**Table of Contents**

[**CHAPTER 1: INTRODUCTION TO APACHE KAFKA 2**](#_gjdgxs)

[Theory 2](#_30j0zll)

[**AN OVERVIEW OF BIGDATA 2**](#_1fob9te)

[Quick Introduction to Hadoop 2](#_3znysh7)

[*Why Hadoop? 2*](#_2et92p0)

[Quick Introduction to Hadoop Distributed File System 3](#_tyjcwt)

[*Block Placement in HDFS 4*](#_3dy6vkm)

[*HDFS Architecture 5*](#_1t3h5sf)

[Quick Introduction to Spark 7](#_4d34og8)

[*What is Spark? 7*](#_2s8eyo1)

[*Why Spark? 7*](#_17dp8vu)

[*Components of Spark 9*](#_44sinio)

[*Spark Data Storage 12*](#_1y810tw)

[Introduction to Kafka 12](#_4i7ojhp)

[*What is Kafka? 12*](#_2xcytpi)

[*Why Kafka? 13*](#_2bn6wsx)

[Confluent Overview 16](#_3as4poj)

[Kafka Use Case 17](#_49x2ik5)

[Introduction to ZooKeeper 19](#_2p2csry)

[*What is ZooKeeper? 19*](#_147n2zr)

[*ZooKeeper Data Consistency 20*](#_vx1227)

[*ZooKeeper Architecture 21*](#_ihv636)

[*Why does kafka need ZooKeeper? 22*](#_32hioqz)

[**LAB EXERCISE 24**](#_1hmsyys)

[**SUMMARY 25**](#_41mghml)

[**REFERENCES 26**](#_3fwokq0)

**CHAPTER 1: INTRODUCTION TO APACHE KAFKA**

**Theory**

This chapter is intended to provide a comprehensive introduction to Apache Kafka, which is the center of focus throughout this book. We shall go through a brief overview of BigData before we begin with Kafka.

**An Overview of BigData**

**Quick Introduction to Hadoop**

Apache Hadoop is an open source distributed framework that allows storage and processing of large data (BigData) sets across cluster of commodity machines. Hadoop overcomes the traditional limitations of storing and computing of data by distributing the data over cluster of commodity machines, making it scalable and cost-effective.

The idea of Hadoop was originated when Google released a white paper about **Google File System (GFS)** - a computing model built by Google which was designed to provide efficient, reliable access to data using large clusters of commodity hardware. The model was then adopted by Doug Cutting and Mike Cafarella for their search engine called “Nutch”. Hadoop was then developed to support distribution for the Nutch search engine project by Doug Cutting and Mike Cafarella. Well, what does the name Hadoop mean? There is no significance for the name and it is not an acronym either. Hadoop is the name that Doug Cutting’s son gave to his yellow stuffed elephant. The name is very unique, easy to remember and sometimes funny. Not only does Hadoop have such name with no significance but also its sub-projects tend to have such names, which are based on names of animals, like Pig, and for the same reason. They are unique, not used anywhere else and are easy to remember.

**Why Hadoop?**

Companies today have been realizing that there is lot of information in unstructured documents spread across the network. A lot of data is available in the form of spreadsheets, text files, e-mails, logs, PDF’s and other data formats that contain valuable information which can help discover new trends, design new products, improve existing products, know customers better and what not. Data is increasing at an alarming rate beyond limits like never before and there are no signs of slowing down, at least in the near future. To deal with such data, we need a reliable and low cost tool to meaningfully process it. Therefore, we use Hadoop. Hadoop helps us process all this BigData which is present in variety of formats reliably, in a very lesser time and in a flexible and cost-effective way.

Let us see why Hadoop is so popular and what it has in store for you.

* **Scalable**: Hadoop is scalable, meaning: you can just start from a single node server and eventually increase more nodes as you need more storage and more computing power.
* **Fault-Tolerant**: Hadoop helps prevent loss of data. All the data which is stored in Hadoop Distributed File System is broken into blocks and stored with a default replication factor of 3. While processing data, if a node goes off, the process does not stop but continues, as the data still exists in other nodes.
* **Flexible**: Hadoop does not require schema. Hadoop can process unstructured, semi-structured and structured data from any kind of source or even from multiple sources.
* **Cost-effective**: Hadoop does not require expensive high end computing hardware. Hadoop works well with a cluster of commodity machines by parallel computing.

**Quick Introduction to Hadoop Distributed File System**

**Hadoop Distributed File System (HDFS)** is a File System which extends over a cluster of commodity machines rather than a single high end machine. HDFS is a distributed large scale storage component and is highly scalable. HDFS can accept node failures without losing data. HDFS is widely known for its reliability. Let us now check out why HDFS stands out of crowd when it comes to Distributed file systems.

