Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Muhammad Abdullah Khan

Distributed and scalable parsing solution for telecom network data

Master's Thesis
Espoo, December 22nd, 2019

Supervisor: Professor Tuomas Aura, Aalto University
Advisors: Samuel Marchal, Ph.D., Aalto University
Toivo Vaje, M.Sc. (Tech), Elisa Oyj

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Muhammad Abdullah Khan		
Title: Distributed and scalable parsing solution for telecom network data			
Date:	December 22nd, 2019	**Pages:**	viii + 69
Major:	Security and Cloud Computing	**Code:**	SCI3084
Supervisor:	Professor Tuomas Aura		
Advisors:	Samuel Marchal Toivo Vaje		

The growing usage of mobile devices and the introduction of 5G networks have increased the significance of network data for the telecom business. The success of telecom organizations can depend on employing efficient data engineering techniques for transforming raw network data into useful information by analytics and machine learning (ML).

Elisa Oyj., a Finnish telecommunications company, receives massive amounts of network data from network equipment manufactured by various vendors. The effectiveness of data analytics depends on efficient data engineering processes. This thesis presents a scalable data parsing solution that leverages Spark, a distributed programming framework, for parallelizing parsing routines from an existing parsing solution. We design and deploy this solution as a component of the organization's data engineering pipeline to enable automation of data-centric operations.

Experimental results indicate that the efficiency of the proposed solution is heavily dependent on the individual file size distribution. The proposed parsing solution demonstrates reliability, scalability, and speed during empirical evaluation and processes a 24-hour network data within 3 hours. The main outcome of the project is an optimized setup with the minimum number of data partitions to ensure zero failures and thus minimum execution time. A smaller execution time leads to lower costs of the continuously running infrastructure provisioned on the cloud.

Keywords:	Cloud Computing, Distributed Computing, Big Data, Data Engineering, Spark, Data Parsing
Language:	English

Acknowledgments

To begin, I praise God for blessing me with rich learning experiences in a multi-cultural environment for expanding my intellectual horizon. Next, I am utterly grateful to my parents for believing in me and sacrificing their precious time and wealth to ensure me being where I am today.

I would especially like to thank Prof. Tuomas Aura for putting faith in me and offering his supervision when others were not so willing. Moreover, I am indebted to Samuel Marchal's constant efforts in refining my literary work despite his busy schedule. Lastly, I am thankful to Toivo Vaje and all my colleagues at Elisa for facilitating me at the office, addressing all my work-related queries and guiding me whenever I lost my way.

Espoo, December 22nd, 2019

Muhammad Abdullah Khan

Abbreviations and Acronyms

3GPP	3rd Generation Partnership Project
AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
BDaaS	Big Data as a Service
BDIaaS	Big Data Infrastructure as a Service
BDPaaS	Big Data Platform as a Service
BDSaaS	Big Data Software as a Service
BI	Business Intelligence
CM	Configuration Management
CPU	Central Processing Unit
DL	Deep Learning
DOM	Document Object Model
EA	Elisa Automate
EB	Exabytes
EC2	Elastic Compute Cloud
ETL	Extract, Transform and Load
GAE	Google AppEngine
GCE	Google Compute Engine
GCS	Google Cloud Storage
HDFS	Hadoop Distributed File System
I/O	Input and Output
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
ML	Machine Learning
MO	Managed Object
NE	Network Element
OS	Operating System
OSS	Operations Support System

PaaS	Platform as a Service
PM	Performance Management
RAM	Random Access Memory
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
SaaS	Software as a Service
SAX	Simple API for XML
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
TB	Terabytes
UI	User Interface
VPC	Virtual Private Cloud
WWW	World Wide Web
XML	Extensible Markup Language
YARN	Yet Another Resource Negotiator

Contents

Abbreviations and Acronyms iv

1 Introduction 1
1.1 Motivation . . . 1
1.2 Industrial context . . . 2
1.3 Goals and contributions . . . 3
1.4 Thesis structure . . . 4

2 Background 5
2.1 Data engineering . . . 5
2.1.1 Definition and purpose . . . 5
2.1.2 Data engineering pipeline workflow . . . 6
2.1.2.1 Ingestion . . . 7
2.1.2.2 Storage . . . 7
2.1.2.3 Processing . . . 7
2.1.2.4 Delivery for analytics . . . 8
2.2 Big data . . . 8
2.2.1 Characteristics . . . 8
2.2.2 Challenges . . . 10
2.3 Cloud solutions for big data . . . 11
2.3.1 Big data as a service (BDaaS) . . . 12
2.3.1.1 Ingestion . . . 12
2.3.1.2 Storage . . . 13
2.3.1.3 Processing and delivery . . . 13
2.3.1.4 Analytics . . . 14
2.3.1.5 BDaaS service models . . . 14
2.4 Spark: A distributed processing framework . . . 16
2.5 Overview of XML structure and parsing . . . 17
2.5.1 Raw network data . . . 19
2.6 Elisa Automate's (EA) existing data engineering strategy . . . 22
2.6.1 Ingestion . . . 24

2.6.2 Storage . . . 24
2.6.3 Processing . . . 24
2.6.4 Delivery for analytics . . . 24

3 Problem definition **25**
3.1 Motivation . . . 25
3.2 Goal of the solution . . . 26
3.3 System requirements . . . 27
3.3.1 Functional requirements . . . 27
3.3.1.1 F1: Data transmission . . . 27
3.3.1.2 F2: Resource allocation . . . 27
3.3.1.3 F3: Distributed parsing . . . 27
3.3.1.4 F4: Schema transformation . . . 27
3.3.1.5 F5: File-format support . . . 27
3.3.1.6 F6: Automation . . . 27
3.3.2 Non-functional requirements . . . 28
3.3.2.1 NF1: Reliability . . . 28
3.3.2.2 NF2: Scalability . . . 28
3.3.2.3 NF3: Speed . . . 28

4 Solution Design **29**
4.1 System Infrastructure . . . 29
4.2 Distributed Spark parser . . . 31
4.3 Solution workflow . . . 34

5 Experimental setup **37**
5.1 System specifications . . . 37
5.2 Sample datasets . . . 38
5.3 Spark configurations . . . 40

6 Evaluation **45**
6.1 Reliability . . . 45
6.2 Scalability . . . 49
6.3 Speed . . . 51
6.4 Summary . . . 52
6.5 Discussion . . . 54
6.5.1 Next steps . . . 57

7 Related work **59**

8 Conclusion **61**

A Appendices **69**

Chapter 1

Introduction

1.1 Motivation

The growing reliance on mobile devices for daily activities has led to an explosion in mobile traffic. As an example, a telecommunications company in the city of Shenzen, China, serves 10 million users and produces 5 TB of data per day [1]. A study [2] suggests that the total mobile traffic is expected to increase by a factor of 5 over the next six years. Based on the study, the traffic is forecasted to reach 136 Exabytes (EB) per month by the end of 2024. Moreover, traffic generated by smartphones is projected to reach 95 % of the total mobile data traffic by 2024. As a result, conducting network management routines such as fault mitigation, network configuration updates, logging and security patching are becoming increasingly challenging. Network operations are evaluated by certain key performance indicators (KPI), which can be inferred from network data generated by various telecom network elements. The KPIs represent the efficiency and performance of the network, consequently having interest to all stakeholders such as consumers, operators, media and the government. Machine learning (ML) and artificial intelligence (AI) coupled with big data and cloud computing resources have enabled the automation of numerous operations in mobile networks. These operations include network alarm prediction, optimization of cell configurations, and energy capacity planning for various base stations.

A study by McKinsey in 2016 highlighted that only a handful of telecom providers have managed to achieve an incremental profit of over 10% despite having employed ML solutions [3]. The major reasons behind this revolve around the characteristics of organizational data. To completely utilize the potential of ML/AI, it is necessary to refine data in terms of *de-duplication, completeness*, and *complexity.* This results in relatively accurate predictions

and evaluations of the network by the ML models.

A cloud-based data engineering system is a solution capable of addressing these problems. *Data engineering* is defined as the process of turning data to information through utilizing concepts of applied statistics, system theory, and decision theory [4]. The medium of automation for data engineering techniques on the cloud is a *data engineering pipeline*. These pipelines utilize data engineering practices to convert raw data to useful information tailor-made for various analytics use cases. A data engineering pipeline firstly ingests raw data into a centralized storage system. Following this, the data undergoes processing in the form of parsing to reduce its complexity and increase its value. Next, the pipeline transforms the data into a format optimized for high-speed transfer. By employing data engineering strategies, these pipelines enable telecom operators to utilize their massive data feeds and polish them for training ML prediction models. The ML models subsequently generate insight from the information which enables timely decisions beneficial for determining interactions between multiple KPIs. Hence, the effects of various KPIs representing network characteristics such as performance, customer experience, and energy consumption can be deduced. Some benefits offered by data engineering solutions are [5]:

- Empowerment of exploratory analytics without major reliance on IT.
- Increased data resource productivity across the organization.
- On demand access to processed data sources for analysts.
- Enhance organizational data usage via automation.

All the stated benefits emphasize the importance of employing data engineering pipelines in the industry. The next section presents a real-world scenario where data pipelines can be leveraged.

1.2 Industrial context

Elisa Oyj, the first Finnish telecom organization founded in 1882, receives large amounts of network management data from multiple base stations and cellular network elements across Finland and Estonia. All this data offers significant value for monitoring the network's health and devising strategies for fault mitigation and maintaining network energy and resource utilization. Daily data feeds for a single mobile vendor have massive volume and often exceed the hardware specifications of the processing infrastructure. A start-up by the name of Elisa Automate (EA) has been initiated within Elisa to

study and utilize its traditional data engineering systems. The existing infrastructure consists of decentralized on-premise data processing sub-systems developed upon various technology stacks. Currently, the existing systems are incapable of receiving the combined data feed due to their limited computational and storage capacities. To cope with this issue, it is necessary to automate on-premise data engineering processes and migrate them to a centralized cloud-based environment. However, real-world data poses many challenges in the successful implementation of cloud-based data engineering pipelines:

- Ingesting heterogeneous data from multiple cellular network elements into the centralized cloud-based system pipeline.
- Ensuring data uniformity across multiple cellular network elements through parsing routines.
- Employing an optimal storage format for the processed data in terms of efficiency of input and output (I/O) operations.
- Defining aggregation and filtration routines to prepare datasets for ML/AI algorithms.

Considering the challenges, this thesis aims to develop a solution that enables data parsing routines within a data pipeline.

1.3 Goals and contributions

The goal of this thesis is to design and evaluate a minimal scalable parsing solution leveraging infrastructure on the cloud for processing a 24-hour feed of raw network data. The proposed solution is designed as a component that will be integrated into the future data engineering pipeline at Elisa. It must address the computation and storage limitations of the existing parsing solution and be integrated with a centralized storage end-point on the cloud. Additionally, the thesis must provide a solution for the parallelization of existing parsing routines based on the structure of raw data. The thesis makes the following contributions:

- Design and implementation of a parallelized workflow for network data parsing.
- Empirical evaluation of the reliability, scalability, and speed of the proposed parsing solution.

1.4 Thesis structure

Chapter 2 provides the background for data engineering followed by the concept of data pipelines. It subsequently introduces the concept of big data followed by cloud-based solutions designed to process it. Next, it introduces a distributed programming framework utilized for constructing the proposed data parsing solution. This is followed by the definition of the raw network data structure and file format. The chapter concludes by introducing the existing data engineering infrastructure at Elisa. Chapter 3 identifies the limitations of Elisa's existing data parsing solution. Subsequently, the goals of the proposed solution are highlighted in light of these limitations. The chapter concludes by identifying key functional and non-functional requirements for the solution. Chapter 4 introduces the solution as a component of the data engineering pipeline and its underlying infrastructure. Next, the solution workflow is introduced. Moreover, the chapter covers design choices adapted to process the data in the solution. Chapter 5 defines the environmental configuration for the implementation. Chapter 6 presents the experimental results and evaluates the solution in terms of the defined system requirements. It also highlights the impact of the findings in determining the effectiveness of the solution in terms of its non-functional requirements. Chapter 7 provides a brief insight into existing research work and their impact on the solution. It also provides a comparison of these works with the solution. Chapter 8 finally concludes the thesis by summarizing the research outcomes and improvements beneficial for future research relevant to scalable data parsing on the cloud.

Chapter 2

Background

This chapter provides background information for data engineering techniques and cloud-based solutions comprising parallel processing frameworks for big data. Section 2.1 firstly defines data engineering and the Extract, Transform and Load (ETL) process. Next, Section 2.2 introduces the concept of *big data* and lists all the characteristics and challenges it poses for telecom organizations in performing analytics. Section 2.3 then introduces cloud-based solutions for handling big data via data engineering pipelines. Following this, Section 2.4 presents an overview of the distributed processing framework and its underlying components utilized in the proposed parsing solution. Subsequently, Section 2.5 introduces the raw network data file format supported by its parsing mechanisms. Finally, Section 2.6 discusses the current data engineering infrastructure at Elisa Automate (EA).

2.1 Data engineering

2.1.1 Definition and purpose

Data engineering is defined as the conversion of raw data into refined information. This process ensures production readiness of data along with its underlying formats, resilience, structure, scaling and security, etc. Data engineering revolves around Extract, Transform and Load (ETL), a process responsible for transforming and assembling freshly acquired data into a suitable format for subsequent analysis tasks. Core tasks involved in ETL include data parsing and transformation. Numerous data-centric tasks such as quality assurance, aggregation from multiple sources, reproducible parsing processes, and managing data provenance [6] are dependent upon these core tasks.

Traditional data engineering systems combine ETL processes with resource-intensive post-processing and sanitization techniques to transform raw data into structured information. The exponential increase in data volume has resulted in organizations employing data engineering pipelines to conduct fundamental ETL operations. A data engineering pipeline is defined as a software platform designed to facilitate automated ETL processes. Data engineering pipelines are responsible for transforming raw data collected from multiple sources into insightful information. The pipeline subsequently channels this refined information to targeted consumers comprising business users, data scientists and application frameworks [7] for ML.

2.1.2 Data engineering pipeline workflow

Most modern data engineering pipelines employed in large organizations have a common workflow that involves end-to-end components from retrieving raw data to supplying refined information to analytics systems. A high-level depiction of this workflow is provided in Figure 2.1.

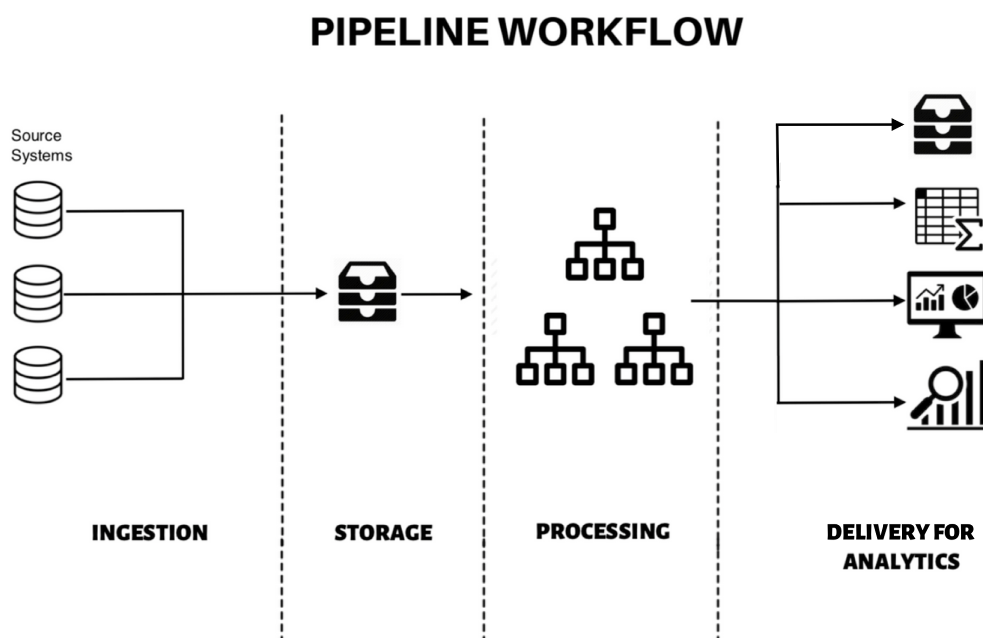

Figure 2.1: A high-level workflow diagram of a modern data engineering pipeline.

The upcoming sections 2.1.2.1, 2.1.2.2, 2.1.2.3, and 2.1.2.4 introduce the basic components of the data engineering pipeline.

2.1.2.1 Ingestion

Ingestion is the process of provisioning data from external sources to a single target location in a unified and consistent format [8]. Enterprises have numerous hardware devices with varying specifications. The multi-sourced raw data is heterogeneous and exists in various formats such as real-time streams, files, and relational databases. The ingestion process can operate in batch-mode or stream-mode depending upon the data format. Data engineering pipelines can also be employed with combinations of multiple operational modes known as lambda architectures.

