Scalable Stream Processing and Management for Time Series Data

Bamdad Mousavi

Thesis submitted to the
University of Ottawa
In partial fulfillment of the requirements
For the Master of Science in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

There has been an enormous growth in the generation of time series data in the past decade. This trend is caused by widespread adoption of IoT technologies, the data generated by monitoring of cloud computing resources, and cyber physical systems. Although time series data have been a topic of discussion in the domain of data management for several decades, this recent growth has brought the topic to the forefront. Many of the time series management systems available today lack the necessary features to successfully manage and process the sheer amount of time series being generated today. In this today we stive to examine the field and study the prior work in time series management. We then propose a large system capable of handling time series management end to end, from generation to consumption by the end user. Our system is composed of open-source data processing frameworks. Our system has the capability to collect time series data, perform stream processing over it, store it for immediate and future processing and create necessary visualizations. We present the implementation of the system and perform experimentations to show its scalability to handle growing pipelines of incoming data from various sources.

Acknowledgements

First and foremost, I would like to immensely thank my thesis supervisor, Dr. Verena Kantere. Her depths of knowledge in the domain of data management, her strong enthusiasm for research and her patience, were of great help during my work on the thesis.

Next, I would like to thank Paris Kerasiotis for non-hesitantly sharing his experience in the big data management with me.

Finally, I would like to thank my parents, Morteza and Mahnoosh, who taught me from an early age the importance of pursuing knowledge for a lifetime. I am where I am today because of their continuous support.

Table of Contents

Table of Figures

1. Introduction

The amount of data being generated has seen an exponential growth in the past two decades. This growth is the result of cloud computing technologies, big data processing frameworks and the Internet of Things (IoT) and ever-increasing bandwidth with each new generation of Cellular networks.

Cloud computing was initially conceived to simplify and streamline Information Technology (IT) infrastructure management for companies. As Infrastructure-as-a-Service (IaaS) services grew more mature, major cloud providers started offering Platform-as-Service (PaaS) and Software-as-a-Service (SaaS) services. This family of cloud services became an enabler for the data generation trend. This was the result of PaaS services giving the businesses unprecedented power in managing their current data repositories and the capability to acquire, collect and process more data. SaaS services provided a platform for new online services, among them a significant stream of mobile applications becoming available on the two major mobile operating systems iOS and Android. The new apps meant a variety of new data being generated by the users that could be processed by the app developers for usage insights. Temporal and Spatial data are the major categories of data in this domain.

The Big Data processing era started with the birth of Hadoop and eventually saw it turned into an open-source project. This was a major catalyzer for Hadoop to turn from a multi-layer big data processing framework to a complete ecosystem of tools tailored toward data processing at scale. This major development along with the capability to provide big data processing as a PaaS service by cloud service providers has been another push toward enlargement of data generation volume and velocity.

IoT owns it birth and development to several different factors. First one is the historical development of wireless sensor networks. The second is the advancements in narrowing down the data transfer channels within silicon chips which has resulted in design and production of processors that are smaller and more energy efficient. The third factor is the development in battery design technologies which has enabled longer lasting batteries capable of operating in more rigid environments. Nowadays we can see Things (sensors paired with processors equipped with an

internet connection) everywhere, from smart thermostats in our living rooms that give us the capability to control our house's temperature, to the deep ends of the ocean to monitor marine life.

The next enabler for the acceleration of data generation trend is the cellular network. The fourth generation of mobile networks, 4G, enabled new services for mobile devices such as video streaming services and caused a shift in social media usage going from personal computers to mobile devices. The fifth generation is making a new level of services possible, such as 4K video streaming on mobile devices and connected vehicles.

This massive amount of data consists of a variety of data types. One large portion of data is composed of texts and images. These two data types have been the focus of data management in academia for a while. The other major portion of this ever-growing global dataset consists of time series data. In its simplest definition time-series data is a form of data that has a timestamp attribute. Time series data has been a topic within data management for several decades. It is widely associated with financial markets as the time characteristic acts as an enabler for forecasting the future trend within the markets. Another domain historically associated with time series data is the health sciences and more specifically model generation for the spread of infectious diseases. However, the four technologies mentioned above have renewed the focus on time series data management because of the new sources and the volume and velocity of its generation within the past few years.

1.1 Sources of time series data

To do a more comprehensive discussion on the sources of time series data, three different application domains need to be studied. These domains are Industrial Internet of Things (IIoT), cyber physical systems, applications and computing infrastructure metrics, transportation fleet management.

1. IIoT is where IoT and industrial plants are combined. One can consider an automated mega factory consisting of multiple production lines with different parts of the production lines and robotic arms each acting as a thing, constantly sending metrics from machine operations or functioning characteristic like the power consumption of the machine or the temperature of its different parts to a central factory monitoring system. This system would use the time series data to make real time decisions based on the anomalies in the data or

change the production capacity depending on the operational situation of different machines.

2. This mega factory could be connected to a smart grid including power generation sources like wind turbine farms, geothermal wells, and solar farms. The smart grid management system can use the metrics gathered from smart meters in the grid, which are in the form of time series data to forecast an upcoming increase in the power needed within the grid and increase the operational capacity of the wind turbines and the output of the geothermal wells simultaneously.
3. The emergence of IaaS, PaaS, and SaaS services has caused a new challenge and an opportunity for IT operations team. They have given application development teams the ability to gather and analyze metrics from applications, computing platforms and infrastructure components. These metrics, in the form of time series data, provide a level of visibility into the whole system operations that was not achievable just a few years ago. This trend has been only accelerated by the utilization of micro-services architecture as a software design pattern and cloud native development practices, including containerization, for the design, development, and deployment of software. The containerization practice has consequently led to widespread adoption of Kubernetes (K8S) as a container-orchestration system which produces significant amount of logs describing the operational status of the system components through its lifetime.
4. Transportation fleet management comes into the picture as part of the supply chain systems of different enterprises. An example of time series generation in this domain would be an e-commerce giant with its own delivery fleet. Each vehicle in the fleet is equipped with a device that tracks the movements of the vehicle through its delivery route and based on the amount of time it has taken for the deliveries so far calculates an approximate delivery time for upcoming deliveries along the route. It then sends notifications to corresponding customers to inform them of any potential delays.

1.2 Data structure of time series data

The next characteristic of time series data to be reviewed is its data structure. The data structure impacts the type of operations that can be performed over the time series data and because of it, changes the overall requirements for time series data management compared to relational data. Time series data is in most cases a combination of a timestamp combined with one or multiple

key-value pairs. The presence of timestamp in the data structure has two major consequences. First it puts an emphasis on the importance of time continuum in the data domain. This could then present itself in the form of an unbounded time series in many domains that need to be collected, processed, and stored for an unforeseen amount of time. Second, the value of time continuum would result in less significance placed on the update operations in a TSMS and a change in how the new data points are written to the data store in the TSMS which in case of most TSMSs is in the form of a write to the tail of the data already present in the data store. This second point causes a new requirement for the TSMS and that is design and implementation of a new indexing mechanism for data points that are optimized for this data structure.

1.3 Challenges of time series management

The combination of the new requirements caused by the data structure and data operations discussed previously, along with the fact that in most cases time series data are unbounded, has introduced a new set of problems and challenges within the data management domain. These challenges cannot be addressed using the data management techniques, mechanisms, and methodologies devised throughout the past few decades for relational data because of the fundamental differences between relational and time series data and the specific requirements of time series data. To better understand these challenges an example scenario will be beneficial. The example of the smart grid discussed earlier in this chapter would serve this purpose very well. The city of Ottawa, ON, in Canada has an approximate population of 1,000,000 people according to latest census data. If we assume that there are 50,000 residences (apartments, townhomes, houses …) in Ottawa and each residence is equipped with a smart meter, then there are 50,000 smart meter measuring residential electricity consumption in Ottawa. Ottawa has a four-season weather with warm summers and cold winters. This means these residences require air conditioning for summer months and heat for winter months. Even in spring and fall seasons the temperature fluctuations throughout the day could be stark. Being located at a northern latitude, the difference in the duration of sunlight throughout the day in summer and winter months is considerable. This will as well significantly impact the amount of electricity used for lighting in the residences. Based on the conditions discussed above each of the smart meters in the residences measure the electricity consumption every 5 minutes and transfer it to the power company. Each smart meter reading would consist of a timestamp as a Long Integer, the meter reading as a Long Integer and the measurement as a Float. Since each Long Integer value and Float would require 8 bytes of data

then the whole measurement for a reading by the smart meter would be 24 bytes. Since the reading is done every 5 minutes each residence would generate 6912 bytes of data. Considering the 50000 residences in Ottawa, the amount of data generated by all the city residences would equal to 329 GB of data per day and 9.8 TB per month. Now if such a smart grid is to be scaled to serve the population of a city the size of New York, NY in the United Stated, the challenges of scalability requirements for such a TSMS and its storage management capabilities becomes more apparent.

In order to look at the current state of the art in development of time series management systems, it is best to look at the most comprehensive literature review of TSMSs have been done by Jensen et al [1]. Their work surveys a wide range of TSMSs that have been developed for various purposes. Upon further investigation into the summary of the systems they reviewed, based on their classification criterion of maturity level, 9 out 27 systems had reached the maturity level, 5 were proof of concepts and 13 were demonstration systems. In terms of system architecture (being distributed) 13 out of 27 systems were centralized and 14 were distributed systems. In terms of the storage scheme 10 systems were using internal stores, 14 systems were using external stores and 3 systems were DBMS extensions. They express the research direction suggested by the experts in the field as “It is proposed such systems should support in-memory, parallel and distributed processing of time series as a means to reduce query processing time enough as means to enable interactive data analytics and visualization”.