|  |  |
| --- | --- |
| **Reliable Data Storage** | HDFS is very much reliable when it comes to data storage. Whatever the data stored in HDFS is replicated by a default replication factor of 3. That means, even if a machine fails, the data will be still available in two other machines. |
| **Cost Effective** | HDFS can be deployed on cluster of commodity hardware and can save you a lot of bucks. High end expensive hardware is not required by HDFS to function. |
| **Big Datasets** | HDFS is capable of storing Petabytes of data over a cluster of machines where a file can range from Gigabytes to Terabytes of size. HDFS is not designed to store huge number of small sized files as the file system meta data is stored in memory of NameNode. |
| **Streaming Data Access** | HDFS provides streaming access to data. HDFS is best suited for batch processing of data and not suitable for interactive processing. HDFS is not designed for applications which require low latency access to data such as OnLine Transaction Processing (OLTP). |
| **Simple Coherency Model** | HDFS is designed to *write once and read many times* access model for files. Appending the content to files is supported at the end but cannot be updated at arbitrary point, and it is also not possible to have multiple writers. Files can only be written by a single writer. |

**Block Placement in HDFS**

Hadoop is designed in such a way that the first block replica is placed on the same node as of client, and the second replica is placed on a different rack from first replica. Third replica is placed on a random node on the same rack as of the second replica. If the replication factor is more, random nodes in the cluster are selected to place the replicas. If a client running outside the cluster stores a file, random node (That isn’t busy) is picked to place the first replica. This way, if a node fails, the data is still available on other nodes of the cluster, and if a rack fails, again, the data is still intact.

**HDFS Architecture**

HDFS is a Master and Slave architecture, in which the Master node controls and assigns jobs to all its slave nodes. The following terminologies are used to describe the Master and Slave nodes.

The Master Nodes in HDFS are:

* NameNode
* Secondary NameNode

The Slave Nodes in HDFS are:

* Data Nodes

These nodes are the core serving roles in HDFS architecture. Let us now look in detail about the roles of each Node and understand them better.

|  |  |
| --- | --- |
| **NameNode** | NameNode is a HDFS daemon which controls all the Data Nodes and handles all the File System operations such as creating a directory, creating a file or reading and writing a file. The NameNode is responsible for managing the File System namespace image. It holds the image in memory, representing how the File System looks like. It also maintains the meta data for all the blocks of files in the File System and also tracks the replication value, so it knows the locations of blocks stored on Data Nodes within the cluster. But the meta data is not stored onto the disk and is every time recreated when it starts. NameNode stores all this information persistently on local disk in the form of namespace image and edit log. The NameNode is the *single point of failure* in the Hadoop cluster. If the NameNode fails, entire cluster fails. |
| **Data Nodes** | Data Nodes are the slave machines controlled by the NameNode, that actually does all the block operations. Data Nodes store and retrieve blocks when asked to do so by NameNode, and periodically inform NameNode with the lists of blocks they store by sending heartbeats. Data Nodes replicate the data physically when instructed by the NameNode on where and how to replicate. |
| **Secondary NameNode** | Secondary NameNode, as its name implies, is not exactly the Secondary NameNode. The secondary NameNode is not a high availability solution and does not automatically take over the responsibilities of NameNode on failure. Its main role is to create checkpoint and take the backup of NameNode periodically. It is like a backup solution to the NameNode. The hardware specifications of secondary NameNode should be similar to that of NameNode. In case of NameNode’s failure, the secondary NameNode can be manually configured to work as a primary NameNode. This is not a high availability solution. |

Now that we have had a quick introduction to Hadoop, let us shift our focus on the main topic of our discussion, Apache Spark.

**Quick Introduction to Spark**

**What is Spark?**

Apache Spark is an open source, fast and unified parallel large-scale data processing engine. It provides a framework for programming with distributed processing of large datasets at high speed. Spark supports the most popular programming languages such as Java, Python, Scala and R. Spark is scalable, meaning, it can run on a single desktop machine or a laptop to a cluster of thousands of machines. Spark provides a set of inbuilt libraries which can be accessed to perform data analysis over a large dataset. However, if your requirement doesn’t get satisfied with the inbuilt libraries, you can write one or explore countless external libraries from open source communities on the internet.

**Why Spark?**

Why use Spark when we have Hadoop? Well, Spark excels as a unified platform for processing huge data at very high speeds for various data processing requirements (will be discussed later in this chapter). Also, Spark is an in-memory processing framework. Spark is considered as a successor of Apache Hadoop. Let us briefly discuss the advantages of Spark over Hadoop.

With the Hadoop ecosystem, we had various frameworks for various data processing requirements. As a developer, you would use MapReduce framework to analyze your data using any of the languages such as Java, C++, Python etc, but a data warehouse engineer who is a SQL expert, has to learn any of the aforementioned programming languages. To overcome this problem, a new framework which runs on the top of Hadoop called “Hive” was introduced. This was also a problem for ETL processing, and so “Pig” was introduced. Similarly tools like “Giraph” and “Mahout” were introduced for Graphs processing and Machine Learning respectively. Moreover, Hadoop is only used for batch processing and there was no way to process data in real time. So, for this a new framework called “Storm” was integrated with Hadoop to work with streaming data. With so many frameworks, organizations had a tough time to maintain all the frameworks and track issues pertaining to them. Fortunately, all this would change with the advent of Spark. As mentioned, Spark is a unified platform which provides all these frameworks as one package with four major components.

Now, what actually does In-memory processing mean? Aren’t all the applications processed in memory only? Well, yes, all the applications are processed in-memory and written back to disk when processing is done, but Spark can process data in-memory and also retain the data within the memory or write to disk. Let us understand this with a figure by comparing Spark with MapReduce.

