Lecture Notes

Apache Kafka

Introduction to Kafka

Batch and Real-Time Processing

In batch processing, data is collected over a period of time and then processed. The period over which the data is collected can be an hour, a day or a week depending on the use case. A common example of batch processing in our daily lives is the generation of electricity bills. Let's take another example. If someone working in the e-commerce domain needs to find out the sales of a particular product in a month, they will collect the data over that month and then process it to find out some meaningful information from it.

On the other hand, in real-time processing, data is processed as soon as it is available in order to derive meaningful information from it. In other words, the data is processed as and when it comes. A common example of real-time processing could be when you go to any bank ATM to withdraw cash, where the data that you enter is processed immediately and you get the cash. Another example could be the detection of fraudulent transactions. When you make an online request for a transaction, the data needs to be processed instantly to ensure that the transaction is not fraudulent.

Some of the key differences between batch and real-time processing are listed below.

1. Generally, in batch processing, data is collected over a period of time, which means that a huge amount of data has to be processed. However, in real-time processing, data is processed as and when it comes, and hence, the amount of data is less.
2. Latency is given the utmost importance in real-time processing. However, this is not the case with batch processing.
3. Generally, real-time processing is expensive as compared with batch processing.
4. In batch processing, if there is some downtime, then we can still live up with it, but in real-time processing, downtime means loss of data and information.

Messaging Systems

Traditional messaging systems used message queues to send messages from the source to the destination. Queues decoupled the source and the destination and helped in asynchronous communication. The source would send the message to the queue, where it gets stored until a consumer read that message. Once a destination or consumer read that message, the message would no longer be present in the queue. If some other destination server wanted to consume the same message, it could not do so, as the message would not be available in the queue. This was one of the biggest challenges faced when a traditional messaging system was used. This is where Kafka comes into the picture and tries to solve this challenge.

Kafka and Its Features

Apache Kafka is an open-source distributed event streaming platform that is used by many companies for high-performance data pipelines, streaming analytics and data integration. Kafka offers the following features to implement various use cases.

- You can publish and subscribe to streams of records.
- You can store streams of records in a fault-tolerant way for as long as you want.
- You can process streams of records as and when they occur.

Kafka offers the capability to publish and subscribe to streams of records. This means that you can publish messages to Kafka and the consumer can subscribe to messages of their choice. A typical real-life example of the pub-sub model could be Dish TV. The service provider offers several channels in different categories such as sports, news and entertainment. The common man can be thought of as a consumer who subscribes to these channels depending on their interests. By making use of the pub-sub model, Kafka tries to overcome the challenge of the traditional messaging system, where one message could be consumed by only one consumer. Kafka tries to bring out the best of the queueing and the pub-sub models.

Use Cases of Kafka

Nowadays, Kafka is used extensively across the industry, and many leading companies are using it. Some of the typical use cases of Kafka are as follows:

1. Log Aggregation: Nowadays, logs are generated everywhere. Businesses need to use logs in order to understand user behaviour. All logs are stored in queues, and then, different departments use these logs to understand more about the users.

2. Stream Processing: There is some data that needs to be analysed in real time. For example, data is coming from Twitter, and you need to count the tweets with a particular hashtag. You need to process this data instantaneously in order to get the required information. Kafka offers the capability to process streams of data.

3. Clickstream Tracking: Whenever you visit any website or a mobile application, you generate clickstreams. Organisations use clickstream tracking to replay the user journey and to extract meaningful information out of it.

4. Messaging System: This is one of the most common use cases of Kafka. Kafka is also used whenever there is a need for asynchronous communication between two applications.

Kafka should be used in the following scenarios:

1. When data is provided in streams

2. When you want to perform motion processing on data
3. When you want to build real-time pipelines and streaming applications

