**Retrieving Data from Kafka - Key Points**

* Most Kafka Consumers will have a “poll” loop which loops infinitely and ingests data from Kafka
* Here is a sample poll loop:



* It is possible to use either [**poll**](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html?highlight=serializer#confluent_kafka.Consumer.poll) or [**consume**](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html?highlight=serializer#confluent_kafka.Consumer.consume), but poll is slightly more feature rich
* Make sure to call [**close()**](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html?highlight=serializer#confluent_kafka.Consumer.close) on your consumer before exiting and to consume any remaining messages
  + Failure to call close means the Kafka Broker has to recognize that the consumer has left the consumer group, which takes time and failed messages. Try to avoid this if you can.

**Lesson Overview**

In this lesson we will discuss:

1. Kafka’s Operational Architecture
2. How Kafka Stores Data
3. Kafka High-Availability and Data-loss prevention with Data replication
4. Data retention policies
5. A deep understanding of these concepts will provide a strong foundation for building stream processing applications.

## Kafka Architecture

* Kafka servers are referred to as brokers
* All of the brokers that work together are referred to as a cluster
* Clusters may consist of just one broker, or thousands of brokers
* [Apache Zookeeper](https://zookeeper.apache.org/) is used by Kafka brokers to determine which broker is the leader of a given partition and topic
* Zookeeper keeps track of which brokers are part of the Kafka cluster
* Zookeeper stores configuration for topics and permissions (Access Control Lists - ACLs)
  + ACLs are Permissions associated with an object. In Kafka, this typically refers to a user’s permissions with respect to production and consumption, and/or the topics themselves.
* Kafka nodes may gracefully join and leave the cluster
* Kafka runs on the Java Virtual Machine (JVM)

**Kafka Clustering - Key Points**

* Kafka servers are referred to as *brokers* and organized into *clusters*.
* Kafka uses Apache Zookeeper to keep track of topic and ACL configuration, as well as determine leadership and cluster management.
* Usage of ZooKeeper means that Kafka brokers can typically seamlessly join and leave clusters, allowing Kafka to grow easily as its usage increases or decreases.

**How Kafka Works - Summary**

In this section we learned:

* A Kafka Broker is an individual Kafka server
* A Kafka Cluster is a group of Kafka Brokers
* Kafka uses Zookeeper to elect topic leaders and store its own configuration
* Kafka writes log files to disk on the Kafka brokers themselves
* How Kafka achieves scale and parallelism with **topic partitions**
* How Kafka provides resiliency and helps prevent data loss with **data replication**

**How Kafka Works - Further Research**

You might be interested in optional reading about some of these related topics, which are beyond the scope of this course:

* [**Why does Kafka need Zookeeper?**](https://www.cloudkarafka.com/blog/2018-07-04-cloudkarafka_what_is_zookeeper.html)
* [**Kafka Design**](https://kafka.apache.org/documentation/#design)
* [**Partitioning**](https://kafka.apache.org/documentation/#intro_topics)
* [**Data Replication**](https://kafka.apache.org/documentation/#replication)

**Partitioning Topics Tips and Equation**

* The “right” number of partitions is highly dependent on the scenario.
* The most important number to understand is desired throughput. How many MB/s do you need to achieve to hit your goal?
* You can easily add partitions at a later date by modifying a topic.
* Partitions have performance consequences. They require additional networking latency and potential rebalances, leading to unavailability.
* Determine the number of partitions you need by dividing the overall throughput you want by the throughput per single consumer partition or the throughput per single producer partition. Pick the larger of these two numbers to determine the needed number of partitions.
  + # Partitions = Max(Overall Throughput/Producer Throughput, Overall Throughput/Consumer Throughput)
  + Example from video, with 3 Producers and 5 Consumers, each operating at 10MB/s per single producer/consumer partition: Max(100MBs/(3 \* 10MB/s), 100MBs/(5 \* 10MB/s)) = Max(2) ~= \*4 partitions needed\*

