Apache Kafka

1. Need for Apache Kafka

- When the focus was moving towards Data Analytics, we required a framework that can deal with 3 V's i.e.
  a) Volume of data
  b) Velocity at which data can be processed
  c) Variety of data

- RDBMS is believed to be the most effective tool to store, manage, manipulate & query the data so even today there is no replacement for RDBMS.
- The major flaw with RDBMS is to deal with 3 V's. RDBMS has a lot of restrictions
  a) Capable of dealing only with a few GBs of data.
  b) Can deal with only normalized data i.e., data should be structured.
  c) Centralized in nature.
  d) Only support for vertical scaling (adding power/RAM/Processors) not horizontal scaling (adding nodes)

No.	Horizontal Scaling (Scale In/Out/ Across)	Vertical Scaling (Scale up/ down)
1.	It can be achieved by increasing / decreasing the number of nodes in a cluster to handle an increase or decrease in workload.	It can be achieved by increasing / decreasing the power of a system to handle an increase or decrease in workload.
2.	E.g., Add / Reduce the no. of VM in a cluster of VMs.	E.g., Add / Reduce the CPU or memory capacity of the existing VM
3.	Workload is distributed across multiple nodes. Here multiple requests get distributed across multiple machines over the network.	A single node handles the entire workload. It relies on multi – threading on the existing machine to handle multiple requests at the same time.
4.	High Performance	Low Performance
5.	There is no downtime because other machines in the cluster offers backup.	High downtime chance since it's a single source of failure.
6.	Load balancing is required here to actively distribute workload across the multiple nodes.	Loading balancing not required in the single node.

- These few limitations brought the need to develop a framework / Architecture that can help to achieve 3 V's

- In between 2000 – 2002, Google started research called Google File System (GFS) for driving their business on the data they were receiving (Data Analytics). It internally used 2 important concepts
  1. Distributed File System
  2. Distributed Processing Engine
- In 2002, Google published a white paper about the concept of distributed framework where they had implemented distributed store & processing engine.
- This white paper was picked by a developer named Daug cutting. He developed a framework called HADOOP to manage all the 3 V's
  1. For implementing Distributed Storage (DS), he used Hadoop Distributed File System (HDFS)
  2. For implementing Distributed Processing (DP), he used Map Reduce Batch Processing System
- Using HADOOP, it was proved that Distributed way of storing & processing data achieved Massive parallel processing that is far more efficient than the traditional approach (RDBMS).
- To overcome the problem of RDBMS vertical scaling, the idea is to have a cluster – based environment where multiple systems are connected over a common network rather having one high end database server. They decided to go with commodity H/W machines that can host this HADOOP framework & can work in a collaborative fashion as cluster – based computing.
- **HyperScaler** – Seamlessly extending / downsizing the capacity of a cluster based on the requirement.

- So, the main intention behind developing the HADOOP framework was to support OLAP (Online Analytical Processing) kind of transaction not for OLTP (Online Transaction Processing).

- Drawbacks of HADOOP
    1. The major bottleneck is to do Full – Fledged Map Reduce (MR) coding on the large volume of data stored.
    2. A lot of disk input – output operations were involved while writing MR code i.e., the HADOOP framework is designed in such a way that all intermediate results will be buffered on the physical disk & the subsequent interfaces will take it from physical disks & they will further reduce it to the final result.
    3. HADOOP framework was specifically designed for batch processing, not for Real – time Data processing.

- Different firms came up the ways to configure interface tools with Distributed Storage & Distributed processing without going in MR coding.
    1. **Yahoo** came up with a scripting tool called **PIG** to achieve some kind of ETL things & configure it to HADOOP (Extract, transform, load). This PIG engine will internally reduce script to MR coding.
    2. **Facebook** came out with HIVE. It gives an environment where we can use SQL (HQL – High QL) & write any type of complex analytical query to trigger it on a huge volume of data stored in this architecture. So, this abstracts us from writing the Java/Python/R code just write a query.

- At the end of 2008, HADOOP reached its saturation point in terms of business, clients & end user's expectations. After 2008, the focus started moving towards an even more faster Processing framework irrespective of whether it will do OLAP or OLTP transactions.

- Keeping the drawbacks of HADOOP framework in mind, Apache Spark was developed around 2008, the need for real – time data processing (process data dynamically) came into the picture.
- Apache Spark is a lightning – fast distributed data processing engine.
- Batch Processing: Process all the data at once. (Data persisted over a period of time pushed to cluster & process it)
- 2 aspects were only considered while developing Spark:
    1. Completely developed as In-Memory tool i.e., all the data in processing state should be kept on RAM.
    2. Support for Batch processing as well as Real – time Data processing.

- Apache Spark has the capability of capturing the data in real-time or near real-time & then respond to the request. **E.g.,** Report Generation for the count of Tweets every 10 minutes.

- Limitations of Apache Spark
    1. It doesn't come with its own storage. So, real – time data processing is fine but how batch processing will be done. So, data needs to be stored at NFS (Network file System) **or** integrated with HDFS (HADOOP Distributed File System) **or** pull the data from RDBMS **or** NoSQL – DB, **or** AWS – S3
    2. Spark framework doesn't have the capacity to keep track of what was the data flown when a particular node is down & where should it continue because there is no mediator to manage.
    3. Spark framework doesn't have the capacity to keep on gathering data for a particular interval of time because in this case input & output buffer will be huge & there are likely chances that RAM might overflow.

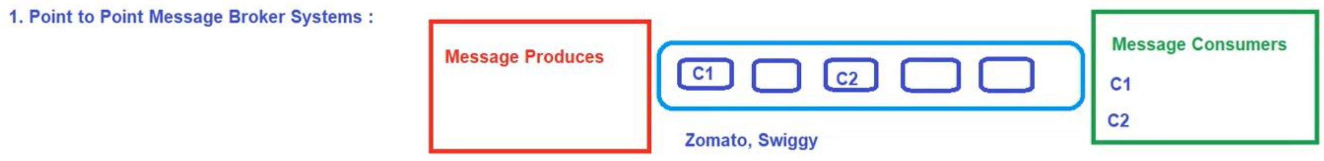
- To overcome the limitation of Apache Spark, there is a necessity of some mediator in between i.e., A Message Broker so that this broker can take the responsibility of reading the data streams, buffering it for a particular duration, dealing with all fault tolerance & in case some nodes go down, from which offset to continue.

