Departamento de Electrónica, Sistemas e Informática
Maestría en Sistemas Computacionales

Adaptive Big Data Pipelines

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN SISTEMAS COMPUTACIONALES

Presenta: **ING. ALDO OROZCO GÓMEZ SERRANO**

Elija un elemento. **PH.D. ALBERTO DE OBESO ORENDAIN**

Tlaquepaque, Jalisco. 22 de septiembre del 2020

ACKNOWLEDMENT

First of all, I am very much obliged to my parents, Laura and Hector, who, mostly inadvertently, motivated me during this journey by always exemplifying that virtue can only be achieved through hard work and perseverance. In a similar manner, I also want to thank my nephew, Dante, whom I deeply respect and consider as my own son, for he has indirectly given me the courage to carry on despite the circumstances.

I would also like to express my gratitude to the *Consejo Nacional de Ciencia y Tecnología (CONACYT)* for granting me the scholarship number 923982 thanks to which I was able to successfully complete my master's degree.

I am also very grateful towards my thesis tutor, ex-boss and good friend, Alberto De Obeso, who provided me with guidance as well as feedback whenever needed and, above all, was the first person to trust my skills and gave me the opportunity to switch careers despite my lack of experience in the area, to what ultimately became my current passion: big data.

Lastly, I want to extend my gratitude to Apex Systems—formerly known as Intersys Consulting—and my current employer, Wizeline, for providing me with financial support during parts of this master's degree.

AGRADECIMIENTOS

Primero que nada, quiero agradecer a mis padres, Laura y Héctor, quienes sin darse cuenta me motivaron durante este viaje al siempre ejemplificar que la virtud solo puede ser adquirida a través de trabajo duro y perseverancia. De forma similar quiero agradecerle a mi sobrino, Dante, a quien respeto profundamente y considero como mi propio hijo, puesto que el me ha dado indirectamente el coraje de seguir adelante a pesar de las circunstancias.

También deseo expresar mi gratitud al Consejo Nacional de Ciencia y Tecnología (CONACYT) por otorgarme la beca con numero 923982, gracias a la cual fui capaz de terminar mis estudios de forma exitosa.

Asimismo, estoy muy agradecido con mi asesor de tesis, exjefe y buen amigo, Alberto de Obeso, quien me guio y proveyó retroalimentación siempre que la necesitaba y, sobre todo, porque fue la primera persona en confiar en mis habilidades y darme la oportunidad de iniciar una nueva carrera a pesar de mi falta de experiencia en el área, lo que finalmente me llevo a mi actual pasión: big data.

Finalmente deseo extender el agradecimiento a Apex Systems—anteriormente conocido como Intersys Consulting—y a mi empleador actual, Wizeline, por apoyarme económicamente durante partes de este grado académico.

DEDICATION

To my parents, friends, and wholeheartedly to anyone who supported or has supported me during my technological, mental and spiritual evolution; I truly hope that you find the same inspiration that led me to this exciting path.

DEDICATORIA

A mis padres, amigos y, de todo corazón, a todos aquellos que me dieron o siguen dándome apoyo durante mi evolución tecnológica, mental y espiritual; de verdad espero que encuentren la misma inspiración que me llevo a este emocionante camino.

ABSTRACT

Over the past three decades, data has exponentially evolved from being a simple software by-product to one of the most important companies' assets used to understand their customers and foresee trends.

Deep learning has demonstrated that big volumes of clean data generally provides more flexibility and accuracy when modeling a phenomenon. However, handling ever-increasing data volumes entail new challenges: the lack of expertise to select the appropriate big data tools for the processing pipelines, as well as the speed at which engineers can take such pipelines into production reliably, leveraging the cloud.

We introduce a system called Adaptive Big Data Pipelines: a platform to automate data pipelines creation. It provides an interface to capture the data sources, transformations, destinations and execution schedule. The system builds up the cloud infrastructure, schedules and fine-tunes the transformations, and creates the data lineage graph.

This system has been tested on data sets of 50 gigabytes, processing them in just a few minutes without user intervention.

RESUMEN

En las ultimas tres décadas, los datos evolucionaron exponencialmente de ser un simple subproducto de software a ser uno de los activos mas importantes de las compañías que se usan para entender a sus clientes y prever tendencias.

El *Deep Learning* ha demostrado que grandes volúmenes de datos limpios proveen mayor flexibilidad y precisión cuando se modela un fenómeno. Sin embargo, manejar volúmenes de datos que se incrementan constantemente conlleva nuevos desafíos: la falta de experiencia para seleccionar las herramientas de datos apropiadas para la construcción de tuberías de procesamiento de datos, así como la velocidad con la cual se pueden llevar dichas tuberías a producción de forma confiable, aprovechando la nube.

Presentamos un sistema llamado *Adaptive Big Data Pipelines*: una plataforma para automatizar la creación de tuberías de datos. Provee una interfaz para capturar fuentes de datos, transformaciones, destinos y agenda de ejecución. El sistema construye la infraestructura de nube, agenda y refina las transformaciones y crea el grafo de linaje de datos.

El sistema ha sido probado en conjuntos de datos de 50 gigabytes, procesándolas en tan solo un par de minutos sin intervención humana.

1. TABLE OF CONTENTS

FIGURES

TABLES

ACRONYMS AND ABBREVIATIONS

ACID	Atomicity, Consistency, Isolation and Durability
ABDP	Adaptive Big Data Pipelines
API	Application Programing Interface
CRUD	Create, Read, Update and Delete
DAG	Directed Acyclic Graph
DW	Data Warehouse
EDW	Enterprise Data Warehouse
ETL	Extract Transform and Load
FSM	Finite State Machine
IaC	Infrastructure as Code
IoT	Internet of Things
JVM	Java Virtual Machine
RDBMS	Relational Database Management System
SQL	Structured Query Language
UI	User Interface
VM	Virtual Machine
YAML	Yet Another Markup Language

Table 1 – Acronyms and abbreviations

2. INTRODUCTION

2.1. Background

As with probably everything else in human history, the concept *Big Data* came into existence as a necessity: exponential economic growth.

In the early 2000s, Google became one of the first companies in collecting and handling massive amounts of user data. In consequence, they soon reached the limits of storage and processing technologies and came up with new mechanisms to tackle the ever-increasing data demand: GFS and MapReduce.

The Google File System (GFS) [1] is the first distributed storage solution capable of dealing with terabytes of data across thousands of disks. MapReduce [2] is, in contrast, the first distributed processing framework that breaks down a complex logic into simpler functions that are executed on a large cluster in parallel.

Both papers marked the beginning of the big data era. Soon after their release, multiple open source tools emerged based on their philosophy. In particular, given the amount of data generated by increasingly complex and inter-connected sensors—Internet of Things (IoT).

Below we outline the most important tools of the data ecosystem:

- **Apache Hadoop -** A distributed framework for both storage and processing. Directly influenced by both papers.
- **Apache Hive** – A data warehousing solution built on top of Hadoop.
- **Apache Spark** – A distributed in-memory computing engine that is capable of 100x faster processing than Hadoop's MapReduce.
- **Parquet** – A columnar binary data format for analytics.
- **Amazon Web Services (AWS)** – The number one cloud provider in the world at the time of writing. It provides countless services ranging from compute to storage on a pay-as-you-go model.
- **Terraform** – An open source Infrastructure as Code (IaC) tool capable of leveraging some of the top cloud providers as of now.
- **Git** – The de facto version control system for any software artifact.
- **Spline** – A data lineage tool built specifically to interact with Apache Spark.
- **DBT** – A data warehousing tool to build data pipelines on pure SQL.
- **Docker** – A tool to build, manage and monitor containers.
- **Kubernetes** – A distributed container orchestration service.

2.2. Justification

Data analytics is the process of dissecting, aggregating, transforming and combining data sets, so that analysts can derive insights from them, in order for businesspeople to understand trends and detect anomalies. It is a crucial process in the companies' operation as it further allows them to understand their customers and generate higher revenue.

Over the last decade, deep learning has demonstrated that the more data users have, the more flexible and precise the modeling of a phenomenon may become. This is leading companies all over the world to collect more data from their customer interactions, for instance through their websites or through Internet of Things (IoT) devices. With this amount of data, most common technologies collapse after a few gigabytes are reached.

Big data tools have matured considerably since the early 2000s, in response to this data volume soar. However, most of these technologies are use-case oriented, complex and require high user intervention and expertise to interconnect effectively.

Companies are under competitive pressure to release products and services, see how clients interact with them, improve and iterate on a timely manner. Given the lack of expertise, companies often take up to a couple months to start organizing and analyzing the data they have and therefore may lay behind on market trends and react slowly to the user's needs.

2.3. Problem description

Nowadays, Startups are materializing at an increasing pace. Although, admittedly, part of this proclivity is due to the media misconception and exaggeration of professional independence, it is also true that these small companies are tackling problems that larger companies often oversee or are unwilling to invest on. Technology has granted Startups the nimbleness to quickly prove hypothesis, for a given service or product, and improve it. Ultimately, this helps companies grow exponentially and expand their target market.

Data plays a decisive role on the Startups' success, as it provides a framework against which hypothesis are factually validated: if a company believes that a given service improvement will be liked by its customers, it will release the new service and measure how users interact and compare against previous iterations to see whether the experiment succeeded or failed.

In order to generate precise hypothesis nowadays, firms are encouraged to collect and process large volumes of data, also known as Big Data, as they are the fuel for cutting-edge machine learning algorithms to achieve unprecedented precision and uncover hidden trends. The usage of big data is further facilitated by the open source community as it has implemented a myriad of tools to handle most data needs.

The skills required to build an end-to-end big data solution are diverse and the offer is low, which means that data experts are extremely sought after, hence expensive. As such, companies often rely on software engineers to build a data ecosystem using open source technology. The insufficiency of fundamental knowledge forces them to naively adopt tools that are not suitable for the companies needs and end up with an inflexible and unscalable system.

In this scenario, businesses are restricted to wait from several weeks to a few months to be able to see pipelines produce the desired outcome. Even when the firm achieves a fairly stable data ecosystem, scalability soon turns into the bottleneck.

Proper migration to the cloud can alleviate scalability issues. However, the technical team must first understand a wide range of services each cloud provider offers, then create the right infrastructure and ultimately lay out a plan to gradually lift and shift to the cloud. This adds up to the delay to production causing companies to lose even more money.

A key limitation in big data pipelines is the ability for data analysts to interact with them effectively in SQL, as many tools only support transformations using programming languages such as Python and Java.

2.4. Hypothesis

Adoption of an self-deployable cloud-based data pipeline building system with a simple SQL interface by data analysts and business users will result in a dramatic time decrease to take advantage of the data.

2.5. Objectives

2.5.1. General objective

Implement a scalable data pipeline building system, capable of generating data pipelines with minimum user intervention, enforcing data best practices (i.e., data lake architecture, pipeline tuning and data governance)

2.5.2. Specific objectives

1. Build containerized microservices for the different services to support the system, so that they are platform agnostic and isolated.
2. Create an infrastructure-as-code (IaC) module capable of building and tearing down the infrastructure to minimize costs.
3. Expose a simple web interface where users can create pipelines.
4. Create a method to extract metadata from local source files.
5. Implement a scheduling service capable of automatically pick up new directed acyclic graphs (DAGs).
6. Develop a generic pipeline capable of handling ANSI SQL queries.
7. Build a service capable of automatically capturing the data lineage of the data pipelines.

2.6. Scientific, technological novelty or contribution

The contribution of this work is aimed toward streamline companies' operations by granting a flexible data pipeline building system, underpinned by some of the top open source projects at the time of writing. In particular, the system encompasses the following data-related areas:

1. Auto-scalable data pipelines
2. Pipeline scheduling
3. Containerized applications
4. Cloud native applications
5. Infrastructure as code
6. Data governance
7. Security

3. STATE OF THE ART

3.1. Google File System (GFS)

The Google File System (GFS) [1] is a scalable distributed file system for large distributed data-intensive applications. The research paper was published by Google in 2003 and it represented one of the first implementations of a highly scalable and battle-tested distributed filesystem.

GFS stores data on a cluster of commodity hardware—nodes—that are assumed to fail, and hence the overall system must recover from any disaster automatically. Some key limitations are that write operations were strictly limited to appends and that files are organized into folders, resembling any other file system; underlying files are physically split into fixed size blocks and replicated across multiple nodes to guarantee consistency and high availability in case a node fails.

Its architecture is comprised of a master node and N slave nodes to store data. This allows users to interact with a single interface which takes care of storing and orchestrating reads and writes.

3.2. MapReduce

MapReduce [2] is a programming model for massive data processing over large datasets. It was inspired by the way functional programming languages parallelize data processing on all available threads. It was published by Google on 2004.

MapReduce, as the name implies, is comprised by two main processes: map and reduce. Map takes any number of inputs and performs a transformation for each input. The output of a mapper tends to be the same number of input records; transformations are performed in parallel by all nodes in the cluster. After that, the Reduce stage starts, which consists of taking data from all nodes and reducing it to execute an aggregation (e.g., to perform a group by, in SQL jargon).

To enable fault tolerance, both mappers and reducers write intermediate data to each node's disk. In this way, if a node fails, its progress can easily be restored and will not have to be recomputed.

MapReduce is architected using a master node and an arbitrary number of worker nodes. The user submits a MapReduce job—a JAR file with a Main class implementing mappers and reducers—to the master, which automatically distributes the load to available nodes and recovers in case of a node failure.

MapReduce's leverages data locality (i.e., computation occurs in the nodes where the data is located) to avoid unnecessary network shuffling and hence speed up the processing.

The following figure illustrates the MapReduce process:

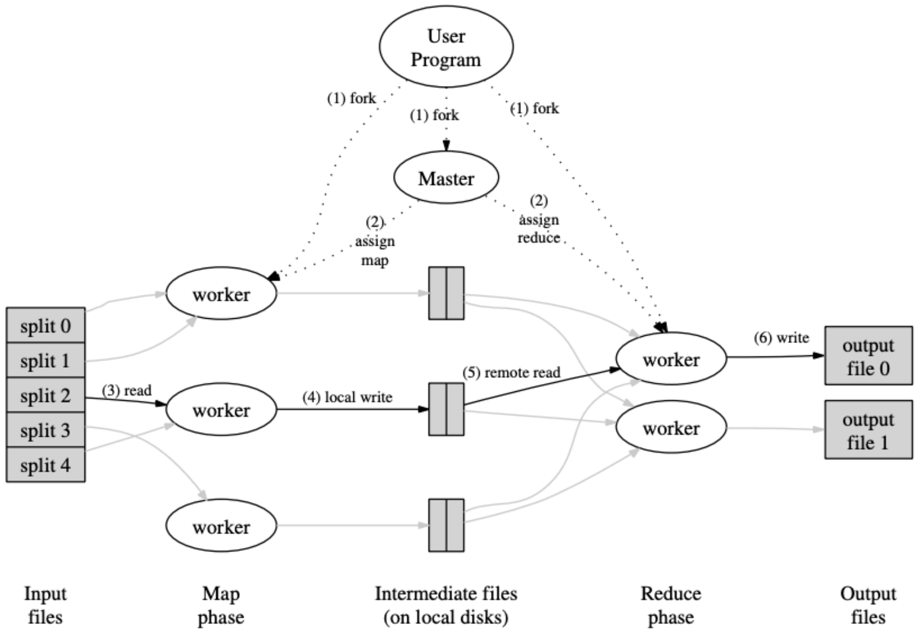


Figure 1 - MapReduce execution overview [2]

Even though MapReduce is losing momentum at the time of writing due to more advanced processing frameworks such as Apache Spark, MapReduce still has the niche of processing large unstructured datasets at scale. For instance, image processing or genome computations can be effectively computed using MapReduce.

3.3. Dremel

Dremel is a scalable interactive query system for massively large read-only datasets [3]. The tool was used at Google to effectively replace interactive analysis previously done in MapReduce jobs. As with many interactive query systems, it is based on SQL, to allow analysts to leverage it.

Dremel was built considering that clusters are shared, meaning that commodity hardware could run an arbitrary number of applications at the same time [3]. Therefore, the possibility of a process failing soared and the system had to be able to deal with such scenario.

The secret behind this technology is the idea of splitting the computation and storage.

Computation uses multi-level execution trees: a root server receives incoming queries, reads the metadata of the underlying tables and routes the queries to intermediate servers until a leaf server, a server with local storage, is reached.

For storage, Google File System (GFS) is used. On top of that, Dremel proposes a new columnar format that is highly efficient for analytic purposes. These files incorporate metadata to describe repetition and definition levels to clearly locate nested values and overall data organization. Thanks to this format, each file block can easily be compressed by removing NULL values.

This file format allows physically splitting records into columns for block storage when writing, and assembly of those records using a Finite State Machine (FSM) when reading.

3.4. Hadoop

In 2006, Doug Cutting, still working at Yahoo, started implementing Hadoop based on both GFS and MapReduce [4]. Hadoop is an open source platform that allows for distributed processing of nearly unlimited large data sets across clusters of computers. It is comprised of the following core components:

- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets [5].

HDFS is often considered the open source implementation of the Google File System and so it shares most of its concepts and the fact that is highly distributed.

As described in [6], HDFS does not substitute any native filesystem, but it rather offers abstraction on top of your local filesystem to provide fault tolerance. Such layer emulates a hierarchical filesystem, such as the native Linux filesystem, comprised by folders and files; however, internally a complex data orchestration takes place.

From a high-level perspective, once a user issues a request to write a file to HDFS, the system will break down the file into fixed-size blocks—currently 128MB. Each block is then replicated to multiple arbitrary nodes so that, in case of a node failure, the system can collect the different pieces from multiple nodes and hence the system is more resilient.

The overall design and access pattern in HDFS is like a Linux-based filesystem:

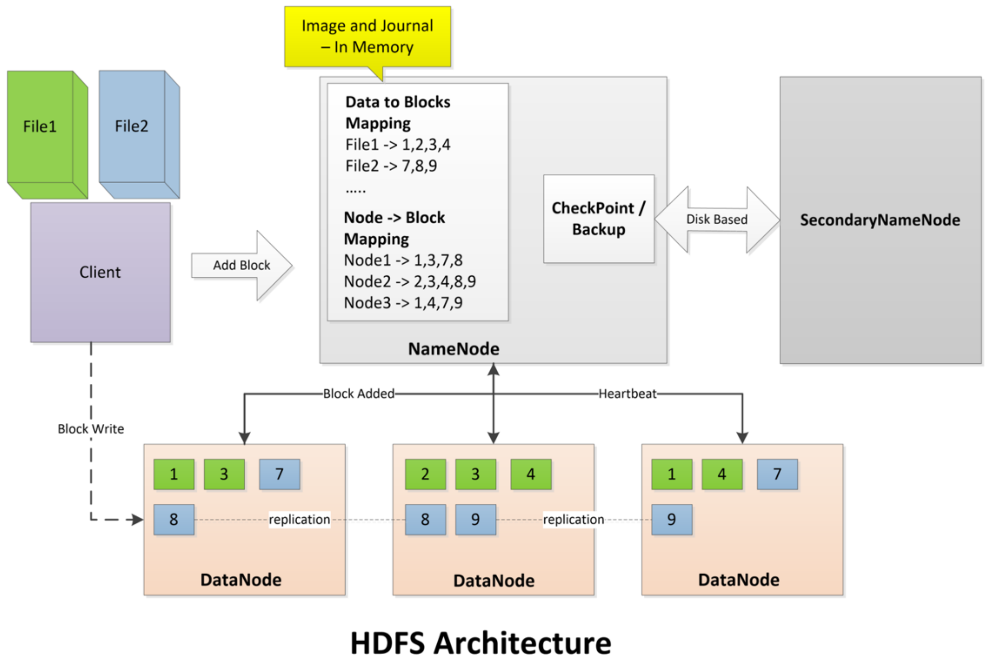


Figure 2 - HDFS architecture

As [7] points out, the three main services in HDFS are:

1. **DataNode:** This service resides within each node in the cluster and is in charge of storing the different blocks on disk.
2. **NameNode:** This service keeps track of all the different blocks and replicas, including attributes such as permissions, modification and access times, namespace and disk space quotas. In short, it works as an index to fetch a given file, block by block from the DataNodes.
3. **SecondaryNameNode:** This service is just a hot replica that is invoked whenever the main NameNode fails to respond to the HDFS client. This is to avoid any single point of failure in case the NameNode were to fail.

Files in HDFS are designed to be read-intensive. Because of this, once a file is written to HDFS, a user can only append data to it and not edit a specific portion, as it will involve a complex block redistribution. Instead, if a user wants to edit the file, he/she will have to modify it using MapReduce and write the result to a different location in HDFS. This immutability guarantees the system's efficiency.

Hadoop's popularity and adoption skyrocketed thanks to its open source philosophy, which allowed companies all over the world to experiment at no licensing cost.

A rich ecosystem was gradually built around Hadoop by the open source community to solve arising data use cases, derived from Hadoop's usage.

The following diagram outlines some of the initial open source components:

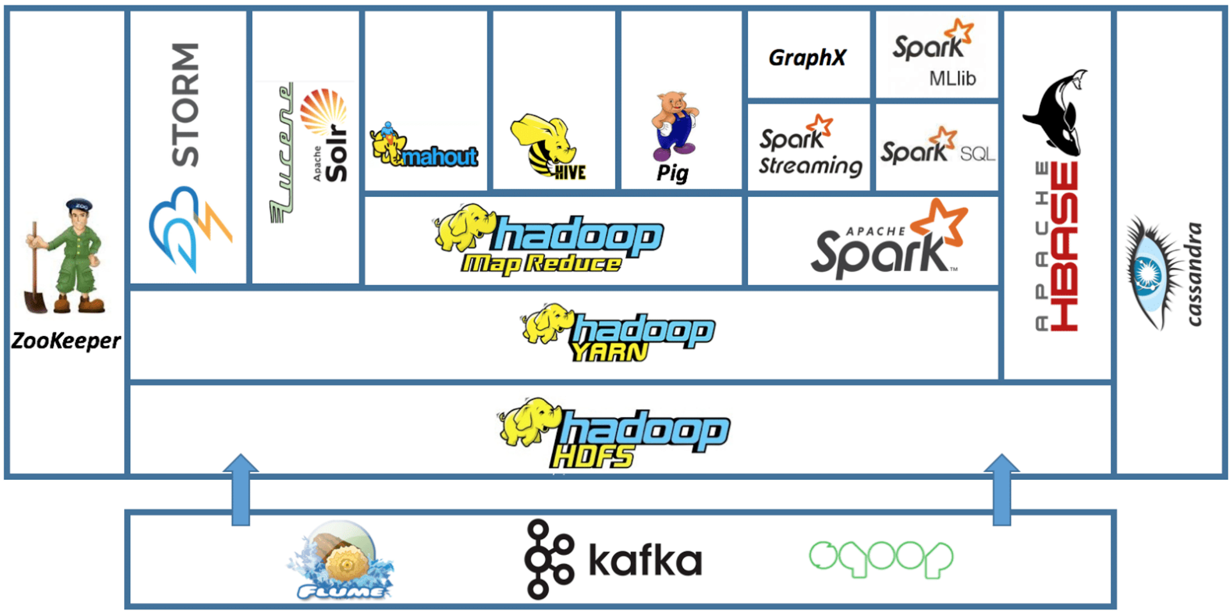


Figure 3 - Hadoop ecosystem [8]

As more industries convinced themselves of Hadoop's benefits, they started to invest time translating their already-existing SQL queries to MapReduce jobs and move data to HDFS. However, those jobs soon became unmanageable in consequence of the framework's inherent complexity. This key pain point was the one that ultimately led to the creation of Apache Hive.