## Naming Conventions

* The only enforced rules for topic names are that they must be less than 256 characters, consist only of alphanumeric characters (a-z, A-Z, 0-9), “.”, “\_”, or “-”.
* There is no idiomatic or universally correct naming convention.
* Naming conventions can help reduce confusion, save time, and even increase reusability.
* Example of a naming convention:
* Consider starting with a namespace, like com.udacity
* Consider segmenting on schema or model type, like com.udacity.lesson, where lesson is the model
* Consider segmenting on event type, like com.udacity.lesson.quiz\_complete, where quiz\_complete is the event

**Data Management - Key Points**

* **Data retention** determines how long Kafka stores data in a topic.
  + [**The retention.bytes, retention.ms settings control retention policy**](https://kafka.apache.org/documentation.html#topicconfigs)
* When data *expires* it is deleted from the topic.
  + [**This is true if cleanup.policy is set to delete**](https://kafka.apache.org/documentation.html#topicconfigs)
* Retention policies may be time based. Once data reaches a certain age it is deleted.
  + [**The retention.ms setting controls retention policy on time**](https://kafka.apache.org/documentation.html#topicconfigs)
* Retention policies may be size based. Once a topic reaches a certain age the oldest data is deleted.
  + [**The retention.bytes setting controls retention policy on time**](https://kafka.apache.org/documentation.html#topicconfigs)
* Retention policies may be both time- and size-based. Once either condition is reached, the oldest data is deleted.
* Alternatively, topics can be **compacted** in which there is no size or time limit for data in the topic.
  + [**This is true if cleanup.policy is set to compact**](https://kafka.apache.org/documentation.html#topicconfigs)
* Compacted topics use the message key to identify messages uniquely. If a duplicate key is found, the latest value for that key is kept, and the old message is deleted.
* Kafka topics can use compression algorithms to store data. This can reduce network overhead and save space on brokers. Supported compression algorithms include: lz4, ztsd, snappy, and gzip.
  + [**compression.type controls the type of message compression for a topic**](https://kafka.apache.org/documentation.html#topicconfigs)
* Kafka topics should store data for one type of event, not multiple types of events. Keeping multiple event types in one topic will cause your topic to be hard to use for downstream consumers.

**In this section you learned:**

* What topic partitioning is and how it can help speed and scalability
* How Kafka replicates topic data for failure recovery
* How to configure data retention policies for Kafka topics
* How to name your Kafka topics
* How to compress data stored in your Kafka topic

**Optional Further Research in Kafka Topics**

* [**Kafka topic settings documentation**](https://kafka.apache.org/documentation.html#topicconfigs)
* [**Confluent blog post on Partitioning**](https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster)

**Kafka Producers**

In this section you will learn the specifics of how Kafka Producers are created and managed. Specifically, you will see how to:

* Synchronously and asynchronously send data to Kafka
* Use key configuration options, such as batch size, client identifiers, compression, and acknowledgements
* Specify data serializers

**Message Compression Types - Advantages and Disadvantages**

See this **[Cloudflare Blog Post](https://blog.cloudflare.com/squeezing-the-firehose/" \t "_blank)** for an excellent summary of compression type pros and cons.

Here is a quick survey of the compression types and their characteristics:

| **Algorithm** | **Pros** | **Cons** |
| --- | --- | --- |
| lz4 | fast compression and decompression | not a high compression ratio |
| snappy | fast compression and decompression | not a high compression ratio |
| zstd | high compression ratio | not as fast as lz4 or snappy |
| gzip | ubiquitous, widely-supported | cpu-intensive, significantly slower than lz4 or snappy |

**Producer Configuration Recap**

In this section we learned the following about Producer configuration:

* acks determine how many ISRs must acknowledge a message before the producer continues
* client.id is an optional, but useful, setting for debugging
* Producers support lz4, snappy, gzip, and zstd compression
  + lz4 and snappy are the fastest algorithms
  + zstd and gzip provide a higher compression ratio
* retries determine how many times to try to resend a message

## Kafka Producers - Summary

Kafka Producers are rich in options and configuration. In this section you’ve seen how to adapt your producer code to a wide-variety of real world situations through configuration.