- Initially Kafka was developed majorly with above intensions.
- At the end of 2011, Kafka open – sourced to Apache foundation & lately developers who previously worked on Kafka left LinkedIn & built a new firm called Confluent in 2014. Confluent developers added more components / functionality to the framework along with Kafka to release it as an Enterprise Kafka version & they launched it around 2015 & gained popularity around 2018.

- The only aspect we look at Kafka is to act as a Message Broker as long as it was Apache Kafka. On the other end, the consumer use to be mostly spark streaming applications.

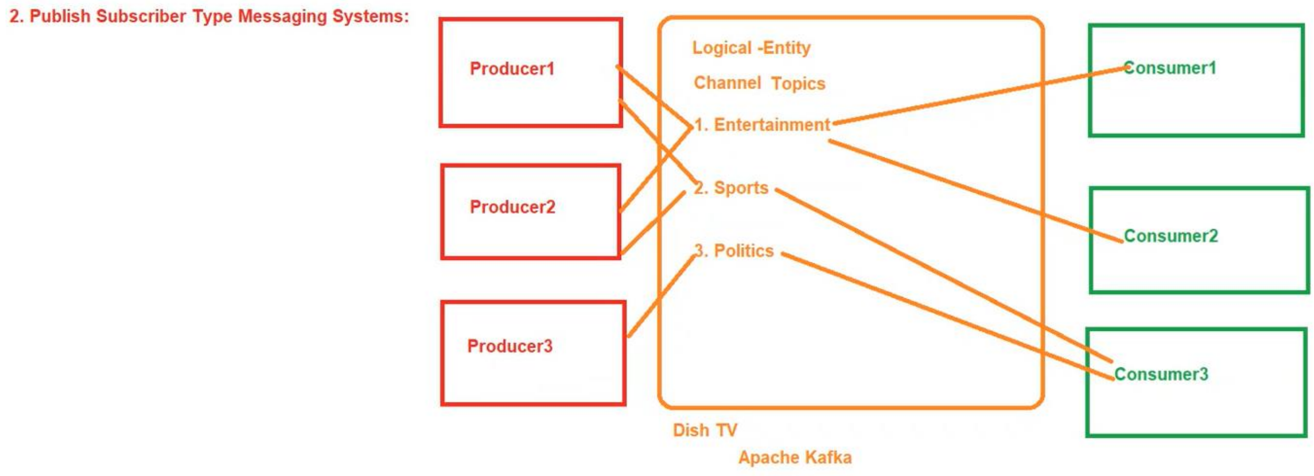
- Now Kafka ecosystem can itself expand & can accommodate the processing part as well. As Kafka evolves, it might take over Spark.

2. Message Brokers – (Point to Point, Publish – Subscriber)



a) Point to Point Messaging System (P2P)

- In this messaging system, each message produced by the producer will be intended for a specific consumer & once the message is consumed it will no longer be available on the queue i.e., Consumer c1 will receive only those messages specified only by c1.
- E.g., Orders placed on Zomato, Swiggy will be specifically for a particular consumer & once you consume it, it will no longer exist in the queue.



b) Publish Subscriber Messaging System

- In this messaging system, producers are publishing their messages to the specific entity/identifier & the consumer can subscribe to any of these entities/channel to receive the data.
- Here, each producer can produce to multiple consumers & each consumer can consume data from multiple producers.
- E.g., Dish TV where multiple consumers can subscribe to multiple packages. That way subscribers to a particular package will receive all channels in that particular package.
- Kafka falls under Publish subscriber messaging system.
- In terms of Kafka, we call this logical entity as a Topic. So, the topic is the one thing that will relate the respective producer to the respective consumer.

3. Apache Kafka Intro

- Kafka is an opensource distributed publish – subscribe messaging system. It is fast, durable, scalable & fault tolerant.
- It acts as a stream processing platform using KStream library & can connect to external system for data import/export by Kafka Connect.
- It's suitable for handling large – scale streaming data.
- The design is highly influenced by Transaction logs.
- Kafka is a distributed system consisting of servers & clients that communicate via a high – performance TCP network protocol. It can be deployed on bare – metal hardware, virtual machines, & containers in on – premise as well as cloud environments
- Servers:
  - Kafka runs as a cluster of one or more servers that can span multiple datacenters or cloud regions.
  - Some of these servers form the storage layer, called the Brokers.
  - Other servers run Kafka Connect to continuously import & export data as event streams to integrate Kafka with our existing systems such as Relational databases or other kafka clusters.
- Clients:
  - They allow you to write distributed applications & microservices that read, write, & process streams of events in parallel, at scale, & in a fault – tolerant manner even in the case of network problem or m/c failures.
  - Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community; clients are available for Java & Scala including the high-level kafka streams library.

4. Use cases of Apache Kafka

- Building real – time streaming data pipeline
- Building real – time streaming application
- CDC (Change Data Capture)
- Messaging, Website Activity Tracking, Metrics,
- Log Aggregation, Commit log

Event Streaming

- Event Streaming is the practice of
  - capturing data in real – time from event sources like databases, sensors, mobile devices, cloud services, & software application in the forms of streams of events;
  - storing these events durably for later retrieval;
  - manipulating, processing & reacting to the event streams in real – time as well as retrospectively;
  - routing the event streams to different destination technologies as needed.

- Event streaming thus ensures a continuous flow & interpretation of data so that the right information is at the right place, at the right time.

UseCases of Event Streaming:

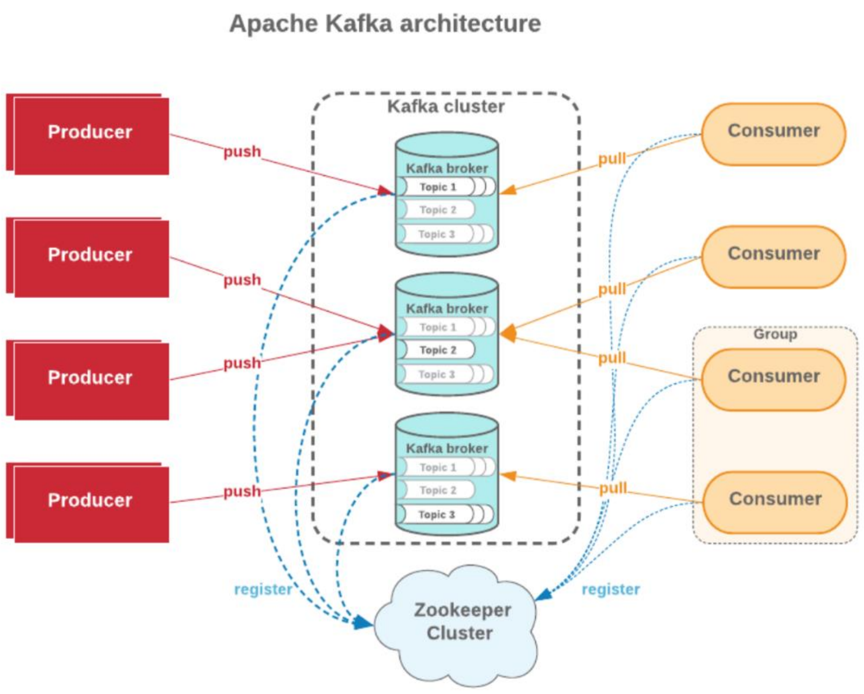
- To process payments & financial transactions in real – time, such as in stock exchanges, banks, insurance.
- To track & monitor cars, trucks, fleets, & shipments in real – time such as in logistics & automotive industry.
- To serve as the foundation for data platforms, event – driven architecture, & microservices.
- To connect, store & make available data produced by different divisions of a company.

