ON BIG DATA MANAGEMENT IN INTERNET OF THINGS

A Thesis Presented to the Department of
Computer Science,
African University of Science and Technology, Abuja

In Partial Fulfilment of the Requirements for the Degree of

Master of Science

By

Mubarak Adetunji Ojewale

Abuja, Nigeria

June, 2016
ON BIG DATA MANAGEMENT IN INTERNET OF THINGS

By

Mubarak Adetunji Ojewale

A THESIS APPROVED BY THE COMPUTER SCIENCE DEPARTMENT

RECOMMENDED: ________________________________

Supervisor, Dr. Ekpe Okorafor

Head, Department of Computer Science

APPROVED: ________________________________

Chief Academic Officer

Date
The Internet of Things (IoT) has generated a large amount of research interest across a wide variety of technical areas. These include the physical devices themselves, communications among them, and relationships between them. One of the effects of ubiquitous sensors networked together into large ecosystems has been an enormous flow of data supporting a wide variety of applications. In this work, we propose a new “IntelliFog-Cloud” approach to IoT Big Data Management by leveraging mined historical intelligence from a Big Data platform and combining it with real-time actionable events from IoT devices at the Fog layer to reduce action latency in IoT applications. This approach is demonstrated through an advertisement service simulation with VoltDB technology where advertisements are being served on mobile phones based on geo-location and highest bids, and displayed from user interests determined by data analytics of activities on the web. Results from the demonstration show very low latency overhead of processing large hundreds of thousands of transactions. This approach improves both action latency and accuracy of real-time decisions in IoT applications.
ACKNOWLEDGEMENT

All praises are due to Allah alone, the Lord of the universe. May his peace and blessings continue to abide by the noble soul of his messenger Muhammad, his household, companions and all those who follow the guidance. Allah has fulfilled His promise and He has helped his slave. AlhamduliLah.

To my father, Alhaji Isa A. Ojewale, my mother, Hajiya Semiat Ojewale (nee Badmus), my brothers (AbdulWaasi, Ridwan and Ashrofs), sisters (Umu Zayd and RahmotaLlah) and all the Ojewales and Badmus, I appreciate your prayers, advices, financial and mental support before and during the course of my program. I pray Allah rewards you all abundantly.

I thank the Nelson Mandela Institute (NMI) considering me worthy of a scholarship for my Master of Science degree program out of many other qualified candidates who merited this award. Your faith in me has and will continue to yield the expected dividends.

I would like to express my deepest appreciation to my supervisor Dr. Ekpe Okorafor whom without his guidance, encouragement and persistence this thesis would not have been possible. You gave us me the needed freedom to explore a lot while giving your timely inputs at the appropriate moments. This has really helped my research skills. Thank you so much sir.

My profound gratitude goes to all the faculties in Computer Science and Engineering department especially Professor Mamadou Kaba Traore, Professor Lehel Csato, Professor Ben Abdallah, Professor Mohamed Hamada and Dr Ekpe Okorafor, my supervisor. I appreciate your constructive criticisms and the knowledge you impacted on me.
This acknowledgement would not be complete without the mention of the “5 of 5”, my Saka, Umu to be wherever she is, Imam Isa, Bro AbdulMajeed, AbdulHakeem, AbdulGaffar, Imam Usman, Ignace, Tabot, Salami, Gbade, Isun, all MSc and PhD students of the Computer Science Stream, and the entire AUST community.
DEDICATION

To Allah; the Beginning without an end; the End without a beginning
# TABLE OF CONTENTS

ABSTRACT .......................................................................................................................... iii
ACKNOWLEDGEMENT ........................................................................................................ iv
DEDICATION .......................................................................................................................... vi
LIST OF FIGURES ................................................................................................................ x
CHAPTER ONE ...................................................................................................................... 1
INTRODUCTION ................................................................................................................... 1
  1.0 Introduction .................................................................................................................. 1
  1.1 Research Question ...................................................................................................... 3
  1.2 Objective of the Research .......................................................................................... 3
  1.3 Implication of Research ............................................................................................. 3
  1.4 Scope of work ............................................................................................................. 4
  1.5 Organization ................................................................................................................ 4
CHAPTER TWO ..................................................................................................................... 5
LITERATURE REVIEW ......................................................................................................... 5
  2.0 Internet of Things (IoT) .............................................................................................. 5
    2.0.1 Why Internet of Things? ....................................................................................... 6
    2.0.2 Applications of IoT ............................................................................................... 6
    2.0.3 Challenges of IoT ................................................................................................ 7
  2.1 Big Data ...................................................................................................................... 8
    2.1.1 Big Data Management .......................................................................................... 10
    2.1.2 Big Data and Internet of Things ......................................................................... 11
  2.2 Data Streams ............................................................................................................. 12
    2.2.1 Data Stream Processing ....................................................................................... 13
    2.2.2 Stream Processing Models .................................................................................. 14
  2.3 Real-Time Data Stream Processing ........................................................................... 14
    2.3.1 Requirements of Real-time Data Stream Processing ....................................... 15
  2.4 Stream Processing Applications ............................................................................... 16
    2.4.1 Aurora ................................................................................................................ 16
    2.4.2 Borealis ............................................................................................................... 17
    2.4.3 Apache Storm ...................................................................................................... 18
    2.4.4 Apache S4 .......................................................................................................... 20
    2.4.5 Apache Samza ...................................................................................................... 22
    2.4.6 VoltDB ................................................................................................................ 23
  2.5 Related Work .............................................................................................................. 24
    2.5.1 Towards Cloud-Based Big Data Analytics for Smart Future Cities ................ 24
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>IoT Big Data Management. (Source Sullivan and Frost)</td>
<td>2</td>
</tr>
<tr>
<td>2.0</td>
<td>Internet of Things (IoT). (Source: ARM)</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>The 5Vs of Big Data. (Source: Arqano)</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Big Data and IoT (Source: Silicon Labs)</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Aurora Architecture. (Source: Abadi et. al.)</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Apache Storm. (Source: Sain Technologies)</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>Apache Storm Architecture. (Source: Business-software)</td>
<td>20</td>
</tr>
<tr>
<td>2.6</td>
<td>Apache S4 architecture. (Source: S. Kamburugamu)</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>Sample Samza Implementation Architecture. (Source: LinkedIn)</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>VoltDb Architecture</td>
<td>24</td>
</tr>
<tr>
<td>3.0</td>
<td>Fog Computing</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Multi-Tier Cloud-Fog Architecture</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Existing Approach from Cloud to Device</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>The Proposed Latency Reducing Approach</td>
<td>34</td>
</tr>
<tr>
<td>4.0</td>
<td>Intelligent Patient Monitoring System</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Intelligent Traffic Light System</td>
<td>40</td>
</tr>
<tr>
<td>4.2</td>
<td>Geo-Location and User Interest Based Mobile Advertisement display</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>VoltDb Server initialization and setup</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>VoltDb Server initialization and setup</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Top five Advertisers in the last five seconds</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Client Workload Statistics showing very low latency and high throughput</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>Server also reports low latency</td>
<td>47</td>
</tr>
</tbody>
</table>
CHAPTER ONE

INTRODUCTION

1.0 Introduction
Advances in sensor technology, communication capabilities and data analytics have resulted in a new world of novel opportunities. With improved technology such as nanotechnology, manufacturers can now make sensors which are not only small enough to fit into anything and everything but also more intelligent. These sensors can now pass their sensing data effectively and in real time due to improvements in communication protocols among devices. There are now, also, emerging tools for processing these data. These phenomena combined have made the Internet of Things (IoT) a topic of interest among researchers in recent years. Simply put, the IoT is the ability of people’s “things” to connect with anything, anywhere and at any time using any communication medium. “Things” here means connected devices of any form. It is estimated that by 2020 there will be 50 to 100 billion devices connected to the internet [2]. These devices will generate an incredible amount of massively heterogeneous data. These data, due to their size, rate at which they are generated and their heterogeneity are referred to as “Big Data”. Big Data can be defined with the famous three characteristics known as the 3Vs: volume, variety, and velocity or sometimes 5Vs, including Value and Veracity [3], [12]. These data, if well managed, can give us invaluable insights into the behaviour of people and “things”; an insight that can have a wide range of applications.

The potentials of incorporating insights from IoT data into aspects of our daily lives are becoming a reality at a very fast rate. The acceptability and trust level is also growing as people
have expressed willingness to apply IoT data analytics results in situations even as delicate as stock market trading [1]. These developments inform the need for efficient approaches to manage and make use of these huge and fast-moving data streams. Distributed processing frameworks such as Hadoop have been developed to manage large data but not data streams. One major limitation of distributed settings such as Hadoop is latency. They are still based on the traditional Store-Process-and-Forward approach which makes them unsuitable for real-time processing, a contrast with the real-time demands of the current and emerging application areas [4]. Store and forward also will not be able to satisfy the latency requirements of IoT data because of the velocity and the unstructured nature of the data. Stream processing frameworks like Apache Storm and Samza are then introduced to solve this problem. In stream processing, data from data sources are continuously processed as they arrive and do not need to be stored first. This improves latency, especially in stateless stream processing which processes data as it comes without reference to the current situation of the system.