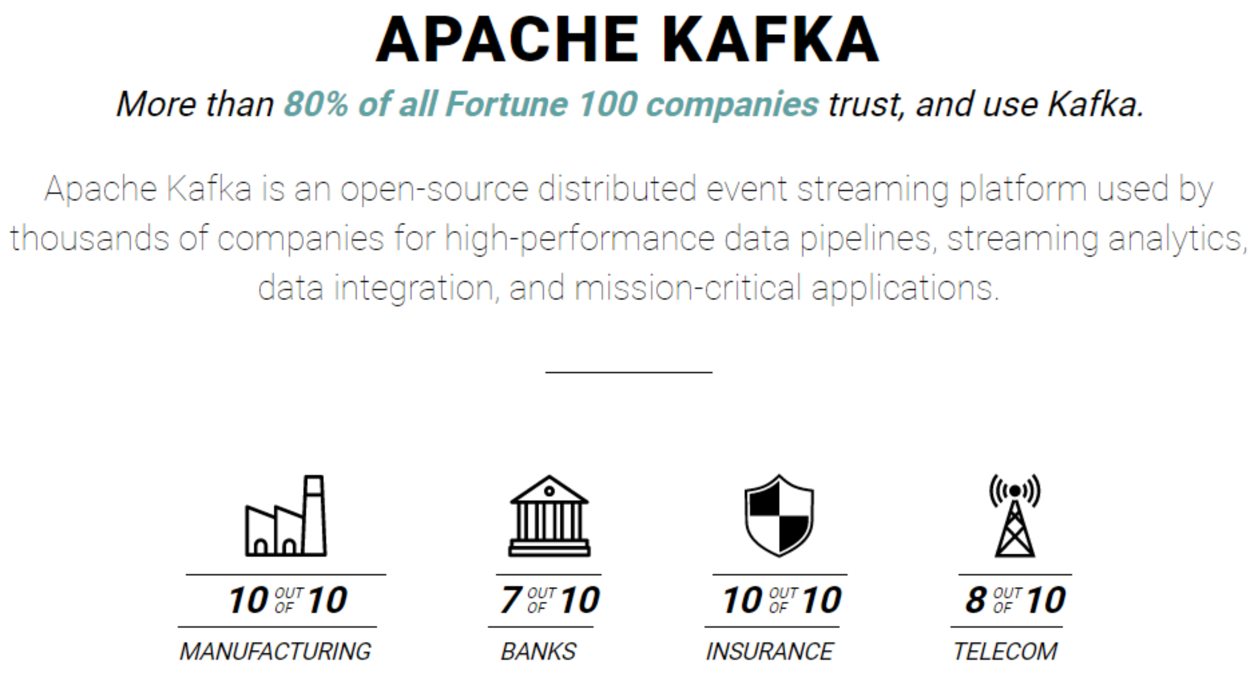
Kafka should not be used in the following situations:

1. When you need to perform batch processing on the data at rest
2. When you want to perform data migration
3. When there is persistent storage, as Kafka does not store the data; instead, it keeps the data in a queue

Some of the leading companies that are using Kafka are listed below.

1. LinkedIn
2. Twitter
3. New York Times
4. Pinterest

The screenshot attached below is of the official website of Kafka. It shows how Kafka is used across different industries.



APACHE KAFKA

More than 80% of all Fortune 100 companies trust, and use Kafka.

