**Apache Kafka Key Concepts**

|  |  |
| --- | --- |
| No. | Topics |
| 1. | What is Kafka, Usecase – Event Streaming, Change Data Capure (CDC), Key features of Kafka |
| 2. | Key terminologies, How kafka works, Architecture of Kafka |
| 3. | Core APIs of Kafka – Admin, Producer, Consumer, Connect, Stream |
| 4. | Kafka Imp. Configurations |
| 5. | How to get started with Kafka/Confluent kafka in system (Refer User Manual) |
| 7. | Replication, ISRs (In Sync Replica), Log Compaction |
| 8. | Kafka Connect |
| 9. | Kafka Streams |
|  |  |
|  |  |

**1. What is Kafka, UseCase – Event Streaming, Change Data Capture (CDC), Key features of Kafka**

**Apache Kafka**

* Kafka is an opensource distributed publish – subscribe messaging system. It is fast, durable, scalable & fault tolerant.
* It acts a stream/event processing platform using Kafka Stream library & can connect to external system for data import/export by Kafka Connect.
* The design is highly influenced by Transaction logs.
* It’s suitable for handling large – scale streaming data.
* 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.

**UseCases of Apache Kafka:**

* Building real – time streaming data pipeline
* Building real – time streaming application / Event Sourcing, Streaming processing
* Change Data Capture (CDC)
* 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.

**Key Features of Kafka**

* Durability, Scalability, Fault tolerance, Partitioning
* High Throughput, Real – time Streaming processing
* Data Retention policies, Dynamic reconfiguration
* Community & Support

**2. Key terminologies, Architecture of Kafka**

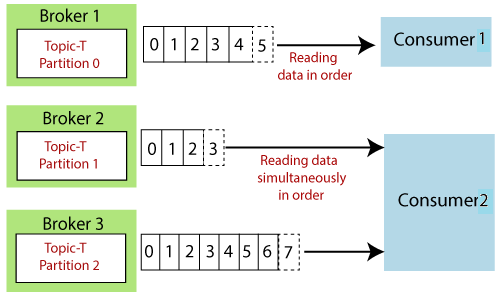
**Key Terminologies**

**1. Producer:** – A producer can be any application who can publish messages to a topic.

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

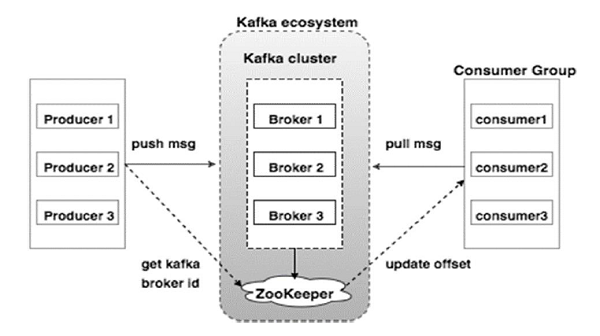
**3. Consumer Group**

* 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 subscribing consumer group.
* Each consumer group processes a subset of partitions, 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:

1. Brokers must maintain an active session with the controller in order to receive regular metadata updates.
2. 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:

1. 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.
2. 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 replica 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. Topic**

* Topics are logical channels or categories to which messages are published.
* Topics can be divided into partitions for scalability & parallelism.
* Each record consists of key, value & timestamp.
* When a topic is created, it is configured with 2 properties: Partition & Replication factor

**Records**

* A record consists of a key/value pair & metadata including a timestamp, headers.
* 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