Figure 1.0: IoT Big Data Management. (Source Sullivan and Frost)
Stream processing frameworks, however, are more general for processing data streams and are not tailored for the specific needs of IoT data management systems. IoT applications typically have strict latency requirements. IoT applications also involve a great deal of Machine-to-Machine (M2M) communications. The latency requirements of emerging IoT applications, no doubt, requires a new approach to reduce latency to its barest minimum and make fast and efficient use of “things” data.

1.1 Research Question

Can we develop a generic Big Data management approach to reduce the latency of intelligent reaction to actionable events in IoT applications?

1.2 Objective of the Research

The aim of this work is to propose and demonstrate a generic, efficient, scalable and robust approach to Big Data management approach in IoT which extracts real-time value from data and demonstrate its operation in an application area. Using existing and emerging computing paradigms, we seek to develop an approach to significantly reduce latency in streaming data from a network of connected devices and thus capture events that trigger actions in real time.

1.3 Implication of Research

This work seeks to propose a general latency-reducing approach to IoT data management independent of data source, type or communication protocol. Finding this approach will improve significantly, the speed and responsiveness of current real-time applications and also broaden the applications of IoT to new latency critical domains. The approach will reduce response time of IoT applications and enable them to react fast enough to suit the requirements
of emerging applications. It will also serve as the underlying principle of both open-source and commercial IoT data management applications.

1.4 Scope of work

The scope of this work includes the following:

I. To provide a new approach to IoT data management with a view to reduce latency.

II. To implement this approach with software tools.

III. To apply this implementation to a challenging use case.

1.5 Organization

An extensive review of literature is contained in the second chapter of this write-up. This includes review of the main concepts, technologies used as well as related work in the research area. The third chapter presents and describes the proposed model, how it works and its latency-reducing advantages. The fourth chapter describes an implementation of the model with results and evaluations and the fifth chapter contains the conclusion and future works on the subject.
CHAPTER TWO

LITERATURE REVIEW

2.0 Internet of Things (IoT)

The Internet of Things (IoT) has become a popular topic in both industry and academia. The term was originally coined by the British technologist Kevin Ashton in 1999, to describe a system where the internet is connected to the physical world via ubiquitous sensors [37]. It can simply be described as an interconnection of a massive number of objects/sensors/devices (things) through a communication network to provide value-added services [1]. IoT is a concept and a paradigm that considers pervasive presence in the environment of a variety of things/objects that through a communication medium and unique addressing schemes are able to interact with each other and with other things/objects to create new applications/services and reach common goals [26].

![Figure 2.0: Internet of Things (IoT). (Source: ARM)](image)

“Things”, in this context, are anything that can connect to any other thing over a communication medium or to the internet. They typically are recognisable and generate and communicate data to one another. IoT’s growth has been rapid over the years as more and more devices are becoming connected and application areas are increasing by the day. With an
estimated figure of nearly 50 billion devices to be connected by 2020, and the advances in data handling capabilities, the possibilities in the IoT world are endless.

2.0.1 Why Internet of Things?

The main aim of the IoT concept is to have a smart and interconnected world and to have “things” capture data without help from humans, process the data and make intelligent decisions, all by themselves. This will engineer a new automated world where we can reduce cost and waste [10]. Analysing data from so many sources can give invaluable insights about human behaviour and decision-making patterns, which is very useful in modern marketing and business intelligence. Also, there are environments that are either unsafe or impossible for humans to go into (for example an oil well) and readings need to be taken and communicated to other nodes. IoT fits well into these environments as data can be obtained analysed and instructions can also be communicated to the devices in these environments.

2.0.2 Applications of IoT

Constant events monitoring is one main application area of IoT. An example is smart cement, cement equipped with sensors to monitor stresses, cracks and warpages on a bridge continuously for maintenance purposes [27]. Other applications areas include health, where patients’ movements, heart rate, blood pressure and all other medical readings can be constantly monitored and actions taken whenever there is a need; transport, to monitor vehicular movements, fuel consumption and operational efficiency; and manufacturing, for safety monitoring and maintenance scheduling [9], [27], [28], [29].
The smart homes, smart grids, smart cars, smart farms and smart cities projects across the globe have been leading the way in IoT application areas [29], [30]. Smart home projects attempt to build applications to control all electronic devices in the home automatically and sometimes remotely. Smart home developers want to achieve a house that can switch off appliances upon detection that no one is using them; devices that can be controlled from work; or a fridge that can create a list of items that are in low supply and send it to a phone [5]. Smart vehicles can sense road conditions, distance from other cars, obstacles, possible collisions, fire, and even more to ensure a safe and automated ride [30]. They can also communicate with safety systems, traffic systems and vehicle maintenance systems.

IoT has also been deployed in supply chain management. Starting from when goods are leaving the factory, they are tracked until they get to the eventual retailer for efficiency, transparency and accountability. Smart grid projects are also concerned with efficient usage of scarce world energy. IoT is deployed to monitor the unsteady supply of renewable energy for electricity generation, balance demand and supply in energy allocation and improve the overall efficiency of the grid. Other applications of IoT include agriculture (smart farms), wearable devices, smart thermostats, oil and gas, and industrial IoT.

2.0.3 Challenges of IoT

Security and privacy have been the biggest challenges of IoT [1], [31]. Many connected devices translate to many entry points for malware. Anything connected to the internet stands a risk of being hacked; all the more so in a situation where all or almost all “things” are connected. Also, data is constantly in motion, which increases the risks of its being intercepted or hacked. Privacy is also a concern for individuals. While the acceptability of IoT is getting wider [1],
many people are still not comfortable with sensors that can track their location, health condition, online activities and transaction details. They think some of this information is too sensitive to entrust to the hands of IoT application vendors.

The lack of common standards among the very many disparate connected devices is also a major challenge for IoT applications. The fact that there are so many protocols, APIs and platforms that integrate with other platforms is becoming a major challenge; with more and more platforms coming on line [31]. Some customers also do not adopt IoT because of lack of concrete use case examples and success stories. While some organizations may want to invest based on the theoretical advantages presented, others insist on seeing a working case scenario before committing their resources – a big challenge in getting IoT applied to new application areas.

2.1 Big Data

There is no specific definition of Big Data that is agreed upon [34]. It is, however, described with some properties that are widely accepted. These properties are often referred to as the 5Vs of Big Data: volume, variety, velocity, veracity and value.

![Figure 2.1: The 5Vs of Big Data. (Source: Arqano)](image-url)
As at 2013 90% of the data in the world had been created in the previous two years [32]. Interestingly, also in 2015, 90% of the data in the world has been created in the previous two years [33]. This shows the geometric growth in the volume of data generated due to various digital activities. In 2013, 28,875 GB of data was generated per second. With the current growth rate, this figure is expected to rise by almost 100% (50,000 GB) by 2018. The above statistics just describe one characteristic of the Big Data – volume. The volume of data to be processed is undoubtedly big; much more serious is the fact that it will only keep growing.

Regarding velocity, the rate at which data is being generated is unprecedented. As at 2014, every minute, there were 2.5 million new Facebook posts, 300,000 tweets, 220,000 new photos on Instagram, 72 hours of new videos on YouTube and over 200 million emails [35]. Considering the growth rate in the internet penetration, one can only imagine the number at present. Combining this with data generated by other devices, websites, sensors and all other sources of data, the velocity of data is very high at present and, as the case for volume, it will only grow.

Traditionally, data is usually structured and data sources are not very disparate. Now, terms like “unstructured” or “semi-structured” data are commonplace and they form a very big part of Big Data. About 2.5 quintillion bytes of data are generated in the world per day from unstructured data sources like sensors, social media posts and digital photos [36]. Big Data contains not just the usual text files; it also contains videos, pictures, sensor data, web logs, tweets, emails, Facebook posts and CCTV footage. The disparate data sources and forms in which data are generated explain the variety of attributes of Big Data. The other two attributes
– veracity and value – have to do with processing of Big Data. Veracity describes quality and how understandable data is, while value describes the economic importance of the data.

2.1.1 Big Data Management

There is no reason to gather so much data if it will not be of any use. But dealing with large, massively heterogeneous and rapid new datasets is not “business as usual”. Collecting, organizing, storing, securing and extracting useful insights from large volumes of structured and unstructured data is a challenge to both academia and industry. Data management is a broad concept referring to the architectures, practices and procedures for proper management of the data lifecycle needs of a certain system. In the context of IoT, data management should act as a layer between the objects and devices generating the data and the applications accessing the data for analysis purposes and services [14].

