electronics

MDPI

Article

Building Advanced Web Applications Using Data Ingestion and Data Processing Tools

Šimun Šprem [1,*], Nikola Tomažin [1], Jelena Matečić [1] and Marko Horvat [2,*]

[1] Syntio, Trg Dražena Petrovića 3, HR-10000 Zagreb, Croatia; nikola.tomazin@syntio.net (N.T.); jelena.matecic@syntio.net (J.M.)
[2] Department of Applied Computing, Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, HR-10000 Zagreb, Croatia
* Correspondence: simun.sprem@syntio.net (Š.Š.); marko.horvat3@fer.hr (M.H.)

Abstract: Today, advanced websites serve as robust data repositories that constantly collect various user-centered information and prepare it for subsequent processing. The data collected can include a wide range of important information from email addresses, usernames, and passwords to demographic information such as age, gender, and geographic location. User behavior metrics are also collected, including browsing history, click patterns, and time spent on pages, as well as different preferences like product selection, language preferences, and individual settings. Interactions, device information, transaction history, authentication data, communication logs, and various analytics and metrics contribute to the comprehensive range of user-centric information collected by websites. A method to systematically ingest and transfer such differently structured information to a central message broker is thoroughly described. In this context, a novel tool—Dataphos Publisher—for the creation of ready-to-digest data packages is presented. Data acquired from the message broker are employed for data quality analysis, storage, conversion, and downstream processing. A brief overview of the commonly used and freely available tools for data ingestion and processing is also provided.

Keywords: data engineering; big data analytics; big data management; data acquisition; data ingestion; change data capture (CDC); data analysis tools; real-time data stream processing

Citation: Šprem, Š.; Tomažin, N.; Matečić, J.; Horvat, M. Building Advanced Web Applications Using Data Ingestion and Data Processing Tools. *Electronics* **2024**, *13*, 709. https://doi.org/10.3390/electronics13040709

Academic Editors: Fabio Grandi and Ping-Feng Pai

Received: 24 December 2023
Revised: 2 February 2024
Accepted: 6 February 2024
Published: 9 February 2024

1. Introduction

In the rapidly evolving digital landscape of the modern digital age, websites are increasingly effective at collecting a wide range of user information and online interactions. This information provides a detailed picture of online behavior that includes personal details, browsing patterns, demographics, and individual preferences. The information collected from contemporary advanced web applications includes, for example, personal identifiers such as names and email addresses, user behavior (including browsing history, click patterns, and time spent on pages), demographic data (age, gender, and geographic location), and preferences (such as product selection, language preferences, and customer settings). As the size and diversity of these data continue to grow, the scientific community is looking for new tools and methods to systematically analyze and utilize this rich information corpus.

This paper deals with the latest tools and technologies utilized by advanced web applications for data management. It focuses on the process of data collection and publication to the cloud using specialized data ingestion tools. These tools are designed to efficiently collect, categorize, and transfer data while ensuring real-time updates and seamless integration with cloud infrastructures [1]. Following data collection, the focus shifts to data processing tools. These are crucial for analyzing, refining, and deriving meaningful insights from raw data. They employ algorithms and machine learning techniques to process data

and identify patterns, trends, and anomalies. The processed data are then either stored in a database for later use or routed back to the web application. This cyclical flow of data enables the web application to dynamically adapt and personalize the user experience based on real-time data analytics.

The aim of this paper is to explain how advanced web applications work, especially in the area of data engineering, focusing on the importance of data ingestion and processing in today's data-rich digital environment. First, we introduce the Dataphos Publisher platform, describe its architecture, and experimentally validate its performance advantages over traditional Change Data Capture (CDC) platforms. Also, this paper gives an overview of different data processing tools. This includes a qualitative exploration of their functionalities, strengths, and limitations, providing readers with a clear understanding of the diverse options that are available in the field. Finally, this paper also provides a practical framework for data engineers building advanced web applications and helps select the right tools for data ingestion and processing. This is meant to make it easier for readers to understand and use these technologies when developing web applications. Figure 1 illustrates an architecture proposed for this comprehensive solution, including a novel tool, Dataphos Publisher, and shows the interrelated roles of data ingestion and processing tools in modern web applications.

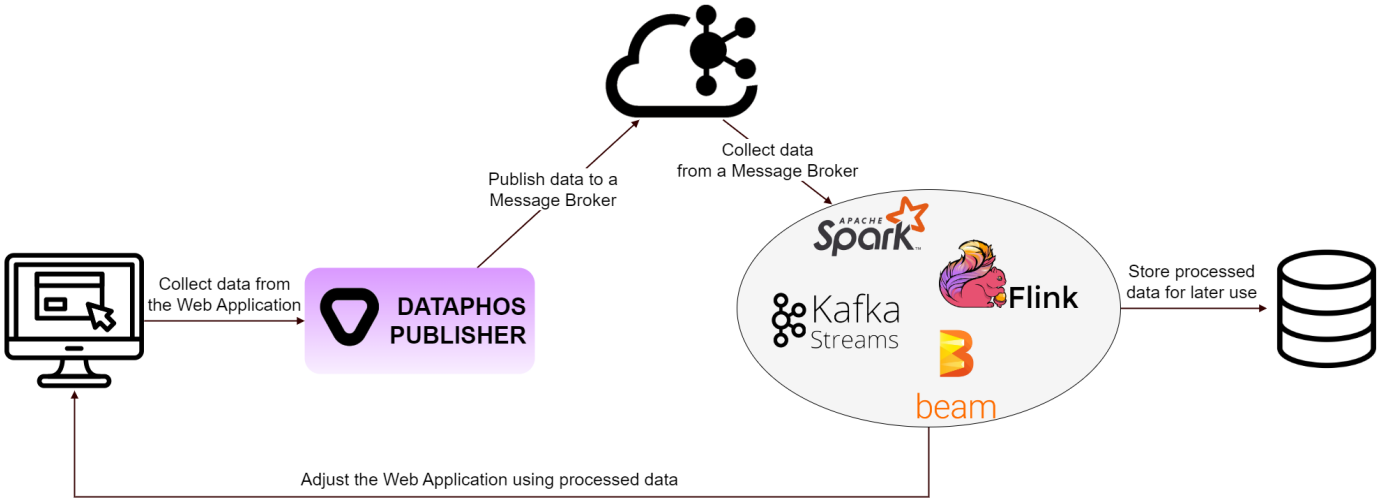

Figure 1. Advanced web applications architecture that uses Dataphos Publisher combined with a set of commonly used web-based data processing tools.

The remainder of this paper is organized as follows. Section 2 provides an in-depth analysis of data ingestion, focusing on specific technologies in web applications such as CDC and Dataphos Publisher, developed by Syntio. Section 3 explores data processing tools, comparing Apache Spark, Apache Flink, Kafka Streams, and Apache Beam. In Section 4, practical use cases are provided with a discussion to illustrate the applications of these tools in real-world scenarios. Finally, Section 5 concludes the paper by summarizing key insights and their implications for advanced web application development.

2. Data Ingestion

Data ingestion is a big data process that involves loading data from one or multiple sources to various destinations that may or may not be in the cloud. As modern systems become increasingly data-oriented, the more data are collected. This is best shown by the fact that 90% of data were collected in the last two years alone, with 2.5 quintillion bytes collected every day [2], and these numbers will likely grow in the coming years. This requires a robust infrastructure comprising data ingestion, data processing, data storage, etc., that can handle such amounts of data.

This section covers the data ingestion part of the infrastructure, the process of moving different types of data (structured, unstructured, and semi-structured) from one place to another for processing. These data might come from different sources, either internal (e.g., data collected by a company) or external (e.g., publicly available data from social networks such as Twitter/X and Facebook) [3].

The first subsection covers the state of the art of data ingestion, and we cover some popular data ingestion tools along with Change Data Capture (CDC), a data ingestion concept which is more thoroughly described in the second subsection. The third subsection covers a completely new player in the data ingestion process, Dataphos Publisher. Finaly, the last part of this section contains a comparison between CDC and Dataphos Publisher.

2.1. State of the Art in Data Ingestion Tools

The transition from on-premises to cloud-based environments has emphasized the importance of data ingestion in the world of big data. This transition is particularly challenging for organizations with extensive historical datasets that require efficient and reliable transfer to cloud platforms. To address these challenges, in this section, we examine the main capabilities and commonplace applications of state-of-the-art data ingestion tools, focusing specifically on Apache Kafka Connect (a key component of the Apache Kafka ecosystem), Apache NiFi, and Apache Flume [4], as well as Airbyte and Meltano [5]. Finally, we describe Change Data Capture (CDC) in the context of data ingestion. Each of these tools has a number of different features and functions that make the migration and management of large amounts of data in cloud environments more efficient, making them important components of the architecture of modern web applications.

Apache tools are compared in [4]. In this study, the researchers report that of all the tools compared, none is optimal for all use cases [4]. A summary of this report can be found in Table 1. It shows which tools implement a specific functionality (reliability, guaranteed delivery, data type versatility, system requirements complexity, stream ingestion, and processing support) and recommends a tool for this functionality [4]. The functionality level for each tool was qualitatively evaluated using a three-point scale: fully implemented functionality, partially implemented functionality, and non-implemented functionality. From the table data, Apache NiFi is shown to be the best tool compared to Apache Flume and Apache Kafka Connect due to its more comprehensive coverage of functionality.

Table 1. Qualitative comparison between the main functionalities of the Apache Flume, Apache NiFi, and Apache Kafka Connect data ingestion tools with a recommendation for the optimal tool for each functionality [4].

Functionality	Flume	NiFi	Kafka	Optimal Tool
Reliability	Partially implemented	Partially implemented	Fully implemented	Kafka
Guaranteed delivery	Fully implemented	Fully implemented	Partially implemented	Flume and NiFi
Data type	Partially implemented	Partially implemented	Partially implemented	Flume, NiFi, and Kafka
System requirements	Partially implemented	Fully implemented	Partially implemented	NiFi
Stream ingestion and processing	Partially implemented	Fully implemented	Partially implemented	Flume, NiFi, and Kafka

Additionally, Table 2 presents a comparison between commonplace performance indicators of the Apache Flume, Apache NiFi, and Apache Kafka Connect data ingestion tools, with recommendations for optimal tools based on their respective capabilities related to specific performance indicators [4].