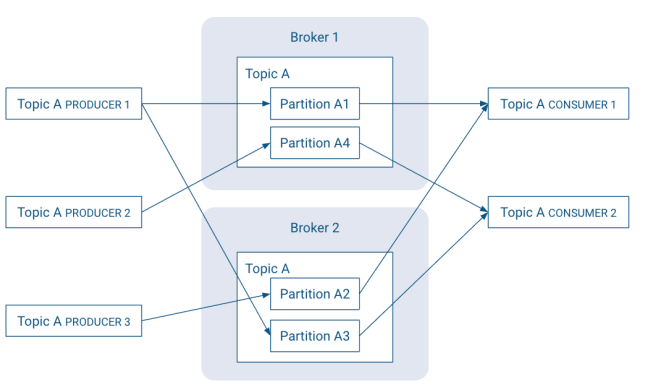
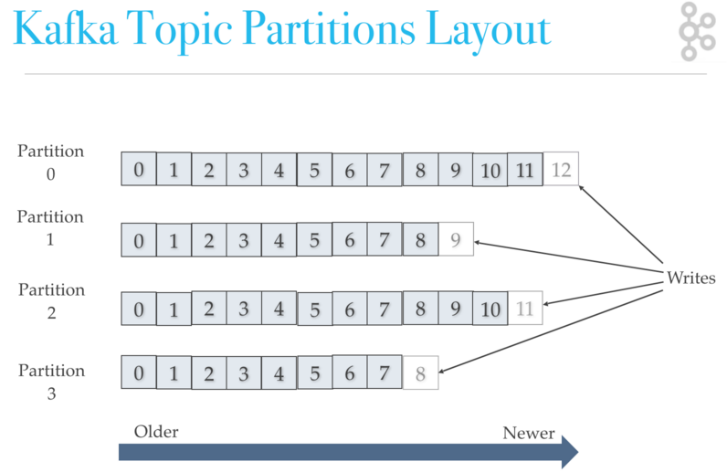
- The record can indicate a specific partition.

- 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.

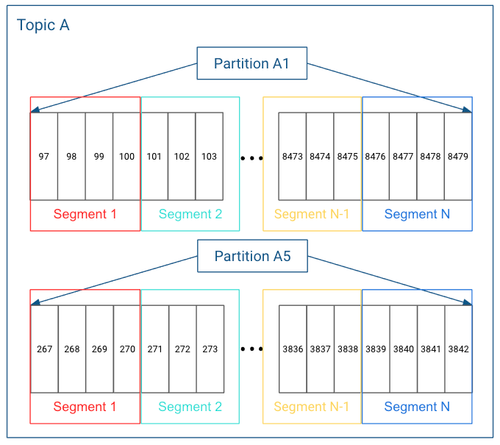
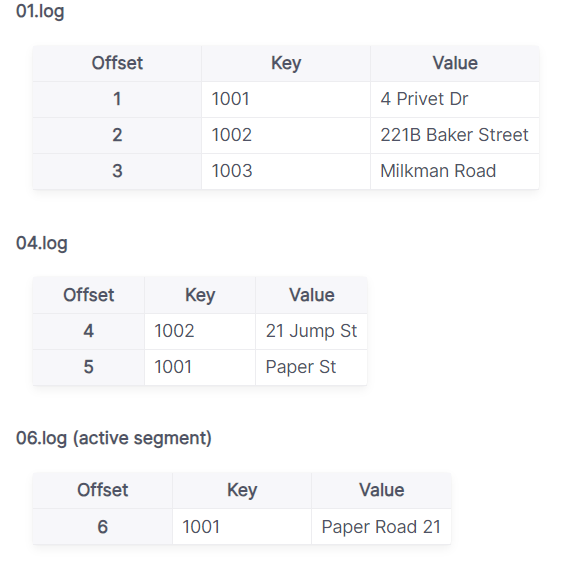
**7. Partitions**

* Topics are broken into order commit logs called Partitions. They are basic unit of Parallelism.
* Each partition can be placed on other brokers to allow multiple consumers to read it parallelly.
* 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.