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

10 OUT OF 10 MANUFACTURING

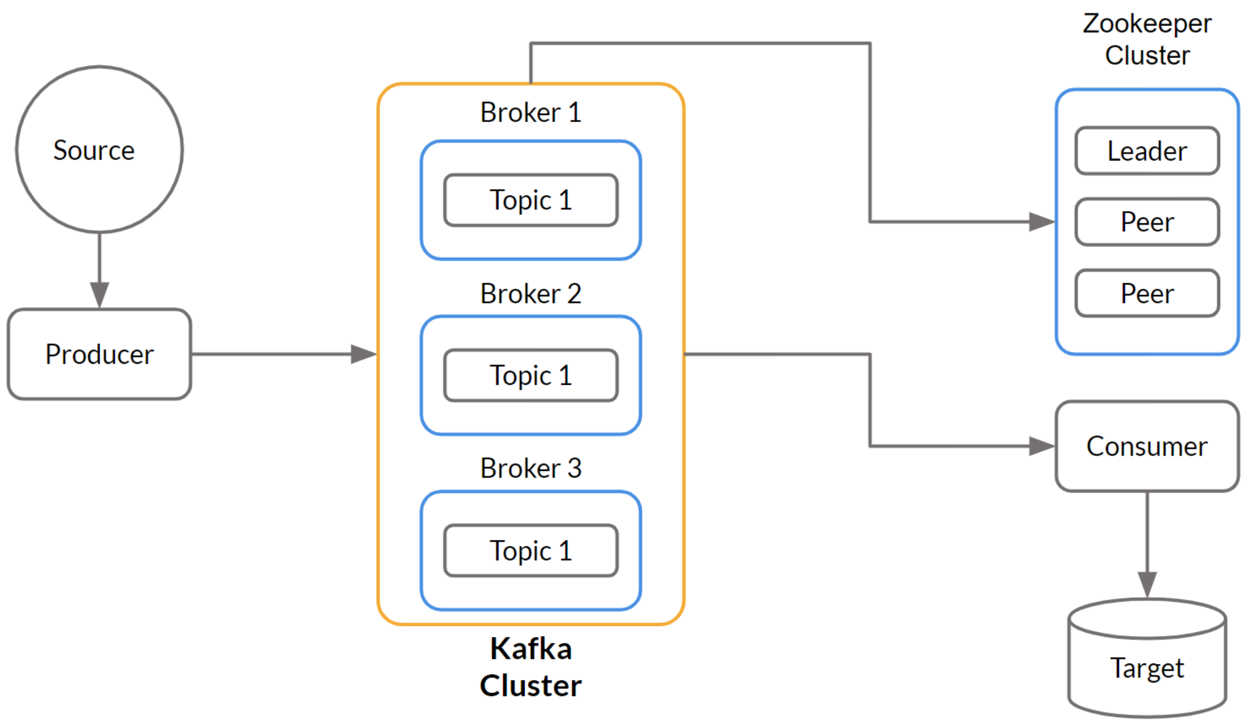
7 OUT OF 10 BANKS

10 OUT OF 10 INSURANCE

8 OUT OF 10 TELECOM

Kafka Architecture

The following diagram illustrates the different components of Kafka.

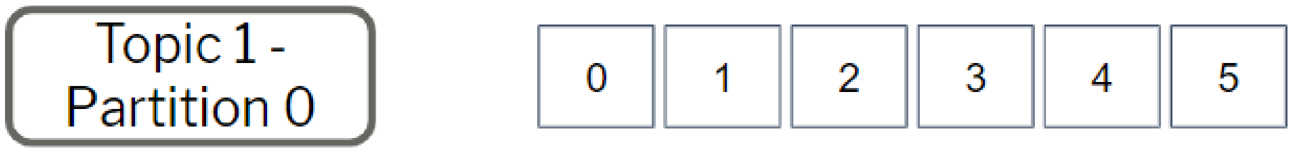


Kafka is a distributed system. A Kafka cluster is made of multiple brokers. Brokers are the machines on which Kafka runs. Kafka uses ZooKeeper to manage the state of the brokers. Brokers store messages/data in topics. Topics can be further divided into partitions. Topics are also replicated across different brokers to ensure fault tolerance, which is a necessity of any distributed systems. The failure of any broker should not lead to any loss of data, and this is ensured by replicating topics across brokers. Then, there is a **source** that produces messages. These messages are taken by a producer and written to the topics. A consumer can then subscribe to the topics of their choice and read messages present in them, and then, consumers can write this data to some external targets/sinks.

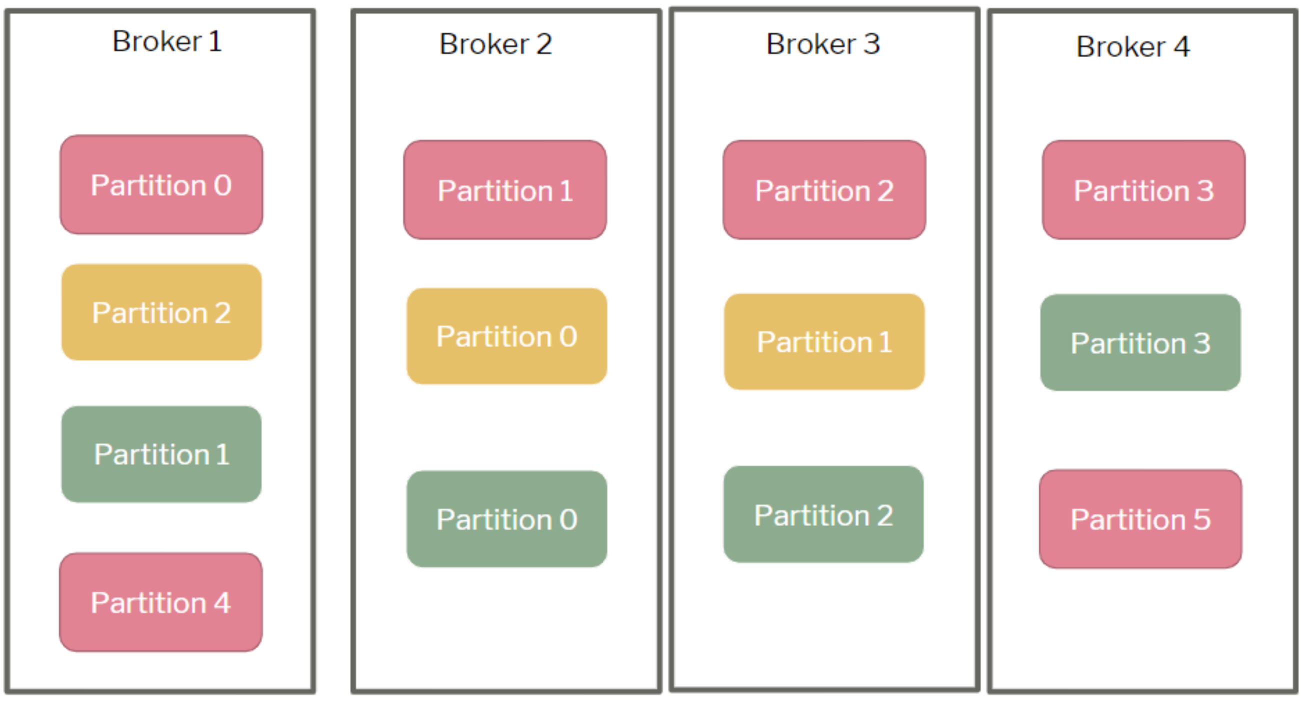
To conclude, a producer writes data to Kafka topics. These topics are stored in brokers, and consumers subscribe to the topics depending on their choice and read messages present in them.