Table 2. Comparison between the Apache Flume, Apache NiFi, and Apache Kafka Connect data ingestion tools' performance indicators along with a recommendation for optimal tools based on their respective capabilities related to specific indicators [4].

Performance Indicator	Flume (Low, Medium, High)	NiFi (Low, Medium, High)	Kafka (Low, Medium, High)	Optimal Tool
Speed	Medium	Medium	Medium	Flume, NiFi, and Kafka
Number of files processed per second	High	Medium	High	Flume and Kafka
Scalability	High	Medium	High	Kafka
Message durability	High	Low	High	Flume and Kafka

As can be seen in Table 2, in terms of speed, all three tools (Apache Flume, Apache NiFi, and Apache Kafka Connect) are rated equally with a "Medium" performance level, indicating no significant difference between them in this regard. When it comes to the number of files processed per second, Apache Flume and Kafka Connect both gain a "High" rating, outperforming Apache NiFi, which is rated "Medium". In terms of scalability, Kafka Connect stands out with a "High" rating, indicating its superior ability to handle larger loads and more complex data processing needs. Flume follows closely behind with a "High" rating, while NiFi follows with a "Medium" rating. Finally, in terms of message durability, Apache Flume and Kafka Connect continue to lead with "High" ratings, demonstrating their robustness in ensuring message persistence and reliability compared to NiFi's "Low" rating.

These results recommend Flume and Kafka as the best choices for scenarios prioritizing high throughput and message durability, with Kafka being particularly suited for scalability needs.

The other two open-source tools, Airbyte and Meltano, were compared in [5], where it was found, after an evaluation of data ingestion tools, that Airbyte is more suitable for the problem of data pipeline design for audit analytics. Furthermore, CDC is a process that identifies and captures only changes to data in a database, thereby increasing its efficiency and then delivering those changes in real time to a downstream process or system [6]. The following section provides more detail about CDC.

2.2. Change Data Capture

Change Data Capture (CDC) is a process that continuously identifies and captures incremental changes in data and data structures from a source such as a production database. The CDC pattern arose two decades ago to help replication software deliver real-time transactions to data warehouses. In these warehouses, data are transformed and then used in analytics applications. CDC enables efficient, low-latency data transfer to operational and analytics users with a low production impact [7].

An important advantage of CDC is that it can pass data quickly and efficiently to both operational and analytical users without too much disruption to the production environment. It captures changes as they occur and ensures that the data in the target system are up to date and consistent with the source system.

CDC is great for modern cloud architectures because it is a highly efficient way to move data across a wide-area network. It is a method of ETL (Extract, Transform, and Load) in which data are periodically extracted from a source system, transformed to meet the requirements of a target system, and loaded into the target system [1,6]. CDC is not only ideal for real-time data movement and an excellent fit to achieve low-latency, reliable, and scalable data replication but also for zero-downtime migrations to the cloud.

There are many use cases for CDC in the overall data integration strategy. They may be moving data into a data warehouse or data lake or creating an operational data store or a replica of source data in real time. CDC helps modernize data environments by enabling

faster and more accurate decisions, minimizing disruptions to production, and reducing cloud migration costs, which, all combined, make CDC the preferred method for data ingestion and movement [6].

CDC is very useful in today's data systems, especially for big data and real-time analytics. It helps companies quickly understand market trends and customer behavior and determine how well operations are running by continuously providing new data. This is very important for applications that require up-to-date data, such as fraud detection and improving customer service and monitoring operations.

2.3. Dataphos Publisher

When moving data from on-premises systems to the cloud, you are restricted to one of two outcomes: either investing in a very expensive CDC solution for fast, real-time processing or replicating your whole database several times a day, which does not give you the most up-to-date information while also placing a lot of stress on the database itself. Most of the time, these solutions will present the data consumers on your platform with raw data from several sources, and they will first have to process it before it can be used. This process is costly, time-consuming, and can be unsafe. Implementing Publisher with business logic at the source provides a more controlled real-time approach to managing data changes. It helps ensure data quality and consistency before the data are propagated to downstream systems, making it particularly valuable in scenarios in which data integrity and business rule enforcement are critical [8].

Figure 2 shows Dataphos Publisher's process, which begins with reading data from an on-premises database. The data are queried and formatted, creating a business object before transferring the data to the cloud. Once the business object is created, the data are serialized, encrypted for security, and sent to a message broker (a mediator between two services able to handle large volumes of data [9]), where they can be pulled by cloud services. The procedure described allows for a constant flow of ready-to-digest data packages to the cloud, and all at a fraction of the cost. This process (i.e., data ingestion) is integral to the whole system because it sets the foundation for downstream data processing, analysis, and reporting, which are essential for decision making in a timely and accurate manner.

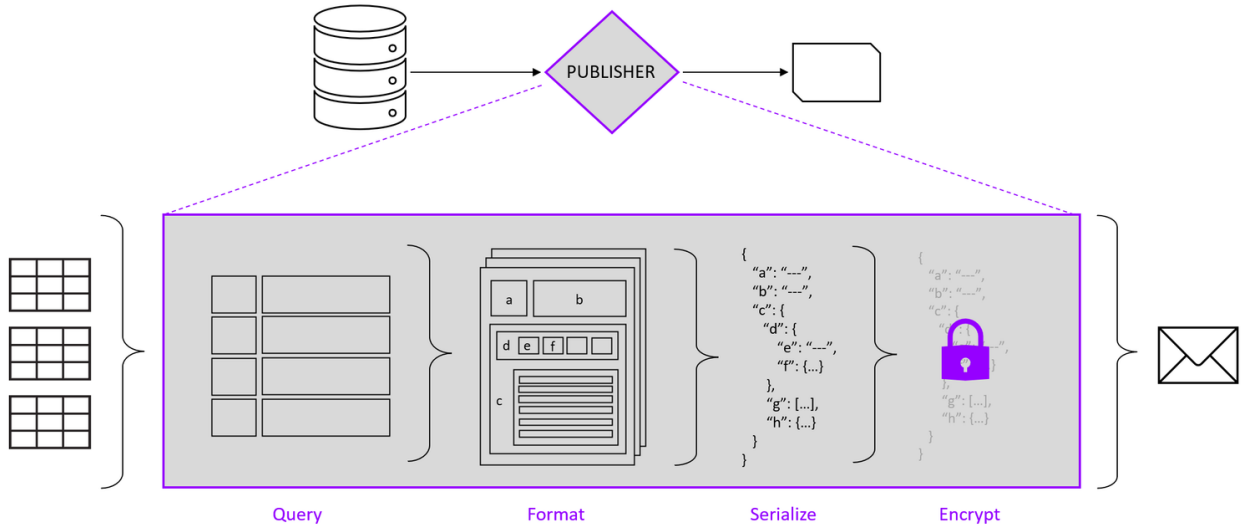

Figure 2. The Dataphos Publisher workflow begins by reading data from an on-premises database [10]. The data are queried, formatted, serialized, encrypted for security, and sent to a message broker where they can be pulled on-demand by appropriate cloud services [8].

2.4. CDC vs. Dataphos Publisher

Both CDC and Dataphos Publisher in the context of advanced web applications play the same role, which is data ingestion. However, they have different approaches when it comes to the task. Figure 3 shows the data flow in CDC. There are seven different actors

in the diagram. Looking at Dataphos Publisher's data flow in Figure 4, it contains only three actors, making the figure not only simpler but also faster. But the main difference between the two is how they carry out the ETL process. While CDC organizes the data right before entering the data warehouse, Publisher does so while the data are still on prem. That means that the data flowing through CDC pipeline are unstructured, while Publisher's data are presented as a business object ready to use. This also means that there is no need to save the data we derived from the business object in the database because they are never transported to the cloud (in case data are being transferred from an on-prem database).

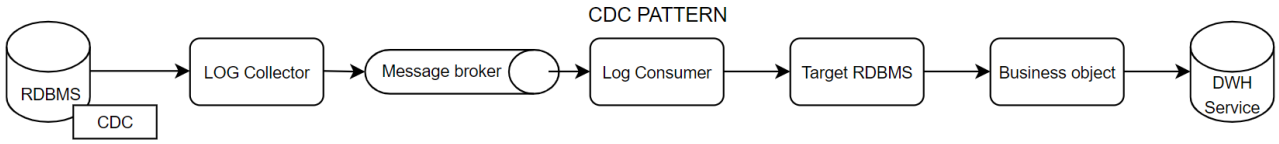

Figure 3. A flow diagram illustrating the Change Data Capture (CDC) data architecture pattern. The pattern begins with a Relational Database Management System (RDBMS) in which CDC is applied to capture changes. These changes are collected by a LOG Collector and passed to a Message Broker. The Message Broker then routes the set of messages to a Log Consumer, which processes the data and applies it to a target RDBMS. Finally, the processed data are used to update business objects which can be utilized by data warehouse (DWH) services for analytics and reporting.

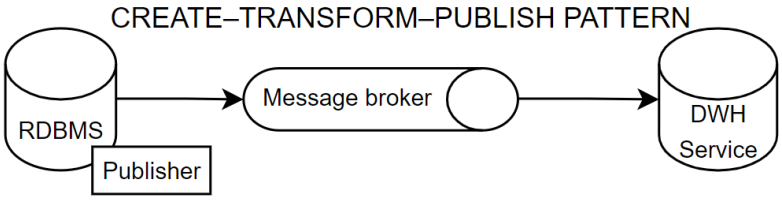

Figure 4. The Create–Transform–Publish (CTP) pattern is a data processing workflow that focuses on the streamlined process of data transformation from creation to utilization in data warehousing. The pattern starts with an RDBMS where the data are created. These data are then forwarded to a message broker, which is responsible for transforming the data as required. After processing, the data are published to a data warehouse (DWH) service for storage, analysis, and future retrieval.

To show that Dataphos Publisher is faster than CDC, as stated earlier, we performed an experiment in which we compared the performance of Dataphos Publisher and a CDC tool, Debezium [10], using standardized datasets of different sizes. This experiment compared performance between the Debezium tool and Dataphos Publisher in the context of real-time row detection and message propagation. We focused on measuring the time elapsed from the detection of added rows in the source system to the point at which the corresponding messages arrived at the message broker. The analysis was conducted with four datasets consisting of 1000, 10,000, 50,000, and 150,000 rows, respectively. The experiment was conducted five times for each dataset. The time required to process all rows was recorded for each run and then averaged. The data in the source database were always generated at around 200 rows per second. In the experiment, we utilized the Microsoft Wide World Importers (WWI) database as the dataset (publicly available at [11]). The WWI database is a sample database developed for use with the SQL Server and Azure SQL Database. It represents a fictional wholesale novelty goods importer and distributor based in the San Francisco Bay Area and serves as a comprehensive platform for learning and demonstrating SQL functionalities in a real-world business context. In the experiment, we used ksqlDB as the processing tool to measure the average time difference from when the message arrived at Apache Kafka to when the row was created in the source. The results of the experiment are provided in Table 3.