2.1.2.2 Storage

Currently, the processing of massive datasets from telecom networks is physically impossible to perform on commercial Relational Database Management Systems (RDBMS). Due to this, data engineering pipelines employ distributed storage architectures. These architectures are inspired by classic multi-tier database application architectures which include partitioning, replication, and the distributed control and caching architecture presented by D. Kossmann et al [9]. From a business perspective, the data engineering pipeline should provide a dynamic and distributed means of storage for data. Enterprise data engineering pipelines commonly demand object-store capabilities enabled across multiple machines. These object stores must provide performance and availability to facilitate frequent data access. Moreover, they must offer low latency and high throughput. Finally, the object-stores must be resilient in handling various application scenarios.

2.1.2.3 Processing

Processing of data involves synchronization, parsing, and indexing operations based on important variables such as time, location and other data descriptors [10]. These operations reduce inconsistencies in the data and enforce a uniform structure upon it by applying pre-defined schemas. As a result, query optimization is enabled along with the loading of the data into distributed data structures provided by parallel programming frameworks. Moreover, data is subjected to aggregation and filtration operations which generate a subset of the original data suitable for specific AI/ML use cases.

2.1.2.4 Delivery for analytics

Refined subsets generated from the processing phase are subsequently persisted in scalable object stores. The last phase of the data engineering pipeline concludes with the provisioning of the post-processed data from the object stores to relevant analytics and visualization endpoints such as ML and deep learning (DL) models, relational databases and Business Intelligence (BI) tools, etc.

2.2 Big data

Since the inception of the world wide web (WWW), the amount of data has surged beyond the processing capabilities of modern stand-alone systems. The term *big data* was coined to signal the need for a novel paradigm for processing data of this magnitude [11]. According to the HACE theorem [12], big data is characterized by large-volume and heterogeneous and autonomous sources that have distributed and decentralized control. It holds the key to success for major companies and economies in today's technological era. The following subsections introduce the concept of big data by highlighting the characteristics and some of the challenges it introduces into data engineering systems.

2.2.1 Characteristics

Big data is characterized by a multi-V model or the 5Vs of big data which are *Velocity, Variety, Volume, Veracity* and *Value* as depicted in Figure 2.2 [13]:

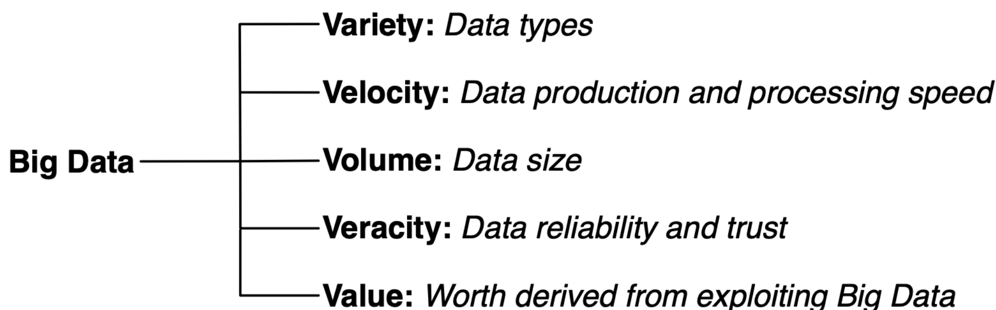

Figure 2.2: Multi-V model describing big data [13].

1. **Value** refers to the process of discovering hidden value from massive datasets of different types [14].
2. **Volume** is the cumulative measurement of different types of data generated from various sources and continuously expanding. This accumulation of large volumes of data can lead to the discovery of hidden information and patterns through data analysis, which in turn contributes to the value of data [14].
3. **Velocity** refers to the rate at which data is produced and subsequently processed. The contents of data are subject to constant change due to the continuous ingestion of independent data collections such as streamed data from multiple sources, archived data, and legacy data [14].
4. **Variety** refers to the various forms of data accumulated via sensors, mobile phones, and social networks e.g. video, image, text, audio, and data logs, in both structured and unstructured formats [14].
5. **Veracity** highlights two aspects of data. Firstly, it concerns the statistical reliability of the data. Secondly, it focuses on the origin of data in addition to its collection and processing methods [15].

In consideration of the multi-V model, each aspect of data determines the design choices for the data engineering pipeline. Firstly, for *variety*, expected data formats need to be identified. Common data formats listed in Figure 2.3 are usually encountered in enterprise systems.

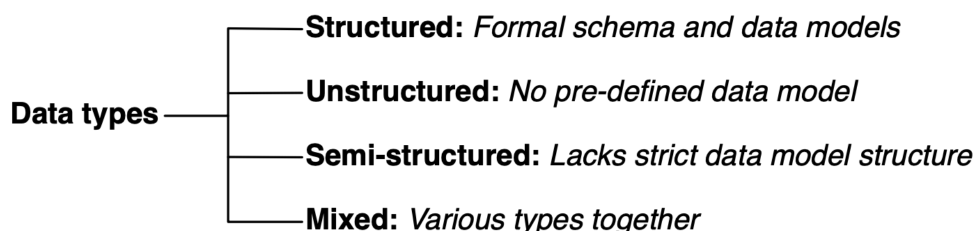

Figure 2.3: Common data formats encountered at organizational levels [13].

Secondly, the rate of data influx into the system as highlighted in Figure 2.4 defines the *velocity* of data. Since data arrives from multiple sources and possesses heterogeneous formats, integrating all this data into a single

analytics solution poses major problems. As highlighted in existing work [16], standard format and interface definition are crucial for efficient integration of data into the analytics solution.

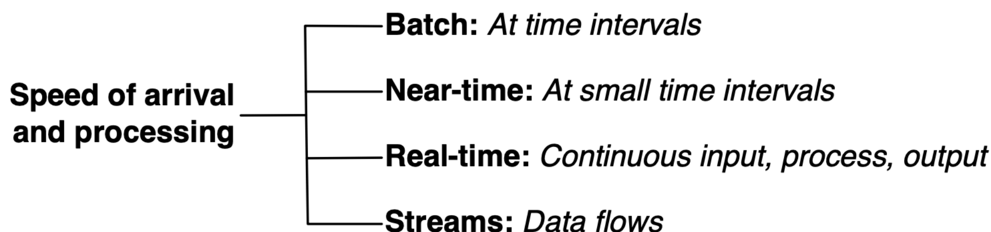

Figure 2.4: Data flow rates [13].

2.2.2 Challenges

The characteristics of big data discussed in the previous section contribute to many challenges in migrating existing systems to the cloud environment. Some of those challenges are:

1. *Availability* refers to the accessibility of system resources when demanded by authorized users. Ensuring sufficient bandwidth for data access is a major issue. As businesses evolve, real-time data access will only increase, which means that the existing infrastructure should comprise additional backup resources to avoid bottlenecks or system failure [17].

2. *Scalability* is the ability of the data engineering system's storage and compute resources to scale up and out in proportion to the datasets, which are expected to grow rapidly in volume.

3. *Data integrity* refers to the ability of data modification only by authorized personnel. The main challenge of data integrity is to ensure data correctness for analytics. Hence, for big data residing across multiple resources, adequate mechanisms must be provisioned to the users to verify data integrity [18].

4. *Data quality*: The emergence of big data has resulted in its generation from various sources, some of which may not always be well-known or verifiable. Hence, poor data quality is yielded due to the data from one source not being consistent with data captured from another [19].

5. *File format transformation*: Effective analytics requires data to be in a unified storage format optimized for computation and I/O operations. Since big data originates from various sources, multiple storage formats are encountered by the data engineering pipelines during ingestion. Hence, adequate transformation mechanisms are required to transform incoming data into a unified storage format which supports analytics operations on the cloud service environment [14].

6. *Data format heterogeneity*: The big data characteristic of *variety* leads to heterogeneous big data, i.e., interconnected data from various sources with different incompatible types and inconsistent representations. This heterogeneous data needs to be consolidated into a consistent structured format. [20].

2.3 Cloud solutions for big data

Data engineering is a highly labor-intensive task for computational and storage resources. This process often stretches existing infrastructure to its limits. Large volumes of data require efficient methods for storage, aggregation and filtering, transformation, and retrieval. Cloud computing services offered by third-party organizations have revolutionized large data management for telecom operators. These services have significantly reduced resource costs and maintenance as compared to on-premise systems.

Cloud computing is defined as "a model for allowing ubiquitous, convenient, and on-demand network access to several configured computing resources (e.g., networks, server, storage, application, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [21].

Cloud service models occur in three categories, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), which are defined as follows [14]:

- *IaaS* refers to the cloud-based hardware equipment provisioned by cloud service providers upon demand.
- *PaaS* are resources operating on a cloud to provide platform computing for end-users.
- *SaaS* refers to applications running on remote cloud infrastructures offered by cloud service providers and accessible through the internet.

Cloud service providers offer various deployment models designed to address various enterprise requirements. Some common deployment models offered by cloud service providers are [13, 22]:

- *Private Cloud* is deployed on a private network, managed by the organization itself. It is suitable in scenarios where maximum network bandwidth is a requirement with no compromise on legal issues and security measures.
- *Public Cloud* is operated by third-party service providers. Dynamic resource provisioning over the internet on-demand via web services allow public clouds to offer high efficiency and resource availability at low costs.
- *Hybrid Cloud* deployment combines both public and private cloud. For this reason, it is favorable for scenarios where the private cloud's limited resources need to be replenished by a public cloud.

2.3.1 Big data as a service (BDaaS)

Cloud computing services offer scalable data storage, parallel processing frameworks, virtualized resources, security and data service integration. These services not only minimize the automation and computerization costs for enterprises but also eliminate infrastructural limitations. Moreover, efficient user access and role management are also provided along with the necessary services. For organizations undergoing digital transformation, it is important to understand the characteristics and challenges posed by big data as discussed earlier. Until the previous decade, the installation and maintenance costs of big data infrastructure had been prohibitive for mid-level organizations. Now, cloud service providers like Google, Microsoft and Amazon are revolutionizing big data solutions in the cloud. Consequently, Big Data as a Service (BDaaS) [23] is gradually becoming a reality. Mature cloud regulation procedures have enabled the establishment of big data infrastructure with minimal installation, maintenance, and integration requirements. The upcoming sections 2.3.1.1, 2.3.1.2, 2.3.1.3 and 2.3.1.4 highlight how the BDaaS model addresses the components of a data engineering pipeline:

2.3.1.1 Ingestion

For data ingestion, the BDaaS model provides the option of executing the enterprise's in-house ETL scripts on compute services. Besides this, it also supports commercial ETL tools and deploys them on infrastructure

provisioned by cloud service providers such as Google, Microsoft, and Amazon. Furthermore, BDaaS provides unified, high-throughput, low-latency open-source platforms for handling real-time data feeds, such as Kafka [24] or Amazon Kinesis [25] running on multiple compute instances. The administration of the ingestion workflow can be administered by cloud service providers or the organization itself based upon the requirements.

2.3.1.2 Storage

BDaaS offers storage mechanisms in the form of inexpensive and scalable object storage services capable of storing unlimited amounts of structured and unstructured data securely. These services are ideal for acting as storage end-points for *cold data*, i.e., archived data that is not accessed frequently. Some examples include Amazon Web Services (AWS) Simple Storage Service (S3) [26], OpenStack Swift [27], Windows Azure Binary Large Object (Blob) storage [28] and Google Cloud Storage (GCS) [29]. Additional distributed storage services include the Hadoop Distributed File System (HDFS), a fault-tolerant distributed file system that utilizes the local file systems of multiple compute nodes in a cluster to store big data [14, 30]. For storing frequently accessed data, i.e., *hot data*, NoSQL databases such as MongoDB [31] and Cassandra [32] can be leveraged for querying unstructured data. Similarly, relational databases such as MySQL [33] and PostgreSQL [34] can be utilized for querying structured data.

2.3.1.3 Processing and delivery

For data processing, BDaaS provides a secure, re-sizeable compute service provisioned on the cloud, which enables the deployment of data services to connect to the object stores mentioned earlier. Some examples include AWS Elastic Compute Cloud (EC2) [35] and Google Compute Engine (GCE) [36]. In addition, *MapReduce*, a simplified programming model designed for processing *big data*, is also offered [37]. For efficient in-memory processing, the model also provides *Spark*, a unified engine for big data processing built on top of the MapReduce framework which supports streaming (discussed in Section 2.4), Structured Query Language (SQL), machine learning and graph processing [38]. These services are provisioned on scalable clusters of compute instances, which subsequently execute the parallel programming framework upon data hosted on storage services, such as S3. An example of this is the Amazon Elastic MapReduce (EMR) service. The processed data can subsequently be delivered to the relevant analytics systems from the S3 storage services.

2.3.1.4 Analytics

BDaaS offers the incorporation of in-house BI tools managed by the organization. It also facilitates the outsourcing of analytics and visualization processes to cloud service providers. With the outsourced services, analytics workflows can be carried out through web applications without the need for ML frameworks and data scientists.

2.3.1.5 BDaaS service models

BDaaS consists of three service models, namely Big Data Infrastructure as a Service (BDIaaS), Big Data Platform as a Service (BDPaaS) and Big Data Software as a Service (BDSaaS). Figure 2.5 illustrates a high-level description of each of these service models concerning the components of the data engineering pipeline:

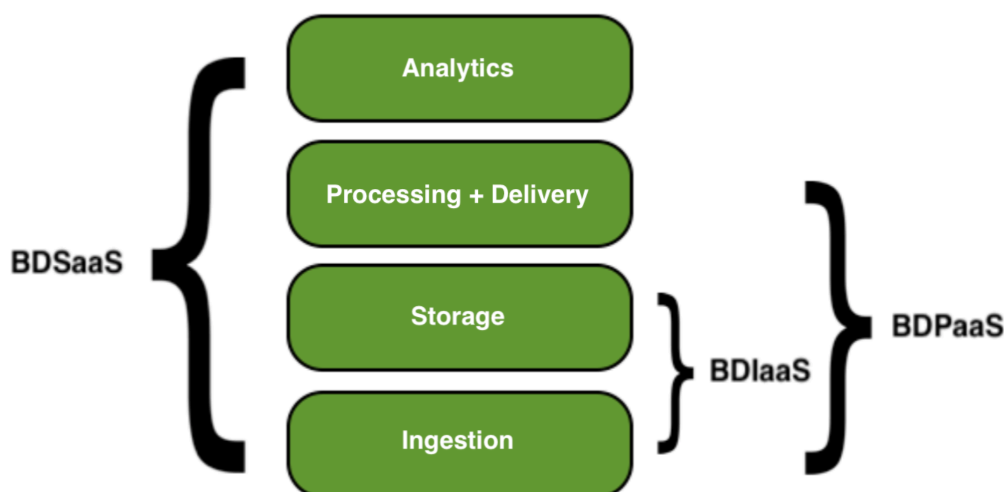

Figure 2.5: High-level illustration of the BDaaS service models.

The upcoming sections 2.3.1.5, 2.3.1.5, and 2.3.1.5 discuss the details of each service model. In addition, a high level comparison is subsequently provided in Section 2.3.1.5.

Big data infrastructure as a service (BDIaaS) The BDIaaS service model is suitable for an organization whose requirements are similar to purchasing a car engine and constructing the frame around it. BDIaaS can leverage the IaaS cloud service model, comprising storage and compute services,

to persist and process big data [23]. The BDIaaS model mainly addresses the ingestion and storage components of the data engineering pipeline.

Big data platform as a service (BDPaaS) The BDPaaS service model is suited to organizations that require a big data platform administered by cloud service providers, allowing users to access and analyze massive data sets [23]. The analytics applications are customized by the organization to operate on the datasets. The BDPaaS model addresses all four components of the data engineering pipeline, i.e., ingestion, storage, and processing and delivery for analytics. The BDPaaS service model can be deployed on all major cloud service providers such as Google (Google App Engine (GAE)) [39], Amazon (AWS) [40] and Microsoft (Azure HDInsight) [41].

Big data software as a service (BDSaaS) The BDSaaS service model is suited to organizations that require an end-to-end solution for exploiting structured and unstructured big data to obtain intelligent real-time results. It is a multi-tenant web-hosted service that enables users to perform self-service provisioning, analysis, and collaboration [23]. The BDSaaS model handles all four components of the data engineering pipeline as well as the analytics and ML operations through online services that encapsulate the underlying platforms and infrastructure. Some examples of BDSaaS service model providers are Panoply [42], Etleap [43], Inzata [44] and Looker [45] etc.

Comparison of BDaaS service models BDaaS service model selection has proven to be a challenge for many organizations inclusive of those in the telecom business. While the variety of services provided on the cloud is more than sufficient, many factors such as the cost, customization flexibility and technical expertise required for the solution need to be evaluated before short-listing a BDaaS cloud service model. Table 2.1 provides a high-level comparison of the various BDaaS service models:

Requirements	**BDIaaS**	**BDPaaS**	**BDSaaS**
Technical expertise	High	Moderate	Low
Customization support	High	Moderate	Low
Support plan cost	Low	Moderate	High

Table 2.1: High-level comparsion of BDaaS cloud service models [23].