Topics and Partitions

Topics are organised collections of data that are stored inside brokers. Each topic in Kafka collects data of a particular type and has a unique name. There can be multiple topics in a cluster, and the number of the topic depends on the storage capacity of the Kafka cluster. Each topic can be further divided into partitions. Partitions of a topic are stored across different brokers to balance the load across brokers and to provide fault tolerance. When messages are being written to partitions, they are assigned an ID called offset ID. It is an incremental ID. The message that is written earlier will have a smaller offset ID, while the message that is written later will have a larger offset ID.



Suppose six messages are being written in a partition of a topic. The first message will have an offset ID as 0, and the last message will have an offset ID as 5. The order in which these messages will be read from the partition is the same order in which they were written to the partition. The message with a lower offset ID will be read first as compared with the message with a higher offset ID. When there are multiple partitions of a topic, multiple consumers can read from these partitions in parallel, thereby increasing parallelism. The messages that are stored in the partition are immutable. The following image illustrates the distribution of partitions across brokers.



In the image given above, each colour represents a particular topic. It illustrates the fact that different partitions of a topic get stored in different brokers to properly balance the load among the brokers.

Producers and Consumers

Producers write messages to topics. Messages are the smallest components when it comes to Kafka. Each message contains a key and value. The key for a message can be null. When the key is null, round-robin is used to write messages to partitions. This is done to ensure that the load is distributed across all the partitions and any particular partition does not get overloaded. Attaching a key to messages will ensure that messages with the same key always go to the same partition of a topic.

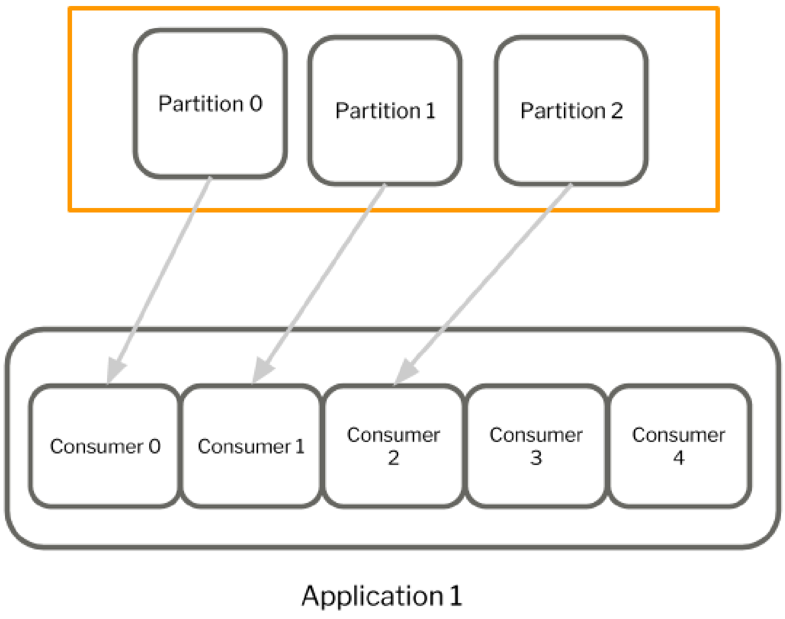
Consumers are responsible for reading data from topics. The data is read in a sequential manner, and the order in which it is read is the same order in which it was written. When a single consumer reads data from multiple partitions, the order is guaranteed at a partition level and not across partitions.

Consumer Groups

When a single consumer reads from multiple partitions, parallelism is not achieved. In other words, it reads from one partition and then reads from some other partitions, so parallelism is not ensured. To bring parallelism, the concept of consumer groups comes into the picture. A consumer group is a set of consumers that are grouped together and given a particular group ID. Each group ID represents one application that tries to read messages from the topics. Using consumer groups, Kafka brings the best of both the models, i.e., the queue and pub-sub models. If two consumers belong to different groups, then both of them can consume the same message. This feature is ensured using the pub-sub model. On the other hand, if two consumers belong to the same group, then they cannot read the same message.