In Table 3, the first row indicates the throughput in terms of the number of data rows per second. The subsequent rows detail the performance of two different tools: Debezium and Publisher. Specifically, the first row after the throughput row shows the time taken

by Debezium to complete the data transfer, while the second row shows the time taken by Publisher.

Table 3. The table represents the average time (in seconds) it took for the number of rows per second to reach a Kafka topic from the source. The first row represents the average time for Debezium, and the second row represents the average time achieved by Dataphos Publisher.

Tool	Number of Data Rows			
	1000	10,000	50,000	150,000
Debezium	4.36	3.46	3.50	3.44
Dataphos Publisher	0.73	1.35	1.29	1.45

The analysis revealed distinct performance characteristics between the CDC tool Debezium and Publisher. Specifically, Publisher exhibited at least 2.3 times better results (depending on the number of rows per second) compared to Debezium. These findings may have implications for scenarios in which rapid data propagation and minimal latency are critical factors.

3. Data Processing Tools

In the dynamic landscape of computing and data processing, the efficiency and scalability of tools play critical roles in driving insight and innovation. As scientific institutions, business organizations, and governments grapple with increasingly large and diverse datasets, the choice of data processing tools becomes a critical decision that directly impacts performance, flexibility, and the ability to gain actionable insights.

This chapter explains three sophisticated data processing tools, Apache Spark [12], Apache Flink [13], and Kafka Streams [14], as well as Apache Beam [15], a tool for creating data pipelines that leverages data processing tools. Each of these tools possesses distinct capabilities that serve specific components of the data processing pipeline. Understanding the functionalities of Apache Spark's fast and widespread processing abilities, Apache Flink's exceptional event stream processing, Kafka Streams' integration into the Apache Kafka ecosystem, and Apache Beam's unified model for both batch and stream processing is crucial for making well-informed decisions in the current data-driven environment. It is important to note that this chapter is just an overview of some data processing tools available, and any comparisons between them are based on other articles as the authors of this article did not measure the performances of the tools.

Although these four tools specialize in different areas, the problems they solve are similar. Spark, Flink, and Kafka Streams are all tools that process data; however, Spark is more oriented toward batch processing, while Flink and Kafka Streams are stream-oriented [16,17]. The difference between Flink and Kafka Streams is that Flink is a more diverse tool and applicable in almost any stream-processing scenario (including scenarios that use the Kafka ecosystem), whereas Kafka Streams is tightly coupled with the Kafka ecosystem [18]. This makes Kafka Streams perfect for projects that already use Kafka but hardly usable in other cases. Finally, Beam is a different tool compared to the other three. It is not a data processing engine but a unified model that allows users to build pipelines in many popular programming languages and execute those pipelines by specifying the execution engine [19]. Despite Apache Beam being a unified programming language and not a data processing engine, it is still reviewed with the rest of the tools.

3.1. Apache Spark

Apache Spark [12,20] is a multi-language, open-source distributed-data-processing framework that has improved the world of big data analytics by speeding up data processing and being very accessible, allowing developers to seamlessly incorporate it into their pipelines [21]. Its strengths are its speed, versatility, and ease of use. Each of these strengths helped Spark become an important component in the big data ecosystem. It offers a fast

in-memory data processing engine that can handle a wide variety of workloads from batch processing to real-time streaming, machine learning, and graph processing [16,20,22]. Spark is based on a resilient distributed dataset (RDD) abstraction model which is an immutable collection of records partitioned across several nodes [19,23,24].

Figure 5 shows the architecture of the Spark cluster. The middle of the diagram is occupied by the Cluster Manager, which is responsible for allocating resources across applications. Spark acquires executors on nodes in the cluster once they are connected which are processes that compute and store data for the application. The application code (JAR or Python files passed to SparkContext on the left) is then sent to the executors. Finally, SparkContext sends tasks to executors to run [21].

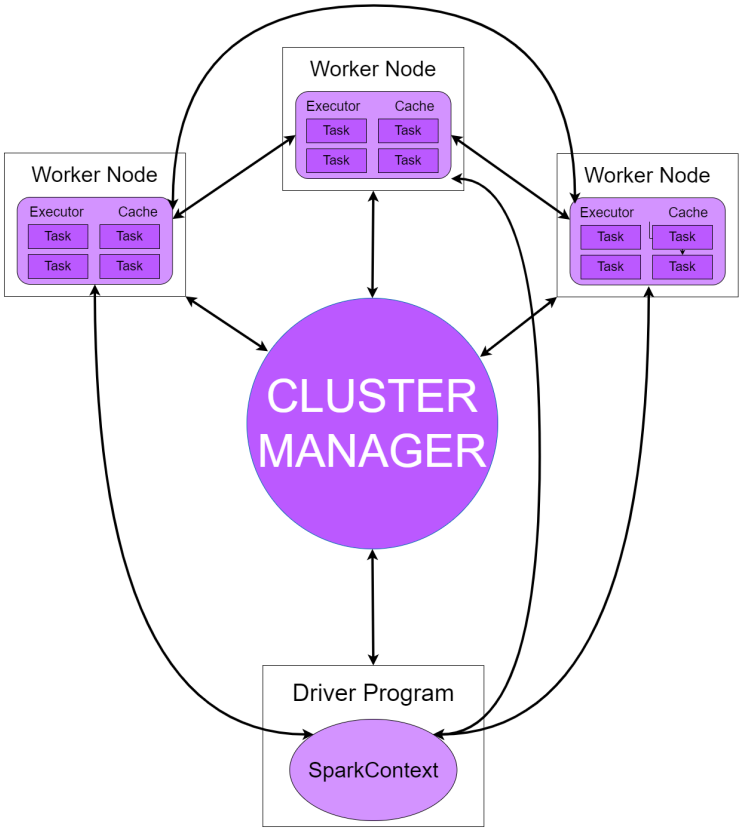

Figure 5. A diagram illustrating the Apache Spark architecture [20]. The Driver Program executes the main() function of the application and is the core of the Spark job. It creates a SparkContext to establish a connection to the Cluster Manager, which allocates resources across the cluster. Worker nodes are assigned tasks by the Cluster Manager. Each Worker Node has Executors that run the tasks and cache data. This architecture allows Spark to distribute data and carry out processing across multiple nodes, enabling efficient parallel processing. Spark provides three cluster modes, each named after one of the popular cluster managers, Mesos [25], YARN [26], and Spark's independent cluster manager (standalone), enabling users to run their Spark applications by allowing the driver process to connect to one of them [27].

3.1.1. Benefits of Apache Spark

Apache Spark is one of the most popular data processing tools on the market [16], and that is for a good reason. For example, Spark took Hadoop's MapReduce and took it to the next level with less expensive shuffles during data processing. What allows it to be so fast is the fact that instead of interacting with a disk (it can interact with a disk as well, but in that case, it is slower), it keeps all its data in memory, or at least it keeps as much of the data in the disk. Spark also supports the lazy evaluation of big data queries, meaning that the values are calculated only before they are used, which optimizes performance.

Given today's need to process a substantial amount of data in real time, Spark supports streaming data processing in addition to batch processing. It is important to emphasize that streaming data in Spark is not really streaming but processing mini-batches, which is an acceptable solution for many use cases. This allows Spark to be used in a wide variety of use cases ranging from streaming ones, such as finances, telecommunications, and IoT, to batch data ones like ML model training, data analysis, etc. [27–29].

Aside from being very fast and versatile, another big factor that influences Spark's popularity is its support for some of the most popular programming languages in the world, Python, Java, and Scala, by offering corresponding APIs [24,30,31]. Additionally, it offers SQL and DataFrame APIs. The accessibility of Spark does not stop there as there is a very rich ecosystem of libraries, packages, and frameworks surrounding Spark, extending its capabilities and covering even more use cases [27,29]. These libraries are written by a very large active and open-source community surrounding Spark, which consequently increases and improves documentation built around Spark, clarifying its processes.

Spark seamlessly integrates with other data processing tools, such as Hadoop, Hive, and HBase, causing effortless merges into existing data infrastructures. With the increasing need to quickly process large amounts of data (not only in big data but also in popular fields such as ML [30,32]), Spark turned out to be one of leading data processing tools available. A summary of Spark's benefits and advantages can be found in Table 4.

Table 4. A list of Apache Spark's key features with their corresponding descriptions.

Strengths	Explanation
Speed and Performance	In-memory data processing and lazy evaluation together put Spark on a higher level compared to traditional MapReduce programs.
Versatility	Spark supports a wide range of data processing workloads, including batch processing, real-time streaming, and machine learning.
Ease of Use	Provides high-level APIs in multiple programming languages, making it accessible to a broad range of data professionals.
In-Memory Processing	Spark stores data in-memory, reducing the need to read from and write to disk, which significantly improves processing speed.
Rich Ecosystem	Spark has a rich ecosystem of libraries and packages, such as Spark Streaming, MLlib, GraphX, and Spark SQL, which extend its capabilities and simplify complex data processing tasks.
Community and Support	Spark boasts a large and active open-source community. This means a wealth of resources, documentation, and support are available, making it easier to troubleshoot issues and learn how to use Spark effectively.
Integration	Spark can seamlessly integrate with various big data tools and platforms, including Hadoop, Hive, HBase, and cloud-based services, enabling users to leverage existing data infrastructure.

3.1.2. Disadvantages of Apache Spark

In the previous section, we discussed Spark's assets. However, there are some shortcomings that also must be taken into consideration when choosing the optimal data processing tool. One of the main benefits of Spark is its speed, which is the result of Spark operating in-memory. This high memory consumption causes higher operational costs, scalability limits, and challenges for data-intensive applications. Also, this means that the minimal requirements of a machine running Spark are much higher compared to disk-based applications like Hadoop MapReduce. If there is not enough memory to store a whole dataset, part of it is stored on disk, and that can degrade performance.