2.4 Spark: A distributed processing framework

Apache Spark is a unified analytics engine for large-scale data processing [38]. It provides multiple application programming interfaces (APIs) designed for various purposes such as processing of unstructured and structured data, data streaming, ML and graph processing. This framework supports many programming languages such as Python, Java, Scala, and R. Furthermore, it is compatible with environments ranging from a single desktop to a cluster of thousands of servers. Due to this functionality, it is the ideal system to start on a low level and to gradually scale-up for big data processing. Figure 2.6 illustrates the libraries and components within Spark [46]:

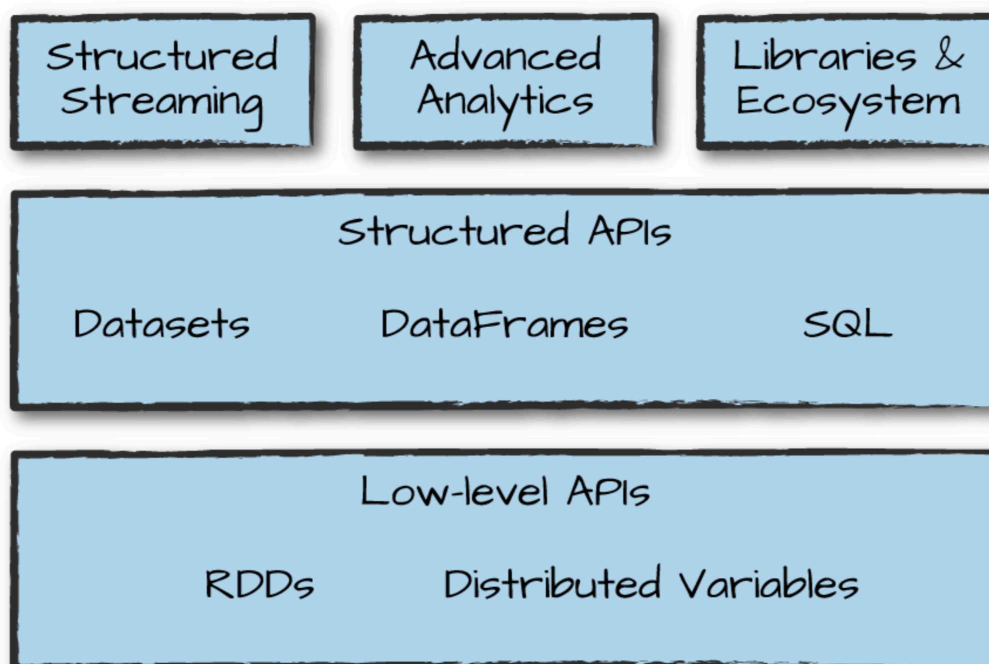

Figure 2.6: An illustration of Spark's components and libraries [46].

As depicted by the figure, Spark's low-level or core API provides two abstractions, i.e., the *resilient distributed database (RDD)* and *distributed variables*. *RDD* is a collection of elements that can be partitioned on a single machine or across a cluster of machines. *Distributed variables* are accessible to various tasks operating on RDD partitions, which reside on multi-

ple machines. Spark's structured API comprises three abstractions, namely *SparkSQL*, *DataFrames*, and *Datasets*. In comparison to the core API, *SparkSQL* requires more information about the structure of the data and the list of operations to be performed on it. Therefore, it performs additional optimizations during execution. The structured datasets that SparkSQL operates on are *Datasets* and *DataFrames*. A *Dataset* is a distributed collection of data similar to an RDD but is strongly-typed through a pre-defined structure or *schema*. This enables query optimization for SparkSQL. A *DataFrame* is a Dataset organized into named columns. DataFrames can be initialized from structured data files, SQL databases, and existing RDDs [38].

2.5 Overview of XML structure and parsing

This section explains the structure and content of the Extensible Markup Language (XML) documents used in the telecommunications industry along with their representation of data. Next, it presents parsing mechanisms for XML files. Finally, it highlights the underlying structure of raw network management data.

XML is a markup language that embodies a rule set used to encode documents. Hence, the document is transformed into a human-readable and machine-readable format. It is designed to provide flexible information identification. It plays a vital role in information exchange in the telecom industry as it is platform-independent and extensible. A typical XML document is built upon a list of *elements* as illustrated in Figure 2.7. Each of these elements comprises three components, i.e., a *start* tag, some *content* and an *end* tag. Both the start and end tags occur in pairs and are enclosed in <and >. Content refers to any simple text enclosed within the start and end tags. An element may include one or more *attributes* in its start tag [47].

```
<Employee>
    <Name>
        <First>Lassi</First>
        <Last>Lehto</Last>
    </Name>
    <Email>Lassi.Lehto@fgi.fi</Email>
    <Organization>
        <Name>
            Finnish Geodetic Institute
        </Name>
        <Address>
            PO Box 15,
            FIN-02431 Masala
        </Address>
        <Country CountryCode="358">Finland</Country>
    </Organization>
</Employee>
```

Figure 2.7: Example of an XML file [48].

Numerous XML parsing technologies exist, but the scope of this thesis focuses on the Document Object Model (DOM). DOM is defined as a platform and language independent interface which represents XML documents as object-oriented models. These models are accessible to applications as iterable trees, as illustrated in Figure 2.8. For this reason, the applications can dynamically alter the structure and content of these trees [47].

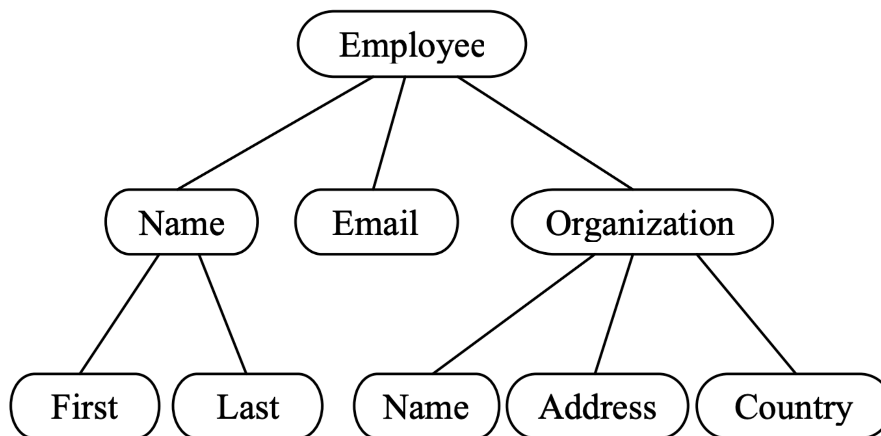

Figure 2.8: The DOM tree generated from the XML file shown in Figure 2.7. [48].

2.5.1 Raw network data

The 3rd Generation Partnership Project (3GPP) [49] is a standards organization that develops protocols for mobile telephony. Vendor-specific XML file format definitions for different types of network management data such as performance management (PM) and configuration management (CM) are standardized and maintained by 3GPP. Alarm types and frequencies at various time intervals for various elements of the network are obtained from PM data to indicate the health and stability of the network. Similarly, hardware configuration parameters acquired from CM data represent many network characteristics such as power or energy utilization. Figure 2.9 provides an overview of the PM data definition for a certain vendor X:

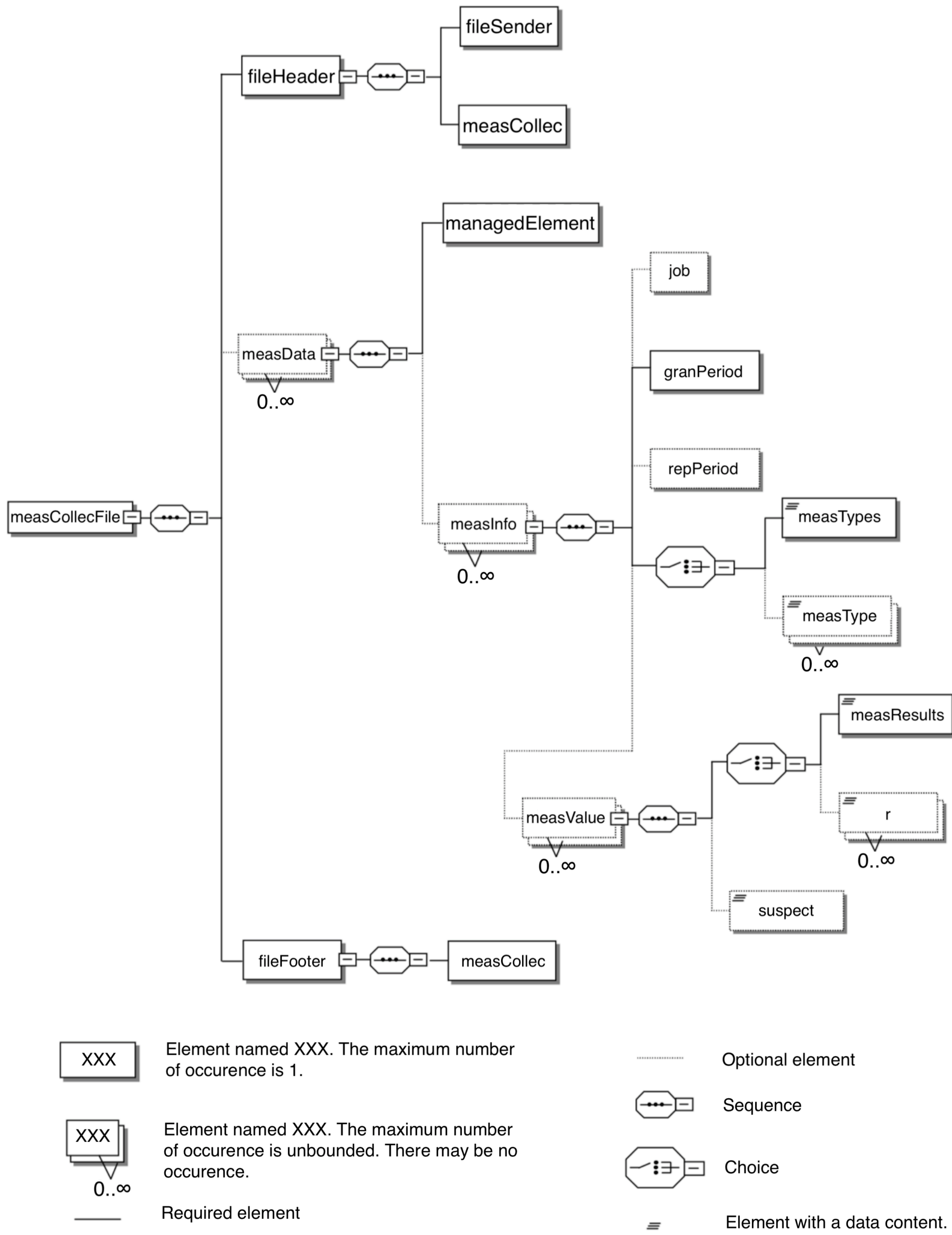

Figure 2.9: An illustration of the PM data definition for vendor X [49]

As illustrated in the figure above, the DOM tree generated for the PM data comprises the *fileHeader* and *fileFooter* elements. Both elements consist of vendor-specific information in addition to the start and end timestamps

for the data capturing event. Furthermore the DOM tree might consist of one or more *measData* elements. Each *measData* element has a *managedElement* child element, which holds information about the base station and its underlying cells. It also has a collection of *measInfo* elements for grouping information about the captured PM variables. All variable identifiers and values reside within the *measTypes* and *measResults* elements, respectively [49].

Similarly, Figure 2.10 illustrates the high-level element definition of the CM data for vendor X:

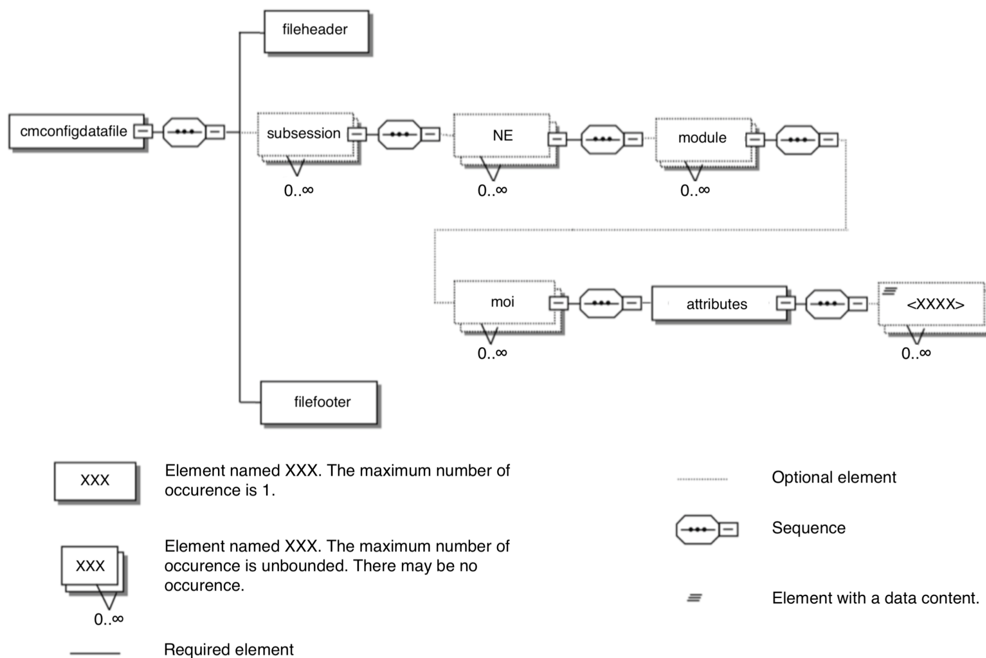

Figure 2.10: An illustration of the CM data definition for vendor X [49]

Before discussing the DOM tree structure for CM, it is necessary to introduce two terms, which are the Network Element (NE) and the Managed Object (MO). A NE is defined as a logical entity for grouping multiple physical entities in a telecom network. A MO refers to the physical entity which serves as a unit of the NE.

As shown in Figure 2.10, the DOM tree for the CM data comprises the *fileheader* and *filefooter* elements. Both elements consist of vendor-specific information in addition to the start and end timestamps for the data capturing event. Furthermore the DOM tree might consist of a collection of *subsession* elements. Each *subsession* element has a collection of *NE* ele-

ments. Each *NE* elements holds information about the NE (base station) in addition to having a collection of *moi* elements. Each *moi* element holds information about the instance of an MO (cells) in addition to having an *attributes* element. The *attributes* element contains multiple elements with varying names, each of which comprises the captured CM variables [49].

The above illustrations point towards complexity and dynamism in the data format. The DOM tree structure varies between vendors. To describe a concrete example, the scope of the thesis is limited to study vendor *X's* data, which has the definitions illustrated earlier. The definitions highlight the inter-dependency of child elements on their respective parents; hence each XML document needs to be supplied as a single unit to the parsing application.

2.6 Elisa Automate's (EA) existing data engineering strategy

This section discusses EA's existing data engineering strategy and maps it to the underlying components of the data engineering pipeline. Component-level details are discussed in Sections 2.6.1, 2.6.2, 2.6.3 and 2.6.4. Consequently, the need for data engineering pipelines in the existing infrastructure is established.