1.4 Our work

The research direction discussed at the end of previous section was the main motivation for this thesis. In this thesis we strive to address the scalability challenges of time series processing and management that the sheer amount of time series data discussed earlier imposes on any TSMS. In this work we present a system architecture to handle time series processing and management at scale and we showcase it can handle processing massive volumes of data at scale. In chapter two a comprehensive review of the work on time series data management in the academic literature is presented. Chapter three discusses the proposed system architecture, the system workflow, and its functionalities. Chapter four discusses the implementation of the system. Chapter 5 presents the examination of our designed systems and how it scales as the size of the incoming data to the system grows and chapter 6 provides a conclusion to our work.

2. Background and related work

In this chapter we do a review of the research work published on the topic of time series data processing and analytics. Our research shows that the work in this domain could be divided into two major categories, the work on system design for time series management and the analytics on time series data. As a result, this chapter is divided into two sections. In section 2.1 we look at time series processing from a combination of system management and time series data warehousing perspective. In section 2.2 we study the work published on the topic of time series analytics. The reasoning behind this decision lies within the fact that the capabilities of a data management system when it comes to data warehousing has direct correlations with the major design decisions that are made during the design process of the system.

2.1 system design and data warehousing for time series data

In this section we study some of the work published on design and implementation of TSMS. They are reviewed from the design decision and system components point of view.

HeteroTSDB [2] is an extensible TSMS with the capability to store and manage high resolution time series data as a backbone for a monitoring system for the infrastructure. HeteroTSDB focuses on managing time series data as key-value pairs. It is designed based on the idea that building a TSMS with loosely coupled components provides a higher level of extensibility. As a result, its architecture includes two key-value stores (KVS). It utilizes an in-memory KVS to increase the write-efficiency and an on-disk KVS to improve the read efficiency. To avoid data loss, because of the system failure in the in-memory KVS, HeteroTSDB implements a write-ahead log (WAL). Other major components of HeteroTSDB include a message broker, a metric reader, a metric writer, a metric cleaner, and a metric mover. HeteroTSDB uses a key-value pair as its core data structure. It stores the metric name as the key and a series of timestamps with values as the value in the pair. It bundles data into regular intervals and writes them to disk. Although the results of the experiments on TSDB sound promising in terms of extensibility, there are a few significant issues with it. First it was built using PaaS services provided by AWS, such as ElasticeCache for Redis and DynamoDB. This prevents the system from being properly benchmarked as there is no direct access to the computing infrastructure hosting these services. The other issue is that HeteroTSDB performs operations on regular time intervals based on the assumption that the arrival

of the data into the system is consistent. It does not consider that the data flow into the system may change over time.

TSDB [3] is a TSMS, designed for management of metrics collected by network monitoring. It tries to address an ambitious set of goals including but not limited to, minimal append and extraction time when handling millions of time series, running update and append operations of different ranges of time series, preserving original value of time series at the time of storage and concurrent read/write operations on the database. TSDB stores time series with fixed intervals. TSDB takes advantage of BerkelyDB as its internal store to unify the storage of both the time series and its metadata.

PhilDB [4] is a TSMS that targets time series management in domains that necessitate updates to the data points already stored in the data store. To achieve its goal, it keeps tracks of the changes to the data by logging them. From a system architecture point of view, it implements this feature by storing the updates separate from the original data. PhildDB stores the metadata in a relational database and the original time series data points within files on the file system. It generates a UUID for the metadata and add the UUID to the filename containing the associated data points to generate the necessary correlation. PhilDB has a centralized architecture.

Plato [5] is a TSMS that combines the powers of a traditional RDBM with methods from signal processing to provide analytics capabilities over spatiotemporal data. It offers a DBMS architecture for sensor data that supports reusable learning algorithm modules. It provides two query languages, ModelQL and InfinityQL for users from different backgrounds. It also provides a hierarchical data store for models inferred by application of signal processing techniques.

servIoTicy [6] is a TSMS developed to serve data generated by IoT devices. It has a two-layer architecture composed of a frontend and a backend. The frontend exposes a REST API for IoT devices to send their data to the TSMS in JSON format. The backend uses Couchbase for data storage. It then utilizes Elasticsearch on top of Couchbase to index the data for faster lookups. The system distinguishes between the original data point and the metadata from the devices by storing them in two different formats. It uses Apache Storm to provide stream processing capability.

Tristan [7] is a TSMS designed for efficient management and storage of time series data It provides effective compression by taking advantage of sparse dictionary representation. It also offers highly

optimizesd storage and query execution over compressed data. Its architecture is composed of three layers, a data acquisition layer, a storage layer, and a query execution layer. It supports AQP to reduce query processing but does not support user defined functions.

Bolt [8] is a TSMS aimed at facilitating the development of IoT applications for connected homes. It uses a tuple as the main data structure for handling the time series data. The tuples consist of a timestamp and key-value pairs. Using the tuples enables Bolt to store both the time series data and the meta data. Built uses encrypted streams to manage incoming time series into the system. The streams are divided into chunks and written to a persistent log file. Bolt keeps the index in memory to speed up the read process. Bolt incorporates a distributed architecture. Its architecture includes a metadata server to provide the information about the available streams and it stores the encrypted streams in Amazon S3 or Microsoft Azure.

MetricQ [9] is a TSMS designed to facilitate data center monitoring. It has a distributed architecture that is composed of a message broker, a management agent, a persistent data store, a set of combinators and aggregators, and data sinks that serve visualization tools. MetricQ uses a flat file schema for data storage instead of a B-Tree or a Hash-Index based on the assumption that the major storage operation on the data is append only.

Few companies in the world need to run infrastructure at the scale of Facebook. One of the challenges of such an endeavor is the monitoring of computing resources to make sure the health status of the systems is in accordance with operations guidelines. This monitoring is performed based on the metrics gathered from servers, services and networking and storage devices in the form of time series data. Facebook's former monitoring system developed in house was called ODS (Operational Data Store). It consisted of a TSDB based on HBase, a query engine and a monitoring and alert system. When ODS reached its peak scalability, which resulted in query execution times reaching tens of seconds, it was managing over 2 PB of data. Facebook then introduced Gorilla [10]. It was designed based on several facts observed from the operation of ODS. These facts included the dominance of write operations over reads, the impact of state transitions within their operational systems, the need for high availability and fault tolerance. Gorilla is an in-memory TSMS that works as write through cache for data which will be written to HBase.

The main data structure in Gorilla is a 3-tuple, consisting of a data point key of type String, a timestamp of type Integer, and a value of type float. Gorilla offers the capability for 12x compression by implementing compression techniques on both the keys and the values. Gorilla performs value compression based on the fact that in a time series there is little difference between two consequent values. So, by XORing two adjacent values in the time series Gorilla could retain on a single bit of difference between the two. It performs compression on the time series based on the delta between two consequent data point in each series and calculating a delta of delta for all the points within the series. The in-memory data structure in Gorilla is called TSMAP. It is a vector based on C++ shared pointers, pointing to time series and a map from time series names to the time series.

Gorilla uses a distributed file system called GlusterFS to achieve on disk persistence. It writes complete blocks of data to GlusterFS every two hours. Data is written to the disk in buffers of 64 KB that usually contain 2 seconds of data. The experience of using Gorilla in production has shown that it has been able to successfully meet all the requirements foreseen for it.

Modern data centers have several thousands of servers. This very large number of servers needs to be constantly monitored. The servers hosting services with high-level of uptime requirements, could be monitored by continually gathering 500 hundred performance metrics [11]. A data with 100000 servers will generate 1 TB of data per day when the data is collected in 15 seconds granularity. In such a scenario not only processing the most recent data is vital but also historical data needs to be preserved for purposes like capacity planning and fault diagnostics. To overcome the challenges associated with the storage and management of these performance metrics Microsoft introduced DataGarage [11]. It uses a combination of TableStore and FileStore approaches to manage data storage with the minimal amount of overhead. The metrics from each server are recorded in a wide table that is subsequently written to a SQL Server Compact Edition (SSCE). Each SSCE file is then distributed among the servers. This approach has several advantages. Each SSCE provides full query capabilities using SQL and minimal storage overhead. It also gives server owners the ability to define their own schema for the tables based on the number of metrics they would want to collect from any given server.

One of the challenges in the growth of IoT has been the burden of network bandwidth needed by high-sampling frequency sensors in a resource constraint environment. IoT platforms have been

traditionally dependent on cloud computing to perform data storage and analysis. Traditional TSMSs are incapable of running in an edge environment because of the limited amount of computing resources. To solve this issue a TSMS named EdgeDB [12] was introduced. It aims to run at the edge location and perform preliminary data analysis and aggregation. EdgeDB introduces three novel mechanisms to address the challenges of time series processing at edge. It enables multi-stream merging between correlated data streams so they could be queried and processed together. Next, EdgeDB introduces Time Partitioned Elastic Index (TPEI), an index with an enhanced level of disk layout to achieve higher level of time-range query performance. Last, it implements Time Merged Tree (TMTree) to elevate write efficiency in the database by merging multiple small writes into a larger single write.