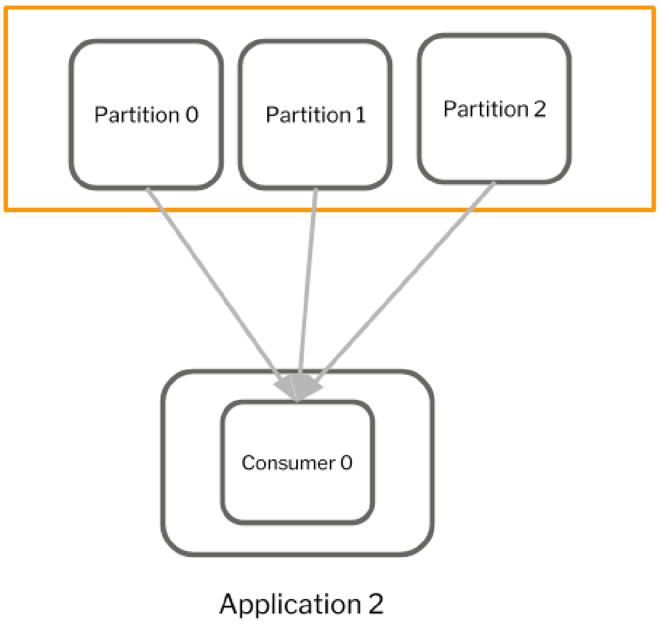
To read messages from Kafka, consumers are assigned partitions. The number of partitions that gets assigned to each consumer depends on the number of partitions and the number of consumers in a group. If the number of partitions is more than the number of consumers in a group, then multiple partitions get assigned to each consumer. On the other hand, when the number of partitions is less than the number of consumers, one partition is assigned to each consumer and some of the consumers are not assigned any partitions. Kafka guarantees that a partition is not assigned to more than one consumer belonging to the same consumer group. This scenario leads to wastage of resources. The ideal scenario that does not lead to any wastage of resources and achieves parallelism is the case when the number of partitions and consumers in a group are equal. In this situation, there is a one-to-one mapping of consumers and partitions. The following images illustrate the concept of the number of consumers and partitions.

MORE CONSUMERS THAN PARTITIONS



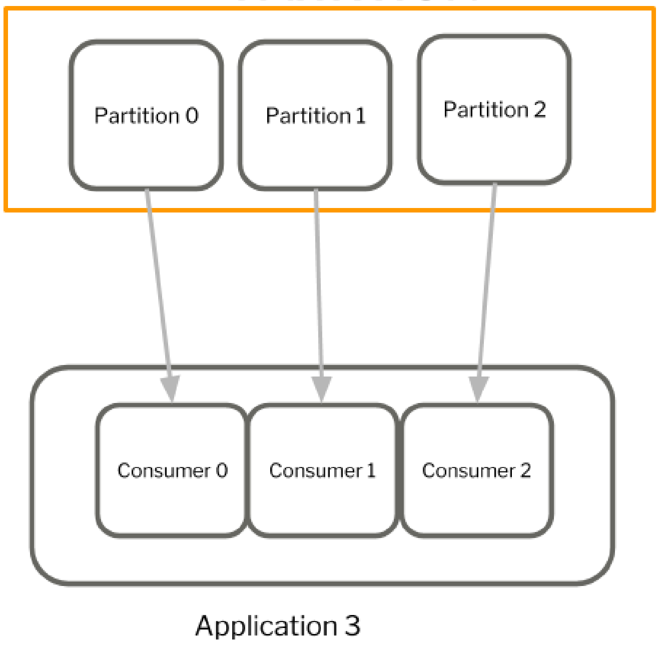
In the image given above, the number of consumers is greater than the number of partitions, leading to wastage of resources.

LESS CONSUMERS THAN PARTITIONS



In the image given above, the number of consumers is less than the number of partitions. Thus, parallelism is not achieved while reading messages.

EQUAL NUMBER OF CONSUMERS AND PARTITION



The same number of consumers and partitions ensure that proper parallelism is achieved and that there is no wastage of any resources.

When a new consumer joins a group or when a consumer leaves a group, the partitions of a topic get rebalanced across the available consumers, and the partitions are reassigned to different consumers when rebalancing happens.

Topic Replication

Topic replication is crucial in any distributed system. In a distributed system, any node can fail at any point in time. This would lead to loss of data and information. To avoid data loss, the topics are replicated across the different brokers present in the Kafka cluster. This helps to avoid data loss in case of any broker failure and also provides fault tolerance. To replicate topics, a replication factor needs to be specified while creating any topic. The minimum value that the replication factor takes is 1, and the maximum value possible for it to take is the number of brokers present in the Kafka cluster. One partition is elected as the leader partition, and this leader partition takes all the reads and writes instructions. The other partitions are followers, and they update themselves with the values present in the leader partition. Whenever a leader partition goes down, ZooKeeper elects a new leader. All the partitions are eligible to be a leader.