Another advantage mentioned in the strengths section was streaming data processing. As previously stated, Spark's "streaming" supports many use cases but not all, as some applications require near-real-time results. In this regard, Spark is inferior to some of the other solutions currently available on the market. The third issue arises from Spark's

advantage: its steep learning curve. As stated in the previous section, Spark's ecosystem is wide and varied, which is another double-edged sword. Spark is a "Swiss army knife" of data processing thanks to its wide range of libraries and frameworks, but using it effectively requires a thorough understanding of it, which may be difficult to achieve for some users. A summary of Spark's disadvantages can be found in Table 5.

Table 5. A list of Apache Spark's disadvantages or challenges with their corresponding descriptions.

Challenges	Explanation
Memory Intensive	While Spark's in-memory processing is one of its strengths, it can also be a weakness when handling very large datasets.
Complexity	Spark's ecosystem is extensive, which can be overwhelming for beginners. The learning curve can be steep, especially when exploring advanced components like Spark Streaming and MLlib.
Cost	The in-memory processing and distributed nature of Spark can result in high memory and processor costs, especially when used in cloud environments. Users need to be mindful of cost management.
Real-Time Streaming	Spark Streaming, while powerful, may not be as low-latency as some forms of specialized stream processing. It is suitable for micro-batch processing but may not be ideal for ultra-low-latency applications.
Resource Management	Spark does not manage resources as efficiently as some other data-processing frameworks. Users need to configure and monitor resource allocation to prevent inefficient resource utilization.

3.1.3. Spark Integration and Managed Services

Apache Spark provides APIs for users to deploy and run on the cloud, making it widely used in various cloud computing platforms and services to power big data and analytics solutions. Several cloud providers offer Spark as a managed service or integrate it into their ecosystems. Table 6 shows four cloud tools and platforms that use Apache Spark in the background:

Table 6. A list of cloud-based services that are used for analytics and data processing that use Apache Spark in the background.

Cloud Service	Explanation
Amazon EMR	Amazon's EMR service allows users to easily deploy Spark clusters for big data processing. It is integrated with other AWS services, making it an ideal choice for data analysis and processing in the Amazon Web Services (AWS) ecosystem [33,34].
Azure HDInsight	HDInsight, Microsoft's cloud-based big data service, incorporates Spark as one of the open-source frameworks offered. Users can launch Spark clusters in Azure for data processing, analytics, and machine learning [35].
Google Cloud Dataproc	Dataproc is a managed and highly scalable service that allows users to operate Apache Hadoop, Apache Spark, Apache Flink, and other applications. It enables the usage of open-source data tools for batch processing, querying, streaming, and machine learning [34,36].
Databricks	While not a cloud platform itself, Databricks is a unified analytics platform built by the creators of Spark. It is frequently hosted on cloud providers such as AWS, Azure, and Google Cloud, providing a managed environment for Spark-based data processing and machine learning [37].

These cloud tools and services provide a seamless environment for organizations to leverage the power of Apache Spark without the complexities of managing Spark clusters themselves. Users can scale their Spark workloads up or down, depending on their data processing needs, and take advantage of the cloud provider's resources and infrastructure.

3.2. Apache Flink

Apache Flink is an open-source, unified stream-processing and batch-processing framework developed by the Apache Software Foundation [38] for the standardized processing of data streams and batch processes [30] and a high-level robust and reliable framework for big data analytics on heterogeneous datasets [19]. It is a distributed streaming data-flow engine written in Java and Scala that executes arbitrary dataflow programs in parallel and in pipelines. Flink provides a high-throughput, low-latency streaming engine with support for event processing and state management. It is fault-tolerant in the event of machine failure and supports exactly-once semantics [13,21,39,40].

The core of Flink is the distributed dataflow engine which executes dataflow programs. Flink's runtime program is a DAG of stateful operators connected with data streams. The two core APIs of Flink are the DataSet API, used in batch processing, and the DataStream API, used in stream processing [32]. The core runtime engine (the green square in Figure 6) serves as an abstraction layer for the two APIs. On top of the core level, Flink combines both domain-specific libraries and APIs [13,41].

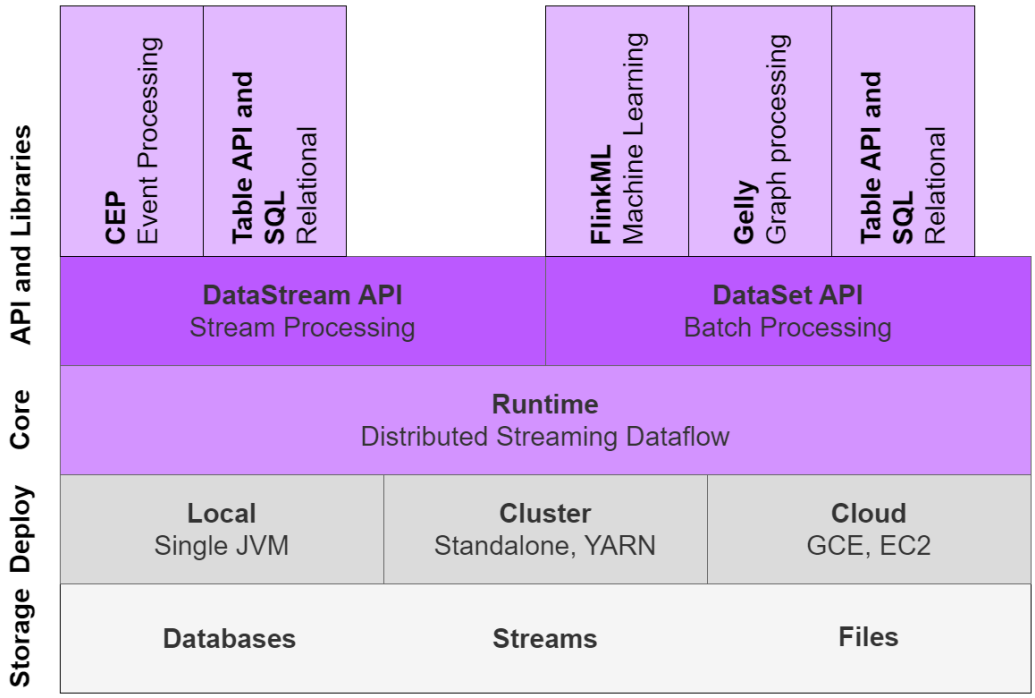

Figure 6. A diagram describing the Apache Flink software stack [38]. The top tier consists of several APIs: Flink ML for machine learning, Gelly for graph processing, and different APIs for batch and stream processing with table and complex event processing (CEP) capabilities [19]. The middle tier contains core APIs for dataset and data stream processing, as well as support for a distributed streaming dataflow. The deploy tier indicates that Apache Flink can be run locally, on clusters, or in the cloud, supporting various environments like YARN and EC2 [13].

Figure 7 shows Flink's Cluster architecture. The runtime consists of a TaskManager and a JobManager, while the Client is not part of the runtime but is used to send a dataflow to the JobManager and can either be detached or attached later.

The JobManager's responsibilities are deciding when to schedule the next task (or a set of them), reacting to finished tasks or execution failures, and coordinating checkpoints and recovery following failures, etc. There must always be a JobManager running, and by adding more, the system gains availability. When there are more JobManagers, there is always one that is declared the leader while the others are on standby. A JobManager consists of three components: a ResourceManager, a Dispatcher, and a JobMaster, which will not be discussed in this paper.

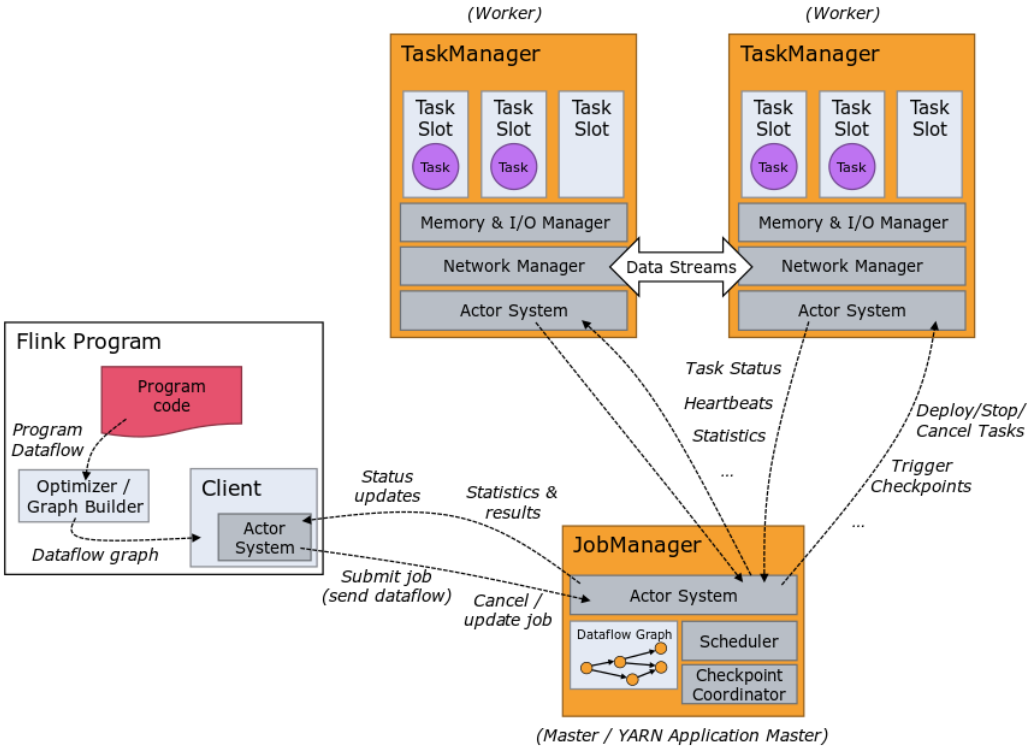

Figure 7. Apache Flink cluster architecture [38]. The three main components are the Flink Client, JobManager, and TaskManagers. The Flink Client submits the Flink Program and initiates the computation, including a graph builder and optimizer. The JobManager, which contains the data flow graph, the scheduler and the checkpoint coordinator, orchestrates the job's execution. TaskManagers are responsible for the execution of tasks in task slots and managing memory and network resources. This architecture enables distributed data processing and fault-tolerant streaming computation.