The volume of data to be managed has outgrown the storage capacity of traditional data storage and analytics solutions. Also, data is generated in real time and insights from data are also demanded in real time. Traditional data management systems cannot scale up to the velocity of the Big Data. The variety of data sources also necessitates a shift from the traditional data management approaches to a new approach that will be tailored to suit the unique characteristics of Big Data. Traditional data management approaches cannot deal with the heterogeneity and variable nature of Big Data, which comes in formats as different as emails, social media data, videos, images, blogs and sensor data as well as “shadow data” such as access journals and web search histories [38]. Proper Big Data management can empower businesses to automate more business processes, operate near real time, and through analytics, learn valuable new facts about business operations, customers, partners, and so on [39].
According to a survey carried out in [39], the Apache Hadoop Framework tools will be the software products most aggressively adopted for Big Data Management in the next three years (referring to 2014-2017). Apache Hadoop is thought to be the best new approach to unstructured data analytics for now. Hadoop is an open-source framework that uses a simple programming model to enable distributed processing of Big Data on clusters of computers. The complete Hadoop technology stack includes common utilities, HDFS, analytics and data storage platforms and an application layer that manages distributed processing, parallel computation, work flow and configuration management [38].

Other emerging popular tools include complex event processing (like SQLstream and Apama) for streaming Big Data, NoSQL databases (like Cassandra) for schema-free Big Data, in-memory databases (for example VoltDb) for real-time analytic processing of Big Data, private clouds, in-database analytics, and grid computing [39]. Another fast-emerging practice in Big Data management is Fog computing; an emerging computing model that brings data management closer to the edge of the network.

2.1.2 Big Data and Internet of Things

A major application of the IoT is in sensing and monitoring. Continuous sensing and monitoring implies rapid and continuous generation of unstructured data. Fifty billion devices are expected to be connected by 2020 [2]. Even if each device generates an average of 1 byte per second, IoT will be generating about 50 GB of data per second by 2020. IoT is, no doubt, playing a significant role in the growth of both the volume, variety and velocity of data. With continuous data streams from “things”, data centres have to handle this additional load of
unstructured, heterogeneous data. This has had effects on Big Data infrastructures as most companies now outsource data storage on cloud services [40].

Automated sensor data is projected to soon overwhelm the data available from more traditional human-centred sources [41], i.e. they will be the main source of Big Data. Dealing with fast ingestion rate of IoT data is a major concern in Big Data collection. Also, the real time requirements of IoT applications have made scalability, distributed processing, and real-time analytics critical functionalities of any Big Data management environment. Usually, Big Data analytics involves batch processing of a large collection of unstructured datasets. But with IoT applications’ real-time requirements, Big Data analytics solutions now have to process and act on data in real time or near real time. Data need to be processed in motion and not the store-process-forward approach any more. IoT has in fact pushed all fronts of Big Data management; from data collection, storage and processing of data.

2.2 Data Streams

Traditional database usually contains records arranged in columns and rows and are typically not concerned with the time of arrival of these data except when the time stamp is used.

The Database Management Systems (DBMS) for these databases are basically concerned with insertions, deletions, updates and queries. Emerging applications (IoT especially), however, have pushed the limits of the traditional databases. Information naturally occurs in the form of a data streams; examples include sensor data, internet traffic, online auctions, and transaction logs (such as web usage logs and telephone call records). A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by time stamp) sequence of items [18]. A data stream consists of an unbounded sequence of tuples with fixed schemas.
2.2.1 Data Stream Processing

Data streams cannot be handled by the traditional DBMS. This is mainly because of the fact that it is practically impossible to store data streams locally in their entirety and that queries on data streams typically run continuously (stateless) or over a window (stateful). The unique characteristics of data streams and continuous queries necessitates data stream processing solutions to have requirements such as order-based and time-based operations; use of approximate summary structures (synopses), since we cannot store data streams; shared execution for scalability; adaptation to different conditions; and real-time reaction to outlier values [18, 19]. Data stream processing is the in-memory, record-by-record analysis of data streams in motion with the objective of extracting actionable intelligence as streaming analytics, and to react to operational exceptions through real-time alerts and automated actions in order to correct or avert the problem [22]. Modern stream processing engines have high requirements in terms of scalability, fault tolerance and latency. They differ from traditional
stream processing engines mainly in their architecture, (master-slave), data pattern (publish and subscribe) and highly parallel execution [25].

2.2.2 Stream Processing Models

Stream processing models process data while still in motion (before and/or without storing). A stream processing system is usually a network of processing units called processing elements (PEs). These PEs receive tuples and perform some computations on these tuples and then produce results in the form of decisions or other tuples. A stream processing engine creates a network of PEs connected in a directed acyclic graph (DAG). The vertices of the graph represent the PEs and the edges represent the tuple flows. Each PE is independent and only communicates with other PEs via messaging. This communication can either be pull-based or push-based messaging [42]. The distribution, load balancing and management of PEs may not be the same in different implementations.

2.3 Real-Time Data Stream Processing

Applications based on the traditional store-first, process-second data management architectures are unable to scale for data streams applications largely due to the velocity and volume of data streams. Even Hadoop-based systems are unable to offer the combination of latency and throughput requirements for real-time data stream applications in industries such as e-commerce, telecoms, IoT and financial institutions [22]. Data stream processing therefore needs to meet the real-time requirements of these emerging applications.

Fast-moving data from emerging applications are most valuable at the time they arrive at the data stream processing system [21]. It is not so useful detecting a potential buyer after the user leaves an e-commerce site, or detecting theft after the burglar has absconded [20]. Another
perspective may be a social network that wants to detect trending conversation topics in minutes; a search site may wish to model which users visit a new page; and a service operator may wish to monitor program logs to detect failures in seconds. Hence the need to process and get value from data streams in real time is paramount. The definition of real time, however, depends on the latency constraints of the application.

![Real-time Stream Analytics](image.png)  
*Figure 2.3: Real time Stream Analytics. (Source: LinkedIn)*

### 2.3.1 Requirements of Real-time Data Stream Processing

Apart from the low latency, high throughput requirements, Stonebraker [23] provided one of the most comprehensive requirement analysis for Real-Time Data Stream Processing. Stonebraker listed eight requirements which encompass most others found in other works. He argued that data must be kept moving (processed on the go without storing first) to achieve low latency. This, he said is the first requirement of real-time stream processing. The second requirement is that the stream processing framework should provide an SQL or SQL-like query language for processing streaming data.
Handling data losses, delays and out-of-order delivery is the third requirement, according to Stonebraker. The stream processing engine should also be predictable and repeated processing over the same data set should give the same results. There should also be an integration between stored data processing and streaming data processing – a mechanism to process both information from the “past” to the “present” at the same time. Data security and availability is also a vital requirement and integrity of data should be guaranteed at all times, even in time of failure. Scalability and distributed processing are also key requirements for stream processing frameworks to handle large and growing data streams and to increase the speed of processing. Lastly, he mentioned real-time response as the last requirement of real-time stream processing. Stream processing should be optimized and done with such low latency that response will be in real time.

2.4 Stream Processing Applications

A review of some already existing and popularly used stream processing applications is presented below.

2.4.1 Aurora

Started as a project in Brown University, Brandeis University and MIT, Aurora is one of the early stream processing applications. It was designed to handle three broad application types in a single framework; real-time applications, archival applications (interested in the past), and spanning applications (interested in both past and present) [43].
Architecture

Aurora’s architecture is very close to the generic stream processing model. It also uses a DAG with boxes and vertexes. The boxes represent the PEs and the vertexes represent the tuple flow. The communication model in Aurora is pull based. Tuples arrive at the queues of a box processor and a scheduler selects which boxes to run. Each box is a stream operator specified in the Aurora query model. Data flow between boxes is in the direction of the edges of the DAGs. Aurora, however, was designed as a single-site stream processing engine and not distributed. Features like fault tolerance were also absent. But by combining Aurora with Medusa, another MIT project stressing distributed processing infrastructure for stream processing, distributed processing and even fault tolerance can be achieved.

![Aurora Architecture](image)

**Figure 2.3:** Aurora Architecture. (Source: Abadi et. al.)

2.4.2 Borealis

Sometimes described as an improved version of Aurora, Borealis [44] is a distributed stream processing engine developed by the same collaborating universities that developed Aurora. It sets out to solve some identified problems in Aurora and other existing stream processing applications at the time. It also uses the DAGs network and adds a few advanced functionalities to take care of the identified shortcomings of existing applications at the time [42]. Dynamic load balancing is one of the main issues addressed in Borealis due to the nature of data streams. They sometimes experience bursts and variations over time scales [44]. Other identified and
addressed problems include dynamic query results revision – if there is an update in processed data – and dynamic query modification to edit queries at runtime.

**Architecture**

Borealis architecture is fully distributed. There is no central controller. It has several interconnected servers running at different locations. These servers, since they are connected, take coordinated actions. Each server has a query processor and an admin module that determines where a query should be run locally or remotely. Each query processor has a box processor, like that of Aurora, that runs individual PEs, a load shedder for load balancing operations, a storage manager for storing streams and a priority scheduler that determines execution order [42]. The data communication system in Borealis is also pull based. The Borealis project is no longer active as the last version was released in 2008 [44].