***1(a) Data Processing with MapReduce***







*Read Write* *Read* *Write* *Read Write Read Write*



In MapReduce, the data present in HDFS or any other *Distributed file system* is read by a MapReduce application and is processed in memory and then written back to disk after the job is complete. If the processed data is again needed for further processing, the data is again read from disk by a MapReduce application, processed in memory and then written back to disk. This process continues as per the requirement, as seen in the figure 1(a). The processes of reading and writing data from and to the disk increase the IO latency and so the overall job duration is increased. This is optimized in Spark as shown in the figure 1(b).

***1(b) Data Processing with Spark***







*Read*



*Write*

In Spark, the data is read from the disk, processed in-memory but, instead of spilling it back to disk, Spark can retain the data within the memory for further processing. So, if the processed data is again required for further processing, the data is already present in the memory and the Spark application processes the data eliminating the IO latency, and therefore the overall time to process the job is significantly reduced. With this, the processing speed when compared to MapReduce has been increased up to 100 times faster. The processed data from a Spark application can either be retained in memory or can be stored to the disk as per the requirement, as shown in the figure 1(b).

The reasons, such as a unified platform for various data processing requirements and High Speed In-Memory processing, have gained worldwide popularity throughout the industry with almost all the major organizations using Spark for their data processing requirements.

**Components of Spark**

Now that we know why Spark is being used, let us dive in more and learn what Spark is made up of. Let us look at each of the major Spark’s components individually and know them in detail. The following figure 1(c) shows the components of Spark.

***1(c) Components of Spark***









Let us look at a brief explanation regarding these components so that we can better understand the Spark components.

|  |  |
| --- | --- |
| **Spark Core** | Spark Core, as the name suggests, is the core component of Spark which has all the basic functionalities for processing large datasets. Some of its functionalities include managing memory, scheduling jobs, fault tolerance, using in-memory computation, referring datasets stored in storage systems etc. Spark Core includes a programming abstraction (API) called Resilient Distributed Datasets also known as RDDs, which is responsible for partitioning data across nodes on a cluster. With the help of these RDDs, the data can be transformed, collected and reduce things together. These RDD APIs can be referred by using any of the programming languages such as Scala, Python, Java and R as shown in the figure 1(c). |
| **Spark SQL** | The Spark SQL component provides the developer with an SQL like interface to work with huge structured data which is distributed over a cluster of nodes. Spark SQL works well with structured and semi structured data. Spark SQL can also work with data sources such as Apache Hive tables, Avro, JDBC, ORC, JSON and Parquet file formats. Spark SQL also allows developers to combine RDD APIs along with Spark SQL code in a single application. |
| **Spark Streaming** | Spark Streaming component of spark deals with processing of real time data also known as Streaming data. The streaming data can be from fleet of web servers, sensors, IOT devices or any other sources which generate data. This enables Spark to ingest data as it is generated in realtime and perform data manipulation on that data. There are three major phases of Spark Streaming. They are *Gathering*, *Processing* and *Data Storage*. Spark Streaming is also fault tolerant and scalable. We will talk a little about Spark Streaming in this book. |
| **Spark MLlib** | Spark MLlib is short for Machine Learning libraries which provides Machine Learning for huge datasets. MLlib contains various Machine Learning algorithms such as *Regression*, *Clustering*, *Classification* and *Collaborative Filtering*. MLlib also contains lower level primitives such as generic gradient descent optimization algorithm. MLlib also uses the linear algebra package called *Breeze* for numerical computing. |
| **GraphX** | GraphX deals with processing of Graphs in very efficient and distributed manner. GraphX extends the RDD APIs, which allows a developer to create a directed multigraph with properties attached to each vertex and edge. |
| **Cluster Managers** | Spark is all about processing massive amounts of datasets by distributing them over a number of nodes and scaling the cluster as required. In order to efficiently perform this task, a cluster manager is required, and Spark has its own cluster manager called *Standalone Scheduler*. Spark can also be deployed using *Hadoop YARN,* *Apache Mesos* or *Kubernetes* as a cluster manager to schedule jobs and manage the resources of the cluster. |

**Spark Data Storage**

Spark supports major file systems such as HDFS, Amazon S3, Azure Blob etc. Spark supports the local file system for storing the data as well. However, using a distributed file system such as HDFS can leverage the power of Spark by distributing the datasets throughout the cluster. Spark is also capable of dealing with various file formats such as text, ORC, parquet etc.

Hadoop and Spark are used to analyze huge amounts of data. This only solves one of the challenges faced with BigData. The other challenge is to actually collect huge amounts of data efficiently. Kafka helps us with this challenge. Let us now proceed with an introduction to Kafka and see how it deals with this challenge.

**Introduction to Kafka**

**What is Kafka?**

Kafka is an open-source, distributed, persistent and fault-tolerant message streaming platform or a central repository, which can handle high volume (trillions) of Publish-Subscribe messages everyday. Publish-Subscribe messaging system is a system where data is produced (Publish) by producers and consumers consume (subscribe) the data. We shall be looking at Producers and Consumers in detail, in the next chapter.