5. Key features of Apache Camel

a) Scalability	f) Real – time stream processing
b) Durability	g) Data Retention policies
c) Fault tolerant	h) Dynamic Reconfiguration
d) Partitioned	i) Community & Support
e) High Throughput	

6. Apache Kafka Architecture

- In this architecture, we have Producer Group (containing n number of producers), consumer Group (containing n number of consumers) & in between comes Kafka ecosystem.



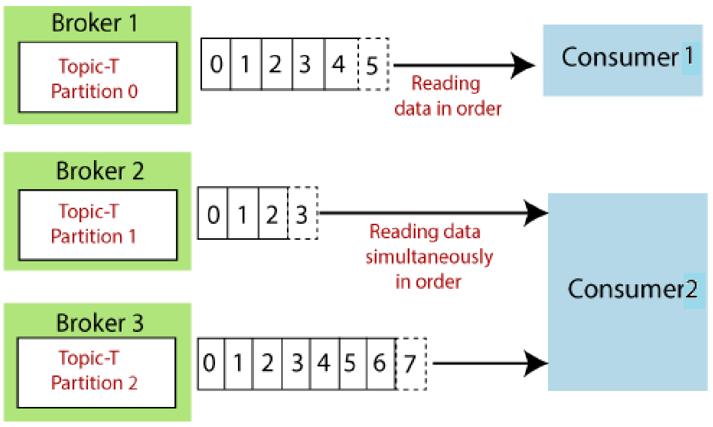
- This Kafka ecosystem primarily consists of:
  a) Kafka Cluster – Primary component of kafka ecosystem
  b) Kafka brokers/Servers/Nodes – Cluster composed of n number of brokers.
  c) Zookeeper – Cluster coordinator service. It helps in interaction & coordination b/w producer group, consumer group & kafka ecosystem.
  d) Topic – Topics are logical channels or categories to which messages are published.

Apache Kafka Terminology

1. Producer – A producer can be any application who can publish message to a topic.

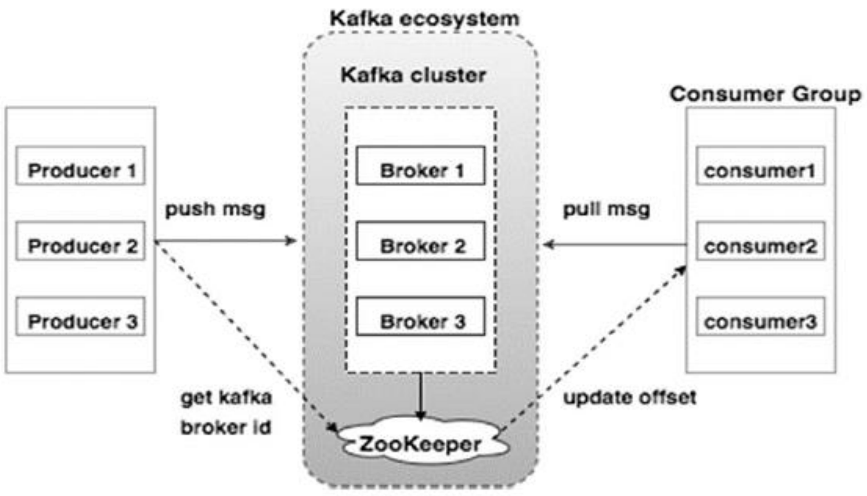
2. Consumer – A consumer can be any application that subscribes to a topic & consuming the messages. A consumer can subscribe to multiple topics.

3. Consumer Group
- Multiple consumers can be grouped under one label called Consumer Group.
- Each record published to a topic is delivered to one consumer instance within the subscribed consumer group.
- Each consumer group processes a subset of partition, allowing for parallel processing & load distribution.



4. Zookeeper

- It acts as a centralized & reliable coordination service for Kafka, ensuring that the distributed components of a kafka cluster can work together seamlessly.
- It helps in managing the dynamic nature of kafka clusters, providing fault tolerance, & enabling coordination among different components.
- It maintains metadata, conducts leader election & helps manage the overall state of the cluster.
- Cluster coordination means conducting leader election, Broker registration & management.
- Topic & Partition information, Consumer Group management, Broker & Topic health monitoring.



5. Kafka Broker

- Kafka brokers are individual servers or machines within the kafka cluster
- They store & manage data, handle producer & consumer requests & participate in the replication & distribution of data.
- There is one broker that is responsible for coordinating the cluster. This broker is called **Controller**.
- In kafka, a special node known as the "controller" is responsible for managing the registration of brokers in the cluster. Broker liveness has 2 conditions:
  a) Brokers must maintain an active session with the controller in order to receive regular metadata updates.
  b) Brokers acting as followers must replicate the writes from the leader & not fall "too far" behind.

Graceful Shutdown

- The Kafka cluster will automatically detect any broker shutdown or failure and elect new leaders for the partitions on that machine.
- This will occur whether a server fails or it is brought down intentionally for maintenance or configuration changes. For the latter case, Kafka supports a more graceful mechanism for stopping a server than just killing it.
- When server is stopped gracefully, it has 2 optimization it will take advantage of:
  a) It will sync all its logs to disk to avoid needing to do any log recovery when it starts. Log recovery takes time so this speeds up intentional restarts.
  b) It will migrate all leader partitions to other replicas prior to shutting down. This will make the leadership transfer faster.
- Control leadership requires this config: **controller.shutdown.enable** = true
- Controlled shutdown will only succeed if all the partitions hosted on the broker have replicas (replication factor > 1 & at least one of these replicas is alive).

