Titre: Title:	On-Demand Health Data Provisioning with Custom Temporary Data Views for Big Data Platforms
Auteur: Author:	Anas Bouziane
Date:	2023
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bouziane, A. (2023). On-Demand Health Data Provisioning with Custom Temporary Data Views for Big Data Platforms [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/53403/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/53403/
Directeurs de recherche: Advisors:	Maxime Lamothe, & Marios-Eleftherios Fokaefs
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

On-demand Health Data Provisioning with Custom Temporary Data Views for Big Data Platforms

ANAS BOUZIANE

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie informatique

Mai 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé:

On-demand Health Data Provisioning with Custom Temporary Data Views for Big Data Platforms

présenté par **Anas BOUZIANE**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Jinghui CHENG, président
Maxime LAMOTHE, membre et directeur de recherche
Marios-Eleftherios FOKAEFS, membre et codirecteur de recherche
Bram ADAMS, membre

DEDICATION

To my parents, my brother, and my sister, who have supported me throughout the entirety of my studies

Thank you for everything.

ACKNOWLEDGEMENTS

I sincerely thank Professor Maxime Lamothe for his help, support, guidance, and valuable tips in completing this work and achieving the redaction of this thesis. Also, I am sincerely grateful to Professor Marios Fokaefs for helping me through this entire master's research project with various crucial advice, suggestions, and implementation mentoring. I thank Clinique Médicale Urbaine du Quartier Latin for their financial assistance and Compute Canada's material support in implementing this work on their cloud infrastructure. I thank my family for their help and encouragement throughout my studies. Finally, I thank Professor Jinghui Cheng and Professor Bram Adams for evaluating my master's thesis.

** The icon "Api" used in this thesis was designed by Smartline – Flaticon.com.*

** The icon "Cube image" used in this thesis was designed by Freepik from www.flaticon.com*

** The icon "Flat design laptop logo template" used in this thesis was designed by Freepik using assets from www.freepik.com*

RÉSUMÉ

Deux décennies depuis l'introduction des plateformes *Big Data*, elles restent néanmoins un sujet complexe ayant plusieurs défis encore non résolus. Il y a toujours un manque de consensus concernant la plateforme idéale pour résoudre les problèmes du monde réel. En effet, chaque cas d'utilisation requiert la conception d'une plateforme de donnée spécifique. Aussi, le fait que les données continuent d'augmenter exponentiellement incite les organisations à adopter des techniques de science de données et d'apprentissage machine afin d'explorer leurs données plus efficacement. Cela nécessite d'établir les fondations d'une architecture aidant à traiter un volume important des données variées de façon rapide tout en maintenant la véracité de ces derniers. En effet, les implémentations du type *Entrepôt de Données* et *Lacs de Données* sont souvent indispensables pour manipuler un tel volume. Cependant, chaque implémentation peut être réalisée de plus d'une façon. Nous nous penchons tout au long de notre recherche vers ces défis tout en parcourant l'état de l'art de différentes techniques explorées. Ce mémoire cible la partie *backend* et stockage au sein d'un plus grand projet concernant les plateformes de données libres d'accès et suivant un modèle sur mesure pour le milieu de la santé en collaboration avec la Clinique Médicale Urbaine du Quartier Latin. Nous présentons aussi l'implémentation d'une architecture *Lac de Données* dans un cas d'utilisation spécifique au domaine de santé. Notre implémentation peut être établie *on-premise* ou dans une infrastructure infonuagique et est constituée de deux parties. La première consiste en un *backend* responsable d'effectuer les tâches d'ingestion, stockage, transformation, et transfert de données. Ce *backend* adresse aussi la problématique qui concerne le type de stockage et les types de bases de données les plus compatibles avec un cas d'utilisation spécifique dans le domaine de la santé. La deuxième partie de la plateforme est responsable de l'analyse de données. Mis à part le lac représentant le stockage principal, la majorité des autres éléments sont *cloud-native* et sont construits dans des conteneurs orchestrés avec Kubernetes. Introduire une infrastructure infonuagique au sein de notre architecture *backend* influence légèrement notre flux de données. Cela permet une modélisation des données et un stockage plus flexible en temps d'exécution selon une approche *schema-on-read*. En d'autres termes, notre architecture aide les utilisateurs à personnaliser leurs données pour chaque projet d'exploration de science de données à partir d'un niveau non atteignable dans les architectures *Big Data* conventionnelles. Finalement, nous évaluons notre architecture d'un point de vue d'utilisabilité.

ABSTRACT

Almost two decades have passed since the introduction of Big Data Platforms (BDPs), yet they remain a confusing subject with multiple challenges that are still unaddressed. There is still a lack of consensus on the ideal Data Platform solution for real-world problems, as every use case requires a specific Data Platform design. Moreover, the claim remains true that data keeps increasing exponentially, making organizations more inclined to adopt machine learning and data exploration techniques to tackle their problems. This necessitates laying out the foundations to design an architecture that can store and process structured, semi-structured, and unstructured data. In other words, there is still a need to determine the best way to process a large volume of diverse data at high speed (velocity) while maintaining the veracity of the processed information. Indeed, implementations that use Data Warehouses and Data Lakes are often necessary for manipulating large data volumes. However, each of these can be set up in various ways. We address those issues in our research by defining, classifying, and categorizing state-of-the-art Data Platform architectures and pointing out the critical elements for building a resilient and performant data backend architecture. Our thesis focuses on the backend and storage part of a more significant open-source on-demand Data Platform project for healthcare in collaboration with Clinique Médicale Urbaine du Quartier Latin. We also present our implementation of a Data Lake for a use-case centred around healthcare. Our implementation can be built on-premise or in the cloud and comprises two parts. The first part consists of a backend that mainly focuses on the ingestion, storage, transformation, and transfer of the data. This backend also addresses the type of storage and database best suited for one specific use case within the healthcare field. The second part is responsible for data analytics. Putting aside the storage repository component, every other element in the Data Platform is cloud-native and is built inside containers orchestrated with the help of Kubernetes. Introducing cloud-native container orchestration in our backend architecture slightly influences the data workflow of our platform. This gives us the flexibility to shape and store the data at execution time following a schema-on-read approach. In other words, our architecture helps users tailor their data for each data science exploration project from levels that are usually unreachable in conventional Big Data architectures. We finally evaluate our solution from an architectural and usability viewpoint.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
AWS	Amazon Web Services
BDP	Big Data Platform
CDP	Cloudera Data Platform
CRD	Custom Resource Definition
DBMS	Database Management System
GCP	Google Cloud Platform
HDP	Hortonworks Data Platform
JSON	JavaScript Object Notation
K8s	Kubernetes
NoSQL	Not Only SQL
OLAP	Online Analytical Processing
PV	Persistent Volume
PVC	Persistent Volume Claim
QoS	Quality of Service
SBT	Scala Build Tool
TDV	Temporary Data View
XML	Extensible Markup Language
YAML	Yet Another Markup Language
YARN	Yet Another Resource Negotiator

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Now that generating tremendous amounts of diverse data has become ubiquitous, traditional storage systems face more significant challenges for storing, cleaning, aggregating, and transforming the collected data into meaningful insights. The medical field is no exception, especially now that the trend is shifting toward digital records. Indeed, a medium-sized healthcare institution in China, according to Wang et al. [1], can register up to 20TB of health data annually. This number may rise to 1PB for larger healthcare institutions. Large amounts of data open the door for revolutionary research opportunities, which may partly explain why machine learning and data science exploration techniques are gaining traction in medical science. Such data can be used for predictive analytics and early diagnosis and help shape epidemiological models. However, the exponential data growth presents an increased demand for storing and managing unprecedented volumes of data from different sources and formats at high speed. Those demands embody the problem that is concerning Big Data. The term is associated with any data characterized by the three primary V describing Volume, Velocity, and Variety [2]. The last one requires particular attention in the medical field and epidemiology, known for having many data formats applied in various medical procedures. For the sake of simplicity, we categorize this variety into three main groups. Structured data, for instance, represent digital records generated through the transaction-based relational data model. Unstructured data can embody X-rays imagery, scanned medical reports, or textual data. Semi-structured data can result from transformed records stored in a non-relational form. Such variety demands non-trivial data processing and storage techniques.

The existing solution that makes Big Data exploitable by organizations is called Data Platforms. Big Data Platforms illustrate a combination of open-source or/and enterprise tools for managing Big Data. Their primary aspiration is to provide stakeholders with valuable business insights related to their domain of interest. Those platforms vary between a combination of open-source tools and proprietary ones. They can either follow an on-premise or cloud-native implementation. Examples of such platforms are the likes of the Cloudera Data Platform [3] (managed open-source Data Platform), Oracle Big Data Platform (proprietary) [4], and AWS Big Data Platform (cloud-native Data Platforms). In one way or another, most platforms use or get inspiration from the Hadoop ecosystem and other open-source tools for managing Big Data. Each one is responsible for one fraction of the Big Data workflow process.

For BDP to work properly, we rely on Big Data architectures to effectively build its backend, thus providing a resilient "backbone" to those platforms. Specifically, we go through the state-of-the-art to bring up and classify the prevalent architectures used in the backend to effectively store and process various data from various sources and formats. We touch upon three widespread high-level architectures: Data Warehouses [5], Data Lakes [2, 6], and Data Lakehouses [7, 8].

Data Warehouses are the traditional data management system architecture solely based on structured data. They remain, until this point, the most applied data architecture in the industry [2]. Yet they clearly show data incompatibilities, concerns, and inconsistencies when processing semi-structured and unstructured data. Despite that, they possess architectures and techniques that let them remain a serious Big Data contender. Data Lakes, however, have been built explicitly around Big Data. They handle various data formats that do not necessarily follow predefined schema patterns. Data Lakes are indeed most suited for managing Big Data and have the potential to unlock access to machine learning and data science exploration techniques. A new emerging concept named Lakehouse tries to merge between Data Warehouses and Data Lakes functionalities [9]. Yet, as a new concept, studies are still being performed on its characteristics and applicability in Big Data.

In essence, our objectives throughout this thesis are:

- Exploring and selecting a backend architecture for Big Data that can efficiently manage health data and address the challenges they give rise to.
- Introduce our custom solution for handling the high "*Variety*" aspect that is widely present in Health Big Data.

Working through achieving the said objectives is supported by four research questions that help us define this thesis's scope and explore different possible solutions.

For the sake of efficiently managing Health Big Data, we must first understand the particularity of its data. The widespread Big Data backend architectures can be implemented in multiple ways and benefit from a panoply of tools and methodologies for ingesting, storing, and processing health data. Different health projects require various data formats to work with. And each data format benefits from different storage tools for efficient data management tasks. Therefore, we must find a way to serve users their queried data regardless of their particularities while at the same time

keeping the solution dynamic, cost-effective and efficient. Thus, we establish the following questions.

1. What are the types and characteristics of the targeted health data?
    a. Do they need a dedicated infrastructure/platform different from the commercially available solutions?
2. What factors do we need to consider for building platforms of such calibre?
    a. What are the widespread Big Data architectures that exist?
    b. What (open-source) data storage tools are being used to accomplish such a purpose?
3. What is the most efficient and flexible way to store health data for different needs?
    a. How can we anticipate the backend storage component for future data exploration projects?
    b. Which data storage model is the best fit for healthcare research projects?
4. Granted, we possess a Data Platform capable of processing various types of health data, how can we ensure it will provide custom data storage models for targeted health projects?

Additional to the known Big Data characteristics (3 Vs), one of the prominent particularities of health data is its high variety aspect. We propose a slightly modified Lambda architecture with such particularities in mind and introduce our custom TDV as a dynamic means for storing compatible tailored datasets, among other features. We explore questions 1 and 2 in sections 2.2, 2.3, and 2.4. Exploring those questions answers how we can help merge between a resilient backend architecture and an on-demand open-source Data Platform. Those sections also helped us build the foundations for answering questions 3 and 4 in chapter 3.

Following the literature study and throughout this thesis, we review the most widespread Big Data Platforms and the tools that incorporate them. We review the leading Big Data architecture implementations from surveys and research papers, point out their respective characteristics, and show how they can be applied in Big Data use cases. We also illustrate each Big Data architecture's different and most famous implementations. Data Lakes, for instance, benefit from a large set of components, tools, and architectures. Some of its solutions call for custom metadata processing or

implement specific features addressing special needs. Others follow well-known architectures such as Lambda or Kappa for real-time data processing capabilities.

It should be noted, however, that our focus remains directed toward the different types of architectures that are most suited for designing the backend of BDPs. Our architecture primarily intends to assist epidemiologists in exploring massive heterogeneous data and use it for data science exploration and machine learning through web-based applications. More specifically, we investigate approaches, methodologies, and designs that can ensure maximum flexibility for (1) helping data scientists/epidemiologists assemble tailored datasets from massive heterogeneous raw data repositories (coming from diverse external sources) at the storage level as opposed to the conventional data workflow which is done at the application level, and (2) helping them dynamically store their targeted datasets in the most compatible data storage model possible.

We present our solution of an on-demand and open-source Big Data Platform prototype that can answer diverse identified requirements related to healthcare generally and epidemiology specifically. Our backend infuses capabilities for efficiently ingesting, storing, and serving a variety of data coming from various sources. Our platform achieves this by deploying containerized resources on-the-fly following the data processing requirements on a hybrid architecture managed by a cloud-native orchestrator operating on a distributed Big Data repository.

More explicitly, our contributions are summarized as follows:

- Investigate health data characteristics and their impact on BDPs; Explore various BDPs and analyze their particularity; Assess the tools implemented in BDPs and examine how they operate; Explore different backend architectures for different needs and review other techniques for tackling Big Data challenges.
- Propose an efficient backend architecture for Big Data in healthcare capable of ingesting, storing, and processing health data following the identified requirements and modern Big Data challenges.
- Present a solution for handling the “Variety” characteristic of Big Data through a flexible architecture that relies on the TDV concept for storing, managing, and serving tailored datasets on demand.

- Build a prototype BDP as a proof of concept for both the backend and the TDVs and evaluate its usability.

The remaining chapters of this thesis are organized as follows. Chapter 2 explores the literature review related to the different subjects of this thesis regarding Big Data Platforms, architectures, tools, backend implementations, storage techniques, and their relation to healthcare and reviews the state-of-the-art accordingly. Chapter 3 further highlights the structure, components composing such a platform, and the backend architecture responsible for the data processing workflow. We also present a prototype made with open-source tools such as the Hadoop ecosystem, Spark, and Kubernetes and give examples of data workflows unique to our architecture. We show how our prototype can work on-premise or in a cloud-native configuration. We also expand on the tools used to build our prototype and elaborate on the considered Data Lake architecture and the key elements that drive architectural decisions for better data maintainability assuring maximum flexibility in data models. The final part of Chapter 3 explains our innovative concept for implementing a schema-on-read data transformation approach and creating a NoSQL Temporary Data View (TDV) for epidemiologists to fetch and create their notebook dataset using the best-suited storage format for each specific data exploration task.

Furthermore, we evaluate our solution in Chapter 4 by using a comparative study between our flexible architecture using the dynamic storage approach and other conventional methods. We first compare our solution with famous Data Lakes from a usability standpoint. Then we compare the flexibility and extensibility of our implementation with different Big Data architectures, such as Data Warehouses. Chapter 5 discusses our solution and answers the questions from Chapter 2. Lastly, in Chapter 6, we discuss our solution, highlight limitations, and address future works for fixing those.

CHAPTER 2 LITERATURE REVIEW

2.1 Introduction

This section elaborates on the state-of-the-art documentation linked to backend distributed storage architectures for Big Data. We gathered different scientific papers that look at various Big Data challenges from a broad and general perspective and gradually narrowed it down to healthcare and epidemiology.

In Section 2.2, we gather insights into the different types of health data, their characteristics, and how they influence Big Data architectural design decisions. Section 2.3 expands on architectures for handling Big Data. Section 2.4. overviews various open-source tools for managing Big Data and where they fit in the overall architecture. Section 2.5 examines different relational and non-relational data models used in health Data Platforms' storage components. The same section also helps lay the groundwork for exploring how the studied data models can be used across layers of our prototyped architecture. Finally, section 2.6 reviews papers exploring various solutions for efficiently storing Big Data using dedicated platforms, generally first, then in healthcare.

2.2 Data in healthcare

Data represent any digitalized record that a person or software has written. The conventional way that was predominantly used to store it was through databases. However, this only applies to data manageable in one location and following a pre-defined schema. Now, the medical field is being exponentially flooded with new data. As reported in [10], 2011 has seen medical-related data measure around 150 Exabytes with an increase between 1.2 and 2.4 Exabytes annually. This trend has dramatically increased since various wearables and other health IoT devices were invented for professional and commercial use. Another fundamental observation is the fact that medical data vary considerably. That is, one hospital can generate data from different sources and formats. Some examples of medical data pointed out by Abougreen [10] are medical images, patient medical records, behavioural data, clinical notes, and genomic data. This variety of data can come up in 3 forms :

Structured: This is the oldest data format known in the field. It is modelled around transactions. By that, we mean any recording of an action/activity that has occurred at some point (payment

transaction, business inquiry, patient registration entry, job application, etc.). Handled by RDBMS in tabular forms and have an extremely rigid structure. They are designed to keep the same format/shape (schema) of every added input. So structured data represent one type of transaction record that repeatedly occurs [8]. It is worth noting that 5% of existing data is structured according to [11].

Semi-structured: Semi-structured data, as its name infers, do not enforce their schema like their structured counterpart. They do not necessarily implement one. A concrete example is XML or JSON formatted data [11-14].

Unstructured: Most newly generated data are unstructured [8]. It could be anything from video/audio recording to image representation. It has no concrete structure and cannot be stored in relational database tables.

Despite the various challenges [15], data help tremendously in preventive medicine (predictive analytics, pattern exploration and detection). It can also be used in treatment monitoring and is essential to study and monitor population health statuses in epidemiology. We may require dedicated infrastructures with high-speed/real-time data processing capabilities based on the gathered data type. Time-series heartbeat data monitoring is a perfect example of such a use case. Effectively managing such tremendous and various amounts of data alongside other challenges resides within the scope of Big Data.

2.3 Big data

2.3.1 3Vs or 15Vs?

We previously identified characteristics unique to Big Data, namely the 3Vs of Big Data; “Volume,” “Velocity,” and “Variety” [16]. Subsequently, more Vs joined the stack. One that has been heavily discussed lately is Veracity. Introduced by IBM, “Veracity” designs the aspect in which a piece of data holds its true value, i.e. veracity reflects the quality of the data but also the degree of uncertainty it holds [2, 10, 17]. The 4V characteristics are predominant when designing Big Data infrastructures. However, other characteristics can also be considered, including “Value,” “Variability,” “Validity,” “Volatility,” “Visualization,” “Virality,” “Viscosity,” “Venue,” “Vocabulary,” “Vagueness,” and “Complexity” (the only one that does not start with a V) [10].

This research focuses on the three main ones ubiquitous to every Big Data project: “Volume,” “Velocity,” and “Variety.” We give the “Variety” characteristic particular attention, given the availability of a wide range of health data. We also acknowledge the “Veracity” for holding an essential status in the medical field, where we avoid abnormalities related to patients and treatment records. This characteristic is, however, tied to metadata management strategies and ontology modelling, which we do not cover in this thesis.