Kafka is written in Scala and is built on top of the ZooKeeper coordination service. The integration of Spark and Kafka enables real-time streaming data analysis. Kafka was built at LinkedIn and later donated to the Apache Software Foundation, making it open-source.

Kafka is popular because of the following features:

* **Scalable**: Kafka can be scaled from a single machine to a cluster of machines spanning data centers with zero downtime. The number of machines required can be scaled as per the requirement.
* **Persistent**: The data or messages are stored and cached in disk instead of memory, making them persistent and durable. Kafka is also fault-tolerant with replications and partitions.
* **Performance:** Kafka provides great performance and stability with huge volumes of publishing and subscribing messages.
* **Distributed**: Kafka is distributed to a cluster of machines and hence processes streams with great speed and efficiency.
* **Real-Time Streaming:** Kafka is capable of processing streams of messages in real-time as they occur.

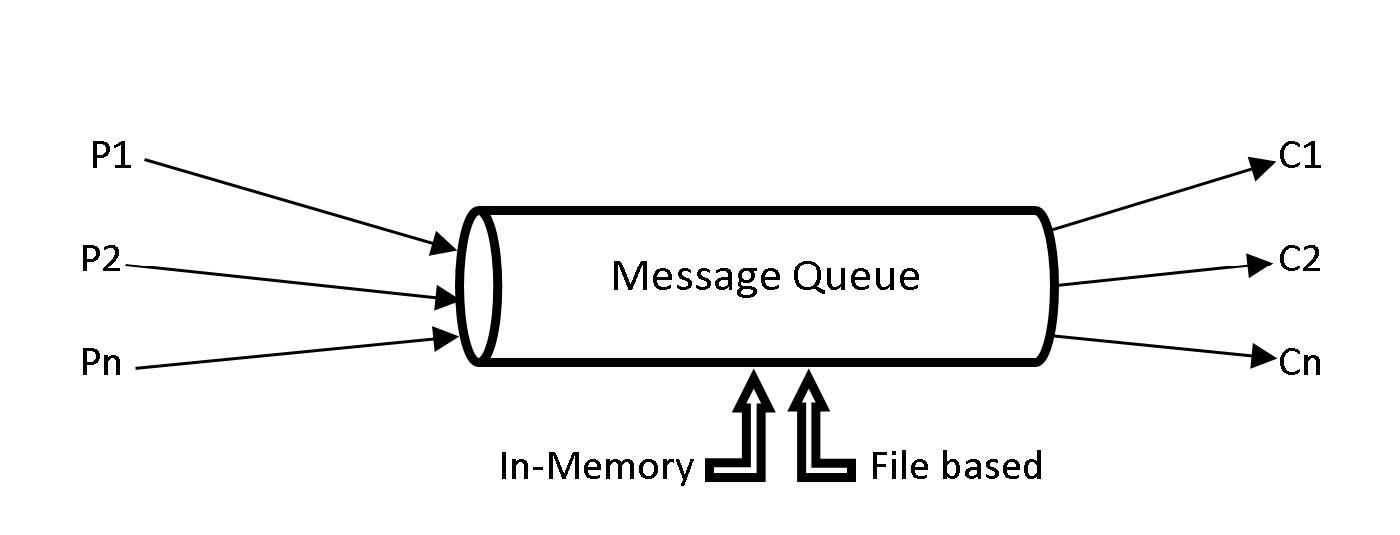
Kafka is used to build real-time streaming data pipelines for steady transfer of data between applications. The use cases include collecting logs from multiple servers, data from sensors etc.

**Why Kafka?**

Why do we need kafka? Don’t we already have message queue or data streaming platforms? How is Kafka better than the traditional data streaming platforms? Let us answer all these questions now. To understand why we need Kafka, we should first understand how the traditional streaming platforms work.

Consider a traditional system with a message queue as shown below. The message queue could be implemented in any of the programming languages. The message queue receives messages from various numbers of processes. These messages are denoted by P1, P2…Pn. This message queue may store data in-memory or on the file system based on the implementation. If the system is memory based, the data will be lost in case of system failure. But if the system is file based, the data will be intact even if the system goes down. The data from the message queue will be consumed by various numbers of consumer processes. These processes are denoted by C1, C2…Cn.

The data is produced by the producers and the consumers consume the data via the message queue. So far so good. However, the problem arises when one of the consumers is connected to a distributed platform such as Spark application. Spark is capable of processing huge amounts of data in a distributed manner. The message queue is not distributed and is implemented in a single machine. Therefore, the traditional message queue is limited by resources of that machine such as CPU core, RAM and disk size, and becomes the bottleneck as it cannot receive huge data similar to Spark.