Remember, no one set of settings works in all scenarios. If your producer application isn’t performing the way you expect, it’s worth revisiting your producer configuration to ensure that the settings make sense for the throughput level you are hoping to achieve.

### Optional Further Reading on Kafka Producers

* [**confluent-kafka-python/librdkafka Configuration Options**](https://github.com/edenhill/librdkafka/blob/master/CONFIGURATION.md)
* [**Apache Documentation on Producer Configuration**](https://kafka.apache.org/documentation/#producerconfigs)
* [**confluent-kafka-python Producer class**](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html?highlight=serializer#producer)

**Kafka Consumers - Key Points**

* client.id is an optional setting which is useful in debugging and resource limiting
* Poll for data to read data from Kafka
  + [**poll**](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html?highlight=serializer#confluent_kafka.Consumer.poll)
  + [**consume**](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html?highlight=serializer#confluent_kafka.Consumer.consume)

## Consumer Offsets - Key Points

* Kafka keeps track of what data a consumer has seen with offsets
  + Kafka stores offsets in a private internal topic
  + Most client libraries automatically send offsets to Kafka for you on a periodic basis
  + You may [**opt to commit offsets yourself**](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html#confluent_kafka.Consumer.commit), but it is not recommended unless there is a specific use-case.
  + Offsets may be sent synchronously or asynchronously
  + Committed offsets determine where the consumer will start up
    - If you want the consumer to start from the first known message, [set auto.offset.reset to earliest]
    - This will only work the first time you start your consumer. On subsequent restarts it will pick up wherever it left off
    - If you always want your consumer to start from the earliest known message, you must [**manually assign your consumer to the start of the topic on boot**](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html?highlight=serializer#confluent_kafka.Consumer.assign)

**Consumer Groups - Key Points**

* All Kafka Consumers belong to a Consumer group
  + The [**group.id parameter**](https://docs.confluent.io/current/installation/configuration/consumer-configs.html) is required and identifies the globally unique consumer group
  + Consumer groups consist of one or more consumers
* Consumer groups increase throughput and processing speed by allowing many consumers of topic data. However, only one consumer in the consumer group receives any given message.
* If your application needs to inspect every message in a topic, create a consumer group with a single member
* Adding or removing consumers causes Kafka to *rebalance*
  + During a rebalance, a broker *group coordinator* identifies a consumer *group leader*
  + The consumer *group leader* reassigns partitions to the current consumer group members
  + During a rebalance, messages may not be processed or consumed

**Topic Subscriptions - Key Points**

* You subscribe to a topic by specifying its name
  + If you wanted to subscribe to com.udacity.lesson.views, you would simply specify the full name as ”com.udacity.lesson.views”
  + Make sure to set allow.auto.create.topics to false so that the topic isn’t created by the consumer if it does not yet exist
* One consumer can subscribe to multiple topics by using a regular expression
  + The format for the regular expression is slightly different. If you wanted to subscribe to com.udacity.lesson.views.lesson1 and com.udacity.lesson.views.lesson2 you would specify the topic name as ”^com.udacity.lesson.views.\*”
  + The topic name must be prefixed with ”^” for the client to recognize that it is a regular expression, and *not* a specific topic name
  + Use **[regexp](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp" \t "_blank)** to specify your regular expressions.
  + See the **[confluent\_kafka\_python subscribe()](https://docs.confluent.io/current/clients/confluent-kafka-python/index.html?highlight=serializer" \l "confluent_kafka.Consumer.subscribe" \t "_blank)** documentation for more information

## Deserializers - Key Points

* Remember to deserialize the data you are receiving from Kafka in an appropriate format
  + If the producer used JSON, you will need to deserialize the data using a JSON library
  + If the producer used bytes or string data, you may not have to do anything
* Consumer groups increase fault tolerance and resiliency by automatically redistributing partition assignments if one or more members of the consumer group fail.