Respawn [13] is another system designed for the efficient management of time series data within IoT domain. It is based on a cloud-to-edge distributed architecture. It utilizes a light-weight time series database named BodyTrack Datastore at the edge layer. As data enters the edge devices, it gets aggregated to enable range-based queries with a low level of latency. This makes Respawn ideal for networks with limited bandwidth or intermittent connections. The down-sampled data is then transferred to the cloud tier that is responsible for long-term storage, analytics, and visualization of aggregated data.

BTrDB[14] is a distributed TSMS proposed by Andersen et al. It is designed to manage sub-milliseconds time series data generated by high precision power meters in an electrical grid. BtrDB proposes a new abstraction of the telemetry data and implements a new data structure to accommodate the abstraction. The abstraction provided by BTrDB is an ordered sequence of time value pairs that distinguishes between the stream by an UUID. BTrDB uses a time partitioning k-ary tree as the main data structure. By using the tree, BTrDB can store data points in the leaves of the tree and the time interval between the data points is presented by the depth of the tree. Since the tree is copy on write, queries run on historical data require similar cost to execute.

As discussed in the introduction, one of the major contributors to the massive velocity in the generation of time series data is IIoT. One of the latest studies trying to solve the issues of scalable data persistence has been presented in [15] where three data stores are compared together. The three data stores used are MongoDB, which is a DocumentDB, Cassandra, a Column Store and InfluxDB, a TSMS. The study is based on large amount (tens of GBs) of real world IIoT data. The

results of the experiments indicated that InfluxDB outperformed its competitors on ingestion and time-based queries. In retrieval performance MongoDB showed better results.

ModelarDB[16] is a TSMS designed to address the need for management of big time series data generated by sensors. It is a model based TSMS that supports multiple models for time series data simultaneously. It also supports lossy and lossless compression for time series. It has a distributed architecture composed of three components, data ingestion, query processing, and segment storage. ModelarDB uses Apache Spark for query processing and Cassandra for data storage. The query engine interface, storage interface, and metadata cache are implemented as a core library. The results of the evaluation of the ModelarDB against other state of the art solutions for time series data management show that ModelarDB achieves a very good performance specially in terms of the data ingestion and query processing.

2.2 Analytics on time series data

In this section we will review related work with a focus on time series analytics and system components that directly impact the analytics of the TSMS discussed in the literature

As discussed in the introduction chapter one of the domains that benefits from improvement in times series management and analytics is the field of cyber physical systems and more specifically smart grids. This domain must deal with the problem of data loss like any other IoT application domain. One of the most comprehensive works to solve this issue has been presented in [17]. It discusses a system designed specifically for smart grids. Although some of the design decisions presented are common in this domain, its significance comes from the data analytics framework built on top of the underlying system. The system provides an extraction engine and a correction engine. The correction engine is responsible for missing values prediction based on a method called sensor-network-regulization-based matrix factorization (SnrMF). SnrMF combines USbR and CSbR to optimize missing value prediction. USbR handles missing value prediction for correlated sensors and CSbR performs missing value prediction for uncorrelated sensors.

Many systems like financial fraud detection, medical diagnosis, network intrusion detection and industrial fault detection rely on anomaly detection. Anomaly detection is the process of identifying outliers in a data series. An outlier is a data point that has a large difference with other data points in the series. Most of the anomaly detection methods are application domain dependent [18]. The Extensible Generic Anomaly Detection System (EGADS) [18] is an effort by Yahoo to

perform application domain agnostic anomaly detection at scale. EGADS provides three mechanisms for anomaly detection, outlier detection, change point detection and detecting anomalous time series. Experiments results show that EGADS was able to successfully meet the requirements at yahoo, mainly for intrusion detection and fault detection. However, since the system and the dataset have been open-sourced further experiments seem necessary to evaluate extensibility of the system for anomaly detection in other application domains.

2.3 Overview and the direction of this thesis

Table 1 lists the systems reviewed in this chapter and two of their characteristics, system architecture i.e., whether it is centralized or distributed, and their support for stream processing over time series data.

System	Architecture	Stream Processing
HeteroTSDB	Distributed	No
TSDB	Centralized	No
PhilDB	Centralized	No
Plato	Centralized	No
servIoTicy	Distributed	Yes
Tristan	Centralized	Yes
Bolt	Distributed	Yes
MetricQ	Distributed	Yes
Gorilla	Distributed	No
DataGarage	Distributed	Yes
EdgeDB	Centralized	Yes
Respawn	Distributed	No
BtrDB	Distributed	Yes
ModelarDB	Distributed	Yes

Table 1: List of reviewed systems in this thesis

Table 1 shows that less than half of the systems reviewed in this chapter from a system design perspective support stream processing and a small majority of them have a distributed architecture. Based on the characteristics of time series data that were discussed in chapter 1, and the strengths and weaknesses of the systems reviewed in chapter 2, we believe that a modern

TSMS needs to offer stream processing capability based on a distributed architecture. We also believe the stream processing capabilities of the said TSMS need to scale as the volume or velocity of the data coming into the system increases. In this thesis we propose a TSMS with a distributed architecture that support stream processing over time series data and has a multi-layer storage component that provides capabilities for near analytics on the near real time and historical time series data. We then build the system from the proposed system components and subsequently measure the scalability of our system to show that it scales properly to meet the demands of the high velocity of time series data generation.

3. Proposed Solution

In this chapter we discuss the system architecture, system functionalities, and system workflow.

3.1 System Architecture

In this section we discuss the architecture of the system on two level, the high-level and the low-level. We also review how every system component fits into the overall architecture of the system.

3.1.1 Architecture of the system

The architecture of the system is shown in Figure 3-1and Figure 3-2. Figure 3-1 shows the high-level architecture of the system. At the high-level the system is composed of four different modules: data collection, data processing, data storage and analytics, and data visualization. In Figure 3-1 the arrows between different system components are representative of the high level direction of the data flow within the system.

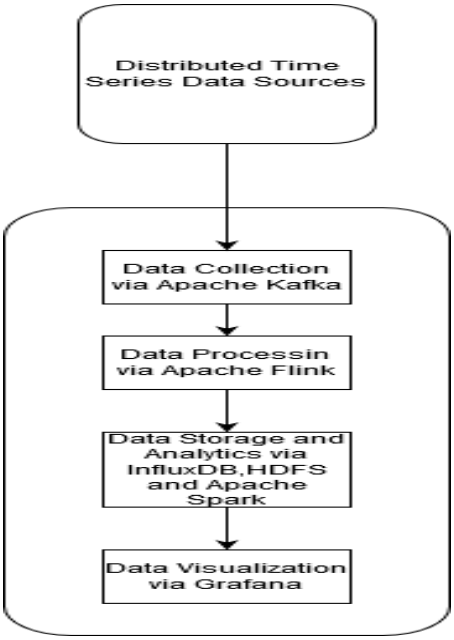

Figure 3-1: High-Level System Architecture

Figure 3-2 shows the low-level architecture of the system. At this level, we can see that the system is built using several open-source software frameworks.

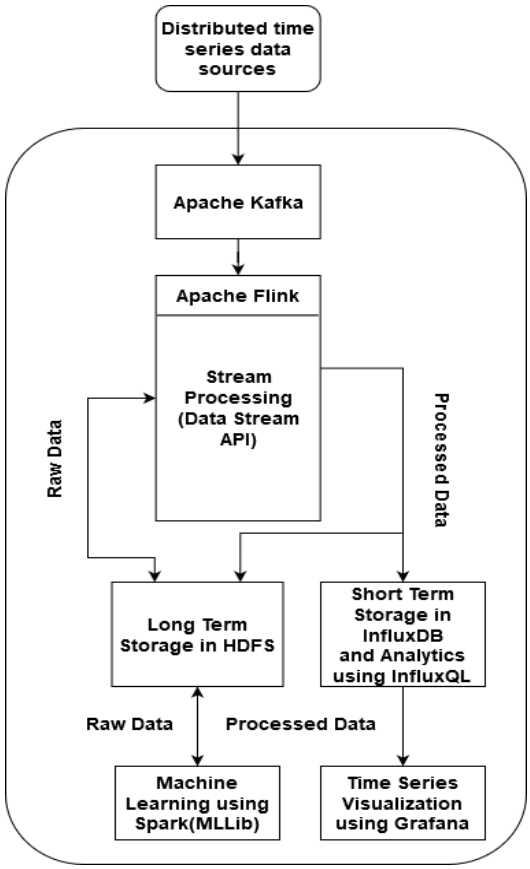

Figure 3-2: Low-Level System Architecture

These frameworks include Apache Kafka, Apache Flink, HDFS, InfluxDB, Apache Spark and Grafana. The arrows between the system components demonstrate corresponding data flow in the respective direction. In the following sections all the system components are reviewed in details and their role within the architecture of the system is discussed.

3.1.2 Apache Kafka

Apache Kafka is a popular event streaming platform. It was initially developed at LinkedIn to handle billions of events generated by its members' activities on the website and feed them into different data driven products at the company [19].

Kafka provides three major functionalities for event stream processing: a publish/subscribe model for stream events, durable storage of stream of events and events stream processing. Kafka works as a distributed system of servers and clients. Each of these servers and clients can be deployed to bare-metal servers, virtual machines, and containers [20].