2.3.2 Big Data Platforms

The term Big Data Platform arises whenever there is a need to tackle Big Data challenges. Although we will not make a complete stop on the subject, we will go over the main lines helpful in understanding Big Data architectures since Data Platforms represent abstract topics like architectures. Concrete examples of Data Platforms are the likes of Cloudera Data Platform (CDP), an open-source Data Platform for both public and private cloud providers. CDP leverages the Hadoop ecosystem to satisfy data requirements. Amazon Elastic MapReduce (EMR) is another cloud-based BDP. Amazon EMR also uses a slightly modified version of the Apache Software Foundation Big Data tools in their platform, such as Flink, Kafka, and Spark. Another example is Azure HDInsight, a Big Data package decoupled from the data storage mainly using Hadoop-based or compatible solutions like Hive, Kafka, Spark, and Hadoop MapReduce. According to Gao et al. [18], a BDP can be modelled as a combination of multiple layers, each responsible for a specific role within the platform.

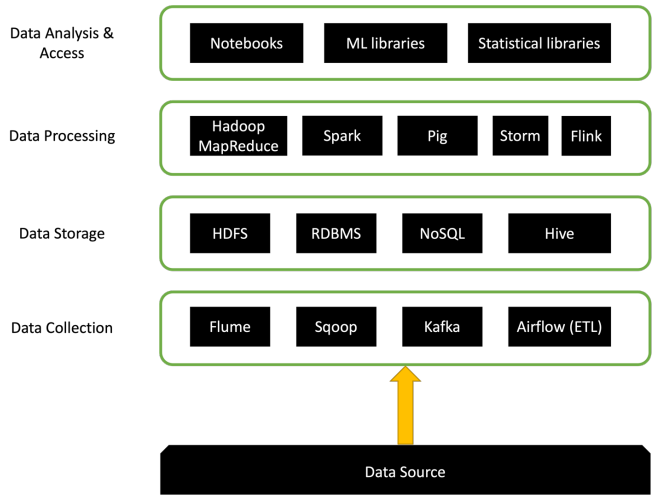

Figure 2.1 Hierarchical View of a Big Data Platform

Our research focuses on three main layers: data collection, data storage, and data processing layer. The other two remaining layers are the analysis and access layers [18]. Since those two layers are not relevant to our research and for the sake of simplicity, we will combine them into one analysis layer. We go over each one of those four layers with a definition and examples of tools that support them :

Collection layer: This layer combines the tools for collecting data from diversified sources. The collection process (the ingestion process) can be pre-planned, delayed, or made in real-time. There is a need to have various tools capable of implementing any of those plausible scenarios. Some of the tools that are used during this process are represented in the following table:

Table 2.1 Collection layer

Tool	**Description**
Kafka [19]	An open-source, distributed streaming platform for stream analytics, data pipeline, and integration.
Sqoop [20]	A tool for transferring bulk data between Hadoop and relational databases.
Flume [21]	A tool for collecting, aggregating, and moving large volumes of data. It is mainly used for log data.

Data storage layer: A Data Platform will have an infrastructure primarily designed for data storage. Because Big Data deals with the Variety and Volume aspect, the storage layer needs tools to store massive amounts of various data formats. This is usually done using horizontal scaling. Hadoop, for instance, uses commodity hardware to store data distributed all over the cluster.

Table 2.2 Data Storage layer

Tool	**Description**
HDFS	Acronyms for Hadoop Distributed Filesystem. This refers to Hadoop's repository file system for storing large files of various formats using commodity hardware [12]. The filesystem allows data replication for fault-tolerant and high-availability scenarios [12].
Hive [22]	A Data Warehousing open-source solution built on top of Hadoop that supports HiveQL, a declarative query language similar to SQL [22].
HBase	A distributed **wide-column** NoSQL database built on top of Hadoop HDFS [12]. It is best used for real-time random access to large-scale datasets [12].
MongoDB	A **document** database from the NoSQL family that relies on JSON-like objects as its data model for better model flexibility and horizontal scaling [23].
Cassandra [24]	Another NoSQL open-source distributed database from the **wide-column** family. It is characterized by its fault tolerance and is highly available using a distributed masterless cluster architecture [24].
CouchDB [25]	A scalable open-source schema-free **document** NoSQL database [25].
Postgesql [26]	An open-source relational database system.

Processing layer: This is part of the backend of any Data Platform. Raw data needs to be transformed and processed for better usability. Hadoop, for instance, uses its MapReduce programming framework to aggregate, clean, and transform large data sets to be efficiently stored or as an entry point to other applications within the ecosystem for additional analysis.

Table 2.3 Data processing layer

Tool	**Description**
Hadoop MapReduce [27]	The programming model used in the Hadoop ecosystem for processing large volumes of data [12]. It consists of a map phase that selects, aggregates and prepares the data for the reduce phase that will apply a specific function to the organized data.
Spark [28]	A robust data processing engine for Big Data. Spark is a concept that differs from Hadoop MapReduce yet shares multiple similarities and can easily be integrated into the Apache Hadoop ecosystem. The highlight of Spark is the in-memory caching when processing large datasets using RDDs and the DAG engine that can process multiple chains of operators as one single job [12, 29, 30].
Storm [31]	A free, scalable, and fault-tolerant open-source real-time computation system that consumes data streams and processes them on the go [31]. It is principally used as a complementary real-time solution to batch processing in Lambda architecture.
Pig [32]	A data processing platform that has its high-level language, Pig Latin. Apache Pig can process data with the Map-Reduce programming framework [32]. It works on top of Hadoop and is manageable by YARN.
Flink [33]	A distributed processing engine for processing streams of data. Besides its essential characteristics (it can be deployed in different ecosystems, vastly scalable), it also offers in-memory data processing for low latency [33].

Analysis layer: This layer represents a collection of tools designed for exploring hidden patterns within the data and bringing up new insights. This includes APIs/libraries, data visualization/representation frameworks, machine learning libraries, and other data science

utilities. The access layer represents the APIs, GUIs, web application notebooks, and other services the end-user uses to analyze and interact with the Data Platform.

Table 2.4 Data analysis layer

Tool	**Description**
Zeppelin [34]	An open-source, multi-purpose notebook for data analytics that integrates with multiple programming languages and libraries.
Jupyter [35]	One of the most popular multi-purpose notebook solutions for Data Analytics.
MLib [36]	Apache Spark's machine learning library.
Tensorflow + Keras [37]	A complete machine learning platform. Mainly used in Deep Learning.
SciKitlearn [38]	Open-source library for machine learning. It provides many utilities such as data preprocessing, model-related applications and more.

To illustrate those concepts in a concrete use case, we take as an example some Big Data Platforms designed for health data. Rouzbeh et al. [39] implemented a BDP solution using open-source technologies that designate the three core layers we previously mentioned. The storage layer uses Hadoop HDFS as a Big Data repository for large and various data. The computation layer uses computing engines such as MapReduce and Spark, NoSQL databases, and the Hive data warehousing solution. The cloud-native service layer (Collection layer) uses notebooks for analytics (e.g., machine learning). The related work section discusses the paper's content in more detail.

2.3.3 Data storage tools and techniques

Each Data Platform must incorporate one or several data storage solutions. Those solutions may vary from relational/NoSQL Database Management Systems to file system repositories. A chosen

Big Data architecture will directly impact the required storage solution. For those reasons, we review the primary storage solutions used in Big Data.

Relational DBMS: This traditional data abstraction model uses tables (also called relations) to express the data stored in a database [40]. Tables have a particular structure that introduces elements, such as the schema representing a fixed database table definition (holding metadata information about the table). Each column in a table represents an attribute, and each row represents a database instance [40]. Using relational databases alone in a Big Data architecture is nearly impossible [41]. But using them alongside other data management systems can sometimes be beneficial, as seen in the use case presented by Winslow et al. [42]. Relational databases are managed through SQL (Structured Query Language).

Kaufmann and Meier, in their book "SQL & NoSQL Databases," [41] defines *NoSQL DBMS* as a variety of majorly open-source non-relational data systems that do not follow a strict and static entity-relation model as in their relational counterpart. The schema concept is either nonexistent or very loose. NoSQL does not use the SQL programming language to query its data [41] and addresses a different set of requirements that are recurrent in Big Data. For instance, projects dealing with large volumes of data require distributed storage infrastructure (high scalability) and use parallel computing that NoSQL databases support. NoSQL also follows a data replication strategy to guarantee high availability and fault-tolerant systems. Such characteristics are almost always sought after in a Big Data project. The consistency, however, is left out to make up for the high availability and partition-tolerant aspects. Most NoSQL databases take after the CAP theorem. They can be defined into four main categories.

Document databases: Like MongoDB, this type of NoSQL family applies a dynamically generated schema to manage data in a flexible format such as YAML or JSON. A document stores its entities as a collection of key-values that can be queried either from the key or the value [43].

Key-value stores: Are considered the simplest form of NoSQL databases and are mainly used to store simple data [44]. Key-value databases do not follow any schema or keep any form of structured data collection as opposed to document databases. Each modelled object must have a unique key within the same namespace. For example, an object employee must have a unique key

representing the entity followed by the items defining the object to store. The values will be mapped to every key.item following the format: emp1.lastname ⟷ <*The_lastname*>.

Wide column stores: This NoSQL database have a tabular structure like relational database tables. But it is designed to store a quasi-limitless number of records in a single row. Those databases are usually implemented in distributed environments, and each column family is stored in the same or the closest node for fast retrievals. They are also optimized for fast reads by grouping columns as a set of column families [41].

Graph databases: Those particular databases store data following an inter-connected graph representation. Records are represented as labelled nodes, each with properties called attributes [45]. Nodes are connected through edges representing their relationships [45]. An example of a famous graph database is Neo4j.

At a lower level, it may be worth mentioning that *file system repositories* are primarily used in Data Lake architectures for designing the "Lake" within. Hadoop has its own file system named HDFS but can support other file repositories like Amazon S3, Azure Blobs, or Google Cloud Storage. Tom White, in his iconic book: "Hadoop the definitive guide," [12] gave the most comprehensive explanation of Hadoop components which HDFS is part of. HDFS differs from conventional local disk filesystems by being distributed, thus heavily relying on networking to orchestrate and manage the partitioned stored data. Storing the data files as distributed blocs across the Hadoop cluster ensures fault tolerance and availability. The blocs are then replicated to guarantee reliability. HDFS manages its blocs by using a namenode (master node) and several datanodes (workers nodes) architecture. The namenode maintains the filesystem namespace and the metadata of all files and directories. The namenode also knows which bloc is stored in a specific datanode. Datanodes retrieve the data blocs and report back to the namenode.

2.4 Different types of Big Data architecture

Three main architectures hold the backend together in a Big Data Platform. The most widespread is the Data Lake system. Other architectures are based on Data Warehouse infrastructures which are still relevant in the industry [2]. And with the emergence of the new Data Lakehouse concept, industries and academia started exploring its applicability.

2.4.1 Architectures based on Data Warehouses

Traditional Data Warehouses have been around for close to 40 years. Despite being designed to handle a fair amount of volume, they were not originally made to solve Big Data challenges. They were primarily exclusive to structured data and had difficulties operating other non-standard formats that increased throughout the years. Warehousing necessitates exceedingly costly vertical scaling from vendors as a solution to keep up with the increasing volume of data. So, an architecture solely relying on classic Data Warehouses will have limited capabilities to solve modern Big Data challenges. Most research papers made the same observation stating Data Warehouses' limitations and difficulties concerning the Big Data issue [16, 46].

Reengineering the infrastructure, however, helped Data Warehouses survive those drastic changes. Since Online Analytical Processing (OLAP) and Business Intelligence (BI) remain an essential and relevant part of business analytics and can only be used to their full extent in a relational data model, the data management industry needed a workaround. In response to those requirements, the Big Data research community had to incorporate the warehousing architecture within other components capable of addressing Big Data challenges. Such solutions emerged with the apparition of ecosystems like Hadoop. Although the latter was designed with a Data Lake architecture in mind, it can be used as part of a pure data warehousing architecture, as demonstrated in [47]. It is possible to implement a Data Warehouse with Hadoop thanks to its data warehousing framework, Hive. In other words, modern data warehousing solutions for Big Data cannot operate without specific ecosystems like Apache Hadoop or any other central data storage repository. Sebaa et al. [47] show a concrete Data Warehouse implementation using Apache Hadoop alongside Hive to process the massive amount of distributed data. They ETL (Extract, Transform, and Load) their data separately on each relational database server before ingesting the result in Hadoop's HDFS system. Hive is then used to structure the data following a warehousing pattern using tables and allowing the usage of queries (HiveQL) which let users generate reports and summaries for analytics and BI. One can argue that modern data warehousing solutions are part of a more significant Data Lake architecture. This is not always true since Data Lakes may contain a warehousing solution depending on their usage.

Likewise, a Data Warehouse within a Hadoop ecosystem may only need the latter to store an already structured ETLed schema-on-write data in a distributed environment. [7] categorizes data architectures by tiers, where a 1-tier architecture represents a traditional Data Warehouse solution, and a 2-tier represents a Data Warehouse within a Data Lake. In this thesis, we do not use standalone Data Warehouses as they offer little support for managing our Big Data requirements, as seen from this section. Moreover, their lack of flexibility undermines our attempt to solve the "Variety" issue with medical data. Therefore, we concentrate on Data Lakes, which offer additional benefits (e.g., ready-to-use architectures for managing data in batch/real-time and better handling of unstructured data) and are better equipped to solve Big Data requirements.

2.4.2 Data Lake-based architectures

Data Lakes are the de-facto architecture used when working with Big Data. The concept emerged following the constraints caused by the dramatic increase of structured, semi-structured, and unstructured data, among many other challenges. It is agreed upon that Data Warehouses alone cannot efficiently process Big Data as they do with standard relational datasets [7]. As previously shown, we may want to use an open-source ecosystem like Hadoop to address many emerging Big Data challenges. Still, we may leverage the ecosystem for what it was initially designed for and use it to implement a solid Data Lake architecture.

One must also note that contrary to Data Warehouses, a lake architecture follows a schema-on-read approach. This means the data workflow process ingests and stores all the data in their raw format inside a distributed repository (i.e., Hadoop HDFS, Amazon S3) and leaves the processing for later. Furthermore, the requirements slightly differ for Data Lakes compared to Data Warehouses, where the former is more adapted to process the data for machine learning and data science end goals.

Rihan et al. [2] categorized Data Lakes by a set of functionalities. In other words, a Data Lake can be (1) a data storage system representing the mean to store distributed raw data across the cluster. (2) an ingestion component responsible for loading data from an external source into a database or file system. (3) a maintenance component responsible for processing the ingested raw data into something more usable for other applications. (4) an exploration component that searches the relatedness of heterogenous data in the lake so users can extract all the needed data for a specific task. Sawadogo et al. [6] introduced Data Lake architectures from a pond and zone perspective.

The pond architecture takes after the idea that a Data Lake contains a set of "ponds" where each one is responsible for storing and processing one specific set of data formats. The architecture usually features a *raw* pond that ingests the data from different sources before transferring them to an *analogical* pond, *application* pond (which can be represented as a Data Warehouse), or *textual* pond. The three ponds may eventually transfer their stored data to the archive pond if deemed valuable for later use.

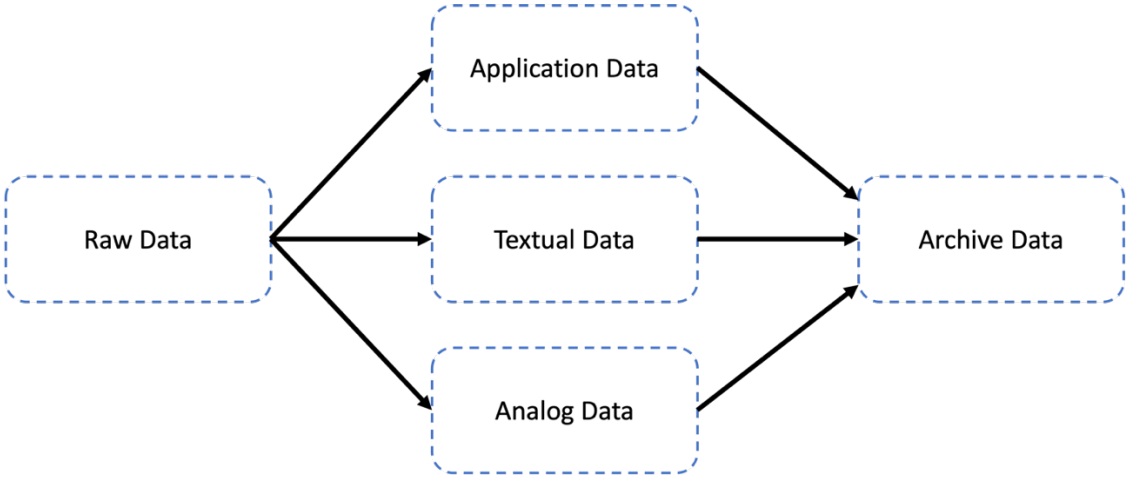

Figure 2.2 Pond architecture

In the zone architecture, the data passes through a subsequent lifecycle instead of subclassing it by format and usage. As an example, Zaloni's Data Lake architecture [48] showcases a transient landing zone mainly for data ingestion, a raw data zone for managing data in its raw format, a trusted data zone for cleaning and validation, a refined data zone for data transformation and preparation before consumption, a discovery and consumption zone that both can be designated as a sandbox for data processing and user data exploration. The zone architecture also proposes a metadata management zone, usually called a governance zone.

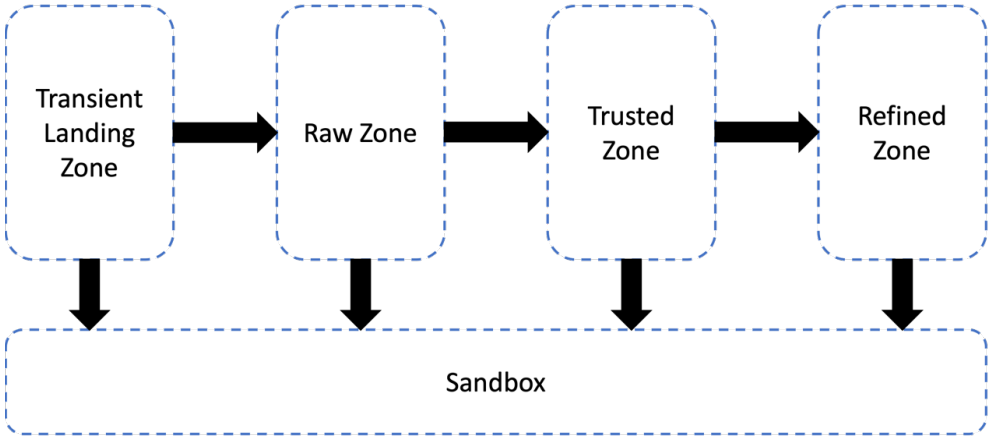

Figure 2.3 Zone architecture

A famous Data Lake architecture known as the Lambda architecture is a variant of a zone architecture [6]. This one is designed to simultaneously handle batch (delayed processing) and real-time data processing. Kappa, another Data Lake architecture similar to Lambda regarding its objectives, uses stream processing as its sole layer to handle real-time and batch data processing through buffers [49]. However, the drawback with Data Lakes is that if not well-maintained, Data Lakes will turn into "swamps" of data. To avoid that, Data Lakes need to have some metadata management capabilities.

Nevertheless, the mentioned benefits of this architecture pushed us to choose the Data Lake for answering our Big Data requirements. They helped us solve the identified "Variety" peculiarity by adding more flexibility to the data model of our architecture. Furthermore, the Lambda architecture proved beneficial for implementing the TDV concept. Another emerging concept named Data Lakehouse may be worth exploring in future works once it has matured enough.