3.5. Hive

Apache Hive is a data warehouse solution that facilitates reading, writing, and managing large datasets residing in distributed storage using SQL [9]. This meant that existing legacy queries could be translated to HiveQL easily to be able to analyze and perform aggregations on the data without the complexity of understanding MapReduce.

Hives structures data into well-known database concepts like tables, columns, rows and partitions. It supports all the major primitive types: integers, floats, doubles and strings; as well as complex types such as maps, lists and structures [10].

Hive creates a layer on top of the data in HDFS. This means that instead of creating a physical table, it only registers metadata about the underlying data whenever the user creates a table. In light of HDFS' architecture, HiveQL does not implement neither INSERTs nor UPDATEs, as the data is mostly write once read multiple times.

Hive is primarily focused on data warehousing, as it performs large file scans to perform complex aggregations and may even store intermediate results on other tables used for business analytics or business intelligence to build dashboards or generate reports. Such analytic process is termed OLAP (Online

Analytics Processing) and differs from OLTP (Online Transaction Processing) which is focused on extracting a given number of rows rather than aggregating on the entire dataset.

Metadata is written in what is known as the metastore. The metastore is fundamentally a central repository that contains information associated with Hive: databases, tables, etc. Multiple services, including Hive driver, uses this information every time a query is issued in order to effectively read the underlying data.

As stated in [11] Hive metastore can be configured in 3 different ways:

1. **Embedded Metastore:** This is the default mode. Hive will use a minimalist database engine, Derby, to store the metadata in the local filesystem. This mode is ideal for local unit testing, but it is far from ideal in production.
2. **Local Metastore:** In this mode, multiple Hive servers are expected to run on the same Hive Service Java Virtual Machine, and they all point to an external metastore database. This delegates the storage to a more robust system (RDBMS). This configuration uses either a JDBC or ODBC driver to connect to the external database.
3. **Remote Metastore:** This configuration uses different metastore servers, in completely separate JVMs, which in turn connect to the external metastore database. In this way, if other processes want to communicate with the metastore server, they can do so through the Thrift Network APIs. The ultimate goal of this configuration is to provide complete isolation between components so that security experts or DevOps engineers can firewall appropriately.

Given the flexibility of this metastore, it is commonly utilized nowadays as the data repository on top of HDFS-like storage (such as Amazon S3 or Google Cloud Storage). Tools like Apache Spark or Presto can then read from this metastore and perform processing in a more efficient fashion, as lots of metadata is stored inherently in the metastore tables.

At the time of writing, Hive supports the following databases:

- Derby
- MySQL
- MS SQL Server
- Oracle
- Postgres

An important downside of Hive is that execution of complex aggregation queries may take from several minutes to many hours. This because Hive translates SQL queries to MapReduce commands under the hood, which is far from ideal, as many MapReduce operations were unnecessarily writing to disk after each MapReduce stage.

3.6. Amazon Web Services (AWS)

In 2003, Chris Pinkham and Benjamin Black, both working at Amazon, presented an internal paper to describe the vision for a new infrastructure, completely automated and standardized. This paper included a proposal of selling virtual servers as a service [12].

After a few months later, multiple discussions and refinement took place which led to the initial concept of Elastic Compute Cloud (EC2).

In 2006, Amazon officially re-launched AWS with multiple base services, including: S3, SQS and EC2. Those services were designed with developers in mind. This allowed Amazon to convey the benefits of their usage across the companies and hence created immense momentum.

As of 2020, Amazon is the biggest cloud provider in the world with over 32 percent in market share [13]. Similarly, at the time of writing, AWS has about 175 services available and releases new ones on a nearly monthly basis [14].

It is absurd mentioning all services AWS has, but some of the most important categories are [15]:

- Analytics
- Application Integration
- Business Applications
- Compute
- Containers
- Customer Engagement
- Database
- Internet Of Things
- Machine Learning
- Management and Governance
- Mobile
- Media services
- Networking and Content Delivery
- Security, Identity and Compliance
- Storage

AWS represents a dramatic shift from the traditional paradigm of in-house data centers, as it delegates most of the operation to the provider. This involves moving from capital expense (CapEx) to operational expense (OpEx). In summary, this means that companies will now have to focus on investing on services and platforms rather than on buying hardware, commodities and spending on IT engineers to set everything up.

The benefits that AWS provides, at a high level, are [16]:

- Easiness to use
- Flexibility
- Cost-effectiveness
- Reliability
- Scalability and high performance
- Security

Finally, AWS' nimbleness has empowered companies all over the world to enhance their products at a quick pace. This is because developers have closely worked with the engineers who are using their products, listen to their struggles and incorporates either product improvements or new services that supplement what is already existing.

3.6.1. Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides secure, resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers [17]. Roughly speaking, it as a service that provides configurable virtual servers on demand.

EC2 provides a simple interface to choose from a wide range of operative systems, lets users configure security characteristics, amount of resources, and even keys to directly log into the instance. Thanks to AWS' infrastructure, instances typically take up to two minutes to spin up and can be terminated at any time to not incur into additional costs.

Finally, EC2 provides the concept of spot instances: the cheapest instances to which users offer a percentage of the original price. However, users may lose those instances once the bid price is over what user offered.

3.6.2. Amazon S3

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. Customer of all sizes and industries can use it to store and protect any amount of data for a range of use cases, such as websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics [18].

Given the fact that S3 is serverless, users can store data at any time and can rest assured their data will be durable, up to 99.99999999999% of the time, meaning that it is nearly impossible to lose data, as it is replicated across data centers, to avoid data corruption and degradation.

Amazon S3 provides different types of storage classes [18], including:

- **Standard** – The highest availability for frequently accessed data.
- **Standard Infrequent Access** – Less frequently accessed data.
- **One Zone Infrequent Access** – Less frequently accessed data, only replicated in one datacenter.
- **Glacier** – Once a month accessed data
- **Glacier Deep Archive** – Once a year accessed data

3.6.3. Amazon IAM

AWS Identity and Access Management (IAM) enables you to manage access to AWS services and resources securely. Using IAM, you can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources [19].

IAM's basic resource is the policy, a JSON document that describes specific actions and resources that can accessed (e.g., access to create instances on EC2 or delete objects from an S3 bucket). Policies can be attached and removed in runtime to allow for automated deployments that reflect changes on near-real time.

IAM policies can be attached to:

- **Users** – a person in the company.
- **Groups** – A collection of users.
- **Roles** – A unique container that can further be attached to AWS services such as EC2.

3.6.4. Amazon VPC

Amazon Virtual Private Cloud (Amazon VPC) lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. You have complete control over your virtual networking environment, including selection of your own IP address range, creation of subnets, and configuration of route tables and network gateways [20].

Amazon VPC works with all compute and database services, in order to rule out unwanted access. In particular, EC2 has an easy integration, so that virtual servers can be deployed using well-known access rules, limiting even the inbound and outbound traffic.

3.6.5. Amazon EMR

Amazon EMR is the industry-leading cloud big data platform for processing vast amounts of data using open source tools such as Apache Spark, Apache Hive, Apache HBase, Apache Flink, Apache Hudi, and Presto. With EMR you can run Petabyte-scale analysis at less than half of the cost of traditional on-premises solutions and over 3x faster than standard Apache Spark [21].

Under the hood, Amazon EMR will deploy a configurable number of EC2 instances already pre-configured to work as a single cluster.

Amazon EMR works well with spot instances, given that most open source tools support losing nodes as part of their architecture. So, by using spot instances, users can have flexibility of autoscaling at lower costs.

3.6.6. AWS Glue

AWS Glue is a fully managed extract, transform, and load (ETL) service that makes it easy for customers to prepare and load their data for analytics. AWS Glue can be configured to discover data sources' metadata (e.g. table definition and schema) using the AWS Glue Data Catalog [22].

Developers working with AWS can benefit from AWS Glue Data Catalog as a serverless alternative of the Hive metastore. For Amazon EMR, all users need to do is configure the cluster to point to the metastore and have IAM role permissions to read/write Glue Data Catalog. Once that is configured and the cluster is running, any Spark application will be able to read and write tables to the Glue Data Catalog [23].

One important feature of the Glue Data Catalog is that all tables have automatic visibility on Amazon Athena, so that right after the table shows up in the catalog, analysts can start running queries on top of the data.

3.6.7. Amazon Athena

Amazon Athena is an interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL. Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries that you run [24].

Athena lets you point to your data in Amazon S3, define the schema, and start querying using standard SQL. Most results are delivered within seconds. With Athena, there's no need for complex ETL jobs to prepare your data for analysis. This makes it easy for anyone with SQL skills to quickly analyze large-scale datasets.

3.7. Airflow

Analogous to Apache Oozie, Apache Airflow is also a platform to programmatically author, schedule and monitor workflows [25]. It shares the concept of DAG, which consists of a collection of tasks to run, organized in a way that reflects their relationships and dependencies and a scheduling time, using CRON expressions.

For any given task, Airflow facilitates Operators which can range from a data warehouse query to a cluster execution to a simple python callable execution. To name a few of the supported ones: Bash, Python, Email, SimpleHttp, MySql, Postgres, Docker, Hive, S3FileTransform, BigQuery, Slack [26].

Airflow can be divided in 3 individual services:

1. **Webserver**: Provides a rich UI where users can visualize, monitor, execute and stop DAGs and/or individual tasks.
2. **Scheduler**: Coordinates the tasks execution, whether is a local execution or a distributed execution using celery.
3. **Worker**: Process that performs the tasks the scheduler has queued up.

All DAGs are created using simple python scripts that fundamentally specify a DAG, an arbitrary number of operators, relationships (or upstream dependencies). Once copied the script into airflow's DAG folder, the scheduler periodically scans the DAG folder and incorporates it, which should be visible through the webserver user interface.

Given the maturity and broad adoption of this tool, advanced features have been introduced, including inter-DAG dependencies using sensors, encrypted connection to a myriad of databases and/or clusters, resources pools, etc.

3.8. Spark

Apache Spark is a distributed compute engine used to leverage the full parallelism of every node in a cluster in an efficient manner. This was possible thanks to its simple yet clever architecture which analyzed the main architecture pain points of MapReduce and refactored them. Some benchmarks even demonstrate that Apache Spark is up to 100 times faster than a typical MapReduce job on Hadoop [27] [28].

To achieve such efficiency, Apache Sparks relies on in-memory computation. This allows both driver and workers to perform all operations in Random Access Memory (RAM) and keep the results in memory until they need to flush it somewhere. Moreover, by dynamically building a DAG, in a similar fashion to Apache Tez, Spark is capable of keeping track of all operations and hence delete redundant vertices.

Spark has 4 high-level packages atop of its stack. Those packages are:

1. **Spark SQL:** To interact with structured data using ANSI SQL.
2. **Spark Streaming:** To process live streams of data.
3. **MLlib:** To execute machine learning techniques using structured and semi-structured data.

4. **GraphX:** To manipulate graphs in a distributed fashion.

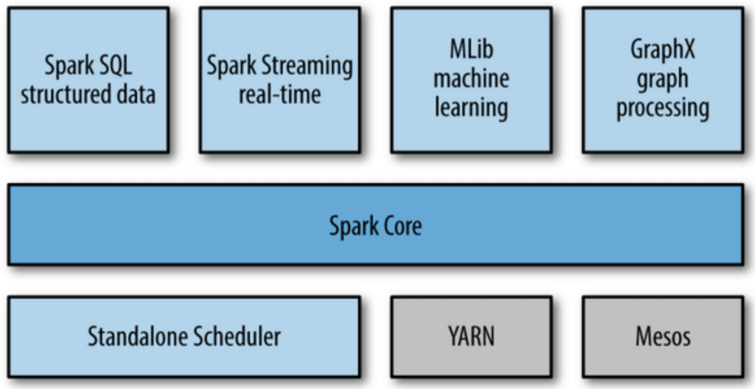


Figure 4 - Spark Modules [29]

As shown in the figure above, Spark relies on a cluster scheduler, or resource negotiator, to request nodes for up and down scaling. Such cluster managers were designed to be plug and play, meaning that Spark can work with any of the following:

1. **Standalone.** The default one when running Spark locally.
2. **YARN.** Hadoop's resource negotiator.
3. **Mesos.** Distributed cluster manager.
4. **Kubernetes.** Cluster orchestrator.

Another key feature of Spark is the separation of functions into two main categories: transformations and actions. Transformations specify a number of steps in which the input data will be changed in a particular way; this will not trigger any distributed computation, but rather will accumulate on Spark's logical plan for future execution, this is called lazy evaluation. Actions on the other hand represent cues for Spark to run the transformations according to the logical plan it assembled along the way.

Spark coins the term Resilient Distributed Dataset (RDD) which, as the name implies, is a distributed collection of structured elements. Since this collection is represented as a typical object, the programmer does not have to worry about complex tuning or any composition to make RDDs work. It works just the same as a collection, with the only difference being that the actual execution takes place on a set of nodes on a cluster rather than in the master [27].

This simplicity allows users to quickly understand data lineage and run complex queries, including merging from a variety of sources and formats, all within a few lines of code.

Latest versions of Spark offer two more sophisticated structures built on top of: Datasets and DataFrames. They allow users to impose data types on all columns of a dataset, support almost native ANSI SQL and offer a more standard interface to read, aggregate and write data [30].

Spark uses a master node and an arbitrary number of worker nodes. Inside those worker nodes a number of executors are allocated. Those executors interface directly with the SparkContext to get information about the task to be executed: where to find the data, what code to execute, etc. This architecture is depicted as follows:

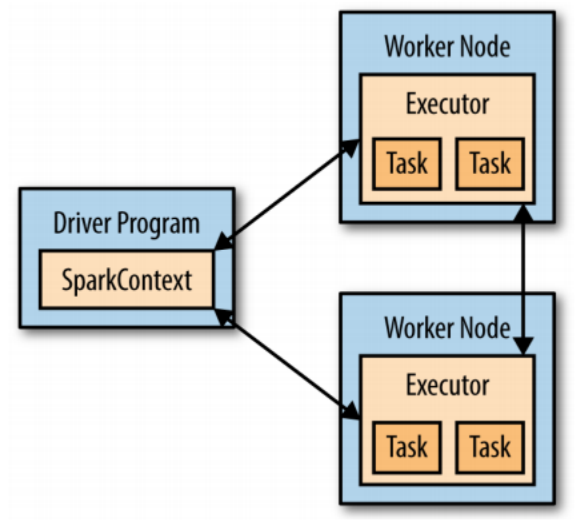


Figure 5 - Components for distributed execution in Spark [29]

3.9. Arrow

Apache Arrow [31] is a cross-language development platform for in-memory big data applications. It specifies a columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware.

The technology is extremely fast for data interchange between services, as there is no overhead in serializing and de-serializing data structures.

For instance, one prominent use case for Arrow is moving PySpark data to Pandas/NumPy. When Spark Dataframes are converted to Pandas Dataframes, the Spark driver first collects all rows, then each row is serialized into Python's pickle format and sent to Python worker processes. This child process, in turn, unpickles each each row into a list of tuples, which are ultimately used to generate the Pandas Dataframe [32].

3.10. Parquet

Parquet [33] is a columnar storage format to support complex and nested data types out of the box. The format was developed by Cloudera and Twitter to work with multi-column datasets. It uses the record shredding and assembly algorithm described in the Dremel paper to simple flattening nested name spaces.

Unlike CSV and JSON, Parquet files are binary files that contain metadata about their content. In this way, without reading/parsing the content of the file(s), Spark can just rely on the metadata to determine column names, compression/encodings, data types and even some basic statistics. The column metadata for a Parquet file is stored at the end of the file, which allows for fast, one-pass writing [34].

Parquet files are immutable, following HDFS' principles and do not natively support schema evolution. Spark can nonetheless merge the schema of multiple parquet directories, but user has to specify a read option for that to occur.

Parquet is optimized for Write Once Read Many (WORM) paradigm—it is slow to write, but incredibly fast to read, especially when you're only accessing a subset of the total columns.

Finally, Parquet natively supports both predicate (or filter) pushdown and projection pushdown [34].

Predicate pushdown is a technique wherein parts of the queries, the filters, are pushed directly to where the data is stored. This means that instead of reading the entire file to memory and executing the filter condition to discard rows, Parquet will only return those values that meet the predicate criteria, saving lots of memory and processing time.

Similar to the predicate pushdown, the projection pushdown is a technique that leverages the columnar format, so that when queries specify a subset of columns, the format only scans and returns those columns, which results in better I/O performance.

The following figure illustrates both predicate and projection pushdown:

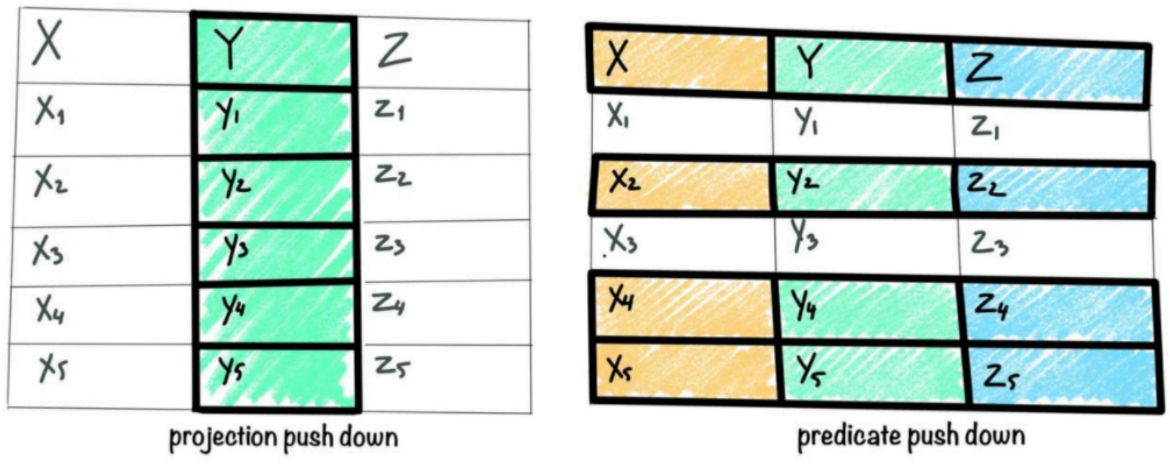


Figure 6 - Project and predicate pushdown [34]

3.11. Presto

Presto is an open source distributed SQL query engine that supports both relational data sources (including MySQL, PostgreSQL, Amazon Redshift and Teradata) and non-relational data sources (including HDFS, Amazon S3, Cassandra and MongoDB) [35]. Presto has been successfully tested over 300PB of data at Facebook.

Some of the most important characteristics of Presto include [36]:

- It is a multi-tenant system capable of concurrently running hundreds of memory, I/O and CPU-intensive queries and scaling.
- Its design allows administrators to setup clusters to query data from various data sources within the same cluster.
- It can be configured to support many use cases with different constraints and performance characteristics.

Presto is well-known as one of the best tools for interactive analysis (e.g., handling hundred GB to a few TB of data) and can easily be connected to BI technologies for dashboard generation.

Other Presto use cases include batch ETL, A/B Testing, and developer/advertiser analytics.

At the time of writing, Amazon Web Services offers a product based on Presto, named Athena, which connects to the Hive metastore of another service called Glue.

3.12. Atlas

Apache Atlas [37] is data governance and metadata management solution. It is comprised by a set of services and helps users discover and classify datasets so that it becomes easy to pinpoint the source(s) of any given table and what transformations led to it.

Atlas includes the following main features:

- Metadata types and instances
- Classification
- Lineage
- Search/Discovery
- Security and Data Masking

Atlas leans on other Hadoop-related technologies, as observed in the following architecture diagram:

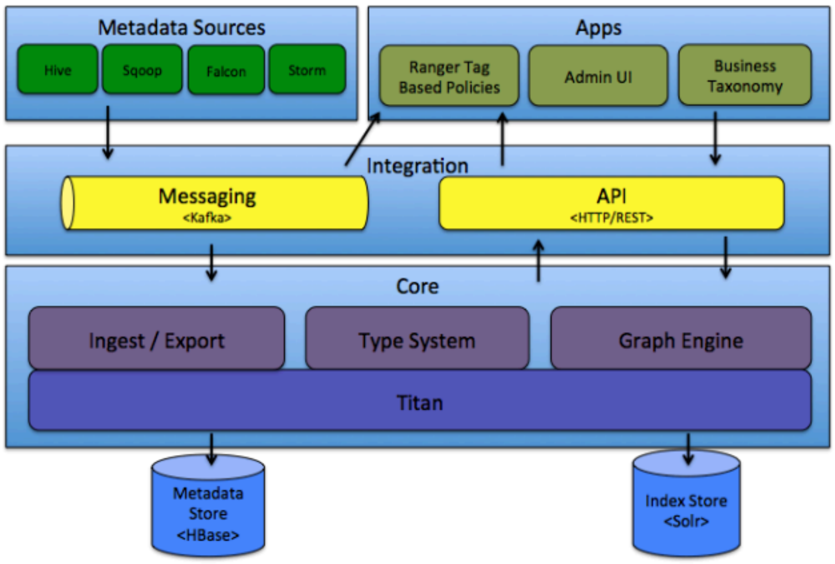


Figure 7 - Apache Atlas architecture [38]

The system collects metadata directly from sources such as Hive, and Sqoop; when a user creates a new table on Hive, it automatically registers the action and populates the data lineage graph, which is shown below:

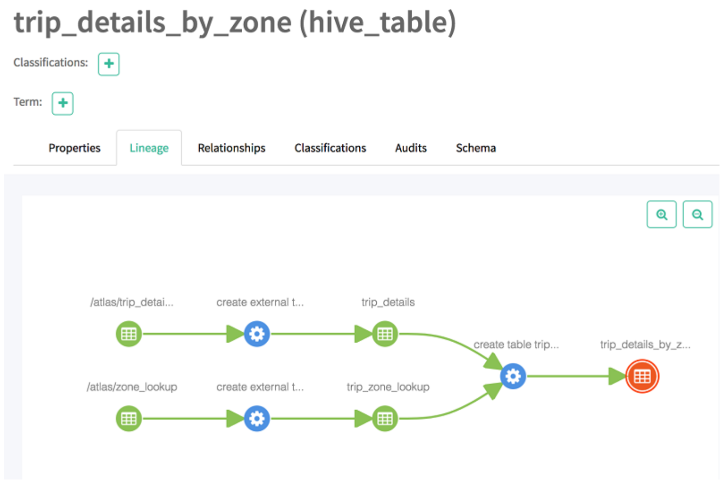


Figure 8 - Atlas lineage

Atlas can only read metadata from Hive, Falcon, Storm and Sqoop. Unlike Hive, the rest of the tools are rarely used in the industry nowadays and therefore Atlas hinders engineers to run Hive queries in order to create new tables, which is impractical; frameworks such as Apache Spark are preferred for data manipulation and data set creation.