Balancing leadership

- Whenever a broker stops or crashes, leadership for that broker's partitions transfers to other replicas. When the broker is restarted, it will only be a follower for all its partitions, meaning it will not be used for client reads & writes.
- To avoid this imbalance, We can configure as: **auto.leader.rebalance.enable**=true
- By default, the kafka cluster will try to restore leadership to the preferred replicas.

6. Topics

- Topics are logical channels or categories to which messages are published.
- Topics can be divided into partitions for scalability & parallelism.

Event/Record/Message

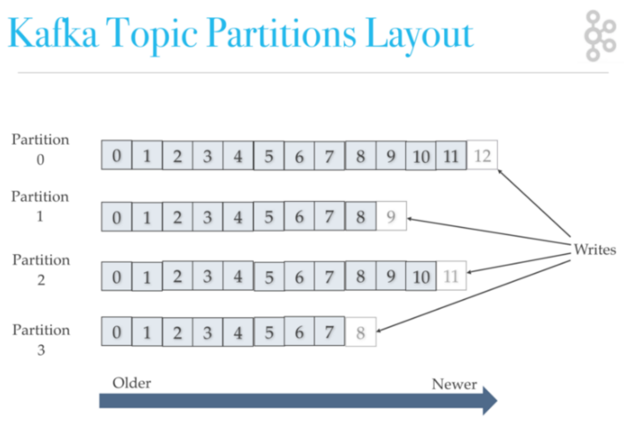
- An Event/Record/Message records the fact that "something happened" in the world or in our business.
- Events are organized & durably stored in Topics.
- Each Event/Record/Message consists of Key, Value & timestamp.
- The key is optional but can be used to identify messages from the same data source.

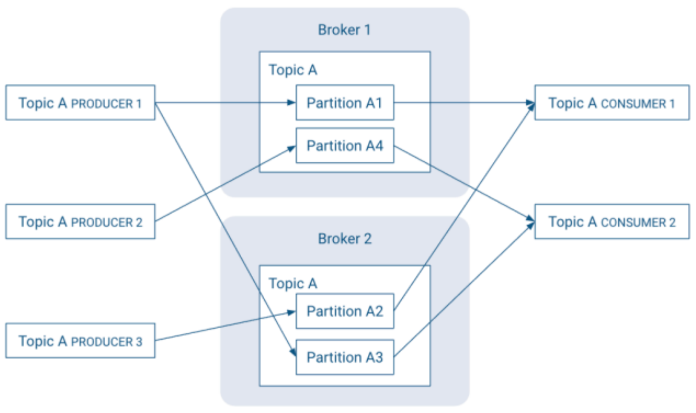
Record order & Assignment

- By default, Kafka assigns records to a partitions round – robin. There is no guarantee that records sent to multiple partitions will retain the order in which they were produced.
- Within a single consumer, our program will only have record ordering within the records belonging to the same partition. This tends to be sufficient for many use cases, but does add some complexity to the stream processing logic.
- Kafka guarantees that records in the same partition will be in the same order in all replicas of that partition
- If the order is important, the producer can ensure that records are sent to the same partition. The producer can include metadata in the record to override the default assignment in one of 2 ways
  - The record can indicate a specific partition.
  - The record can include an assignment key.
- The hash of the key & the number of partitions in the topic determines which partition the record is assigned to. Including the same key in multiple records ensures all the records are appended to the same partition.

7. Partition

- Topics are broken into order commit logs called Partitions. They are the basic unit of Parallelism.
- Each partition can be placed on other brokers to allow multiple consumers to read it parallelly.
- Partitioning is the concept of segregating data into subsets & increasing the parallelism of writing & reading the data. On the other hand, if one of the brokers goes down, we need more copies of that topic.
- So, we need to define while creating a topic is how many replications should happen for that particular topic i.e., Replication Factor.
  - Note: (Ideal cases) [N – No. of nodes/brokers & P – No. of Partitions]
  - Case 1: If N == P for Topic T1 then each node will contain 1 Partition.
  - Case 2: If N > P for Topic T1 then each N nodes will have 1 Partition & other (N – P) nodes will be empty.
  - Case 3: If N < P for Topic T1 then there may be a possibility that Nodes will have more than 1 partition.
- Replication of data is always done at partition level.
- Each topic has one leader partition. If the replication factor is greater than one, there will be additional follower partitions (For the replication factor M, there will be M – 1 follower partitions)
- Any Kafka client (a producer or consumer) communicates only with the leader partition for data. All other partitions exist for redundancy & failover. Follower partitions are responsible for copying new records from their leader partitions.
- Ideally, the follower partitions have an exact copy of the contents of the leader. Such partitions are called **in-sync replicas** (ISR).





8. Partition Segments

- Kafka divides topics partition data into segment files (with .log suffix) stored on the file system.
- Each segment file is named using the offset of the first message (base offset) contained. For e.g., the segment 04.log contains the message with offset 4 as first entry.
- The last segment in the partition is called the active segment & it's the only segment to which new messages are appended to.

01.log

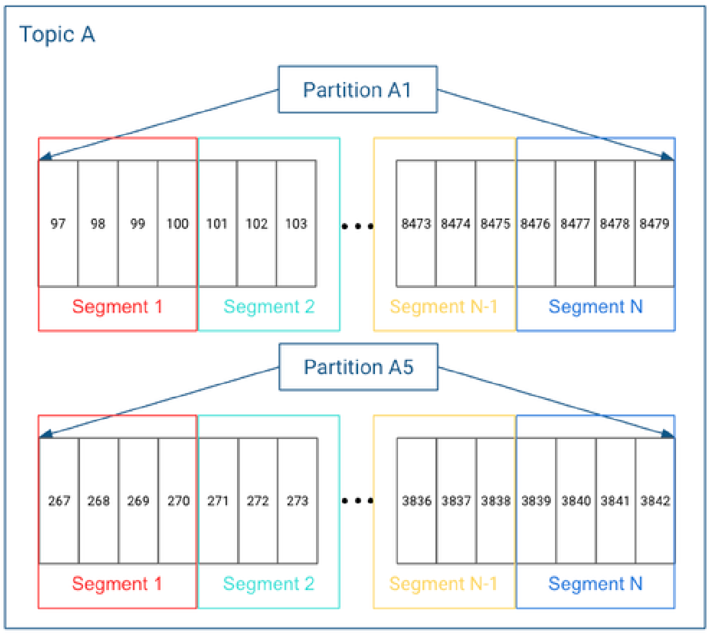
Offset	Key	Value
1	1001	4 Privet Dr
2	1002	221B Baker Street
3	1003	Milkman Road

04.log

Offset	Key	Value
4	1002	21 Jump St
5	1001	Paper St

06.log (active segment)

Offset	Key	Value
6	1001	Paper Road 21



- When the segment file reaches a certain size or age, Apache kafka will create a new segment file. This can be controlled by following configs

No.	Config.	Description
1.	segment.bytes	creates a new segment when current segment becomes greater than this size. This setting can be set during topic creation & defaults to 1GB
2.	segment.ms	forces the segment to roll over & create a new one when the segment becomes older than this value.