Network management routines are conducted via periodic analysis of network management data comprising various network-based events. This data is collected in raw format from multiple vendor-specific cellular devices and Operations Support System (OSS) end-points. This data subsequently undergoes processing and is stored in multiple storage end-points designed for various use cases. Figure 2.11 illustrates the workflow of the existing data engineering strategy at EA:

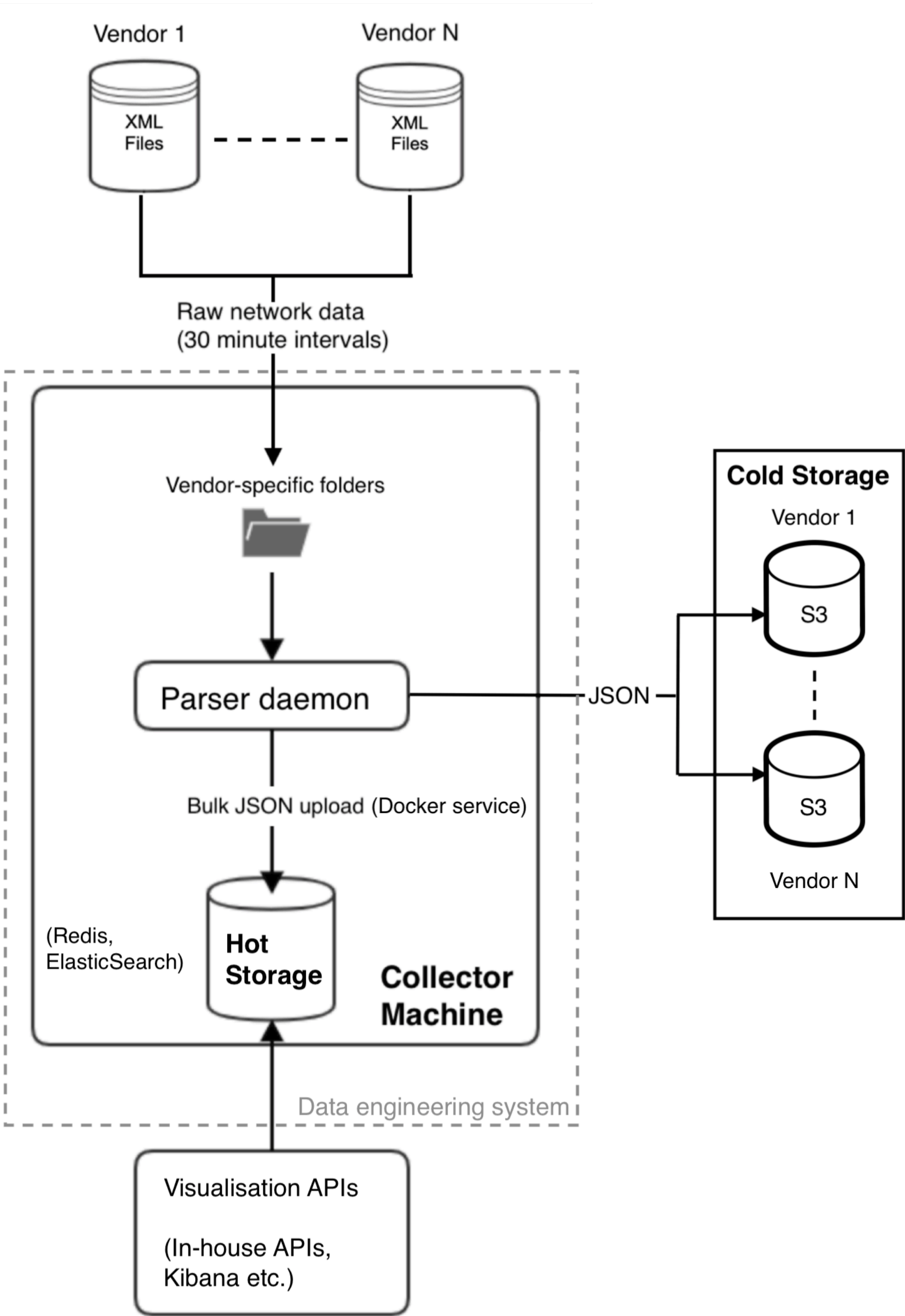

Figure 2.11: An illustration of the existing data engineering workflow at EA.

2.6.1 Ingestion

Raw network management data streams are received from multiple cellular devices spread across multiple regions through Simple Network Management Protocol (SNMP) traps. A collector machine schedules these SNMP traps and persists them in the resulting data streams as semi-structured XML files. Data collection cycle occurs at 30-minute intervals.

2.6.2 Storage

On completion of an interval of data ingestion, the collector machine stores the captured raw XML files into a vendor-specific directory structure on its disk. The retention time of this raw data is short due to the limited disk capacity of the collector machine. Based on the frequency of expected data access, hot and cold data storage end-points are dedicated to the data after it undergoes processing. For cold data, scalable third-party object stores such as Ceph [50] and MinIO [51] are allocated to store vendor-specific data separately. Similarly for hot data, the collector machine schedules containerized services on Docker [52] to export the processed JSON data to time-series data storage and searching services such as ElasticSearch [53] and Redis [54].

2.6.3 Processing

Upon receiving an entire 30-minute payload of raw data, the collector machine activates the parsing daemon. This daemon converts vendor-specific data from different OSS to JavaScript Object Notation (JSON) format based upon customized parsing functions designed for network management data specifications defined in vendor-specific catalogs. The parsing functions are executed by a multi-threaded Python application that utilizes the available processing cores on the collector machine. The JSON files yielded are subsequently transmitted to the hot and cold data stores mentioned in Section 2.6.2.

2.6.4 Delivery for analytics

As mentioned earlier in Section 2.6.2, the processed data is exported to ElasticSearch and Redis. In-house analytics and visualization APIs coupled with open-source APIs such as Kibana [55] subsequently retrieve the data from these storage end-points.

Chapter 3

Problem definition

This chapter presents the problems encountered in the processing of telecom network data. Infrastructural limitations of the existing data engineering system are identified, motivating the necessity to have a parallelized data parsing solution capable of handling massive datasets. The chapter concludes by highlighting the system requirements necessary to address all the identified limitations.

3.1 Motivation

Raw network data comprises PM and CM data. PM data carries several important performance counters such as the data transmission rate, downtime of each base station and alarm events occurring in the network. Similarly, CM data contains event-based hardware configuration parameters for devices installed on the base stations. A 24-hour compressed PM data feed can reach up to a 15 GBs of compressed files with an average compression ratio of approximately 10%. Although CM data is relatively insignificant in volume, the combined network data from PM and CM is categorized as big data. The existing data engineering system at EA utilizes a single centralized collector machine for capturing periodic intervals of network data. This collector machine also dedicates its computational resources for the network data processing routines. Due to the increasing bandwidth and volume of the raw data, the existing data engineering system has the following limitations:

- A single machine is dedicated to processing the data without any mechanism for parallelism. The machine requires 8 to 9 minutes to process 30-minute intervals of network data. Based on estimation, a cumulative processing time of approximately 7 hours is required for the entire

24-hour dataset. This leads to unnecessary delays in initiating the analytics workflows.

- Limited random access memory (RAM) and disk capacity of the single collector machine prevents parsing an entire 24-hour PM dataset in one pass.

- The JSON file format is not optimized for loading massive datasets with complex schemas into processing frameworks. As a result, conducting repetitive analytics and ML modeling tasks on a large dataset is time-consuming for data scientists due to prolonged read operations on the files.

3.2 Goal of the solution

For designing an efficient solution to the limitations described earlier, the consideration of resource provisioning and application optimization is crucial. The proposed solution must be equipped with scalable object storage to ingest raw network data within timeframes of 24 hours instead of short 30-minute intervals for cold storage. The retention policy for this data should also be prolonged to enable historical analysis. The solution must have adequate memory for loading multiple GBs of compressed raw data into the processing framework. Moreover, it should be equipped with sufficient computational resources to enable maximum parallelization of the parsing routines. Finally, the solution must store the processed data in a file format optimized for high-speed reading and loading into applications. As far as resource provisioning is concerned, the BDaaS cloud service models discussed in Section 2.3.1.5 addresses the majority of the computational and storage requirements. The goals of the proposed parsing solution are as follows:

- Support for dynamic allocation of computational and storage resources in a cloud environment.

- Support distributed parsing routines for unstructured datasets.

- Support the conversion of processed data into a file format optimized for compression and loading data into the applications for efficient querying, analytics, and visualization.

3.3 System requirements

Considering the solution goals, the requirements for the cloud-based solution are categorized under functional and non-functional requirements. The following sections introduce these two requirements categories respectively:

3.3.1 Functional requirements

Automation of data parsing within the data engineering pipeline has the following functional requirements:

3.3.1.1 F1: Data transmission

The solution must be able to read raw data from the input data store and write parsed data to the output data store.

3.3.1.2 F2: Resource allocation

The solution must have sufficient memory and disk capacity to create data structures to hold the results of parsing a 24-hour raw XML feed.

3.3.1.3 F3: Distributed parsing

The solution must be able to distribute the parsing workload across multiple compute nodes.

3.3.1.4 F4: Schema transformation

The solution must parse raw data into a unified structure similar to the JSON structure yielded by the existing solution at EA.

3.3.1.5 F5: File-format support

The solution must be able to yield the structured data as a file format optimized for high-speed reading and compression.

3.3.1.6 F6: Automation

The solution must initiate scheduled parsing jobs without human intervention.

3.3.2 Non-functional requirements

The data parsing solution must fulfill the following non-functional requirements to achieve the desired mode of operation:

3.3.2.1 NF1: Reliability

The solution must successfully parse the entire input dataset even if some of its resources encounter hardware or software failure.

3.3.2.2 NF2: Scalability

The solution must have the ability to parse large volumes of raw data with its dedicated infrastructure.

3.3.2.3 NF3: Speed

The solution must successfully parse 24-hour raw data feeds within a maximum time frame of 3 hours.

The proposed system will be evaluated through multiple experiments. The experimental evaluation will determine the degree to which the proposed solution addresses each of the non-functional requirements. But prior to this information, Chapter 4 introduces the design and workflow of the proposed parsing solution.

Chapter 4

Solution Design

This chapter introduces the proposed parsing solution as a component of the data engineering pipeline along with its internal design and workflow. Section 4.1 firstly presents the architecture for the data engineering pipeline employed by EA. Subsequently, Section 4.2 introduces the application-level design of the solution. The chapter concludes with Section 4.3 highlighting the workflow of the solution. In each section, design choices are explained in light of the relevant functional requirement.

4.1 System Infrastructure

For solution deployment, AWS has been selected as the cloud service provider for hosting the data engineering pipeline based upon feasibility studies regarding security that is beyond the scope of the thesis. The following diagram illustrates the development infrastructure of the data engineering pipeline and highlights the proposed parsing solution as a component within the pipeline:

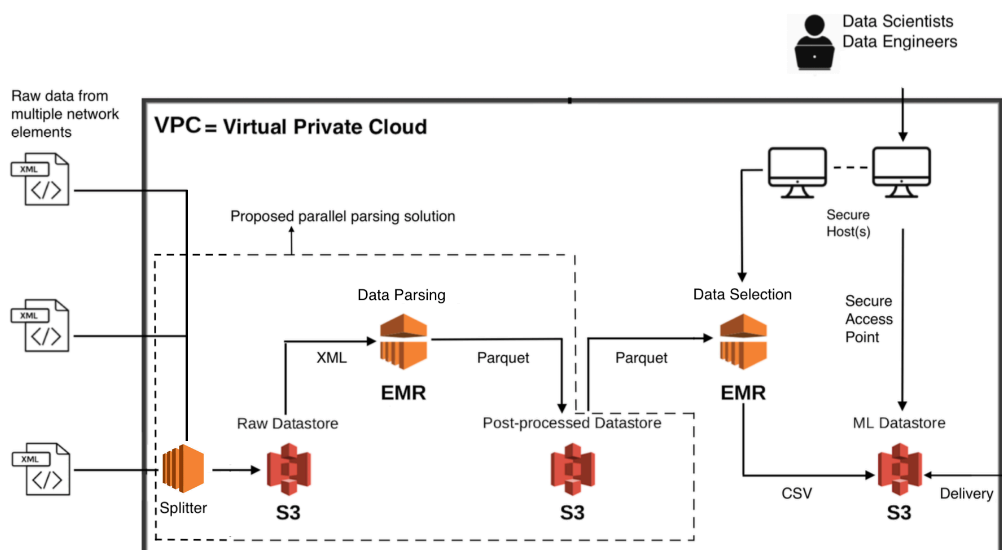

Figure 4.1: Overview of the developed data engineering pipeline. The proposed parsing component is enclosed within the dotted lines.

The infrastructure follows the hybrid cloud model comprising a Virtual Private Cloud (VPC) that is isolated from external internet access except for trusted sources. The VPC receives on-demand services from the AWS public cloud. The data pipeline architecture is based on the BDPaaS service model. Moreover, the solution utilizes microservices provisioned by AWS to assign resources with configurable storage and compute capacities. *Microservices* is defined as an approach to software development where software is built on minor independent services that communicate over well-defined APIs. The architecture models provided by AWS thus possess the flexibility of scaling in terms of computational and storage capacity and ensuring high availability in the process.

F2: Resource allocation We chose the Amazon S3 [56] service to host the data storage for the parsing component. It is an object storage service provided by AWS that offers scalability, data availability, security, and performance according to a usage-based cost model. This storage service provides data partitioning based on key features into buckets and supports both structured and unstructured file formats [56]. The service persists files within a virtual directory structure, which is useful for data categorization. As depicted by Figure 4.1, separate buckets have been assigned for the raw data and the parsed data. The underlying data in each bucket is categorized

by type, i.e., PM and CM data and by duration e.g. 24-hour data, weekly data or monthly data. For this thesis, only 24-hour data feeds are considered. The S3 service is solely designed for storage and has no dependency on the life time of other services and computational resources.

We employed Amazon EMR [57] to perform scalable computations in the parsing component. EMR is a cloud-native platform that utilizes open-source tools to enable big data processing. An EMR cluster comprises multiple Elastic Compute Cloud (EC2) [58] machines or instances allocated by Amazon as computational resources. The dynamic scalability of EC2 instances equips the EMR service with the elasticity to execute big data parsing and transformation jobs. EMR provisions clusters with both transient and persistent lifetimes. These clusters automatically scale according to the input data to undergo processing.

F1: Data transmission The EMR service consists of the Elastic MapReduce File System (EMRFS), a customized implementation of the Hadoop file system. It provides an EMR cluster with the ability to directly read and write between Amazon S3 buckets. Moreover, since it is based on the Hadoop file system, it also supports the transmission of data into parallel programming frameworks such as MapReduce [57].

A dedicated EC2 instance, *Splitter*, has been deployed within the infrastructure as a future iteration of the original solution workflow. This instance detects large XML files being sent to the EMR cluster and leverages the Simple API for XML (SAX) parser to split them into more manageable files while retaining their internal structure.

4.2 Distributed Spark parser

We chose to construct the proposed parsing solution on a Spark application, which offers multiple features and APIs as discussed earlier in Section 2.4.

F3: Distributed parsing Since PM and CM data is semi-structured with a non-uniform schema, the solution leverages the RDD API to read the raw data into the application and execute multiple parallel processes across a cluster of nodes. An RDD is an immutable object collection partitioned across multiple machines. The concept of RDD *lineage* ensures that each RDD possesses sufficient information about its derivation to reconstruct itself in case of failure. RDDs are initialized from files residing in any Hadoop-supported file system which in this scenario is S3. The application ensures

the reusability of RDDs by offering RDD caching in shared memory across the cluster [59].

A simplified component model depicting cluster management by the parallel parsing application is shown in Figure 4.2. A cluster manager keeps track of permissions regarding task scheduling on worker nodes and the resource allocation [60].

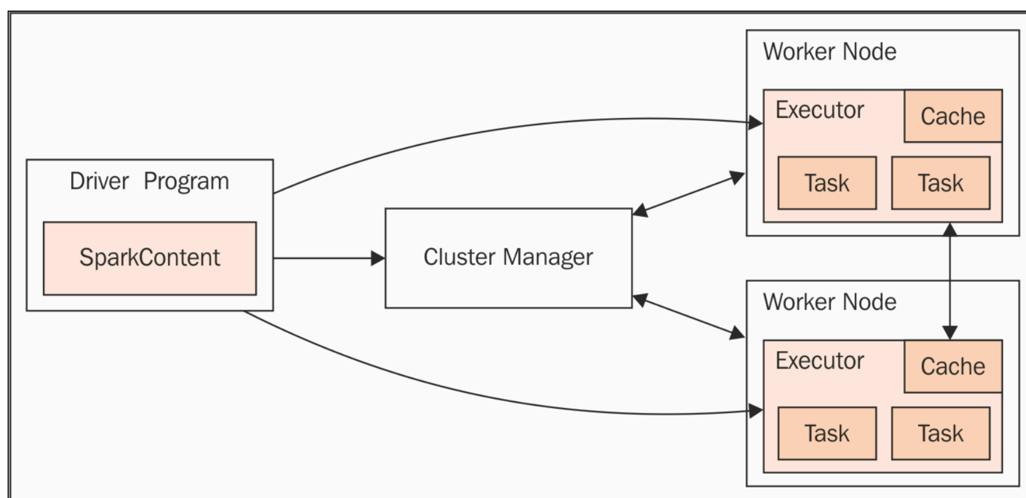

Figure 4.2: Cluster management in Spark based parsing solution [60].

RDD operations supported by Spark are categorized as *transformations* and *actions*. An *action* operation initiates a computation job and returns a value to the program or writes data to external storage. *Transformation* operations, on the other hand, are lazy operations that result in a new RDD. Lazy operations mean that the new RDDs are not computed until an action requiring them is initiated. In addition to these operations, Spark also supports the persistence of RDDs in the cache as mentioned earlier. Figure 4.3 describes all the RDD action and transformation operations [59].

	Operation		Signature
Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V,V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T,T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.,* HDFS

Figure 4.3: RDD operations in Spark. Seq[T] denotes a sequence of elements of type T [59].

F4: Schema transformation Spark also offers structured APIs such as DataFrames and Datasets for reading data into table-like structures optimized for querying records and transformations. But the limitation of these APIs is that they expect data to be in a uniform structured format. Since the RDD API supports transformation operations on unstructured data, structural constraints can be enforced on the RDD contents. In addition to these, RDD operations can also execute user-defined functions. Support for Python libraries on Spark enables the parsing routines of the existing solution at EA to be imported into the proposed parsing solution. Hence, the generated schema format is similar to the existing JSON format. Once data has been reformatted, the compatibility of RDDs with DataFrames and Datasets allows the data to be migrated to the structured APIs, which are optimized for searching and manipulating the data in place.