2.4.3 The Data Lakehouse paradigm

Lately, a newly introduced concept is trying to incorporate some of the Warehouse and Data Lake functionalities into one architecture. The first will help Data Platforms provide BI, reports, and OLAP processes. In contrast, the second will provide machine learning and data science support using the data stored in an open-file format in the Lakehouse.

Armbrust et al. [7] argue that their Lakehouse solution solves one specific form of architecture introduced as "two-tiers," which consists of an embedded Data Warehouse inside a Data Lake

system. The paper focuses on ways to lessen the complexity of operations like ETLs for Data Warehouses while supporting the processing of various data formats for machine learning and data science applications. According to [9], the Data Lakehouse uses data virtualization approaches to merge the Data Warehouse with the Data Lake. Therefore, a Lakehouse system is a Data Lake with an integrated Data Warehouse containing explicit metadata management policy, SQL performance optimization, and other relational-based concepts such as ACID…etc.

The Lakehouse example introduced in [7] does not show a novel architectural idea but refactors existing ones and recommends using systems like Delta Lake to address different Big Data challenges. So, in a sense, a Lakehouse system is an organized Data Lake/Data Warehouse system. This means a Data Lake can be a Data Lakehouse if adequately formatted and addressed [7, 50]. In the case of [50], Begoli et al. presented their implementation of a Data Lakehouse system-backed architecture for biomedical research. They built a Lakehouse architecture from their initially pre-planned Data Lake system by storing the data in an open file format (e.g., Parquet [51]), offering support for relational data querying, machine learning, and data science workflow. This was done while ensuring a low cost of ownership and avoiding data staleness. Still, it must be noted that Lakehouse systems represent a new concept. Their implementation, advantages, and disadvantages are yet to be explored and assessed. There is so much uncertainty with the Lakehouse architecture that we decided not to implement it in our solution. The architecture has not yet matured and has unknown weaknesses. More surveys need to be done to uncover the actual value of this novel concept. However, we made decisions similar to Lakehouse implementations from [7] and [50] regarding the open-file format stored in the central repository using binary-encoded files for efficient storage and read performance.

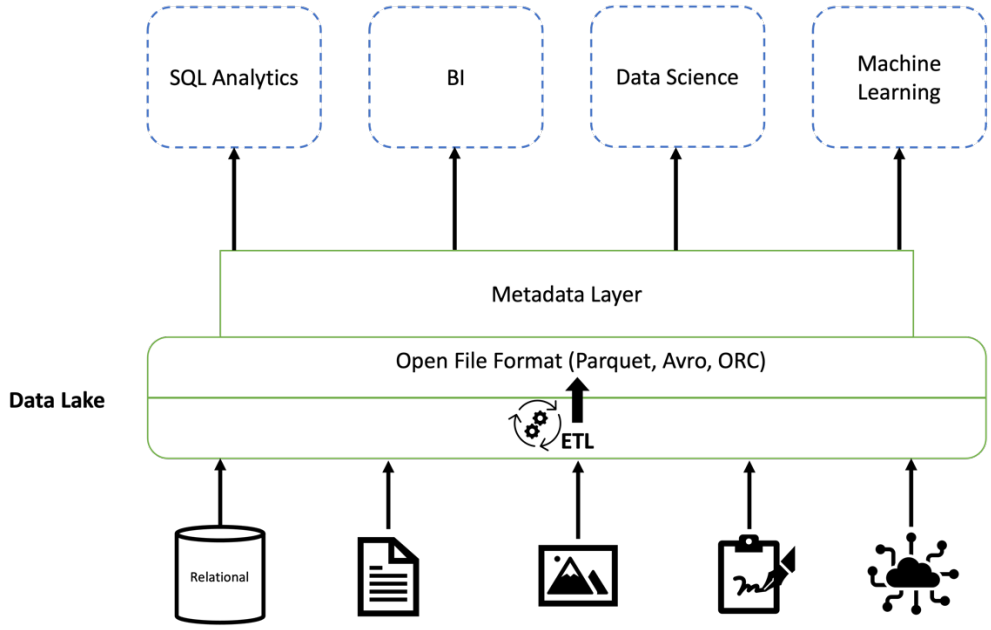

Figure 2.4 Data Lakehouse Architecture

2.5 Related work

We aim to design an adequate backend architecture for Big Data to process, store, and manage health data efficiently. To achieve that, we reviewed different solutions from research papers that explore backend architectures for efficiently storing and processing Big Data in healthcare. We also examined the technologies used to achieve such purposes and how they are used within those architectures. To avoid passing by anything critical related to Data Platforms, storage, and management, we review them from a general perspective and segue through documents focusing on healthcare.

2.5.1 Big Data Platforms: Anatomy and Utilization

2.5.1.1 A General Prospect

To see whether or not Big Data Platforms and the tools that make them are domain-specific, we reviewed one survey by Rouhani and Rotbei [46] on exploring criteria, selecting, and evaluating BDPs for multi-usage purposes. They concluded from their research that requirements primarily direct BDPs' selection choices. Their literature study helped them categorize functional criteria like batch processing, stream processing, data integration, storage, and management. Other non-

functional ones are reliability, scalability, interoperability, and portability. From those, they selected a couple of Data Platforms and applied weights and decision matrices to evaluate and rank them. The explored Data Platforms were Hadoop-based, like Cloudera, Hortonworks, MapR, PivotalHD, and IBM BigInsights. But their study considered a broader range of BDPs and data processing engines such as Spark. Complementary to this work, we explored the possibility of building our BDP using the Hortonworks BDP distribution. However, solutions like those leave little to no freedom in creating innovative solutions as they come in packages with a predefined version for each BD tool. We, therefore, decided to consider other alternatives.

Oussous et al. [16] surveyed the concept of Big Data, its application, its challenges, and some of its widespread platforms. They compared the tools of each BDP layer, highlighted various difficulties organizations might encounter when manipulating massive amounts of data, and described the most widespread BDP solutions (MapR, Hortonworks, Cloudera, HDInsight, Amazon EMR, IBM InfoSphere BigInsights, …). Though the authors identified the most prominent domain fields that benefit from Big Data, they did not point them out as decision factors regarding the tools, techniques, architectures, and Big Data Platform distributions. According to them, the data processing workflow (batch, streaming), performance, scalability, technological compatibility, flexibility, cost, and deployment complexity, among other aspects, are more substantial to consider.

Prasad and Agarwal [52] see Big Data as a combination of storage management and data computing tools and techniques. Following their vision of Big Data, they produced a two-part survey. The first part studied the tools and techniques for computing Big Data. The second part focused on data storage tools and mechanisms. They also compared different tools available in different DBPs and highlighted their pros and cons. They focused on platforms like Hadoop, Cloudera, IBM Netezza, and Apache Giraph for data computing and storage solutions like HBase, Cassandra, Neo4j, and Hive. We used the findings from the two papers (along with [46]) to confirm the applicability of BDPs in different fields before applying them to our healthcare-related project. [52] specifically served as a BD tool catalogue for various BDP implementations we used during prototyping.

Ouafiq et al. [53] used Hadoop to implement a Big Data Platform for smart cities. In their case, they identified three types of generated data; Sensors and IoTs (DGM), human-generated through social media (DHO), and data generated using system administration tools (DMP). They used a

Data Lake with batch and real-time capabilities to ingest, process, store, and analyze the data. Their architecture IISOBA is modelled after pre-determined use-case scenarios. In other words, they know what types of data are expected and how to process them precisely through a pre-defined data workflow. However, the technologies used did not differ from those mentioned earlier. Most were part of the Hadoop ecosystem (MapReduce, Hive, NiFi, and Sqoop). They also used Spark, Kafka, and file serialization tools like Apache Avro [54] and Apache Parquet. We built upon solutions like these that use Hadoop Big Data tools in real-world applications. As concrete examples, they also helped us confirm that there is no difference between other domain BDPs and the healthcare ones regarding tools.

As in most research studies, the Hadoop ecosystem almost always comes out as the building foundation of Big Data science platforms. Liu et al. [55] tested the capabilities of their conceived Hadoop-based Data Platform using 200 compute nodes donated by Yahoo. Their data science platform did not differ much from other widespread Hadoop-based ones. Nevertheless, the authors used it to draw multiple conclusions, notably when using the Hadoop ecosystem in academia and research. They emphasized the lack of flexibility in enterprise-ready distributions of Hadoop-based platforms, like in the case of HDP, CDP, and other cloud-based distributions from AWS, GCP, and Azure. They noted difficulties in implementing automated configuration setups in some instances and frequent delays related to software distributions as some distributions do not frequently update their platforms. The relatively high cost could also be detrimental in specific circumstances. All those difficulties were disadvantageous for the research community, especially those looking to experiment using the latest software features or wanting to introduce different configurations and architectures. Per those findings and following the conclusions from [46], we also confirmed that unrestricted implementations were more suitable for our research since we also have an exploratory aspect for addressing targeted problems, like in the case of the Temporary Data Views.

2.5.1.2 A Healthcare Perspective

Following the same vision, Sebaa et al. [47] elaborates on their medical Data Warehouse using an architecture based on the Hadoop platform. They also designed a conceptual data model that goes along with medical Big Data in a Hadoop-Warehouse implementation. They introduced their approaches to address multiple issues and limitations that traditional Data Warehouse systems

encounter while managing medical Big Data. Their architecture was centred around Hadoop HDFS and Hive. They applied ETL on data from various external sources and stored them in database servers. They then replicated the data across DataNodes in Hadoop. Hive was used as a Data Warehouse structure for reporting, visualizing, and planning tasks. It is worth noting that they used data partitioning and bucketing to address Hive's lack of primary and foreign key concepts. Even though the authors introduced their platform as a Data Warehouse solution, it should be noted that the nature of the implementation shows a 2-tiers architecture like the one presented in [7]. Such architecture is usually made of a Data Warehouse within a Data Lake. Despite proposing an exciting use for Data Warehouses in the medical field. Compared to our work, this solution is still limited to a smaller range of medical applications. It lacks flexibility for treating different data types and does not have mechanisms for handling real-time processing.

Another solution that heavily relies on open-source solutions for building an on-premise data science platform was presented in [39]. The authors gave a hierarchical view of their platform. The storage layer uses HDFS and Ceph [56], an open-source storage system, for storing various data formats. The computation layer uses different sorts of Hadoop technologies for various use cases. Spark and Hadoop MapReduce were used for data processing, Hive for relational data management, and Apache HIPI for image data processing. The service layer represents a Kubernetes cluster for handling and scaling different Jupyter Notebook containers for the end user. The authors also argued that integrating Kubernetes and containers as part of the Data Platform brings many advantages in terms of scalability and ease of management for any application in a distributed cluster. It also helps to make any application reproducible, which is one of the main goals of their research. Like this work, we focus more on open-source BD tools to build our solution. On the other hand, our work elaborates more on containerization as we explore implementing sophisticated BDPs with K8s from the backend side.

McPadden et al. [57] used a Hortonworks Data Platform for building their version of a medical Data Platform. Their objective was to (1) present examples of Hadoop-based (open-source) Data Platforms for academic healthcare systems for planned and real-time situations. (2) Showcase potential use cases that may require such platforms. One important observation was the comparison study between different data storage in the Data Lake. They noticed that binary-encoded snappy-compressed file formats were more cost-effective in storage (up to 80%) and read time than other

data file formats. [7] and [8] also discussed the benefits of storing data in formats like Parquet, Avro, or ORC. Though using BDP like Hortonworks in our research showed limitations regarding flexibility, their snappy-parquet storage strategy in the Data Lake storage repository indicates the same considerable promise in our use case.

While some papers presented their solution for a Data Platform from a general perspective, others completely diverged from the state-of-the-art that is implicitly agreed upon by the research community. For instance, Winslow et al. [42] developed WaveformECG, an open-source Data Platform for managing electrocardiogram (ECG) data for cardiovascular research. The platform aims to store/retrieve, analyze, visualize, and annotate time-series data and their metadata. The architecture is simple, with a storage layer made of PostgreSQL for storing data-specific information such as metadata and the open-source Hadoop-Zookeeper-HBase-OpenTSDB stack to store and manage time-series data. The data processing layer contained libraries for data processing and analysis. The front-end was made using web portals. The general architecture did not show any messaging streaming tool for real-time ingestion and was centred around ingesting the data from special ECG clinical devices. Yet the authors mentioned a connection with I2B2 Clinical Data Warehouse. Contrary to this work, we intend to build a backend for more health data formats, not just ECG data. We noted, however, that the HBase-OpenTSDB combination was a good solution for managing time-series data. Our solution can include OpenTSDB among the many data formats deployment available.

Though their study did not involve Hadoop, Samra et al. [58] followed a similar approach to design Gene2D, a massive genetic disorders data repository for research and analysis. They aim to create a centred NoSQL repository for holding all the genetic data from different relational databases. They build a custom ETL process to transform the data to the required document data format before storing it in the MongoDB repository. Compared to our work, their single NoSQL warehouse implementation lacks the flexibility of using more than a single data model, thus limited to a smaller application range.

Ehwerhemuepha et al. [59] developed HealtheDataLab, a cloud computing data science platform to assist data scientists in building machine learning models in pediatrics. The architecture was mainly hosted on Amazon Web Services (AWS). Their version of the Data Lake uses Amazon S3 to manage various health data. Their solution also included Amazon Elastic MapReduce (EMR),

which gathers several Apache tools such as Hive and Spark. The latter was their computing engine for data processing and analysis. Jupyter Notebook was used for developing and representing their results. Their solution focused mainly on EMR data but claimed to handle other databases containing other medical data sources. The paper proved many vital points that further support our research and architectural design. First, it is beneficial for health researchers to select their desired dataset to study one specific problem for a particular project. Second, cloud computing not only facilitates the first point but also makes possible the development of machine learning models for all sorts of data science exploration tasks. In other words, such a platform can help users leverage Big Data analytics tools and data science to new heights. In this thesis, we build upon these two points within an architecture capable of managing real-time data.

Calderón-Gómez et al. [60] devised another architecture based on cloud microservices they named SPIDEP. Such a platform aims to detect infectious diseases in elderly patients at an early stage via telemonitoring. Their architecture is based entirely on Docker containers and managed by Kubernetes. The platform was developed with PHP (Laravel framework) and Python (Django), and each container had Linux Ubuntu 18.04 LTS. Their solution stored the data in SQL (MariaDB) and NoSQL (Cassandra or MongoDB) databases, where each database was linked to a specific service on the Kubernetes cluster. Though the authors significantly showcased the benefits of using a cloud-native architecture based on different inter-communicated microservices, their paper lacks a concrete example showcasing a more granular architecture with the Big Data tools used and how they interoperate. Our work provides concrete examples of a healthcare-focused BDP with open-source tools with batch and real-time processing capabilities. We also built upon the containerization idea and pushed it toward dynamic storage using TDVs.

Kamran Ghane [61] implemented a 2-tiers Big Data Lake architecture with mechanisms for automatically creating and maintaining its Data Warehouse. Their solution is heavily centred around traditional dimensional Data Warehouses. Still, the authors claim it can process structured, semi-structured, and unstructured data. This makes the whole dynamics of their system committed to managing pipelines, ETLs, and schema preparation. This minimizes the flexibility of Big Data Platforms and undermines the potential of choosing between the available data storage models to address some Big Data storage issues. The solution was built on Kubernetes and Docker, which is not explored enough and holds many advantages compared to conventional designs. The platform

used Apache Kafka for stream messaging and Spark for data processing. Apache Cassandra represented the Data Lake, and Google BigQuery illustrated their version of a Data Warehouse. The architecture parallels our Data Lake architecture for each layer of BDPs. We, however, address the lack of flexibility problem by using dynamically generated NoSQL views through the TDVs within our cloud-native solution based on K8s.

Other researchers like Jovanovic et al. [62] sees the issue of Big Data usability from a data integration workflow. The authors argue that one must inevitably provide a cross-analysis of domain data from various sources to extract valuable insights from the data. This droves them to design Quarry, a user-centred Data Platform that uses hypergraph metadata to address those issues. Their Data Platform was tested on data from the W.H.O. on neglected tropical diseases. Despite stating the benefits and particularities of such BDP, the authors focused on the specificity of their hypergraphs-based metadata management solution and global schema modelling. They did not explicitly show a real-case scenario of their solution working from a broader perspective from ingestion to presentation and analysis. In conclusion, Quarry can complement our architecture regarding workflow modelling through metadata and schema modelling.

2.5.2 Big Data Backend Architectures

Sanla and Numnonda [63] made a comparative study between 2 state-of-the-art Big Data architectures that rely on Data Lakes. Their study aims to help choose between complete Big Data analytic architectures for quickly, accurately, and efficiently handling data in planned schedules (batch) or in real-time. The two explored architectures were the Lambda and Kappa architectures. The results showed that Lambda outperforms Kappa in accuracy but uses slightly more resources (CPU, RAM). The results also showed that Lambda has a higher installation and maintenance cost than Kappa but is less prone to errors. In other words, kappa is fitter for organizations seeking quick results without much precision, whereas Lambda is preferred for high-accuracy results and resilience to errors. Similar to this study, our research uses a Lambda architecture for its higher-accuracy advantage, a crucial aspect of our partners in the medical field. In healthcare, especially in epidemiology, higher accuracy is always preferable.

Gillet et al. [64] developed a system based on the Lambda architecture for processing and analyzing user tweets. They relied on Akka and Kafka for managing the data flow, Hadoop HDFS as a master

dataset along with MapReduce for raw data processing in the batch layer, Kafka Streams for the speed layer, and a polystore of 4 databases (PostgreSQL, Neo4j, HDFS, Cassandra) in the serving layer. They mainly used Scala for different implementations. While, according to the authors, the architecture facilitated the integration of other databases by adding new insertions during the implementation process, their architecture was not dynamic. They had to manually create each data store beforehand when required, unlike our implementation, which generates data stores dynamically and on demand. Their architecture may be fit to add data stores easily, but the conventional infrastructure doesn't ease up that process showing limitations in this specific area.

Another implementation example of the Lambda architecture was made by Munshi and Mohamed [65] for processing Big Data from smart electrical grids. Their implementation was made using different Apache Hadoop-compatible technologies. Apache Flume was used for data collection and distribution to the batch and speed layer. The batch layer used Hadoop HDFS for storing raw, heterogeneous data and computed the batch views using MapReduce. The speed layer processed incoming streams with Spark. Data querying was done using Spark SQL, Hive, and Apache Impala. Alghamdi and Bellaiche [66] also implemented their deep learning models for intrusion detection systems for IoTs using the Lambda architecture. The speed layer was used for binary classification to detect whether there is an incoming anomaly in real-time. The batch layer was used as a multi-classifier for categorizing the nature of the anomaly. The authors used HDFS for storing the ingested IoT data and Spark as a data processing tool. The two papers present a typical implementation of Lambda architecture related to our architecture. Similarly to their implementation, we used the Hadoop-based BD tools in our architecture. The difference is that we offer more considerable scalability with our Just-in-time K8s Cluster.

Besides the two mainly used Big Data Lake architectures, namely Lambda and Kappa, Cassavia and Masciari [67] explored another design named Sigma. Their architecture contains three layers, a data layer that ingests and stores all the immutable raw data from various sources and an engine layer that includes a batch and real-time engine for computing and storing the data in the serving layer for querying. Their implementation aims to decouple the batch from the real-time streaming, thus allowing the execution of the architecture either in batch or in real-time independently from one another. Their batch implementation uses HBase as the source Data Warehouse and Spark as the batch engine to process and store the data in a fixed MongoDB in the serving layer. Their real-

time implementation also writes the data flow in HBase and simultaneously passes it to Apache Storm in the engine layer for processing. The paper, however, did not mention whether the data was transformed and the means to do so. There was no sign of dynamic storage characteristics in the architecture either. Moreover, the architecture requires future studies to conclude its applicability in a production-ready environment. However, the implementation inspired us to explore different variations within the Lambda architecture.