* 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**

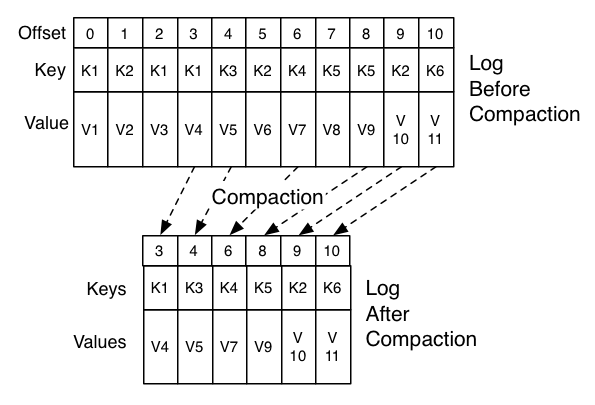
* Offset represents the position of a consumer within a partition.
* Consumers commit offsets to kafka (zookeeper), tracking their progress. This ensures that they can resume processing from the last committed offset in case of failures or restarts.
* Kafka provides the option to store all the offset 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 leader & the other 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.



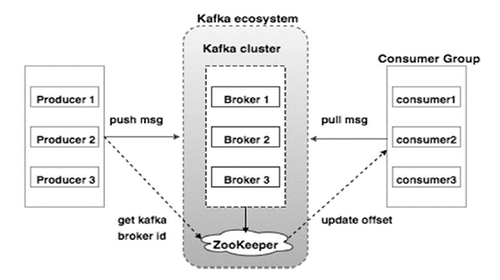
**12. Kafka Connect**

* Kafka Connect is a framework for integrating kafka with external systems.
* It simplifies the development of connectors for ingesting data from or delivering data to various sources & sinks.

**13. Kafka Streams**

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

**Architecture of Kafka**



* On high level overview, there is a cluster containing multiple brokers, where one is leader & rest are followers having replica of data. 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**

**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

**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.

**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 form the same offset. (The leader ensures followers are kept in sync)

**Fault Tolerance** – In the event of a leader or broker failure, Kafka ensures quick leader election & data replication from in – sync replica.

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

**3. 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** – 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) Stream API** – It allows transforming streams of data from input topics in the Kafka cluster.

**4. Kafka Configuration**

|  |  |  |
| --- | --- | --- |
| No. | Configuration | Description |
| 1. | broker.id | Id of broker & must be set to a unique integer for each broker. |
|  | Listeners | **a)** Comma-separated list of URIs we will listen on locally & the listener names. If the listener’s name is not security protocol, listener security protocol map must also be set. i.e., listener.security.protocol.map  **b)** Format: listeners = listener\_name://host\_name:port  **c)** Listener names & port numbers must be unique unless one listener is an IPV4 address & the other listener is an IPV6 address (for the same port)  **d)** Specify hostname as 0.0.0.0 to bind all interfaces & leave hostname empty to bind to default interface.  **e)** E.g. of legal listeners lists:  PLAINTEXT://myhost:9092, SSL://:9091, CLIENT://0.0.0.0:9092, REPLICATION://localhost:9093, PLAINTEXT://127.0.0.1:9092, SSL://[::1]:9092 |
|  | advertised.listeners | **a)** Listeners to publish to Zookeeper for clients to use, if different than the listeners config property. In IaaS environments, this may need to be different from the interface to which the broker binds.  **b)** If this is not set, the value for listeners will be used. Unlike listeners, it is not valid to advertise the 0.0.0.0 meta – address & there can be duplicated ports in this property, so that one listener can be configured to advertise another listener’s address.  **c)** This can be useful in some cases where external load balancers are used. |
|  | listener.security.protocol.map | Map between listener names & security protocols. This must be defined for the same security protocol to be usable in more than one port or IP.  For e.g., internal & external traffic can be separated even if SSL is required for both. Concretely, the user could define listeners with names INTERNAL & EXTERNAL and this property as INTERNAL:SSL, EXTERNAL:SSL |
|  | log.retention.bytes |  |
|  | log.retention.hours | The minimum age of a log file to be eligible for deletion due to age. |
|  | cleanup.policy | a) This config designates the retention policy to use on log segments.  b) The “delete” policy will discard old segments when their retention time or size limit has been reached.  c) The “compact” policy will enable log compaction, which retains the latest value for each key.  d) It is also possible to specify both policies in a comma – separated list (e.g., “delete”, “compact”). In this case, old segments will be discarded per the retention time & size configuration, while retained segments will be compacted. |
|  |  |  |
|  |  |  |
|  |  |  |