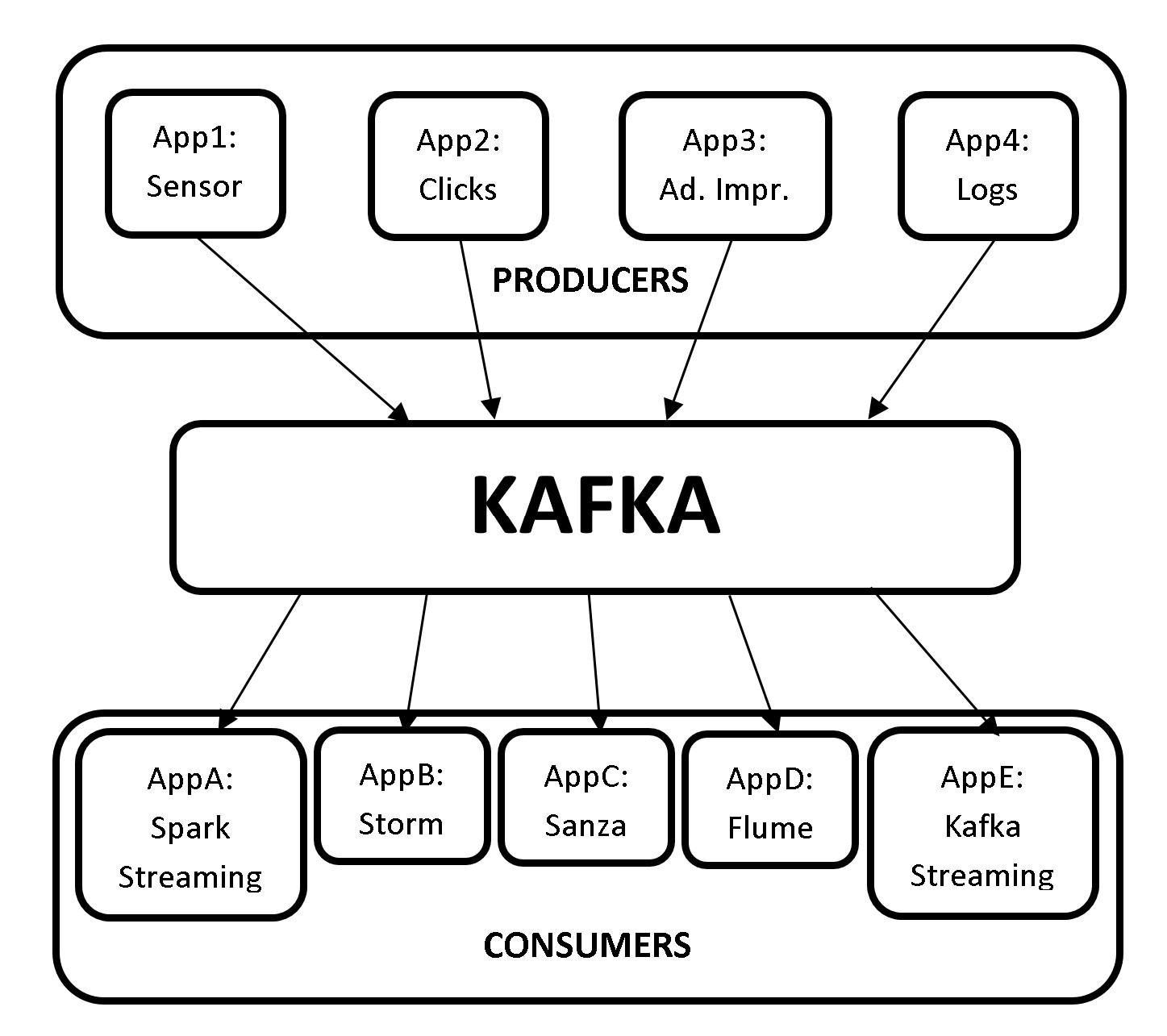
***1(d) Traditional Message Queue***

In order to cater a distributed processing application such as Spark, it is only efficient to have a message queue distributed across several machines. The limitation of machine resources such as CPU, RAM and disk was one of the reasons to develop Kafka, which is a distributed message queue.

Let us now look at another reason why Kafka had to be developed. Consider an application *app1* which generates data during its operations. This data is required by an analytics application, say *appA*. The data is simply transferred via an interface from *app1* to *appA*. Later there comes another application, say *app2* that generates data. This data is also required by the analytics application *appA* for more analytics. So, an interface has to be impemented to send the data from *app2* to *appA*. Once the interface for *app2* is implemented and tested, the interface for *app1* should also be tested in order to make sure the new implementation has not broken something with the old implementation.

Eventually more operational (*app3, app4, app5…*) and analytics (*appB, appC, appD…*) applications were developed which require data from operational to analytics applications. For example, data from *app4* might be needed by *appB* and *appD* or data from *app3* might be required by *appA* and *appC.* There might be many such possibilities for transferring data from one or more operational applications to one or more analytics applications. Everytime a new operational application generates data for analytics application, a new interface has to be implemented, and all the interfaces should be tested to check if something is broken due to the new implementations. This becomes very hard to maintain when there are so many applications and interfaces. The entire system should be tested every time a new interface between applications is implemented.

All these problems lead to the development of Kafka. Instead of having different interfaces for different applications, all the operational applications send the data to Kafka. The analytics applications can then consume the data from Kafka, making it a central repository. The figure below shows a pictorial representation of how data is being produced and consumed with Kafka as a central repository.



***1(e) Kafka Message Queue***

As seen from the picture above, we need not build interfaces and test them to transfer data every time a new application is implemented. All the producers generate the data to Kafka and the consumers pull the data from Kafka, making it a central repository.