Marinov [68] studied the potential of using HBase and OpenTSDB to store time-series data in Lambda architecture. Time-series data are widely used in electrocardiogram (ECG) records for storing heartbeats. The author highlighted the importance of using a distributed wide-column NoSQL database such as HBase for efficiently storing time-series data. He also stressed how a Lambda architecture helped build a Big Data system for managing massive amounts of data with real-time capabilities. Finally, he gives vital points for storing time-series data in a wide-column database and introduces OpenTSDB. This time-series database sits on top of HBase, which optimizes the storage of that specific kind of data. This helped our research explore support for time-series data using Lambda and further backed our decision to choose it as our BDP architecture for our Data Lake.

2.5.3 Big Data's Backend and Storage Management Strategies

Other papers focused on exploring independent solutions for treating one specific aspect of Big Data storage and processing. As coined by Stonebraker and Çetintemel in 2005 [69], there is no "one size fits all" in the data storage model. This means each data model is and remains as relevant as any other. Comparative studies have demonstrated this in different data storage models, such as [43]. Following this conclusion, new solutions emerged, like the early BigDAWG polystore system [70]. The authors intended to create a system that can handle multiple data model representations in one homogenous interface. The polystore architecture consists of "islands" tied to different storage engines through programmatically defined mappers called shims. The BigDAWG query language then combines SCOPE and CAST operations to select data model boundaries and interpret various data models in one uniform query. While this solution proved efficient, it had many overheads. Its unique implementation added a steep learning curve for anyone wanting to extend it. However, our TDV was designed with a combination of different modern open-source

tools like containers and the Spark data processing engine, each with its specific responsibilities. This lessens the complexity for newcomers to extend the model to a polystore without much hassle.

Zane Bicevskaa and Ivo Oditis [71] studied the possibility of using NoSQL DBMS instead of their relational counterpart in Data Warehouses. Even with the rising market shares of NoSQL databases, they are yet to be seen as a complete Database Management System capable of performing complex data processing tasks. This leaves lots of unexplored potential for using NoSQL in Data Warehouses. For instance, NoSQL has the advantage of managing semi and unstructured data over the relational model, easing the possibility of querying by text search. Another point made was that Agile-style development on Data Warehouses becomes a possibility with a NoSQL implementation, contrary to the design and implementation of relational Data Warehouses, known for being inflexible. They also demonstrated how different NoSQL data models can present data more efficiently. For instance, they used examples to show how one record can contain various data sets. Subsequently, the findings of this paper align with our NoSQL implementation through the TDVs. Taking inspiration from their evaluation strategy, we could also prove the querying advantages in our evaluation use case.

Other studies, such as the one by Eremeev, Ivliev, and Vagin [72], highlighted the impacts of including NoSQL databases in the organization's Big Data architecture, especially in healthcare, where various data formats coexist. The authors used ontology to formalize their data modelling and storage process, thus facilitating data processing with advanced techniques such as machine learning for vision pathologies. This proves that our research, with its NoSQL TDVs, has an affinity with Machine Learning and Data Science applications. Another interesting fact is that NoSQL databases have the potential to work directly at the database layer with machine learning algorithms rather than at the application layer with Python. Celesti et al. [45] implemented and compared two systems for using machine learning to help remote patients via telerehabilitation. Their findings showed that although the two implementations were equally accurate, the system that worked at the database layer with Neo4j performed better than the traditional Python approach at the application layer. Such findings emphasize the potential that NoSQL databases can bring when included in a Big Data architecture. We introduced NoSQL databases in our TDV system within our architecture for similar reasons. This innovative approach extends the usability of our solution for enabling direct DB to Machine Learning processing.

One conclusion we can already draw from most reviewed papers is that health data, by definition, has many properties of Big Data's Vs, as stated by Sen and Mukherjee [73]. This justifies the importance of using NoSQL data stores for storing and managing various heterogeneous data. Albeit the authors' main research objective was associated with ontology for health data, which constitutes a different research subject, they still noted an essential aspect regarding data storage. They mentioned how cloud infrastructures could be profitable for health data but can be detrimental and costly, especially when massive unused datasets are kept indefinitely for data analytics purposes. The cloud-native architecture we propose in this thesis strengthens the mentioned advantages and mitigates the cost-related drawbacks by including NoSQL through the Temporary Data Views.

Vanhove et al. [74] use a data transformation technique to guarantee an interface with dynamic storage and polyglot persistence capabilities on (but not exclusive to) legacy applications. Their solution, however, is more focused on data migration of legacy storage systems, dynamic changeovers between databases, less downtime, and polyglot persistence. Their solution used a canonical model to transform the source data store into a new data model. Their model works on schema transformation first and uses the result to convert the source data model from one database to another. They implemented a Data Lake Lambda architecture to guarantee batch and real-time data processing. Following a similar logic, Abdelhedi et al. [75] introduced a technique for transforming relational databases (contained in a Data Lake) to a NoSQL document-store Data Warehouse (OrientDB) on medical data. The objective was to create a NoSQL Data Warehouse from different relational databases through extraction and unification. The transformation was made through a model-driven approach in 3 steps. (1) NoSQL database creation from the relational model. (2) translation of keys to references. (3) merging of similar classes from different databases. The research, however, was only made for relational database transformation and not all kinds of data in the Data Lake. Another point worth mentioning is that their transformation was limited to document-oriented NoSQL databases, which limits flexibility regarding the various data models available. Our TDV-based implementation differs from the two previous works because we follow a more manageable approach with K8s that does not only translate one data model to another. Their solution might share some common grounds with ours regarding intentions, yet significant differences need to be pointed out. The authors proposed a solution that uses the Lambda

architecture to dynamically provision raw data from either the batch or stream layer to a permanent transformed data persistence storage to query from. It was not explicitly stated that the transformed data storage acted as the serving layer in the Lambda architecture, although that seemed to be the case. Their Polyglot system mechanism uses data migration to make the switch between data models possible. In other words, the migration is done on a permanent transformed data persistence and acts more like a model translation mechanism.

Our Temporary Data View concept integrates into a more significant Data Platform architecture that follows the same on-demand strategy. In contrast, the authors in [75] and even [74], to some extent, present their solution as an independent component. Another detail is the difference between their implementation of polyglot data storage and ours. For instance, we intend to help users choose from different data models to provide temporary views, contrary to the solution from [74] that stores the dataset once in a transformed data storage and keeps migrating the data as needed from one data model to another. We also notice differences in the storage lifecycle of the transformed persistence, where we have designed our temporary views to have a finite life cycle within our just-in-time cluster.

CHAPTER 3 DATA PLATFORM: BACKEND — ARCHITECTURE & IMPLEMENTATION

The previous literature review helped explore three principal points related to our thesis, namely, (1) explore various BDPs and their particularity, and assess the tools they implement and the way they operate; (2) explore the state-of-the-art in Big Data backend architectures; (3) explore techniques to store, manage, and serve Big Data on-demand. The last point relates to the particular question of addressing the flexibility to serve highly varied health data through modernized tools. Throughout the different reviewed papers, we identified a possible solution that brings the answer to storing and serving highly various data within an adequate backend architecture for BDPs using modernized, cost-effective, and highly scalable Big Data infrastructure.

From a high-level perspective, we presented the primary components used to design Big Data architectures during the previous section. We learned that to create a scalable and resilient Big Data Platform, we will likely need to develop an architecture built around Data Lakes since they offer better ways to answer many Big Data challenges. However, the organization's requirements will direct tool choices and the remaining architectural design decisions (security, analytics, …). [76] reported that technology, rather than the requirements, drives many decisions inside Big Data projects. From a software engineering perspective, this inevitably leads to endless issues beyond what we can discuss in this thesis.

In our case, the requirements were centred around predictive analytics, which relies on machine learning and data science, with the possibility to provide Business Intelligence (BI) and reporting when necessary. Since epidemiology could require performing data analysis in real-time, a potential backend design to those requirements could be Kappa or Lambda architecture. We chose Lambda since its batch layer is critical for introducing the Temporary Data View concept (TDV), which we discuss in detail in this chapter. The TDV concept uses modern microservice architecture on an on-demand cluster to dynamically generate tailored datasets for each data exploration project.

To address the three points mentioned above, we organized this chapter accordingly. Section 3.1 discusses and gives an overview of our backend architecture and states the tools used on each part. Section 3.2 focuses on the batch-processing layer of our architecture from a backend perspective and expands on different phases during batching, namely, ingestion, storage, and backend

processing. The same section also briefly presents our prototype solution. Section 3.3 build upon the storage part from section 3.2.4 by introducing the TDV concept and describing its structure and mechanism. Section 3.4 details our prototype BDP and defines its two significant subcomponents. It must be noted, however, that neither our architecture nor the TDVs are bound to any specific technology used in building our prototype.

3.1 Data Platform — Architecture Overview

Following a top-down approach, we intend to clearly describe our chosen backend architecture for an on-demand Data Platform model in this chapter. We start with a high-level description that will lay out the most relevant goals the architecture tries to achieve and the means to do so. We increasingly add more in-depth technical elements and concepts toward a more granular understanding of architectural elements. Among the technical details we aim to introduce is the novel concept of Temporary Data Views (TDV) and the importance of including it in our Lambda architecture which embodies an integral part of our research.

3.1.1 Lambda architecture

As mentioned in Chapter 2, Lambda architecture is a subvariant of the Data Lake zone architecture. It usually comprises three components. A batch, a streaming, and a serving layer. The reason that drove such design is due to the limitation brought up by the CAP theorem [77]. It is irrational to think that all three characteristics (i.e., *Availability*, *Consistency*, and *Partition Tolerance*) would reside in harmony within the same ecosystem [77, 78]. At the same time, every distributed system in a Big Data ecosystem guarantees its *Partition Tolerant* characteristic, which means one will be forced to choose between *Consistency* and *Availability*. One way to include all three aspects is to design a backend with properties associated with the Lambda architecture. Only then can we ensure a highly available distributed system with eventual consistency characteristics. Uncovering how the Lambda architecture defies the CAP theorem will push us further from our main subject. Yet, it is still vital to stress this architecture's important role by breaking down the backend into small pieces following a layered approach, hence overtaking the CAP challenge.

A regular Lambda architecture will either directly ingest data from various sources, which is limited in practice, or use a data messaging tool like Apache Kafka to feed the backend with data

to store in a Data Lake file system repository such as HDFS. The data stored in this situation is part of *batch processing*. Other tools will take charge of processing the massive amount of historical data stored in the lake and transfer the result to the batch view in the serving layer. In the case of real-time generated data, the data will still be stored in the Data Lake, but it will simultaneously be pushed through the streaming layer for further processing. One might use a real-time processing tool like Apache Storm to process and store the newly ingested data in real-time views. It remains widely the responsibility of the developer to decide on ways to query the data from both the batch and real-time views.

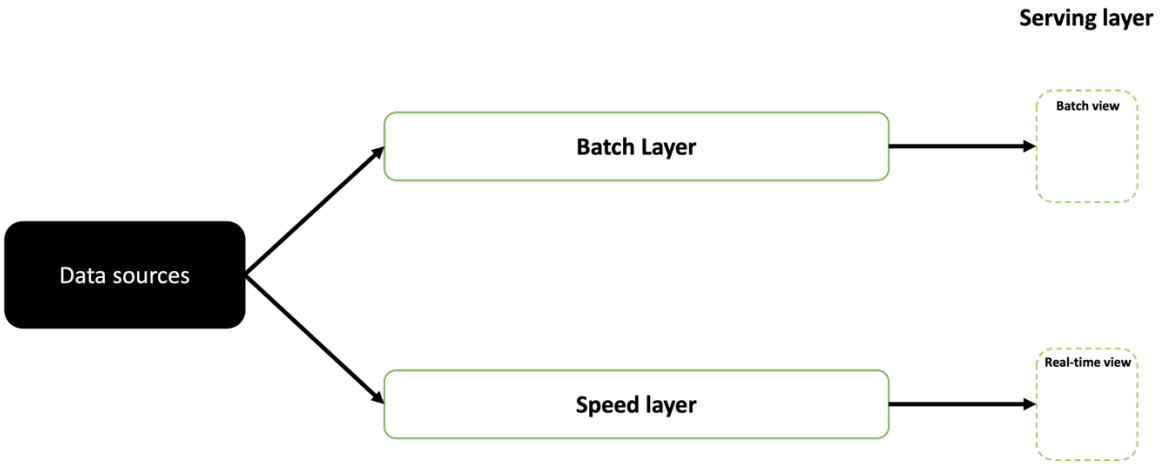

Figure 3.1 Illustration of a Lambda Architecture

3.1.2 Implementation Overview and technology choices

The basic structure of our Data Platform follows a Lambda architecture. One distinction to point out is that the on-demand nature of the platform pushed us to implement most components in Kubernetes. This provides many advantages regarding scalability but also brings up the idea of offering just the necessary resources depending on the situation. Furthermore, the storage mechanism specific to our project, namely the Temporary Data Views, is compatible with both the stateful and stateless nature of Kubernetes.

Using a Lambda architecture implies designing two separate data workflows for batch and real-time processing. The technology tools used for each workflow were chosen accordingly.

Batch processing:

We intend to use batch processing to ingest and store the data in either raw or (preferably) Parquet-encoded format in a Lake storage file system (i.e., HDFS) for later use. The ingestion procedure could be done in 2 possible ways. Our first implementation was done using tools such as Apache Sqoop and Spark. The ingestion, in this case, can be seen as a data migration process where we use Apache Sqoop to import whole relational databases into the Data Lake. Subsequently, we used a Spark script for ingestion as it offers more connectors to relational and non-relational data models. Those methods require developers to manually process the ingestion phase each time new data is generated. In other words, those methods are primarily for data migration workflows. The other ingestion method relies on messaging brokers like Apache Kafka. The difference between the two is that it makes the ingestion processing more automatic. Brokers like Kafka can help ingest the incoming batches of data in a more automated and timely fashion.

Therefore, we also set Kafka in our cluster to continuously feed newly written data to HDFS. Following the batch ingestion process, our architecture uses Spark scripts to import the raw data from the storage file system (HDFS) and create a custom batch view in a temporary database on the cloud with the help of Kubernetes. The data processing phase in the batch workflow is solely done with Spark on Kubernetes. We use Spark to clean, aggregate, and transform the data before sending it to the Temporary Data Views. The data analysis and presentation layers use Spark MLib in Apache Zeppelin and are part of the work presented in [79] by Fahimimoghaddam.

Stream processing:

We add a real-time workflow for the newly added data with the help of Apache Kafka (ingestion) and Apache Storm (data processing). This helps create a real-time view of hourly-updated data. On the other hand, the data analysis and presentation layers use Spark MLib and Zeppelin, as stated in the case of the batch layer. It should be noted, though, that [79] gives more details on the stream processing part of this project.

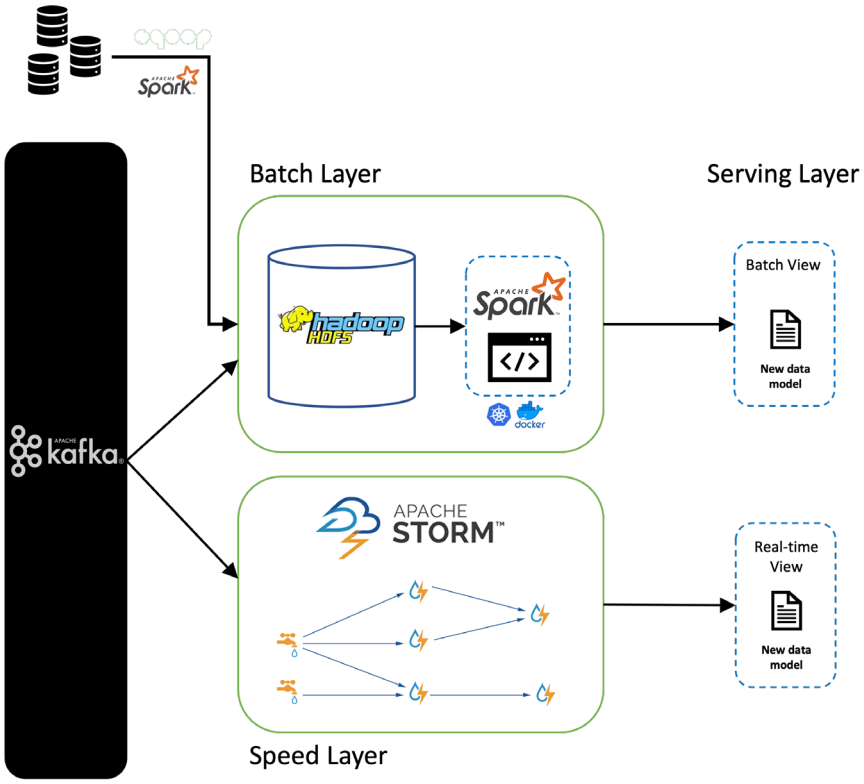

Figure 3.2 High-level view of our Lambda architecture implementation

3.2 Backend Architecture — Our Implementation

This section aims to clarify the architectural elements and tasks involved in each phase of our backend implementation. Thus, giving a more precise and organized picture of what defines this thesis.

3.2.1 Architecture Anatomy

Our Data Platform consists of 2 main clusters. We designate the one related to data ingestion and storage as the *Persistent Cluster* and the one specialized for distributed and heavy processing tasks (data processing, analytics) as the *Just-in-time Cluster* (Cloud native and managed by K8s).