The TaskManagers (also called workers) are responsible for dataflow and buffering and exchanging data streams. There must always be at least one TaskManager. Each TaskManager consists of at least one Task Slot, and by adding more, concurrency is increased [13].

3.2.1. Benefits of Apache Flink

Flink is a powerful tool specified for stream processing. It has high throughput and very low data-processing latency (in nanoseconds). Aside from stream processing, Flink as a secondary mode also supports batch processing, making it versatile and applicable in a wide range of use cases. Flink provides a guarantee for data consistency through exactly-once processing semantics, meaning that the data will be received only once, ensuring it arrives and preventing duplicates. Event time processing allows Flink to be suitable for event-driven applications that process data based on event occurrences as well as event reception.

In its architecture, Flink allows for the duplication of components, increasing the application's durability and ability to tolerate machine failures. Also, a wide range of libraries and connectors expand Flink's usages, allowing it to solve machine learning problems (with FlinkML) and complex event problems (with FlinkCEP) and, with the help of many connectors, allows Flink to communicate with various data sources, databases, and messaging systems [32,42].

Finally, Flink has a large, growing, and active open-source community built around it, and it is used in many organizations across different industries, which simplifies the process of learning and flattens the learning curve. A summary of Flink's strengths and benefits can be found in Table 7.

Table 7. A summary list of Apache Flink's key features with their corresponding descriptions.

Strengths	Explanation
Real-Time Stream Processing	Flink can handle high-throughput streaming data, making it suitable for applications that require instant insights and rapid decision-making.
Exactly-Once Semantics	It provides strong guarantees for data consistency through exactly-once processing semantics.
Versatile Processing	It is capable of both batch processing and stream processing within the same framework.
Stateful Processing	Flink supports stateful processing, allowing it to maintain and manage states across time and data.
Event Time Processing	The ability of Flink to handle event time makes it well-suited for event-driven applications that process data based on when events occur, not when they are received.
Fault Tolerance	Provides built-in fault tolerance mechanisms, ensuring that processing jobs continue running even in the presence of node failures or other issues.
Rich Ecosystem	Flink has a rich ecosystem of libraries and connectors such as ones to various data sources, databases, and messaging systems.
Community and Industry Adoption	There is a growing and active open-source community built around Flink, which is used by many organizations across different industries

3.2.2. Disadvantages of Apache Flink

Just like with Spark, Flink also has some downsides, making it inferior to some tools in certain areas. A big issue Flink presents to new users is the complexity of its setup, especially in on-prem or on self-hosted environments. Hardships during deployment might cause developers to step away from Flink before deploying it successfully. Additionally, there is a steep learning curve for new users because of its extensive ecosystem, and mastering it takes a lot of time. Another issue that Flink has compared to some other frameworks is its immature community, which can lead to a smaller selection of third-party tools.

When it comes to processing, Flink blooms with stream processing but lacks other types. Even though support for batch processing exists, it is not nearly as efficient as stream processing, especially with large-scale jobs. Flink also manages resources suboptimally compared to other frameworks, and to optimize it, users need to manually tune the settings for resource allocation [42]. A summary of Flink's disadvantages and challenges can be found in Table 8.

Table 8. A summary list of Apache Flink's disadvantages with their corresponding descriptions.

Challenges	Explanation
Complexity	Setting up and configuring Flink clusters can be complex, especially in on-premises or self-hosted environments.
Learning Curve	Flink's large feature set may require a steeper learning curve for new users. The ecosystem is extensive, and mastering all aspects of Flink may take time.
Resource Management	While Flink offers strong fault tolerance, it may not manage resources as efficiently as some other frameworks. Users often need to fine-tune resource allocation.
Community and Ecosystem Maturity	Though Flink has a growing community, it may not be as mature as some other stream-processing frameworks, which can lead to a smaller selection of third-party integrations and tools.
Limited Support for Batch Use Cases	Since Flink supports batch processing, it may not be as optimized for large-scale batch jobs as some other batch-processing frameworks.

3.3. Kafka Streams

Kafka Streams is an important member of the Kafka ecosystem which allows developers to build applications and microservices using Kafka clusters as inputs and outputs. It is a stream-processing framework used for building streaming microservices in the Kafka ecosystem. Each instance of a stream-processing application is a Java application that uses a lightweight yet powerful Kafka Streams library, which allows it to communicate and coordinate the execution and partitioning of the processing using the Kafka broker [18,43,44]. Though it is not the central point of the ecosystem, it is a valuable addition to it. Before Kafka Streams, developers had two options. The first one was to use producer and consumer APIs and write your own code that would process the data. This solution would become unnecessarily complex once non-trivial operations were included in the stream. This was due to the producer and consumer APIs not having any abstractions to help with such use cases, so the developer would be left to their own devices as soon as they added some more advanced operations to the event stream. The second solution was to incorporate a streaming platform, which added unnecessary complexity to the solution and added unnecessary power [43]. The Kafka community recognized these problems and added Streams to the Kafka ecosystem which could be seamlessly incorporated into the solution and solve the same problems as the other streaming platforms but use less power [17].

Figure 8 shows the Kafka Streams architecture. Input comes from stream partitions which map to Kafka topic partitions, and data records in the stream map to a message. Each consumer creates a number of tasks that are assigned to a list of partitions to achieve greater parallelism of the application. Every task creates its processor topology according to the assigned partitions. In Kafka Streams, a task is the smallest parallel unit of work. Parallelism is bounded by the number of partitions because the highest possible parallelism occurs when the number of tasks is equal to number of partitions [14,17].

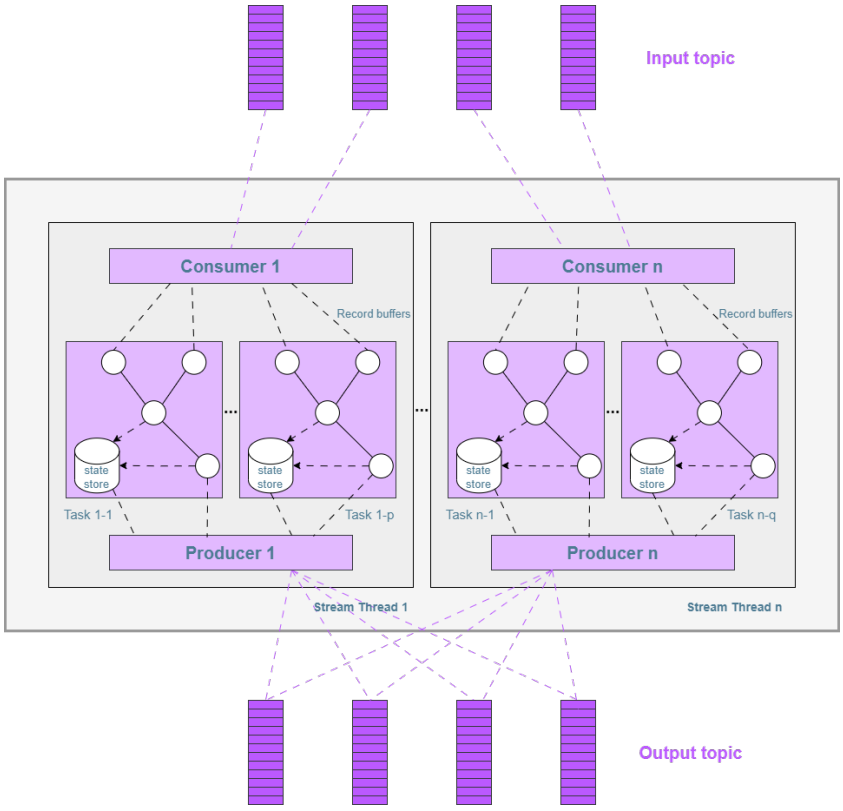

Figure 8. The Kafka Streams architecture is optimized for building real-time data processing applications. Source processors (input) ingest data from Kafka topics and sink processors (output) publish processed data back to Kafka. State stores maintain a local state, while internal processors are utilized for different tasks such as filtering, aggregating, and joining data streams [43].

3.3.1. Benefits of Kafka Streams

Kafka Streams stands out as a powerful tool for handling real-time data with many advantages, the first one being scalability. Since Kafka Streams is part of the Kafka ecosystem, there are a lot of similarities. Streams work with partitions, and by adding more partitions and consumer groups, Kafka Streams can handle a greater load, thus becoming more scalable. The reliability of Kafka is also transferred to Streams, where one can deploy a Kafka Streams cluster in which all instances are aware of each other. In case one instance fails, the others recognize the fault and automatically start another instance [17].

Deploying Kafka Streams is also relatively easy. While some other data processing tools require you to build a whole cluster that connects to an existing application, Streams uses the Kafka infrastructure, which requires no extra work if the application already has Kafka [17]. To add Streams to a system, the user simply adds it to the dependency of their Java project [43].

Kafka Streams processes stream data as opposed to some other data processing tools that use micro-batching. You can see the difference in Figure 9. This makes Streams better for cases in which it is extremely important to have data in real time. The tradeoff here is that with micro-batching, systems achieve close to real time and high throughput, while with streaming, latency is extremely low but throughput is negatively affected. However, thanks to the Kafka architecture, Streams can increase throughput simply by adding another partition, mitigating this issue [17,43,45]. A summary of Kafka Streams' benefits can be found in Table 9.

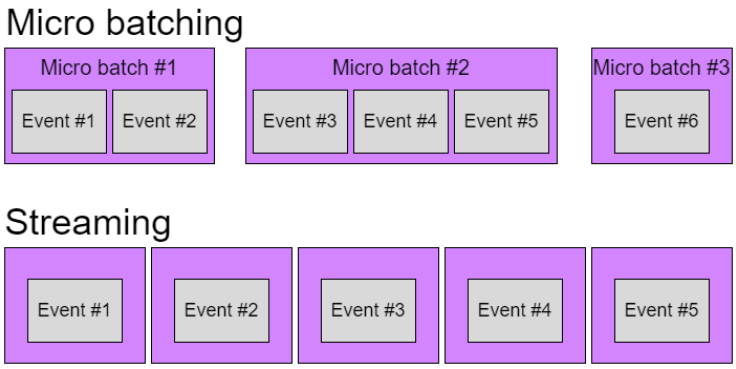

Figure 9. The difference between micro-batching and streaming. Two paradigms of data processing in Kafka Streams are micro-batching and event-at-a-time processing (streaming). The upper part of the diagram shows micro-batching, in which events are collected in small batches before processing. The lower part illustrates event-at-a-time processing, where each event is processed individually as it arrives [43].