### 2.4.3 Apache Storm

Storm, originally built by Twitter, is an Apache open-source project. It is a distributed real-time computation system for processing large volumes of high-velocity data in parallel, at scale and in real time.

![Apache Storm](Source: Sain Technologies)
Storm has a robust fault tolerance system and guarantees that data is processed at least once. It is scalable and very fast also; a benchmark clocked it at over a million tuples processed per second per node. Storm is very popular in stream processing and sometimes envisaged as doing to stream processing what Hadoop did to Big Data [46].

**Architecture**

A storm cluster has three sets of nodes: Nimbus nodes, Zookeeper nodes and the Supervisor nodes [42]. Nimbus is the main server where a user program is submitted for execution. Nimbus then distributes this code among worker nodes for execution. A Worker node is composed of workers. Each Worker contains a variable number of Slots. A Slot is an available computation unit which has dedicated CPU resources [46]. Fault tolerance is achieved by keeping track of the progress of Worker nodes so that failed computations can be restarted or relocated to other nodes; this is also done by Nimbus [42].
A Supervisor is a daemon run by the set of worker nodes in a Storm cluster. The coordination between the Supervisor nodes and Nimbus happens through the Zookeeper node. Storm jobs is being represented as user-defined work flows, denoted as topologies. These topologies are also work flows but vertices belong to two main categories: Spouts and Bolts [46]. The Spouts are simply the data entry points. They receive data from one or more data sources and they transmit data to Bolts. Bolts are execution units for user-defined operations considered to be atomic. Storm does not support stateful operators and thus does not support window incremental processing, but they can be added through an API extension [46].

2.4.4 Apache S4

Started by Yahoo! and later donated to Apache, S4 is a general-purpose, distributed, scalable, partially fault-tolerant, pluggable platform that allows programmers to easily develop
applications for processing continuous unbounded streams of data [47]. S4 stands for Simple Scalable Streaming System. S4 is fully distributed and performs processing in real time. It employs the Actor model for computations [42]. This processing model is based on the MapReduce concept.

![Apache S4 architecture. (Source: S. Kamburugamuv)](image)

**Architecture**

The basic processing units of S4 are called PEs. These PEs are arranged in a dynamic network with a DAG layout at runtime. A runtime instance of a PE is identified using the PE’s code and configuration, key attributes for the events, the events consumed by the PE, and value of the keyed attributes [42].

Each PE instance has its own unique key value attributes except that the PEs take input into the S4 framework, like the Spouts in Storm. Dealing with key attributes of very large value domains can be problematic, however, as it creates an unnecessarily large number of PEs. PEs are hosted by Processing Nodes which also perform the hashing key value attributes.
Coordination between nodes is achieved through the Zookeeper. The “partial” fault tolerance of S4 is also achieved through the Zookeeper. If the Zookeeper detects failure, it distributes the tasks of that PE to other PEs, but there are no message delivery guarantees.

2.4.5 Apache Samza

Apache Samza, now also an Apache incubator project, was developed and open sourced by LinkedIn to meet the stream processing needs of the company. Samza’s goal is to provide a lightweight framework that helps in building applications to process continuous streams of unbounded tuples [48]. A Samza stream is an immutable unbounded collection of messages of the same type. Messages can be added to or deleted from streams. A logical collection of processing units that act on a stream and produce output is called a Job in Samza [42]. A Job is the user program that consumes and processes a set of input streams. A Job creates a network topology that processes messages like the runtime DAG in S4. Streams are partitioned into sub-streams and distributed across the parallel processing tasks. A task is the processing element in a Samza network like the PEs of S4. Samza has a good fault tolerance system, providing at least one message delivery guarantee. It also has inbuilt support for stateful streaming.

Architecture

Samza’s architecture consists of three components:

1. A streaming layer: responsible for providing partitioned streams that are replicated and durable
2. An Execution layer: responsible for scheduling and coordinating tasks across the machines
3. A processing layer: responsible for processing the input stream and applying transformations [48].
It typically uses Apache Kafka, a distributed messaging application, for the streaming layer. It also uses the Apache YARN for the execution layer while the Samza API sits at the processing layer.

![Sample Samza Implementation Architecture](image)

*Figure 2.7: Sample Samza Implementation Architecture. (Source: LinkedIn)*

### 2.4.6 VoltDB

VoltDB is the in-memory NewSQL database built to solve the problem of managing Fast Data (Streams) generated from emerging data sources. Unlike other streaming solutions and data collectors, VoltDB focuses mainly on the requirements of Fast Data with unlimited scalability, the ability to perform real-time data collection, exploration, analysis and taking action all with the transactional consistency (ACID) of traditional relational databases [49]. It also has a good fault tolerance system, recovery system in time of failures and SQL query support.
2.5 Related Work

Much work has been done with respect to IoT data management but, in this section, we focus on the ones related to data, analytics and performance. We briefly summarize some of the related works.

2.5.1 Towards Cloud-Based Big Data Analytics for Smart Future Cities

Khan et al. [13] proposed a cloud-based data management and analytics service for a cloud cities application. They proposed an architecture that provides basic components to build necessary functionality for a cloud-based Big Data analytical service for smart cities data. This architecture was implemented in both Hadoop and Apache Spark on the Bristol open data. The results show that Hadoop incurs more overhead, especially in job submission, than Spark and that Spark is more appropriate for the Bristol open data. The part of the open data analysed is the data about quality of life measured by indicators such as crime rate, security, etc. Their
work is an important contribution to the smart cities projects, a very popular application of IoT. Their work, however, does not look into real-time analytics.

2.5.2 A Data-Centric Framework for Development and Deployment of Internet of Things Applications in Clouds

Khodadadi et al. [8] proposed another cloud-based data-centric framework for development and deployment of IoT applications. The framework architecture manages data collection, filtering and load balancing from different sources, producing both structured and unstructured data. It was demonstrated with an application to compute tweet sentiments of the six biggest capital cities in Australia. This application was built on top of the Aneka Cloud Application Platform. This framework focuses on data collection and its abstraction from developers and it is an important contribution towards finding a generic approach to IoT data collection management. It, however, does not deal with data analytics.

2.5.3 A CIM-Based Framework for Utility Big Data Analytics

Zhu et al. [11] proposed a framework to make disparate and incompatible datasets usable, interoperable and valuable across the enterprise. This framework, though not specifically for IoT, enables data from disparate sources to be more “homogeneous” by proposing a Common Information Model (CIM) standard for information interchange between data sources. This approach will require data sources to use the CIM adapters; a condition many IoT devices may not be willing or able to meet. This approach is not also concerned with performance as it is mainly concerned with making data more homogeneous from the source for easy analytics. It is also specifically designed for the utility industry and not a general framework for Big Data analytics.
2.5.4 Data Management for the Internet of Things: Design Primitives and Solution

Abu-Elkheir et al. [14], having investigated the issues of IoT data management, also introduced a model framework for IoT data management. This model provided layered stages of data management. The work proposed six layers of IoT data management including a “things layer”, communication layer, source/data layer, federation layer, query layer and application layer. Their work also does not involve any specific handling of real-time processing and optimization issues. It focused mainly on how data is collected, stored and processed. The model is also focused on batch processing.

2.5.5 An Architecture to Support the Collection of Big Data in the Internet of Things

Cecchinel et al. [15] also developed an architecture support for a data collection framework in IoT. This framework handles IoT Big Data issues such as sensor heterogeneity, scalability data and even reconfiguration capability. The use was also used demonstrated on sample projects. The work, however, focuses on the data collection part and does not look into the data processing. Also the framework handles data aggregation at the bridge to make data arrive at the middle-ware in a unified format. Their approach will work fine if the IoT data management application is location specific and data always arrive through the bridge. It may not work when data sources are widely dispersed and data come in directly through the internet without passing the bridge.

2.5.6 Lambda Architecture

The Lambda architecture [17] was proposed by Nathan Marz. The architecture is generic for large stream processing, giving the options of real-time processing and batch processing concurrently. In the Lambda architecture, incoming data is dispatched to both the batch layer
and the speed layer for processing. The batch layer manages the master dataset (an immutable, append-only set of raw data), and also pre-computes the batch views. The service layer indexes batch views for low latency, ad-hoc queries and the speed layer deals with recent data and real-time analytics only. Incoming queries can be typically answered by merging the results from batch views and real-time views.

The Lambda architecture received some criticisms, however, especially in terms of its complexity [59]. Some others criticized the redundancy in implementing almost identical processes in both layers. Lambda architecture is also criticized for the one-way data flow and immutability of data. Lambda’s shortcomings also include its inability to build responsive, event-oriented applications.