Running Producers and Consumers

All the following commands need to be run from the Kafka directory. Make sure that when running these commands, you are inside the Kafka directory.

The command to start the ZooKeeper server is as follows.

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

The command to start the Kafka server is as follows.

```
bin/kafka-server-start.sh config/server.properties
```

The command to create a topic named **test** is as follows.

```
bin/kafka-topics.sh --create --bootstrap-server localhost:9092
--replication-factor 1 --partitions 1 --topic test
```

The command to list all the topics is as follows.

```
bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
```

The command to start a consumer that will listen to messages from a topic named **test** is as follows.

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test
--from-beginning
```

The command to start a producer that will push messages to a topic named **test** is as follows.

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

The command to delete a topic named **test** is as follows.

```
bin/kafka-topics.sh --delete --bootstrap-server localhost:9092 --topic test
```

Kafka Connect

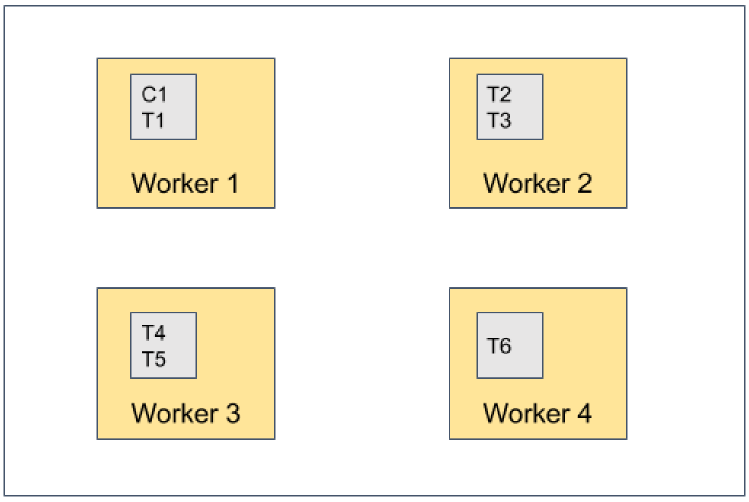
Kafka Connect is a framework to connect Kafka with external systems. These external systems could be any database or streaming application. The data can be moved either from an external system to a Kafka topic or from a Kafka topic to any external system. Connectors are responsible for moving this data. Depending on the direction in which data has to be moved, connectors are classified as source and sink connectors. Source connectors collect data from an external system and write it to a Kafka topic. Whereas, sink connectors move data from a Kafka topic to other external systems. Many of these connectors have been open-sourced and are available in the public domain for use.