9. Offset Management

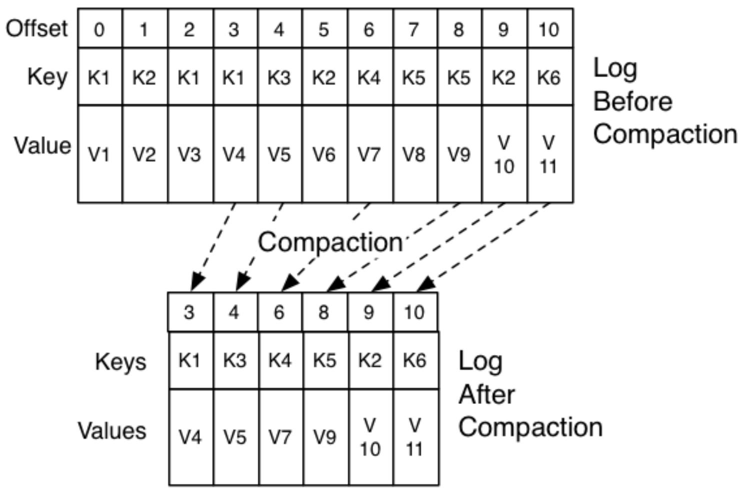
- Offset represents the position of a consumer within a partition.
- Consumers commit offsets to Kafka for tracking their progress. This ensures that they can resume processing from the last committed offset in case of failure or restart.
- Kafka provides the option to store all the offset for a given consumer group in a designated broker (for that group) called the group coordinator i.e., any consumer instance in that consumer group should send its offset commits & fetches to that group coordinator(broker).
- Consumer groups are assigned to coordinators based on their group names.
- A consumer can look up its coordinator by issuing a **FindCoordinatorRequest** to any kafka broker & reading the **FindCoordinatorResponse** which contains the coordinator details. The consumer can then proceed to commit or fetch offsets from the coordinator broker. In case the coordinator moves, the consumer will need to rediscover the coordinator. Offset commits can be done automatically or manually by consumer instance.
- When the group coordinator receives an **OffsetCommitRequest**, it appends the request to a special compacted kafka topic name **__consumer_offsets.**
- The broker sends a successful offset commit response to the consumer only after all the replicas of the offset topic receive the offsets. In case the offsets fail to replicate within a configurable timeout, the offset commit will fail and the consumer may return the commit after backing off.

10. Topic Partition Replication

- Each partition has multiple replicas for fault tolerance. Replicas are distributed across different brokers.
- One replica is designated as the Leader & others are followers.

11. Log Compaction

- Kafka supports log compaction i.e., retaining only the latest message for each key in a partition.
- This is useful for scenarios where maintaining the latest state for a set of keys is critical.
- It addresses use cases & scenarios such as restoring state after application crashes or system failure, or reloading caches after application restarts during operational maintenance.



11. Kafka Connect

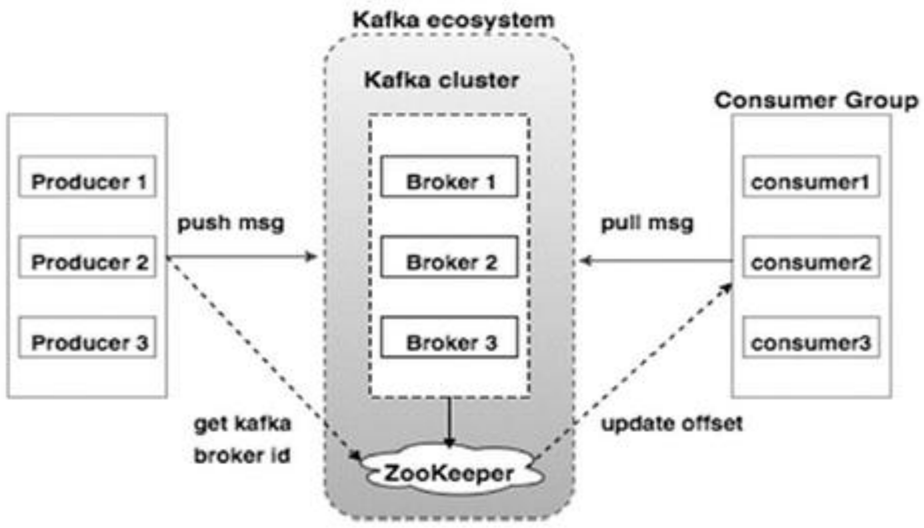
- Kafka connect is a framework for integrating Kafka with external system.
- It simplifies the development of connectors for ingesting data from or delivering data to various sources & sinks.

12. Kafka Streams

- Kafka Stream is a stream processing library that allows developers to build real – time applications & microservices using kafka as underlying infrastructure.

7. High level Architecture of Kafka

- On high level overview, there is a cluster containing multiple brokers where one is leader & rests are followers having replica of data.
- Now the leader broker hosts a leader partition where the producer writes & consumer reads.
- Rest broker hosts follower partitions & hence will replicate the same data to different disk location for fault tolerance.



Architecture Flow

1. Producer writes data to a Leader i.e., Producer sends message to the leader partition of the specified topic. The leader appends messages to its local log.

2. Replica synchronization i.e., Replication of data is always done at partition level. Leader replicates messages to followers, ensuring they have the same set of messages.

ISR (In – Sync Replica) represents replicas that are up to date with the leader

3. Consumer reads from the Leader i.e., consumers always read from the leader partition. Once read, it acknowledges the broker that it has read the message successfully & also commits the offset so that in case of failure, it can come back & read from the same offset (the leader ensures followers are kept in sync)

Fault Tolerance – In the event of a leader or broken failure, kafka ensures quick leader election & data replication from in-sync replicas.