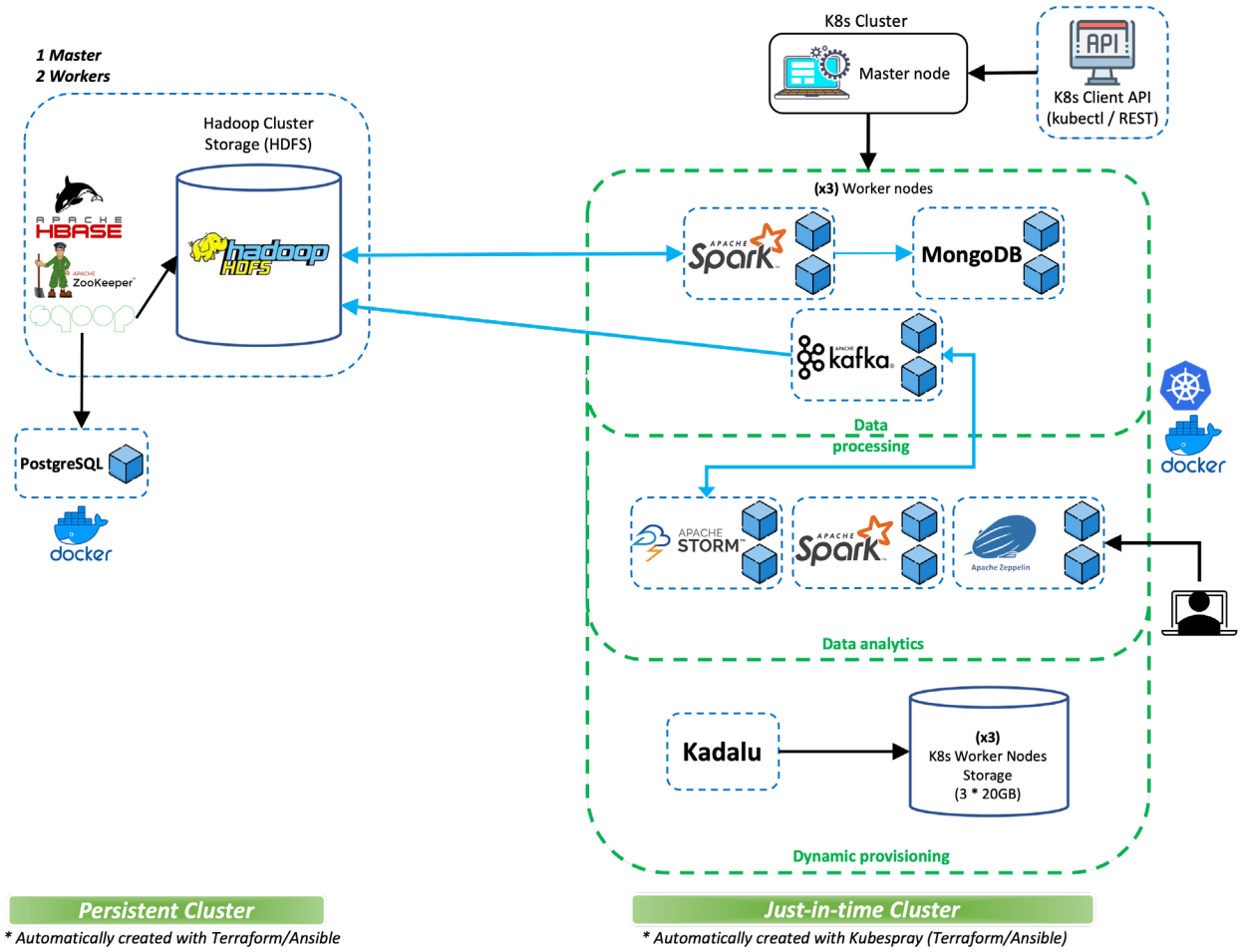

Figure 3.3 General overview of the prototyped Data Platform [80-82]

The ingestion and storage phase for our Lambda architecture's batch and part of the speed processing layers are performed within the *Persistent Cluster*. However, the *Just-in-time Cluster* performs the data processing, transformation, analytics, and temporary storage tasks. This decision is partly because the Temporary Data Views concept in our Data Platform requires highly scalable data processing capabilities. Moreover, cloud providers' cheap scaling cost help achieves that. Also, the temporary nature of the data storage model fits better in the *Just-in-time Cluster*. So, the serving layer of our Lambda architecture resides within the *Just-in-time* cluster.

3.2.2 Batch processing

The batch layer's primary role in a Lambda architecture is to store the ever-growing ingested data in the Data Lake's storage infrastructure, namely HDFS. Cassavia and Masciari [67] noted that the

stored data should be immutable as it facilitates the batch-processing workflow. The other objective of this layer is to perform data cleaning, processing, and modelling of the stored data in the Lake. In other words, the batch layer should be able to provide batch views for any query coming from the analytics component of the platform. We design our Lambda architecture to create batch views across the serving layer differently than the traditional procedure. By keeping our system storage flexible with a schema-on-read strategy, we adopt our novel on-demand Temporary Data View system in our data workflow. In summary, we make Spark the main processing engine during batching and use its native Scala programming language to clean, process, and transform the data and store it in the batch view in the serving layer.

3.2.3 Ingestion

Ingestion in our platform is about extracting and importing raw data from outside sources to the Data Lake. Mathis [49] mentions two ingestion strategies centred around the pushing and pulling of the raw external data. We recognize the importance of the two strategies and lay the groundwork for each one. The pull approach, for instance, is necessary for existing data to be imported to the lake's storage file system, in our case HDFS. Since, in our use case, relational data represents the majority of health data in our possession, Sqoop was an ideal tool for migrating the old relational data model to a Lake system.

Nonetheless, Apache Sqoop is mainly for ingesting internal proprietary data. That is data that was recorded at some point in time by the organization (e.g., health institution). On a side note, it is also possible to efficiently use Apache Spark as a tool for data ingestion. We used the cloud native distribution of Spark in K8s to get the data from external sources and store them in Snappy-Parquet encoded format in HDFS. We must also point out that the platform may have to ingest data from external sources (i.e., ministry of Health, W.H.O., etc.) as soon as they are generated or in a near real-time fashion. This brings up the necessity of a push ingestion approach. For such cases, we must have a data streaming infrastructure that will buffer the incoming data to be later added to HDFS. Since the Lambda architecture has a de-facto streaming layer, it was easier for us to use the same messaging queues (Apache Kafka) for micro-batching newly pushed data to both the batch and speed layers.

Well-regulated metadata help avoid turning the Data Lake into a swamp. For this reason, the ingestion phase should always take into consideration the metadata management of the records when available. Apache Spark and Kafka ease this aspect by providing a custom interface for cleaning the data and collecting its associated metadata while storing it in a more flexible file format (e.g., binary). However, managing metadata is a large and broad subject in the field of Data Lakes. Some methods rely on separate implementations alongside the Data Lake ecosystem and are described in Rihan et al. [2].

3.2.4 Data storage system and format

Our BDP relies on two different storage systems for persistence. The first one represents the simple procedure of storing raw, ingested data in HDFS. It comes, however, with few standards aiming at better managing the data in the lake. For instance, we set the storage format in HDFS to follow a certain binary encoding standard (i.e., Parquet, Avro) and keep the schema along with other metadata whenever possible. The HDFS-stored data must remain immutable to create an infinite number of batch views and remain a resilient data source we can fetch from whenever the TDV is corrupt, deleted, or reloaded.

The other storage procedure is dynamic and on-demand and can be extended to offer a polyglot data model storing system. This dramatically helps define our on-demand Data Platform model. Since the main idea behind our platform is to only keep part of the backend in a stateful permanent persistency (*Persistent Cluster*) and make the rest (analytics and data processing) semi-stateless and scalable on-demand using containers and Kubernetes (*Just-in-time Cluster*), we set our Lambda's serving layer to be part of the *Just-in-time Cluster*. The data model, mainly within the scope of the NoSQL family, in this case, is defined through the data transformation mechanism during the data processing phase.

3.2.5 Processing — Data Transformation

The second storage mechanism is our solution for providing dynamic on-demand storage persistence for the machine learning and data science goals mentioned in the introduction. The dynamic and polyglot data storage model of said mechanism makes it possible for users to query from various data models without being attached to a pre-defined one. To make such a solution

work, we had to move the serving layer away from the persistent cluster to its on-demand counterpart and offer an interface flexible enough for developers to specify their dataset and define their desired data model. This shows the importance of having an intermediary data transformation procedure. Therefore, we designate Apache Spark as an excellent and powerful data processing engine for transitionary data transformation processes. It offers maximum flexibility for developers to choose from the various NoSQL data models available. Another advantage it holds is its Kubernetes cloud-native distribution, a better alternative regarding scalability and resource usage costs. We used the Google Cloud version of the Spark Kubernetes Operator, which takes away much of the overhead of Spark configuration for most developers, only requiring creating Spark Jobs with YAML file manifests.

The transformation mechanism uses Spark-Scala to clean, process, transform, and transfer the data to the serving layer. It remains, however, the developer's responsibility to define the details of each data transformation step. This was necessary to ensure a flexible data transformation system that guarantees polyglot data storage.

3.3 Temporary Data Views (TDVs)

One innovative concept of this project is the temporary data storage provisioning within a just-in-time Data Platform infrastructure. This dynamic provisioning mechanism supports a polyglot data storage model. In other words, developers will be able to dynamically create datasets from Data Lakes that support different sorts of NoSQL data storage models.

Our TDV provides custom storage with different data models to various data users. We do so by breaking down the problem into many small subsets. Instead of having a variety of users querying from the same Big Data source, we create custom temporary, smaller but big enough datasets for each data user profile. In this case, the datasets are custom-made for each situation. Because of their not-so-permanent storage nature, the temporary views fit better in the Kubernetes cluster using databases Operators and CRDs (Custom Resource Definition). The security aspect of our implementation, which, while not a central topic of this thesis, improves by generating smaller subsets of temporarily cloud-stored data. In other words, keeping the whole data in a secure central repository and working only on a dynamically generated subset of temporary data for targeted projects follows a better security strategy.

3.3.1 Containerization of NoSQL Data Views on K8s

The increasing popularity of microservice architectures and the applicability of those in cloud-native environments paved the way for implementing containerized software in a K8s environment [83-85]. It remains, however, uncommon to think of incorporating NoSQL databases as part of the cloud-native infrastructure. The reason for such unwillingness was that, historically, K8s were built for managing stateless applications [83]. So, the most common way to build cloud-native infrastructures was to isolate the database from K8s. After introducing support for managing stateful containers, more complex settings became a possibility. It remains challenging to create and manage a distributed cluster of containerized databases on K8s as they require higher scalability, particular data replication and synchronization capabilities than other containerized apps [83, 84].

Kubernetes is still growing, and its drawbacks are continuously being improved at high speed by both Kubernetes and each database firm introducing its cloud-native version. The research community is also starting to be active in that field. For instance, Perera et al. [84] implemented its built-in operator solution for a highly-available Postgres cloud-native database proving that it is indeed possible to address and enhance all the known K8s drawbacks related to performance, security, and reliability still actively discussed when it comes to database implementation in K8s. Other works like the one made by Lo et al. [85] focused on the different approaches for building and deploying NoSQL databases on K8s. In their case, they implemented and compared two methods for deploying a Docker-containerized HBase cluster on K8s that were based on sharing containers within the same Pod. The two implementations showed differences in their read/write performances. Wu et al. [86] and Kochovski et al. [87] highlighted the containerized NoSQL databases in K8s within their data platforms. The study from [86] used HDFS and HBase in K8s and noticed a slight loss in read/write performance in HDFS but did not notice any significant changes for HBase. They concluded that the loss was more than acceptable. [87], on the other hand, focused on finding optimal ways to deploy databases using QoS metrics. The study used the containerized version of the NoSQL Cassandra wide-column database to decide whether it should be placed on edge, fog, or cloud infrastructure. In this thesis, our focus is more directed toward the architecture side of cloud-native persistency. Our work is not exclusive to storage optimization on K8s. Still, we use the constructor's Operator framework or Helm chart [88] to overcome any

mentioned difficulties. Those solutions guarantee better efficiency since each DB manufacturer constantly updates them.

It must be noted that, in this case, K8s and the database manufacturers' web and GitHub pages give a more up-to-date status regarding the advancement of their databases on K8s. MongoDB, for instance, provides its community [89, 90] and enterprise [91] distribution of K8s Operators, meaning its database can ensure all the needed features for using it on K8s. Other examples of known databases supporting K8s are the likes of Neo4j [92], Cassandra [93, 94], and even distributed SQL databases such as CockroachDB [95], among many others.

3.3.2 Mechanism

Explaining the mechanisms of the Temporary Data View concept is akin to describing part of the platform's data workflow life cycle. From the immutable raw data stored in HDFS, we use Apache Spark to fetch, clean, transform, and store the data in the serving layer (TDV). The robust data processing engine plays a pivotal role in the data transformation and storage process of TDVs. We use the Spark Kubernetes Operator for setting up a Spark cluster capable of natively running its applications on Kubernetes following the same known workload as other apps. Since Spark 2.3, the cluster manager component has started supporting K8s as an official cluster orchestrator. It can be decoupled from YARN, the primary Hadoop resource manager previously used inside Big Data projects. Now, we use the Operator pattern as recommended to add support for managing the Spark life cycle on K8s more consistently. One advantage of using Spark on Kubernetes is the easy setup of the Spark cluster and the possibility of declaratively deploying Spark jobs using YAML files.

We deployed our Spark job with Scala, the programming language native to Spark. This decision is exclusive to this segment of the project. In other words, we use Spark-Scala for the Temporary Data Views generation and management and Pyspark for data analytics. Scala is a better programming language alternative for Big Data aggregation, transformation, and management tasks for better performance, scalability, and security than Python.

Gupta and Kumari [96] compared Scala to Python as a programming language for Spark. Omar and Jumaa [97] performed the same type of study but between Scala and Java. Both papers concluded that Scala has the upper hand in terms of performance. [96], however, pointed out other

characteristics that make Scala a better choice for data processing jobs. For example, the static Scala type safety is preferable for Big Data processing, facilitating compilation-time errors and code testing. Another essential aspect they highlighted was the possibility of using multi-threading in Scala (useful for Big Data processing), unlike in Python. Therefore, in this thesis, we adopted Scala rather than Python.

Performance-wise, using Scala for the TDVs is more relevant. Since Scala is statically typed (the compiler knows each variable/expression at runtime), it compiles faster at runtime, unlike the dynamically typed characteristic of Python, which can be error-prone and requires more resources for the compiler to run. We can also take advantage of multi-threading in Spark-Scala to boost its performance. Scalability is another characteristic that gives an advantage to the Scala language over Python, especially for bigger-scale projects. One last aspect that pushes us to adopt Scala is the security aspect. Statically typed Scala limits variables of being re-written, which gives less opportunity for bugs and application vulnerabilities to occur.

We ran our Temporary Data View process on an HIV dataset of 35000+ patients initially stored in a PostgreSQL database. The dataset has four tables described as follows:

Table 3.1 Description of the HIV database tables

Table name	**Records**	**Description**
Patient	35670	describes the patients of the HIV database
Allergy	23864	gives a full description of the allergies of the said patients
Medication	538015	Prescribed medication for the patient of the HIV database
Alert	14819	states every alert related to the patient

With the help of Spark, we aggregate the four tables into a newly created object, *Patient*, which restructures its object format by adding new nested attributes. The Spark job will also write the new Patient persistence in a MongoDB cloud database in Kubernetes. That temporary storage will then serve users' queries from their notebooks. For better clarity, we present chunks of the Scala

code used in the demonstration (also available in the project's GitHub repository [98]). We also demonstrate each step of the procedure, from the reception of the backend creation request to the establishment of the Temporary Data View in MongoDB (K8s).

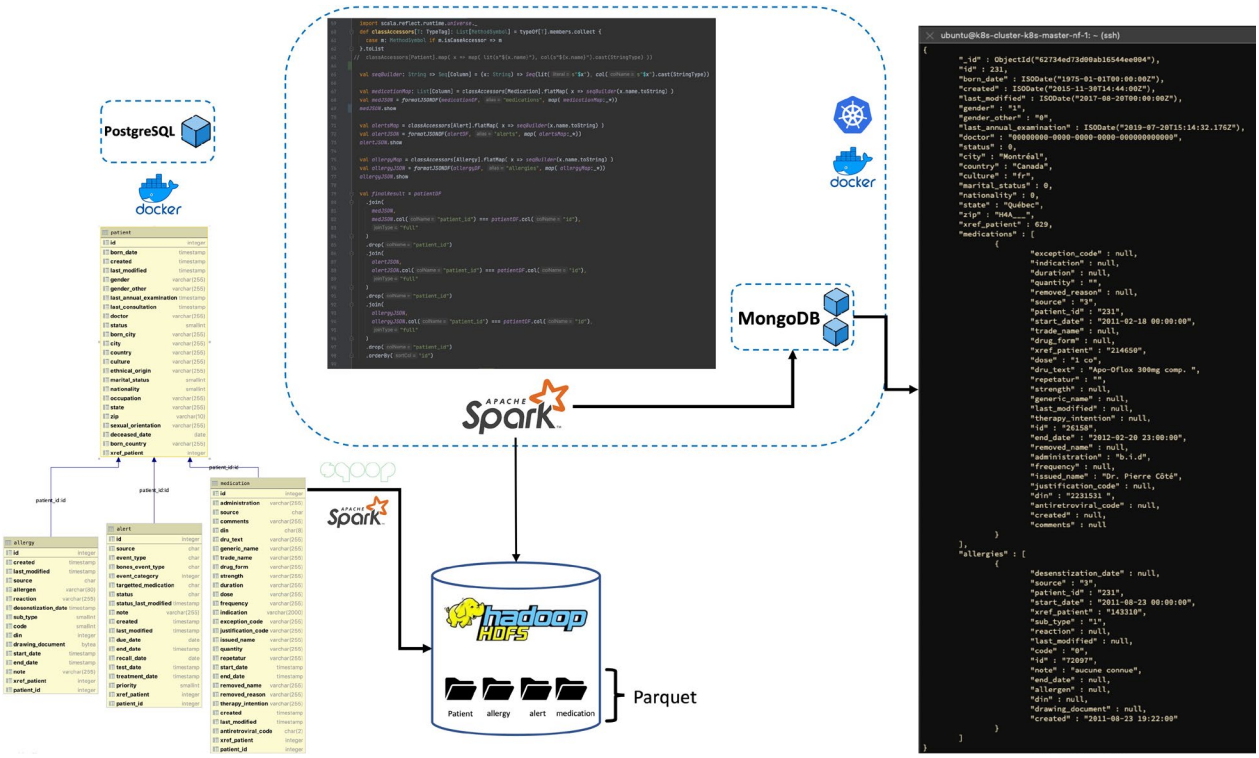

Figure 3.4 Mechanism of the Temporary Data View workflow [82]

The Scala script reads the four tables stored in Parquet (binary-encoded data) from HDFS in the Persistent cluster and stores them as *DataFrame* objects in the Spark memory. The script also performs data cleaning beforehand, mapping the three tables with the patient following its unique id. One thing specific to the Scala language is its compactness compared to other languages like Java or Python. For example, the code fragment from Figure 3.5 defines a function aggregating all medications for a specific patient and storing it in a key-value store in JSON format. The Scala script uses SBT and Maven for building the Spark application to be later deployed by the Spark on Kubernetes Operator.

```scala
import org.apache.spark.sql.functions._
def formatJSONDF(dataFrame: DataFrame, alias: String, cols: Column): DataFrame =
  dataFrame.groupBy( cols = 'patient_id)
    .agg(
      collect_list(
        cols
      )
        .as( alias = s"$alias")
    )

import scala.reflect.runtime.universe._
def classAccessors[T: TypeTag]: List[MethodSymbol] = typeOf[T].members.collect {
  case m: MethodSymbol if m.isCaseAccessor => m
}.toList

val seqBuilder: String => Seq[Column] = (x: String) => Seq(lit( literal = s"$x"), col( colName = s"$x").cast(StringType))

val medicationMap: List[Column] = classAccessors[Medication].flatMap( x => seqBuilder(x.name.toString) )
val medJSON = formatJSONDF(medicationDF,  alias = "medications", map( medicationMap:_*))
medJSON.show
```

Figure 3.5 Spark-Scala data transformation script

The latest transition of Kubernetes applications from their initial stateless-driven design to gradually support stateful features with the help of *StatefulSets* empowers temporary persistence to present new potentials for implementing cloud-native database storage infrastructures. Contrary to other architectures, we moved the serving layer to the *Just-in-time* cluster managed by Kubernetes. This tremendously eases the building of custom storage with different data models, which is crucial to the dynamic and polyglot aspects of the Temporary Data Views.

3.3.3 Use-case example

For better clarity, we give a comprehensive yet straightforward workflow of a possible application of the TDV. In our case, we included it in our data pipeline following a pull ingestion approach. We assume all preparations, like Spark script implementation and modelling, were done beforehand.

1. We start the TDV launch script (available on GitHub)
2. The script creates a Jar application from the Spark script and then sends it to our cluster.
3. The script declaratively deploys a Spark job from the YAML file using *kubectl*.
4. Spark-on-K8s sends a new spark-submit command to create a Job to run the Jar app.
5. Spark-on-K8s processes the data and stores it on a NoSQL database on K8s.

At the end of the procedure, users generate a temporary persistence of the selected data from the Data Lake that they can query using web-based notebooks or any other application that can connect to a database.