An event in Kafka is the representation of the occurrence of a phenomenon in the real world. The conceptual modeling of an event usually consists of a key–value pair, a timestamp, and optional metadata. The Kafka clients discussed earlier come in two groups, publishers, and subscribers. Publishers are the clients that write events to Kafka and subscribers are clients that read the events from Kafka. Kafka achieves high-level of scalability by completely decoupling publishers from subscribers as an important design decision. In Kafka events are stored in a logical structure called topics. The publishers in Kafka write events to topics and the subscribers read the data from the topic. A Kafka application can have as many topics as needed. Kafka can retain the data in each topic for as long as required. Topics can be partitioned into multiple buckets and distributed onto several brokers (servers) in Kafka. This replication prevents occurrence of single point of failure in the system and in case of a broker failure a replica broker will continue serving the topic [20].

3.1.3 Apache Flink

In the early days of big data era, data stream processing and batch processing were considered two different categories of applications. Each used different programming models and were run by different systems [21]. Over time the data generation trend shifted toward more and more data being generated in a continuous manner from a variety of sources like web application, application logs, IoT devices, to name but a few. Architectural patterns such as Lambda Architecture that had historically dominated the big data processing domain, utilize separate systems for low latency stream processing and high accuracy batch processing introduce a high level of latency (induced by batches) and complexity (integration of various systems and duplication of business logic). They also add a random level of inaccuracy since the application code does not properly handle the time dimension of the data. This issue is specifically troubling when it comes to processing time series data [21].

Apache Flink was built based on a unified model of real-time analysis and batch processing both in its data processing engine and the programming model that it exposes to the developers. Flink provides an adaptive windowing mechanism that allows for early and approximate as well as delayed and accurate results on the same data stream. Flink provides two different notions of time, event time, and processing time. By doing so, it offers a high level of flexibility to the programmers on how to program necessary computations over the data stream [21].

In addition to a stream processing framework, Flink also provides a batch processing framework to support both traditional use cases and operations that are not supported by the stream processing API. Flink does so by providing specialized API, data structure and algorithms and dedicated scheduling strategies [21].

Because of Flink's distributed architecture, its setup is a combination of processes running across a cluster of machines. As a result, it must solve the usual challenges of running distributed applications, such as resource allocation and management, persistent data storage management and failure handling. Instead of implementing these functionalities internally, Flink utilizes existing cluster management platforms like YARN and K8S. It also gives the option to be run with a stand-alone configuration [21].

A Flink setup is a combination of four components: a JobManager, a ResourceManager, a TaskManager, and a Dispatcher.

The JobManager is the master process within Flink's architecture, responsible for running a single Flink application. Flink creates a new instance of JobManager for each new application. The application that JobManager receives to execute is a combination of three components, a JobGraph, a logical data flow graph, and a JAR file consisting of all the necessary classes, libraries, and external resources that the application requires to run. The JobManager turns the JobGraph into a physical dataflow graph (called ExecutionGraph), a combination of tasks that can be performed in a parallel. After the creation of the ExecutionGraph, the JobManager sends a request to the ResourceManager to schedule the required resources for the parallel execution of the tasks. After allocation of enough slots, it ships the tasks to the TaskManager to be executed [22].

Flink can be deployed on top of different resource managers such as Apache Mesos, K8S, or Apache YARN. The role of resource manager in Flink's architecture is to provide Task Slots from a TaskManager to the Job Manager upon request. If a task manager cannot offer the required number of slots, the ResourceManager will contact a resource provider to spin up new containers [22].

The Dispatcher acts as the entry point in Flink's architecture by providing a REST interface for submission of applications. It transfers the submitted applications to the JobManager for execution.

Figure 3-3 shows the architecture of Apache Flink [23].

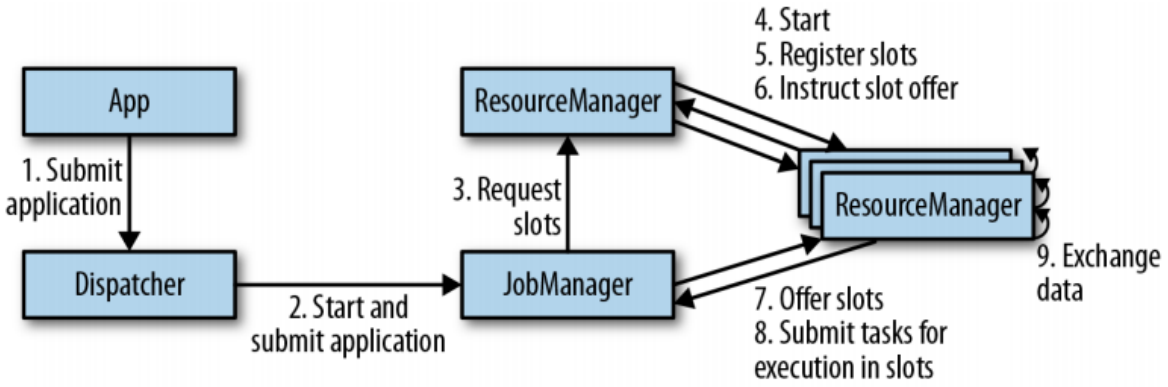

Figure 3-3: Architecture of Flink from [22]

Parallelism in Flink can happen via tasks at three different layers: at the data level the tasks are subtask, at the operator level and at the application level. Figure 3-4 shows the role of slots in the implementation of parallelism in Flink [22].

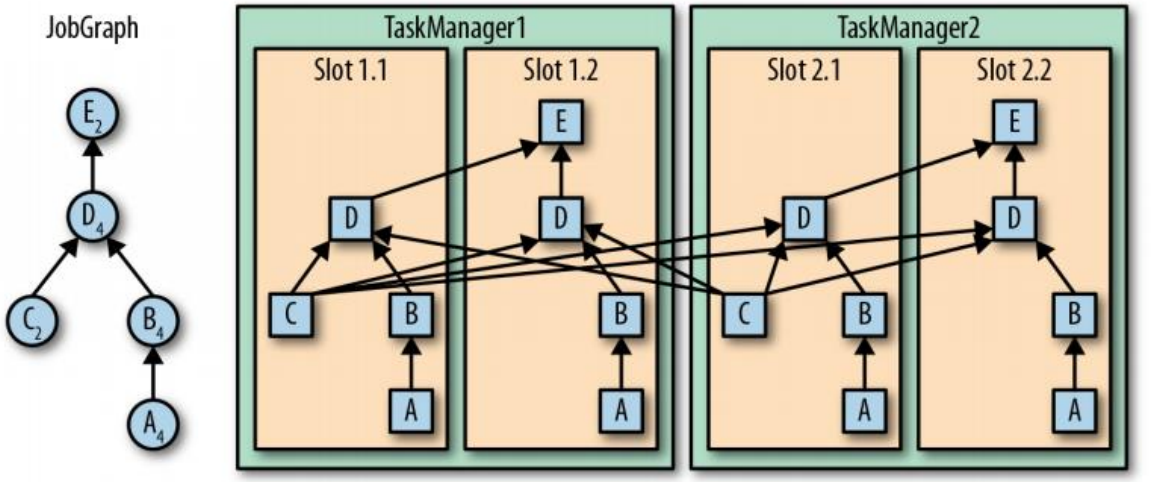

Figure 3-4: Role of slots in application execution in Flink from [22]

A TaskManager uses a single JVM Process and executes tasks via multithreading. By utilizing multithreading instead of separate processes, the TaskManager avoids overhead process swapping and intra-process communication costs.

Flink's execution of stream processing is based on a direct acyclic graph (DAG). This DAG is a combination of two import elements within Flink, stateful operators and data streams. Flink divides operators into subtasks and streams into partitions to achieve true parallelism. The stateful operators are used to perform computation logic on the data streams. These operators (filters, maps, and windowing) are in most cases the direct implementation of their corresponding algorithms. An example data flow graph is shown in Figure 3-5:

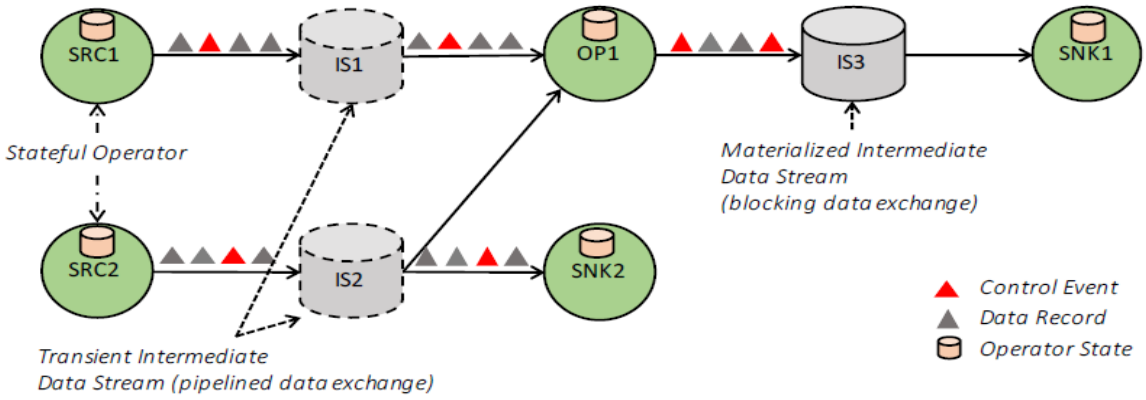

Figure 3-5: An example data graph from [21]

Flink exposes multiple high-level APIs to programmers: DataStream API, Dataset API, Table API & SQL. These APIs support Java and Scala programming languages. Flink also provides a python API. For this thesis, we will only be focusing on the DataStream API.