F5: File-format support EMR clusters are pre-installed with Apache Spark, Hadoop HDFS and MapReduce. This enables them to convert input data, loaded within the Spark application context, to the Parquet format. "The Parquet format is an efficient columnar data representation designed to support efficient compression and encoding schemes. It allows compression schemes to be specified on a per-column level and is future-proofed to allow adding more encodings as they are invented and implemented" [61]. This format is optimized for loading massive datasets into data structures.

4.3 Solution workflow

F6: Automation To initiate the workflow without human intervention, we utilized AWS Step Functions [62]. AWS Step functions have the ability to initiate a Spark application based on a schedule. They provide the service of automatically triggering jobs on the EMR's master EC2 instance. Moreover, these functions also track each step and initiate retries in case of errors, ensuring resilience and system availability for existing and future datasets [62]. An end-to-end depiction of the workflow is provided by Figure 4.4.

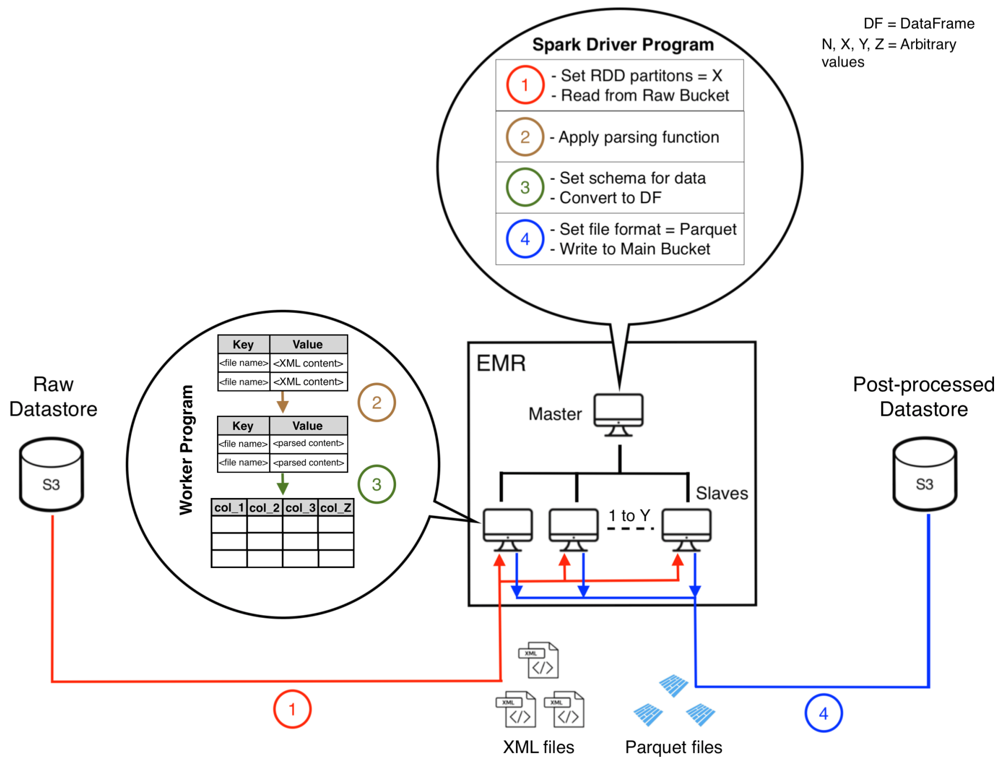

Figure 4.4: A high-level illustration of the solution workflow from raw XML data to processed data in the Parquet format

The data engineering pipeline persists raw PM and CM data as a collection of XML files stored in the *Raw Datastore* highlighted in Figure 4.1. This data is categorized in the bucket by its type and date (24 hours). Once a complete 24-hour dataset is imported to the S3 bucket, the solution workflow highlighted in Figure 4.4 is initiated by the AWS Step function by activating the Spark driver program on the master node of the *Data Parsing* EMR

cluster. The driver program subsequently activates four sequential steps.

Step 1 initiates the reading of raw XML files from the *Raw Datastore* into an RDD via the Spark application. The RDD API supports RDD formation via two methods. The first method assigns unique keys to each line read from an I/O stream directed towards one or more XML files to create an RDD. The second method involves assigning the XML file name as the key and its underlying content as the value of an RDD. This particular RDD is known as a PairRDD. Due to the complex structure and intra-element dependencies of the DOM tree generated from the raw XML files, the PairRDD structure is employed in this scenario. The reason is to ensure that the entire content of each XML file is available to the specific node assigned to parse it. For this reason, there is no data dependency between the nodes of the cluster and parallelism can be exploited. After the content has been imported into a PairRDD, it is split into equally sized partitions comprising multiple XML files. The quantity of XML files per partition is dependent on the relationship between the partition size and the individual XML file sizes. The driver program subsequently requests the cluster manager to randomly assign these partitions to the nodes of the EMR cluster.

As illustrated earlier in Figure 4.2, each node comprises one or more executors responsible for carrying out the remaining steps of the workflow as shown in Figure 4.4. An executor is assigned pre-defined computation and storage resources of the node. These include the amount of RAM and the number of Central Processing Unit (CPU) cores. Based on these capacities, the executor spawns an adequate number of tasks to process the assigned partitions in parallel. As a result, multiple executors running across the nodes of the EMR cluster parse the partitioned data in a distributed fashion. *Step 2* leverages the RDD's map operation to apply the parsing function to each of the XML files. The functions are mapped to the *values* or content of each XML file based upon their *key* or qualified name of the file.

The parsing and organization of the data via RDD transformations enables the DataFrame API to convert the PairRDD into a structured DataFrame. *Step 3* enables this for firstly converting the structured content for each key-value pair of the PairRDD into Row format [63] by applying a map operation on the structured *value* content. The Row format [63] enforces Spark's primitive data types upon the structured content to give it the shape of a relational database. DataFrames are constructed based on these formats upon a simple invocation of the *toDF()* method.

Step 4 finally saves the resulting DataFrame in the Parquet format to the *Post-processed Datastore*. Each node concurrently saves its assigned partition independently of the other nodes. Once the writing phase is complete, the context is switched from the worker programs on the nodes to the driver

program on the master node. The driver program subsequently terminates successfully, signaling to the Step function that the workflow for a single batch of the network data has reached completion.

In summary, the proposed solution utilizes Apache Hadoop's resource manager or Yet Another Resource Negotiator (YARN) on the EMR cluster to access the S3 buckets and exchange data between them. It subsequently converts the data into a partitioned and parallelizable dataset via Spark's RDD API. The partitioned dataset is distributed among the EMR's nodes, which ensures load-balancing and prevents per-node disk and memory overload errors. Furthermore, it also provides fault-tolerance and reliability as each partition can be re-constructed on each node in case of task-level failures. The executors operating on each node support user-defined parsing functions. PySpark provides access to all the necessary Python libraries for these functions. This ensures the reusability of EA's existing parsing techniques. Secondly, the conversion of the RDD to a DataFrame provides a data representation for the parsed content similar to a relational database. Lastly, multiple tasks running under each executor utilize the CPU cores and RAM of each node to boost the performance of the solution for massive datasets.

Chapter 5

Experimental setup

This chapter introduces the experimental setup for the empirical evaluation of the proposed solution. Section 5.1 firstly highlights the design details and system specifications for the experimental framework. Next, Section 5.2 introduces the sample datasets considered for the experiments. The chapter concludes with Section 5.3 which discusses the rationale behind the Spark application configurations selected for the experiments.

5.1 System specifications

As illustrated earlier in Figure 4.1, the *Data Parsing* EMR cluster functions as the core platform for conducting the experiments. For each experiment, the raw data sample is acquired from the *Raw Datastore*. Similarly, after being parsed, the processed data sample is saved in the *Post-processed Datastore*. The EMR cluster is built upon multiple Amazon EC2 instances. Amazon provides a diverse collection of EC2 instance types, each designed for varying use cases. These instance types are grouped into families consisting of different combinations of CPU, network capacity, RAM, and disk storage. This feature provides added flexibility for addressing the resource requirements of the applications. Table 5.1 provides the specifications for the *Data Parsing* EMR cluster:

Data Parsing EMR specifications			
Node quantity	*EMR version*	*Cluster type*	*Spark version*
4	5.22.0	Persistent	2.4.0

Table 5.1: List of specifications for the *Data Parsing* EMR cluster.

Since EMR version 5.x, support for Python 3.6 has also been provided. During the project implementation, version 5.22.0 was considered as it fulfilled all the Python library requirements of EA's existing parsing modules. Furthermore, Spark 2.4.0 has been employed in the selected EMR version [64]. Lastly, Amazon offers the EMR service in two modes, i.e., *persistent* and *transient*. *Persistent* mode keeps the cluster active at all times whereas *transient* mode terminates the cluster upon completion of a job. In this scenario, the persistent mode is favorable firstly since reliability, one of the key performance requirements of the solution, is compromised during the spawning of the cluster which takes around 10 to 20 minutes. Secondly, the frequent cluster reformations incur additional cost. Table 5.2 provides the EC2 instance specifications deployed in the EMR cluster:

EC2 instance specifications				
Instance name	*Instance type*	*vCPU*	*RAM (GB)*	*Storage (GB)*
m5.2xlarge	master	8	32	128
m5.2xlarge	worker	8	32	1000
m5.2xlarge	worker	8	32	1000
m5.2xlarge	worker	8	32	1000

Table 5.2: EC2 instance specifications for the *Data Parsing* EMR cluster [58].

Since this solution has been designed as a research project, the EMR operates in a non-production environment. Based on this, the instance family selected for the experiments is the M5 instance family. This family is the latest generation of general-purpose instances. It provides a balanced set of computational, memory, and storage resources. As a result, it is an all-round choice for conducting processing jobs [58].

5.2 Sample datasets

As discussed earlier in Section 3.2, the proposed parsing solution has been designed to process a 24-hour dataset. Since the solution workflow is identical for both PM and CM data (Figure 4.4), the sample datasets have been chosen for PM data only as it is larger in volume. To study the behavior of the solution concerning reliability, scalability, speed and ultimately the best configurations for Spark, a total of five dataset samples were chosen for the experiments.

Data selection To identify the problems introduced by real-world data capturing, network data archives stored in EA's existing data engineering solution have been used for extracting the samples used in the experiments. These archives represent live data captured from telecom equipment belonging to a single vendor. Archives spanning over 24-hours are stored within a folder and consists of multiple sub-folders for categorizing the network equipment by region. Within each of the region-specific folders lie compressed XML files representing the PM data. Since one region may have more network equipment than another, the number of PM KPIs or counters represented by each XML file may vary. Hence, selecting samples from different regions ensures diversity but at the price of possible bias introduced by the variation between the compressed XML files. The benefit of multi-region selection is the exposure of all edge-cases of the data processing workflow for the proposed solution.

Each file is a subset for a 24-hour PM data feed. For ensuring consistency in the samples, each of them has been generated from the same 24-hour dataset. The largest sample is the entire 24-hour data feed itself. The overall size of each sample is approximately twice the size of the previous one as illustrated in Table 5.3.

Sample Datasets					
Sample #	*Size (GB)*	*Records (millions)*	*XML files (thousands)*	*Min Size (KB)*	*Max Size (KB)*
Dataset 1	0.36	12.04	8.35	12.30	77.80
Dataset 2	0.67	24.46	16.36	16.40	81.90
Dataset 3	1.40	49.07	32.26	8.20	81.90
Dataset 4	3.00	100.48	49.17	12.30	4608
Dataset 5	5.60	205	93.04	8.20	9932.80

Table 5.3: List of the PM dataset samples selected for the experiments.

In Table 5.3, the *Records* field indicates the combined number of network data counter records in the entire dataset. The table also lists the total number of XML files (*XML files*) for each dataset along with the minimum (*Min Size*) and maximum (*Max Size*) sizes of the compressed XML files in each dataset.

5.3 Spark configurations

Static vs. dynamic configuration Resource allocation is one of the most crucial aspects of a Spark application when it comes to reliability and resource utilization. As discussed earlier in Section 4.3, an executor is the main work unit responsible for carrying out tasks or jobs within a Spark application. By default, Spark applications initiated on the EMR cluster are allocated executors dynamically by the Spark driver program after collaboration with a cluster resource manager such as YARN. This feature allows flexibility in terms of the number of executors allocated for an application. But it remains insufficient for determining critical configurations such as the **number of executors**, **memory allocation per executor** and the **number of partitions** for the input data. Due to this, it is recommended to configure Spark applications to employ static resource allocation based on custom parameters supplied by the user.

Parameter definitions Since Spark 2.0, the `SparkSession` object is responsible for managing the connection to a Spark application. It contains a `SparkConf` object, which accepts a set of parameters to configure the application. These parameters are represented as key/value pairs. Table 5.5 highlights these parameters, which determine the performance and reliability of an application.

Spark Configurations				
Name	*Description*	*Value*	*Restrictions*	*Guidelines*
spark.driver .memory	The size of the Spark driver	1024 MB	In YARN cluster mode, no larger than YARN container including overhead.	A higher setting may be required if collecting large RDDs to the driver or performing many local computations.
spark.executor .memory	The size of the Spark executor	1024 MB	One executor plus overhead cannot be larger than one request (a single YARN container).	Larger Spark workers may prevent out-of-memory errors for jobs with unbalanced workloads but they are inefficient.
spark.driver. cores	Number of virtual cores allocated to the driver	1	The number of cores available in the YARN container.	A good number is 5. It should be scaled up according to the resources.
spark.executor. cores	Number of virtual cores allocated to each executor	1	The number of cores available in the YARN container.	A good number is 5. It should be scaled up according to the resources.
spark.executor. instances	Number of executors	2	The initial number of executors on the cluster.	It should be scaled up according to the resources.
spark.default. parallelism	Parallelism within the RDD	2	The number of partitions in RDDs.	At least 2 times the total number of cores available on the cluster.

Table 5.5: A list of Spark configuration parameters critical to the tuning of an application [65].

For achieving the goal of maximum resource utilization, we configured the EMR cluster to execute Spark applications with pre-defined settings computed through formulas endorsed by Spark and AWS users [65, 66]. Equation sets 5.1, 5.2, and 5.3 depict variables and formulas employed for the computation of the Spark application configuration parameters:

$$\begin{gathered} N_e = \text{executors per node} \\ E_{cores} = \text{cores per executor (supplied by user)} \\ N_{vc} = \text{virtual cores per node} \\ TN_{mem} = \text{total memory per node} \\ TE_{mem} = \text{total executor memory} \\ N_{mem} = \text{YARN allocated memory per node} \\ E_{mem} = \text{available memory per executor} \\ E_{memoverhead} = \text{overhead memory per executor} \\ N_{nodes} = \text{nodes in the cluster} \\ N_{e_{instances}} = \text{executors in the cluster} \\ P = \text{partitions for the data} \\ X = \text{parallelism factor (supplied by user)} \end{gathered} \tag{5.1}$$

$$\begin{gathered} N_e = \left\lfloor \frac{N_{vc} - 1}{E_{cores}} \right\rfloor \\ TN_{mem} = \frac{N_{mem}}{1024} \\ TE_{mem} = \left\lfloor \frac{TN_{mem}}{N_e} \right\rfloor \\ E_{mem} = \lfloor T_{e_{mem}} * 0.90 \rfloor \\ E_{memoverhead} = \lceil T_{e_{mem}} * 0.10 \rceil \\ N_{e_{instances}} = N_e * N_{nodes} - 1 \,\text{(one executor reserved for driver)} \\ P = N_{e_{instances}} * E_{cores} * X \end{gathered} \tag{5.2}$$

$$\begin{gathered} \texttt{spark.executors.cores} = E_{cores} \\ \texttt{spark.executor.memory} = E_{mem} \\ \texttt{spark.driver.memory} = E_{mem} \\ \texttt{spark.driver.cores} = E_{cores} \\ \texttt{spark.executor.instances} = N_{e_{instances}} \\ \texttt{spark.default.parallelism} = P \end{gathered} \tag{5.3}$$

As illustrated in Equations set 5.3, `spark.default.parallelism`, `spark.executor.memory` and `spark.executor.instances` are determined based on two user-supplied variables. These are the number of CPU cores per executor (E_{cores}) and the multiplier for the default parallelism (X).

Number of executors The formula for determining the number of executors per node (N_e) indicates that a single core from each node is reserved for the Hadoop daemons [66]. Moreover, a single executor is not able to span across multiple nodes as indicated by the floor function. Also, out of the total executors allocated over the entire cluster ($N_{e_{instances}}$), one is reserved for the Spark driver.

Executor memory The memory available on each node (TN_{mem}) is always less than the per-node memory specification as YARN allocates an adequate portion for the Operating System (OS) [65]. The available node memory is subsequently divided among the number of executors per node (N_e) to yield the total memory allocated per executor (TE_{mem}). From this allocation, 10% is reserved for overhead ($E_{memoverhead}$) and the remaining is dedicated for core computations (E_{mem}). Spark allocates the same (E_{mem}) for all the executors across the nodes.