3.13. Spline

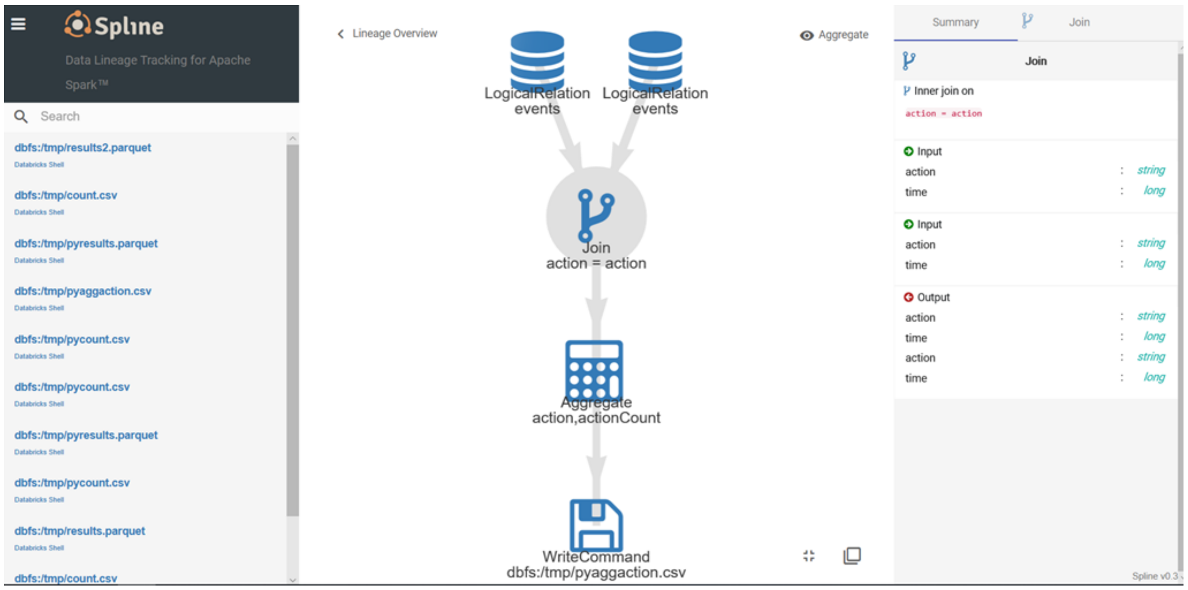
Spline [39] is a data lineage tracking tool designed to work with Spark out of the box. The tool leverages the broad adoption of Apache Spark in the industry not only as a standalone service, but as the backbone of other services such as Hive and Apache Mahout [40].

Spline is designed with three principles in mind:

1. **Lightweight** – no heavy computation overhead for spark execution
2. **Unobtrusive** – if anything goes wrong while computing the data lineage, the spark job will not be affected
3. **Easy to use** – preferably be able to initialize with a single line of code

Spline's clever design leverages Spark internal execution plan to extract information about the datasets being read, the transformations applied, and the destination(s) so that the actual graph with the lineage is automatically stored, once the datasets are written.

The following figure shows an example of a data lineage graph in the Spline web UI:



dbfs:/tmp/results2.parquet
Databricks Shell
dbfs:/tmp/count.csv
Databricks Shell
dbfs:/tmp/pyresults.parquet
Databricks Shell
dbfs:/tmp/pyaggaction.csv
Databricks Shell
dbfs:/tmp/pycount.csv
Databricks Shell
dbfs:/tmp/pycount.csv
Databricks Shell
dbfs:/tmp/pyresults.parquet
Databricks Shell
dbfs:/tmp/pycount.csv
Databricks Shell
dbfs:/tmp/results.parquet
Databricks Shell
dbfs:/tmp/count.csv

Join

Inner join on
`action = action`

Input
action : string
time : long

Input
action : string
time : long

Output
action : string
time : long
action : string
time : long

Figure 9 - Data Lineage in Spline UI

Splines stores the metadata in a database (similar to the Hive metastore). Thanks to this, Spline is highly resilient by storing the lineage information in a separate system, allowing the agent to restart in case of a failure or the system to create new instances and pointing them to the metastore.

At the time of writing, Spline supports the following metastores:

- **MongoDB:** Primarily for testing purposes.
- **ArangoDB:** To leverage the full potential and reliability of Spline.

3.14. Terraform

Terraform is an IaC tool created by Hashicorp. It is used for building, changing and versioning infrastructure safely and efficiently [41]. It works with most of the top cloud providers and provides a rich syntax that allows users to deploy complex architectures in a reproducible and descriptive way.

Terraform uses configuration files written in Hashicorp Configuration Language (HCL) to describe infrastructure components—the syntax of which resembles a typical JSON format, but with support to advanced features such as looping and dependency injection.

The main advantage of using Terraform in comparison to other cloud provider specific tools is that companies can avoid locking into the provider. This means that whenever companies need to move their infrastructure to a different cloud provider, they just need to create the equivalent modules in other Terraform repository and execute terraform apply. The rest is taken care of by Terraform.

One of the most advanced design principles of terraform syntax is that it is declarative rather than imperative, i.e., every time a user executes terraform apply, terraform will look up the difference between the actual state and the desired state and will plan the updates, deletes and creates in the optimal order, making the deployment of new infrastructure components, seamlessly.

Often times, people think of Terraform as an alternative or substitute for configuration management tools such as Ansible, Puppet, etc. But, on the contrary, they tend to complement each other just fine. For instance, users can first lay the infrastructure groundwork using Terraform; once that is ready, they can use Ansible to configure and install dependencies for a fresh start.

3.15. Git

Git is an open source distributed version control system [42]. Version control helps software developers supervise code changes of an artifact or product incrementally over time. Thanks to this, several developers can work on the same project at the same time, without interfering with each other's changes.

Git uses the term repository to refer to a file system folder that contains source code to keep track of. The user simply has to initialize a repository running a git init command on the folder to monitor, and git takes care of the rest. Once initialized, the repository will automatically detect deltas on all files and users can include specific pieces of those deltas to create commits.

A commit is an object that represents a set of changes on a given repository. Git forms these commits by recording the deltas of each file, the parent commit's reference, as well as information related to the author, the commiter, the commit message and timestamp of creation. To represent a commit, Git uses a SHA1 hash of all gathered data to ensure uniqueness.

To further organize and allow for more fluent working environments, Git creates the concept of branches. Branches are independent work lines in that can coexist without interfering with each other. By default, Git uses master as the first branch, but users can create as many as they want. A typical workflow involves a few dozen branches to keep track of different features being developed on the repository.

Git streamlines storage by only storing deltas physically in the *.git* folder, rather than complete code snapshots, in comparison to previous version control tools.

The following figure depicts a typical Git workflow:

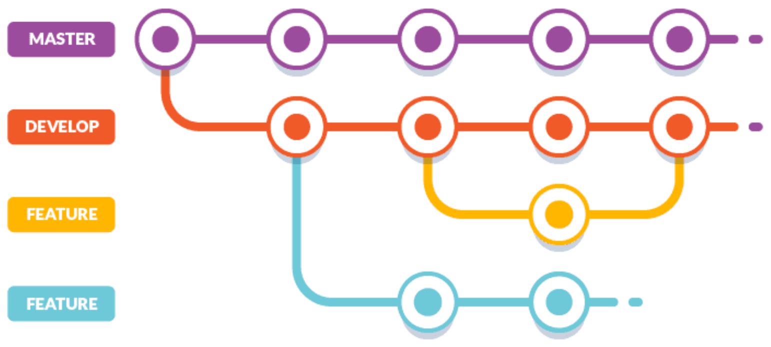


Figure 10 - Git workflow [43]

3.16. Docker

Docker is an open source tool for developing, shipping, and running various applications in a fast, secure and isolated way [44]. Docker enables the applications to run separately from the host infrastructure and treat the infrastructure like a managed application. This is achieved by deploying applications in what is known as containers.

Containers leverage two Linux features: kernel namespaces and CGROUPS; which allow containers to execute independently within a single host without interfering with each other.

Containers share the same underlying kernel and each container is constrained to only use a defined portion of resources available in the host machine, unlike traditional virtual machines, which inexorably demand an operating system [44]. In consequence, if the Linux kernel fails, all containers running on that host will certainly go belly up.

Containers are extremely lightweight granted that they only require a small software layer on top of the kernel. For instance, an entire Ubuntu VM installation may be well beyond 5 GB in size, whereas the Alpine image is around 5 MB in size.

To import a Docker image, users either pull existing ones from Dockerhub—the official repository for pre-built images—or they can create their own using a Dockerfile, that dictates a recipe for building that image. Once the image is done, Docker can use it to create any number of containers.

Even though containers work isolated from the external world, they commonly depend upon interaction with the external world, e.g., exposing a port for a web server, or interacting with files from a given mount point. For the latter case, Docker features volumes.

Docker volumes are named filesystem trees managed by Docker. Essentially, the user, or the script for that matter, only specifies the mount point from the local filesystem and the container path where that mount point will be visible, along with any file permissions. This is of the utmost importance since containers are ephemeral in nature; so, once they are deleted, all data produced within will be wiped out.

In light of all these benefits, architects and developers have been either migrating their existing microservices to containers, or designing new ones using them as backbone. However, keeping track and deploying a myriad of services becomes cumbersome and error prone on the long run. Docker provides a new level of abstraction called Docker Compose to solve this.

Docker compose is a high-level declarative abstraction, which uses a YAML file to represent all services, volumes, networks, configurations and even start up dependencies (e.g., service A requires service B to start properly). Docker compose figures out a way to deploy them either locally or on a cluster [45].

3.17. Kubernetes

Kubernetes is an open source orchestrator for deploying containerized applications [46]. It was original developed by Google and inspired by a decade of experience deploying scalable, reliable systems in containers. It radically changes the way applications are developed and deployed in the cloud.

As a Kubernetes user, you can define how your application should run and the ways they should be able to interact with other applications. Users can easily scale services up and down, perform graceful rolling updates and switch traffic between versions of applications to test features, or rollback problematic deployments [47]. This is also known as blue green deployment.

Kubernetes uses a master server that coordinates the deployment of the containers in the cluster and is responsible for monitoring and deciding where to allocate containers in an optimal way.

Rather than using simple containers, Kubernetes defines pods as its basic abstraction mechanism. A pod can be thought as either one or more intimately related services that come packed together and can reach each other natively.

Kubernetes supports a myriad of other advanced features, some of which are outlined below [47]:

- **Replication set:** A feature to scale and monitor health on pods. These are used to increase the number of pods given a certain life cycle rule.
- **Stateful set:** Specialized pod controller that offers ordering and unique guarantees, so that pods can be finely orchestrated.
- **Ingress controller:** A high level abstraction that allows simple host or URL based HTTP routing. The idea is that the outside world can have access to specific services in the cluster.

4. RELEVANT CONCEPTS

4.1. SQL

The Structured Query Language (SQL) has been the de facto domain language to read and aggregate databases for nearly 5 decades [48] and was developed in 1970 by IBM researchers [49]. SQL allows users to transform and query data sets using an expressive syntax that closely resembles set theory concepts, rather than a typical programming language such as C or Java.

SQL is a language to read, insert, update and delete fields in Relational Database Management Systems (RDBMS) (until Dremel and all NoSQL databases came into existence). RDBMS organize data in a hierarchical way, starting from the concept of database. A database is a high-level abstraction that encompasses a group of related tables. Tables in turn are a data set with a defined schema that are composed by columns and rows.

Queries can be split into two categories, depending on their purpose [50]:

- Data Manipulation Language (DML): SELECT, INSERT, UPDATE and DELETE
- Data Definition Language (DDL): CREATE, ALTER, DROP

RDBMS, and by extension traditional SQL systems, were originally designed with individual machines in mind, as the underlying data structures that SQL engines utilize are optimized for concurrency in a multi-threaded environment to comply with ACID transactions (Atomicity, Consistency, Isolation and Durability) [51]. Once the data overpasses certain volume, in the order of billions of rows, SQL queries become slow and computationally inefficient.

4.2. Data Warehouse

A Data Warehouse (DW) or Enterprise Data Warehouse (EDW) is a system to collect and manage data from a variety of sources. It is commonly used as the backbone behind BI systems for analysis and reporting [52].

Data in a warehouse is commonly historical, consolidated, time variant non-volatile, integrated and subject-oriented [53].

Data warehouses typically provide high performant storage as well as optimizations in performing calculations and aggregations at scale [54].

Unlike any other Relational Database Management System (RDBMS), capable of Create, Read, Update and Delete (CRUD) process, data warehouses perform Reads using a normalized data model in the start scheme [54] and involve a much more complex architecture.

4.3. ETL

The process of creating new tables in a data warehouse is often referred to as Extract, Transform and Load (ETL). It describes the process of reading data from an arbitrary data source (e.g., a Data Warehouse, a relational database, files in a distributed object store); applying some aggregation or enrichment such as cleaning, concatenating, grouping, etc.; and storing the result in a destination (similar to the ones observed as data sources) [55].

One of the key considerations while choosing the ideal ETL tool is the data connectors available. A data connector refers to the ability for a tool to read data from a specific system natively. In most modern tools, connectors could easily be a NoSQL database, a data warehouse or a basic RDBMS.

As described by [56], ETL systems add value to the data, as they:

- Remove mistakes and correct missing data
- Provide documented measures of confidence in data
- Capture the flow of transactional data for safekeeping
- Adjust data from multiple sources to be used together
- Structure data to be usable by end-user tools

As the complexity of the transformations grows, the latency between the data acquisition and results availability increases considerably. Furthermore, whenever the tool finds an exception while transforming the data, the processing tool will simply fail, and the data will not be stored at all.

4.4. Vertical and horizontal scaling

Vertical scaling is the process of increasing a server’s capacity, either the memory size or the storage.

This was possible thanks to the evolution of CPUs and the steady transistor size reduction. Unfortunately, this trend was rapidly coming to an end. Researchers found out that CPU clock frequencies directly influence heat radiation on the device, which will eventually reach the device’s maximum temperature, hence downgrading the microcontroller’s performance [57].

Moreover, despite the huge efforts of hardware designers, the transistor’s size cannot be infinitely reduced; at some point, transistors are so small that quantum effects prevent them from working properly [58]. All this forecasted that vertical scaling would have severe limitations, beside costs, in the near future.

Intel and AMD came up with revolutionary CPUs featuring multiple cores within a single chip, as well as optimized CPU cache. This allowed programs to perform operations in true parallelism and effectively fetch frequently used data.

Horizontal scaling alleviates the above-mentioned challenges by interconnecting commodity hardware in a cluster so that they can work as a single system. This means that, in order to scale, a system administrator or IT engineer will only have to attach more computers to the network and the system will automatically pick up the new capacity [59].

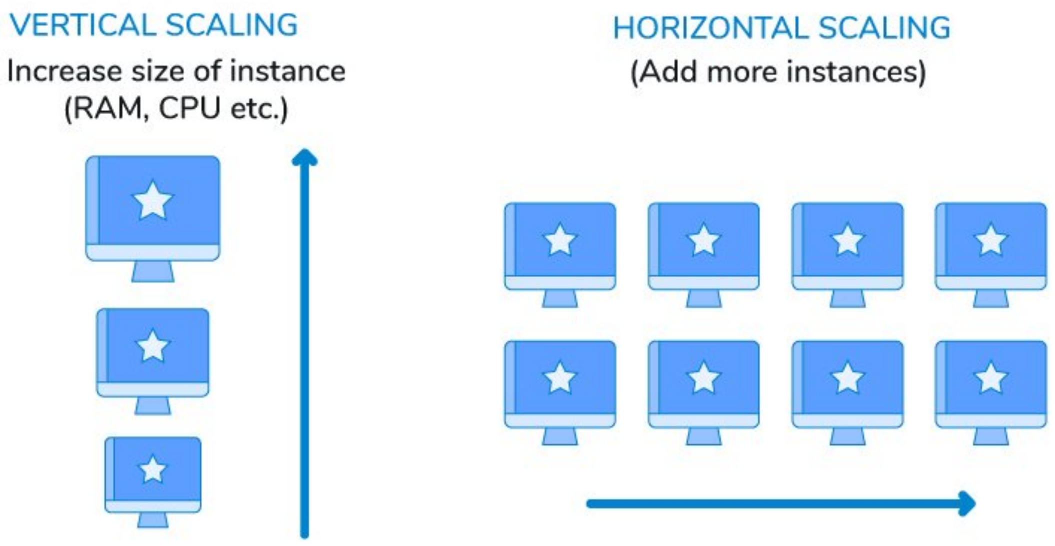


Figure 11 - Difference between horizontal and vertical scaling [60]

4.5. Distributed systems

The evolution of computer systems has come a long way since their inception but has seen a tremendous paradigm shift in the XXI century. Prior to that, computers that were purpose-built became general-purpose and so many different applications could run on a single machine by sharing an operating system that coordinated the execution.

Gradually, machines were networked together, and client-server architecture were born. This meant that commodity computers could be connected to a mainframe to execute processing: the clients made requests and the server dispatched them [61].

In the early 2000s, the rise of internet and large-scale datacenters consisting of thousands of low-cost commodity computers networked together gave rise to the development of distributed systems [61].

The term distributed system describes many commodity computers interconnected and acting as a single system. In essence, a distributed system leverages horizontal scaling to increase capacity.

One important shift from vertical scaling mentality is that processes need to dive into smaller pieces, depending on the data size, so that each computer can work independently on a piece. In the end, the master or coordinator node will take care of orchestrating the whole operation and instruct the slaves to dump the data elsewhere [62].

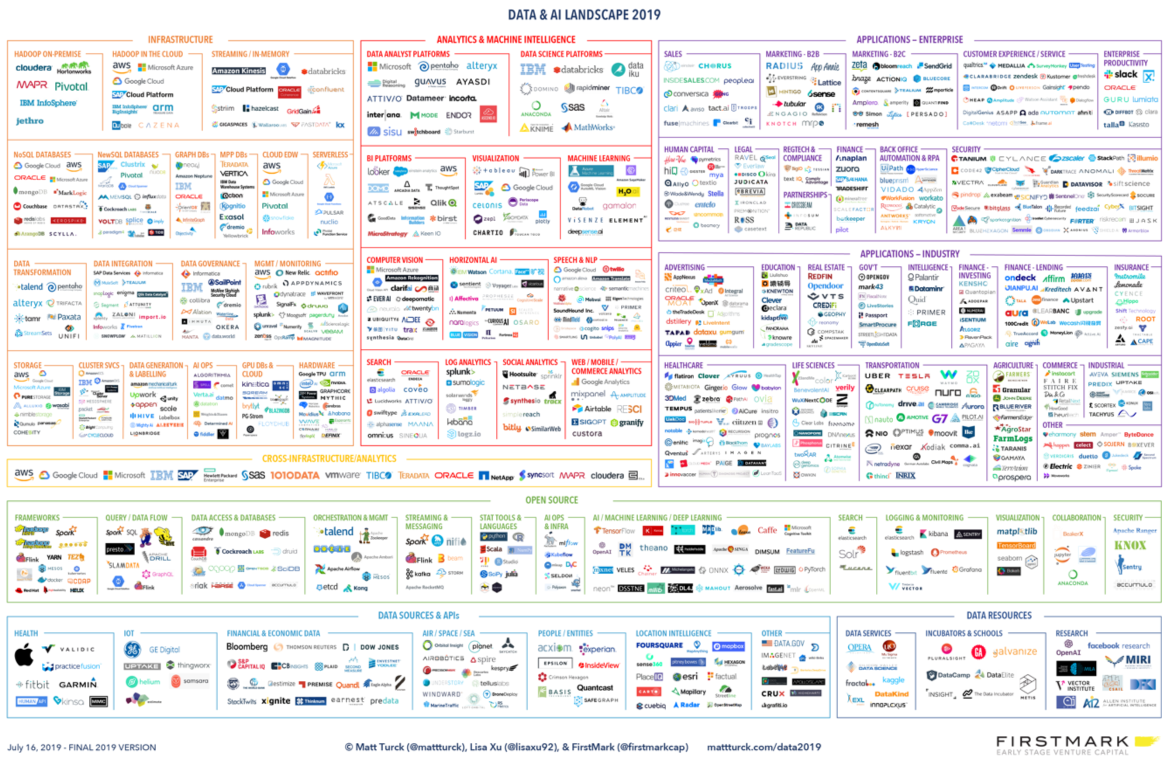
4.6. Big Data

In 2005, Roger Mougalas from O'Reilly Media coined the term Big Data for the first time. It broadly refers to a wide range of large data sets almost impossible to manage and process using traditional data management tools—due to their size, but also their complexity [63].

Over the years, the term has been refined and nowadays scholars as well as industry experts identify 5 characteristics that any Big Data project has in common:

1. **Velocity** – Speed at which data should be processed
2. **Volume** – The amount of data
3. **Value** – The insights that can be extracted from raw data
4. **Variety** – The different formats that can be handled
5. **Veracity** – Trustworthiness or accuracy of the data [62]

To give the reader an idea of the Big Data ecosystem, in 2019 the following infographic showed the most popular tools, grouped by category.



DATA & AI LANDSCAPE 2019
INFRASTRUCTURE
ANALYTICS & MACHINE INTELLIGENCE
APPLICATIONS – ENTERPRISE
APPLICATIONS – INDUSTRY
CROSS-INFRASTRUCTURE/ANALYTICS
OPEN SOURCE
DATA SOURCES & APIs
DATA RESOURCES
July 16, 2019 - FINAL 2019 VERSION
© Matt Turck (@mattturck), Lisa Xu (@lisaxu92), & FirstMark (@firstmarkcap)
mattturck.com/data2019
FIRSTMARK
EARLY STAGE VENTURE CAPITAL

Figure 12 - Big data landscape [26]

4.7. NoSQL

SQL has been the cornerstone of many processing tools and data analysis/business intelligence tools. SQL databases are ideal to store information with a well-known schema in a per-row basis. ACID consistency allows users to segregate data in different tables (fact and dimension tables) so that all data is normalized and stored as efficiently as possible.

However, as data started to proliferate, and grow in complexity, thanks to IoT devices, APIs and customer telemetry; SQL proved not to be nimble and flexible enough to accommodate such data. For instance, if a given table required a schema update to incorporate a new column, users would need to update the whole table, even if the prior record did not require so. This generated problems while wrangling the data, as NULL values are the default for the newly created column. In consequence, downstream logic must handle those special values in a graceful way, which often cripples the development of new features in an Agile manner.

To tackle all those limitations, NoSQL philosophy came into existence. NoSQL databases systems are non-relational databases uniquely intended to give high accessibility, reliability, and scalability for enormous data. NoSQL databases can store unstructured data such as email, multimedia, documents, and social media with high performance and very low latencies [64].

NoSQL flexibility allows for schema evolution by not limiting the user to specify it once a collection or a table is created.