Table 9. A summary list of Kafka Streams' key features with their corresponding descriptions.

Strengths	Explanation
Scalability	Streams can scale horizontally to handle increasing data volumes and processing requirements. You can add more processing nodes as needed to accommodate growing workloads.
Reliability	Adding more than one instance (and making a cluster) allows nodes to rebuild an instance in case of a failure.
Ease of Use	Since Kafka Streams leverages the familiar Kafka ecosystem, it is easier for organizations already using Kafka to incorporate stream processing without introducing new technologies.
Real-Time Data Processing	Kafka Streams is designed for real-time data processing. It can handle high-throughput and low-latency data streams, allowing instant insights and quick decision making.
No External Dependencies	Kafka Streams' functioning is independent of external libraries or additional components. This simplicity leads to easier deployment and maintenance.
Rich Ecosystem	There is a rich ecosystem of connectors, libraries, and tools built around Kafka as well as Streams, making it well-suited for integration into various data processing workflows and applications.

3.3.2. Disadvantages of Kafka Stream

Although Kafka Streams might prove to be a great streaming data processing tool, especially when it comes to systems using Kafka, there are some areas in which Streams does not work as well as its competitors. Mastering Kafka Streams is a long and difficult process. Even though it is easy to start with Kafka Streams, the journey becomes harder later.

When it comes to processing batches, Kafka Streams is not as good as Flink or even Spark because the other two (especially Spark) are more focused on batch streaming [45,46]. Also, resource management can be challenging, and users need to adjust the settings to achieve optimal results. A summary of Kafka Streams' disadvantages can be found in Table 10.

Table 10. A summary list of Kafka Streams' disadvantages with their corresponding descriptions.

Challenges	Explanation
Learning Curve	While Kafka Streams is relatively easy to start using for simple tasks, mastering it may take time and expertise.
Limited Batch Processing	While Kafka Streams can perform batch processing, it is primarily designed for stream processing.
Resource Management	Efficiently managing resources, such as CPU and memory, can be challenging when dealing with complex stream-processing topologies.
Limited Third-Party Integration	While Kafka Streams has a rich ecosystem, it may not offer as many third-party integrations as some other stream-processing frameworks.
Monitoring and Debugging	Advanced monitoring and debugging capabilities may not be as mature as those of other stream-processing frameworks.

The final issue is that there are not many third-party tools, reducing flexibility. This is due to the Streams' immaturity compared to other data processing tools. This also causes a lack of advanced monitoring and debugging capabilities which unnecessarily over-complicate troubleshooting complex topologies.

3.4. Apache Beam

Apache Beam is a unified model for both batch and parallel stream processing. It is not a data processing tool in the traditional sense. Instead, Apache Beam offers a programming model that allows data engineers to develop data processing pipelines in diverse programming languages [47] that can be executed on various data processing engines, such as Apache Flink, Apache Spark, and Google Cloud Dataflow [15,29,48]. Apache Beam does not process data by itself but rather defines the pipeline, along with the actual data processing engine. The purpose of Apache Beam is to simplify the usage of data processing engines by allowing the user to write pipelines in the programming language of their choice and to switch between these engines in an instant. The main goal of Apache Beam is to streamline the use of data processing engines. This is accomplished by allowing users to write pipelines in their preferred programming language with the flexibility to seamlessly switch between different processing engines as needed.

Apache Beam introduces a few abstractions, the first one being a pipeline. It is a complete description of the ETL job that will be executed, containing a description of the sources, operators, and sinks and the way they are connected. This pipeline is then executed by a Beam Runner. Another abstraction is a PCollection, which is a distributed, potentially unbounded, immutable, homogeneous dataset. Finally, PTransform is a data processing operation in the pipeline. Each PTransform takes data from one or more PCollections and produces zero or more PCollections as output [15,29,41]

The program describing the pipeline is called a Driver program. It first creates a pipeline objective and then creates the initial PCollection from some external source. Then it creates one or more PTransforms which perform operations on this initial PCollection,

creating altered PCollections which are potentially used again to derive new PCollections. Finally, using an IO operation, the final PCollection is written to an external source. This whole pipeline is executed using one of Beam's runners [29,41].

Figure 10 shows the model of Beam. The code for data pipelines can be written in Java, Python, Go, or any other language that supports Beam SDK. The pipeline is specified using the programming language which is passed to the Runner API that converts it into a generic standard language that can be used by the execution languages (Apache Flink, Cloud Dataflow, Apache Spark, etc.). The Fn API then provides language-specific SDK workers that act as a Remote Procedure Call (RPC) interface of User-Defined Functions (UDFs) embedded in the pipeline as a specification of the function. The selected runner finally executes the pipeline [15,22,49].

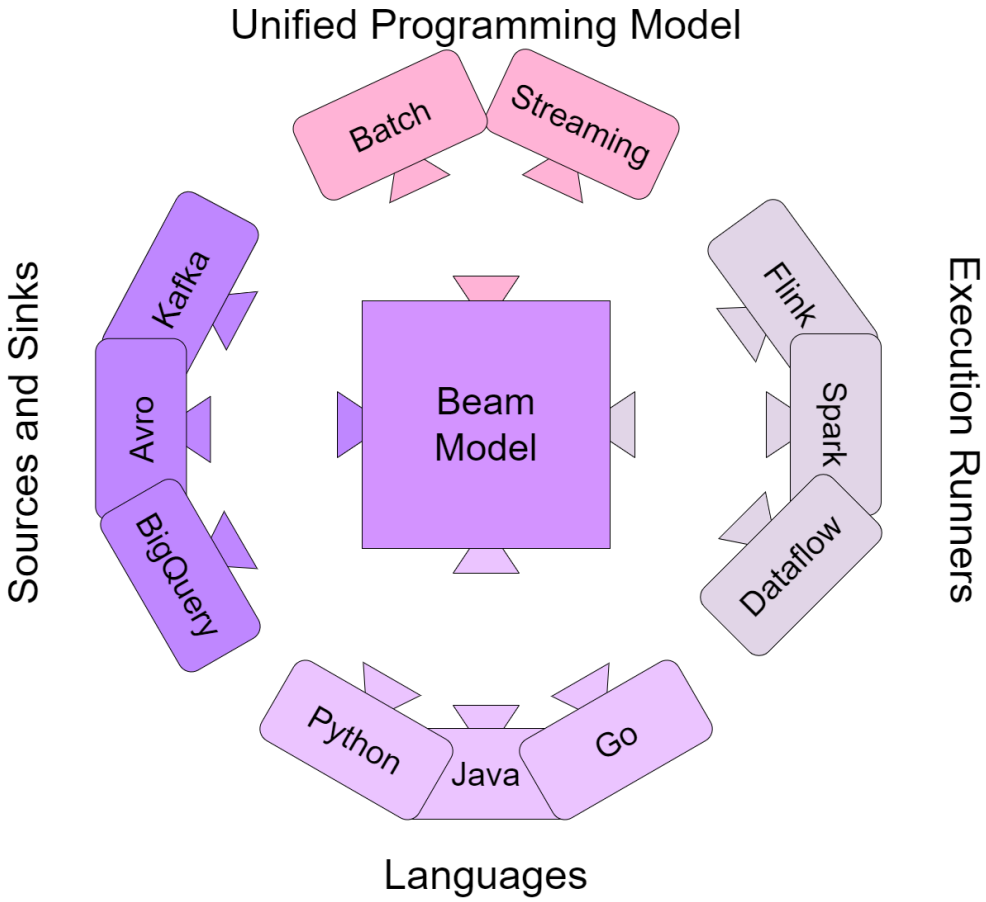

Figure 10. Apache Beam's model has a language-agnostic framework. Data pipelines are written in various SDKs like Java, Python, or Go and use the Runner API for pipeline construction. The pipelines can be executed on multiple processing engines (e.g., Apache Flink, Cloud Dataflow, Apache Spark) because of the Fn API, which facilitates communication between the SDKs and the runners [22,50]. Each category (incoming data model, execution runners, languages, and sources and sinks) is represented with separate color.

3.4.1. Benefits of Apache Beam

The main benefit of Apache Beam is in allowing the user to define data processing pipelines in different programming languages (e.g., Python, Java, Go, etc.) as well as their preferred data processing engine. Beam then translates the code into a language the data processing engine understands. This allows the user to quickly switch data processing engines, simplifying testing, which helps the user pick the correct data processing engine. This also means that no matter if the data come in a stream or in batches, Beam supports both because it is not the one performing the processing but rather the tool used to select

the processing engine. This is also true for resiliency; Beam inherits the fault tolerance of the processing engine it uses. If the user does not want to use a data processing engine, Beam comes with a built-in engine called Direct Runner, which should be used mainly for testing purposes.

Aside from picking any data processing engine, Beam can also be integrated with many different systems, such as data storage systems and messaging platforms. This further allows users to integrate Beam into almost any ecosystem. Furthermore, Apache Beam requires users to only consider four essential aspects: the nature of the calculated results, specifications for event time, specifications for processing time, and the relationships between result refinements. This streamlined approach enables users to concentrate on the logical composition of their data-processing jobs without studying the details of various runners and implementations. Apache Beam prioritizes ease of learning, emphasizing the logical aspects of data processing over the complexities of physical parallel processing orchestration [29]. Apache Beam is an open-source project with an active community of contributors, making Beam a well-maintained framework with great support [51]. A summary of Beam's strengths and benefits can be found in Table 11.

Table 11. A summary list of Apache Beam's key features with their corresponding descriptions.

Strengths	Explanation
Unified Model	Provides a unified programming model for both batch and stream processing, making it versatile and eliminating the need to learn and manage two separate frameworks.
Flexibility	Provides the flexibility to implement custom data processing logic and offers control over how data are processed and transformed.
Portability	Beam can run on various data processing engines, including Apache Spark, Apache Flink, and Google Cloud Dataflow, giving users flexibility and vendor lock-in avoidance.
Language Agnostic	It supports multiple programming languages, including Java, Python, and Go, making it accessible to a wide range of developers.
Data Ingestion	Supports a wide range of data sources and formats for data ingestion, enabling users to process diverse data types.
Ecosystem and Community	Has a growing ecosystem of connectors, libraries, and extensions, and an active open-source community, providing resources, support, and an expanding set of capabilities.