**Confluent Overview**

Confluent is a data streaming platform based on Apache Kafka, and is founded by the creators of Apache Kafka. Confluent expands the capabilities of Apache Kafka not only to Publish-Subscribe messages, but also to a full-scale event streaming platform that enables to store and process real-time streams. Confluent Data Streaming Platform consists of Apache Kafka as core component.

Confluent data streaming platform provides the following components, making it a complete distribution of Apache Kafka.

* **Apache Kafka:** Apache Kafka is the core component of Confluent Platform. Apache Kafka is an open-source, distributed, persistent and fault-tolerant message streaming platform or a central repository, which can handle high volume (trillions) of Publish-Subscribe messages every day.

However, Apache Kafka is not a complete data streaming platform. It only provides data storage and interfaces for reading and writing data. It does not directly integrate with other services such as RDBMS. With Confluent’s other components, the capabilities of Kafka can be extended such that it can integrate with other services.

* **Kafka Connect:** Kafka Connect is used to transfer data to and from Kafka. HDFS, JDBC, S3, Elasticsearch, etc are some of Kafka connectors that transfer data to and from Kafka.
* **Kafka REST Proxy:** The Kafka REST proxy provides a RESTful interface to a Kafka cluster. The Kafka REST proxy can be used to send and receive messages, view the state of the cluster and perform administrative actions.
* **Kafka Streams:** Kafka Streams is a powerful yet easy to use client library for stream processing and analysis. With Kafka Streams processing layer, we can perform transformations or analysis by reading the real-time data and write the results back to Kafka.
* **Schema Registry:** Schema Registry is a serving layer for metadata. Schema Registry provides a RESTful interface for storing and retrieving AVRO schemas. It makes sure that the data which is being sent and received is in a common format i.e., checking schema compatibility for Kafka. We shall look at this in detail in the upcoming chapters.
* **KSQL:** KSQL is a streaming SQL engine for Kafka to run queries on data stored in Kafka cluster. KSQL is used internally on Kafka streams for processing.

We shall be only be focusing at Apache Kafka throughout this book. However, let us look at a use case to better understand how all these components are integrated to form an end-to-end pipeline using Confluent Data Streaming platform.

**Kafka Use Case**

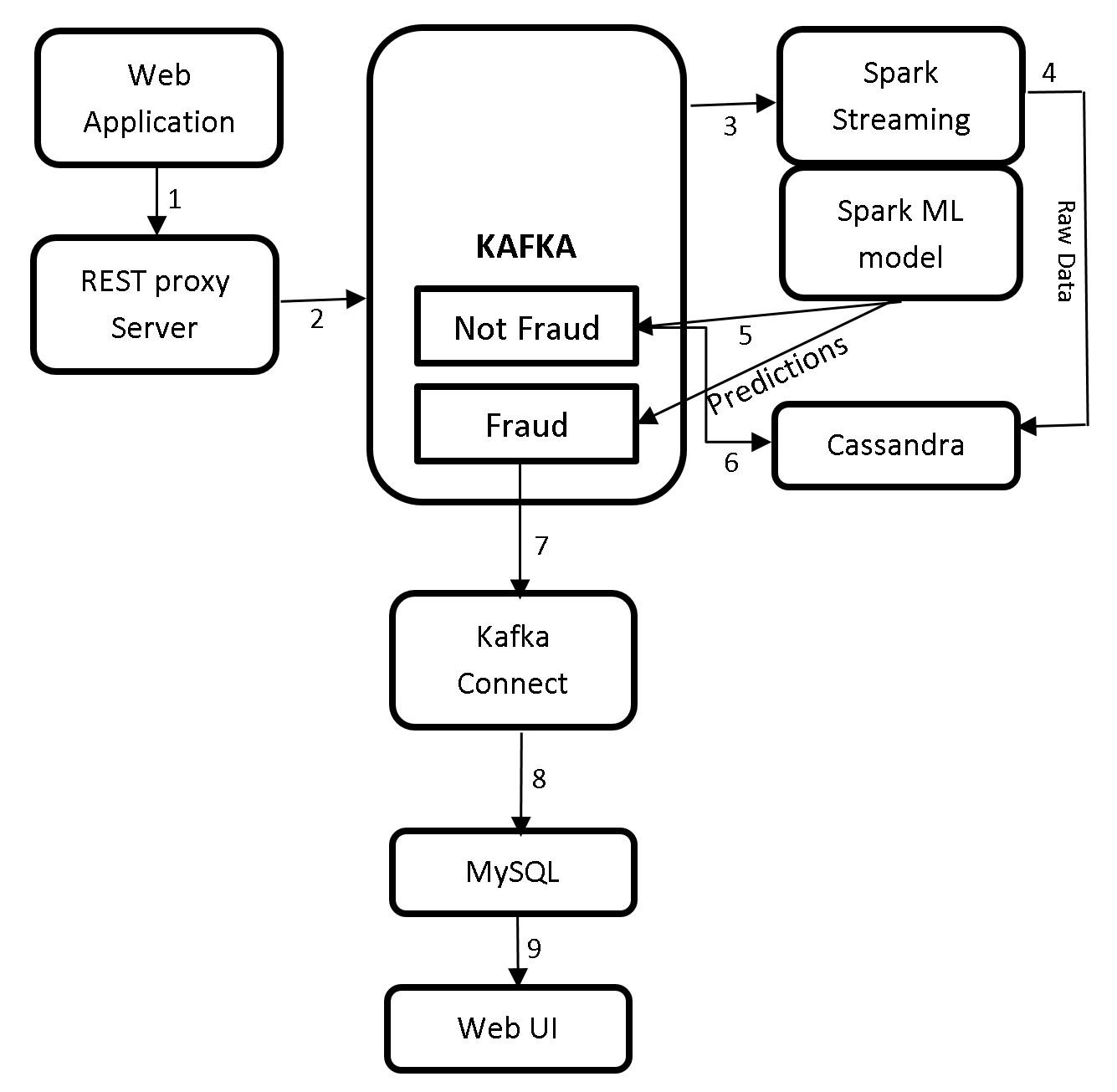
Let us now look at a Fraud Detection use case at high level, to better understand how all the components of Confluent can be integrated to form an end-to-end pipeline. We shall be looking at an use case of fraud detection in credit card transactions.