Scalability – Kafka scales horizontally by distributing partitions across multiple brokers.

Zookeeper coordinates leader election, maintains metadata & manages the overall state of the kafka cluster.

8. Role of Zookeeper in Kafka

- It acts as a centralized & reliable coordination service for Kafka, ensuring that the distributed components of a Kafka cluster can work together seamlessly.
- It helps in managing the dynamic nature of kafka clusters, providing fault tolerance, & enabling coordinating among different components.
- For interaction b/w producers & brokers, Zookeeper helps the producer in term of Broker id i.e., to which broker producers are supposed to publish the message.
- Similarly, for interaction b/w brokers & consumers, Zookeeper helps the consumer in terms of Broker id & offset of the data.
- Cluster Coordination – Leader Election, Broker registration & management.
- Topic & Partition information
- Consumer Group management
- Broker & Topic health monitoring

Note: From Kafka 2.8.0, there have been efforts to reduce Kafka's dependency on Zookeeper, with ongoing work to replace it with a self – managed metadata store.

9. Core APIs of Kafka – Admin, Producer, Consumer, Connect, Stream

a) Admin API – It allows managing & inspecting topics, brokers, & other kafka objects.

b) Producer API – It allows application to send streams of data to topics in the kafka cluster.

c) Consumer API – It allows applications to read streams of data from topics in the kafka cluster.

d) Connect API

- **Kafka Connect API** is used to build & run reusable data import/export connectors that consume (read) or produce (write) streams of events from & to external systems & applications so they can integrate with Kafka.
- It allows implementing connectors that continuously pull from some source system or application into kafka or push from kafka into some sink system or application.
- E.g., A connector to a relational database like PostgreSQL might capture every change to a set of tables. However, in practice, you typically don't need to implement your own connectors because the Kafka community already provides hundreds of ready – to – use connectors.

e) Stream API

- It allows transforming streams of data from input topics in the Kafka cluster.
- **Kafka Stream API** is used to implement stream processing applications & microservices. It provides high-level functions to process event streams, including transformations, stateful operations like Aggregations & Joins, windowing, processing based on event – time, & more.

10. Kafka Connect

- Kafka Connect is a framework/tool for scalably & reliably streaming data between Apache kafka & other systems.
- It makes it simple to quickly define connectors that move large collections of data into & out of Kafka.
- Kafka Connect can ingest entire databases or collect metrics from all our application servers into Kafka topics, making the data available for stream processing with low latency.
- An export job can deliver data from Kafka topics into secondary storage & query systems tor into batch systems for offline analysis.

Features of Kafka Connect

1. **A common framework for Kafka connectors** – Kafka Connect standardizes integration of other data systems with kafka, simplifying connector development, deployment, & management.
2. **Distributed & standalone modes** – scale up to a large, centrally managed service supporting an entire organization or scale down to development, testing, and small production deployments.
3. REST interface
4. Automatic offset management
5. Distributed & scalable by default
6. Streaming/batch integration

Running Kafka Connect

- Kafka Connect currently supports 2 modes of execution: Standalone (Single Process), Distributed

1. Standalone mode

- In standalone mode, all work is performed in a single process. This configuration is simpler to setup & get started with & may be useful in situations where only one worker makes sense (e.g. collecting log files), but it doesn't benefit from some of the features of Kafka connect such as fault tolerance.
- We can start a standalone process with the following command:

  bin/**connect-standalone.sh** config/**connect-standalone.properties** [connector1.properties ….]

- All workers (both standalone and distributed) require a few configs:

Configuration	Description
bootstrap.servers	List of kafka servers used to bootstrap connections to kafka.
key.converter	Converter class is used to convert b/w Kafka connect format & the serialized form that is written in Kafka. This controls the format of the keys in messages written to or read from kafka, & since this is independent of connectors it allows any connector to work with any serialization format. Examples of common formats include JSON, & Avro
value.converter	This controls the format of the values in message written to or read from kafka, & since this is independent of connectors it allows any connector to work with any serialization format. Examples of common format include JSON & Avro
plugin.path	A list of paths that contain Connect plugin (connectors, converters, transformations). Before running quick starts, users must add the absolute path that contains the example **FileStreamSourceConnector** & **FileStreamSinkConnector** packaged in **connect-file-"version".jar**, because these connectors are not included by default to the CLASSPATH or the plugin.path of the Connect worker.
offset.storage.file.filename	(Imp. Configuration option specific to standalone mode) File to store source connector offsets.

2. Distributed mode

- It handles automatic balancing of work, allows us to scale up (or down) dynamically, & offers fault tolerance both in the active tasks & for configuration & offset commit data.
- We can start a distributed process with the following command:

bin/connect-distributed.sh config/connect-distributed.properties

- In the distributed mode, Kafka Connect stores the offsets, configs, & task statuses in Kafka topics. It can be manually created or automatically.

Configuration	Description
group.id	(Default value is connect-cluster) unique name for the cluster, used in forming the Connect cluster group; note that this must not conflict with consumer group IDs.
config.storage.topic	(Default value is connect-configs) topic to use for storing connector & task configurations; This should be a single partition, highly replicated, compacted topic.
offset.storage.topic	(Default value is connect-offsets) topic to use for storing offsets; This topic should have many partitions, be replicated, & be configured for compaction.
status.storage.topic	(Default value is connect-status) topic to use for storing statuses; This topic can have multiple partitions, & should be replicated & configured for compaction.

- In distributed mode, the connector configurations are not passed on the command line. Instead, use the REST API described below to create, modify & destroy connectors.

Configuring Connectors

- In both standalone & distributed mode, they are included in the JSON payload for the REST request that creates (or modifies) the connector.
- In standalone mode, these can also be defined in a properties file & passed to the Connect process on the command line.
- Most configs are connector dependent, so they can't be outlined here. However, there are a few common options:

Configuration	Description
Name	Unique name for the connector. Attempting to register again with the same name will fail.
connector.class	The java class for the connector
tasks.max	Maximum no. of tasks that should be created for this connector. The connector may create fewer tasks if it can't achieve this level of parallelism.
key.converter	(optional) override the default key converter set by the worker
value.converter	(optional) override the default value converter set by the worker
Topics	(Mandatory for Sink Connector) A comma separated list of topics to use as input for this connector
topics.regex	A java regular expression of topics to use as input for this connector.

- For any other options, we should consult the documentation for the connector.
- **Transformations** – Connectors can be configured with transformations to make lightweight message-at-a-time modifications. They can be convenient for data messaging & event routing.