3.4.2. Disadvantages of Apache Beam

While Beam is a great tool, it still has some challenges to overcome. The first part is complexity; even though Beam simplifies some things mentioned in Section 3.4.1., it also complicates the structure when pipelines have both batch and stream processing requirements. Also, mastering Beam's model takes time and steepens the learning curve.

On one hand, Beam's flexibility is a great selling point, but on the other hand, some processing engine optimizations might not be fully accessible through Beam. This makes them suboptimal compared to the use of these engines alone. An additional factor that impedes Beam's application is suboptimal resource management because low-level resource management is abstracted away. Also, the additional layer of abstraction introduced by Beam creates at least some level of overhead, downgrading the system's performance.

Finally, Beam might not be compatible with the newest version of a processing engine, which prevents Beam users from using the specific version of a processing engine until a new version of Beam that supports that version of the engine is released [51]. A summary of Beam's challenges can be found in Table 12.

Table 12. A summary list of Apache Beam's challenges with their corresponding descriptions.

Challenges	Explanation
Complexity	Increased complexity when designing and managing pipelines that have both batch and stream processing requirements.
Learning Curve	Understanding and mastering the complexities of Beam's model can imply a steep learning curve.
Performance Overhead	The portability layer can introduce performance overhead, comparable to using a framework specific to the processing engine, as it must adapt to different underlying engines.
Limited Features	Some specific features and optimizations available in the underlying processing engines may not be exposed by default, which may require additional work for utilization in Apache Beam.
Resource Management	Efficiently managing and allocating resources, such as CPU and memory, can be challenging as Beam's model abstracts away some low-level resource management.

4. Discussion

This section discusses different scenarios and provides the preferred solution for developing an advanced web-based application for data processing tasks. When it comes to choosing the best data ingestion tool, Dataphos Publisher outperforms CDC, as demonstrated in Section 2.3, which is why we believe Publisher is a better overall solution. Also, Apache Beam can be used if the user requires flexibility, such as developing pipelines in their preferred programming language and easily transitioning between data processing engines.

When it comes to selecting the best data processing tool, matters become more complicated because none of the tools we reviewed are best suited to every use case [32,41]. In the following two subsections, different use cases will be discussed. The solutions to the problems in the use cases were chosen based on the research conducted in this paper and are, in general, optimal solutions; however, in some use cases, they may be suboptimal in comparison to other solutions.

Figure 11 presents a flowchart with general guidelines for selecting the appropriate data processing tool. The following sections provide real-world examples in which a solution to a data processing tool selection dilemma is proposed based on the algorithm shown in Figure 11.

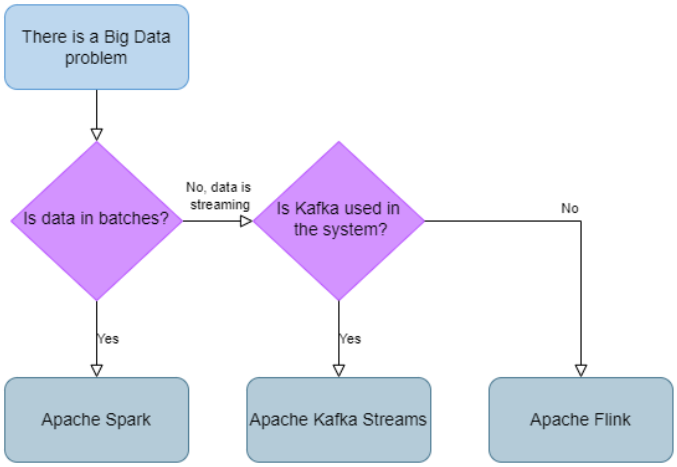

Figure 11. A flowchart for the selection of the optimal data processing tool. The selection process starts with a big data problem. If the data are being ingested in batches, then Apache Spark would be the most suitable solution out of the three data processing tools covered in this paper. However, if the data are streaming, then the data processing tool would be chosen based on which message exchange system is used; if it is Kafka ecosystem, then we would suggest the Apache Kafka Streams tool, and if there is no Kafka in the system, we recommend Apache Flink.

4.1. Use Case 1

Use case 1 involves a sophisticated fitness application designed to help users set and track their fitness goals. This app synchronizes with a smartwatch and collects important data such as heart rate, exercise duration, and the type of physical activity. As the user exercises, the smartwatch continuously transmits these data to the backend system via a message broker, distinct from Apache Kafka. The backend system processes this information and forwards it back to the user's smartwatch or smartphone. Using these data, the app performs a customized data analysis and provides a comprehensive set of statistics, including calories burned, remaining sets for goal achievement, and more. Crucially, this entire process occurs in real time, which requires fast data processing.

In this scenario, Apache Flink proves to be the most suitable choice among the data processing tools discussed in our paper. Flink excels in managing streaming data, which aligns perfectly with the application's need for immediate data processing. While Apache Spark is also a viable option, Flink offers a more direct solution as it processes continuous data streams as opposed to the Spark's approach, which simulates streams through micro-batching. Kafka Streams is not the ideal solution here due to the absence of Apache Kafka in the application's message-brokering system. However, if Kafka had been used as a message broker, Kafka Streams would have been the preferred choice for data processing.

4.2. Use Case 2

Use case 2 involves a retail analytics platform that analyzes sales data and dynamically updates product recommendations. This application continuously collects sales data throughout the day. At the end of the day, it processes this information to identify trending products and items that are frequently viewed by the current user. The processed data are then sent back to the app, which can recommend products that the user is likely to like and provide information about product availability in stores or delivery options.

For this scenario, Apache Spark proves to be the optimal choice due to its ability to process large amounts of data. Apache Flink, while also capable, is more tailored to processing streaming data and may not be as efficient in this context. Kafka Streams is not the preferred option here, similar to Flink and additionally due to the lack of integration of the Kafka ecosystem into the application framework. Therefore, Apache Spark, with its robust data processing capabilities, is the most suitable tool to improve the user experience on this retail analytics platform.

5. Conclusions

The objective of this paper is to explain how advanced web applications for data processing work, emphasizing the significance of data ingestion and processing in the contemporary digital environment. Firstly, we have detailed the architecture of Dataphos Publisher and conducted an experimental analysis to compare its performance with traditional CDC platforms. In the experiment, Dataphos Publisher demonstrated superiority over CDC. Secondly, this paper provides a comprehensive overview of various data processing tools. This includes a qualitative exploration of their functionalities, strengths, and limitations, providing the reader with a clear overview of the wide range of available options. Finally, the paper provides practical guidance for data engineers on how to build advanced web applications, including a framework for selecting the most appropriate tools for data ingestion and processing based on the application scenario. The motivation for this was to improve the understanding and use of these technologies and thus facilitate the development of sophisticated and efficient web applications.

Dataphos Publisher follows a "Create–Transform–Publish at source" pattern, utilizing a Relational Database Management System to transform data into structured business objects right there at the source. Data undergo formatting, serialization, compression, and encryption, resulting in a substantial reduction in network traffic. This approach also ensures robust security, making it suitable for use on both public Internet and private networks. In the context of advanced web applications, it serves as a data ingestion tool as

it handles huge amounts of data very quickly. When a user interacts with the app, data are collected, and the publisher uploads it to a message broker in the cloud.

Data processing is the second major big data process addressed in this paper after data ingestion. The primary objective of data processing is to handle vast quantities of data, either in real time or in large segments or batches. Each of the data processing tools described in this paper provides distinct advantages and disadvantages, and none of the tools are dominant in all scenarios. This paper provides a comprehensive qualitative examination of three tools: Apache Spark, Apache Flink, and Kafka Streams. Additionally, it explores Apache Beam, a tool specifically designed for constructing data pipelines.

Apache Spark shows its distinct advantages in in-memory processing and closes the gap between batch and real-time data analyses. Its extensive ecosystem of libraries and packages tailored to machine learning, graph processing, and more showed its multi-faceted capabilities. Apache Flink proved to be a great tool for real-time data processing characterized by low latency and exactly-once semantics. Its capabilities in processing high-throughput data streams, coupled with stateful processing, made it a formidable force, especially in industries that demand immediate insights and event-driven workflows. Kafka Streams, like Flink, has proven itself as a real-time data processing tool because it is a library that is tightly interwoven with the Kafka ecosystem. It offers seamless event-time processing and exactly-once semantics, making it a must-have player in industries in which data streaming and event-driven applications dominate. Apache Beam introduces a unified model, promising versatility, and portability. With support for multiple languages and compatibility with various processing engines, it has proven to be a great tool that provides a single customizable framework that can orchestrate both batch and streaming data.

The future of web development depends on advanced applications. A robust and scalable backend infrastructure is essential for this development. The use of Dataphos Publisher together with advanced data processing tools is an example of this approach and sets standards for the efficient and dynamic development of advanced web applications. Future work in this area is likely to focus on further improving the efficiency and scalability of data ingestion and processing tools and addressing new challenges related to data security, privacy, and compliance on increasingly interconnected digital platforms.

The integration of artificial intelligence (AI), machine learning (ML), and deep learning (DL) technologies into data processing tools is another area that should be intensively researched. This could lead to more intelligent, predictive, and adaptive web applications that not only respond to users' needs but also anticipate them. In addition, the development of more sophisticated data processing algorithms and the exploration of new data storage technologies will facilitate the processing of larger amounts of data with greater speed and accuracy.

In terms of practical applications, future research could look at developing more user-friendly interfaces for these complex tools to make advanced data processing accessible for a wider range of users, including those with non-technical backgrounds. In addition, there is a growing need for research in optimizing data processing for different types of networks from high-speed fiber optic networks to 5G telecommunication networks and slower, intermittent connections in remote areas.

Author Contributions: Conceptualization, Š.Š.; methodology, Š.Š.; validation, N.T. and M.H.; investigation, J.M. and N.T.; resources, Š.Š.; writing—original draft preparation, Š.Š., N.T., J.M. and M.H.; writing—review and editing, Š.Š., N.T., J.M. and M.H.; supervision, Š.Š. and N.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data is contained within the article.