DataStream programs implement different transformations on the data streams coming into Flink via various sources. Flink programs can run standalone or within other applications. It supports execution on the local JVM or over a cluster [22].

The DataStream API is named based on the DataStream class that it provides. This class represents a collection of data in a Flink program. Objects of DataStream class are immutable and can only be manipulated using the transformations offered by the DataStream API [22].

The concept of time is represented in Flink via two conceptions, the event time and the processing time. The event time is a representation of when the event occurred, and the processing time is a representation of the time the event was processed by the system which is usually marked by the system clock. In addition to these two time concepts, Flink also defines an element called

Watermark. A watermark is a global marker for the progress of data processing within the system. It consists of an attribute t which donates all the events lower than t have already been ingested by the corresponding operator. Watermarks are usually created at the source of the DAG representing the execution of the current application and are passed along through the operators as the processing continues. The utilization of the watermark depends on the complexity of the operators. Simpler operators only forward the watermark over through the execution pass, while more complex operators first perform calculation based on the watermark and then forward it [22].

A Flink program is composed of multiple steps common to data stream programs [22]:

1- Get a copy of the Execution environment instance,
2- Import in the initial data into the program,
3- Implement the transformations on the data from step 2,
4- Specify the destination for the data,
5- Initialize program execution.

In our System, Apache Flink receives the data from Apache Kafka via Flink Kafka Consumer. It then manages data via two channels. The first channel uses Flink data Stream API to perform processing over the streaming time series data. After processing, the data is written to InfluxDB and HDFS. The second channel transfers the raw data received from Kafka directly to HDFS. The data would be used later for further analytics operations using Apache Spark.

3.1.4 InfluxDB

InfluxDB is an open-source time series database developed and maintained by InfluxData. Over time it has turned from a times series management engine to a time series toolkit including time series collection, processing, and visualization components. It is also currently offered as PaaS on Amazon Web Services and Microsoft Azure. Figure 3-6 shows the InfluxDB architecture as a time series management toolkit [23].

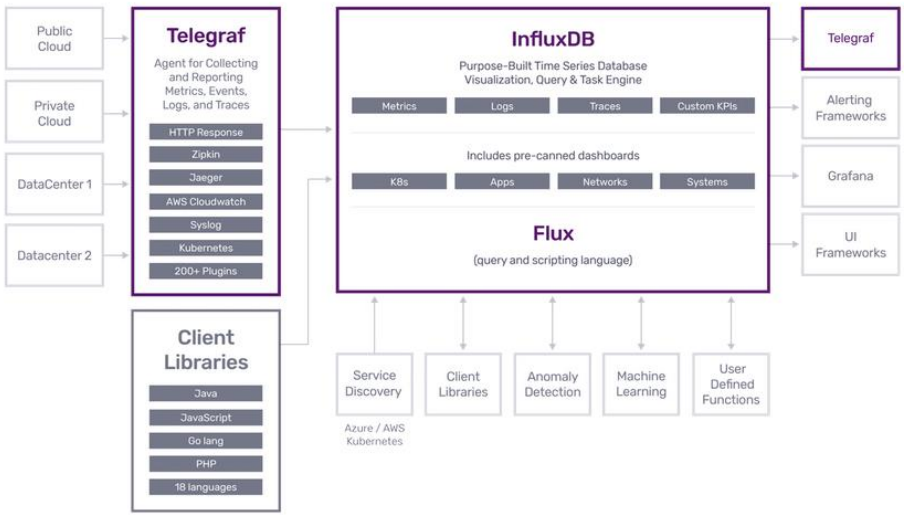

Figure 3-6: InfluxDB Architecture from [23]

To manage time series data, InfluxDB uses a group of data elements as follows [24]:

- Timestamp: InfluxDB requires incoming data to have a timestamp attribute which will be stored in time column.
- Measurement: this element represents the entity for which the data is stored in InfluxDB.
- Fields: fields include field key and field value.
- Field key: is the name of the field which is stored in the _field column.
- Field value: is the value for the associate field key.
- Field set: is a group of key-value pairs of associated with the time stamp.
- Tag set: tag sets include additional key-value pair that are store within corresponding columns in InfluxDB.

The key role that InfluxDB plays in our system architecture is fast storage and retrieval of fresh data. The data expires after a certain period, depending on the data application domain requirements using InfluxDB's retention policy configuration parameter. The second role of InfluxDB is acting as a data source for data visualization in the system. This connection is explained later in this chapter.

3.1.5 HDFS

The Hadoop Distributed File System (HDFS) is a highly scalable and fault tolerant distributed file system. It was designed to execute on low-cost hardware and provide high throughput IO operations [25]. It has a master/slave architecture consisting of a single Namenode and a collection

of Datanodes. The Namenode plays the role of the master and the Datanodes play the role of the slaves. The architecture of HDFS is shown in Figure 3-7 [25].

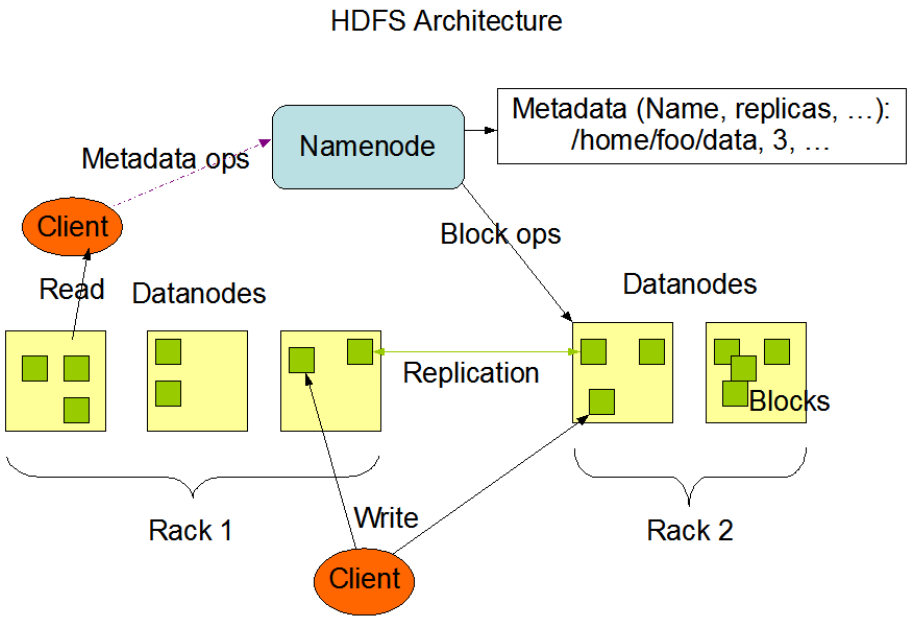

Figure 3-7: HDFS Architecture from [25]

HDFS is used as the long-term storage in our proposed system. Using HDFS in our system provides three major benefits. First, it is a distributed file system that allows for scalability as the size of the incoming data grows over time. This is a very important design decision since this system is designed for scalable management of time series data. Second, HDFS could be deployed to commodity servers thus lowering the overall cost of the system. Third, since it is the de facto file system for Hadoop platform, it allows for extending the system via integration of other tools from Hadoop ecosystem to provide additional functionalities.

3.1.6 Apache Spark

"Apache Spark is a unified analytics engine for data processing at large-scale" [26]. The term unified refers to the fact that Spark provides structured and semi structured data processing, batch processing, stream processing, machine learning algorithms libraries and graph processing. It provides high-level APIs for Java, Scala, Python, and R. Its main components include Spark SQL for structured data processing [27], MLlib for machine learning [28], GraphX for graph processing [29] and Spark Streaming for stream processing [30].

In our architecture Apache Spark is used as the machine learning engine. It has access to the raw data and processed data residing on HDFS. It will read data from HDFS and use it as the input to the ML algorithms provided via MLlib library and writes the model results to HDFS.

3.1.7 Grafana

Grafana is an open-source metrics visualization tool with connectors to a large collection of data providers. It offers a variety of visualization such as bar charts, histograms, heat maps etc. It is used for time series data visualization in the system. It provides a dedicated plugin to integrate with InfluxDB and could provide live views of the data stored on influx. The visualization could be refreshed on certain intervals to make sure they are representing the most recent data. Depending on the data generation domain, different types of visualization could be created in Grafana to showcase various attributes of the data [31].

3.2 System Workflow

The entry point to the system consists of Kafka producers. The producers collect the data from potentially distributed data sources and pass it on to a Kafka topic. The topic transfers the data to Flink Kafka consumer. In the next step, within Flink, two separate data pipelines are formed. The first pipeline transfers the raw data coming in from Kafka directly to HDFS. The second pipeline performs computations over the streaming time series data. After computations, the processed data is transferred via two channels to both InfluxDB and HDFS. Within InfluxDB the data could be queried further using InfluxDB's query processing language. Next Grafana will read the data from Influx using certain queries to generate dashboards.

The raw data coming into the system that was written to HDFS by Flink would then be used as the training dataset for machine learning algorithms from MLlib based on the data application domain. The result of the machine learning algorithm would then be subsequently written back to HDFS.