5. How to get started with Kafka/Confluent kafka in system (Refer User Manual)

(Here we’re using Linux Ubuntu wsl)

Step 1: Download Java & then Kafka from official store. Kafka 3.x.x compatible with Java 11+

Step 2: Extract & put it in /opt/kafka folder

Step 3:

**Option 1:** We can start zookeeper & then kafka directly from /opt/kafka using below commands

|  |
| --- |
| bin/zookeeper-server-start.sh config/zookeeper.properties (To start zookeeper) |
| bin/kafka-server-start.sh config/server.properties (To start kafka) |

**Option 2:** We can start Kafka & Zookeeper server in the background by writing shell script

Shell Script: **kafkastart.sh**

|  |
| --- |
| #!/bin/bash  sudo nohup /opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties > /dev/null 2>&1 &  sleep 5  sudo nohup /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties > /dev/null 2>&1 & |

Shell Script: **kafkastop.sh**

|  |
| --- |
| #!/bin/bash  sudo nohup /opt/kafka/bin/zookeeper-server-stop.sh -daemon /opt/kafka/config/zookeeper.properties > /dev/null 2>&1 &  sleep 5  sudo nohup /opt/kafka/bin/kafka-server-stop.sh -daemon /opt/kafka/config/server.properties > /dev/null 2>&1 & |

* Giving executable permissions to the file
* **sudo** chmod +x kafkastart.sh
* **sudo** chmod +x kafkastop.sh
* Go to folder containing kafkastart.sh, kafkastop.sh
* To run kafkastart.sh – **./kafkastart.sh**
* To run kafkastop.sh – **./kafkastop.sh**

**Option 3:** Create the systemd unit file for **zookeeper**: sudo nano /etc/systemd/system/**zookeeper.service**

|  |
| --- |
| /etc/systemd/system/zookeeper.service [Unit] Description=Apache Zookeeper service Documentation=http://zookeeper.apache.org Requires=network.target remote-fs.target After=network.target remote-fs.target  [Service] Type=simple ExecStart=/opt/kafka/bin/zookeeper-server-start.sh /opt/kafka/config/zookeeper.properties ExecStop=/opt/kafka/bin/zookeeper-server-stop.sh Restart=on-abnormal  [Install] WantedBy=multi-user.target |

* **Reload the daemon to take effect:** sudo **systemctl** daemon-reload
* Create the systemd unit file for Kafka service: sudo nano /etc/systemd/system/**kafka.service**

|  |
| --- |
| [Unit]  Description=Apache Kafka Service Documentation=http://kafka.apache.org/documentation.html  Requires=zookeeper.service  [Service]  Type=simple  Environment="JAVA\_HOME=/usr/lib/jvm/java-11-openjdk-amd64" ExecStart=/opt/kafka/bin/kafka-server-start.sh /opt/kafka/config/server.properties ExecStop=/opt/kafka/bin/kafka-server-stop.sh  [Install]  WantedBy=multi-user.target |

* **Reload the daemon to take effect:** sudo **systemctl** daemon-reload

To start zookeeper & kafka service & check its status (if you’re using systemd files)

* **Start zookeeper first**: sudo **systemctl** start zookeeper
* **Status of zookeeper**: sudo **systemctl** status zookeeper
* **Start the kafka service**: sudo **systemctl** start kafka
* **Status of Kafka service**: sudo **systemctl** status kafka