Partitioning and parallelism Lastly, the number of input data partitions is dependent upon (N_e), (E_{cores}) and the parallelism factor (X). (E_{cores}) determines the number of jobs an executor executes in parallel. Ideally, the number of partitions must always be more than or equal to the total available CPU cores on the cluster. Larger and fewer partitions provide a good speed up but compromise on reliability and application stability in case of massive files in each partition overburdening the executor resources. Smaller and more abundant partitions provide stability but have an upper bound for the speed as there is frequent inter-node data transfer and I/O bottlenecks [65].

Configuration presets For the experiments, we defined three presets for the Spark settings. Each preset was designed based on three objectives. These are the maximum utilization of the CPU cores, maximum memory allocation per executor and a balance between the first two objectives. The first objective addresses computationally intensive Spark applications where data partitions can be managed by a large number of executors with minimum memory and compute resources. The second addresses memory-intensive applications where the data volume requires a few executors equipped with maximum memory and compute resources. The third and final objective is

suitable for applications of a balanced nature, i.e., dependent upon data volume and computation. Based upon the objectives, the first preset is designed to exploit the maximum amount of cores on a node and ensure maximum utilization of all cores within the EMR. The second preset is expected to target datasets yielding large data partitions that require significant memory to be loaded onto the Spark worker application. Due to this, the executors are equipped with multiple cores and sufficient memory to complete the job. The third preset is designed to find a balance between the first two presets in terms of data volume and computational utilization. Table 5.6 provides the Spark configuration presets employed for the experiments:

Spark Configuration Presets				
Preset #	E_{cores}	$N_{e_{instances}}$	$Parallelism\ factor$ (X)	*Objective*
Preset 1	1	21	{ 2, 4, 6, 8 }	CPU core utilization
Preset 2	3	6	{ 2, 4, 6, 8 }	Balanced
Preset 3	5	3	{ 2 ,4, 6, 8 }	Memory allocation

Table 5.6: List of the Spark configuration presets selected for the experiments.

The experimental phase involves a total of 60 trials with 20 for each Spark preset. Each preset has 4 groups of trials based on the varying parallelism factor for determining the number of data partitions, i.e, $X = \{\ 2,\ 4,\ 6,\ 8\ \}$. Subsequently, each group has a total of 5 trials for each of the datasets listed in Table 5.3. Chapter 6 highlights the results from the experiments designed for the empirical evaluation of the solution.

Chapter 6

Evaluation

This chapter presents the experimental results and evaluates the solution concerning its non-functional requirements, i.e., reliability, scalability, and speed. The goal of the evaluation study is to determine the optimal Spark configuration parameters required to make the application reliable but not unnecessarily expensive in terms of the total number of data partitions and the number of parallel executors. Section 6.1 firstly provides the reliability statistics obtained from the experiments conducted for the three Spark configuration presets. Next, Section 6.2 visually illustrates the relationship between the total execution time, dataset size, and the parallelism factor governing the total data partitions, thus highlighting the solution's scalability. Section 6.3 subsequently discusses the execution times of the experiments, highlighting the extent to which the solution addresses the speed requirement. Section 6.4 summarizes the overall efficiency of the solution with respect to its reliability, scalability and speed. Furthermore, it briefly explores minor data partitioning manipulations within the proposed solution to study the results on its non-functional requirements. The chapter concludes with Section 6.5, which highlights how the file size distribution affects the efficiency of the solution. Furthermore, it briefly covers some additional steps employed in the proposed solution. Results from the second iteration of the proposed solution are subsequently discussed to emphasize the effect of file size distribution on the solution.

6.1 Reliability

After the execution of the experimental trials, a comparative analysis was conducted for the three Spark configuration presets discussed in Section 5.3. In Spark, executors spawn multiple *tasks* for parsing each data partition. To

study the reliability of the system, the relationship between the task failure percentage and the parallelism factor X was studied. Figures 6.1, 6.2 and 6.3 illustrate these relationships for the three presets.

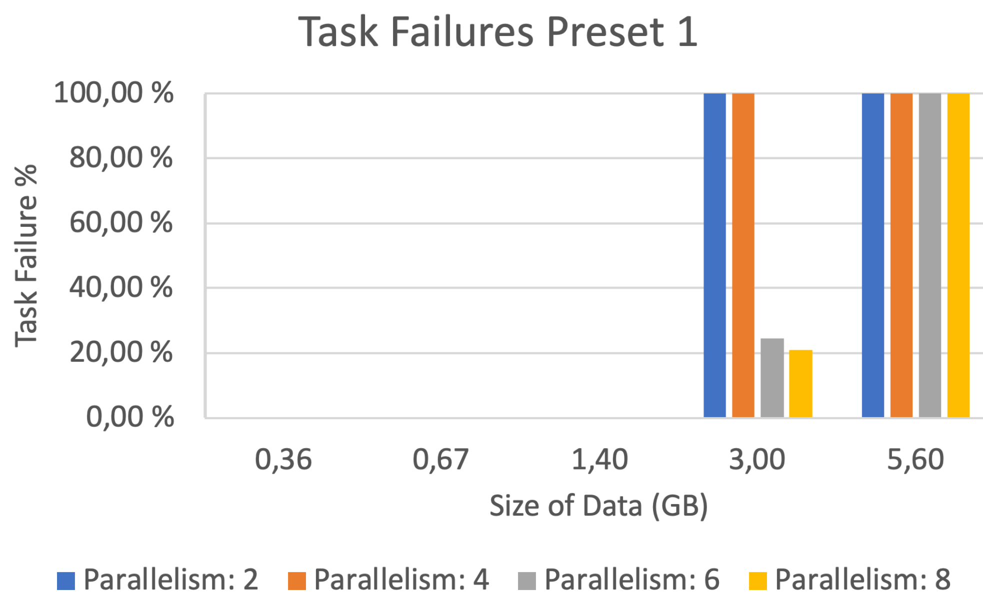

Figure 6.1: An illustration of the relationship between the task failure percentage and parallelism factor for Preset 1.

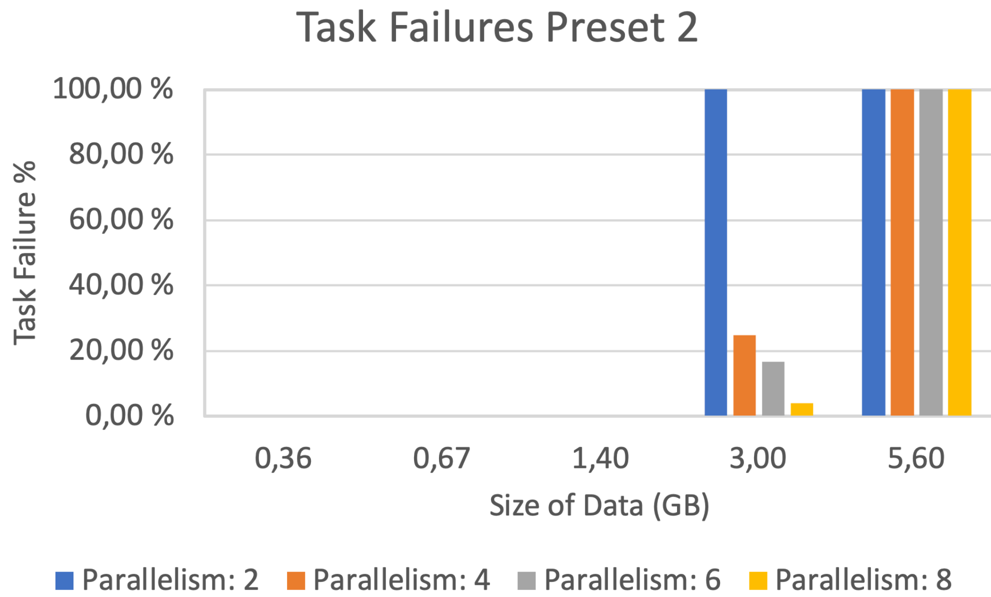

Figure 6.2: An illustration of the relationship between the task failure percentage and parallelism factor for Preset 2.

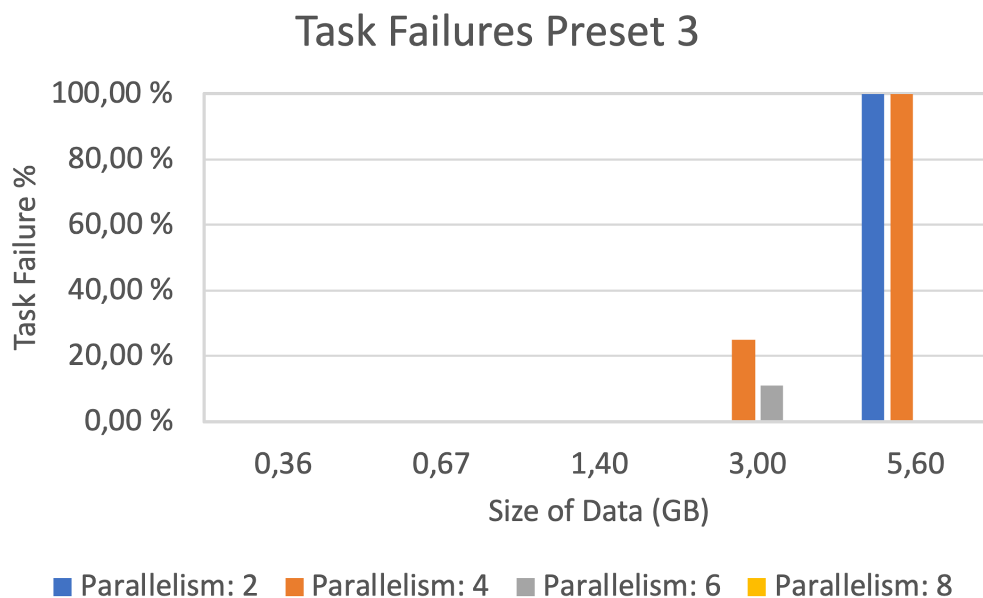

Figure 6.3: An illustration of the relationship between the task failure percentage and parallelism factor for Preset 3.

As illustrated in the figures above, the number of successful tasks increases with the parallelism factor, i.e., the multiplier for the number of data partitions. Since the EMR resource allocation for the experimental phase was static as highlighted by Table 5.2, the cluster had an upper bound for the size of the raw data. But the resources allocated for the experiments were sufficient to parse an entire 24-hour dataset as depicted by Figure 6.3. Tasks spawned by the executors mostly fail due to memory overflow errors. The reason behind insufficient memory is the size of XML files being handled by the tasks. Since the atomic data units being processed are the XML files themselves, the parallelism factor is used to govern how many XML files are sent to each task for parsing. Increasing the parallelism factor increases the total data partitions which consequently decreases the number of XML files per partition. Having more partitions reduces task failures. The downside of abundant partitions is execution time overhead due to network latencies in transferring data partitions across multiple nodes in the EMR. Therefore, a good parallelism factor should neither be too large nor cause executor failures. In comparison to the other two presets, preset 3 had the lowest number of executors configured with the highest memory and the number of cores.

Executors which receive partitions composed of relatively large XML files are more likely to face memory overflow errors. If all the executors encounter memory overflow errors, the whole Spark execution is terminated with 100% task failure. The results indicate the fact that datasets 4 and 5 have relatively larger XML files as compared to datasets 1, 2 and 3. As highlighted by Figure 6.3, successful parsing is demonstrated for all the datasets for a parallelism factor of 8. Hence, the proposed solution parses entire 24-hour network data reliably with preset 3.

6.2 Scalability

To study the solution's scalability, the relationship between the job execution time and the parallelism factor X was studied for each preset. Figures 6.4, 6.5, and 6.6 illustrates these relationships:

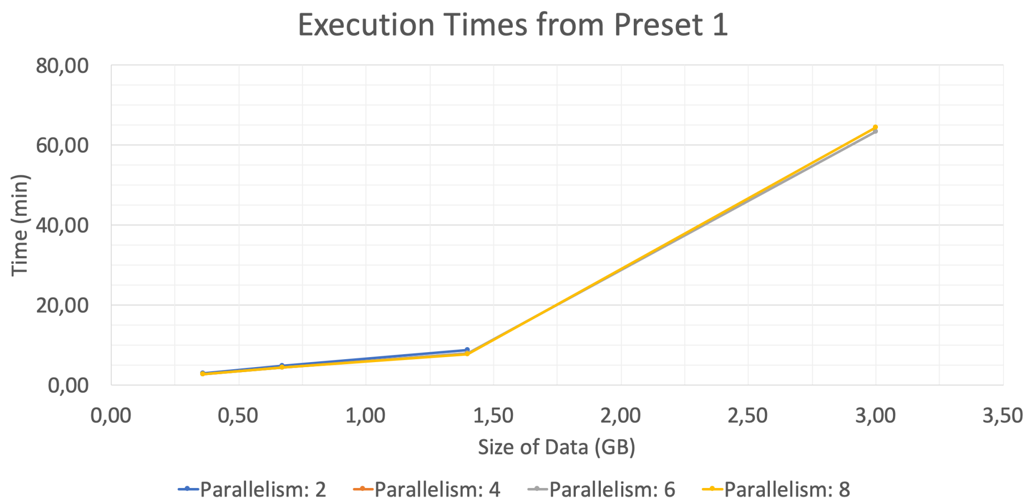

Figure 6.4: An illustration of the relationship between execution time and the parallelism factor for Preset 1.

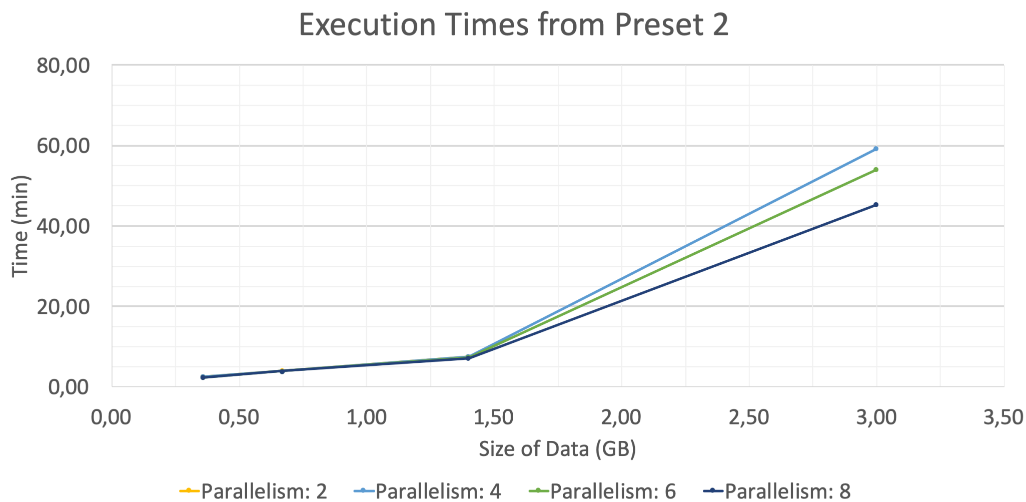

Figure 6.5: An illustration of the relationship between execution time and the parallelism factor for Preset 2.

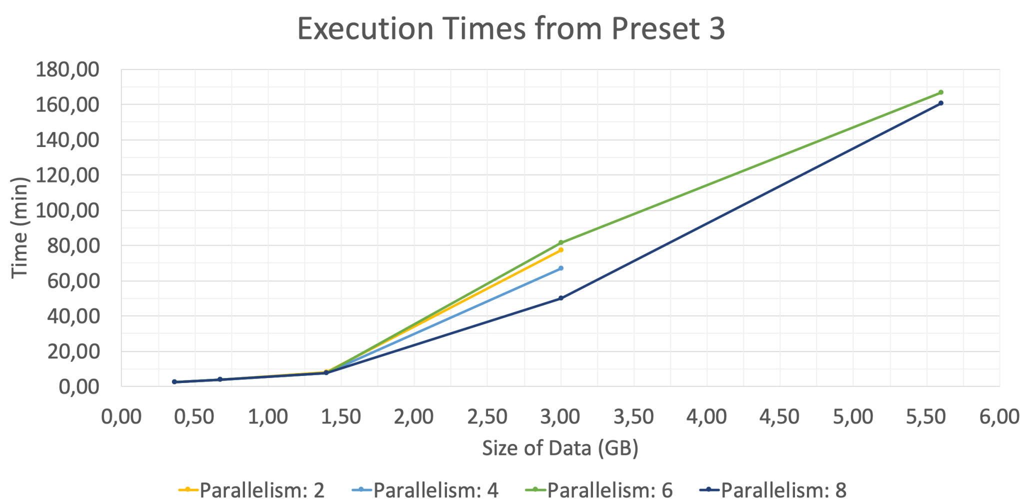

Figure 6.6: An illustration of the relationship between execution time and the parallelism factor for Preset 3.

As highlighted by the results, the solution manages to demonstrate scalability for data of increasing volume. The results also signal the fact that parsing is a memory-intensive application and computational resources do not affect the reliability of the solution. Therefore, preset 3 has the least

number of task failures in parsing the entire 24-hour dataset. Furthermore, task failures encountered with presets 1, 2 and in some of the trials with preset 3 indicate that large XML files require more executor memory and hence cause errors. Increasing the parallelism factor increases the number of data partitions which consequently reduces the number of large XML files within a single partition and therefore reduces the memory requirement per executor. Detailed results have been provided in the Appendices A.