As described in [5], NoSQL databases can be divided into four main categories:

1. Key-value stores
2. Columnar or wide-column stores
3. Document databases
4. Graph databases

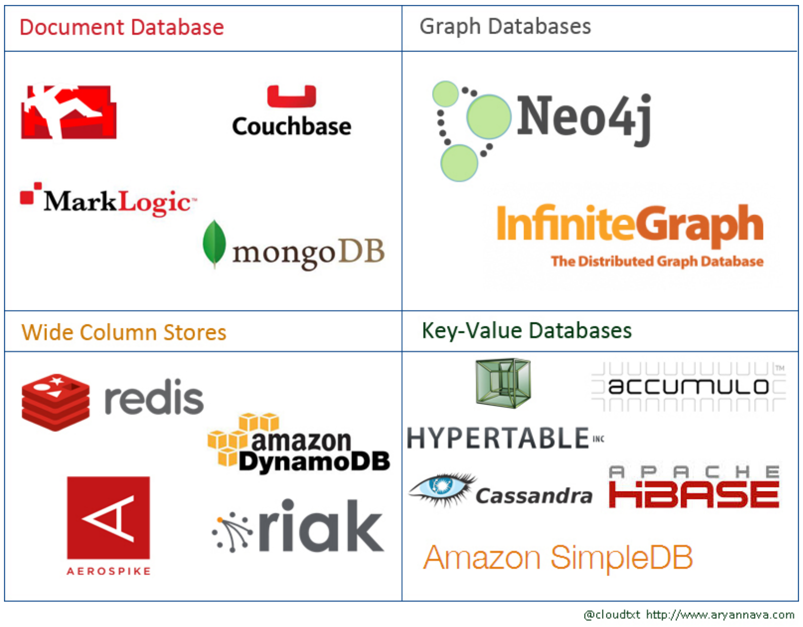


Figure 13 - Different NoSQL databases [65]

Key-value databases are good to represent simple and close-to structured data that can be queried with a unique key. Some examples are: Redis, Riak and Voldemort.

Columnar datastores are used in a similar fashion as SQL databases, since they can be thought of as a table with columns. One key difference is that instead of having NULLs in a column when a record does not have the value, the whole column is missing. This allows to save lots of storage and also improve the read speed. These databases are optimized for large analytics when only a set of columns are required, as they organize same columns within the same file. This allows to scan files quicker and retrieve huge volumes of data efficiently. Also, compression is easier to perform on these records as they are all of the same type within the same column. Some examples include Cassandra and HBase.

Document databases are somehow similar to key-value databases as they use a unique id to represent a document, however document databases are capable of storing much more complex structures and the documents can be enriched with new fields at any time. This enables users to evolve schemas as soon as new properties are discovered for a particular document. The most prominent example is MongoDB.

Graph databases are the least common type of database used. They represent records with nodes or vertexes on a graph and each record may be related, connected to another via an edge, to a bunch of other records. This type of relationship is found frequently on social media where an individual has multiple friends and each of them have their own set of friends. One example is Neo4J.

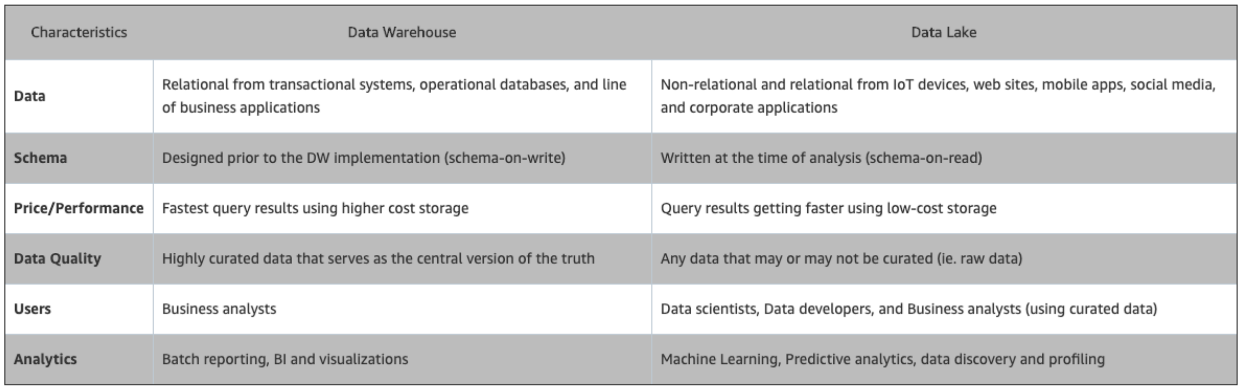
Although the use cases for NoSQL are increasing and the popularity is surging, NoSQL is by no means a SQL replacement. In fact, as with most technologies in the Big Data world, NoSQL can go hand-in-hand with SQL datastores, each one storing a specific type of data. Distributed processing tools can then take the data from multiple sources and aggregate it to generate rich reports and visualizations.

4.8. Data Lake

A Data lake represents a data storage in which all formats are permitted. Usually, they store files from different sizes and formats and is commonly replicated across data centers for further fault tolerance. It is the analogous of a local filesystem in which files are the abstraction to represent some piece of information. As the description implies, there is no restriction to what type of data it may contain [62].

Data lakes are very good for data exploration and getting new insights from unexplored data, as it allows to stash in any data and analyze it in the future. As such, Data lakes are frequently the starting point for onboarding data from the outside world or an application.

The following figure compares data warehouses and data lakes:



Characteristics	Data Warehouse	Data Lake
Data	Relational from transactional systems, operational databases, and line of business applications	Non-relational and relational from IoT devices, web sites, mobile apps, social media, and corporate applications
Schema	Designed prior to the DW implementation (schema-on-write)	Written at the time of analysis (schema-on-read)
Price/Performance	Fastest query results using higher cost storage	Query results getting faster using low-cost storage
Data Quality	Highly curated data that serves as the central version of the truth	Any data that may or may not be curated (ie. raw data)
Users	Business analysts	Data scientists, Data developers, and Business analysts (using curated data)
Analytics	Batch reporting, BI and visualizations	Machine Learning, Predictive analytics, data discovery and profiling

Table 2 - Data warehouse vs data lake comparison [66]

4.9. ELT

Extract, Load and Transform (ELT) processes are a simple yet clever variation of ETLs. ELTs are better suited for Data Lakes, whereas ETLs are almost always used to feed Data Warehouses. ELT process extracts data in the same way as in the ETL approach. Then, the data is loaded into the target Data Lake or Warehouse. Once loaded, the transformations and business logics are applied using native SQL drivers (for Data Warehouses) or in-memory transformations (with Apache Spark, for instance). This helps saving costs by pushing down the transformations to the distributed compute engine and ensures data is checkpointed first, making the whole process fault-tolerant [67].

The following figure compares ETLs and ELTs:

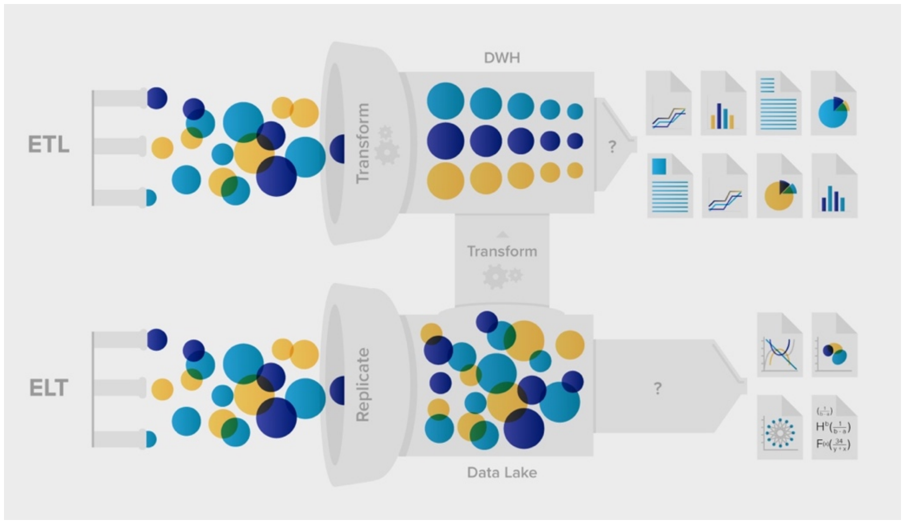


Figure 14 - ETL vs ELT [68]

4.10. Batch and streaming processing

Batch processing [69] consists in reading a large volume of historical data at rest aggregating and enriching it and writing it elsewhere. Given that this process often takes a huge amount of data, results could take from hours to a couple days to come out. Historically, this has been the common ETL processing used in data warehouses.

Streaming processing [69] refers to the ability of handling data arriving on-the-fly in the form of streams. Stream systems are not particularly good comparing live data to that coming from the pipe, as the main goal is to either perform simple aggregations or de-duplications.

4.11. Data governance

Data governance enables an organization to create a systematic data standardization, monitoring and management process, so that all data flows are transparent to any authorized user. To establish an effective data governance system, it needs to be set up in connection with corporate governance, IT governance, and ITA / EA from a company-wide perspective [70].

Data governance involves not only tools, but also a solid methodology to handle data in an efficient yet secure manner. Some of the most important aspects of data governance involve security, catalog, quality and policies, as observed in the following figure:

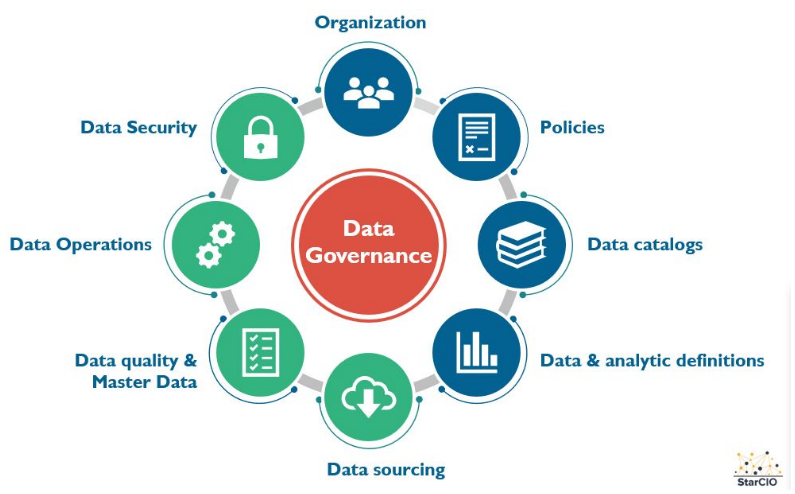


Figure 15 - Data Governance components [71]

A crucial aspect of data governance is data lineage, also known as data provenance, which can be defined as the data flow control and visibility of all pipelines across the organization. Data lineage helps users drill down into any step of a dataflow process, keep traceability of what transformations are being carried out and what type of data is being utilized for a given output. Literature around data lineage commonly associates it with metadata management, since part of the lineage traceability requires understanding the metadata of a given dataset before doing any processing.

Even though the use of data lineage approaches is a promising way for big data management, the process is rather complex. The challenges include scalability of the lineage store, fault tolerance, accurate capture of lineage for black box operators and many others. These challenges must be addressed carefully and trade-offs between them need to be evaluated to make a realistic design for data lineage capture [72].

4.12. Cloud platforms

The time required for businesses to acquire and set up servers could take from a couple days to a whole month and needed a huge investment to continuously upgrade to the latest hardware. Even so, after a couple years, those servers may be obsolete. Finally, managing those servers becomes a gigantic burden as more servers are bought to handle extra capacity.

On-premise hardware allowed corporations to split their workloads into multiple computers, leveraging the power of newest tools. However, this entailed a new set of problems, as they had to manage the infrastructure by themselves, which involved hiring specialized people to:

- Update the operating systems of all computers
- Keep the important software up to date
- Diagnose whenever a network issue occurred
- Monitor the overall health of the datacenter
- Replace any damaged hardware

To circumvent all of this, cloud computing became extremely popular and is, at the time of writing, one of the most profitable services in the world. Cloud computing refers to a number of on-demand services of compute power, database, storage, applications and other IT resources via internet in a pay-as-you-go pricing model [73].

According to [74], cloud service models can be broadly categorized as:

- **Software as a Service (SaaS).** Cloud user release their applications on a hosting environment, which can be accessed through internet from diverse terminals (e.g. web browser, PDA, etc.). Cloud consumers do not have control over the multi-tenant cloud infrastructure. Cloud consumers' applications are managed in a single virtual environment on the SaaS to leverage optimized amount of resources in terms of availability, speed, security, maintenance and disaster recovery.
- **Platform as a Service (PaaS).** PaaS is a development platform supporting the full "Software Lifecycle" which allows cloud consumers to develop cloud services and applications (e.g. SaaS) directly on the PaaS cloud. Hence the difference between SaaS and PaaS is that SaaS only hosts completed cloud applications whereas PaaS offers a development platform that hosts both completed and in progress cloud applications.
- **Infrastructure as a Service (IaaS).** Cloud consumers directly use IT infrastructures (processing, storage, networks, and other fundamental computing resources) provided in the IaaS cloud. Virtualization is extensively used in IaaS cloud in order to integrate/decompose physical resources in an ad-hoc manner to meet growing or shrinking resource demand from cloud consumers. The basic strategy of virtualization is to set up independent virtual machines (VM) that are isolated from both the underlying hardware and other VMs.

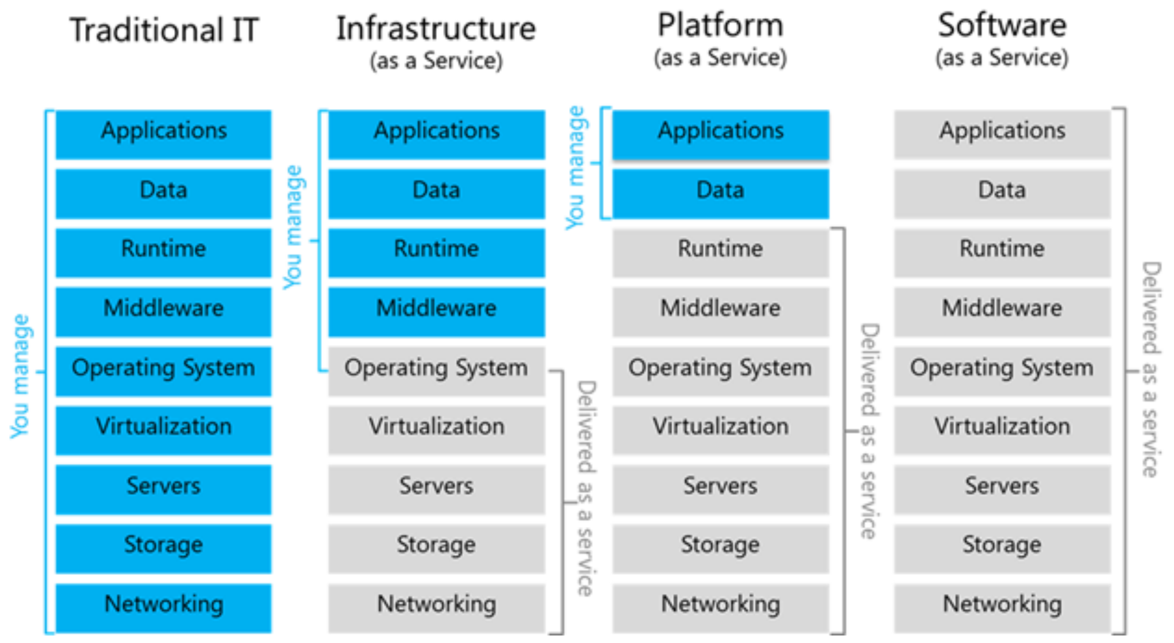


Figure 16 - Cloud service models [75]

4.13. Infrastructure as Code

As cloud services are embraced, developers started to work on ways to keep track of all the infrastructure they were generating, so that they could easily replicate, version and update infrastructure components. This concept is known as Infrastructure as Code (IaC).

Infrastructure as code [76] is a code-centric approach for infrastructure automation. Commonly, a tool is used to deploy, update and destroy cloud infrastructure changes on demand, for multiple cloud providers in contrast to proprietary tools such as AWS CloudFormation.

When talking about IaC it is important to highlight the main differences with a configuration management tool. The latter refers to tools designed to install and manage software on existing server. Some examples of configuration management tools are: Chef, Puppet, Ansible and SaltStack [77].

Such tools commonly utilize the YAML format to specify the configurations and installations that need to take place on a given server, or fleet of servers. As described in [77] offers a number of advantages:

- **Coding conventions:** Tools like ansible enforce consistent and predictable structure, including documentation, code layout, parameters and secret management.
- **Idempotence:** Refers to the ability for an artifact to produce the same result no matter how many times the process is executed. This ensures that users will always have a given configuration, no matter how many times they try to execute it, rather than having duplicates.
- **Distribution:** Configuration management tools are designed specifically for managing large number of remote servers as a single group.

When the infrastructure is defined as code, users can speed up the software delivery process including the following aspects:

- **Self-service:** Removing the dependency of sysadmins, as infrastructure can be deployed and destroyed automatically by a service.
- **Speed and safety:** If the deployment of the infrastructure is automated, developers do not need to worry about their changes breaking anything on the infrastructure, as the tool takes care of finding that optimal state to not alter already existing infrastructure.
- **Documentation:** Similar to any piece of code, IaC can contain comments and a folder hierarchy that can easily describe the components the infrastructure uses.
- **Version control:** Being code means that users can enter the code in a version control system such as git, and you can rollback to specific versions of the infrastructure effortlessly.

4.14. Microservices

A monolith service represents an immutable black box wherein all the business logic runs—and is deployed—as a single system. A monolith is almost always easier to develop initially and involves little to no architecture planning. In fact, most legacy systems nowadays work this way. If a piece of the business logic fails, the rest of the system will eventually crash as a consequence; due to the close dependency between libraries and/or function calls.

In light of this tight coupling, monoliths do not scale well. For instance, if the system is all of a sudden bottle necked due an unusual website traffic spike, and only specific pieces of the code require to increase capacity, the monolith is not capable of automatically assign new resources or prioritize that specific logic.

Microservices architecture, on the other hand, considers business logic components as individual and loosely coupled elements (services) that have bounded contexts [78]. Each service must be small, isolated and define ways to interact with the external world (e.g., through APIs and ports).

Microservices are harder—and take longer—to design and develop, as they need special mechanisms to communicate and strategies for failover, disaster recovery and scalability.

Microservices follow the three-dimensional scalability model, or scale cube [78]:

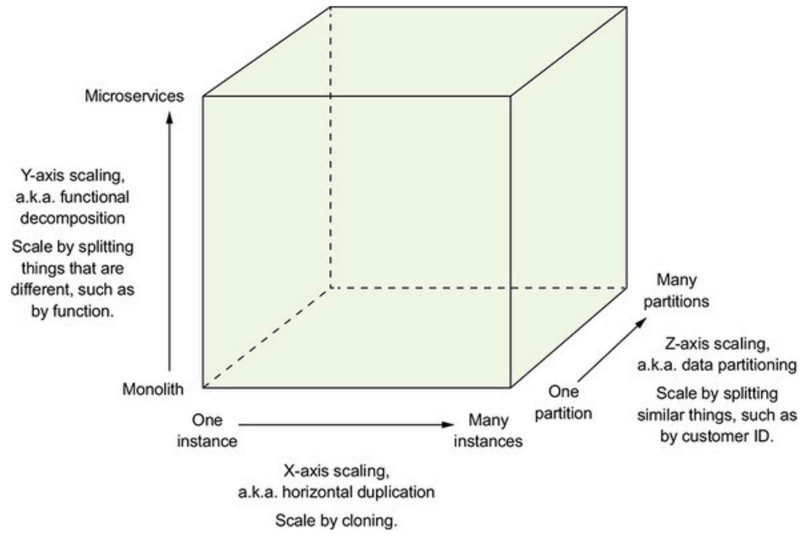

Microservices
Y-axis scaling, a.k.a. functional decomposition
Scale by splitting things that are different, such as by function.
Monolith
One instance
Many instances
X-axis scaling, a.k.a. horizontal duplication
Scale by cloning.
One partition
Many partitions
Z-axis scaling, a.k.a. data partitioning
Scale by splitting similar things, such as by customer ID.

Figure 17 - Scaling cube

In the x-axis we have horizontal duplication, which includes adding more instances. To supplement this concept, a load balancer is commonly used in front of the services, so that all requests go through it and are delivered to any number of services of the same nature.

The z-axis refers to having different instances of the service working on different parts of the data (i.e., sharding). A router is used in this case to determine where the request should go to.

The y-axis is specifically aimed to decompose, and route requests based on its type (e.g., customer, order, review, etc.). This is the basis of all microservices.

Modularity should be the top priority when working with microservices: a small group of developers should be able to focus on a piece of service without interfering with the development of other teams. In this way, teams achieve full independence and are able to evolve systems more efficiently.

Another benefit of modularity is that systems can be tested and deployed independently; if done properly, the rest of the services should not notice any downtime. Overall, this leads to much smaller code bases and in consequence quicker ramp up from project's newcomers.

As data shuffling is one of the most expensive operations any service may incur in, each microservice must be designed to allocate its own database inside it and only expose the most important data via APIs or Remote Procedure Calls (RPCs).

For APIs, the most used ones at the time of writing are RESTful, whereas RPCs the most efficient one is gRPC, developed by Google.

4.15. Serverless

Serverless is an implementation model where the cloud provider is accountable to execute a given function or piece of code, by managing underlying compute and storage resources without user intervention [79]. In this model, you pay only for execution time of your function or the throughput it required, rather than the servers running underneath.

Serverless is sometimes referred to as Function as a Service (FaaS) as the actual function is executed. To further illustrate this statement, think of a typical application that requires an entry point—a main function, for instance; that entry point will then, given specific conditions, reach the function to execute. In a cloud environment, the server setup is also mandatory to allow the function to be invoked. FaaS abstracts away all those complexities from the user and let them develop reactive applications that closely resemble—or in some cases are—microservices.

AWS Lambda, Google Cloud Functions, IBM Cloud Functions and Microsoft Azure Functions are just a few FaaS products provided by the major cloud vendors nowadays.

The serverless architecture is also an event-driven architecture since each function is invoked in consequence to one or more events [80]. A primary advantage of this architecture is the possibility to execute—theoretically—unlimited number of functions in parallel. Combined with the cloud philosophy of pay for what you use, this may lead to enormous savings for the company.

4.16. REST

Representational state transfer (REST) is an architecture style for distributed hypermedia systems [81]. REST web applications expose information about the service organized by its resources. It enables clients to take actions on resources such as creating new ones (e.g., purchases, profiles) or edit existing ones (e.g., update user's password).

A RESTful API is deemed a callable object that, once invoked, will transfer the representation of the requested resource's state [82]. When a developer calls an API to fetch a specific user's data, the API may return the state of that user, including name, age, address and phone number.

REST APIs commonly return data in JSON format, as it allows for nested and complex structures such as maps, lists and internal structures. Some infrequent APIs may also return XML or HTML which may indicate an outdated or purpose-specific service.