**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:

|  |  |  |
| --- | --- | --- |
| No. | Configuration | Description |
| 1. | bootstrap.servers | List of kafka servers used to bootstrap connections to kafka. |
| 2. | 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 |
| 3. | 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 |
| 4. | 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. |
| 5. | 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.

|  |  |  |
| --- | --- | --- |
| No. | Configuration | Description |
| 1. | 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. |
| 2. | 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. |
| 3. | 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. |
| 4. | 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:

|  |  |  |
| --- | --- | --- |
| **No.** | **Configuration** | **Description** |
| 1. | Name | Unique name for the connector. Attempting to register again with the same name will fail. |
| 2. | connector.class | The java class for the connector |
| 3. | 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. |
| 4. | key.converter | (optional) override the default key converter set by the worker |
| 5. | value.converter | (optional) override the default value converter set by the worker |
| 6. | Topics | (Mandatory for Sink Connector) A comma separated list of topics to use as input for this connector |
| 7. | 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

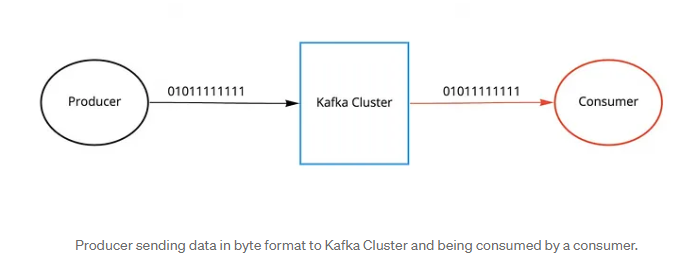
* By default, if no listeners are specified, the REST server runs on port 8083 using the HTTP protocol.
* When using HTTPS, the configuration has to include the SSL configuration. By default, it will use the ssl.\* settings.

|  |  |  |
| --- | --- | --- |
| **No.** | **Configuration** | **Description** |
| 1. | Name | Unique name for the connector. Attempting to register again with the same name will fail. |
| 2. | connector.class | The java class for the connector |
| 3. | 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. |
| 4. | key.converter | (optional) override the default key converter set by the worker |
| 5. | value.converter | (optional) override the default value converter set by the worker |
| 6. | Topics | (Mandatory for Sink Connector) A comma separated list of topics to use as input for this connector |
| 7. | topics.regex | A java regular expression of topics to use as input for this connector. |

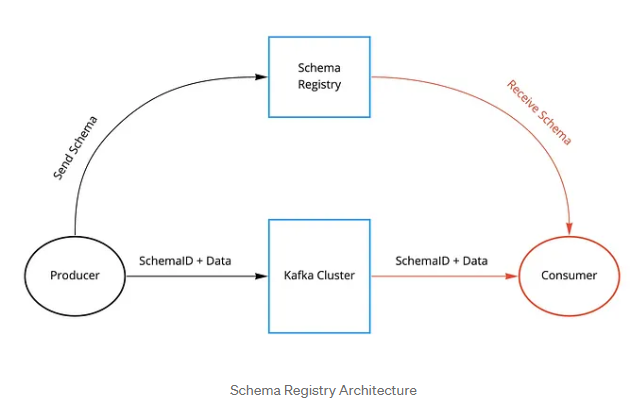
|  |  |  |
| --- | --- | --- |
| **No.** | **URIs** | **Description** |
| 1. | GET /connectors | Returns a list of active connectors |
| 2. | POST /connectors | Create a new connector; the request body should be a JSON object containing a string name filed & an object config field with the connector configuration parameters. |
| 3. |  |  |
| 4. |  |  |
| 5. |  |  |
| 6. |  |  |
| 7. |  |  |