Conflicts of Interest: Authors Šimun Šprem, Nikola Tomažin and Jelena Matečić are employed by the company Syntio. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Meehan, J.; Aslantas, C.; Zdonik, S.; Tatbul, N.; Du, J. Data Ingestion for the Connected World. In Proceedings of the CIDR, Chaminade, CA, USA, 8–11 January 2017; Volume 17, pp. 8–11.
2. Wu, X.; Zhu, X.; Wu, G.Q.; Ding, W. Data Mining with Big Data. *IEEE Trans. Knowl. Data Eng.* **2013**, *26*, 97–107.
3. Alwidian, J.; Rahman, S.A.; Gnaim, M.; Al-Taharwah, F. Big Data Ingestion and Preparation Tools. *Mod. Appl. Sci.* **2020**, *14*, 12–27. [CrossRef]
4. Mătăcuţă, A.; Popa, C. Big Data Analytics: Analysis of Features and Performance of Big Data Ingestion Tools. *Inform. Econ.* **2018**, *22*, 25–34.
5. Bylund, A. Data Pipeline Design for Audit Analytics: Data Ingestion Tools Evaluation & Proof of Concept. Master's Thesis, Umeå University, Faculty of Science and Technology, Department of Applied Physics and Electronics, Umeå, Sweden, 2023.
6. Tank, D.M.; Ganatra, A.; Kosta, Y.P.; Bhensdadia, C.K. Speeding ETL Processing in Data Warehouses Using High-Performance Joins for Changed Data Capture (CDC). In Proceedings of the 2010 International Conference on Advances in Recent Technologies in Communication and Computing, Kottayam, India, 16–17 October 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 365–368.
7. Petrie, K.; Potter, D.; Ankorion, I. *Streaming Change Data Capture*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2018; ISBN 9781492032519.
8. Dataphos Publisher: The Accelerator to Your Decision Making Process. Available online: https://www.syntio.net/en/labs-musings/publisher-the-accelerator-to-your-decision-making-process (accessed on 23 January 2024).
9. Hegde, R.G.; Nagaraja, G.S. Low latency message brokers. *Int. Res. J. Eng. Technol.* **2020**, *7*, 5.
10. Debezium Reference Documentation. Available online: https://debezium.io/documentation/ (accessed on 23 January 2024).
11. WideWorldImporters—Data Dictionary. Available online: https://dataedo.com/samples/html/WideWorldImporters/doc/WideWorldImporters_5/home.html (accessed on 23 January 2024).
12. Introduction to Apache Spark with Examples and Use Cases. Available online: https://www.toptal.com/spark/introduction-to-apache-spark (accessed on 23 January 2024).
13. Carbone, P.; Ewen, S.; Haridi, S.; Katsifodimos, A.; Markl, V.; Tzoumas, K. Apache Flink™: Stream and Batch Processing in a Single Engine. *Bull. Tech. Comm. Data Eng.* **2015**, *38*, 12.
14. Isah, H.; Abughofa, T.; Mahfuz, S.; Ajerla, D.; Zulkernine, F.; Khan, S. A Survey of Distributed Data Stream Processing Frameworks. *IEEE Access* **2019**, *7*, 154300–154316. [CrossRef]
15. Spæren, T. Performance Analysis and Improvements for Apache Beam. Master's Thesis, University of Oslo, Oslo, Norway, 2021; p. 108.
16. Ibtisum, S.; Bazgir, E.; Rahman, S.M.A.; Hossain, S.M.S. A Comparative Analysis of Big Data Processing Paradigms: MapReduce vs. Apache Spark. *World J. Adv. Res. Rev.* **2023**, *20*, 1089–1098. [CrossRef]
17. Wang, G.; Chen, L.; Dikshit, A.; Gustafson, J.; Chen, B.; Sax, M.J.; Roesler, J.; Blee-Goldman, S.; Cadonna, B.; Mehta, A.; et al. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In Proceedings of the SIGMOD '21: Proceedings of the 2021 International Conference on Management of Data, Xi'an, China, 20–25 June 2021; pp. 2602–2613.
18. Biernat, N.A. Scalability Benchmarking of Apache Flink. Bachelor's Thesis, Kiel University, Department of Computer Science, Software Engineering Group, Kiel, Germany, 29 September 2020.
19. Salem, F. Comparative Analysis of Big Data Stream Processing Systems. Master's Thesis, Aalto University, School of Science, Espoo, Finland, 2016; 89p.
20. Cluster Mode Overview. Available online: https://spark.apache.org/docs/latest/cluster-overview.html (accessed on 23 January 2024).
21. Nasr, K. Comparison of Popular Data Processing Systems. Master's Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2021; p. 76.
22. BEAM (Batch + strEAM) Your Data Pipelines on Google Dataflow. Available online: https://towardsdatascience.com/beam-batch-stream-your-data-pipelines-on-google-dataflow-2e3230bcdc21 (accessed on 23 January 2024).
23. Bonner, S.; Kureshi, I.; Brennan, J.; Theodoropoulos, G. Exploring the evolution of big data technologies. In *Software Architecture for Big Data and the Cloud*; Morgan Kaufmann: Burlington, MA, USA, 2017; pp. 253–283.
24. Ahmad, U.S.; Miyim, A.M.; Ali, M.S. Evaluation of Open-Source Tools for Big Data Processing. *Dutse J. Pure Appl. Sci.* **2022**, *8*, 10. [CrossRef]
25. Running on Mesos. Available online: https://spark.apache.org/docs/latest/running-on-mesos.html (accessed on 23 January 2024).
26. Apache Hadoop YARN. Available online: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html (accessed on 23 January 2024).
27. Tang, S.; He, B.; Yu, C.; Li, Y.; Li, K. A Survey on Spark Ecosystem: Big Data Processing Infrastructure, Machine Learning, and Applications. *IEEE Trans. Knowl. Data Eng.* **2022**, *34*, 71–91. [CrossRef]
28. Fernández-Gómez, A.M.; Gutiérrez-Avilés, D.; Troncoso, A.; Martínez-Álvarez, F. A New Apache Spark-based Framework for Big Data Streaming Forecasting in IoT Networks. *J. Supercomput.* **2023**, *79*, 11078–11100. [CrossRef] [PubMed]
29. Liu, J.; Zhu, T.; Zhang, Y.; Liu, Z. Parallel Particle Swarm Optimization Using Apache Beam. *Information* **2022**, *13*, 119. [CrossRef]
30. Tran, Q.; Nguyen, B.; Nguyen, L.; Nguyen, O. *Big Data Processing with Apache Spark*; Tra Vinh University Journal of Science: Tra Vinh, Vietnam, 2023; Volume 13.
31. Nazari, E.; Shahriari, M.H.; Tabesh, H. Big Data Analysis in Healthcare: Apache Hadoop, Apache Spark, and Apache Flink. *Front. Health Inform.* **2019**, *8*, 14. [CrossRef]

32. García-Gil, D.; Ramírez-Gallego, S.; García, S.; Herrera, F. A Comparison on Scalability for Batch Big Data Processing on Apache Spark and Apache Flink. *Big Data Anal.* **2017**, *2*, 1. [CrossRef]
33. Amazon EMR (Elastic MapReduce). Available online: https://www.techtarget.com/searchaws/definition/Amazon-Elastic-MapReduce-Amazon-EMR (accessed on 23 January 2024).
34. Gupta, U.; Sharma, R. A Study of Cloud-Based Solution for Data Analytics. In *Data Analytics for Internet of Things Infrastructure*; Sharma, R., Jeon, G., Zhang, Y., Eds.; Springer: Cham, Switzerland, 2023.
35. Azure HDInsight Documentation. Available online: https://learn.microsoft.com/en-us/azure/hdinsight/ (accessed on 23 January 2024).
36. Dataproc. Available online: https://cloud.google.com/dataproc?hl=en (accessed on 23 January 2024).
37. The Databricks Data Intelligence Platform. Available online: https://www.databricks.com/product/data-intelligence-platform (accessed on 23 January 2024).
38. General Architecture and Process Model. Available online: https://nightlies.apache.org/flink/flink-docs-release-1.1/internals/general_arch.html (accessed on 5 February 2024).
39. Hlupić, T.; Puniš, J. An Overview of Current Trends in Data Ingestion and Integration. Poslovna inteligencija d.o.o., Zagreb, Croatia. In Proceedings of the MIPRO 2021, Opatija, Croatia, 27 September–1 October 2021.
40. Espinosa, C.V.; Martin-Martin, E.; Riesco, A.; Rodríguez-Hortalá, J. FlinkCheck: Property-Based Testing for Apache Flink. *IEEE Access* **2019**, *7*, 150369–150382. [CrossRef]
41. Saxena, S.; Gupta, S. *Practical Real-time Data Processing and Analytics*; Packt Publishing: Birmingham, UK, 2017; pp. 89–106.
42. Rahman, T.; Jagannarayan, N.; Kannan, A. *Advances in Management, Social Sciences and Technology by Dr. Tazyn Rahman 2*; Empyreal Publishing House: Vasundhara Ghaziabad, India, 2021.
43. Apache Kafka Architecture. Available online: https://kafka.apache.org/35/documentation/streams/architecture (accessed on 23 January 2024).
44. Seymour, M. *Mastering Kafka Streams and KsqlDB*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2021; p. 410.
45. Tallberg, S. A Comparison of Data Ingestion Platforms in Real-Time Stream Processing Pipelines. Master's Thesis, Mälardalen University, School of Innovation Design and Engineering, Västerås, Sweden, 2020.
46. Van Dongen, G.; Van den Poel, D. Evaluation of Stream Processing Frameworks. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 1845–1858. [CrossRef]
47. Li, S.; Gerver, P.; MacMillan, J.; Debrunner, D.; Marshall, W.; Wu, K.-L. Challenges and Experiences in Building an Efficient Apache Beam Runner for IBM Streams. *Proc. VLDB Endow.* **2018**, *11*, 1742–1754. [CrossRef]
48. Žaja, M.; Čavrak, I.; Lipić, T. Benchmarking Apache Beam for IoT Applications. In Proceedings of the 44th International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, 27 September–1 October 2021; pp. 272–277.
49. Lukavsky, J. *Building Big Data Pipelines with Apache Beam: Use a Single Programming Model for Both Batch and Stream Data Processing*; Packt Publishing Ltd.: Birmingham, UK, 2022.
50. Apache Beam Overview. Available online: https://beam.apache.org/get-started/beam-overview/ (accessed on 5 February 2024).
51. Hesse, G. A Benchmark for Enterprise Stream Processing Architectures. Ph.D. Thesis, Universität Potsdam, Potsdam, Germany, 2022.