The demonstration in this section is a simple manipulation to serve as a proof of concept. We leave it to the developer's discretion to include more granularity and automation in their pipeline commands. Using a bash script to process and demonstrate the TDV steps was enough for our demonstration, but it can be encapsulated in other automation tools like Ansible.

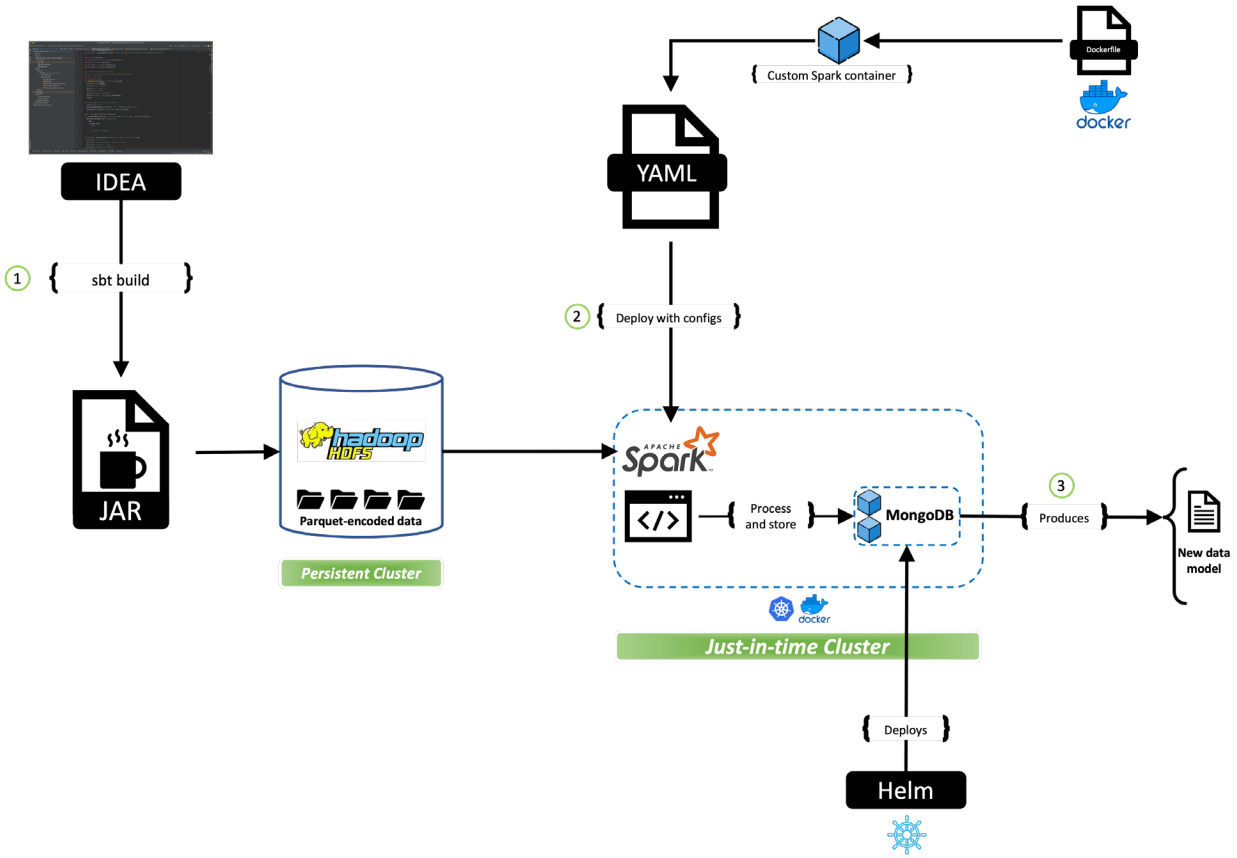

Figure 3.6 TDV example workflow [82]

3.4 Big Data Platform Prototype Description

We apply the three explored subjects of our study, namely the backend, storage, and TDVs, within a prototyped BDP. This example of a prototype used for the Big Data Lake setup and the TDVs is organized as follows.

3.4.1 The Persistent Cluster

The persistent cluster uses Hadoop as its primary ecosystem. As a mainstream solution that has been around for the last decades, Hadoop represents an affordable open-source Big Data ecosystem which can be set up using commodity hardware [12]. We use Hadoop in the persistent cluster mainly for its HDFS, representing an excellent solution for storing structured, semi-structured, and unstructured data in our Data Lake. The Hadoop cluster is managed by its resource manager, YARN, and uses Apache Zookeeper to synchronize every installed application across the cluster nodes. Apart from establishing the tools necessary for a working Hadoop cluster, we also provided the persistent cluster with data management components such as the relational data migration tool Apache Sqoop.

Setting up Data Platforms requires several steps with multi-layered complexity for each ecosystem component. To ease the creation process, we create such a cluster with the help of automatic provisioning and configuration management tools hosted in the OpenStack cloud resource provider. With the help of Terraform, we make the necessary resources infrastructure for the cluster consisting of one master and two worker nodes of similar specs.

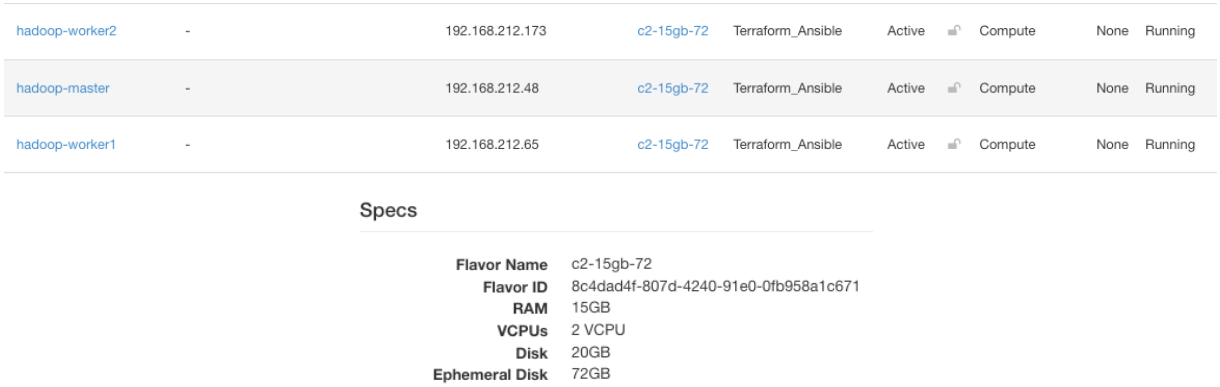

Figure 3.7 Specs of the BDP prototype of the just-in-time cluster

After the initial setup of the persistent cluster, we run our Ansible script to provision the necessary Hadoop ecosystem with its sub-components, along with initial user roles, permissions, and inter-node SSH communication protocol for both the master and worker nodes. We also successfully used Ansible to test the feasibility of automating the Sqoop-ingestion procedure.

As it seems unlikely that Hadoop HDFS would be selected in production unless the configuration is on-premise, other equivalent solutions exist by different cloud providers, like Amazon S3 and Azure Blob storage. We do not impose HDFS or other technology for managing Data Lakes central repositories. The takeaway is to use a solution that decouples the storage from the data processing tools.

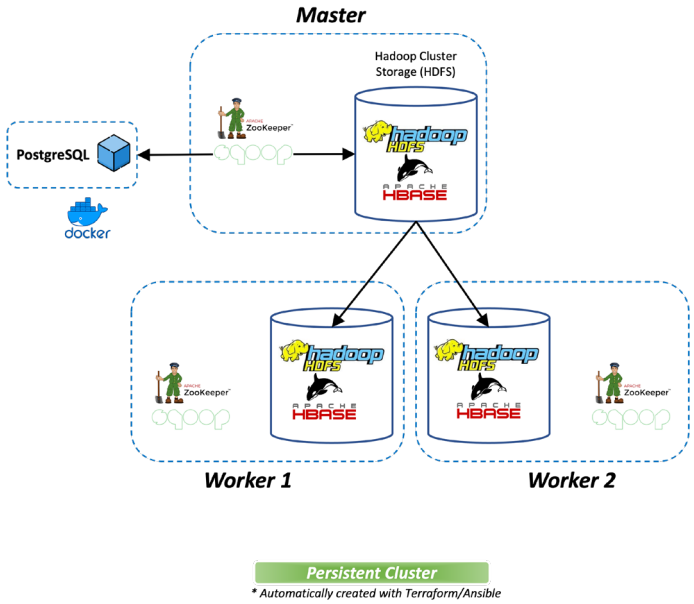

Figure 3.8 Persistent Cluster [82]

3.4.2 The Just-in-Time Cluster

The *Just-in-time Cluster* prototype was created with another automation solution called Kubespray [99]. According to the official definition on their website, Kubespray is a mix of Ansible playbooks and Terraform configuration files, among many others, that can be used to create and manage a production-ready Kubernetes cluster. Its primary purpose is to help ease the overhead of creating a production-ready Kubernetes distribution with complete networking protocols. Kubespray supports AWS, Azure, GCP, OpenStack and bare metal servers and is ideal for building our K8s-managed cluster on OpenStack. To make up for the lack of in-house components such as dynamic provisioning (available in every other commercial cloud provider), we used Kadalu. This open-source tool uses GlusterFS for dynamically provisioning PVCs to K8s applications. We use *kubectl* as the primary client API for setting up, running, and modifying K8s applications. The *Just-in-time*

Cluster comprises a K8s cluster of three workers and one master node. We use the cluster for multi-tenant and heavy-processing tasks involving data analytics.

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State
k8s-cluster-k8s-node-nf-3	-	192.168.212.193	c4-15gb-144	kubernetes-k8s-cluster	Active	Compute	None	Running
k8s-cluster-k8s-node-nf-2	-	192.168.212.213	c4-15gb-144	kubernetes-k8s-cluster	Active	Compute	None	Running
k8s-cluster-k8s-node-nf-1	-	192.168.212.247	c4-15gb-144	kubernetes-k8s-cluster	Active	Compute	None	Running
k8s-cluster-k8s-master-nf-1	-	192.168.212.149	c8-30gb-288	kubernetes-k8s-cluster	Active	Compute	None	Running
k8s-cluster-bastion-1	-	192.168.212.222, 206.12.99.99	c1-7.5gb-36	kubernetes-k8s-cluster	Active	Compute	None	Running

Figure 3.9 Specs of the BDP prototype of the just-in-time cluster

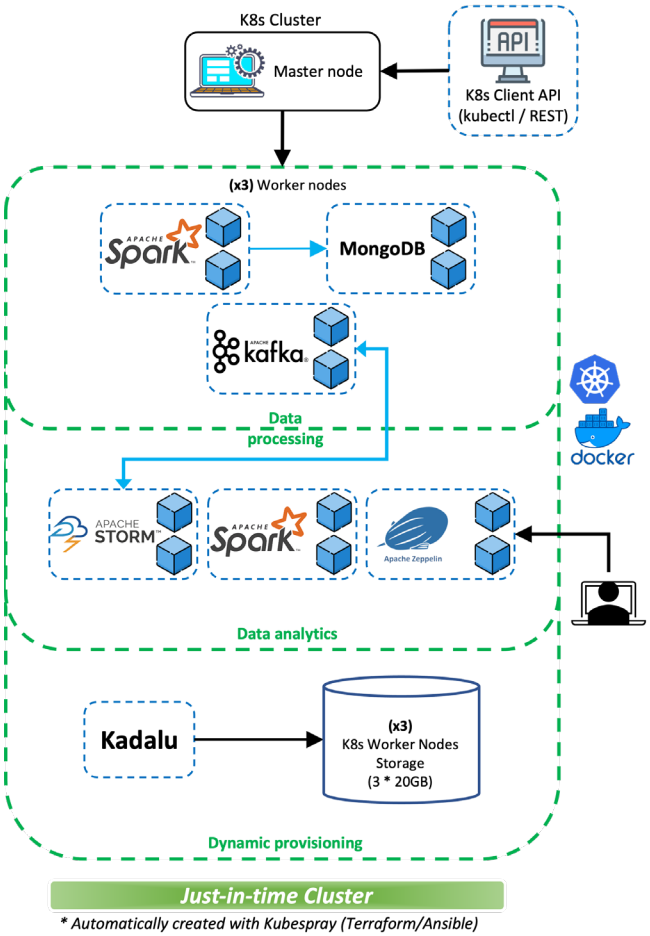

Figure 3.10 Just-in-time Cluster [80-82]

In the scope of our TDVs, the Spark, Kafka, and MongoDB components are the most relevant within the Just-in-time Cluster. The latter represents the serving layer of our Lambda architecture. As mentioned, the Spark component imports the data from HDFS and makes the necessary transformational changes before transferring the new data model to the temporary database. Notice that MongoDB is only the database we've chosen for this prototype test run. The Temporary Data Views are built with polyglot data storage in mind, so we can select any data storage that can be containerized and managed by K8s. K8s has indeed opened many possibilities in this regard. With the constant progress made in K8s about stateful and permanent storage solutions, notably after introducing concepts such as StatefulSets, and CRDs that make possible the extension of K8s applications with the Operator framework possible, it is now possible to create different containerized data storage models quickly and dynamically.

Following the user's desire to end or postpone their dataset exploration, we recommend creating data snapshots for later use. Like during its creation phase, we propose using Apache Spark to store each project's Temporary Data Views records in HDFS. Another alternative would be to use indexation and metadata to help keep the dataset in HBase's Data Lake and use it to recreate the temporary view on-demand. This is a subject we intend to address in future works.

In summary, in our case, the Just-in-time Cluster prototype facilitates provisioning isolated container applications on-demand. For instance, the NoSQL database available on K8s can be deployed with few Helm commands in a reasonable amount of time. That can also be automated using any designated tool for the task (bash, Ansible).

CHAPTER 4 ARCHITECTURE EVALUATION

Our research aims to help data scientists (or anyone capable of running a web-based notebook for data science exploration) create their datasets at the storage level instead of the conventional setting at the application layer. Creating a custom dataset is not only related to the chosen data from the lake. We can imagine a use case with massive heterogeneous data from multiple sources to be ingested in a central data repository. We have seen throughout this study that health data is highly diverse. With that in mind, how can a data scientist know in advance and pick the needed data for one specific data exploration project?

Furthermore, how can we prepare, rearrange, and adequately store the chosen data? By "adequate manner," we mean preparing, aggregating, and cleaning the selected data in a dynamic, cost-effective, and acceptable time range. This is also associated with storing any dataset in its most suitable data storage model to query from. Our architecture is a combination of tailored solutions that answers those use cases. Following the same evaluation method as in [47] and [71], we propose three separate evaluation plans to demonstrate the flexibility, usability, and innovative aspect of our architecture. **The first** evaluation plan compares our architecture with a similar one done with conventional means. **The second** study compares the extensibility of our architecture by showing the polystore nature of our TDVs and the capability of providing any data storage model from (but not limited to) the NoSQL family. **The third** shows the feasibility of dynamically implementing a new data model on demand by comparing it to a conventional implementation.

4.1 Dynamic storage capabilities of the TDV – 1st scenario

In this first comparison study, we compared our dynamic procedure with the conventional one. By convention, we define methods potentially used in the industry or multiple papers from the related works section that do not generate on-demand data stores as in [64]. We demonstrate our novel dynamically generated TDV process with the help of a use-case scenario that we unwrap in the following steps:

4.1.1 Ingest Data from Sources (D. Ingest. 1)

Data ingestion is closely dependent on the running data workflow. The organization may want to import reasonably voluminous legacy data from outside sources. Another ingestion scenario may

want to proceed with delayed ingestion in batches at planned times. Real-time data ingestion follows a slightly different procedure. Our architecture considered those scenarios and used different approaches to handle them. We use Apache Sqoop to ingest legacy data from relational databases into HDFS using a pre-defined hierarchical namespace in raw CSV data format. We later used Apache Spark for the same use case and noticed significant usability, flexibility, and extensibility improvements. Spark gives more data format options to choose from before ingesting the latter in HDFS. We ended up storing the ingested data in the binary-encoded Snappy-Parquet. As Price demonstrated in [57], this tremendously improves the storage space (up to 80% more cost-effective in terms of storage and read time compared to other data file formats) and makes the data prone to corruption errors. Our HIV ingested data shows a 90.36% size reduction of 132.8 MB ingested CSV compared to 12.8 MB using the Snappy-compressed Parquet format. The ingestion script with Snappy-Parquet is available in the work repository at [100].

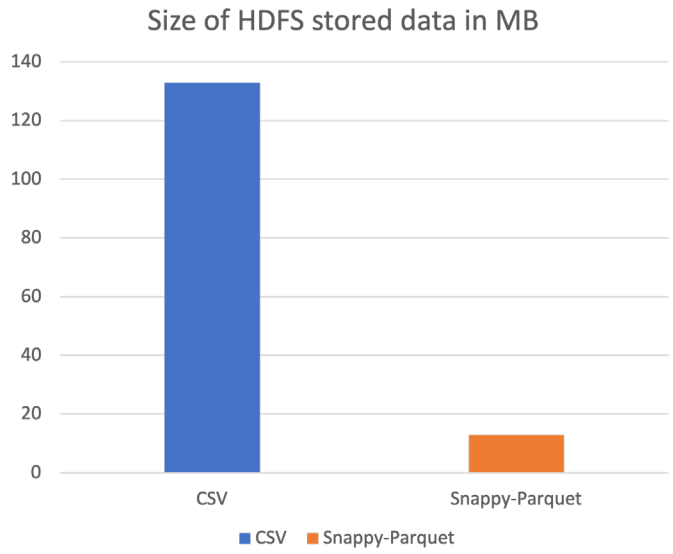

Figure 4.1 Comparison of HIV data stored in two data formats in the Data Lake

Apache Kafka manages the different ingestion scenarios (batch and real-time). Both strategies will ingest the data into the central repository (HDFS). The data ingested in the batch layer will remain in HDFS until a TDV request is called. The stream layer will also forward the data from Kafka to Apache Storm for further analysis.

4.1.2 Prepare the TDV for Receiving the Transformed Data (TDV Prep. 1)

Upon receiving a notification to set up a TDV, the Kubernetes cluster will create the operator responsible for storing the necessary views. The novelty is that we use a microservice architecture to prepare a temporary storage persistence. This uncovers new potentials with the advent of containers and Kubernetes. We used a two-nodes cluster to prepare the NoSQL document database MongoDB to set up the TDV. The advantages of using this sort of strategy let us leave the data model decision at a later stage which adds more flexibility in choosing the data model that is best fit for each data science project. We created our TDV structure using the Kubernetes Operator framework. A Helm chart is also viable when the database manufacturer provides one or if the backend team invests in creating one. Both methods are equivalent; the only difference is that a Helm chart help encapsulates all the *kubectl* commands in one chart.

4.1.3 Process the Data Through Spark-K8s and create TDV (Process. 1)

Once the TDV structure is ready, we declaratively (through YAML files) run the Spark on Kubernetes Operator to schedule a Spark job. Our Scala script simulates a data workflow procedure that reads the Parquet files from HDFS, cleans, sorts out the needed data, and aggregates the data using the *DataFrame* Spark object. Lastly, we transform the *DataFrame* object to JSON format before storing the result in MongoDB TDV through the Spark-MongoDB connector. The script for transforming and storing the data is available at [101].

4.1.4 Write the result on the chosen NoSQL TDV (Stor. 1)

Storing the resulting transformed data to the NoSQL TDV is dictated by the NoSQL-provided connector. In the case of MongoDB, we used the API's library to store the modelled object in a MongoDB-compatible format and used the function to append the data in the TDV.