6.3 Speed

As illustrated in Figures 6.4, 6.5 and 6.6 in the previous section, all the three presets provide similar performance. Theoretically, parsing is a sequential process that requires traversal through the contents of a file, which has a time complexity of $O(n)$. In all the experiments for the three presets, the execution time remains linear for the first three smaller datasets 1, 2 and 3 but deviates significantly for the remaining two larger datasets 4 and 5. Raw data is read as unsplittable XML files, which are divided into equally-sized data partitions. Since the RDD is partitioned randomly, some partitions might be assigned a handful of large files while others might receive numerous small files. Hence, a task receiving a partition consisting of large XML files will take longer to parse the data. Section 6.5 provides more insight for these bottlenecks in speed. The existing parsing solution at EA takes approximately 8.5 minutes to parse 30 minutes of data on a single machine. In comparison, the proposed parsing solution processes 0.36 GB (roughly 6.25% or 90 minutes of the 24-hour data) in 2.5 minutes on three machines. Furthermore, a rough estimate for parsing the entire 24-hour dataset for the existing solution was 6.8 hours or 408 minutes. Figure 6.7 depicts a comparison of the execution time of the proposed solution and the estimated execution time of the existing solution.

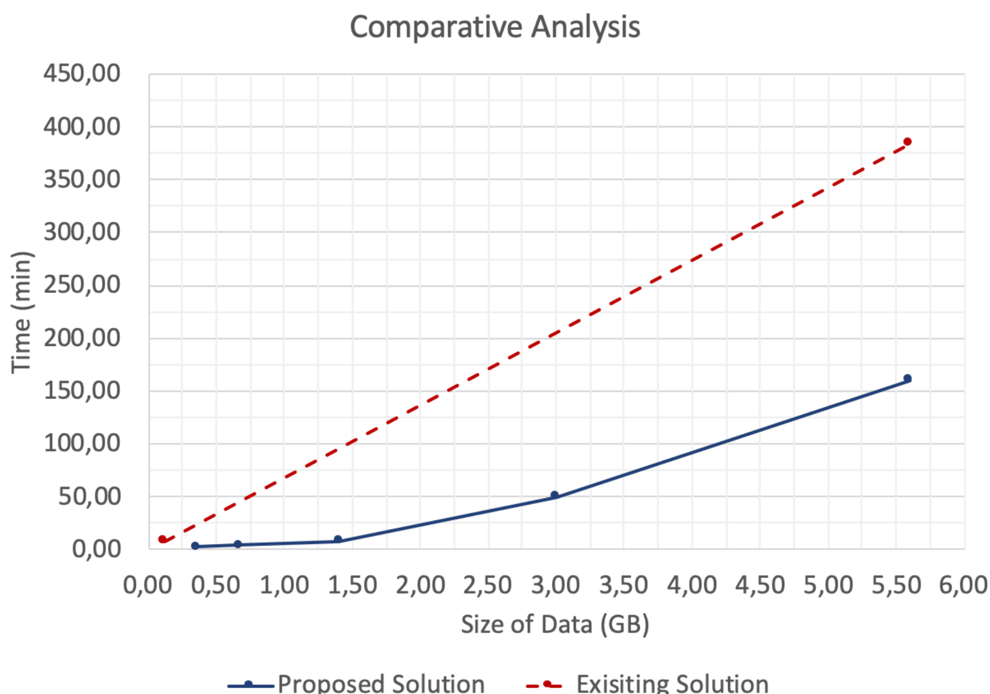

Figure 6.7: A comparative analysis of the speed of the proposed and existing parsing solutions.

As indicated by Figure 6.7 the proposed solution achieves a significant speed-up. As identified by the experiments, the solution manages to parse the entire 24-hour dataset within 160 minutes or approximately 2.67 hours. Hence, a speed-up factor of approximately 2.55 is a considerable improvement.

6.4 Summary

As illustrated in Figure 6.7, the overall speed of the proposed solution is superior to the existing solution. On the other hand, the solution is unable to attain the expected efficiency that a distributed processing framework offers. Ideally, the relationship between the execution time and the size of the data must be linear. But in the experiments for datasets 4 and 5, the execution time exceeds the expectation considerably. As mentioned earlier, this is caused by unevenly sized XML files in the datasets resulting in the executors receiving imbalanced workloads and also causing out-of-memory failures. The main reason behind the out-of-memory failures is large XML files. For parsing, the solution leverages Python libraries to load the entire

DOM tree of each XML file into memory. Therefore, all inherent content of the XML file along with temporary variables and data structures utilized by the parsing application explode in memory and exceed the maximum executor memory. To avoid this, one method is to split large XML files into smaller ones retaining their schema. The second method is to increase the parallelism factor and hence the data partitions to the extent that task failures are non-existent within the application. In the second method, a higher parallelism factor leads to more inter-node file transfers within the cluster as every node has to keep a copy of the files to ensure fault-tolerance. Since this approach introduces execution time overheads due to frequent data transmissions within the cluster nodes, splitting the XML files is a feasible solution as the next step for improving the current solution.

Executor workload imbalances are detected by examining the execution times of each task from the Spark application History Server User Interface (UI) [38]. Table 6.1 highlights the distribution of task execution times for preset 3 on dataset 5:

Execution time distribution				
Minimum	*25th percentile*	*Median*	*75th percentile*	*Maximum*
35 s	1.4 min	1.6 min	6.0 min	31 min

Table 6.1: Distribution of task execution times for Preset 3 on Dataset 5.

In summary, the experiments suggest that Spark applications must be configured to ensure maximum memory allocation for each executor. As highlighted by the results, preset 3 causes the least number of task failures and hence better reliability compared to the other presets. Also, for a fixed set of resources, the results suggest that the `spark.default.parallelism` or the total number of data partitions must be at least 8 times the total number of CPU cores available in the EMR cluster to attain optimal reliability and speed in parsing a 24-hour dataset. In the case of dynamic EMR resource allocation through auto-scaling policies, studies [65] suggest a value between 2 to 4 times the number of CPU cores in the cluster. This deviation in the parallelism can be accounted for by the varying XML file sizes within the datasets. Thus, a higher parallelism factor leads to a large number of granular data partitions and hence more reliability, scalability, and speed. In addition, the number of executor instances (N_e) and the executor memory (E_{mem}) also affect the reliability of the solution as indicated earlier.

6.5 Discussion

The proposed parsing solution was not able to attain the optimum scalability for the larger two datasets in the experimental phase. A major reason behind this was the uneven file size distribution of the raw XML data, which leads to imbalanced workloads among the executors as highlighted by Table 6.1. Since the proposed solution is designed to parse XML files as indivisible units, the per XML file workload for each executor cannot be distributed evenly. For this reason, large XML files cause a significant rise in the overall execution time as highlighted in Figure 6.6 even after the optimal Spark configuration. Figures 6.8 and 6.9 illustrate the degree of imbalance introduced by the varying XML file sizes by comparing the smallest and largest sizes of the compressed XML files in the five datasets selected for the experiments:

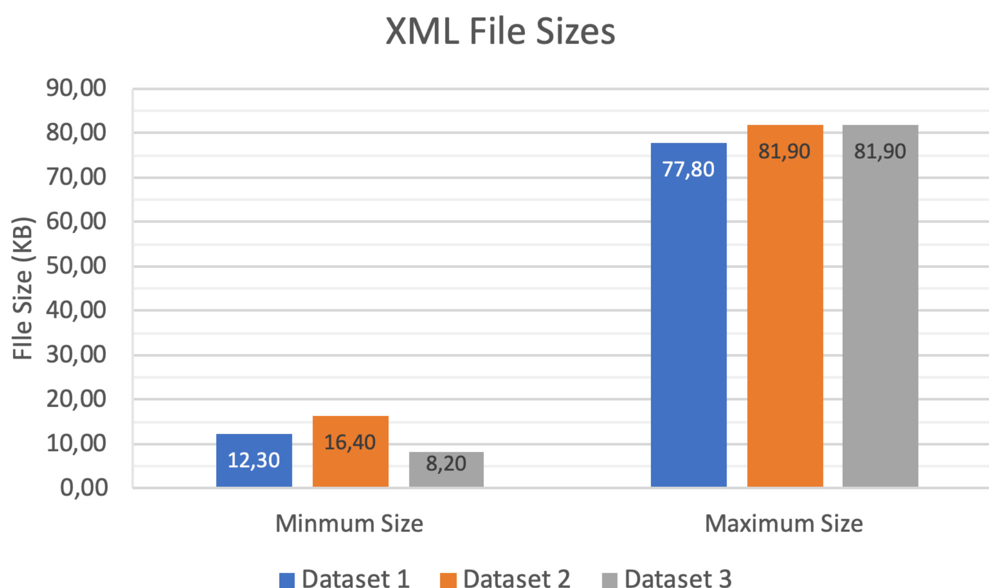

Figure 6.8: An illustration of the minimum and maximum compressed XML file sizes in Datasets 1, 2 and 3.

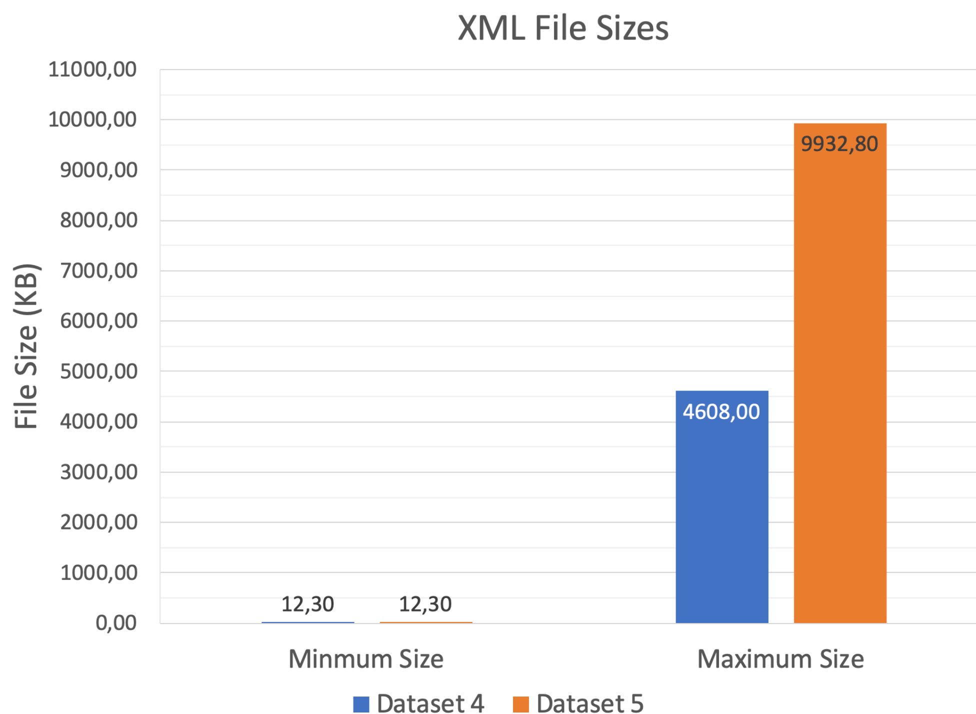

Figure 6.9: An illustration of the minimum and maximum compressed XML file sizes in Datasets 4 and 5.

Upon closer inspection of the compressed XML file size distribution for each dataset with a folder explorer tool, we grouped the sizes into three categories, i.e., *Small*, *Medium* and *Large*. *Small* files have sizes under 100 KB. *Medium* files have sizes between 100 KB and 5 MB. Lastly, *Large* files have sizes above 5 MB. The absence of *Medium* and *Large* file size categories in the three smaller datasets 1, 2 and 3 accounts for their results in terms of scalability and speed. Similarly, Figure 6.10 illustrates the file distributions in the larger two datasets 4 and 5 by the aforementioned categories.

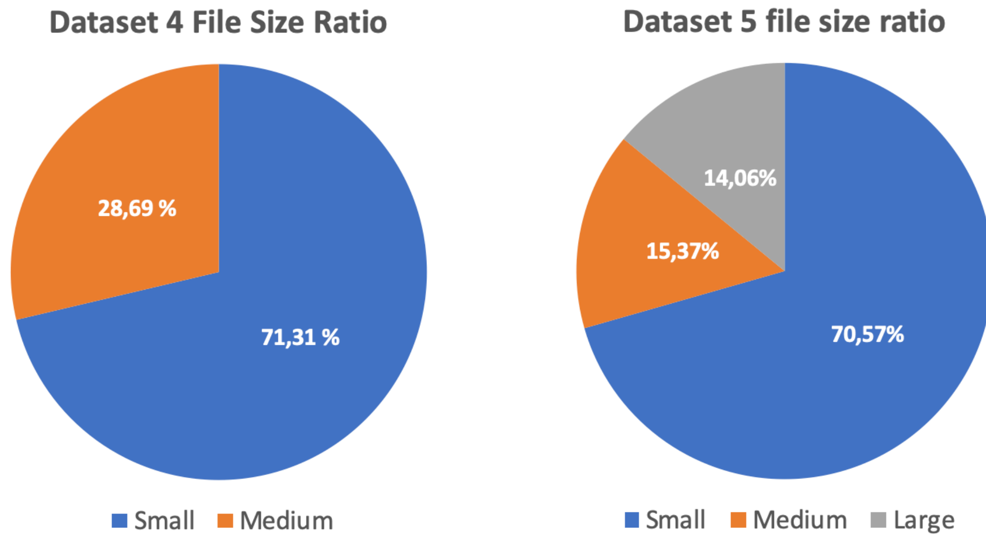

Figure 6.10: An illustration of the distribution of XML files by size categories in Datasets 4 and 5.

The presence of the only *Small* files in dataset 1, 2 and 3 indicate the reason for significant deviation in the execution times for datasets 4 and 5. To ensure an even distribution of *Small*, *Medium* and *Large* files for each executor, a salting technique is applied to the RDD. This technique involves the addition of a random number to the RDD keys for each XML file, resulting in more efficient shuffling of the XML files to the partitions. Table 6.2 highlights the distribution of task execution times for preset 3 on dataset 5 after applying the salting technique:

Execution time distribution (after salting)				
Minimum	*25th percentile*	*Median*	*75th percentile*	*Maximum*
4.5 min	5.5 min	6.1 min	11 min	27 min

Table 6.2: Distribution of task execution times for Preset 3 on Dataset 5.

The salting technique boosts the per executor resource utilization by ensuring even distribution of the XML files belonging to different size categories. But it still fails to make a significant improvement in the scalability and speed of the solution as the executors have considerably large workloads and consequently might fail to meet the resource requirements for certain data partitions. Furthermore, salting also introduces execution time overhead for the key randomization procedure, which is proportional to the size

of the data. As a result, the earned benefit is not worth the additional execution time overhead of repartitioning and shuffling the XML files across the executors. As discussed in the previous section, task failures and longer execution times are a result of large XML files within the partition. Despite even distribution of the XML files, the salting technique only ensures that an approximately equal number of files is allotted to each partition assigned to an executor. But the effective processing of each XML file concerning its size and content is beyond the capability of the Spark application. Hence, when large compressed XML files are extracted into memory, the content and intermediate data structures holding the raw data for parsing explode in size and exceed the maximum resources assigned to each executor. A quick solution to prevent memory exceptions is to limit the number of files within each data partitions by increasing the number of data partitions. But since the ultimate goal of the solution is to achieve reliability at the minimum possible cost by reducing the execution time, this solution is not viable.

6.5.1 Next steps

Due to the time constraints for the thesis, dynamic EMR resource allocation strategies and auto-scaling policies were not employed for the proposed solution. Also, since the infrastructure was in a test environment, the experimental phase was conducted on data for a specific vendor. As a second iteration to the proposed solution, we designed an application to detect XML files with sizes occurring within the *Large* category and splitting them through a SAX parser into smaller files with more manageable content. A SAX parser operates in a sequential manner identical to a DOM parser with the only difference being the memory usage. A DOM parser loads the entire tree of tags and their underlying content into memory. It is suitable for manipulations within the tag content and keeps the state of parent and child tags during the parsing process. A SAX parser, on the other hand, follows a stateless method of loading a single tag into memory at a time. It is suited to scenarios which only require traversal and storage of the XML content without any modifications. Due to resource usage expenses on the cloud, we were unable to conduct an empirical evaluation for additional tests with the split XML files on all the five datasets. Therefore, we conducted three dry runs on Dataset 5 after splitting each XML file based on the frequency of a tag within its DOM. As depicted by Figure 2.9, we selected the *measValue* tag and set a cut-off value of 1000 occurrences of the stated tag to split the XML file. The splitting process yielded numerous small files instead of a smaller quantity of large XML files with roughly the same cumulative size. Table 6.3 provides the results obtained from the dry runs.

Post-Splitting Dry Runs			
Dry Run #	*Time (minutes)*	*TaskFailures (%)*	*Parallelism Factor*
1	44	0	2
2	47	0	2
3	51	0	2

Table 6.3: Results obtained from the three dry runs conducted after splitting the large XML files.

