services. In *CCS '10 Proceedings of the 17th ACM conference on Computer and communications security*, pages 735–737, 2010.

[61] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Achieving secure,scalable ,and fine-grained data access control in cloud computing.pdf. *Ieee Infocom*, pages 1–9, 2010.

[62] a Zanella, N. Bui, a Castellani, L. Vangelista, and M. Zorzi. Internet of Things for Smart Cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014.

[63] Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu, and Weijun Qin. IOT Gateway: BridgingWireless Sensor Networks into Internet of Things. *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 347–352, 2010.