3.3 System Functionalities

Stream processing over time series data: Apache Flink is the only stream processing engine that performs on the fly data processing. It provides a comprehensive stream processing API called DataStream API. One of the major contributions to the power of Flink in time series data comes from the implementation of windows. This makes Flink an ideal choice for performing computations over unbounded time series data.

Data Analytics and Machine learning: The system architecture enables the conduction of data analytics in two different forms. The first form is performed through the most recent data stored in InfluxDB via aggregations and identification of outlier values. Within the fast storage environment provided by InfluxDB, the user can use the inherent SQL-like query language, called InfluxQL [32] or the functional language named Flux [33], to query and process data. An example of this form of analytics would be identification of VMs running in a datacenter with 95^{th} percentile CPU usage to be specifically monitored. A second example would be detection of changes in the average speed of vehicles in the past 5 minutes in a specific part of a highway to search for and detect potential collisions. The second form of analytics derives from the power of Apache Spark's MLlib. The most common type of analytics performed on time series is forecasting. Spark's MLlib library provides a set of machine learning algorithms suitable for forecasting on the processed or raw data stored in HDFS. An example for this type of analytics would be forecasting future CPU consumption of long running VMs (defined by a specific threshold) for capacity planning purpose for the data center. A second example would be forecasting the energy consumption of large physical infrastructure like a shopping mall based on the electricity consumption of the individual stores and the common area to prepare for specific times of the year like the Christmas shopping. On an even larger scale the system could be used to do forecasting for electricity consumption in a North American City by the corresponding power company to prepare for surge caused by simultaneous usage of TV during Baseball, Hockey or Football matches.

4. Implementation of the System

In this chapter we discuss different aspects of the implementation of the system. For the purpose of this thesis, we only built the part of the system including Kafka, Flink, InfluxDB and Grafana. This is a work in progress, and we will be adding HDFS and Spark's MLlIB in the future.

4.1 Infrastructure

To create the infrastructure needed for the implementation of the system, we used Compute Canada (CC)'s cloud service. CC's cloud runs on Opens Stack platform. CC's Open Stack provides predefined flavors for VMs. We selected Ubuntu as the operating system of choice. The resource specifications for these VM are listed in the table 2 below:

VM	CPU	RAM	DISK
Kafka	2 VCPU	15 GB	95 GB
Flink	4 VPCU	15 GB	165 GB
Influx and Grafana	2 VCPU	7.5 GB	56 GB

Table 2: List of the VMs used for implementation of the system

Each VM was assigned a private IP address. The connections to the VM were made via SSH by using the tool Putty on Windows 10. Open Stack implements access to the VMs via Security Groups (SG). The default SG provided by CC was modified to provide access to the VMs only Via SSH, HTTP and HTTPS ports.

4.2 Development

In this section we describe different aspects of the development of the system.

4.2.1 Kafka

We downloaded Apache Kafka 2-12_2.6.0 and installed in on the Kafka VM. We decided to choose Java as the programming language of choice to implement the Kafka application. IntelliJ Idea was used as the IDE to implement the application. In order to manage dependencies within the Java project we used Maven, a java package dependency management framework. The dependencies shown in Figure 4-1 were used in our Java project.

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.6.0</version>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.30</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.3</version>
</dependency>
<dependency>
    <groupId>com.opencsv</groupId>
    <artifactId>opencsv</artifactId>
    <version>4.5</version>
</dependency>
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.10.0</version>
</dependency>
```

Figure 4-1: Kafka Maven project dependencies

4.2.2 Flink

We downloaded Apache Flink 1.11.1 and installed in on the Flink VM. We decided to choose Java as the programming language of choice to implement the Kafka application. IntelliJ Idea was used as the IDE to implement the application. To manage dependencies within the Java project we used Maven. The dependencies shown in Figure 4-2 were used in our Java project.

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-java</artifactId>
    <version>1.11.2</version>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_2.12</artifactId>
    <version>1.11.2</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-clients_2.12</artifactId>
    <version>1.11.2</version>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_2.12</artifactId>
    <version>1.11.2</version>
</dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-kafka_2.11</artifactId>
        <version>1.10.3</version>
    </dependency>
```

Figure 4-2: Flink Maven project dependencies

4.2.3 InfluxDB

We downloaded and Installed InfluxDB 2.0 on Influx and Grafana VM.

4.2.4 Grafana

We downloaded and installed Grafana version 7.2.0 on the Influx and Grafana VM and configured Grafana to connect to InfluxDB.

5. Experimentation

In this chapter we discuss the experimentation performed to measure the feasibility and scalability of the system.

5.1 The source dataset

We used Azure Public Dataset version 2.0 as our source data set. This is an open-sourced data set that could be found in [34]. The publication of this dataset is the result of the work published in [35]. This dataset contains a set of CPU readings from all the VMs from all azure subscription for a month starting from 00:00:00 AM of the first day of the month until 11:59:59 of the last day of the month. This dataset is provided as a set of 195 files with the CSV format totaling approximately 235 GB. Each row of the files contains the timestamp, hashed string of the VM ID, a min CPU, average CPU and max CPU for 5-minute time intervals. It is important to note that our system is designed to handle streaming time series data. We are using static files as a simulation because they contain real data that could be used as streaming data source for our system in a production environment.

5.2 Processing and visualization of data

To show that our system is capable of successfully ingesting and processing time series data we created a data pipeline in Kafka to ingest data from the source files. We parsed the file line by line within Kafka and passed each line to our Kafka producer. The producer then transmits the data to a topic with the replication factor of 1, because we are running Kafka in the stand-alone mode. We then connect to the Kafka broker from Flink and start reading Kafka records from the topic. In Flink, we created a window of 15 minutes for each VM based on the VM ID and calculate the average CPU utilization in the VM for the corresponding window. These calculated averages are then sinked to an InfluxDB bucket. Grafana will read the records from the bucket in InfluxDB and create visualization. Figure 5-1 shows an example graph from average CPU within Grafana.

Figure 5-1: CPU usage graphs in Grafana

Figure 5-2 shows a zoomed in graph from average CPU consumption chart in Grafana.

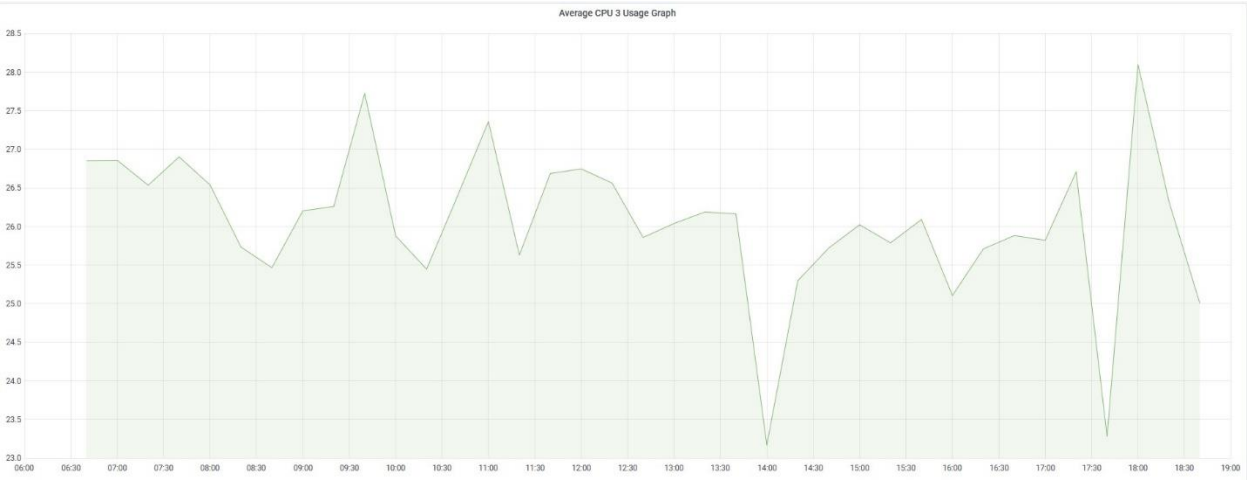

Figure 5-2: Zoomed-in graph from Grafana

5.3 Measuring the scalability of the system

To examine the scalability of the system, we initially monitored CPU usage of the user space and system space of the VM hosting Flink. This decision was based on Flink being the main component of the system responsible for stream processing of time series data in our proposed system architecture. It is important to note that based on the architecture of the system discussed in chapter 3, the system is hosted on Compute Canada cloud environment. The are no additional user

applications hosted on any of the VMs except the applications necessary for the purpose of building the system. This makes the user space and system space CPU usage within the system, a very good representation of Flink's behaviour in terms of CPU utilization and how it manages its relationship with system components. For the experiments, we tested the CPU usage with increasing number of files read by Kafka and consumed by Flink in 4 steps, doubling the numbers each step. We started with 1 file and performed our tests with 2, 4, and 8 files subsequently. Each of the files have approximately 1000000 rows. The files were read row by row via concurrent threads and the Kafka records generated from each row were passed to a single Kafka producer since Kafka producer is thread safe.

Figure 5-3show CPU usage of the user space of the Flink VM for a 5-minute interval.