Based on the results from Table 6.3, the entire 24-hour takes an average time of 47.33 minutes. In comparison to the existing solution at EA, the reduction in execution time has been brought down further to 87 % with a speed-up factor of 9.37. Provisioning equally-sized granular XML files to the solution has assured two characteristics. Firstly, due to equal workload among the executors, the scalability of the solution now remains uniform throughout the five datasets, similar to its uniformity for the three smaller datasets 1, 2 and 3. Hence, the execution time is linearly dependent upon the size of the dataset. Secondly, granular XML files have now resulted in the executors parsing each XML file in shorter durations. Since the parsing process is sequential, therefore smaller files lead to better exploitation of the overall parallelism of multiple executors.

Chapter 7

Related work

This chapter highlights existing research work on big data processing on the cloud. It briefly discusses the approaches from the relevant research papers and compares their published results with those of the proposed parsing solution.

Related studies [67, 68] conducted on Spark and Hadoop feature cloud-based infrastructure for the evaluation of the solution. These solutions leverage the RDD API in Spark combined with scalable heterogeneous resources from the cloud in the form of EC2 instances. The first solution [67] employs a 9-node cluster to processes half a Terabyte (TB) of data in under 4 hours. From the aspect of data volume, our proposed solution processes around 60 GB on uncompressed XML files on 3 homogeneous nodes. Converting these metrics to a single node comparison with the same time-frame, a processing speed of 0.042 TB per hour compared to our solution's 0.007 TB per hour highlights two details. Firstly, it exposes the potential parallelism not being utilized in our proposed solution. As illustrated by the scalability plots in Chapter 6, the solution's throughput can be brought up to 0.027 TB per hour after improving the overall execution time to around 50 minutes for a 24-hour network data feed as demonstrated in the second iteration for the solution. Secondly, the specifications of the EC2 instance also contribute to the throughput. As highlighted in our results, 3 homogeneous EC2 instances compared with 9 heterogeneous instances would lead to an inconclusive comparison. Lastly, the processing operations are computationally-intensive arithmetic operations in the related solution. For our solution, the operations are memory-intensive scanning and string storage operations. Similarly, the solution [68] receives input data as a stream instead of as files through the Spark streaming API. Operating on a scalable cluster of 100 EC2 nodes, it offers a processing throughput of 60 million records per second. In comparison, our proposed solution offers a processing

rate of 0.021 million records per second on 3 nodes (0.7 million when scaled to a 100 nodes). After the second iteration, the throughput of the proposed solution has been increased to 0.075 million records per second (2.5 million when scaled to a 100 nodes). Once again, the emphasis is brought upon the way the input data is loaded into the executors across the cluster. Streaming ensures consistent data inflow on a controllable scale instead of taking a considerably large amount of time to load the entire dataset at once. Despite these shortcomings, the important facts indicated by these results are that Spark coupled with YARN provides sufficient scalability for handling massive datasets. Furthermore, 51.3% memory utilization for caching frequently accessed data and an error rate of 3% for data-intensive operations [67] signals the degree of efficiency and reliability that can be achieved with Spark.

To support input data complexity encountered by our solution, results [69] from a comparative analysis between Hadoop MapReduce and Spark suggest that Hadoop provides more stability in executing memory-intensive operations compared to Spark. This condition favors scenarios where the input file size exceeds the memory resources allocated to Spark executors. This choice is not favorable in situations where a large input file can be divided evenly without losing its schema. Since Hadoop does not provide a significant speed-up compared to Spark, it is viable to manipulate the input data to adhere to the distributed structure of the Spark programming framework.

Other work [70, 71] was focused on the optimization of Spark. The first study [70] focuses on the Spark engine by presenting an improvement to its shuffling engine by the name of *Sparkle*. Although this approach benefits most from machines with high memory specifications, it still offers a 1.6x to 5x performance improvement for scale-out clusters. As of Spark 2.0, this optimization is already available. In the context of our proposed solution, this study clearly states that file shuffling across multiple nodes in a cluster does not cause a major performance degradation. The second study [71] focuses more on Spark parameter tuning. It highlights 12 application-specific parameters, some of which have already been addressed in our solution design. As suggested by the results, parameter tuning brings down the execution times of applications by 20%. In support of these results, parameter tuning for our solution enhanced the overall reliability and enabled it to process an entire 24-hour dataset with proper parameters. Besides, compression codec choices and shuffle strategies for the input files could be considered as potential future work.

Chapter 8

Conclusion

In this thesis, we proposed a distributed parsing solution designed to utilize scalable cloud-based resources to process massive amounts of telecom network data. Our solution utilizes Spark as the programming framework for distributed computations and AWS as the platform for resource provisioning.

After deployment on a fixed set of resources in the cloud, the solution managed to parse an entire 24-hour PM dataset of roughly 60 GB in 2.67 hours. In comparison to the existing solution at EA, an approximate 39 % reduction in the execution time with a speed-up factor of 2.55 signals great potential in the direction of distributed computing alternatives for data engineering processes at large organizations. Furthermore, the solution transforms the processed data into an organized format resembling relational databases to support data filtration and aggregation and analysis.

Our solution was designed to serve as a component of a data engineering pipeline. For achieving this, it was required to possess the ability to read raw data from multiple sources, allocate adequate resources for the processing phase and store the processed data in designated end-points within the pipeline without any human intervention. The solution fulfilled these requirements by leveraging micro-services provided by AWS. Although the resource allocation was static in the scope of this thesis, it proved sufficient for the datasets used in the experimental phase. For parsing mechanisms, the solution was required to distribute the raw data among multiple workers to conduct a parallelized workflow. Also, the solution was expected to transform the structure of the data to adhere to strict rules similar to those of a relational database. Lastly, it was required to store this refined data in a file-format optimized for loading onto analytics applications for subsequent querying and analysis. The solution fulfilled these expectations by employing the Spark programming framework to load raw data into distributed RDDs and subsequently transform them into a structured DataFrame after applying

parsing routines imported from EA's existing solutions. Utilizing Spark, the solution saved the resulting DataFrame as Parquet, a columnar file-format optimized for high-speed loading of large datasets into applications and supporting superior compression codecs.

For evaluating the efficiency of our solution and the optimal Spark parameters, we defined experiments to test its reliability, scalability, and speed. A total of five datasets of varying sizes were processed by the parsing solution operating on three different Spark presets. The solution managed to fulfill the reliability and speed requirements for all the datasets on the optimal preset. Scalability was demonstrated for the smaller three datasets but not for the remaining two larger datasets. Upon closer inspection, large XML files present within the larger two datasets overburdened the Spark executor resources, thus affecting the overall scalability. After employing an additional mechanism for splitting the XML files in the solution, the desired scalability results were eventually achieved and the reduction in execution time increased from 39 % to 87 % with a speed-up factor of 9.37

In conclusion, the solution demonstrates a high potential for tackling parsing challenges posed by big data in the telecom environment. As suggested by related studies, Spark is designed to offer scalability, and many published results also highlight this fact. For future work, the employment of autoscaling policies through AWS for better resource utilization and provisioning would enable this solution to adapt to massive data volumes without any significant design changes. Lastly, the exploration of the AWS EC2 instance families coupled with the deployment of clusters comprising heterogeneous instances would also provide valuable insight regarding the maximum speedup that can be achieved by this solution.

Bibliography

[1] Constantinos Costa and Demetrios Zeinalipour-Yazti. Telco big data research and open problems. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 2056–2059. IEEE, 2019.

[2] Ericsson Mobility. Ericsson mobility report, 2018.

[3] Jacques Bughin. Telcos: The untapped promise of big data. `https://www.mckinsey.com/industries/telecommunications/our-insights/telcos-the-untapped-promise-of-big-data`.

[4] Investopedia. Data analytics. `https://www.investopedia.com/terms/d/data-analytics.asp`.

[5] Hyoun Park James Haight. Closing the data preparation gap in telecommunications. Technical report, Blue Hill Research, 2016.

[6] Florian Endel and Harald Piringer. Data wrangling: Making data useful again. *IFAC-PapersOnLine*, 48(1):111–112, 2015.

[7] Wayne W. Eckerson. Modern data pipelines. Technical report, Eckerson Group, 2016.

[8] John Meehan, Cansu Aslantas, Stan Zdonik, Nesime Tatbul, and Jiang Du. Data ingestion for the connected world. In *CIDR*, 2017.

[9] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 579–590. ACM, 2010.

[10] Cun Ji, Qingshi Shao, Jiao Sun, Shijun Liu, Li Pan, Lei Wu, and Chenglei Yang. Device data ingestion for industrial big data platforms with a case study. *Sensors*, 16(3):279, 2016.

[11] Tim O'reilly. *What is web 2.0.* " O'Reilly Media, Inc.", 2009.

[12] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. *IEEE transactions on knowledge and data engineering*, 26(1):97–107, 2013.

[13] Marcos D Assunção, Rodrigo N Calheiros, Silvia Bianchi, Marco AS Netto, and Rajkumar Buyya. Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79:3–15, 2015.

[14] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information systems*, 47:98–115, 2015.

[15] Yuri Demchenko, Paola Grosso, Cees De Laat, and Peter Membrey. Addressing big data issues in scientific data infrastructure. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 48–55. IEEE, 2013.

[16] Danyel Fisher, Rob DeLine, Mary Czerwinski, and Steven Drucker. Interactions with big data analytics, interactions, v. 19 n. 3. *May+ June*, 2012.

[17] Dimitrios Zissis and Dimitrios Lekkas. Addressing cloud computing security issues. *Future Generation computer systems*, 28(3):583–592, 2012.

[18] R Sravan Kumar and Ashutosh Saxena. Data integrity proofs in cloud storage. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*, pages 1–4. IEEE, 2011.

[19] K Weber, G Rincon, A Van Eenennaam, B Golden, and J Medrano. Differences in allele frequency distribution of bovine high-density genotyping platforms in holsteins and jerseys. In *Western section American society of Animal science*, page 70, 2012.

[20] Dunren Che, Mejdl Safran, and Zhiyong Peng. From big data to big data mining: challenges, issues, and opportunities. In *International conference on database systems for advanced applications*, pages 1–15. Springer, 2013.

[21] Peter Mell and Tim Grance. The NIST definition of cloud computing (draft), NIST spec. *Publ*, 800:145, 2011.

[22] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *2009 Fifth International Joint Conference on INC, IMS and IDC*, pages 44–51. Ieee, 2009.

[23] Zibin Zheng, Jieming Zhu, and Michael R Lyu. Service-generated big data and big data-as-a-service: an overview. In *2013 IEEE international congress on Big Data*, pages 403–410. IEEE, 2013.

[24] Apache Software Foundation. Kafka: A distributed streaming platform. `https://kafka.apache.org/`.

[25] Amazon Web Services (AWS). Amazon Kinesis Data Streams. `https://aws.amazon.com/kinesis/data-streams/`.

[26] Amazon Web Services (AWS). Amazon S3. `https://aws.amazon.com/s3/`.

[27] OpenStack. OpenStack Swift. `https://docs.openstack.org/swift/latest/`.

[28] Microsoft Azure. Blob Storage. `https://azure.microsoft.com/en-us/services/storage/blobs/`.

[29] Google Cloud. Google Cloud Storage. `https://cloud.google.com/storage/`.

[30] Apache Software Foundation. Apache Hadoop. `https://hadoop.apache.org/`.

[31] MongoDB. The database for modern applications. `https://www.mongodb.com/`.

[32] Apache Software Foundation. Manage massive amounts of data, fast, without losing sleep. `http://cassandra.apache.org/`.

[33] Oracle Corporation. MySQL. `https://www.mysql.com/`.

[34] PostgreSQL. PostgreSQL: The world's most advanced open source relational database. `https://www.postgresql.org/`.

[35] Amazon Web Service (AWS). Amazon EC2. `https://aws.amazon.com/ec2/`.

[36] Google Cloud. Compute Engine. `https://cloud.google.com/compute/`.

[37] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[38] Apache Software Foundation. Apache Spark. `https://spark.apache.org/`.

[39] Google Cloud. Google App Engine. `https://cloud.google.com/appengine/`.

[40] Amazon Web Services (AWS). Start building on AWS today. `https://aws.amazon.com/`.

[41] Microsoft. Azure HDInsight. `https://azure.microsoft.com/en-in/services/hdinsight/`.

[42] Panoply. Simple data management for analytics. `https://panoply.io/`.

[43] Etleap. Perfect data pipelines from day one. `https://etleap.com/`.

[44] Inzata. Transform the data you have into the answers you need. `https://www.inzata.com/`.

[45] Looker. The data you need now. a platform built for tomorrow. `https://looker.com/`.

[46] Bill Chambers and Matei Zaharia. *Spark: the definitive guide: big data processing made simple.* " O'Reilly Media, Inc.", 2018.

[47] Li Laxmi Bhuyan Zhao and Laxmi Bhuyan. Performance evaluation and acceleration for XML data parsing. In *9th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), Austin, Texas.* Citeseer, 2006.

[48] L Letho and Tiina KILPELÄINEN. Real-time generalization of geodata in the web. *International Archives of Photogrammetry and Remote Sensing*, 33(B4/2; PART 4):559–566, 2000.

[49] 3GPP. The mobile broadband standard. `https://www.3gpp.org/`.

[50] Ceph. The future of storage. `https://ceph.io/`.

[51] MinIO. Object storage for AI. `https://min.io/`.

[52] Docker. Modernize your applications, accelerate innovation. `https://www.docker.com/`.

[53] Elastic. ElasticSearch: The heart of the Elastic Stack. https://www.elastic.co/products/elasticsearch.

[54] RedisLabs. Redis. https://redis.io/.

[55] Elastic. Kibana: Your window into the Elastic Stack. https://www.elastic.co/products/kibana.

[56] Amazon Web Services (AWS). Amazon S3. https://aws.amazon.com/s3/.

[57] Amazon Web Services (AWS). Amazon EMR. https://aws.amazon.com/emr/.

[58] Amazon Web Services (AWS). Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/.

[59] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[60] Romeo Kienzler. *Mastering Apache Spark 2. x*. Packt Publishing Ltd, 2017.

[61] Apache Software Foundation. Apache Parquet. https://parquet.apache.org.

[62] Amazon Web Services (AWS). Amazon Step Functions. https://aws.amazon.com/step-functions/.

[63] Apache Software Foundation. Spark Row interface. https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/Row.html.

[64] Amazon Web Services (AWS). Amazon emr 5.x release versions. https://docs.amazonaws.cn/en_us/emr/latest/ReleaseGuide/emr-release-5x.html.

[65] Holden Karau and Rachel Warren. *High performance Spark: best practices for scaling and optimizing Apache Spark*. OReilly Media, Inc., 2017.

[66] Amazon Web Services (AWS). Best practices for successfully managing memory for Apache Spark applications on the Amazon EMR. https://aws.amazon.com/blogs/big-data/best-practices-for-successfully-managing-memory-for-apache-spark-applications-on-amazon-emr/.

[67] Wei Huang, Lingkui Meng, Dongying Zhang, and Wen Zhang. In-memory parallel processing of massive remotely sensed data using an apache spark on Hadoop YARN model. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(1):3–19, 2016.

[68] Zhijie Han and Yujie Zhang. Spark: A big data processing platform based on memory computing. In *2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 172–176. IEEE, 2015.

[69] Lei Gu and Huan Li. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 721–727. IEEE, 2013.

[70] Mijung Kim, Jun Li, Haris Volos, Manish Marwah, Alexander Ulanov, Kimberly Keeton, Joseph Tucek, Lucy Cherkasova, Le Xu, and Pradeep Fernando. Sparkle: Optimizing spark for large memory machines and analytics. *arXiv preprint arXiv:1708.05746*, 2017.

[71] Anastasios Gounaris and Jordi Torres. A methodology for spark parameter tuning. *Big data research*, 11:22–32, 2018.

Appendix A

Appendices

	Dataset1	**Dataset2**	**Dataset3**	**Dataset4**	**Dataset5**
P=2	2.90	4.80	8.70	N/A	N/A
P=4	2.70	4.50	7.90	N/A	N/A
P=6	2.80	4.40	7.90	63.50	N/A
P=8	2.80	4.40	7.70	64.50	N/A

Table A.1: Execution time (in minutes) for Preset 1. **P** represents the parallelism factor.

	Dataset1	**Dataset2**	**Dataset3**	**Dataset4**	**Dataset5**
P=2	2.60	4.00	7.50	N/A	N/A
P=4	2.50	3.90	7.50	59.20	N/A
P=6	2.30	3.90	7.25	54.00	N/A
P=8	2.30	3.90	7.10	45.30	N/A

Table A.2: Execution time (in minutes) for Preset 2. **P** represents the parallelism factor.

	Dataset1	**Dataset2**	**Dataset3**	**Dataset4**	**Dataset5**
P=2	2.50	4.00	8.00	77.30	N/A
P=4	2.50	3.80	7.40	66.70	N/A
P=6	2.30	4.00	7.50	81.50	166.80
P=8	2.50	4.00	7.50	50.00	160.30

Table A.3: Execution time (in minutes) for Preset 3. **P** represents the parallelism factor.