REST APIs commonly leverage the HTTP protocol, which defines two components:

1. **Endpoint:** An identifier for the resource you are interested in. Given a hierarchy of resources—e.g., company, team, employee—the endpoint can define the broadest category first and go deeper into the subcategories: *company/<company_id>/team/<team_id>/employee/<employee_id>*.
2. **Operation:** The action to take upon the resource, which is guided by the HTTP method or verb. The most used verbs are: GET, POST, PUT and DELETE.

An advantage of RESTful APIs is the fact that they decouple client and server: both can work independently of each other, if the server goes down, the clients gets its corresponding error message clearly hinting that the server is not up and running.

RESTful APIs commonly utilize the following error codes to roughly describe the result that is returned to the user:

CATEGORY	DESCRIPTION
1xx: Informational	Communicates transfer protocol-level information
2xx: Success	Indicates that the client's request was accepted successfully
3xx: Redirection	Indicates that the client must take some additional action in order to complete their request
4xx: Client Error	The client made an invalid request.
5xx: Server Error	The server takes responsibility for these error status codes

Table 3 - HTTP status codes [83]

In order for an API to be RESTful, it must adhere to 6 constrains:

1. **Uniform interface:** Refers to the ability for clients to call the exact same way the API, no matter where it is invoked from. Also, it includes a homogeneous resource name.
2. **Client/server separation:** As the name implies, the client and the server act independently and the communication is only initiated by the client: the server just waits for requests to be queued up.

3. **Stateless:** The server does not remember anything about the user who uses the API. This is often translated as idempotence, which means that no matter how many times you execute a request, you should always obtain the same result.
4. **Layered system:** Between the client and the server, there might be any number of layers (security, caching, processing, etc.) that interact with the server to provide the request's response.
5. **Cacheable:** The server may send information hinting that some data was cached. If so, it will provide a means to decommission or refresh it.
6. **Code-on-demand:** (Optional) In some cases, the resource may be a code snippet, so the code is returned in plain text to the user to store and execute later on.

Several microservices and serverless applications nowadays leverage REST APIs as a means to communicate with each other efficiently and decoupled. This is crucial, as there could be no dependencies between microservices, and REST APIs ensure that communication is always working, regardless of the state of the service.

4.17. Data engineers, analysts and scientists

Over the recent years, the role of developers and integration teams has shifted significantly with respect to the technologies and the responsibilities each engineer has. With the advent of micro services, for instance, developers were able to deliver higher quality software in a more predictable way and generating less bugs by decoupling large chunks of code into bite-size modules. Once the number of microservices and software products were humongous, it became hard and error-prone to manage infrastructure to deliver on time. This is when DevOps gained momentum.

Site Reliability Engineers, often times known as DevOps Engineers, are usually in charge of isolating the pieces of code in containers (Docker, most of the time) and deploy them on a given cluster. At the same time, developers and analysts had to share responsibilities to be able to handle the data challenges that the companies were facing in order to be able to perform more organic analytics. In consequence the following roles appeared: Data Engineer, Data Scientist and Data Analyst.

A Data Engineer is a developer focused on big data pipelines creation. His/her primary duty is to design, implement, test and improve data pipelines to move data from multiple sources to another while performing some transformation or cleaning. One of the key responsibilities is the optimization of the pipelines on a cloud environment [84], since the costs of overusing resources may soar unexpectedly if the right considerations are not in place. Data Engineers also participate in introducing new tools or updating previous ones, create jobs to execute ETL or ELT processing on a periodic basis, cleaning dirty data from its raw form, exposing APIs for analysis, and managing the metadata so that subsequent processes can leverage the data without worrying about any missing piece.

A Data Scientist combines parts of the developers and analysts. They are in charge of extracting value of the data. They are particularly good at machine learning and proficient on selecting the most suitable models for a given problem, optimizing and testing hypothesis based on data, as well as generating future predictions [84]. Data Scientists have much more contact with businesspeople than Data Engineers, as they generate proof of concepts for a particular business opportunity.

Data Analysts are those in charge of performing day-to-day analysis to fill in reports and observe common behaviors. They have even more contact with businesspeople. Analysts are specialized in SQL and BI tools to leverage diagnosis and exploratory work. Data Scientists' discoveries may trigger the incorporation of new analysis into the pipeline, which is performed by the Analysts.

5. APPROACH

In this section we will establish the methodology that will be followed throughout the execution of this project. Then, we will state the initial proposal, backed up by requirements and assumptions, to better delimit the scope of the project.

5.1. Methodology

As with many software engineering projects nowadays—and being the standard in the technology industry—we chose Agile to develop and continuously improve both the documentation, as well as the code base while keeping a dynamic communication.

Agile process is an iterative approach in which customer satisfaction is the utmost priority and the customer has direct involvement in evaluating the software [85]. This process follows the software development life cycle which includes requirement gathering, analysis, design, coding, testing and delivery.

Agile came into existence in 2001 and organizations have tried to adopt it ever since, with the promise of improving software development and overall product quality. However, few studies have been made to measure the effectiveness of this methodology in the industry [86].

One of the most embraced processes of Agile is Scrum, which is an agile way to manage a project, usually related to software development. Scrum relies on self-organizing, cross-functional teams in order to achieve agility to deliver [87].

Preliminary analysis demonstrates that a large number of big organizations have adopted Scrum, including Amazon, Yahoo Music and BabyCenter [86].

To effectively harness the power of the methodology, a combination of organization mindset and product necessity is required. For the sake of this project, the background of both my tutor and I, as well as the necessity of the project, fit into the Agile process.

Given the time restrictions, we agreed on synching up every two months at minimum and a demo with all the features by the end of each semester. In case something goes off-track we can easily get in touch.

5.2. User Stories

User stories are general explanations of a software feature written from the perspective of the end user or customer. The purpose of a user story is to articulate how a piece of work will deliver a particular value back to the customer [88].

Data Analyst
I want to have a simple web interface so that I can create new pipelines
I want infrastructure to be automatically set up once I start the system, so that my pipelines can be built on top of it
I want to know when the basic infrasture is ready, so that I can start submitting pipelines.
I want to specify data sources residing in my data center as the transformation input
I want to specify transformations, in ANSI SQL, that will be applied to one or more data sources
I want to specify the destination table name where the result of the transformation shall be stored
I want to specify a specific date time pattern, using CRON expressions, so that this pipeline can be executed then.
I want to visualize the progress of the pipeline creation, so that I can keep track of all intermediate steps.
I want to easily access the data lineage system, so that I can visualize the relationship between datasets in my cluster as well as the transformations.
I want to easily access the scheduler system, so that I can visualize all active pipelines.
I want to disable already-existing pipelines, so that they do not longer run.
I want files to be automatically migrated to the cloud, once I create the pipeline.
I want to be able to query results, in ANSI SQL, from the output table, so that I can verify if the output makes sense.

Table 4 - Data Analyst user stories

DevOps Engineer
I want pipeline infrastructure to be automatically managed by the microservices, so that I do not need to manage it myself.
I want a means to access different resources, so that I can configure and update when needed.

Table 5 - DevOps Engineer user stories

Security Engineer
As a security engineer, I want to be able to visualize trails from the infrastructure, so that I can audit what services did what.
As a security engineer, I want processing cluster to be hidden from the outside world, in a private VPC, so that my nodes are more secure.
As a security engineer, I want cloud services to only have the absolute necessary access to other services, via firewalls.

Table 6 - Security Engineer user stories

Business User
As a business user, I want infrastructure to be automatically created and torn down whenever the system is not using it, so that I can save money.
As a business user, I want pipelines to be optimized for to the data that will be consumed, so that I both the execution time is reduced and the costs keep at minimum.

Table 7 - Business user stories

5.3. Requirements

Requirement	Comments
The user shall provide a valid AWS credentials file before deploying the system locally.	Permissions must include EC2FullAccess, EMRFullAccess and S3FullAccess
The system shall be architected using independent microservices.	
Microservices shall be containerized, so that they can be platform independent.	
The system shall include a convenience script to start the webserver, infrastructure manager and pipeline administrator.	
The code repository shall contain all templates and scripts required for both the local agent as well as the cloud agent to work properly.	
The input data shall be migrated to the data lake.	
The output data shall be stored in the data mart.	
The system shall block the user from entering new pipelines until the basic infrastructure is ready.	This can be achieved by disabling the create button and/or the fields to enter the pipeline specifics.
The infrastructure microservice shall deploy the basic infrastructure at startup.	
The infrastructure microservice shall notify when the basic infrastructure is ready.	
The infrastructure microservice shall provide an API to get the status of the infrastructure setup.	
The infrastructure microservice shall provide an API to build infrastructure modules.	
The infrastructure microservice shall provide an API to tear down infrastructure modules.	
The pipeline microservice shall provide an API to create a new pipeline.	
The pipeline microservice shall verify the existence of the file(s) used as input.	
The pipeline microservice shall extract the metadata from the input file(s).	

The pipeline microservice shall coordinate the migration of the input datasets to the data lake.	
The webserver shall check for any missing or incorrect field in the web page.	
One or more remote virtual machine shall be set up to handle the remote setup of the system.	The remote system shall contain the required services to handle scheduling and lineage tools.
The DAG manager microservice shall include a folder where all DAGs are to be stored.	
The DAG manager microservice shall trigger the creation of tables on top of the input datasets.	
The DAG manager microservice shall create the scheduling script using the SQL transformation and the table inputs.	
The remote server setup shall include all necessary databases for both the scheduling and the lineage microservices	
The lineage microservice shall automatically capture the data lineage from the execution of all DAGs	
The different components of the scheduling microservice, shall be deployed independently.	The components include the webserver, the scheduler and the celery worker

Table 8 – Requirements

5.4. Assumptions

- The cloud provider is Amazon Web Services (AWS).
- An AWS account is available.
- Input files are located in the host filesystem or mounted onto it.
- Docker is properly installed and configured.
- An IAM role is available and has a policy with the following actions:
    - s3:*
    - elasticmapreduce:*
    - ec2:*
    - glue:*
- Input files are in either CSV or JSON format.
- A copy of the Adaptive Big Data Pipelines repository [89] is already available in the local filesystem.
- The system is manually triggered by a script.
- A minimalistic non-reactive web interface will be provided.
- SQL transformations are not validated for correctness.
- Jinja templates are not supported.
- Dashboards are not provided.

- Previously created DAGs' edition is not supported.
- Logging is available exclusively via Docker.

5.5. Proposal

5.5.1. Architecture

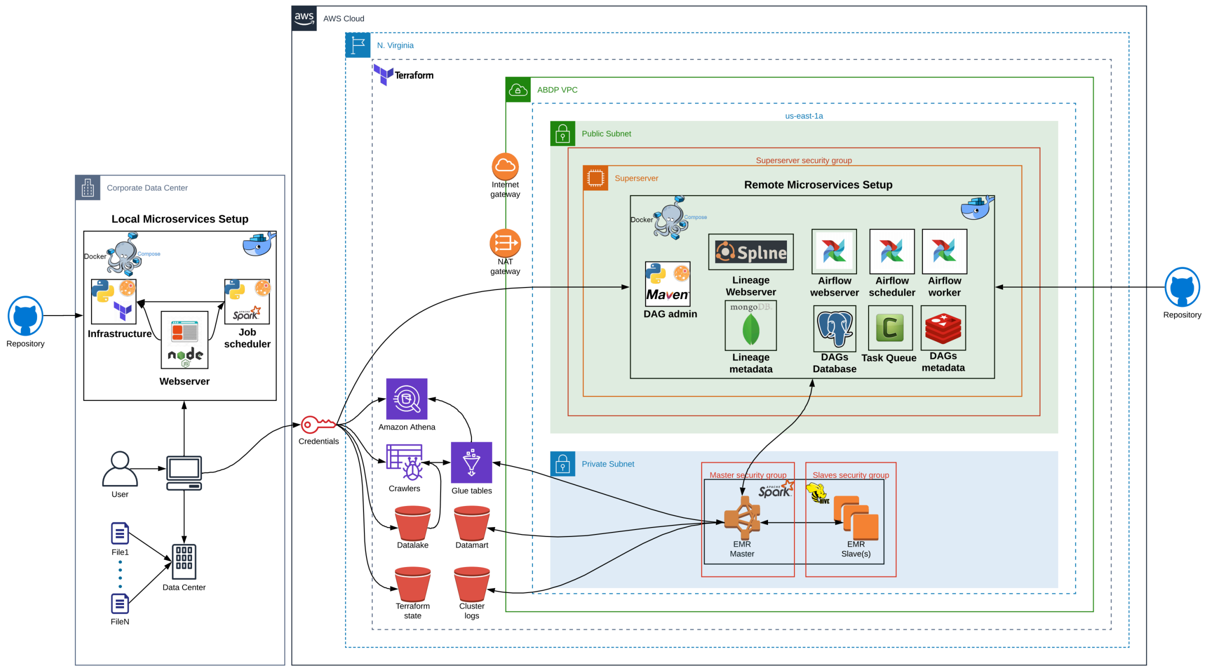


Figure 18 - Technical architecture of the adaptive data pipelines

5.5.2. Description

The adaptive big data pipelines system is a multi-layered architecture, comprised of several interwoven software artifacts; all of which are orchestrated as individual microservices and deployed using Docker Compose. Hence, it is worthwhile first providing a broad level overview so that it becomes clear once we dive into the individual components.

At a high level, the system is spun up on the company's data center. The system provides a web interface the user can interact with locally using a web browser. At startup, the cloud infrastructure is set up and configured automatically under the hood. The web interface is fully enabled once the minimum dependencies are up and running. At this point, the user could enter information about the pipeline to be created:

1. Name of the pipeline.
2. Up to three data sources path in the local filesystem.
3. Table aliases for each data source.
4. Transformations and/or aggregations in ANSII SQL.
5. Output table name.
6. A CRON expression.

Once the above information is entered, and the user clicks on Submit, the pipeline extracts metadata from the files, including the file size, format, schema, and row count; these metadata will later on be used to tailor the pipeline transformations on Apache Spark for maximum performance and lowest cost.

The local microservices (i.e., webserver, infrastructure and job scheduler) coordinate with the remote super server—the central cloud virtual machine in charge of creating the DAG and fine-tuning the cluster settings to run smoothly. The webserver sends the gathered pipeline requirements to the job scheduler, which calculates the metadata, depending on the file. Then, it starts the migration process to the data lake.

Once the migration is ready, the job scheduler creates a Glue crawler over the data, in order to generate the corresponding tables on top of the data. Once that is done, the DAG manager creates the Airflow DAG specifying the parameters for the pipeline's execution.

The configured DAG will be automatically launched once Airflow picks it up from the DAGs folder. For the ETL execution, an EMR ephemeral cluster—i.e., a cluster that is created with the unique purpose of performing a data transformation and then it is automatically wiped out—is created. The cluster is bespoken to the data volume and transformation necessities (number of nodes and Spark configurations). It will perform the specified transformations and dump the result in the output table.

The above described ETL will report the Spark data lineage to Spline and it will automatically make it available for users through the Spline UI to drill into specific transformations.

Communication is key when talking about microservices. Therefore, both the infrastructure and DAG admin microservices expose an endpoint for other services to poll their current processing state. This is particularly useful in the web interface, so that its components are reactive (i.e., to the first time the infrastructure is being set up, to disable controls and avoid having inconsistent states). Also, when submitting a job, the user can see first-hand what is happening on the remote server with the pipeline creation and execution.

Once the DAG is up and running, the user just sits and relaxes: the pipeline is automatically scheduled to run according to the CRON task provided at design time. The scheduling and execution are taken care of by Airflow.

As briefly outlined, microservices are created as individual Docker containers that expose a minimalistic web server with a set of well-defined REST APIs/endpoints to interact with each other. In this manner we achieve the following:

- Fault tolerance
- Modularity
- Separation of concerns
- Scalability

- Isolation
- Decoupling
- Easiness to install dependencies

Now that we described the main pieces of the system, we can focus on breaking down the architecture in bite-size segments. First off, we can classify the system in two broad contexts:

- On-premise
- Cloud

The on-premise context is comprised by the three initial microservices: the webserver, the job scheduler and the infrastructure (manager), whereas the cloud system is comprised by all the cloud infrastructure, created on runtime and 9 other microservices are created: 6 related to Airflow, 2 to Spline and the DAG admin.

Alternatively, the system may also be divided in the following more granular blocks:

1. Cloud infrastructure
2. Web interface
3. Data Migration
4. DAG setup
5. ETL processing
6. Lineage tracking

Cloud infrastructure uses Terraform to create the corresponding infrastructure at startup. It offers an endpoint to consult the status of the infrastructure creation. This is mainly used in the web interface to enable and disable controls. Many variables are generated during the cloud infrastructure setup, including:

- IP address of the remote server
- Security group ids
- Roles ids
- Bucket names
- Public and private subnet ids

This information is stored. Once the pipeline is submitted, the job scheduler polls this information to communicate and coordinate pipeline creation with the DAG admin.

The web interface is comprised by a backend micro-engine using Nodejs, and a simple web page that the user can consult. As described above, controls are disabled until the underlying infrastructure is running to avoid accidents. This service communicates with both infrastructure and job scheduler to submit pipelines.

Data migration is taken care of by the job scheduler. This process involves extracting the metadata of the input files and copying the files to the data lake. This process leverages the infrastructure outputs. Migration highly relies on the network connection, so the bigger the file, the longer it will take to copy it. Once the migration is ready, a Glue table is created using a Glue crawler. This process finishes once the crawler successfully finishes creating the table(s).

DAG setup occurs entirely on the remote server given a cue from the on-premise data center. This process creates a python file that contains both the transformations and the complementary data so that the Airflow DAG can be picked up. The resulting file is stored in the DAG bag for Airflow to find it.

Given the complexity of Airflow, all its components are segregated in different containers, including the scheduler, the task queue, the DAG metastore, the webserver, etc.

ETL processing occurs both once the DAG is successfully picked up the first time, and every time the CRON expression indicates an execution. This process will create an ephemeral cluster, on EMR, tailored to the data volume and schema requirements.

The internal Spark application template is controlled by the DAG admin service and is automatically deployed to the data lake for EMR to pick it up.

The cluster is configured to read from the Glue metastore and so it leverages the Glue tables both for input and output. Along with that, Spark SQL has full support for ANSII SQL syntax, which allows the application to execute the plain query as is. The cluster is terminated after the processing is done to not incur in unnecessary charges.

Data lineage tracking is comprised by a web server and a MongoDB database. The latter is where the lineage metadata is automatically stored from the EMR cluster. Once the transformation is evaluated, i.e., when the Spark application writes the results to Glue metastore, the application automatically takes care of sending that information to the lineage database. At this point, Spline's UI can pick up the latest lineage updates instantaneously and users can visualize the transformations as a Directed Acyclic Graph (DAG).

Security is of the utmost importance in modern cloud systems. Data pipelines are by far the most vulnerable of all systems, as often times the information that is transported contains either personal identifiable information (PII) or any hashed values related to passwords. Therefore, the adaptive data pipelines infrastructure is generated with all critical components isolated in a private subnet—inside the VPC—and the access is restricted via security groups, which only allow access from the data center's IP address and inside the cloud infrastructure to the mandatory components (analogously to how a firewall operates).

Furthermore, by attaching roles to the different components, we can control what access they have over other services. For instance, the EMR cluster has only access to create EC2 instances, which in turn have access only to read from the metastore and S3. Any other malicious action will ensue an access denied. This prevents that, in case of a hacker makes his/her way into the data center, the component is restricted to the bare minimum services and hence the impact is severely reduced.

6. EXPERIMENTS AND RESULTS

In this section we describe the main experiments and results obtained during the validation of the adaptive big data pipelines. All the code as well as the initial documentation can be found in the open source repository on GitHub [89].

Since this project is aimed towards big data processing, the first obvious prerequisite is to get a hold of an interesting and large data set, preferably in the order of dozens to a few hundred GBs.

Unfortunately, most of the data sets available in internet (Kaggle, UCI, etc.) proved not useful for any of the following reasons:

- Format is neither CSV nor JSON.
- Many thousand small files instead of few hundred big ones.
- Extremely dirty data.
- Volume of data is too low (KB or MB).
- Data analysis will not provide an interesting and real-world scenario.
- Data is not related to any other dataset to perform combinations.

Given the above reasons, I decided to create my own synthetic data set that resembles real-world as closely as possible, while keeping the format, relations and data volume. In this case, I created data for a medium-sized ecommerce company.

One important oversimplification of data generators available online is random information production. For sheer data processing time estimation, this may suffice, however, my approach must also consider a slight data skewness—that is, not all users behave the same way and not all countries generate the same number of events. For this particular case we will use normal distributions to generate tendencies [90].

I generated 3 data sets:

- Users
- Sessions to users
- Sessions purchases

6.1. Tables

Figure 19 represents an ERD diagram with the overview of the tables structure. Specific details are elaborated in the following sub section.

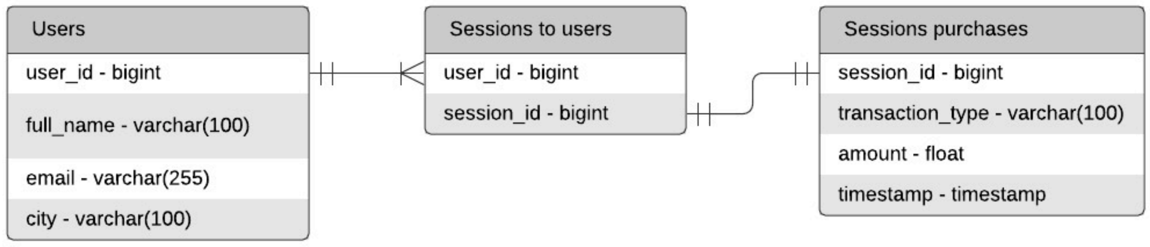


Figure 19 - ERD diagram

6.1.1. Users

This dimension data set contains information of 10 million unique users. It contains basic information from the platform users. The file format will be CSV. The table contains following fields:

- user_id
- full_name
- email
- city

I leveraged Mexican's top 500 most common male and female names, along with the top 1000 most common last names [91]. Mexican **full names** are made up of one or more names followed by two last names: the first one is the father's father surname and the second is the mother's father surname. Considering that, I could encode up to 500 million unique names (although in real life, a perfect match between two full names is not unheard of).

Emails are generated by concatenating name and surnames using a dot. Emails will have gmail extension (i.e., @gmail.com).

The **city** field is generated based on 12 of the most important mexican cities:

1. Mexicali
2. Hermosillo
3. Veracruz
4. Guadalajara
5. Monterrey
6. Tampico
7. CDMX

8. Pachuca
9. Tuxtla Gutierrez
10. Cancun
11. Colima
12. Tepic

6.1.2. Sessions to users

This data set is an intermediate table between users and purchases. It only contains unique identifiers from both tables to join them, as there is a one-to-many relationship between users and purchases (i.e., any given user may buy an arbitrary number of times). Files will be given in JSON format.