4.2 Conventional storage method – 2nd scenario

In the case of a conventional use-case scenario. Things will unwrap more manually with less scalability and portability. A typical scenario is akin to the simulation we made with the help of our Hadoop cluster. This simulation will showcase the differences between our novel approach and

the conventional one discussed in the next section. We included the data transformation script in the work repository [98].

4.2.1 Prepare Every Necessary Tool Beforehand (Prep. 2)

For this use case, we had to preplan the exact data workflow for the simulation, preparing the necessary tools for ingesting, transforming, aggregating, and storing the data, even if some tools will only be used temporarily. We installed Sqoop on our Hadoop cluster for ingesting legacy data from an old PostgreSQL database. Sqoop helped transfer the data to HDFS in CSV semicolon-separated format. We then installed Apache Pig for data cleaning, aggregation, and transformation. One could also use Java apps to manually set up a Map and Reduce phase. However, this method is excessively complex and time-consuming for developers, even more for this evaluation. We also installed the required NoSQL database for storing the TDV. In our case, we set up a standalone MongoDB on one of the two nodes of the *Persistent Cluster*.

4.2.2 Ingest Data from Sources (D. Ingest. 2)

The ingestion in the conventional scenario does not differ from its predecessor in the first scenario. The only difference is that we used Sqoop to ingest the data since the data processing engine, in this case, does not use Spark but Hadoop MapReduce through Pig.

4.2.3 Run the transformation, cleaning, and aggregation process with Pig (Process. 2)

With the help of Apache Pig, we wrote a Pig Latin script that reads the data from HDFS, trims the necessary data to include in the transformation, then models each data entity as an object to write to MongoDB. Apache Pig relies on the Hadoop MapReduce programming framework to run its script but lessens the complexity of using the framework directly on a Java application. Pig adds a layer on top of Java and introduces its own language, Pig Latin, which takes care of many programming abstractions for developers.

4.2.4 Write the result on the chosen NoSQL storage (Stor. 2)

Since we installed MongoDB in advance in our cluster, we wrote the temporary view on it through Pig. This one has a connector for MongoDB, albeit not as flexible as the one available for Spark.

4.3 Comparative study discussion

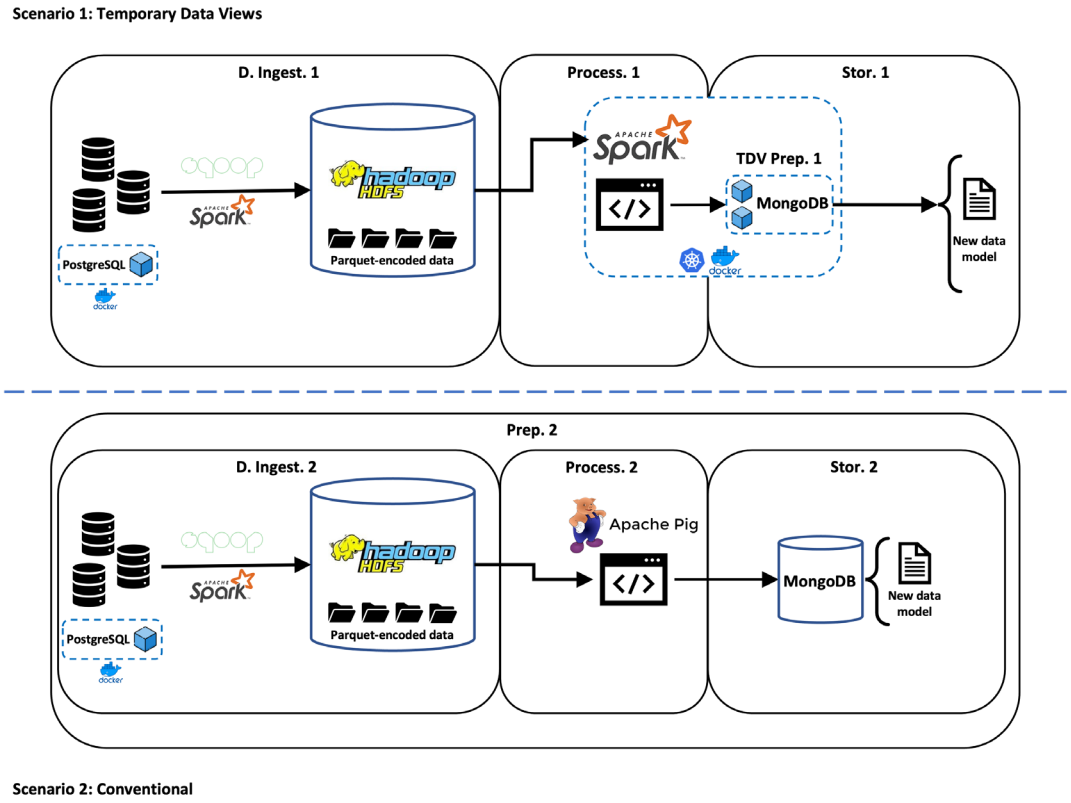

Figure 4.2 Illustration of both scenarios with steps [82]

Even though both scenarios seem to provide the same result on the surface, the conventional method presents several limitations. First, the traditional approach is tightly coupled to the overall cluster. There is no separation between the data repository and the data processing tools. This comes up with consequences. The cleanup, for instance, is more complex in this situation. We must manually uninstall the TDV database cluster (in this case, MongoDB). Not only that, but it's not as scalable and cost-effective as in the first scenario with K8s. Our architecture is more scalable since we can declaratively add or reduce resources with better components isolation on K8s. Another characteristic of our solution is that it is cost-effective since we only temporarily use the

storage and data processing resources without affecting the storage cluster. Our lightweight solution does not rely on heavy virtualization, as in virtual machine configurations.

Scenario 1 is, in fact, better designed for gracefully shutting down and cleaning the TDVs when the data exploration project has ended or is being suspended. Erasing StatefulSets is more straightforward than manually cleaning a dispersed temporary view on a traditional database cluster. Lastly, we must point out that potential harm to the security and privacy of health data is narrowed down with scenario 1. We only use a small subset of the targeted data for the exploration. We don't give complete and direct access to the Data Lake as in scenario 2.

Also, the overall performance, though not efficiently and accurately quantifiable, shows better results in scenario 1. This result may improve even more. As demonstrated in [74], tools like Spark require lots of tunings and optimizations depending on the data size to process. But it is also challenging to measure the performance of manual manipulations without a group of targeted healthcare experts that are supposed to work on this BDP. We have yet to encounter such a group and are looking to perform those studies when our partners eventually implement the BDP on their chosen cloud infrastructure. Another difference is that our architecture uses Pods (or containers) to isolate TDVs from the *Persistent Cluster*, ensuring a loose coupling between the central storage repository and the backend processing module. This will ensure that our TDV remains independent from any configuration matrix, thus guaranteeing better compatibility and portability. That last aspect also makes our solution applicable in any environment (bare-metal or private/public cloud provider), which helps avoid many vendors' lock-in scenarios.

The usability of our architecture eases up the setup. It avoids many overheads for installing and configuring different data processing tools and NoSQL TDVs by automating many configuration and deployment steps. This introduces a soft learning curve for those whose sole purpose is to install and use the Big Data tools. Our architecture can benefit from the built-in monitoring and self-healing features of K8s and its fault-tolerant and resource management to provide better reliability. Scenario 1 also shows better maintainability since each component of the TDVs on K8s can harmlessly be upgraded or modified without consequences.

4.4 Extensibility of the TDV architecture

To further demonstrate the extensibility of our architecture, we perform another simulation built on top of the one from the 1st scenario of section 4.1 that shows how we can easily create a new data model besides an old one to query from. This scenario showcases the variety of data model representations our architecture can provide compared to a traditional relational Data Warehouse system. This also shows the possibility of building simple Polystore systems using our TDV implementation by evading the implementation overheads from complex systems built from the ground up, as in the case of the BigDAWG Polystore system [70].

4.4.1 Multiple NoSQL data format representation

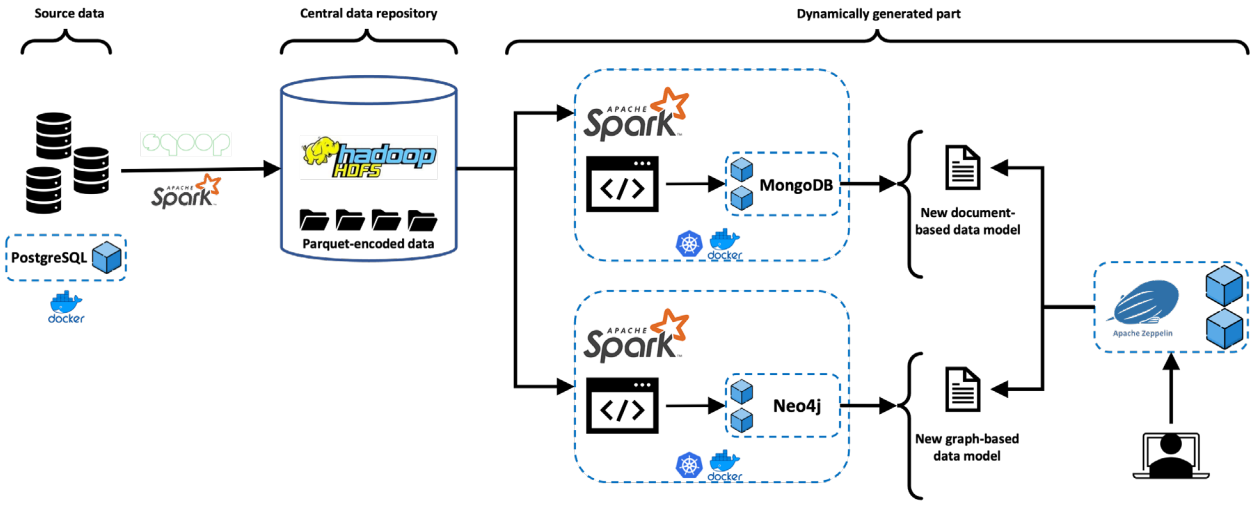

Figure 4.3 Extensibility of the TDVs with multiple data models [82]

Using a relational Data Warehouse within the Data Lake, we can use the same data workflow presented in section 4.2 (conventional method) to provision a TDV. The warehouse representation can be Hadoop Hive or a relational database. It will still not change the lack of options to choose from, especially when the situation demands switching to another data model. The end-user is limited to continuously querying their data using only one data format. On the contrary, our method is flexible in that regard. Using a graph-based data model, we performed a simulation within our architecture to efficiently serve the data; this time, we chose the Helm chart distribution of Neo4j to set up an infrastructure for our TDV. The process is not much different from section 4.1. All we do is dynamically provision PVC and PV for a Neo4j node. We then use Spark to feed the graph

TDV with the new data. A TDV-based graph might be used when geospatial data needs to be stored and queried.

Simultaneously, we illustrate the importance of using multiple data models in the serving layer. We take as an example the four tables mentioned in section 3, namely, Patient, Alert, Allergy, and Medication. They were initially stored in 4 separate tables in PostgreSQL following an entity-relation format. This means that queries would require expensive join operations. In our simulation, we modelled the four tables in one object, Patient, which encapsulates all the other table information. By changing the data model to fit the data exploration project, we have more query options to opt for. We can, for example, query the whole patient object in one go without expensive joints. Our architecture opens the doors for a polystore system implementation. We dynamically added a graph-based TDV using the Neo4j database software solution besides a document-based TDV (MongoDB). This makes it possible for the end users to query from both TDVs simultaneously. Using Spark MLib through a notebook web-based system tremendously helps in such a use case. Lastly, our architecture is flexible enough to implement machine learning models directly from the database, as seen in [45]. This new concept shows better performance and can be easily set up with the help of TDV.

4.4.2 Supported data models

Since testing every dynamically generated TDV available on the market would be unproductive, we established a comprehensive table that gathers the most prominent and compatible cloud-native storage solutions that can be plugged in on the fly as a TDV solution.

Table 4.1 K8s – NoSQL compatibility

Data persistence	**NoSQL family**	**Available on K8s**	**Configuration method**	**Spark connector**
Redis	Key-value	Yes	Operator configuration (enterprise edition) Helm chart	Yes
Couchbase	Key-value/ Document	Yes	Operator	Yes
MongoDB	Document	Yes	Operator configuration Helm chart	Yes
OrientDB	Document/Graph	Somehow	Helm chart (community edition)	Yes
Neo4j	Graph	Yes	Helm chart	Yes
Cassandra	Wide-column	Yes	Operator (K8ssandra)	Yes
HBase	Wide-column	Somehow	Custom Operator	Yes

4.5 TDV in Practice

In this final study, we use our TDV to show how such implementation helps optimize querying in practice. Specifically, we query the anonymized HIV clinical data using a Data Warehouse architecture. Subsequently, we use our Data Lake-based architecture to transform and generate a TDV to query from. Then we compare and discuss the differences between the two approaches. This use case contains four tables holding the patient's medical information records. We want to change the data model to encapsulate the patient's entity as one data object.

4.5.1 Querying Using a Warehouse Architecture

We use the Persistent Cluster to set up the relational database and feed it with a dump of PostgreSQL imported data to model a PostgreSQL Data Warehouse implementation. As in a traditional Data Warehouse, we ETL the data dump from HDFS to Postgres. As requested, we provide a view containing all the patient's medical data records. Following the relational model approach, we make a query with multiple JOIN operations. This is presented as follows:

```
postgres=# SELECT * FROM polycmuql.patient p
FULL JOIN polycmuql.allergy a
ON p.id = a.patient_id
FULL JOIN polycmuql.alert
ON p.id = alert.patient_id
FULL JOIN polycmuql.medication
ON p.id = medication.patient_id
WHERE p.id = 7;
```

Figure 4.4 Example of querying a Patient object in PostgreSQL

4.5.2 Querying Using TDV

Using the TDV, we dynamically generate a MongoDB instance. Afterwards, we transform the data from the HDFS central repository with Spark to request the patient's complete information in one record. Following the same approach as previously shown, we query our data from a K8s instance as follows:

```
> db.hiv.find({id: 7}).pretty()
```

Figure 4.5 Example of querying a Patient object in MongoDB

We added to the annex the queried object used in this comparison.

CHAPTER 5 DISCUSSION

This research reviewed and identified various yet tightly-linked areas in the field of BDPs that could be approached from a new modernized angle. The first area has to do with different data storage strategies in healthcare. The second area of interest relates to the backend supporting such requirements. The third subject of interest was a review of prominent BD architectures that can deliver our innovative solution. The literature review could answer some of the identified areas of interest. Other parts of this thesis could be done with experimentation and design. The research questions outlined in the literature review helped us reach an overall conclusion. We run over those questions and answer them to summarize our main findings.

5.1 Question 1: Discussion

1. *What are the types and characteristics of the targeted health data?*
    a. *Do they need a dedicated infrastructure/platform different from the commercially available solutions?*

The research literature review chapter and related work section answer the first two questions. Healthcare data have the same characteristics as any other from the Big Data category. The only particularity is that healthcare data emphasize the "Variety*"* characteristic given the wide availability of different medical data formats. The "Veracity" and "Value" characteristics can also be added to the main three ("Volume," "Variety," and "Velocity"), given the fact that medical data deal with sensitive and personal information most of the time. Still, we have seen from the related work section that health-related research uses the same Big Data tools and architectures as any other Big Data project. The Hadoop ecosystem (or technologies inspired by it) is broadly used along with Spark in the backend. The platforms may be commercial, free, hosted on-premise or in private/public cloud infrastructure. Regardless, they will always hold the same main functionalities: ingesting, storing, processing, and serving Big Data. Healthcare is no exception. Therefore, our architecture was designed with such observations in mind. Notably, we propose a solution that better assimilates the "Variety" characteristic of Big Data by expanding on a broader range of data models to choose from at execution time.

The “Volume” remains applicable in any Big Data configuration setup, whereas the “Velocity” was managed thanks to the Lambda configuration. The “Veracity” and “Value” were omitted since they require designing a metadata strategy and ontology model that are outside the scope of this thesis. Lastly, through a prototype, we presented a BDP using Hadoop, Spark, and K8s to manage the Big Data workflow's ingestion, storage, and processing layers.

5.2 Question 2: Discussion

2. *What factors do we need to consider for building platforms of such calibre?*
    a. *What are the widespread Big Data architectures that exist?*
    b. *What (open-source) data storage tools are being used to accomplish such a purpose?*

We have seen that Big Data generally considers three mainstream architectures: the Data Warehouse, the Data Lake, and the Data Lakehouse. There are no clear lines that precisely define each one of them. Some research papers regard those architectures as mutually exclusive, whereas others use them as a combination of a single architecture. The articles we reviewed preferred Data Lake-based architectures. The reason is that in its traditional form, the Data Warehouse showed its limitation, especially regarding flexibility and horizontal scaling concerning the massive data increase over time.

On the other hand, the Data Lakehouse is in its early stages. Not enough studies have been conducted, and only a dozen papers have been published. Many uncertainties remain concerning the Lakehouse paradigm and need to be clarified. For instance, it would be helpful to know how well this architecture integrates with the various Big Data tools available and how flexible its data storage is vis-a-vis the different models from the relational/non-relational spectrum. Questions about the manageability and flexibility of its data workflow also need more exploration, and a thorough investigation of its weaknesses needs to be done.

As a result, the Data Lake is and remains the best architecture for Big Data. All the tools invented for the Big Data workflow address a challenge within the Lake infrastructure. It has matured enough and possesses a variety of solutions for solving the same problem.

Based on the answers to these questions, we adopted a Lambda architecture for our BDP. It allows us to handle different incoming data in batch and real-time. Furthermore, its batch layer was proven necessary to implement our novel on-demand concept of temporary storage we called the Temporary Data Views. Our architecture is compatible with many open-source tools for each layer of the data workflow and has a broader range of tools and architectures than the Data Warehouse. It is more mature than the Data Lakehouse, making it a safer architecture to implement to avoid unpredictable scenarios related to Big Data management.

5.3 Question 3: Discussion

3. *What is the most efficient and flexible way to store health data for different needs?*
    a. *How can we anticipate the backend storage component for future data exploration projects?*
    b. *Which data storage model is the best fit for healthcare research projects?*

We have seen throughout our research that health data can take several formats. For this reason, having a Data Lake with a central repository holding various immutable data following a schema-on-read fashion is more recommended. This way, we can better anticipate the storage to query from at later stages of the data processing. Furthermore, the different surveyed papers showed that each health-related use case needs a custom data model, which makes it challenging to settle for a specific data model before deciding on the data to explore during the analytics phase. Therefore, our architecture tries not to exclude any existing data model by (1) delaying its creation later in the data processing workflow. (2) dynamically creating such data stores using scalable cloud-native components with K8s.

5.4 Question 4: Discussion

4. *Granted, we possess a Data Platform capable of processing various types of health data, but how can we ensure it will provide custom data storage models for targeted health projects?*

The TDVs represent the conclusion to address this question. Our approach to solving this issue was to introduce the TDVs in a Data Lake Lambda architecture. They brought up that much-needed

flexibility by allowing dynamic data generation with the help of a processing engine and containerization. The TDVs took advantage of the scalability, modularity, and cost-effectiveness of K8s to rapidly, automatically, and dynamically generate a containerized database that can temporarily store the Spark-transformed data to its targeted data model. TDVs will support data scientists by providing them with data stored in its most compatible data storage model, granting a much more organized dataset to help them query without much hassle.