2.6 Chapter Summary

The chapter describes the concept of the IoT, Big Data and the intersection between the two. It also discusses Big Data management in IoT. Data streams, and real-time data streams processing were also discussed in detail. The chapter further discusses the requirement for stream processing frameworks and reviews some of the existing frameworks. It finally looked at the works of other authors related to this work.
CHAPTER THREE

ANALYSIS

3.0 Latency

The question of how real is “real time” is often answered by how long getting a response can take before data become obsolete, start depreciating in value or even become useless. Latency refers to the time to get a computation/communication request satisfied. Examples are the time to confirm that a debit card transaction is not fraudulent or the time to recommend another article to a user on a website. Detecting a fraud after the transaction has been completed is of no use, and also taking 50 seconds to confirm that a transaction is not fraudulent will put off most customers. Also, detecting what a visitor to a website might like to read after he/she might have left the website is useless and taking too much time to load a page because of that can also discourage customers from visiting. There are more critical systems like stock trading where milliseconds latency improvements can mean gains in millions of Dollars for companies. Latency requirements of IoT systems are typically strict. Constant monitoring systems especially need to react to actionable events at near zero latency.

3.0.1 Types of Latency

Hackathorn [59] identified three types of latency:

- Data latency: the time taken to collect and store the data
- Analysis latency: the time taken to analyse the data and turn it into actionable information
- Action latency: the time taken to react to the information and take action.
Streaming in itself has reduced data latency to a great extent since data are always in motion and not stored before processing. Data are no longer stored before processing; rather they are processed while data are still in motion. Improvements in communication technologies such as the LTE have also contributed to the reduction of data latency. Distributed stream processing engines can now process millions of tuples per second.

There is, of course, still the need for improvement in analysis and data latency and they are active areas in research. Of the three identified latency types, action latency is most critical to IoT as “things” most of the time do not only send data but also do need to receive feedback to act on the information taken. Reducing action latency, however, requires developing a new approach to compute and communicate feedbacks to “things” at near zero latency.

3.1 Fog Computing

One can think of cloud computing as a manufacturing company (an application) where all customers have to buy goods (access services) from the factory (cloud) alone. Fog computing would be the company deciding to open retail outlets at locations close to the customers and the retail stores will give a level of access to the customer; if not exactly, it will be close to what they can obtain from the factory. The Fog computing paradigm extends cloud computing and services to the edge of the network; close to the devices. It is a hierarchical distributed platform for service delivery providing computation, communication and storage at a layer that is much closer to the devices than the cloud.

Fog computing is described as a highly virtual platform that provides compute, storage and networking services between end devices and traditional cloud computing data centres, typically, but not exclusively located at the edge of network [62]. Fog computing is not meant
to replace or cannibalize cloud computing; rather, it is to complement it in applications where
the traditional cloud computing may not be suitable. Examples of such applications include
deo-distributed applications, large-scale distributed control systems, and applications with very
strict latency requirements [60]. It is also not a trivial extension of cloud computing. It has
some distinctive characteristics that make it more than just an extension.

These characteristics according to Bonomi [62] include:

- **Edge location, location awareness and low latency:** The idea of supporting endpoints
  with rich services at the edge of the network is said to be one of the origins of Fog
  computing. This also has to do with applications with strict latency requirement.

- **Geographical distribution:** Unlike the more centralized cloud, the services and
  applications targeted by the Fog demand widely distributed deployments. This is a key
  feature in Fog computing.

- **Very large number of nodes:** this is as a consequence of the wide geo-distribution.
  Applications such as smart connected vehicles and traffic system and smart grids would
  typically require a very large number of nodes.

- **Support for mobility:** It is essential for many Fog applications to communicate directly
  with mobile devices (phones or wearable devices), and therefore support mobility
  techniques.

- **Real-time interactions:** Fog applications involve real-time interactions rather than batch
  processing. In fact, one of the underlying reasons of the Fog approach is real time
  interactions.

- **Predominance of wireless access:** Access is predominantly wireless over a network and
  may or may not need internet communication.
• **Heterogeneity:** Fog nodes come in different forms, and will be deployed in a wide variety of environments.

• **Interoperability and federation:** Fog components must be able to interoperate, and services must be federated across domains.

• **Support for online analytic and interplay with the Cloud:** The Fog is positioned to play a significant role in the ingestion and processing of the data close to the source.

---

**Figure 3.0: Fog Computing**

3.2 **Multi-Tier Fog-Cloud Architecture**

Multi-tier Fog-Cloud Architecture was proposed by John et al. in [61]. It is a product of superimposing Fog computing on cloud computing resulting in a three-tier distributed computing architecture with the “Fog Servers” serving as the near-end computing proxies between the front-end devices and the far-end servers. The first tier of the architecture, called the front-end basically consist of “things”; devices generating data and transmitting data. The
second tier, called the near-end servers, consist of the Fog Servers. These Fog Servers have some storage and processing capabilities which front-end devices can access while incurring minimal amount of communication latency. The third tier consists of the far-end servers. These are cloud infrastructures for batch and near real-time analytics and Big Data management. The main objective of the architecture is to optimize communication and computation efficiency; which makes it suitable in meeting the IoT latency requirements.

Figure 3.1: Multi-Tier Cloud-Fog Architecture

3.3 Analysis of the Existing/Traditional Approach

3.3.1 From Devices to Cloud

The early approach to managing IoT data is for the device to send data directly to the cloud via a communication medium, in real time. Big Data analytics are also done at the cloud layer and decisions, where applicable, are communicated to the devices from the cloud to the devices via the communication medium also. IoT applications are typically highly distributed and the
velocity of data is high. This can cause congestion on the network and communicating actionable events and decisions can be a challenge. An actionable event can be caught in a busy channel and can be delayed with other messages. This is a major contributing factor to action latency. Also, even when the channel is free, the distance between the cloud layer and the devices is usually too wide. Processing actionable events at a layer closer to the devices was suggested and that introduced the Fog concept to IoT data management.

The Fog layer enables distributed processing at the edge of the network; very close to the devices.

Figure 3.2: Existing Approach from Cloud to Device
3.3.2 Existing Fog Approach

The Fog approach reduced latency significantly and brought processing close to the edge of the network. It is also sometimes used for dynamic real-time load balancing. Processing of geo-sensitive data is also easier and data is aggregated before transmission to the cloud layer for further processing. However, the existing Fog approach does not provide a mechanism for making intelligent decisions in real time while taking mined intelligence from Big Data analytics into consideration. Decisions are typically based on the actionable events alone and/or the real-time state of the system.

3.4 The Proposed Latency-Reducing Intelli-Fog Approach

![The Proposed Latency Reducing Approach](image)

*Figure 3.3: The Proposed Latency Reducing Approach*
3.4.1 The intelli-Fog Layer

The intelli-Fog Layer of the proposed approach differs from the existing Fog approaches in that it is capable of making more intelligent decisions; without any significant latency increase. This layer communicates both with the IoT devices as well as the cloud layer where data analytics is performed. Communication with devices involves taking in raw data/actionable events that are observed or reported by IoT devices. It also involves communicating decisions/actions taken at on it (the cloud layer) back to the IoT devices. This is done in parallel with communication with the cloud layer. The communication with the cloud layer is also a two-way communication. Aggregated data are sent to the cloud for storage and analytics and mined intelligence from these historical data are sent back to the intelli-Fog layer. Other information necessary for decision making can also be sent to the intelli-Fog layer.

3.4.2 The Cloud Layer

The cloud layer is not too different from the traditional cloud layer. Data storage, and analysis still take place in the cloud layer. The only difference is that decisions making has been moved from the cloud layer to the intelli-Fog layer and that data are sent to the cloud layer after aggregation. Aggregation reduces the number of data to be transmitted and stored and provides a sort of filter to ensure that only useful data is transmitted and stored.

3.5 Chapter Summary

This chapter contains review of latency and the different IoT data management architecture approaches. The existing system and its benefits as well as the problems associated with it were
discussed. It contains a description and analysis of the existing approaches as well as a full detailed analysis of the proposed approach in this work.
CHAPTER FOUR

USE CASES AND IMPLEMENTATION

4.0 Introduction
In this chapter, some use case scenarios of the proposed approach are presented. One of the use cases is also implemented. The use case, tools and programming languages used are discussed. The implementation and screen-shots from the latency benchmark readings are also presented.

4.1 Use Case Scenarios
In this section, we present some use case scenarios of the new approach proposed in the previous chapter.
4.1.1 Intelligent Patient Monitoring System.