The fields included in this table are:

- user_id
- session_id

6.1.3. Sessions purchases

This last data set represents the actual purchases of the users. It will be given in JSON format. Files will contain the following schema:

- session_id
- transaction_type
- amount
- timestamp

transaction_type can be *credit_card*, *cash* or *paypal*.

amount is a decimal ranging from 0 to 2000. Represents the amount spent on this transaction.

timestamp represents the purchase unix timestamp. It ranges from 2019 to mid 2020.

6.2. Automating data generation

Since I needed to carry out a few tests over different volumes of data, I decided to generate three python scripts, one for each data set, and add command line arguments to specify how many records and how many files to split data sets into.

Given the cost implications of storing, distributing and scanning large volumes of data, I ended up creating the following arrangement:

1. 10 million unique users divided in 5 CSV files
2. 300 million unique relations between users and purchases divided in 30 JSON files
3. 300 million unique purchases divided in 30 JSON files

The above-mentioned data is then generated using the following high-level script:

- ./create_data.sh 10000000 5 300000000 30

The resulting data is about 45 GB in size. This is perfect for demonstration purposes, as it appropriately emulates a medium-sized company data volume.

6.3. Datacenter remote setup

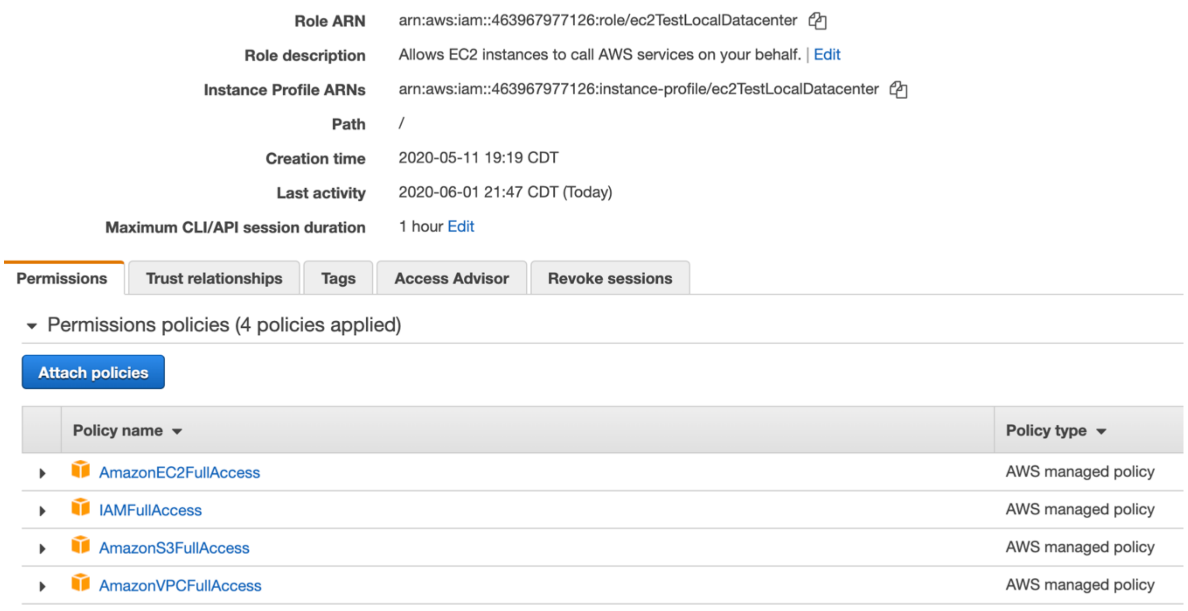
As described in previous sections, data migration takes place along the pipeline creation. Since data is directly copied to S3, a mechanism called multi-part upload is implemented in the job scheduler microservice. This process speeds up the upload process by dividing files into roughly equal-sized chunks and delegating each thread a set of chunks to transmit.

Despite such improvement, data transfer is ultimately constrained by the internet bandwidth of the data center—or, in my case, my internet provider. Early tests showed that 1 GB files could take up to 40 mins to upload, depending on the network conditions. To circumvent this limitation, an EC2 instance is created. We call this instance the data center (in resemblance to an actual data center with decent internet connection).

The data center instance is created with two purposes in mind: generate the synthetic data efficiently (using last generation CPUs) and run the local data center logic—including the webserver, job scheduler and infrastructure.

The instance type I chose is c5n.xlarge, as it features a high-speed network card that allows for rapid data transfer to S3 while migrating. Moreover, in order to avoid copying credentials over to the instance, an EC2 role is attached to the instance. As described in the assumptions, the role has the following access:

- VPC full access
- EC2 full access
- S3 full access
- IAM full access



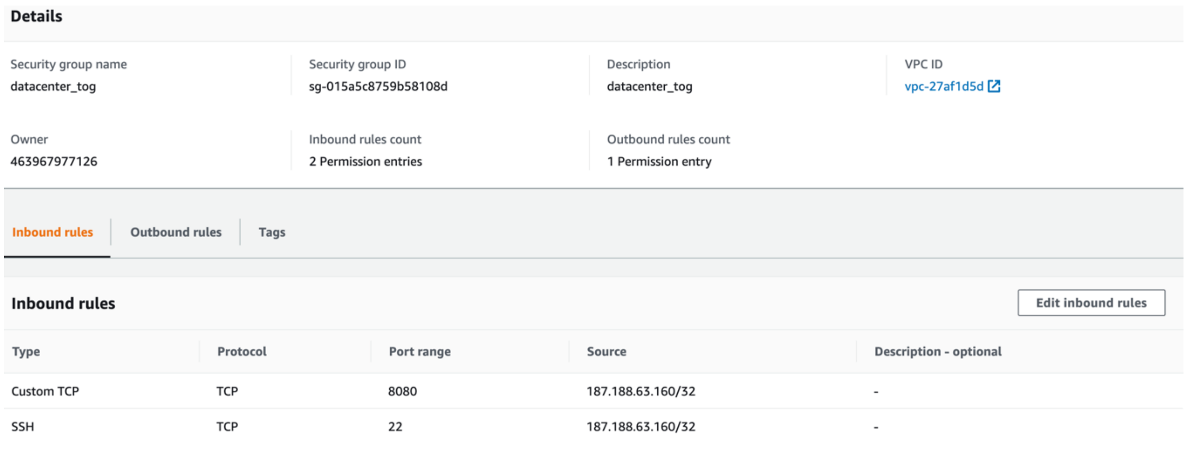
Role ARN	arn:aws:iam::463967977126:role/ec2TestLocalDatacenter
Role description	Allows EC2 instances to call AWS services on your behalf. | Edit
Instance Profile ARNs	arn:aws:iam::463967977126:instance-profile/ec2TestLocalDatacenter
Path	/
Creation time	2020-05-11 19:19 CDT
Last activity	2020-06-01 21:47 CDT (Today)
Maximum CLI/API session duration	1 hour Edit

Policy name	Policy type
AmazonEC2FullAccess	AWS managed policy
IAMFullAccess	AWS managed policy
AmazonS3FullAccess	AWS managed policy
AmazonVPCFullAccess	AWS managed policy

Figure 20 - Data center permissions

Notice that full access is granted on all 4 services. This is to avoid any complications while experimenting. Further restrictions could be applied to increase security. I will elaborate on this on the future work section.

A security group is created for the data center in order to have inbound traffic access on ports 22 (SSH) and 8080 (webserver's port to dispatch the actual web form) to the public IP from where I was conducting my experiments. Since my service provider sporadically changes my public IP, I had to periodically update it on the console. In my case, this was particularly notorious while trying to SSH into the instance and no warning message was shown immediately.



Details

Security group name	Security group ID	Description	VPC ID
datacenter_tog	sg-015a5c8759b58108d	datacenter_tog	vpc-27af1d5d

Owner	Inbound rules count	Outbound rules count
463967977126	2 Permission entries	1 Permission entry

Inbound rules

Type	Protocol	Port range	Source	Description - optional
Custom TCP	TCP	8080	187.188.63.160/32	-
SSH	TCP	22	187.188.63.160/32	-

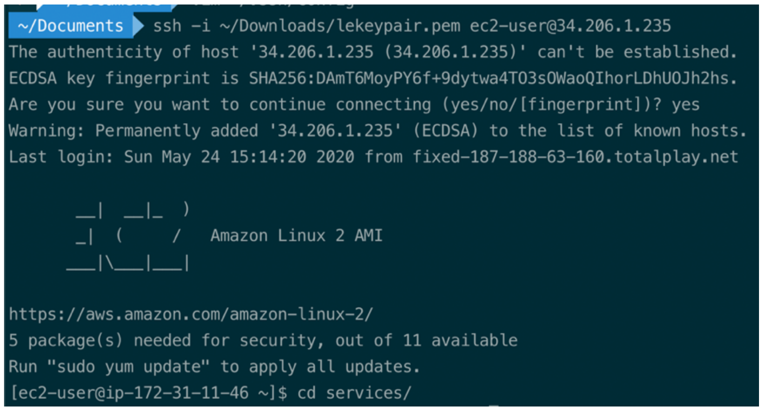
Figure 21 - Data center security group

At this point, I already had most of the adaptive pipeline's code already implemented—or so I thought. The original design contemplated the webserver as well as the local services to be executed locally (thence the name).

In order to successfully execute the local setup remotely, I had to strip off the microservice interaction from the client-side script and incorporate it in the webserver logic. The client-side script redirects the requests to the webserver, and it calls the corresponding microservices on its end.

6.4. Spin up local services

Once the data center is created, SSH into it using the keypair generated for it (*lekeypair.pem* in my case).



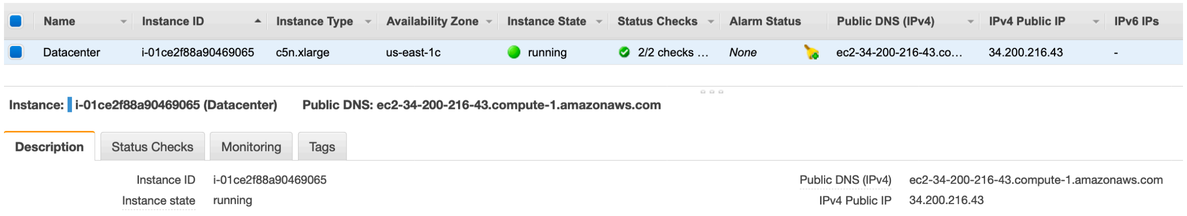
```
~/Documents  ssh -i ~/Downloads/lekeypair.pem ec2-user@34.206.1.235
The authenticity of host '34.206.1.235 (34.206.1.235)' can't be established.
ECDSA key fingerprint is SHA256:DAmT6MoyPY6f+9dytwa4TO3sOWaoQIhorLDhUOJh2hs.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '34.206.1.235' (ECDSA) to the list of known hosts.
Last login: Sun May 24 15:14:20 2020 from fixed-187-188-63-160.totalplay.net

       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
5 package(s) needed for security, out of 11 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-11-46 ~]$ cd services/
```

Figure 22 - Log into the data center

Once the data is generated in the data center, as per previous sections outline, the repository is cloned, and we navigate to the services folder. From there, the local setup can be instantiated using the convenience script as follows:

- ./setup.sh local

Once the services are up and running, we copy the data center's public IP, from EC2 dashboard, and enter it into our web browser with port 8080:



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP	IPv6 IPs
Datacenter	i-01ce2f88a90469065	c5n.xlarge	us-east-1c	● running	✓ 2/2 checks ...	*None*	ec2-34-200-216-43.co...	34.200.216.43	-

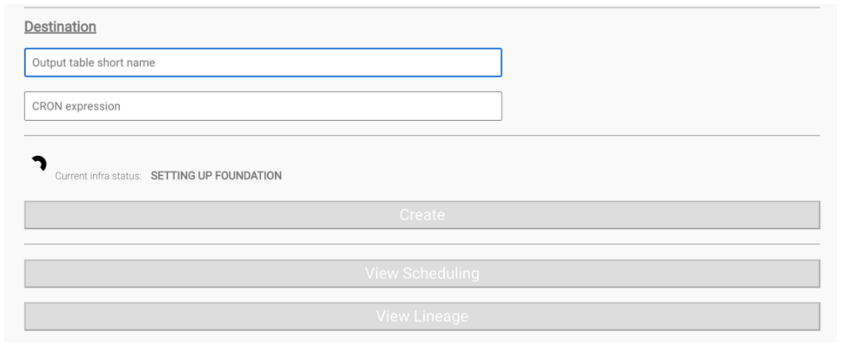
Instance: i-01ce2f88a90469065 (Datacenter) **Public DNS:** ec2-34-200-216-43.compute-1.amazonaws.com

Instance ID	i-01ce2f88a90469065
Instance state	running
Public DNS (IPv4)	ec2-34-200-216-43.compute-1.amazonaws.com
IPv4 Public IP	34.200.216.43

Figure 23 - Public IP location of data center

- <data_center_ip>:8080

Once the web form is loaded, the underlying infrastructure build is triggered. Right above the Create button, a text label shows the status of the build. All buttons are disabled while the underlying infrastructure is being created to avoid inconsistencies.


Destination

Output table short name

CRON expression

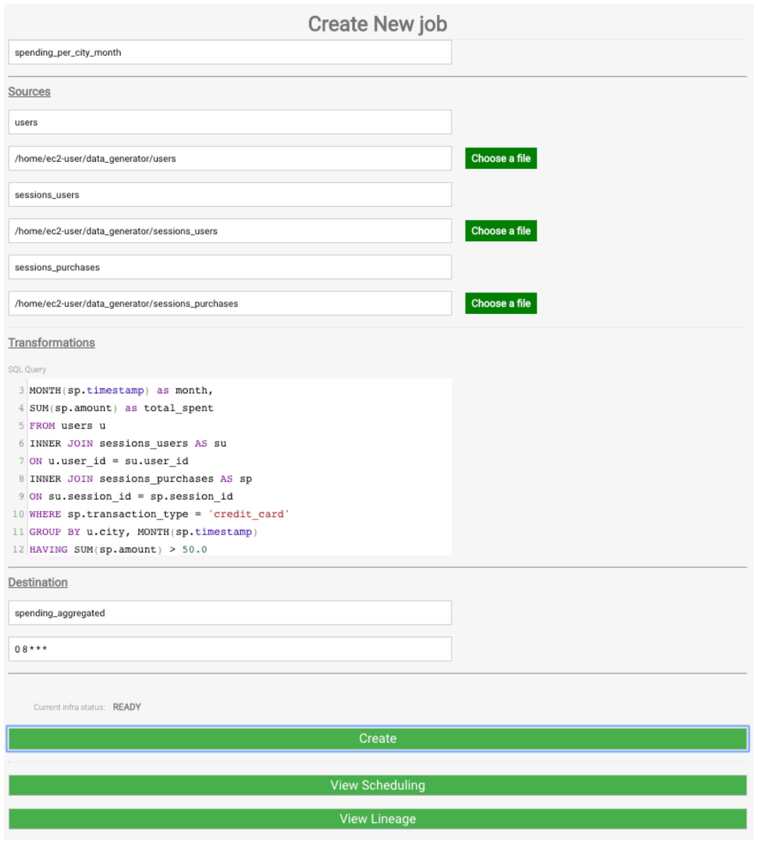
Current infra status: SETTING UP FOUNDATION

Figure 24 - Disabled controls for web UI

The four stages of the infrastructure creation are:

1. INITIALIZING
2. SETTING UP FOUNDATIONS
3. SETTING UP SUPERSERVER
4. READY

Once in READY state, the buttons will turn green, meaning that they are now enabled. We now proceed to enter the pipeline information including pipeline name, 3 data source paths (from the data center's filesystem perspective) and names, SQL transformation, output table name, and schedule time.


Create New job

spending_per_city_month

Sources

users

/home/ec2-user/data_generator/users

sessions_users

/home/ec2-user/data_generator/sessions_users

sessions_purchases

/home/ec2-user/data_generator/sessions_purchases

Transformations

SQL Query

```
MONTH(sp.timestamp) as month,
SUM(sp.amount) as total_spent
FROM users u
INNER JOIN sessions_users AS su
ON u.user_id = su.user_id
INNER JOIN sessions_purchases AS sp
ON su.session_id = sp.session_id
WHERE sp.transaction_type = 'credit_card'
GROUP BY u.city, MONTH(sp.timestamp)
HAVING SUM(sp.amount) > 50.0
```

Destination

spending_aggregated

0 8 * * *

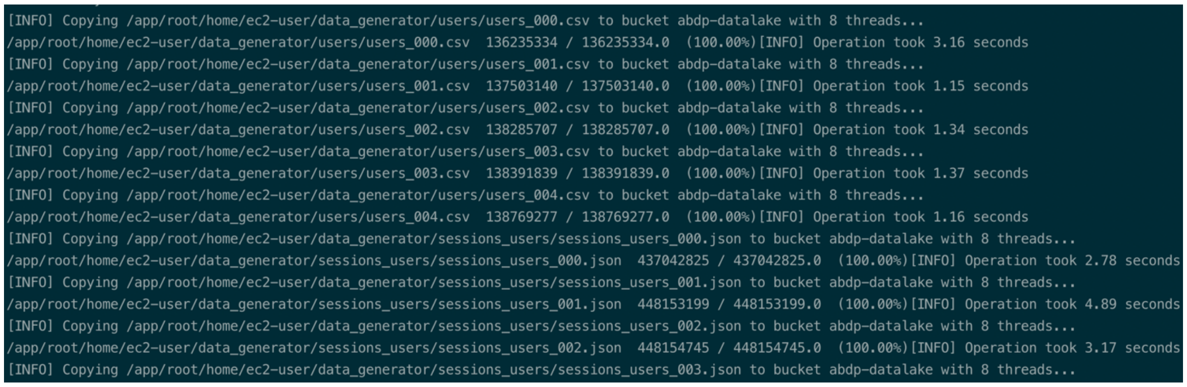
Current infra status: READY

Figure 25 - New pipeline submission

The transformation we use in this experiment provides insights as to the total spend users have in total, per city per month, where the spending is above 50 pesos.

```
SELECT
u.city,
MONTH(FROM_UNIXTIME(sp.timestamp)) as month,
SUM(sp.amount) as total_spent
FROM users u
INNER JOIN sessions_users AS su
ON u.user_id = su.user_id
INNER JOIN sessions_purchases AS sp
ON su.session_id = sp.session_id
WHERE sp.transaction_type = 'credit_card'
GROUP BY u.city, MONTH(FROM_UNIXTIME(sp.timestamp))
HAVING SUM(sp.amount) > 50.0
```

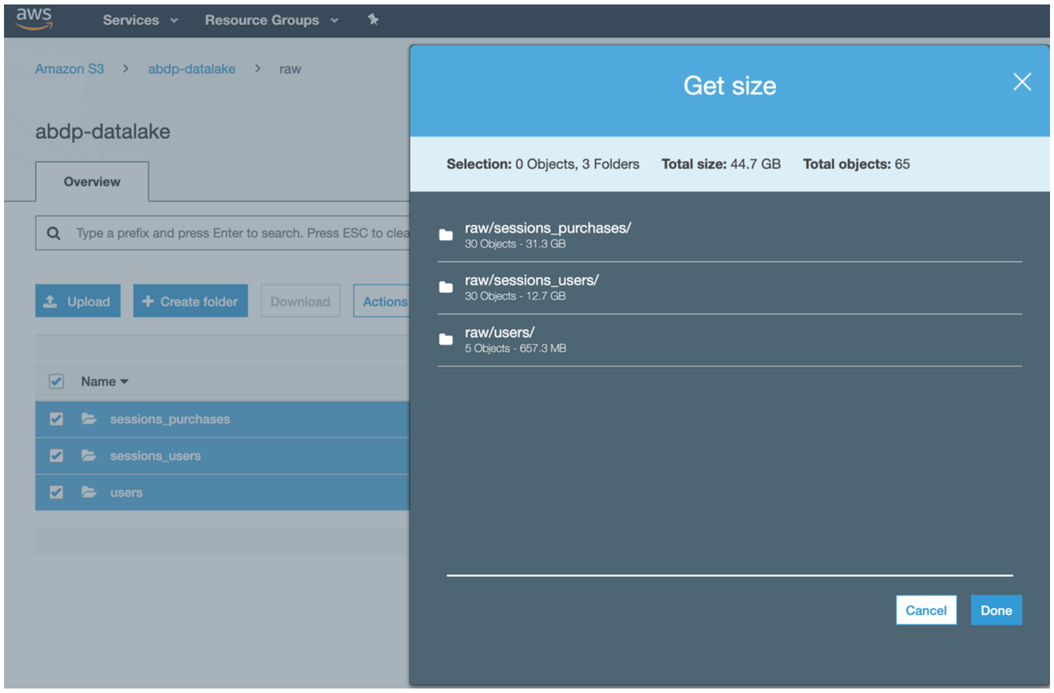
After clicking the create button, the pipeline creation kicks off. As described in previous chapters, the process extracts the metadata from the files, and migrates to the data lake in S3.



```
[INFO] Copying /app/root/home/ec2-user/data_generator/users/users_000.csv to bucket abdp-datalake with 8 threads...
/app/root/home/ec2-user/data_generator/users/users_000.csv  136235334 / 136235334.0  (100.00%)[INFO] Operation took 3.16 seconds
[INFO] Copying /app/root/home/ec2-user/data_generator/users/users_001.csv to bucket abdp-datalake with 8 threads...
/app/root/home/ec2-user/data_generator/users/users_001.csv  137503140 / 137503140.0  (100.00%)[INFO] Operation took 1.15 seconds
[INFO] Copying /app/root/home/ec2-user/data_generator/users/users_002.csv to bucket abdp-datalake with 8 threads...
/app/root/home/ec2-user/data_generator/users/users_002.csv  138285707 / 138285707.0  (100.00%)[INFO] Operation took 1.34 seconds
[INFO] Copying /app/root/home/ec2-user/data_generator/users/users_003.csv to bucket abdp-datalake with 8 threads...
/app/root/home/ec2-user/data_generator/users/users_003.csv  138391839 / 138391839.0  (100.00%)[INFO] Operation took 1.37 seconds
[INFO] Copying /app/root/home/ec2-user/data_generator/users/users_004.csv to bucket abdp-datalake with 8 threads...
/app/root/home/ec2-user/data_generator/users/users_004.csv  138769277 / 138769277.0  (100.00%)[INFO] Operation took 1.16 seconds
[INFO] Copying /app/root/home/ec2-user/data_generator/sessions_users/sessions_users_000.json to bucket abdp-datalake with 8 threads...
/app/root/home/ec2-user/data_generator/sessions_users/sessions_users_000.json  437042825 / 437042825.0  (100.00%)[INFO] Operation took 2.78 seconds
[INFO] Copying /app/root/home/ec2-user/data_generator/sessions_users/sessions_users_001.json to bucket abdp-datalake with 8 threads...
/app/root/home/ec2-user/data_generator/sessions_users/sessions_users_001.json  448153199 / 448153199.0  (100.00%)[INFO] Operation took 4.89 seconds
[INFO] Copying /app/root/home/ec2-user/data_generator/sessions_users/sessions_users_002.json to bucket abdp-datalake with 8 threads...
/app/root/home/ec2-user/data_generator/sessions_users/sessions_users_002.json  448154745 / 448154745.0  (100.00%)[INFO] Operation took 3.17 seconds
[INFO] Copying /app/root/home/ec2-user/data_generator/sessions_users/sessions_users_003.json to bucket abdp-datalake with 8 threads...
```

Figure 26 - job_scheduler multi-part copy progress

Once the migration is done, we can take a look inside the abdp-datalake bucket, under the raw folder, to find the data successfully stored in folder with the name of the tables.