REST API

- Since Kafka Connect is intended to be run as a Service, it also provides a REST API for managing connectors. This REST API is available in both standalone & distributed mode.
- The REST API server can be configured using the listeners configuration option. This field should contain a list of listeners in the following format: protocol://host:port, protocol2://host2:port2. Currently supported protocols are http & https.

  For example: listeners=http://localhost:8080, https://localhost:8443

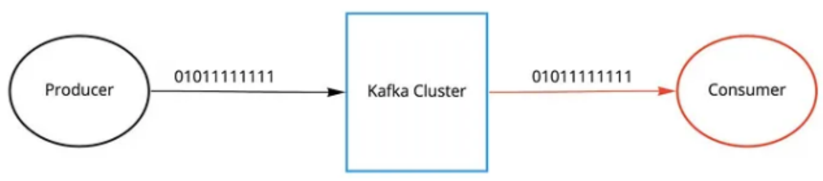
- By default, if no listeners are specified, the REST server runs on port 8083 using the HTTP protocol.
- When using HTTPS, the configuration has to include the SSL configuration. By default, it will use the ssl.* settings.

Configuration	Description
Name	Unique name for the connector. Attempting to register again with the same name will fail.
connector.class	The java class for the connector
tasks.max	Maximum no. of tasks that should be created for this connector. The connector may create fewer tasks if it can't achieve this level of parallelism.
key.converter	(optional) override the default key converter set by the worker
value.converter	(optional) override the default value converter set by the worker
Topics	(Mandatory for Sink Connector) A comma separated list of topics to use as input for this connector
topics.regex	A java regular expression of topics to use as input for this connector.

URIs	Description
GET /connectors	Returns a list of active connectors
POST /connectors	Create a new connector; the request body should be a JSON object containing a string name filed & an object config field with the connector configuration parameters.

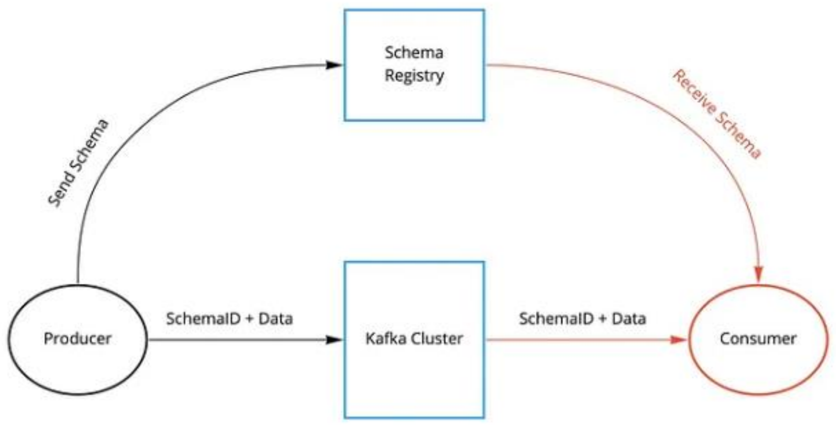
11. Why Schema Registry

- Kafka, at its core, only transfers data in byte format. There is no data verification that's being done at the Kafka cluster level. In fact, Kafka doesn't even know what kind of data it is sending or receiving; whether it's a string or integer.



Producer sending data in byte format to Kafka Cluster and being consumed by a consumer.

- Due to the decoupled nature of Kafka, Producers & Consumers don't communicate with each other directly, but rather information transfer happens via Kafka topic. At the same time, the consumer still needs to know the type of data the producer is sending in order to deserialize it.
- Image if the producer starts sending bad data to kafka or if the data type of our data gets changed, the downstream consumers will start breaking. We need a way to have a common data type that must be agreed upon.
- That's where Schema Registry comes into the picture. It's an application that resides outside of our kafka cluster & handles the distribution of schemas to the producer & consumer by storing a copy of schema in its local cache.

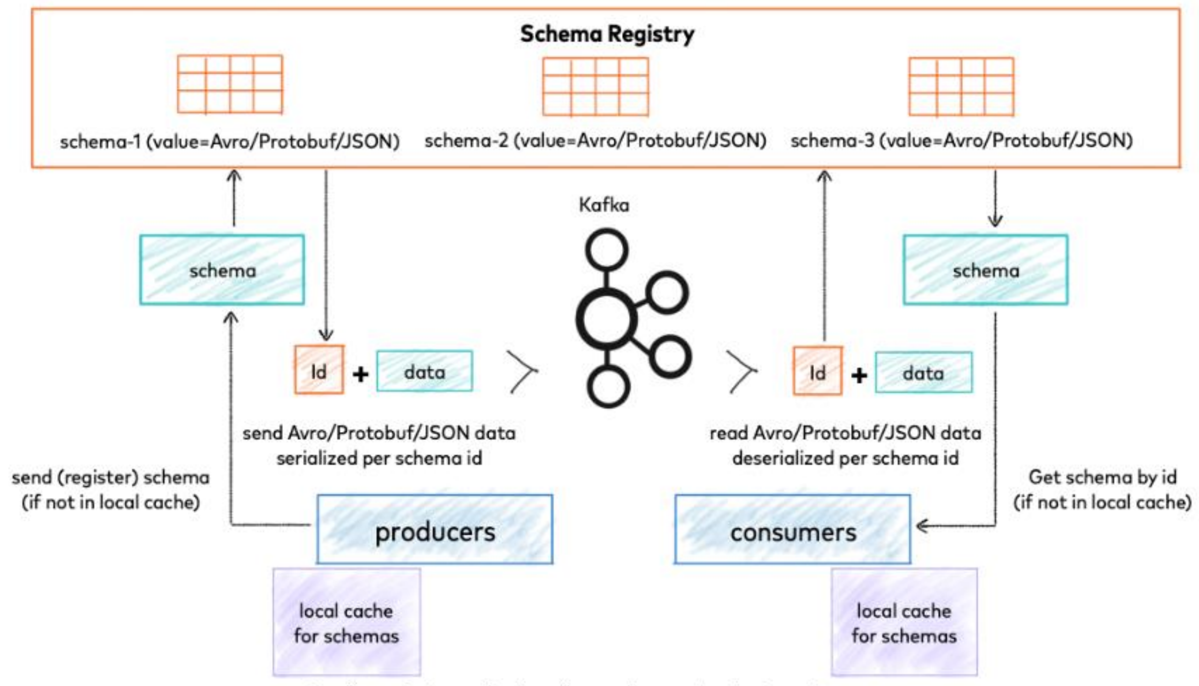


Schema Registry Architecture

- With the schema registry in place, the producer before sending the data to Kafka, talks to the schema registry first & checks if the schema is available.
- If it doesn't find the schema, then it registers & caches it in the schema registry. Once the producer gets the schema, it will serialize the data with the schema & send it to kafka in binary format prepended with a unique schema ID.
- When the consumer processes this message, it will communicate with the schema registry using the schema ID it got from the producer and deserialize it using the same schema.
- If there is a schema mismatch, the schema registry will throw an error letting the producer know that it's breaking the schema agreement.