Figure 4.0: Intelligent Patient Monitoring System
A patient monitoring machine will sound an alarm by sensing an abnormal reading on patient data, but the alarm cannot state the likely problem that the health care officer should look out for immediately. With the proposed system, a patient monitoring system can be made more intelligent to make these decisions based on some Big Data analysis on a patient’s medical history as well as other available relevant medical records to predict likely complications to watch out for. An intelli-Fog layer can also go further to communicate this reading to the patient's doctor or a doctor on duty; and communicate the patient's necessary medical record too for speedy, accurate and efficient health care service.
4.1.2 Smart Cities (Intelligent Traffic Light)

Figure 4.1: Intelligent Traffic Light System
In this use case, smart traffic lights serve as the intelli-Fog layer of a traffic management system. The smart traffic light senses both vehicles and pedestrians on the road. It monitors the state of each lane and uses the M2M communication protocol with other smart traffic lights to get the state of neighbouring devices. It aggregates these data and then sends them to a central cloud layer for data analytics. The cloud layer sends back information on likely busy lanes and periods and suggestions on how to pass cars in each lane. This information, together with the state of other lanes is used by a smart traffic light to pass cars on its lane.
4.1.3 Geo-Location and User Interest Based Mobile Advertisement display

Figure 4.2: Geo-Location and User Interest Based Mobile Advertisement display
A geo-targeted advertisement display system on a web browser or mobile phone can determine which advertisement to serve based on the highest bidder and location in real time. Digital marketing these days seek better efficiency in terms of serving advertisements that are relevant to the user. So, adding user interests, based on users’ activities on their devices and/or the web pages they visit, will enable a more efficient digital advertising. The intelli-Fog approach can help achieve this with the web browser/mobile phone serving as the intelli-Fog layer. Here, data of user activities are aggregated and sent to the cloud Big Data analytics system. The cloud layer, in turn, returns user interest categories. This information, together with user location information and price of advertisement bids, are then used in serving advertisements.

4.2 Use Case Implementation (Intelli-Fog Advertisement Display Based on Location and User Interest)

The intelli-Fog layer of the third use case scenario in the previous section was implemented to demonstrate the proposed approach. The implementation simulates how advertisements are served on mobile phones in real time based on customer location. There are several threads simulating advertisement bids, advertisement requests, and the benchmark threads monitoring and reporting performance of the system in terms of latency. In this program, advertisements are displayed based on location, the highest bidder and also user interests. The user interest, although hard-coded in this program, is assumed to be obtained from data analytics on user activities on the web. Also, expired bids are deleted from the VoltDB memory in the geospatial app. In this implementation expired advertisement bids are exported to a file where they will be imported into Hortonworks Hadoop for analytics so advertisers can have an insight into how their bids are faring.
4.2.1 Technology and Tools

**VoltDb**

VoltDB is the in-memory NewSQL database built to solve the problem of managing fast data (streams) generated from emerging data sources. VoltDB focuses mainly on the requirements of fast data with unlimited scalability, the ability to perform real-time data collection, exploration, analysis and taking action all with the transactional consistency (ACID) of traditional relational databases [49]. The choice of VoltDb for demonstration of this approach is mainly because of its superior latency claims in terms of performance; even above popular streaming frameworks like Storm. While most of these claims can be traced to authors from the VoltDB community themselves, the choice was made due to the absence of counter-claims in literature about VoltDB's performance. Also VoltDB allows per-event decision making and its in-memory feature removes completely disk read/write latency.

**Java**

Java is a powerful computer programming language that is fun for novices and appropriate for experienced programmers to use in building substantial information systems. It emphasizes achieving program clarity through the proven techniques of object-oriented programming. This application is based on Oracle’s Java 2 Platform, Standard Edition (J2SE). Oracle provides an implementation of this platform, called the J2SE Development Kit (JDK) that includes the minimum set of tools needed to write software in Java.
Hortonworks Data Platform

The Hortonworks data platform is an open-source data management package that contains most of Apache’s open-source Big Data and stream analytics software. It runs on a virtual system (Virtual Box or VMWare) using the Ambari server.

![Figure 4.3: VoltDb Server initialization and setup](image)

### 4.2.2 The Developed Advertisement Display Simulation

The advertisement display simulation is a modification of an inbuilt VoltDb “geospatial” app. This app was adapted to the intelli-Fog layer for the Advertisement Display model. The major changes included were the database structure of the advertisers table. Each advertiser row was modified to include a category from a range of categories representing user interests. Accordingly, advertisement bids and advertisement requests are generated with categories. These categories are used, together with geo-location data, to serve advertisements.
The application initializes by loading configuration settings in the `AdBrokerBenchmark.java` file. These include information like duration of running, display interval, server name, the time when a bid will expire etc. The application then connects to VoltDB in-memory database at this point. Once the initialization is successful, a Java class generates advertisement requests and another Java class generates advertisement bids. Advertisement bids and requests are processed at a throughput of about an average of 110,000 transactions per second.
At intervals of five seconds, the app displays the top five advertisers, their category as well as revenue for the time interval. In the example above, “Bright’s Wonderful Foods” in the food category is the top advertiser for a consecutive five seconds interval gaining $6.03 and $2.58 consecutively.

After the advertisement bids and advertisement requests have stopped, the latency benchmark is evaluated. It gives the client’s workload statistics in terms of throughput and latency.

![System Server Statistics](image)

*Figure 4.6: Server also reports low latency*

In this session, the average throughput was at about 112,735 transactions per second and the average latency was 24.67ms. The benchmark also gives the latency breakdown of some stages of execution. Percentile latency is reported at 10th, 20th, up to 99.9th percentile. The system server also reports an average latency of 6.94ms.

From the implementation, the average latency of running about 112,000 transactions per second is 24.67 ms. This implies that more relevant geo-location based advertisements can be served to users in real time with a latency of 24.67 ms which cannot be noticed by the user. The additional latency of analysing the user’s interest and communicating advertisement requests to the cloud before advertisements are served have both been eliminated. These communication and computation latency can add significant latency which will disrupt the real-time experience of the user. Since the intelli-Fog layer is distributed, it is scalable and latency will not increase with more devices joining the advertisement display system.
4.4 Chapter Summary

The chapter contains discussions on use case scenarios of the proposed model in this work. It also contains discussion on the implementation of one of the use case scenarios. The use case implemented was the advertisement display scenario where advertisements are served to users based on location and the user interests. All expired advertisements are also exported back to the cloud layer for (Big) data analytics and intelligence mining. The implementation and screen-shots of latency benchmarks were also included in the chapter.
5.0 Summary

This work focuses on the reducing the time it takes to react to actionable events in IoT applications. It investigates how real-time value can be obtained from the Big Data typically emitted by IoT application. The work presents an extensive literature review of IoT and Big Data, as well as the relationship between the two concepts. It also discussed data streams and existing stream processing frameworks. The work also looked into related work by other authors as well as the existing approaches in reacting to actionable events from IoT applications. This work finally presents a new latency-reducing approach to processing Big Data and actionable events in IoT applications; illustrates use case scenarios and also implements a particular use case.

5.1 Conclusion

A new latency-reducing approach of processing Big Data in a way to react to actionable events under strict latency requirements has been proposed in this work. Fog computing paradigm, which has been widely advocated for IoT applications, is leveraged to reduce action latency in IoT applications. The work proposed an “intelli-Fog” layer which caches mined intelligence from Big Data and makes it available for decision making at the Fog layer. This makes intelligent response to actionable events faster and closer to the devices. Immediate decision making and Big Data analytics can also be taking place concurrently in this new approach with just a single data entry point unlike the Lambda architecture.
5.2 Recommendations

The proposed approach is still very new and had only been tested on just one of the use cases presented in this work. Future work should look into testing the approach by implementing in use cases and comparing the benchmark with existing approaches. While testing the approach against existing approaches, however, careful attention should be paid to the environments to make sure the programs run on the same hardware and using the same communication facilities.

Also this work investigates the architectural part of IoT Big Data management. A second phase of the work should focus on the Big Data processing part. Data ingestion, message queues and stream processing methods there are most suitable for IoT or can be optimized for IoT applications should be investigated.
References


[16] Rhodes P., “Build an IoT analytics solution with Big Data tools” InfoWorld, 2015 available at:


[46] Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, Yves Caniou, “Parallel and Distributed Stream Processing: Systems Classification and Specific Issues”. 2015. <hal-01215287>


54


### Appendix

1. **Regions Class: A class defining the geographic regions in polygons**

```java
package geospatial;

import java.util.Random;
import org.voltdb.types.GeographyValue;

/**
 * Defines a set of regions as polygons roughly around the South Bay area in California.
 */
public class Regions {
    private static final Random RAND = new Random(888);
    private static final GeographyValue[] POLYGONS = {
        GeographyValue.fromWKT("POLYGON ((-121.877094307528
37.393549788232, -121.857799131047 37.402928988778, -121.875569345227
37.414949501508, -121.877094307528 37.393549788232))");
        GeographyValue.fromWKT("POLYGON ((-121.937723896967
37.406109873323, -121.948270936942 37.402928988778, -121.937723896967
37.406109873323))");
        GeographyValue.fromWKT("POLYGON ((-121.830292925514
37.381400226271, -121.836005605124 37.381400226271, -121.830292925514
37.381400226271))");
        GeographyValue.fromWKT("POLYGON ((-121.95945657135
37.393217865646, -121.956605845646 37.393217865646, -121.95945657135
37.393217865646))");
        GeographyValue.fromWKT("POLYGON ((-121.94944763937
37.380647066371, -121.937723896967 37.380647066371, -121.94944763937
37.380647066371))");
        GeographyValue.fromWKT("POLYGON ((-121.864150845646
37.39868826539, -121.956232850697 37.39868826539, -121.864150845646
37.39868826539))");
        GeographyValue.fromWKT("POLYGON ((-121.857799131047
37.406109873323, -121.850359217055 37.381400226271, -121.857799131047
37.406109873323))");
    }
}
```