Selection: 0 Objects, 3 Folders **Total size:** 44.7 GB **Total objects:** 65

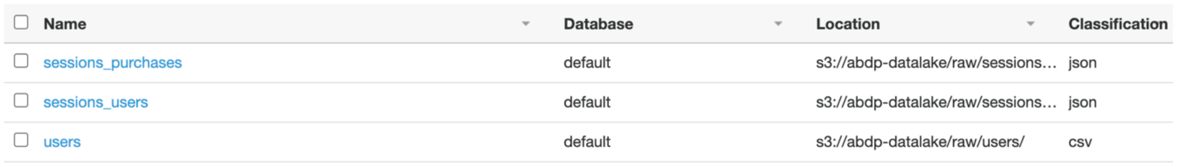
raw/sessions_purchases/
30 Objects - 31.3 GB

raw/sessions_users/
30 Objects - 12.7 GB

raw/users/
5 Objects - 657.3 MB

Figure 27 - Data sets migrated to the data lake

After that, a set of AWS Glue crawlers are created (if not existing yet) to scan the folders and create Glue tables on top of the data, which will then be used by Spark to process the data sets as tables.

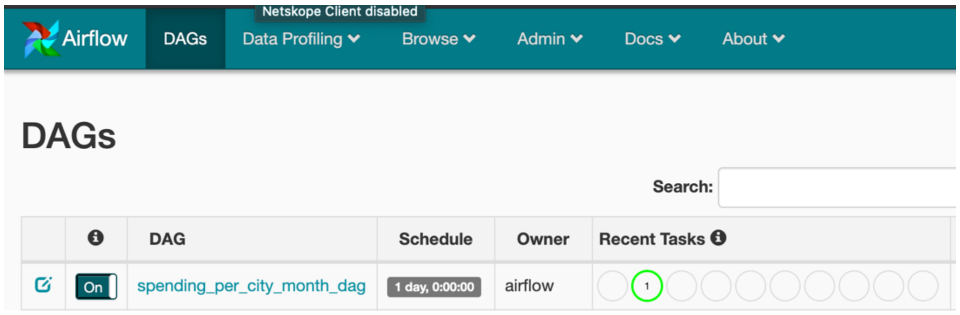


Name	Database	Location	Classification
sessions_purchases	default	s3://abdp-datalake/raw/sessions…	json
sessions_users	default	s3://abdp-datalake/raw/sessions…	json
users	default	s3://abdp-datalake/raw/users/	csv

Figure 28 - Data sets available as Glue tables

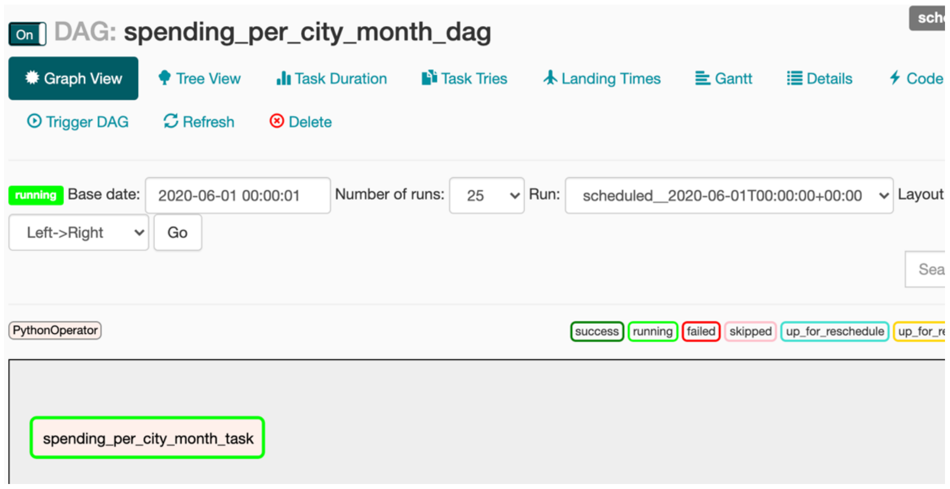
Once the crawlers finished and the new tables are available, the job scheduler coordinates with the dag admin to build a new Airflow DAG. The DAG is based on a template available in the service. The dag admin replaces the placeholder configurations with the actual values sent from the data center and so the Airflow Scheduler pick it up and is now shown in the Airflow webserver.

From the adaptive big data pipeline's webserver, the user can click on View Scheduling and to visualize new DAG on Airflow's main panel. By default, Airflow will execute the job after being incorporated to the DAG bag.



		DAG	Schedule	Owner	Recent Tasks
	On	spending_per_city_month_dag	1 day, 0:00:00	airflow	1

Figure 29 - New DAG created and visible from Airflow webserver UI



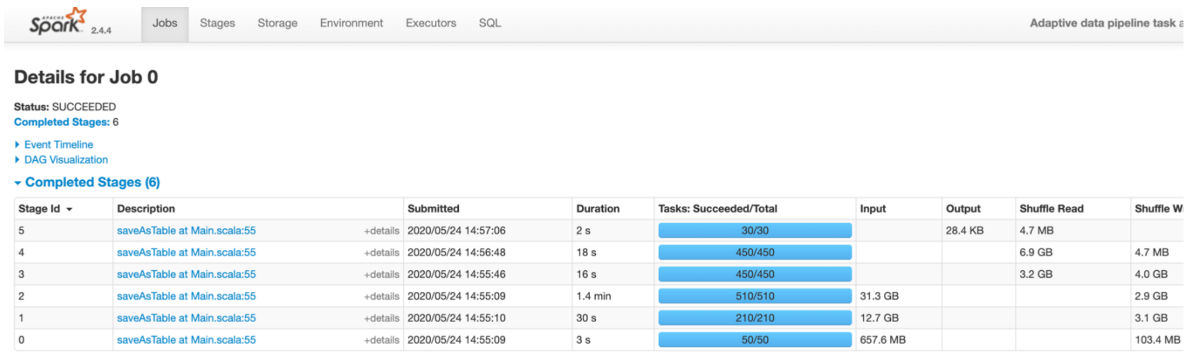
running Base date: 2020-06-01 00:00:01 Number of runs: 25 Run: scheduled__2020-06-01T00:00:00+00:00 Layout: Left->Right

Figure 30 - Pipeline task being automatically executed by Airflow

The airflow job will automatically create an ephemeral EMR cluster and pass the configurations on to the spark job as command line arguments.

As part of the cluster setup, the number of nodes is determined by the number of partitions, which in turn vary according to the data volume. This ensures that the maximum resource allocation is attained.

The Spark job that is automatically executed by EMR consists of a template, built in Scala, that executes a SQL transformation passed on as command line argument. The job executes it using the Spark SQL, then dumps the result to both S3 and Hive metastore (on AWS Glue). Once done, the cluster automatically terminates to not incur in additional charges.



Details for Job 0

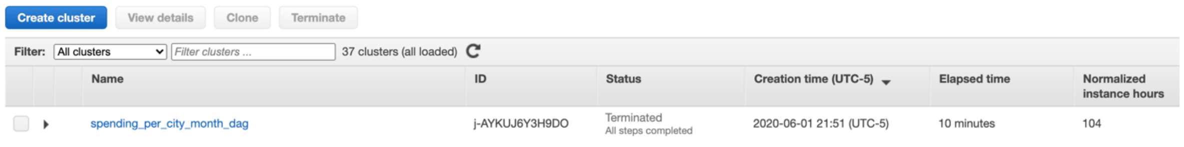
Status: SUCCEEDED
Completed Stages: 6

- Event Timeline
- DAG Visualization

Completed Stages (6)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle W
5	saveAsTable at Main.scala:55	+details	2020/05/24 14:57:06	2 s	30/30		28.4 KB	4.7 MB	
4	saveAsTable at Main.scala:55	+details	2020/05/24 14:56:48	18 s	450/450			6.9 GB	4.7 MB
3	saveAsTable at Main.scala:55	+details	2020/05/24 14:55:46	16 s	450/450			3.2 GB	4.0 GB
2	saveAsTable at Main.scala:55	+details	2020/05/24 14:55:09	1.4 min	510/510	31.3 GB			2.9 GB
1	saveAsTable at Main.scala:55	+details	2020/05/24 14:55:10	30 s	210/210	12.7 GB			3.1 GB
0	saveAsTable at Main.scala:55	+details	2020/05/24 14:55:09	3 s	50/50	657.6 MB			103.4 MB

Figure 31 - Spark stage's metrics after processing the transformation

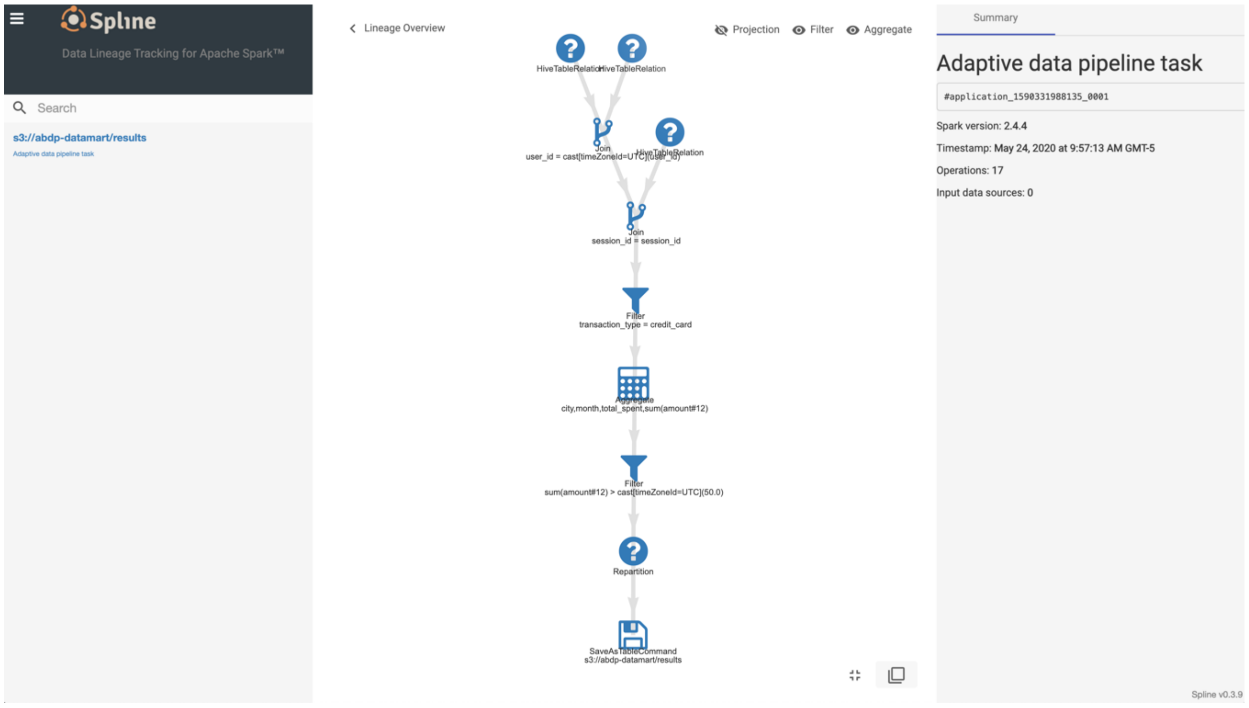


Filter: All clusters | Filter clusters ... | 37 clusters (all loaded)

		Name	ID	Status	Creation time (UTC-5)	Elapsed time	Normalized instance hours
		spending_per_city_month_dag	j-AYKUJ6Y3H9DO	Terminated All steps completed	2020-06-01 21:51 (UTC-5)	10 minutes	104

Figure 32 - EMR cluster terminated successfully after processing

The data lineage is automatically written in Spline. Once the Spark job finishes, the user can access it by clicking on View Lineage from the webserver's page. In Spline, all transformations for a given job can be visualized by clicking on the left pane, on the output name.


Spline
Data Lineage Tracking for Apache Spark™
Search
s3://abdp-datamart/results
Adaptive data pipeline task
Lineage Overview
Projection
Filter
Aggregate
HiveTableRelation
HiveTableRelation
Join
user_id = cast[timeZoneId=UTC](user_id)
HiveTableRelation
Join
session_id = session_id
Filter
transaction_type = credit_card
Aggregate
city,month,total_spent,sum(amount#12)
Filter
sum(amount#12) > cast[timeZoneId=UTC](50.0)
Repartition
SaveAsTableCommand
s3://abdp-datamart/results
Summary
Spline v0.3.9

Adaptive data pipeline task

#application_1590331988135_0001

Spark version: 2.4.4
Timestamp: May 24, 2020 at 9:57:13 AM GMT-5
Operations: 17
Input data sources: 0

Figure 33 - Pipeline's anatomy on Spline

6.5. Benchmark

In order to have a perspective as to the benefits of this solution, 3 different business scenarios are put forward in which an end-to-end cloud-based big data pipeline is required:

1. Using the Adaptive Big Data Pipelines
2. Having a Data Engineer and a DevOps Engineer
3. Having a Data Analysts

For each scenario, we showcase 2 different metrics:

1. Cost in Mexican pesos
2. Time to get the insights

In the following sub sections, the main assumptions are presented for each scenario.

6.5.1. Adaptive Big Data Pipelines

In this scenario, the company already has either a business or a data analyst. According to Glassdoor [92], the salary of a Data Analyst is around 21,000 Mexican pesos.

Now, since ABDP is an open source tool, companies do not incur in costs using it.

According to the AWS billing page, in April—the month I was doing most of my tests—the total cost of of processing 45 GB was 14.8 USD, or 329 Mexican pesos.

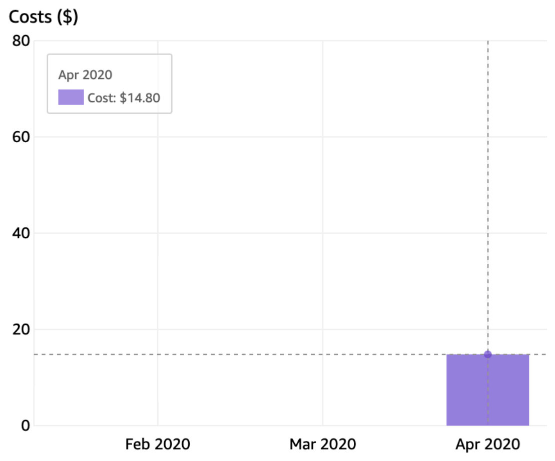


Figure 34 - AWS Costs ABDP April

The average time it takes for a data analyst to have results ready to query on Athena, once the SQL query is ready, breaks down to:

- 3 mins filling in the web form
- 8 minutes cloud setup
- 4 mins file copy
- 8 minutes data processing

Which add up to 23 minutes.

In conclusion:

- Employee costs – 21,000 Mexican pesos.
- Cloud costs – 329 Mexican pesos.
- Time – 23 minutes.

6.5.2. Data Engineer and DevOps Engineer

According to Glassdoor [93] [94], Data Engineers make on average 26,000 Mexican pesos, whereas a DevOps Engineer earns over 46,000 Mexican pesos.

For the sake of simplicity, we add 100 USD on cloud costs for the experimentation, or 2,227.22 pesos, to the original 329 pesos, which adds up to 2,256 Mexican pesos the first month.

In the consulting firms I have worked on, I have seen a couple projects with similar end-to-end pipelines requirements. On average this will be the epics and the time estimation:

Epic	Duration	Responsibility
Set up cloud infrastructure	3 weeks	DevOps
Set up repositories	1 week	DevOps
Develop Spark SQL job	2 weeks	Data Engineer
Install and configure Airflow	2 weeks	Data Engineer
Tests and fine tune of the pipeline	2 weeks	Data Engineer

Table 9 - Epics to build and end-to-end pipeline

Since the DevOps and Data Engineer can work independently for the most part, we assign a maximum of 2 months for project completion. This translates in 52,000 Mexican pesos for the Data Engineer and 92,000 Mexican pesos for the DevOps Engineer. Both salaries add up to 144,000 Mexican pesos.

Since the Data Analyst is the one designing the SQL query, we incorporate him as per the ABDP estimation.

In conclusion:

- Employee cost – 144,000 + 42,000 = 186,000 Mexican pesos.
- Cloud cost – 2256 + 329 = 2585 Mexican pesos.
- Time – 6 weeks.

6.5.3. Data Analyst

In this last scenario, we consider the same salary pointed out in the ABDP section for a single data analyst across all the project. The following table shows the activities the analysts will have to work on in a sequential manner. The first ones involve training, for the sake of simplicity, will incur in an additional 500 USD or 11,130 Mexican pesos.

Epic	Duration
AWS Fundamentals	4 weeks
Security Fundamental	2 weeks
Big Data Fundamentals	6 weeks
Spark, Airflow, Terraform and Git	6 weeks
Set up cloud infrastructure	3 weeks
Set up repositories	1 week
Develop Spark SQL job	2 weeks
Install and configure Airflow	2 weeks
Tests and fine tune of the pipeline	2 weeks

Table 10 - Data Analyst Epics

Which add up to 28 weeks (7 months).

In conclusion:

- Employee cost – 21,000 * 7 = 147,000 Mexican pesos.
- Cloud cost – 2256 + 329 * 6 = 4,230 Mexican pesos.
- Training cost – 11,130 Mexican pesos.
- Time – 28 weeks.

6.6. Summary

Aggregating the above sub section's results, both costs and time estimation, we get the following table:

Scenario	Cost (Thousand pesos)	Time (days)
ABDP	21.3	0.015
DataEng + DevOps	188.5	42
Data Analyst	162.3	196

Table 11 - Cost and time benchmark

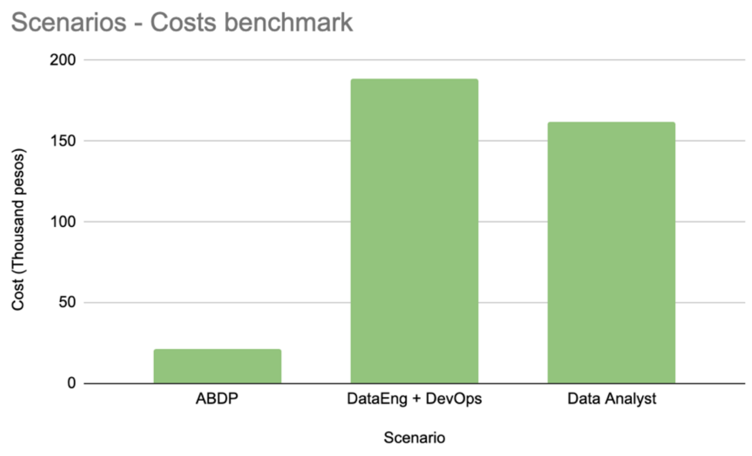


Figure 35 - Costs benchmark

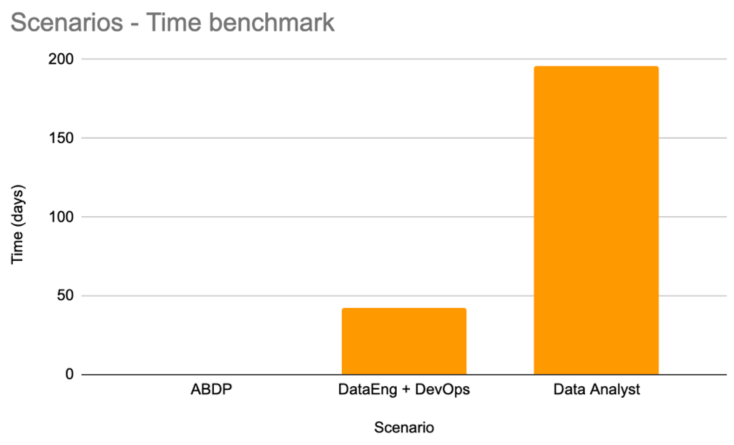


Figure 36 - Time benchmark

As observed in the benchmark results, there was a dramatic reduction of both time (99.96% and 99.99%, respectively) and costs (88.7% and 86.8% respectively).

Given all the above, we demonstrated how easy it is for business, and data analysts with no knowledge on distributed systems, to construct scalable pipelines with significant cost and time savings.

7. CONCLUSIONS

7.1. Conclusions

The ABDP is an open-source initiative. This is granted that all tools this project leverages are open source, and I also believe that the only way technology has been able to flourish exponentially is thanks to the community support and contributions. In consequence, the repository had to be available publicly—and in fact is, at the time of writing. While transitioning from a private repository to a public one, I learned that using plain text files or environment variables to share AWS credentials between services was an eyesore practice. At that point, I started migrating towards IAM roles, which could be easily managed from the infrastructure microservice. This ensured that credentials were directly attached to the services that require access to other services in a secure way.

The first obstacle I faced while investigating the state of the art for this project is the lack of academic research on practical big data tool integration. Most research papers and books would either cover one specific tool—explaining how it works internally and architectural decisions—or elaborate on how they combined a few tools to solve a unique use case. Likewise, when compared to similar data-related areas such as machine learning, I observed the latter to be more compelling for academia, as it has a stronger mathematical component to it. All this forced me to underpin a good portion of this project on either online blogs or some niche books, which is often times seen with contempt by the academia.

At the inception of this project, back in August 2018, my experience—and by extension, my knowledge—on big data tools was not particularly broad or deep. As with most thesis, I recognize there were several assumptions and initial misconceptions that drove me towards cul-de-sacs, which translated into countless months spent prototyping and discarding a fair number of tools. On the one side, this forced me to narrow down the scope of the system to account for the uncertainty. On the other hand, I was able to integrate more cutting-edge tools, not originally envisioned in order to provide more usability to the end user.

The pipeline fine-tuning involved a simple algorithm to determine right number of partitions the EMR Spark job was going to use for shuffling operations—join and group by. Originally, Spark was intended to read the number of records and schema, in order to determine the partition count. However, since the execution of Spark was local, when working with dozens of GB, the execution took a lot of time or even fail due to lack of memory. Plus, those metadata did not practically improve the pipeline execution. Ultimately, I decided to boil down the algorithm to the total file size.

When dealing with the public cloud, one important metric to bear in mind is costs. One huge downside of the pay-as-you-go model is that you may inadvertently spend hundreds of dollars in just a few days. Even though I was fully aware of the costs most of the services entail, while experimenting some eventuality happened, and my costs would soar—and I ended up pulling my hair off when the bill arrived. So, I would have saved a few dozen dollars by reserving instances instead of using on-demand instances.

AWS provides a staggering number of services to allow for scalable and powerful application development, ranging from Infrastructure as Code all the way to Serverless (e.g., Lambda, Kinesis, Glue). Even though Serverless may look enticing, it can easily cause vendor lock-in. In my professional experience, I have seen several times customers trying to migrate from one platform to another. The tighter the coupling with the cloud provider services, the longer the migration is going to take. This was the main motivation behind using open-source tools, including Terraform for the infrastructure: in this way, if someone wants to implement ABDP in, say, Google Cloud Platform, he/she only needs to mirror the components in the infrastructure microservice using the exact same syntaxis.