**Q. 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.



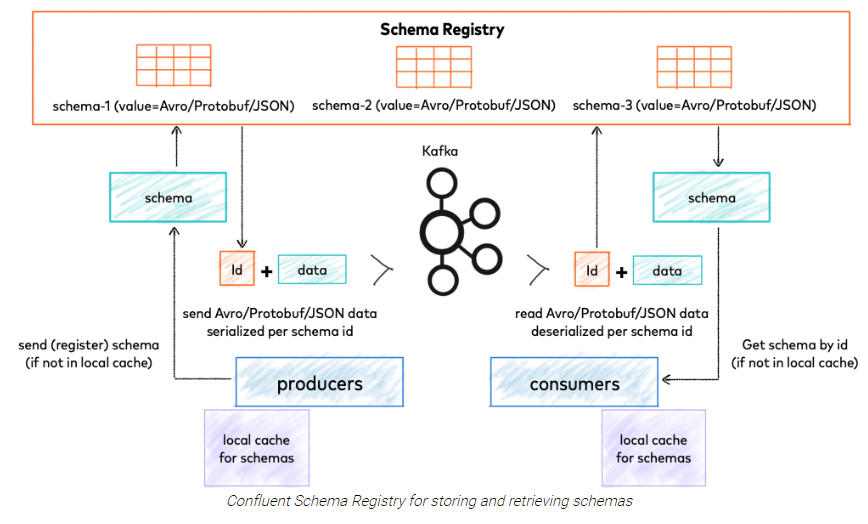
* Due to the decoupled nature of Kafka, Producers & Consumers don’t communicate with each other directly, but rather information transfer happens via Kafka topic. At the same time, the consumer still needs to know the type of data the producer is sending in order to deserialize it.
* Image if the producer 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.

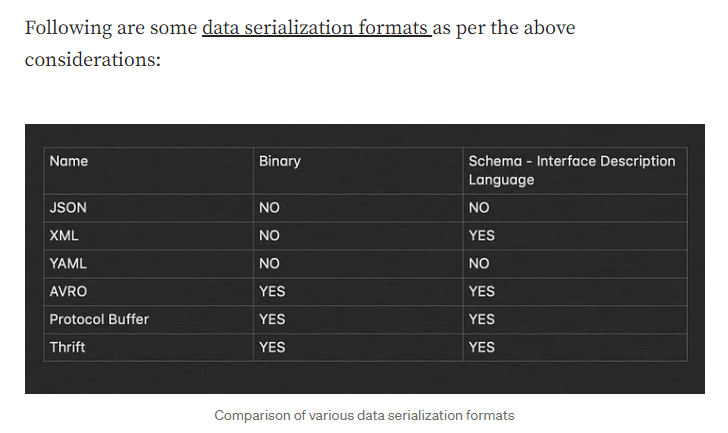


* With the schema registry in place, the producer before sending the data to Kafka, talks to the schema registry first & checks if the schema is available.
* If it doesn’t find the schema, then it registers & caches it in the schema registry. Once the producer gets the schema, it will serialize the data 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.





Q. When does QueueFullException occur in the producer?

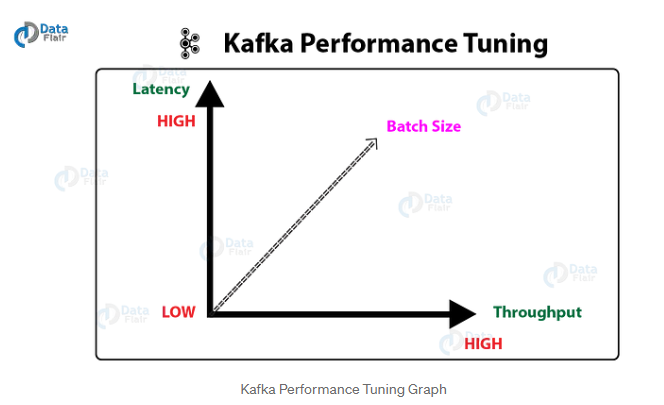
* QueueFullException occurs when the producer attempts to send message at a pace not handleable by the broker.

**Q. 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.



**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.