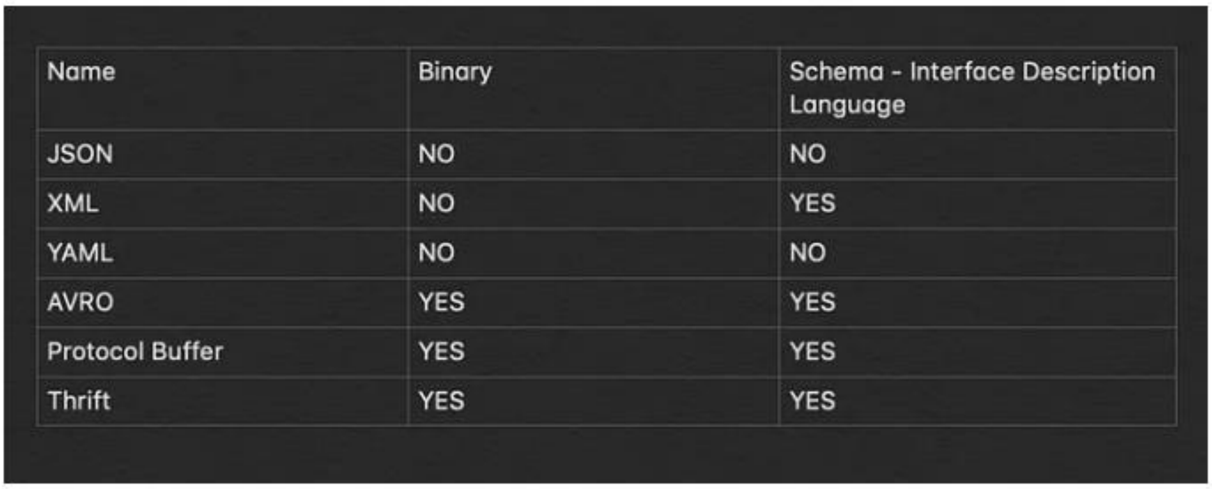
Q. Schema Registry Overview

- Schema Registry provides a centralized repository for managing & validating schemas for topic message data, and for serialization & deserialization of the data over the network.
- Producers & Consumers to Kafka topics can use schemas to ensure data consistency & compatibility as schemas evolve.
- Schema Registry is a key component for data governance, helping to ensure data quality, adherence to standards, visibility into data lineage, audit capabilities, collaboration across teams, efficient application development protocols, & system performance.


Schema Registry
schema-1 (value=Avro/Protobuf/JSON)
schema-2 (value=Avro/Protobuf/JSON)
schema-3 (value=Avro/Protobuf/JSON)
Kafka
schema
schema
Id
data
Id
data
send Avro/Protobuf/JSON data serialized per schema id
read Avro/Protobuf/JSON data deserialized per schema id
send (register) schema (if not in local cache)
Get schema by id (if not in local cache)
producers
consumers
local cache for schemas
local cache for schemas

Confluent Schema Registry for storing and retrieving schemas

Following are some data serialization formats as per the above considerations:



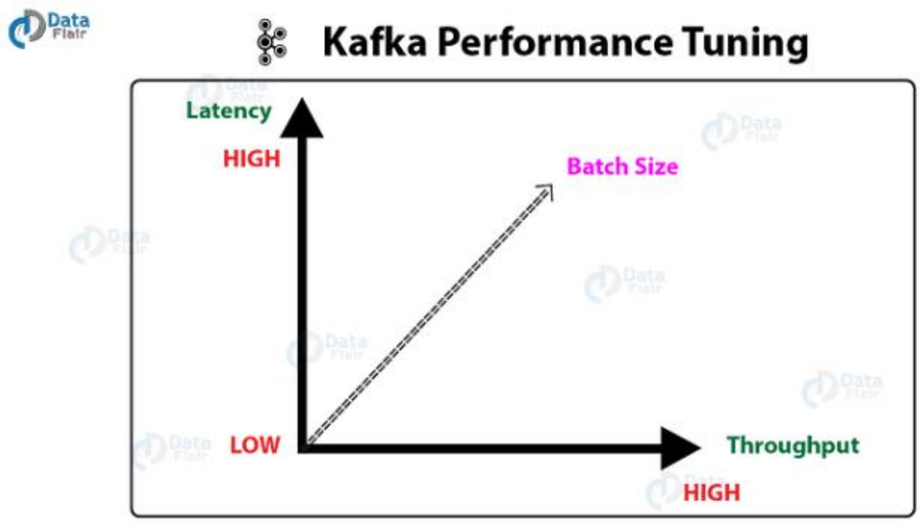
Name	Binary	Schema - Interface Description Language
JSON	NO	NO
XML	NO	YES
YAML	NO	NO
AVRO	YES	YES
Protocol Buffer	YES	YES
Thrift	YES	YES

Comparison of various data serialization formats

12. Kafka Performance tuning

- Performance tuning involve 2 important metrics:
  1. Latency measures – how long it takes to process one event
  2. Throughput measures – how many events arrive within a specific amount of time.

- Most systems are optimized for either latency or throughput. Kafka is balanced for both.
- Tuning our producers, brokers & consumers to send, process, & receive the largest possible batches within a manageable amount of time results in the best balance of latency & throughput for our kafka cluster.
- QueueFullException occurs when the producer attempts to send message at a pace not handleable by the broker.



Kafka Performance Tuning Graph

Tuning Producers

- Two parameters are particularly important for latency & throughput: batch size & linger time

1. **batch.size** – It controls how many bytes of data to collect before sending messages to kafka broker. Setting high value means the producer is sending data all the time, this will give the best throughput. This doesn't impact latency.

2. **linger.ms** – It sets the maximum time to buffer data in asynchronous mode. This improves throughput, but the buffering adds message delivery latency. Increase linger.ms for higher throughput & higher latency in our producer.