56
GeographyValue.fromWKT("POLYGON ((-122.070415124774, 37.342078602196, -122.059857857089, 37.352632244821, -122.055481272611, 37.363189516478, -122.064225699621, 37.3697532057, -122.074782967278, 37.37566100956, -122.080968771371, 37.388216739479, -122.079159551756, 37.348264406289, -122.070415124774))"),
GeographyValue.fromWKT("POLYGON ((-121.811676472986, -121.81299397728, 37.34514966966, -121.82130041728, 37.36113614994, -121.82974572616, 37.35587050387, -121.839363722627, 37.346193865165, -121.83376388357, -121.825952234086, 37.334048398494, -121.816794108399, 37.337001003652, -121.811676472986, 37.34514966966))"),
GeographyValue.fromWKT("POLYGON ((-121.7615029261, -121.75649997807, 37.406771549585, -121.812650629806, 37.390019316393, -121.811714484108, 37.380085757661, -121.784092546486, 37.388379900852, -121.77802592184, 37.4000279372034, -121.793780925376, 37.407707695283, -121.805222306558, 37.406771549585))"),
GeographyValue.fromWKT("POLYGON ((-122.006116100962, -122.011760459403, 37.298459224174, -122.008292250917, 37.29754786455, -122.006116100962, -122.011760459403, 37.29754786455)"),
GeographyValue.fromWKT("POLYGON ((-121.997568540152, -122.016000688743, 37.409414603312, -122.015646369052, 37.390628135029, -122.019921220461, 37.39098245472, -122.019921220461, 37.390628135029))"),
GeographyValue.fromWKT("POLYGON ((-122.118570249798, -122.109770432626, 37.27842685217, -122.08730516972, 37.281934132174, -122.087751878916, 37.291728491691, -122.09720440538, 37.294330175184, -122.102600382918, 37.28614361123, -122.096482049875, 37.27842685217))"),
GeographyValue.fromWKT("POLYGON ((-122.038349410981, -122.037212415512, 37.352928556949, -122.109770432626, 37.368813110194, -122.122158247403, 37.38209030544, -122.138614155152, 37.374412359813, -122.13639665068, 37.35638908357, -122.118570249798, 37.352928556949))"),
GeographyValue.fromWKT("POLYGON ((-121.880073129207, -121.8412574836, 37.405397592909, -122.020394410981, 37.405397592909))").
GeographyValue.fromWKT("POLYGON ((-122.190059774575 37.423284506276, -122.19369268814 37.40120380402, -122.174177445851 37.40622810031, -122.16785175129 37.426389644947, -122.190059774575 37.423284506276))")

GeographyValue.fromWKT("POLYGON ((-121.96745980503 37.424715858575, -121.95528537778 37.43678742105, -121.959562600183 37.453366287979, -121.97619393312 37.457873592433, -121.988368040036 37.445802029958, -121.984000813632 37.42922163029))")

GeographyValue.fromWKT("POLYGON ((-121.984000813632 37.390662441789, -122.158035932322 37.40622810031, -122.16785175129 37.426389644947, -122.190059774575 37.423284506276))")

GeographyValue.fromWKT("POLYGON ((-122.158035932322 37.40622810031, -122.16785175129 37.426389644947, -122.190059774575 37.423284506276))")

GeographyValue.fromWKT("POLYGON ((-122.120295017623 37.366571458845, -122.124985344282 37.388146406208, -122.141324633454 37.373296990488, -122.120295017623 37.366571458845))")
64
import java.util.Random;

public static double MIN_LNG = -122.183915;
public static double MAX_LNG = -122.742392;
public static double MIN_LAT = 37.254242;
public static double MAX_LAT = 37.459942;

public static GeographyValue pickRandomRegion() {
    int whichRegion = RAND.nextInt(POLYGONS.length);
    return GeographyValue.fromWKT(POLYGONS.get(whichRegion));
}
The Ad Broker Class:

```java
package geospatial;

import java.io.IOException;
import java.util.Random;
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import org.voltdb.CLIConfig;
import org.voltdb.VoltTable;
import org.voltdb.client.Client;
import org.voltdb.client.ClientConfig;
import org.voltdb.client.ClientFactory;
import org.voltdb.client.ClientStats;
import org.voltdb.client.ClientStatsContext;
import org.voltdb.client.ClientStatusListenerExt;
import org.voltdb.client.NullCallback;
import org.voltdb.client.ProcCallException;
import org.voltdb.types.GeographyPointValue;

/**
 * This class simulates an ad broker application, with two streams of
 * data entering a VoltDB cluster:
 * + The logins of devices from a particular place
 * + The bids generated by businesses wishing to advertise on those devices
 * This class initiates several tasks in parallel:
 * + The simulation of businesses making bids to show ads
 * + The simulation of devices requesting ads
 * Periodic display of stats (e.g., counts of ads displayed for the top 5 businesses)
 * Periodic deletion of expired bids and historical data. (In a real app,
 * this data would probably be exported.)
 */
public class AdBrokerBenchmark {

    // handy, rather than typing this out several times
    static final String HORIZONTAL_RULE =
        "----------" + "----------" + "----------" + "----------" +
        "----------" + "----------" + "----------" + "----------" +
        "\n";

    // validated command line configuration
    final AdBrokerConfig m_config;
}
```
// Reference to the database connection we will use
final Client m_client;

// Instance of Random for generating randomized data.
Random m_rand;

// Benchmark start time
long m_benchmarkStartTS;

// Statistics manager objects from the client
final ClientStatsContext m_periodicStatsContext;
final ClientStatsContext m_fullStatsContext;

final ScheduledExecutorService m_scheduler = Executors.newScheduledThreadPool(4);

// Parameters of generated bids:
// The bid generator will generate one bid (where the advertiser
// is chosen randomly) at a rate of BID_FREQUENCY_PER_SECOND.
// Each generated bid will begin at the current timestamp and last
// for BID_DURATION_SECONDS. Thus, at any given time we'll have
// (BID_DURATION_SECONDS * BID_FREQUENCY_PER_SECOND) active bids
// in the system.
static final int BID_DURATION_SECONDS = 20;
static final int BID_FREQUENCY_PER_SECOND = 5;

// Before the benchmark begins, the bid generator will randomly
// generate NUM_BID_REGIONS polygons. Each generated bid will
// randomly choose one of these polygons as its bid region.
// The number chosen here is based in the number of active bids
// that exist at any given time.
static final int NUM_BID_REGIONS = BID_DURATION_SECONDS * BID_FREQUENCY_PER_SECOND;

// Devices will log in with a device id between 0 (inclusive) and
// NUM_DEVICES (exclusive).
static final int NUM_DEVICES = 1000000;

/**
 * Uses included {@link CLIConfig} class to
 * declaratively state command line options with defaults
 * and validation.
 */
static class AdBrokerConfig extends CLIConfig {
    @Option(desc = "Interval for performance feedback, in seconds.")
    long displayinterval = 5;

    @Option(desc = "Benchmark duration, in seconds.")
    int duration = 60;

    @Option(desc = "Warmup duration, in seconds.")
    int warmup = 5;

    @Option(desc = "Comma separated list of the form server[:port] to
    connect to.")
    String servers = "localhost";
@Option(desc = "Maximum TPS rate for benchmark.")
int ratelimit = 150000;

@Option(desc = "Report latency for async benchmark run.")
boolean latencyreport = true;

@Option(desc = "Filename to write raw summary statistics to.")
String statsfile = "report.txt";

@Option(desc = "User name for connection.")
String user = "";

@Option(desc = "Password for connection.")
String password = "";

@Override
public void validate() {
    if (duration <= 0) exitWithMessageAndUsage("duration must be > 0");
    if (warmup < 0) exitWithMessageAndUsage("warmup must be >= 0");
    if (displayinterval <= 0)
        exitWithMessageAndUsage("displayinterval must be > 0");
    if (ratelimit <= 0)
        exitWithMessageAndUsage("ratelimit must be > 0");
}

class StatusListener extends ClientStatusListenerExt {
    @Override
    public void connectionLost(String hostname, int port, int connectionsLeft, DisconnectCause cause) {
        if ((System.currentTimeMillis() - m_benchmarkStartTS) < (m_config.duration * 1000)) {
            System.err.printf("Connection to %s:%d was lost.\n", hostname, port);
        }
    }
}

public AdBrokerBenchmark(AdBrokerConfig config) {
    m_config = config;

    ClientConfig clientConfig = new ClientConfig(config.user, config.password, new StatusListener());
    clientConfig.setMaxTransactionsPerSecond(config.ratelimit);

    m_client = ClientFactory.createClient(clientConfig);

    m_rand = new Random(777);

    m_periodicStatsContext = m_client.createStatsContext();
    m_fullStatsContext = m_client.createStatsContext();