The usage of credit cards has been on the rise from the past few decades, and so are the frauds revolving around cards. People make transactions using credit cards round the clock. This transaction data can be analysed to prevent fraud in real-time and minimize monetary loss.

1. Let’s consider that all the credit card transactions are recorded in a web application.
2. These transactions are then transferred to Kafka message queue using the REST proxy server. The REST proxy server provides an interface to Kafka.
3. The transactions are then pulled from Kafka by Kafka Streaming or Spark Streaming to apply transformations or analysis. *Please note that Spark Streaming is generally used to process real-time messages similar to credit card transactions, since we feed the processed data to a Machine Learning model.*

After processing the streams, the data is internally fed to a Spark Machine Learning model to predict if the transaction is a fraud or not. Confluent Kafka does not provide Machine Learning module and hence Spark MLlib will be used for predictions.

1. The unpredicted raw data from Spark Streaming is also saved to Cassandra for further processing, if required.



***1(f) Kafka Use Case***

1. There will be two outcomes for this predictions i.e., genuine transactions and potential fraud transactions. This data is sent to Kafka.
2. Usually there will be more genuine or non-fraud transactions than the fraud transactions. Hence, all the genuine transactions are sent to a NoSQL database such as Cassandra.
3. The potential fraud transactions are then transferred to a RDBMS database such as MySQL using Kafka Connect.
4. These records can then be pulled from MySQL and displayed on webUI.
5. The domain experts can then take necessary actions to determine if these predicted potential fraud transactions are actually fraud. If it is determined that a transaction is fraud, the card issuer can block or hold the card from making transactions to prevent further loss.

This is how the Confluent Kafka components can be used to build an end-to-end pipeline.

Before we wrap up this chapter let us see what is ZooKeeper and why does Kafka need ZooKeeper.

**Introduction to ZooKeeper**

**What is ZooKeeper?**

ZooKeeper is an open source, robust distributed coordination service for distributed applications. ZooKeeper is an open source Apache Software Foundation project, it’s available for free and ready to use. ZooKeeper helps in overcoming many of the common challenges faced by distributed applications. ZooKeeper can be used for synchronization, sequential consistency and coordination between distributed applications. It helps in maintaining the configuration information which can be shared to all the nodes in a distributed system. ZooKeeper also helps in group services such as leader election and many more. ZooKeeper is reliable and fast yet very simple to work with. With ZooKeeper you can build reliable, distributed data structures for group membership, leader election, coordinated workflow, and configuration services. You can also build generalized distributed data structures like locks, queues, barriers, and latches.

ZooKeeper provides an eventually consistent view of its *znodes* which are nothing but files or directories in a file system. It provides basic CRUD operations such as creating, updating, and deleting *znodes*. It provides an event-driven model in which clients can watch for changes to specific znodes, for example if a new child is added to an existing *znode*. ZooKeeper is a high availability service as it consists of a set of ZooKeeper servers known as *Ensemble* (cluster), with each of the server holding an in-memory image of the distributed file system to serve client read requests. Each server also holds a persistent copy on disk.

One of the servers in the *Ensemble* is dynamically chosen by consensus as the leader, and all other servers are followers. The leader is responsible for all writes and for updating the changes to its followers. When the majority of followers update a change successfully, the write succeeds and the data is still available even if the leader fails. When a leader fails, a new leader is again dynamically chosen by consensus within the *Ensemble*. This eliminates the single point of failure scenario, and the *Ensemble* keeps on doing its work as it should.

When a client connects to ZooKeeper, it is provided with the list of servers in the *Ensemble*. The client connects to one of the servers in the ensemble at random until a connection is established. Once connected, ZooKeeper creates a session with a pre-specified timeout period by client. The ZooKeeper client automatically sends heartbeats periodically to keep the session alive if no operations are performed for a while, and automatically handles failover. If the connection between ZooKeeper and client fails, the client automatically detects this and retries to connect to a different server in the *Ensemble*. After it is reconnected, the same client session is retained while the failure has occurred.

**ZooKeeper Data Consistency**

ZooKeeper provides with the following guaranteed consistencies:

**Sequential consistency:** Updates from a client to the ZooKeeper service are applied in the order they are sent. Since all writes go through the leader, the global order is simply the order in which the leader receives write requests.