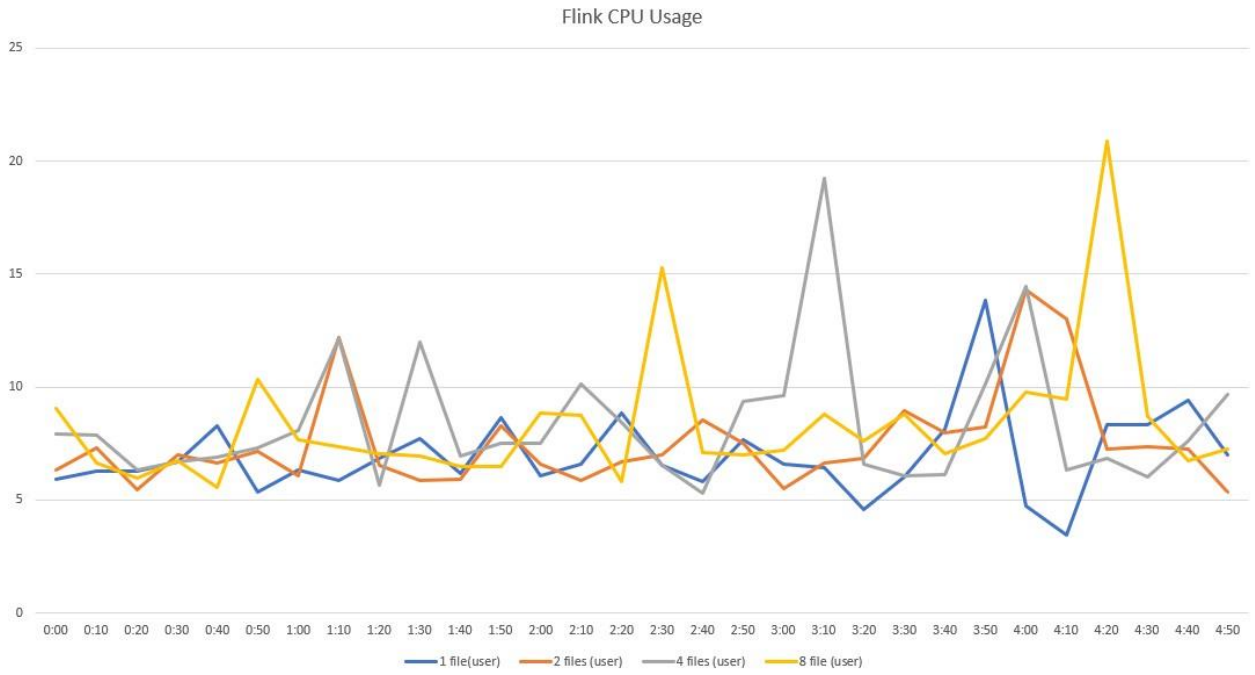

Figure 5-3: User Space CPU usage in Flink VM

Figure 5-4 shows CPU usage of the system processes on the VM for the same 5-minute interval of the corresponding user space for each test.

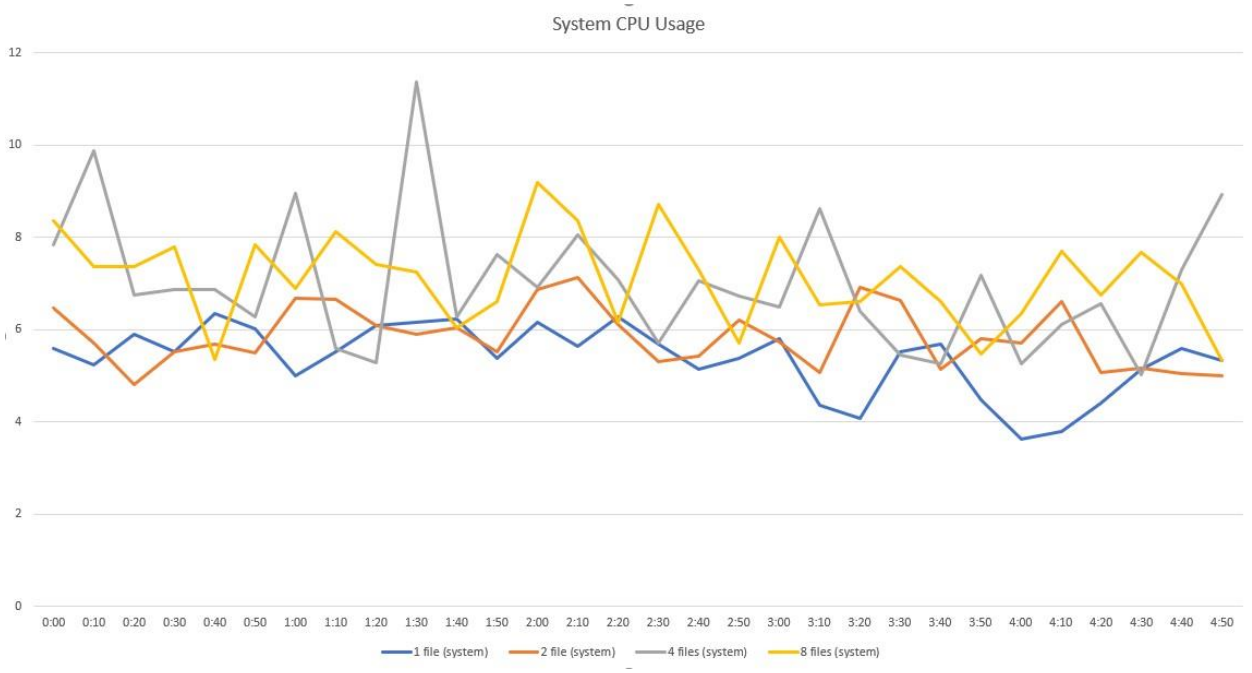

Figure 5-4: System CPU usage in Flink VM

Figure 5-3 shows an over all predictable pattern for the application. Each line represents periods of regular activity with spikes happening in intervals. Each of the tests (with different file numbers) show the same pattern. The spikes are caused by Flink's buffering mechanism that is in place for improving the network performance by utilizing the buffers to avoid wasting bandwidth through transfer of non-full IP packets.

Figure 5-4 shows a predictable pattern of system space CPU usage. Although there are some spikes such as one happening at 1:30 for the 4 files test, these occasional spikes are negligible as the all the tests show a linear increase in CPU consumption as we increased the amount of incoming data.

At the next stage of the experimentation, we introduced intervals into the incoming data as the amount of data coming into the system increased. For each number of producers with performed experimentations with 0.1 second intervals.

Figure 5-5 shows CPU usage of Flink with 1 Kafka producer and 0.1 second interval:

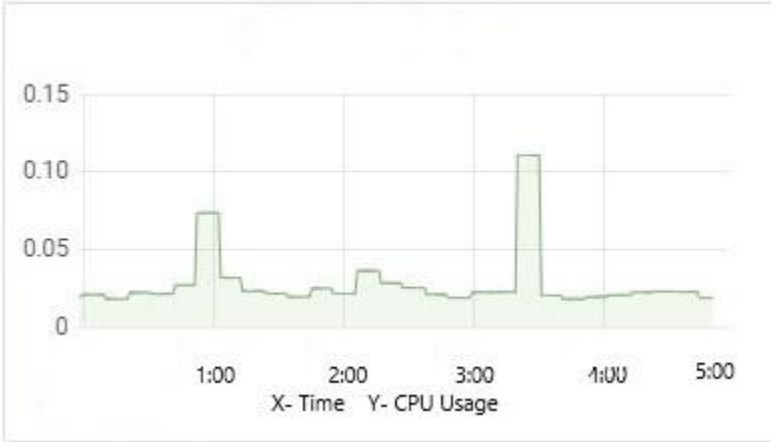

Figure 5-5: Flink CPU usage with 1 Kafka producer and 0.1 second interval in data arrival

Figure 5-6 shows CPU usage of Flink with 2 Kafka producer and 0.1 second interval:

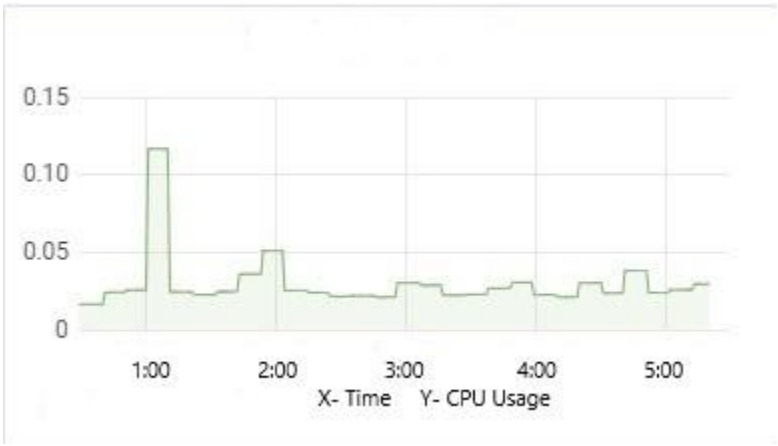

Figure 5-6: Flink CPU usage with 2 Kafka producers and 0.1 second interval in data arrival

Figure 5-7 shows CPU usage of Flink with 4 Kafka producer and 0.1 second interval:

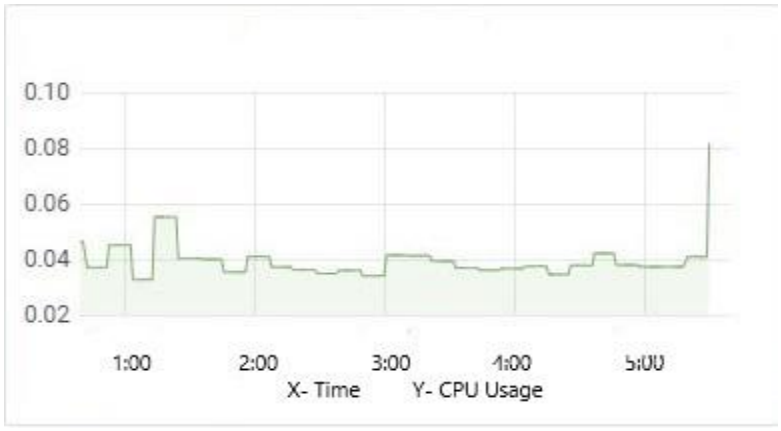

Figure 5-7: Flink CPU usage with 4 Kafka producers and 0.1 second interval in data arrival

Figure 5-8 shows CPU usage of Flink with 8 Kafka producer and 0.1 second interval:

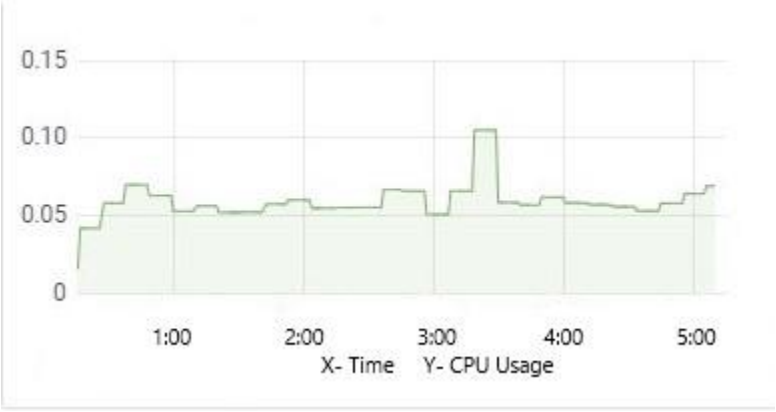

Figure 5-8: Flink CPU usage with 8 Kafka producers and 0.1 second interval in data arrival

The second set of experiments with the introduction of the intervals into data arrival show the same pattern of resource usage consumption in Flink as the first set of experiments. It can be seen in the figures 5-5, 5-6, 5-7 and 5-8 that CPU usage over time is linear and includes occasional spikes which quickly disappear. This resource consumption behaviour shows that our proposed system is capable of scaling as the amount of time series data entering the system increases.

6. Conclusion

In this thesis we focused on scalable time series data processing and management. We comprehensively looked at the importance of time series data and its application and generation domains. We reviewed the related work in the academic literature and discussed their strengths and shortcomings. Our study showed that many of the TSMSs that have been developed are still using the practices from the domain of relational data management and how newer systems are proposing more novel approaches to time series data. One of the major results of our review was the need for any new TSMS to have a distributed architecture to be able to scale to the volume and velocity of big time series data.

We then proposed a system architecture for end-to-end time series data processing and management, by using open-source data processing frameworks that suit our goal of time series data processing. We then implemented the system and showed that our system is capable of time series data management from generation to consumption by the end user. We next performed experiments to investigate the scalability of our proposed system. The results of the experiments showed the ability of the core of our system, Apache Flink, to scale properly. Based on the results of the implementation of and the experiments on our system we believe that our proposed system is capable of management of time series data at scale regardless volume of the of data coming into the system

For future work we plan on three different works. First, we plan to investigate different techniques and mechanism for preprocessing of time series data before it reaches the stage at which live computations are performed on it. This is an important task since time series data could have missing data points or be out of order. Although Flink has a watermark mechanism to deal with out of order data, this could help with increasing the performance of computations. Second, we are planning to investigate the feasibility of running Apache Kafka on edge devices and how it would help in reducing the amount of data loss in IoT platforms in resource constrained environments. Third we plan to investigate current compression techniques for time series data and design and implement a new storage layer on top of HDFS optimized for storage of historical time series data and increasing the performance of analytical queries on a wide time range on that data. As the

amount of time series data increases this would be an inevitable approach to reduce the overall cost of time series data processing.

7. References

[1] S. K. Jensen, T. B. Pedersen and C. Thomsen, "Time Series Management Systems: A Survey," in IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 11, pp. 2581-2600, 1 Nov. 2017.

[2] Y. Tsubouchi, A. Wakisaka, K. Hamada, M. Matsuki, H. Abe and R. Matsumoto, "HeteroTSDB: An Extensible Time Series Database for Automatically Tiering on Heterogeneous Key-Value Stores," 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 2019, pp. 264-269.

[3] L. Deri, S. Mainardi, and F. Fusco, “tsdb: A compressed database for time series,” in Proc. 4th Int. Workshop Traffic Monitoring Anal., 2012, pp. 143–156.

[4] A. MacDonald, “PhilDB: The time series database with built-in change logging,” PeerJ Comput. Sci., vol. 2, p. e52, 2016, Art. no. 18.

[5] Y. Katsis, Y. Freund, and Y. Papakonstantinou, “Combining databases and signal processing in plato,” in Proc. 7th Conf. Innovative Data Syst. Res., 2015.

[6] J. L. P_erez and D. Carrera, “Performance characterization of the servIoTicy API: An IoT-as-a-service data management platform,” in Proc. 1st Int. Conf. Big Data Comput. Serv. Appl., 2015, pp. 62–71.

[7] A. Marascu, et al., “TRISTAN: Real-time analytics on massive time series using sparse dictionary compression,” in Proc. Int. Conf. Big Data, 2014, pp. 291–300.

[8] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, “Bolt: Data management for connected homes,” in Proc. 11th Symp. Netw. Syst. Des. Implementation, 2014, pp. 243–256

[9] T. Ilsche, D. Hackenberg, R. Schöne, M. Bielert, F. Höpfner and W. E. Nagel, "MetricQ: A Scalable Infrastructure for Processing High-Resolution Time Series Data," *2019 IEEE/ACM Industry/University Joint International Workshop on Data-center Automation, Analytics, and Control (DAAC)*, 2019, pp. 7-12

[10] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. “Gorilla: a fast, scalable, in-memory time series database”. Proc. VLDB Endow. 8, 12 August 2015, pp. 1816–1827.

[11] Charles Loboz, Slawek Smyl, and Suman Nath. 2010. “DataGarage: warehousing massive performance data on commodity servers”. Proc. VLDB Endow. 3, 1–2 (September 2010), pp.1447–1458.

[12] Y. Yang, Q. Cao and H. Jiang, "EdgeDB: An Efficient Time-Series Database for Edge Computing," in IEEE Access, vol. 7, pp. 142295-142307, 2019.

[13] M. Buevich, A. Wright, R. Sargent and A. Rowe, "Respawn: A Distributed Multi-resolution Time-Series Datastore," 2013 IEEE 34th Real-Time Systems Symposium, Vancouver, BC, Canada, 2013, pp. 288-297.

[14] M. P. Andersen and D. E. Culler, “BTrDB: Optimizing storage system design for timeseries processing,” in Proc. 14th Conf. File Storage Technol., 2016, pp. 39–52.

[15] S. Di Martino, L. Fiadone, A. Peron, A. Riccabone and V. N. Vitale, "Industrial Internet of Things: Persistence for Time Series with NoSQL Databases," 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Napoli, Italy, 2019, pp. 340-345.

[16] S. K. Jensen, T. B. Pedersen, and C. Thomsen, ``ModelarDB: Modular model-based time series management with spark and cassandra," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1688_1701, 2018.

[17] Chang Wang, Yongxin Zhu, Weiwei Shi, Victor Chang, P. Vijayakumar, Bin Liu, Yishu Mao, Jiabao Wang, and Yiping Fan. 2018. “A Dependable Time Series Analytic Framework for Cyber-Physical Systems of IoT-based Smart Grid”. ACM Trans. Cyber-Phys. Syst. 3, 1, Article 7 January 2019

[18] Nikolay Laptev, Saeed Amizadeh, and Ian Flint. 2015. “Generic and Scalable Framework for Automated Time-series Anomaly Detection”. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD '15). Association for Computing Machinery, New York, NY, USA, 1939–1947.

[19] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. “Building a replicated logging system with Apache Kafka”. Proc. VLDB Endow. 8, 12 August 2015, pp. 1654–1655.

[20] https://kafka.apache.org/documentation/#introduction

[21] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. et al. (2015) “Apache flink: Stream and batch processing in a single engine”. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4)

[22] Hueske F, Kalavri V. “Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications”. O'Reilly Media; 2019 Apr 11.

[23] https://www.influxdata.com/time-series-platform/

[24] https://docs.influxdata.com/influxdb/v2.0/reference/key-concepts/data-elements/

[25] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[26] https://spark.apache.org/

[27] https://spark.apache.org/sql/

[28] https://spark.apache.org/mllib/

[29] https://spark.apache.org/graphx/

[30] https://spark.apache.org/streaming/

[31] https://grafana.com/

[32] https://docs.influxdata.com/influxdb/v2.0/query-data/influxql/

[33] https://docs.influxdata.com/influxdb/v2.0/query-data/get-started/

[34] https://github.com/Azure/AzurePublicDataset

[35] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms”. In Proceedings of the 26th Symposium on Operating Systems Principles(SOSP '17). Association for Computing Machinery, New York, NY, USA, 153–167.