    System.out.print(HORIZONTAL_RULE);
    System.out.println(" Command Line Configuration");
    System.out.println(HORIZONTAL_RULE);
    System.out.println(config.getConfigDumpString());
    if (config.latencyreport && config.ratelimit == Integer.MAX_VALUE) {
        System.out.println("Warning: Latency reports will be disabled.\n");
    }
}
System.out.println("WARNING: Option latencyreport is ON, and no rate limit is set. " + "Expect high latencies to be reported.
");

static void connectToOneServerWithRetry(Client client, String server) {
    int sleep = 1000;
    while (true) {
        try {
            client.createConnection(server);
            break;
        } catch (Exception e) {
            System.err.printf("Connection failed - retrying in %d second(s).\n", sleep / 1000);
            try { Thread.sleep(sleep); } catch (Exception interrupted) {}
            if (sleep < 8000) sleep += sleep;
        }
    }
    System.out.printf("Connected to VoltDB node at: %s.\n", server);
}

static void connect(final Client client, String servers) throws InterruptedException {
    System.out.println("Connecting to VoltDB...");
    String[] serverArray = servers.split(",");
    final CountDownLatch connections = new CountDownLatch(serverArray.length);

    // use a new thread to connect to each server
    for (final String server : serverArray) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                connectToOneServerWithRetry(client, server);
                connections.countDown();
            }
        }).start();
    }
    // block until all have connected
    connections.await();
}

public void schedulePeriodicStats() {
    Runnable statsPrinter = new Runnable() {
        @Override
        public void run() {
            printStatistics();
        }
    };
    m_scheduler.scheduleWithFixedDelay(statsPrinter, m_config.displayinterval, m_config.displayinterval, TimeUnit.SECONDS);
}

/**
 * Print stats for the last displayinterval seconds to the console.
 */
public synchronized void printStatistics() {
    ClientStats stats = m_periodicStatsContext.fetchAndResetBaseline().getStats();
    long time = Math.round((stats.getEndTimestamp() - m_benchmarkStartTS) / 1000.0);

    System.out.printf("%02d:%02d:%02d ", time / 3600, (time / 60) % 60, time % 60);
    System.out.printf("Throughput %d/s, ", stats.getTransactionThroughput());
    System.out.printf("Aborts/Failures %d/%d", stats.getInvocationAborts(), stats.getInvocationErrors());
    if (m_config.getLatencyReport()) {
        System.out.printf(" Avg/95%% Latency %.2f/%.2fms", stats.getAverageLatency(),
                        stats.getKPercentileLatencyAsDouble(0.95));
    }
    System.out.print("\n");

    VoltTable[] tables = null;
    try {
        tables = m_client.callProcedure("GetStats", m_config.getDisplayInterval()).getResults();
    } catch (IOException | ProcCallException e) {
        e.printStackTrace();
        System.exit(1);
    }

    VoltTable expired = tables[0];
    //System.out.printf("Expired Ad requests exported to the Cloud layer for further analysis\n");
    // m_client.writeSummaryCSV(expired, "MyExport.csv");
    while (expired.advanceRow()) {
        System.out.printf("%-40s  %9.2f    %d\n", expired.getString(0),
                          expired.getDouble(1), expired.getLong(2));
    }
    System.out.println();
}

/*/---------------------*/

VoltTable hits = tables[0];
hits.advanceRow();
assert hits.getLong(0) == 0;
long unmetRequests = hits.getLong(1);
hits.advanceRow();
assert hits.getLong(0) == 1;
long metRequests = hits.getLong(1);
long totalRequests = unmetRequests + metRequests;
double percentMet = ((double)metRequests) / totalRequests) * 100.0;

System.out.printf("Total number of ad requests: %d, %3.2f%%
resulted in an ad being served\n", totalRequests, percentMet);

VoltTable recentAdvertisers = tables[1];
System.out.println("Top 5 advertisers of the last " +
m_config.displayinterval + " seconds, " +
"by sorted on the sum of dollar amounts of bids won:\n");
System.out.println("Advertiser Category Revenue Count\n"------- ------
--     -------   -----\n"----
while (recentAdvertisers.advanceRow()) {
    System.out.printf("%-40s %-20s  %9.2f    %d\n",
        recentAdvertisers.getString(0),
        recentAdvertisers.getString(1),
        recentAdvertisers.getDouble(2),
        recentAdvertisers.getLong(3));
}
System.out.println();

public synchronized void printResults() throws Exception {
    ClientStats stats = m_fullStatsContext.fetch().getStats();

    System.out.println(HORIZONTAL_RULE);
    System.out.println(" Client Workload Statistics\n"HORIZONTAL_RULE);
    System.out.println("Average throughput:            %,9d txns/sec\n",
        stats.getTxnThroughput());
    if (m_config.latencyreport) {
        System.out.println("Average latency:               %,9.2f ms\n",
            stats.getAverageLatency());
        System.out.println("10th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.1));
        System.out.println("25th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.25));
        System.out.println("50th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.5));
        System.out.println("75th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.75));
        System.out.println("90th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.9));
        System.out.println("95th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.95));
        System.out.println("99th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.99));
        System.out.println("99.5th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.995));
        System.out.println("99.9th percentile latency: %,9.2f ms\n",
            stats.kPercentileLatencyAsDouble(.999));
    }
System.out.printf("Reported Internal Avg Latency: %9.2f ms\n", stats.getAverageInternalLatency());

System.out.println("\n" + HORIZONTAL_RULE);
System.out.println(" Latency Histogram");
System.out.println(HORIZONTAL_RULE);
System.out.println(stats.latencyHistoReport());
}

// 4. Write stats to file if requested
m_client.writeSummaryCSV(stats, m_config.statsfile);

private void shutdown() {

    // Stop the stats printer, the bid generator and the nibble deleter.
    m_scheduler.shutdown();

    try {
        m_scheduler.awaitTermination(60, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    try {
        // block until all outstanding txns return
        m_client.drain();
        // close down the client connections
        m_client.close();
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
}

private GeographyPointValue getRandomPoint() {
    double lngRange = Regions.MAX_LNG - Regions.MIN_LNG;
    double lng = Regions.MIN_LNG + lngRange * m_rand.nextDouble();

double latRange = Regions.MAX_LAT - Regions.MIN_LAT;
    double lat = Regions.MIN_LAT + latRange * m_rand.nextDouble();

    return new GeographyPointValue(lng, lat);
}

private void requestAd() {
    long deviceId = Math.abs(m_rand.nextLong()) % AdBrokerBenchmark.NUM_DEVICES;
    GeographyPointValue point = getRandomPoint();

    try {
        m_client.callProcedure(new NullCallback(),
        "GetHighestBidForLocation", deviceId, point);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
public void runBenchmark() throws Exception {
    System.out.println(HORIZONTAL_RULE);
    System.out.println(" Setup & Initialization");
    System.out.println(HORIZONTAL_RULE);

    // connect to one or more servers, loop until success
    connect(m_client, m_config.servers);
    System.out.println("\n\n" + HORIZONTAL_RULE);
    System.out.println(" Starting Benchmark");
    System.out.println(HORIZONTAL_RULE);

    System.out.println("\nStarting bid generator\n");
    BidGenerator bidGenerator = new BidGenerator(m_client);
    long usDelay = (long) ((1.0 / BID_FREQUENCY_PER_SECOND) * 1000000.0);
    m_scheduler.scheduleWithFixedDelay(bidGenerator, 0, usDelay,
            TimeUnit.MICROSECONDS);

    //System.out.println("\nStarting nibble deleter to delete expired
data\n");
    NibbleDeleter deleter = new NibbleDeleter(m_client,
            m_config.displayinterval + 1);
    // Run once a second
    m_scheduler.scheduleWithFixedDelay(deleter, 1, 1,
            TimeUnit.SECONDS);

    System.out.println("\nWarming up...");
    final long warmupEndTime = System.currentTimeMillis() + (10001 * m_config.warmup);
    while (warmupEndTime > System.currentTimeMillis()) {
        requestAd();
    }

    // reset the stats
    m_fullStatsContext.fetchAndResetBaseline();
    m_periodicStatsContext.fetchAndResetBaseline();

    // print periodic statistics to the console
    m_benchmarkStartTS = System.currentTimeMillis();
    schedulePeriodicStats();

    // Run the benchmark loop for the requested duration
    // The throughput may be throttled depending on client configuration
    System.out.println("\nRunning benchmark...");
    final long benchmarkEndTime = System.currentTimeMillis() + (10001 * m_config.duration);
    while (benchmarkEndTime > System.currentTimeMillis()) {
        requestAd();
    }

    printResults();
    shutdown();
}

public static void main(String[] args) throws Exception {
    // create a configuration from the arguments
    AdBrokerConfig config = new AdBrokerConfig();
    config.parse(AdBrokerBenchmark.class.getName(), args);
AdBrokerBenchmark benchmark = new AdBrokerBenchmark(config);
benchmark.runBenchmark();