Tasks are the implementation of how data is copied to and from Kafka. These tasks are coordinated by connectors that perform the actual movement of data. A single job can be broken down into multiple small tasks to increase parallelism.

Workers are responsible for executing connectors and tasks. A Kafka Connect cluster is a group of workers working together to execute connectors and tasks. Depending on the number of workers, workers are further classified into two modes: Standalone mode and distributed mode.

In standalone mode, a single worker process is responsible for executing all connectors and tasks. A few settings are required, as there is only one worker involved. Fault tolerance is not guaranteed in this scenario. This mode is useful for development and testing Kafka Connect on local machines.

In distributed mode, many worker processes start simultaneously and automatically coordinate between themselves to schedule the execution of connectors and tasks. As many worker processes run together, fault tolerance is ensured. When a worker stops working or a new worker joins the cluster, tasks are rebalanced among existing workers. The following image illustrates the concepts of workers, tasks and connectors.



Kafka Connect Cluster

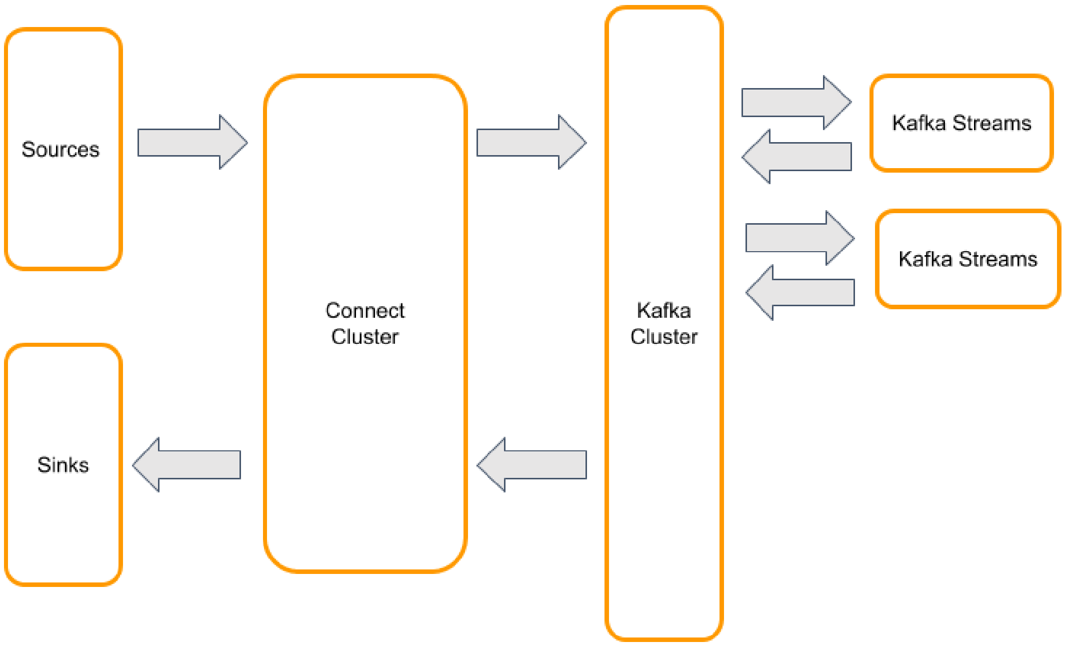
A typical Kafka Connect cluster is made of many workers. Each worker is responsible for executing connectors and tasks.

Kafka Streams

Kafka Streams is a client library used to build applications and microservices. The input and the output data is stored in the Kafka cluster. Kafka Streaming applications take the data present in a Kafka topic, apply some operations on it and write it back to some other Kafka topic. It is available through a Java library.

The following image shows a typical Kafka Stream setup.

KAFKA STREAMS