CHAPTER 6 CONCLUSION AND FUTURE WORK

6.1 Summary

This thesis explored the architecture and storage techniques for building a resilient backend of a Big Data Platform compatible with healthcare data workflows (Health Big Data). We explored and answered the four research questions through the literature review and our proposed solution. We looked for a method to include in our solution that could allow for dynamically generating the dataset to query from at a later stage, more precisely during the data science exploration phase. More specifically, in this thesis, we explored a backend and storage architecture within a BDP that expands on the "Variety" aspect of Big Data without undermining the other Vs. We confirmed that a Data Lake architecture for BDP is more mature, available, and complete for healthcare analytics. Its Lambda variant proved efficient for batch and real-time data processing. It facilitated the establishment of the TDVs, our flexible solution for providing various data models to query from on-demand.

6.2 Limitations

While our architecture has proven to address the research questions and provide a solution to identified problems, we identified, throughout our research, questions and limitations that must be addressed in future work. Our work focused more on providing a solution from an architectural perspective and targeted specific issues in the storage and backend component. However, to make our Big Data Platform more complete and production-ready, we must surround it with an ontology. The modelling would help shape a conceptual object design, thus facilitating the automation tasks during the data transformation phase with the TDVs. The evoked polystore system extensibility of our TDV is still a simplification of a more complex concept that needs additional building layers that takes care of a uniform querying system and may use ontology to organize the conceptual model of the data. A metadata strategy would also help keep the Big Data more organized in the central Lake repository. Adding a data lineage strategy will also help keep up with the traceability of the data. Advanced work on the platform's security, administration roles, data governance, and access control should be performed during the third part of this Big Data Platform project. Our evaluation was done in a more qualitative way and would greatly benefit from other quantitative

ones once pertinent health data become available. Finally, work on benchmarking needs to be done to quantify the performance of the K8s-hosted NoSQL TDVs regarding their data storage models' workloads on different data workflow scenarios.

6.3 Future work

All the limitations evoked are planned for future work of a still-ongoing project. The first version of the Big Data Platform will soon be implemented on Azure cloud. We intend to implement the Data Lake with the Lambda architecture with all the foundational components for the ingestion, storage, processing, and analytics. The step afterwards will introduce the explored TDVs to the platform, followed by an ontology model and a strong metadata orchestration strategy for the backend. This will also ease building a complete TDV-backed polystore system. We will undergo a new evaluation strategy based on surveys from domain experts once we reach an agreement. We also intend to collaborate with them to perform new query-centred evaluations and not just use mock queries. Future projects presently being discussed also include the integration of the natural language processing framework for querying and automating the TDV generation process. Alternatively, we will explore and test our TDVs on other novel architectures like the Data Lakehouse and inspect their limitations.

REFERENCES

[1] M. Wang *et al.*, "Big data health care platform with multisource heterogeneous data integration and massive high-dimensional data governance for large hospitals: Design, development, and application," *JMIR Medical Informatics,* vol. 10, no. 4, p. e36481, 2022.

[2] R. Hai, C. Quix, and M. Jarke, "Data lake concept and systems: a survey," *ArXiv,* vol. abs/2106.09592, 2021.

[3] "Cloudera." https://www.cloudera.com/ (accessed 2022).

[4] Oracle. "Oracle Big Data Platform." https://www.oracle.com/ca-en/big-data/ (accessed 2022).

[5] M. Ross and R. Kimball, *The data warehouse toolkit: the definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[6] P. Sawadogo and J. Darmont, "On data lake architectures and metadata management," *Journal of Intelligent Information Systems,* vol. 56, no. 1, pp. 97-120, 2021.

[7] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics," in *Proceedings of CIDR*, 2021.

[8] B. Inmon, M. Levins, and R. Srivastava, *Building the Data Lakehouse*. Technics Publications, 2021.

[9] V. Bureva, "Index Matrices as a Tool for Data Lakehouse Modelling," *Annual of "Informatics" Section of the Union of Scientists in Bulgaria,* vol. 10, pp. 81-105, 2019.

[10] A. N. Abougreen, "Big Medical Data Analytics Under Internet of Things," in *Efficient Data Handling for Massive Internet of Medical Things: Healthcare Data Analytics*, C. Chakraborty, U. Ghosh, V. Ravi, and Y. Shelke Eds. Cham: Springer International Publishing, 2021, pp. 25-44.

[11] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International journal of information management,* vol. 35, no. 2, pp. 137-144, 2015.

[12] W. Hadoop, "T.: Hadoop: The Definitive Guide," ed: O'Reilly Media Inc, 2015.

[13] C. Forresi, E. Gallinucci, M. Golfarelli, and H. B. Hamadou, "A dataspace-based framework for OLAP analyses in a high-variety multistore," *The VLDB Journal,* vol. 30, no. 6, pp. 1017-1040, 2021.

[14] F. Mostajabi, A. A. Safaei, and A. Sahafi, "A Systematic Review of Data Models for the Big Data Problem," *IEEE Access,* vol. 9, pp. 128889-128904, 2021.

[15] K. Slime, A. Maizate, and L. Hassouni, "Processing and analyzing health data in a big data context: aspects and implementations," in *International Conference on Advanced Intelligent Systems for Sustainable Development*, 2020: Springer, pp. 299-307.

[16] A. Oussous, F.-Z. Benjelloun, A. A. Lahcen, and S. Belfkih, "Big Data technologies: A survey," *Journal of King Saud University-Computer and Information Sciences,* vol. 30, no. 4, pp. 431-448, 2018.

[17] N. Shehab, M. Badawy, and H. Arafat, "Big Data Analytics Concepts, Technologies Challenges, and Opportunities," Cham, 2020: Springer International Publishing, in Proceedings of the International Conference on Advanced Intelligent Systems and Informatics 2019, pp. 92-101.

[18] C. Gao, W. Luo, and F. Xie, "Software Selection Test of Enterprise-Level Big Data Analysis Platform."

[19] "APACHE KAFKA." https://kafka.apache.org/ (accessed 2022).

[20] "Apache Sqoop." https://sqoop.apache.org/ (accessed 2022).

[21] "Welcome to Apache Flume." https://flume.apache.org/ (accessed 2022).

[22] A. Thusoo *et al.*, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment,* vol. 2, no. 2, pp. 1626-1629, 2009.

[23] S. Murugesan and I. Bojanova, *Encyclopedia of cloud computing*. John Wiley & Sons, 2016.

[24] A. Cassandra. "Cassandra Basics." https://cassandra.apache.org/_/cassandra-basics.html (accessed 2022).

[25] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide: time to relax*. " O'Reilly Media, Inc.", 2010.

[26] PostgreSQL. "PostgreSQL: The World's Most Advanced Open Source Relational Database." https://www.postgresql.org/ (accessed 2022).

[27] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM,* vol. 51, no. 1, pp. 107-113, 2008.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud,* vol. 10, no. 10-10, p. 95, 2010.

[29] Spark. "ML Pipelines." https://spark.apache.org/docs/3.3.1/ml-pipeline.html#details (accessed 2022).

[30] A. Or. "Understanding your Apache Spark Application Through Visualization." Databricks. https://www.databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html (accessed 2022).

[31] "Apache Storm." https://storm.apache.org/ (accessed 2022).

[32] "Welcome to Apache Pig!" https://pig.apache.org/ (accessed 2022).

[33] "What is Apache Flink? — Architecture." https://flink.apache.org/flink-architecture.html (accessed 2022).

[34] "Apache Zeppelin." https://zeppelin.apache.org/ (accessed 2022).

[35] "Jupyter." https://jupyter.org/ (accessed 2022).

[36] "MLlib is Apache Spark's scalable machine learning library." https://spark.apache.org/mllib/ (accessed 2022).

[37] "TensorFlow." https://www.tensorflow.org/ (accessed 2022).

[38] "Scikit-learn." https://scikit-learn.org/stable/getting_started.html (accessed 2022).

[39] F. Rouzbeh, A. Grama, P. Griffin, and M. Adibuzzaman, "Collaborative Cloud Computing Framework for Health Data with Open Source Technologies," in *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 2020, pp. 1-10.

[40] P. Revesz, *Introduction to databases*. Springer, 2010.

[41] A. Meier and M. Kaufmann, "Data Management," in *SQL & NoSQL Databases*: Springer, 2019, pp. 1-23.

[42] R. L. Winslow, S. Granite, and C. Jurado, "WaveformECG: A platform for visualizing, annotating, and analyzing ECG data," *Computing in science & engineering,* vol. 18, no. 5, pp. 36-46, 2016.

[43] S. Uzunbayir, "Relational Database and NoSQL Inspections using MongoDB and Neo4j on a Big Data Application," in *2022 7th International Conference on Computer Science and Engineering (UBMK)*, 2022: IEEE, pp. 148-153.

[44] D. Sullivan, *NoSQL for mere mortals*. Addison-Wesley Professional, 2015.

[45] A. Celesti, F. Celesti, A. Galletta, M. Fazio, and M. Villari, "Improving machine learning algorithm processing time in tele-rehabilization through a nosql graph database approach: A preliminary study," in *2020 IEEE Symposium on Computers and Communications (ISCC)*, 2020: IEEE, pp. 1-6.

[46] S. Rouhani and S. Rotbei, "Big data platforms: in the lens of selection and evaluation approach," *Journal of Decision Systems,* pp. 1-30, 2021.

[47] A. Sebaa, F. Chikh, A. Nouicer, and A. Tari, "Medical big data warehouse: architecture and system design, a case study: improving healthcare resources distribution," *Journal of medical systems,* vol. 42, no. 4, pp. 1-16, 2018.

[48] B. Sharma, *Architecting data lakes: data management architectures for advanced business use cases*. O'Reilly Media, 2018.

[49] C. Mathis, "Data lakes," *Datenbank-Spektrum,* vol. 17, no. 3, pp. 289-293, 2017.

[50] E. Begoli, I. Goethert, and K. Knight, "A lakehouse architecture for the management and analysis of heterogeneous data for biomedical research and mega-biobanks," in *2021 IEEE International Conference on Big Data (Big Data)*, 2021: IEEE, pp. 4643-4651.

[51] D. Vohra and D. Vohra, "Apache parquet," *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools,* pp. 325-335, 2016.

[52] B. R. Prasad and S. Agarwal, "Comparative study of big data computing and storage tools: a review," *International Journal of Database Theory and Application,* vol. 9, no. 1, pp. 45-66, 2016.

[53] E. M. Ouafiq, M. Raif, A. Chehri, and R. Saadane, "Data Architecture and Big Data Analytics in Smart Cities," 2022.

[54] D. Vohra and D. Vohra, "Apache avro," *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools,* pp. 303-323, 2016.

[55] F. C. Liu, F. Shen, D. H. Chau, N. Bright, and M. Belgin, "Building a research data science platform from industrial machines," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016: IEEE, pp. 2270-2275.

[56] "Architecture." https://docs.ceph.com/en/latest/architecture/ (accessed 2023).

[57] N. Price, K. Rodgerson, W. Byron, and H. M. K. M. SM, "A Scalable Data Science Platform for Healthcare and Precision Medicine Research," 2018.

[58] H. Samra, A. Li, and B. Soh, "GENE2D: A NoSQL integrated data repository of genetic disorders data," in *Healthcare*, 2020, vol. 8, no. 3: MDPI, p. 257.

[59] L. Ehwerhemuepha, G. Gasperino, N. Bischoff, S. Taraman, A. Chang, and W. Feaster, "HealtheDataLab – a cloud computing solution for data science and advanced analytics in healthcare with application to predicting multi-center pediatric readmissions," *BMC Medical Informatics and Decision Making,* vol. 20, no. 1, p. 115, 2020/06/19 2020, doi: 10.1186/s12911-020-01153-7.

[60] H. Calderón-Gómez *et al.*, "Telemonitoring system for infectious disease prediction in elderly people based on a novel microservice architecture," *IEEE Access,* vol. 8, pp. 118340-118354, 2020.

[61] K. Ghane, "Big data pipeline with ML-based and crowd sourced dynamically created and maintained columnar data warehouse for structured and unstructured big data," in *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, 2020: IEEE, pp. 60-67.

[62] P. Jovanovic, S. Nadal, O. Romero, A. Abelló, and B. Bilalli, "Quarry: A User-centered Big Data Integration Platform," *Information Systems Frontiers,* vol. 23, no. 1, pp. 9-33, 2021/02/01 2021, doi: 10.1007/s10796-020-10001-y.

[63] A. Sanla and T. Numnonda, "A comparative performance of real-time big data analytic architectures," in *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, 2019: IEEE, pp. 1-5.

[64] A. Gillet, É. Leclercq, and N. Cullot, "Lambda Architecture pour une analyse à haute performance des données des réseaux sociaux," in *INFormatique des ORganisations et Systèmes d'Information et de Décision*, 2019.

[65] A. A. Munshi and Y. A.-R. I. Mohamed, "Data lake lambda architecture for smart grids big data analytics," *IEEE Access,* vol. 6, pp. 40463-40471, 2018.

[66] R. Alghamdi and M. Bellaiche, "A deep intrusion detection system in lambda architecture based on edge cloud computing for IoT," in *2021 4th International Conference on Artificial Intelligence and Big Data (ICAIBD)*, 2021: IEEE, pp. 561-566.

[67] N. Cassavia and E. Masciari, "Sigma: a scalable high performance big data architecture," in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2021: IEEE, pp. 236-239.

[68] M. Marinov, "Using HBase to Implement Speed Layer in Time Series Data Storage Systems," *International Journal of Advanced Computer Science and Applications,* vol. 13, no. 2, 2022.

[69] M. Stonebraker and U. Cetintemel, ""One size fits all": an idea whose time has come and gone," in *21st International Conference on Data Engineering (ICDE'05)*, 5-8 April 2005 2005, pp. 2-11, doi: 10.1109/ICDE.2005.1.

[70] J. Duggan *et al.*, "The bigdawg polystore system," *ACM Sigmod Record,* vol. 44, no. 2, pp. 11-16, 2015.

[71] Z. Bicevska and I. Oditis, "Towards NoSQL-based data warehouse solutions," *Procedia Computer Science,* vol. 104, pp. 104-111, 2017.

[72] A. Eremeev, S. Ivliev, and V. Vagin, "Using NoSQL databases and machine learning for implementation of intelligent decision system in complex vision patalogies," in *2018 3rd Russian-Pacific Conference on Computer Technology and Applications (RPC)*, 2018: IEEE, pp. 1-4.

[73] P. S. Sen and N. Mukherjee, "Designing a NoSQL Database for Efficient Storage and Retrieval of Health Data," in *2021 8th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2021: IEEE, pp. 262-269.

[74] T. Vanhove, M. Sebrechts, G. Van Seghbroeck, T. Wauters, B. Volckaert, and F. De Turck, "Data transformation as a means towards dynamic data storage and polyglot persistence," *International Journal of Network Management,* vol. 27, no. 4, p. e1976, 2017.

[75] F. Abdelhedi, R. Jemmali, and G. Zurfluh, "Relational Databases Ingestion into a NoSQL Data Warehouse," *arXiv preprint arXiv:2203.06949,* 2022.

[76] A. Tönne, "Big Data is no longer equivalent to Hadoop in the industry," *Datenbanksysteme für Business, Technologie und Web (BTW 2017),* 2017.

[77] N. Marz. "How to beat the CAP theorem." http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html (accessed 2022).

[78] J. Warren and N. Marz, *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.

[79] G. Fahimimoghaddam, "A Customizable On-Demand Big Data Health Analytics Platform Using Cloud and Container Technologies," Polytechnique Montréal, 2021.

[80] Smartline, "Api," ed. https://www.flaticon.com/free-icons/api: Flaticon.

[81] Freepik, "Flat design laptop logo template," ed. https://www.freepik.com/free-vector/flat-design-laptop-logo-template_15291885.htm#page=19&query=computer%20logo&position=7&from_view=search&track=sph: Freepik, 2021.

[82] Freepik, "Cube," ed. https://www.flaticon.com/free-icon/cube_551227?related_id=551125&origin=search: Freepik.

[83] E. Truyen, M. Bruzek, D. Van Landuyt, B. Lagaisse, and W. Joosen, "Evaluation of container orchestration systems for deploying and managing NoSQL database clusters," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018: IEEE, pp. 468-475.

[84] H. Perera *et al.*, "Database scaling on Kubernetes," in *2021 3rd International Conference on Advancements in Computing (ICAC)*, 2021: IEEE, pp. 258-263.

[85] T.-C. Lo, C.-Y. Tao, J.-B. Chang, and C.-K. Shieh, "Performance Comparison of Containerized HBase Clusters on Kubernetes," in *2022 IEEE International Conference on Recent Advances in Systems Science and Engineering (RASSE)*, 2022: IEEE, pp. 1-7.

[86] S. Wu, X. Wang, B. Tang, X. Li, J. Zhu, and K. Deng, "A Cloud Storage Framework for Massive Meteorological and Oceanographic Data and the Application of Virtualization Technology," in *2020 International Conference on Space-Air-Ground Computing (SAGC)*, 2020: IEEE, pp. 25-32.

[87] P. Kochovski, R. Sakellariou, M. Bajec, P. Drobintsev, and V. Stankovski, "An architecture and stochastic method for database container placement in the edge-fog-cloud continuum," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019: IEEE, pp. 396-405.

[88] "HELM The package manager for Kubernetes." https://helm.sh/ (accessed 2023).

[89] MongoDB. "MongoDB Community Kubernetes Operator." https://github.com/mongodb/mongodb-kubernetes-operator (accessed 2023).

[90] C. Hatton. "Run Secure Containerized MongoDB Deployments Using the MongoDB Community Kubernetes Operator." https://www.mongodb.com/blog/post/run-secure-containerized-mongodb-deployments-using-the-mongo-db-community-kubernetes-oper (accessed 2022).

[91] MongoDB. "MongoDB Enterprise Kubernetes Operator." https://www.mongodb.com/docs/kubernetes-operator/master/ (accessed 2023).

[92] Neo4j. "The Neo4j Operations Manual v5 - Kubernetes." https://neo4j.com/docs/operations-manual/current/kubernetes/ (accessed 2023).

[93] K8ssandra. "K8ssandra." https://k8ssandra.io/ (accessed 2023).

[94] Datastax. "Cassandra and Kubernetes." https://www.datastax.com/dev/kubernetes (accessed 2023).

[95] CockroachDB. "CockroachDB Kubernetes Operator." https://github.com/cockroachdb/cockroach-operator (accessed 2023).

[96] Y. K. Gupta and S. Kumari, "A study of big data analytics using apache spark with Python and Scala," in *2020 3rd International Conference on Intelligent Sustainable Systems (ICISS)*, 2020: IEEE, pp. 471-478.

[97] H. K. Omar and A. K. Jumaa, "Big data analysis using apache spark MLlib and Hadoop HDFS with Scala and Java," *Kurdistan Journal of Applied Research,* vol. 4, no. 1, pp. 7-14, 2019.

[98] A. Bouziane. "Prototype BDP for TDV." https://github.com/High-drogen/BDP-with-TDV (accessed 2023).

[99] K. SIGs. "Deploy a Production Ready Kubernetes Cluster." https://github.com/kubernetes-sigs/kubespray (accessed 2022).

[100] A. Bouziane. "Prototype BDP for TDV — Ingestion." https://github.com/High-drogen/BDP-with-TDV/tree/main/Ingestion (accessed 2023).

[101] A. Bouziane. "Prototype BDP for TDV — Transformation and storage." https://github.com/High-drogen/BDP-with-TDV/tree/main/Transformation%20and%20write%20to%20TDV (accessed 2023).

APPENDIX A QUERIED OBJECT FROM SECTION 4.5.2