In my experience, by utilizing thousands of small files—roughly 100KB each—the pipeline's performance is severely impacted, as Hive tables need to serialize and deserialize more files in order to provide the data to Spark. To attain the ideal performance file sizes were above 100 MB should be favored. On the other hand, if the data sets are small—less than 100 MB in size—and I zipped them into a single file, that would also worsen the pipeline's performance.

When working with a microservice architecture and a client-side JavaScript artifact, I discovered that JavaScript cannot directly refer to the DNS names provided by docker compose, as it lies outside the services scope; the web server, on the other hand, does have access to the microservices by DNS name.

The most perplexing thing I found while looking for data to work with for this project is the lack of good quality data on internet. I struggled for months to find something remotely interesting. Platforms such as Kaggle are flooded with datasets that are either small or are unstructured formats (e.g., text files or images). Because of this, I decided to create my own scripts to generate as much data as I wanted. One of the challenges of following this path is that I had to emulate real-world tendencies using statistics, which led me to the binomial distributions. However, as they are computationally expensive over large datasets (i.e., over hundreds of billions of records), the task of generating such data becomes challenging. In the end, I was not able to represent real world tendencies with the granularity I desired, but it was acceptable enough for the experimentation.

Another architectural pattern I understood and implemented down the road is microservices. Originally, I started the implementation as a huge monolith, similar to what most small company would do if not properly guided. However, soon I discovered that dependencies were all over the place and it was going to become impossible to manage. Therefore, I revamped the architecture to leverage docker compose. At first it was complex, but it became easier to manage on the long run. In this way I also achieved separation of concerns and isolation, two crucial factors of any scalable architecture.

The input data formats supported in this project (i.e., JSON and CSV) proved to be quite inefficient, as the benchmark with Spark showcased only a small portion of the real file sizes is being scanned when consuming the tables. Furthermore, if files were in parquet format, they read would have been much faster. However, I purposefully avoided using parquet as no company out there—unless it is considerably data mature—has data in parquet format, but rather JSON or CSV.

Several hundred—if not thousand—big data projects are available over the internet. Those on the spotlight are the ones that allegedly have had more adoption by the community and had been commonly incubated by the Apache foundation. With that many options at hand, most tools gradually died out over the years.

I noticed that the main reason was the poor adaptability to cloud environments and non-Hadoop-compliant tools; to name a few: Apache Oozie, Apache Atlas, Apache Falcon and Apache Pig.

To my surprise, there were a few highly interesting projects that, despite being under the radar, may be extremely useful for real-world scenarios. Such is the case for Spline, a tool I found two weeks before presenting the initial demo to my tutor. I learned that tools that seamlessly integrates with other tools is crucial to get attention from the community.

By working with Apache Airflow, I learned how quickly open-source projects are developing. In a matter of two years, many things changed, including the variety of operators available. As briefly mentioned in the future work section, I started knowing that Bash and Python operators existed. Six months ago I discovered that the amount of operators is ever-increasing and are suitable for more specific use cases: MySQL, Redshift, BigQuery, EMR, to name a few.

Data is the crux of this project. As such, I decided to include as much of the logic as possible within private subnets inside AWS. This becomes complicated to manage at first, as you have to incorporate additional components to make those instances have access to internet securely, including a NAT gateway and an internet gateway, as well as the appropriate routing tables.

Finally, I realized that data migration to the cloud may be a slow process if you do not have the right network infrastructure. I noticed that my network connection was causing all sorts of problems when migrating files and since no failover mechanism was implemented, upon a failure I would have to start the migration all over again. Hence, when data volumes exceed 1 TB, AWS recommends using Snowball, which is a special appliance to transfer large volumes of data to S3 physically.

7.2. Future work

In virtue of this project primary focus on big data, the web UI—or more precisely, the web form—was considered a means to only enter pipeline information, and easily access both the lineage and scheduling. Therefore, the UI lacks some basic functionality, among which I consider the following as the most important ones:

- Multiple user log in.
- A way to keep track of the process of pipeline creation and execution.
- A screen to inspect, edit and remove existing pipelines.
- A textbox area to issue queries to AWS Athena.
- A panel to keep track of the health of underlying services.
- An option to tear down infrastructure modules—super server and foundation—on demand.

For simplicity, this project assumed a single server instance—called super server—in order to host all remote microservices. Splitting this into multiple instances, using either ECS or EKS (managed Kubernetes service) can circumvent the single point of failure.

Another restriction of this project is the IAM policies. For ease of testing, we provided coarse permission boundaries (e.g., full access to S3, IAM, EC2). This is far from ideal for production environments. Therefore, further investigation is necessary to narrow down those policies to what is truly required by every service—restricting all the way down to the resources.

Another important feature to incorporate is VPN integration with the security groups, so that all remote services, including scheduling and lineage, can be accessed from the VPN no matter where the employee is working, as long as it has access to the VPN. Currently, the only whitelisted public IP is the one the from where the infrastructure is built.

During the investigation and implementation of this project, a rather new feature of Airflow gained momentum, and remained under my radar: EMR operators. Such operators facilitate the EMR cluster creation and monitoring, instead of having custom boto3 code to perform such actions on your behalf.

Furthermore, early experiments on Airflow seemed to indicate that a fully automated Airflow DAG (i.e., a Python script) was not entirely feasible. The problem is the way Airflow reads the files and fills the DAG bag, as it needs to locate a DAG object within the main script. In consequence, I had to leave most configurations as hardcoded variables that are automatically replaced once the DAG is created based on the template. A configuration-based approach would be preferable in this case, to allow for future modifications.

Another fixed element of this architecture is the amount of partitions that Spark calculates for the ongoing data ingestion. Partition count should be recalculated once new data comes in to preserve the pipeline efficiency.

This project only accounts for one-time only data migrations. Since the scheduling service is meant to process historical data commonly on a daily basis, a mechanism would need to be implemented to constantly and intelligently migrate any data generated in the data center to the data lake.

I recently stumbled upon AWS KMS, a service to manage encryption keys. This feature may be included in the infrastructure microservice in order to enable server-side at-rest encryption and secure data even further, to consider highly sensitive data.

Another interesting feature to include is a way to turn on and off independent services when they are not in use, hence avoiding unnecessary expenses. Alternatively, the use of serverless options such as Lambda could be considered.

From professional experience, I learned that when working with high volumes of data in private subnets, costs may dramatically increase. The current implementation uses a NAT gateway to route traffic to the internet. However, most of the traffic the EMR cluster would require is both to S3 and Glue. VPC endpoints could be used and associated to the routing table in order to force local traffic through them. This would not only reduce costs, as the VPC endpoints traffic costs is a fraction compared to the NAT gateway ones, but it also ensures that privately handled data only flows to those specific services without leaving AWS' infrastructure. This would be extremely beneficial for highly sensitive data or PII.

REFERENCES

[1] S. Ghemawat, et. al., “The Google File System”, 2003

[2] J. Dean, S. Ghemawat; “MapReduce: Simplified Data Processing on Large Clusters”, 2004

[3] S. Melnik, et al., “Dremel: Interactive Analysis of Web-Scale Datasets”, 2010

[4] S. Bappalige, *An introduction to Apache Hadoop for big data*, August 2014. Accessed on: March 2020. [Online]. Available: https://opensource.com/life/14/8/intro-apache-hadoop-big-data

[5] Apache Software Foundation, *Apache Hadoop*. Accessed on: February 2020. [Online]. Available: https://hadoop.apache.org/

[6] H. Karambelkar, *Apache Hadoop 3 Quick Start Guide*, Packt, 2018

[7] K. Shvachko, et al., "The Hadoop Distributed File System", 2010

[8] K. Goel, *Introduction to Hadoop*, December 2019. Accessed on: April 2020. [Online]. Available: https://intellipaat.com/blog/tutorial/hadoop-tutorial/introduction-hadoop/

[9] Apache Software Foundation, *Apache Hive*. Accessed on: January 2020. [Online]. Available: https://hive.apache.org

[10] A. Thusoo, et al., "Hive - a petabyte scale data warehouse using Hadoop", 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), Long Beach, CA, 2010, pp. 996-1005.

[11] Dataflair team, *Hive Metastore – Different Ways to Configure Hive Metastore*, March 2020. Accessed on: April 2020. [Online]. Available: https://data-flair.training/blogs/apache-hive-metastore/

[12] B. Black, *EC2 origins*, January 2019. Accessed on: March 2020. [Online]. Available: http://blog.b3k.us/2009/01/25/ec2-origins.html

[13] K. Stalcup, *AWS vs Azure vs Google Cloud Market Share 2020: What the Latest Data Shows*, February 2020. Accessed on: March 2020. [Online]. Available: https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/

[14] AWS, *What is AWS*, 2020. Accessed on: February 2020. [Online]. Available: https://aws.amazon.com/what-is-aws/

[15] AWS, *Cloud Products*, 2020. Accessed on: February 2020. [Online]. Available: https://aws.amazon.com/products/

[16] AWS, *Benefits at a Glance*, 2020. Accessed on: March 2020. [Online]. Available: https://aws.amazon.com/application-hosting/benefits/

[17] AWS, *Amazon EC2*, 2020. Accessed on: July 2020. [Online]. Available: https://aws.amazon.com/ec2/

[18] AWS, *Amazon S3*, 2020. Accessed on: July 2020. [Online]. Available: https://aws.amazon.com/s3/

[19] AWS, *AWS Identity and Access Management*, 2020. Accessed on: July 2020. [Online]. Available: https://aws.amazon.com/iam/

[20] AWS, *Amazon Virtual Private Cloud*, 2020. Accessed on: July 2020. [Online]. Available: https://aws.amazon.com/vpc/

[21] AWS, *Amazon EMR*, 2020. Accessed on: July 2020. [Online]. Available: https://aws.amazon.com/emr/

[22] AWS, *AWS Glue*, 2020. Accessed on: July 2020. [Online]. Available: https://aws.amazon.com/glue

[23] AWS, *Using the AWS Glue Data Catalog as the Metastore for Spark SQL*, 2020. Accessed on: July 2020. [Online]. Available: https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-glue.html

[24] AWS, *Amazon Athena*, 2020. Accessed on: July 2020. [Online]. Available: https://aws.amazon.com/athena

[25] "*Apache Airflow Documentation*", Accessed on: April 2020. [Online]. Available: https://airflow.apache.org/docs/stable/

[26] "*Concepts*", Accessed on: April 2020. [Online]. Available: https://airflow.apache.org/docs/stable/concepts.html#operators

[27] M. Zaharia, et al., "Spark: Cluster Computing with Working Sets", University of California, Berkeley, 2009

[28] Apache Software Foundation, *Apache Spark*, 2020. Accessed on: January 2020. [Online]. Available: https://spark.apache.org/

[29] H. Karau, et al., *Learning Spark, lightning-fast data analysis*, 2015, O'Reilly Media, Inc.

[30] J. Damji, *A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets*, July 2016. Accessed on: February 2020. [Online]. Available: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

[31] Apache Software Foundation, *Apache Arrow*, 2020. Accessed on: March 2020. [Online]. Available: https://arrow.apache.org/

[32] B. Cutler, *Speed up PySpark with Apache Arrow*, July 2017. Accessed on: March 2020. [Online]. Available: https://arrow.apache.org/blog/2017/07/26/spark-arrow/

[33] Apache Software Foundation, *Parquet Documentation*, 2018. Accessed on: March 2020. [Online]. Available: https://parquet.apache.org/documentation/latest/

[34] "*Big Data file formats*", April 2020. Accessed on: April 2020. [Online]. Available: https://luminousmen.com/post/big-data-file-formats

[35] AWS, *Introduction to Presto (PrestoDB).* Accessed on: April 2020. [Online]. Available: https://aws.amazon.com/big-data/what-is-presto/

[36] R. Sethi *et al.*, "Presto: SQL on Everything," *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, Macao, Macao, 2019, pp. 1802-1813.

[37] "*Apache Atlas – Overview*". Accessed on: February 2020. [Online]. Available: https://atlas.apache.org/#/

[38] "*Apache Atlas – High Level Architecture*", June 2019. Accessed on: February 2020. [Online]. Available: https://atlas.apache.org/2.0.0/Architecture.html

[39] S. Gopalani and R. Arora, "Comparing apache spark and map reduce with performance analysis using k-means," International Journal of Computer Applications, vol. 113, no. 1, 2015.

[40] "*Spline – Data Lineage Tracking and Visualization Solution*", 2020. Accessed on: February 2020. [Online]. Available: https://absaoss.github.io/spline/

[41] Terraform, *Introduction to Terraform*. Accessed on: February 2020. [Online]. Available: https://www.terraform.io/intro/index.html#what-is-terraform-

[42] S. Madurapperuma, *The Essential Git Handbook*, April 2019. Accessed on: March 2020. [Online]. Available: https://www.freecodecamp.org/news/the-essential-git-handbook-a1cf77ed11b5/

[43] Bitbull, *Git Flow: how it works*, November 2016. Accessed on: February 2020. [Online]. Available: https://www.bitbull.it/en/blog/how-git-flow-works/

[44] Preeth E N, F. J. P. Mulerickal, B. Paul and Y. Sastri, "Evaluation of Docker containers based on hardware utilization," 2015 International Conference on Control Communication & Computing India (ICCC), Trivandrum, 2015, pp. 697-700.

[45] J. Nickoloff and S. Kuenzli, *Docker in Action, Second Edition*, October 2019, Manning

[46] B. Burns, et al., *Kubernetes: Up and Running, 2nd edition*, 2020, O'Reilly Media, Inc.

[47] J. Ellingwood, *An Introduction to Kubernetes*, May 2018. Accessed on: May 2020. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes

[48] C. Brooks, *What is SQL?*, January 2014. Accessed on: February 2020. [Online]. Available: https://www.businessnewsdaily.com/5804-what-is-sql.html, "What is SQL?"

[49] E. Codd, "A Relational Model of Data for Large Shared Data Banks", 1970

[50] P. Xavier, *An Absolute Beginners Guide to SQL*, January 2019. Accessed on: February 2020. [Online]. Available: https://medium.com/better-programming/absolute-beginners-guide-to-sql-601aad53f6c9

[51] IBM, *ACID properties of transactions*, April 2020. Accessed April 2020. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.4.0/product-overview/acid.html

[52] Guru99, *What is Data Warehouses? Types, Definition & Example*. Accessed on: March 2020. [Online]. Avaliable: https://www.guru99.com/data-warehousing.html

[53] N. A. Farooqui and R. Mehra, "Design of A Data Warehouse for Medical Information System Using Data Mining Techniques," 2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC), Solan Himachal Pradesh, India, 2018, pp. 199-203.

[54] I. Sutedja, P. Yudha, N. Khotimah and C. Vasthi, "Building a Data Warehouse to Support Active Student Management: Analysis and Design," 2018 International Conference on Information Management and Technology (ICIMTech), Jakarta, 2018, pp. 460-465.

[55] Diouf P., Boly A., et. al.; Variety of data in the ETL processes in the cloud: State of the art; 2018 IEEE International Conference on Innovative Research and Development (ICIRD); May 2018

[56] R. Kimball, J. Caserta, *The Data Warehourse ETL Toolkit*, 2011, Wiley

[57] B. Zhu, et al., "Electromagnetic radiation study of Intel Dual Die CPU with heatsink," 2008 8th International Symposium on Antennas, Propagation and EM Theory, Kunming, 2008, pp. 949-952

[58] A. Thompson, *Scientists Have Made Transistors Smaller Than We Thought Possible*, October 2016. Accessed on: March 2020. [Online]. Available: https://www.popularmechanics.com/technology/a23353/1nm-transistor-gate

[59] C. Roy, M. Pandey and S. SwarupRautaray, "A Proposal for Optimization of Data Node by Horizontal Scaling of Name Node Using Big Data Tools," 2018 3rd International Conference for Convergence in Technology (I2CT), Pune, 2018, pp. 1-6.

[60] S. Singh, *Understanding The Difference Between Horizontal & Vertical Scaling*, December 2019. Accessed on: March 2020. [Online]. Available: https://www.redswitches.com/blog/difference-between-horizontal-vertical-scaling

[61] B. Burns, *Designing Distributed Systems: Patterns and Paradigms for Scalable Reliable Services*, February 2018, O'Reilly Media, Inc.

[62] M. Kleppman, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, 2017, O'Reilly Media, Inc.

[63] M. Turck, et al., *DATA & AI LANDSCAPE 2019*, 2019. Accessed on May 2020. [Online]. Available: http://mattturck.com/wp-content/uploads/2019/07/2019_Matt_Turck_Big_Data_Landscape_Final_Fullsize.png, "A Turbulent Year: The 2019 Data & AI Landscape"

[64] Kumar J., Garg V., "Security analysis of unstructured data in NOSQL MongoDB database", 2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN), 2017

[65] "*SQL WORLD*". Accessed on: April 2020. [Online]. Available: https://www.complexsql.com/difference-between-sql-and-nosql/

[66] AWS, *What is a data lake?*, 2020. Accessed on: March 2020. [Online]. Available: https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/

[67] V. Ranjan, "A Comparative Study between ETL (Extract-Transform-Load) and E-LT (ExtractLoad-Transform) approach for loading data into a Data Warehouse", 2019

[68] M. Smallcombe, *ETL vs ELT: 5 Critical Differences*, February 2020. Accessed on: March 2020. [Online]. Available: https://www.xplenty.com/blog/etl-vs-elt/

[69] H. Cao, M. Brown, L. Chen, R. Smith and M. Wachowicz, "Lessons Learned from Integrating Batch and Stream Processing using IoT Data," 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), Granada, Spain, 2019, pp. 32-34.

[70] Kim H., Cho J., "Data Governance Framework for Big Data Implementation with a Case of Korea", 2017 IEEE International Congress on Big Data (BigData Congress), 2017

[71] I. Sacolick, *What is Data Governance? Data practices that address risk and drive opportunities*, 2018. Accessed on: April 2020. [Online]. Available: https://blogs.starcio.com/2018/07/what-is-data-governance.html

[72] Tang M., Shao S., et. al.; "SAC: A System for Big Data Lineage Tracking", 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019

[73] AWS, *What is cloud computing?*. Accessed on: May 2020. [Online]. Available: https://aws.amazon.com/what-is-cloud-computing/

[74] B. Abbasov, "Cloud Computing: State Of The Art Research Issues", Conference on Application of Information and Communication Technologies (AICT), 2014

[75] Microsoft, *Cloud Service Models (IaaS, PaaS, SaaS) Diagram*, September 2018. Accessed on: March 2020. [Online]. Available: https://blogs.msdn.microsoft.com/dachou/2018/09/28/cloud-service-models-iaas-paas-saas-diagram/

[76] J. Sandobalín, E. Insfran and S. Abrahão, "On the Effectiveness of Tools to Support Infrastructure as Code: Model-Driven Versus Code-Centric," in IEEE Access, vol. 8, pp. 17734-17761, 2020.

[77] Y. Brikman, Terraform: Up & Running, 2nd Edition, September 2019, O'Reilly Media, Inc.

[78] C. Richardson, Microservices Patterns: With Examples in Java, November 2018, Manning

[79] M. M. Rahman and M. Hasibul Hasan, "Serverless Architecture for Big Data Analytics," 2019 Global Conference for Advancement in Technology (GCAT), BANGALURU, India, 2019, pp. 1-5.

[80] A. Parres-Peredo, I. Piza-Davila and F. Cervantes, "Building and Evaluating User Network Profiles for Cybersecurity Using Serverless Architecture," 2019 42nd International Conference on Telecommunications and Signal Processing (TSP), Budapest, Hungary, 2019, pp. 164-167.

[81] "*What is REST*". Accessed on: May 2020. [Online]. Available: https://restfulapi.net/

[82] S. Ben, *What is REST – A Simple Explanation for Beginners, Part 1: Introduction*, September 2017. Accessed on: May 2020. [Online]. Available: https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f

[83] "*HTTP Status Codes*". Accessed on: March 2020. [Online]. Available: https://restfulapi.net/http-status-codes/

[84] B. Rogojan, *What Is The Difference Between A Data Engineer And A Data Scientist*, February 2019. Accessed on: January 2020. [Online]. Available: https://towardsdatascience.com/what-is-the-difference-between-a-data-engineer-and-a-data-scientist-a25a10b91d66

[85] Sharma, Sheetal & Sarkar, Darothi & Gupta, Divya. (2012). Agile Processes and Methodologies: A Conceptual Study. International Journal on Computer Science and Engineering. 4.

[86] C. C. Silva and A. Goldman, "Agile Methods Adoption on Software Development: A Pilot Review," 2014 Agile Conference, Kissimmee, FL, 2014, pp. 64-65.

[87] Mountain Goat Sofware, *Scrum*, 2020. Accessed on: July 2020. [Online]. Available: https://www.mountaingoatsoftware.com/agile/scrum#:~:text=The%20Scrum%20model%20suggests%20that%20projects%20progress%20via%20a%20series%20of%20sprints.&text=Scrum%20methodology%20advocates%20for%20a,to%20perform%20during%20the%20sprint.

[88] Atlasian, *User Stories with Examples and Template*, 2020. Accessed on: July 2020. [Online]. Available: https://www.atlassian.com/agile/project-management/user-stories#:~:text=A%20user%20story%20is%20the,work%20in%20an%20agile%20framework.&text=software%20user's%20perspective.-,A%20user%20story%20is%20an%20informal%2C%20general%20explanation%20of%20a,value%20back%20to%20the%20customer.

[89] A. Orozco, "Adaptive Big Data Pipelines – Github Repository", January 2020. Accessed on: May 2020. [Online]. Available: https://github.com/aldoorozco/adaptive_data_pipelines

[90] T. Yiu, *Understanding the Normal Distribution (with Python)*, November 2019. Accessed on: May 2020. [Online]. Available: https://towardsdatascience.com/understanding-the-normal-distribution-with-python-e70bb855b027

[91] Datamx, *Muestra de Nombres y Apellidos Comunes en México*. Accessed on: May 2020. [Online]. Available: http://datamx.io/dataset/muestra-de-nombres-y-apellidos-comunes-en-mexico

[92] Glassdoor, *Sueldos Para Data Analyst en Mexico*. July 2020. Accessed on: July 2020. Available: https://www.glassdoor.com.mx/Sueldos/mexico-data-analyst-sueldo-SRCH_IL.0,6_IN169_KO7,19.htm?countryRedirect=true

[93] Glassdoor, *Sueldos Para Data Engineer en Mexico*. July 2020. Accessed on: July 2020. Available: https://www.glassdoor.com.mx/Sueldos/m%C3%A9xico-data-engineer-sueldo-SRCH_IL.0,6_IN169_KO7,20.htm

[94] Glassdoor, *Sueldos Para DevOps Engineer en Mexico*. July 2020. Accessed on: July 2020. Available: https://www.glassdoor.com.mx/Sueldos/devops-engineer-sueldo-SRCH_KO0,15.htm