This is how a typical Kafka Streams setup looks. The data is read from the Kafka topic, processed and then written back to some other Kafka topic. This allows the processing of the stream of data in real-time. The streaming application reads data as and when it is written in the input Kafka topic, applies some operations on it and writes it back to another Kafka topic in near real-time.

It is primarily used for:

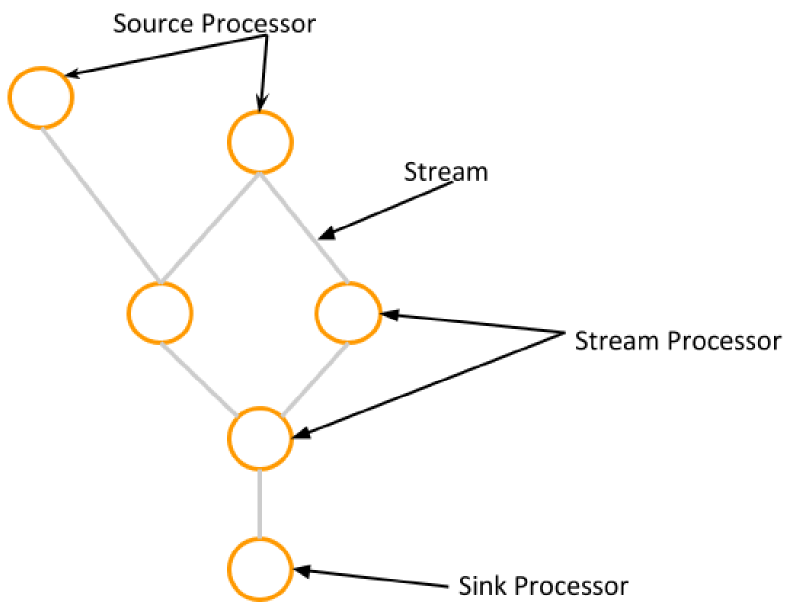
- Transformations,
- Simple processing,
- Anomaly detection and
- Monitoring.

Stream Processing Topology

A stream represents an unbounded, continuously updating data set. It is an ordered, replayable and fault-tolerant sequence of immutable data records, where a data record is defined as a key-value pair. Any program that processes a stream of data is called a Stream Processing Application.

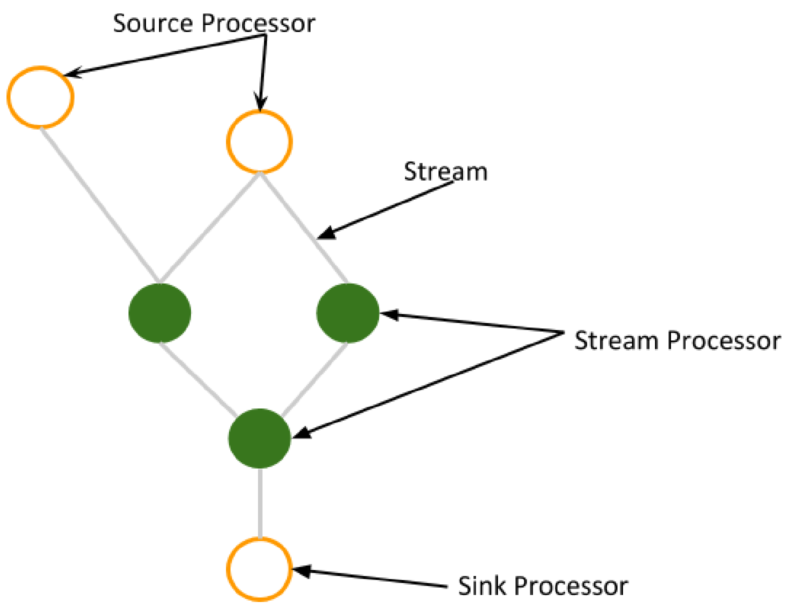
The following image illustrates the stream processing topology.

STREAM PROCESSING TOPOLOGY



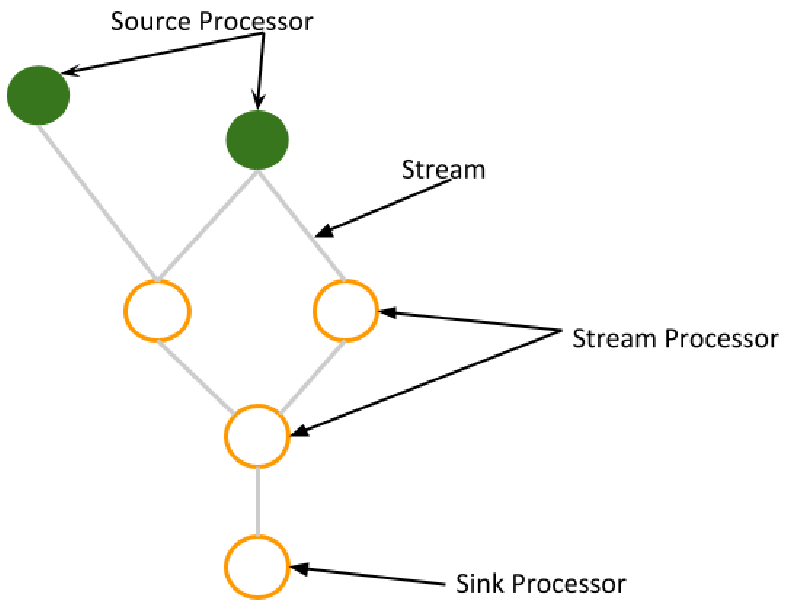
A processing topology is a graph wherein stream processors act as nodes that are connected by streams, and these streams act as edges.

STREAM PROCESSOR



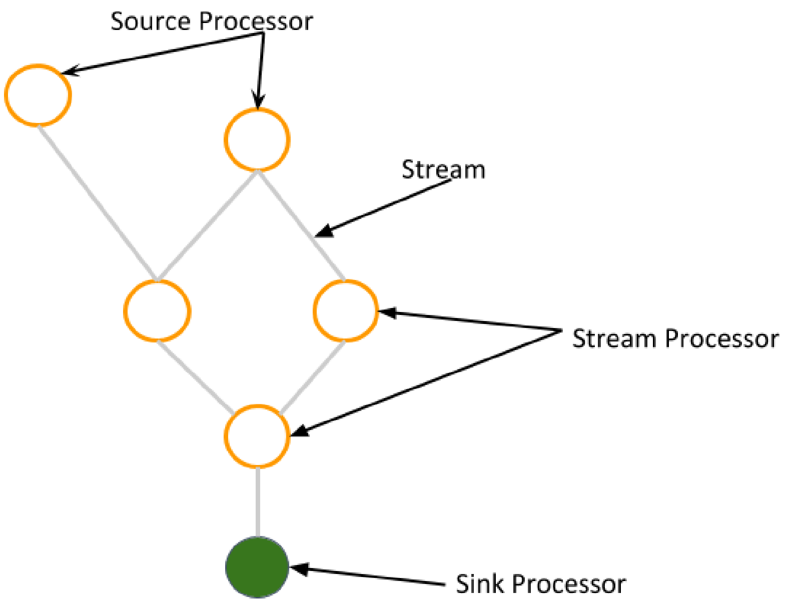
A **stream processor** is a node in the processor topology. It represents a processing step to transform data in streams by receiving one input record at a time from its upstream processors in the topology and applying its operation on it. It may subsequently produce one or more output records to its downstream processors.

SOURCE PROCESSOR



A **source processor** is a special type of stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forwarding them to its downstream processors.

SINK PROCESSOR



A **sink processor** is a special type of stream processor that does not have downstream processors. It sends any received records from its upstream processors to a specified Kafka topic.