**Single System Image:** The Single System Image guarantees that a client will see the same view of the ZooKeeper service regardless of the server in the ensemble that it is connected to.

**Atomicity:** There are no partial failures. The updates from a client to ZooKeeper service either succeed or fail. For example, assume a client sends an update to a server, but before the response is received the network connection is lost or the server goes down. Now, did the update get through to the server? If yes, did the operation complete successfully? The only way to know the answers to all these questions is when the server/network is back up again. ZooKeeper though cannot help with network problems or partial failures, but it handles through atomicity. If the network/server goes down during an update operation, the operation is marked failed, else it is marked as success.

**Reliability:** If the update is successful, it is persistent and will not be rolled back. The update will only be overwritten when client makes a new update. The updates are still available even when the server fails.

**Consistent Client View:** A client’s view of the system is guaranteed to be up-to-date within a certain time bound, generally within tens of seconds. If a client does not observe system changes within that time bound, then the client assumes a service outage and will connect to a different server in the *Ensemble*.

**ZooKeeper Architecture**

The ZooKeeper architecture consists of leader and follower servers. The collection of these servers is known as *Ensemble.* The number of servers in a ZooKeeper Ensemble should always be an odd number. The reason behind this is because we need majority during the voting process of electing a leader. Let us now look at the responsibilities of leader and follower servers.

* **Leader**: When an *Ensemble* is first started, voting process to choose a leader takes place. During the voting process a leader is elected and the process is complete as soon as a simple majority of followers have synchronized their state with the leader. After leader election is complete, leader is responsible to handle all the write requests from clients, and changes are committed to all followers. Once a majority of followers have persisted the change, the leader commits the change and notifies the client of a successful update. There should always be a leader. If leader is down, all the existing followers go through the voting and elect a new leader.
* **Followers**: Followers function is similar to that of the leader by allowing clients to connect them and send, read, and write requests to them, but the writes are forwarded to leader.
* **Observers:** Observers are the non-voting members of an *Ensemble* which do not participate in the voting process, but only hear the voting results*.* When there are more followers who participate in voting, the write performance significantly drops, and hence Observers are added to *Ensemble* as to address this issue. Observers improve ZooKeeper’s scalability, and we can increase the number of Observers as much as we like without harming the performance of votes. Observers function is exactly the same as Followers, where clients connect to them and send, read, and write requests to them. Observers forward these requests to the Leader like Followers do, but they then simply wait to hear the result of the vote.

**Why does kafka need ZooKeeper?**

Kafka cannot be started without ZooKeeper. We must first start ZooKeeper service before we start Kafka. Based on the explanation of Zookeeper in the sections above, Kafka needs ZooKeeper for the following.

* **Electing a Controller:** Kafka consists of brokers to handle requests such as sending and receiving messages. The broker acts as a mediator for both producers and consumers to handle the requests. A broker is a Kafka server. Multiple brokers form a Kafka cluster. Since a Kafka cluster has multiple brokers, we need to elect a leader among these brokers to maintain the cluster state. The broker which we elect as a leader is called controller and is responsible to maintain the leader-follower relationships. In case of a failure of a broker, the controller’s responsibility is to tell all the replicas to act as partition leaders, in order to fulfill the duties of the partition leaders on the broker that is about to fail. ZooKeeper is used to elect this controller. ZooKeeper also makes sure there is only one leader, and also elects a new leader in case of failure.
* **Cluster Status:** ZooKeeper keeps tracks of all the brokers, and periodically checks if any of the brokers that are part of the cluster have failed.
* **Topic Configuration:** ZooKeeper makes track of existing topics, partitions for each topic, replica locations, preferred leader and configuration override information for each topic.
* **Access Control Lists:** ZooKeeper maintains Access Control Lists for each topic, i.e., the read and write permissions of clients.

Do not worry if you do not understand the new concepts at this time. We shall look at brokers, replications, partitions, topics etc in detail in the next chapter. It will be clear why we need ZooKeeper once we understand the architecture of Kafka in detail.

That’s all for the theory for this chapter.

**LAB EXERCISE**

"There are no activities required for this lab"

**SUMMARY**

Kafka is an open-source, distributed, persistent and fault-tolerant message streaming platform or a central repository, which can handle high volume (trillions) of Publish-Subscribe messages every day. Publish-Subscribe messaging system is a system where data is produced (Publish) by producers and consumers consume (subscribe) the data.

Kafka is written in Scala and is built on top of the ZooKeeper coordination service. The integration of Spark and Kafka enables real-time streaming data analysis. Kafka was built at LinkedIn and later donated to the Apache Software Foundation, making it open-source.

ZooKeeper is an open source, robust distributed coordination service for distributed applications. ZooKeeper is an open source Apache Software Foundation project, is available for free and ready to use. ZooKeeper helps in overcoming many of the common challenges faced by distributed applications. ZooKeeper can be used for synchronization, sequential consistency and coordination between distributed applications. It helps in maintaining the configuration information which can be shared to all the nodes in a distributed system.

**REFERENCES**

* http://kafka.apache.org/
* https://zookeeper.apache.org/
* http://spark.apache.org/
* http://hadoop.apache.org/
* https://www.confluent.io/