**Performance Considerations - Summary**

Monitoring Kafka Consumers, Producers, and Brokers for performance is an important part of using Kafka. There are many metrics by which to measure your Kafka cluster. Focus on these key metrics to get started:

* Consumer Lag: The difference between the latest offset in the topic and the most recently committed consumer offset
* Producer Response Rate: The rate at which the broker is responding to the producer indicating message status
* Producer Request Latency: The length of time a producer has to wait for a response from the broker after sending a message
* Broker Disk Space
* Broker Elections

**Further Research**

* [**DataDog blog post on monitoring Kafka**](https://www.datadoghq.com/blog/monitoring-kafka-performance-metrics)
* [**Confluent article on monitoring Kafka**](https://docs.confluent.io/current/kafka/monitoring.html)
* [**New Relic article on monitoring Kafka**](https://blog.newrelic.com/engineering/new-relic-kafkapocalypse/)

# Glossary of Terms: (same as provided at beginning of lesson)

* Broker (Kafka) - A single member server of the Kafka cluster
* Cluster (Kafka) - A group of one or more Kafka Brokers working together to satisfy Kafka production and consumption
* Node - A single computing instance. May be physical, as in a server in a datacenter, or virtual, as an instance might be in AWS, GCP, or Azure.
* Zookeeper - Used by Kafka Brokers to determine which broker is the leader of a given partition and topic, as well as track cluster membership and configuration for Kafka
* Access Control List (ACL) - Permissions associated with an object. In Kafka, this typically refers to a user’s permissions with respect to production and consumption, and/or the topics themselves.
* JVM - The Java Virtual Machine. Responsible for allowing host computers to execute the byte-code compiled against the JVM.
* Data Partition (Kafka) - Kafka topics consist of one or more partitions. A partition is a log which provides ordering guarantees for all of the data contained within it. Partitions are chosen by hashing key values.
* Data Replication (Kafka) - A mechanism by which data is written to more than one broker to ensure that if a single broker is lost, a replicated copy of the data is available.
* In-Sync Replica (ISR) - A broker which is up to date with the leader for a particular broker for all of the messages in the current topic. This number may be less than the replication factor for a topic.
* Rebalance - A process in which the current set of consumers changes (addition or removal of consumer). When this occurs, assignment of partitions to the various consumers in a consumer group must be changed.
* Data Expiration - A process in which data is removed from a Topic log, determined by data retention policies.
* Data Retention - Policies that determine how long data should be kept. Configured by time or size.
* Batch Size - The number of messages that are sent or received from Kafka
* acks - The number of broker acknowledgements that must be received from Kafka before a producer continues processing
* Synchronous Production - Producers which send a message and wait for a response before performing additional processing
* Asynchronous Production - Producers which send a message and do not wait for a response before performing additional processing
* Avro - A binary message serialization format
* Message Serialization - The process of transforming an applications internal data representation to a format suitable for interprocess communication over a protocol like TCP or HTTP.
* Message Deserialization - The process of transforming an incoming set of data from a form suitable for interprocess communication, into a data representation more suitable for the application receiving the data.
* Retries (Kafka Producer) - The number of times the underlying library will attempt to deliver data before moving on
* Consumer Offset - A value indicating the last seen processed of a given consumer, by ID.
* Consumer Group - A collection of one or more consumers, identified by group.id which collaborate to consume data from Kafka and share a consumer offset.
* Consumer Group Coordinator - The broker in charge of working with the Consumer Group Leader to initiate a rebalance
* Consumer Group Leader - The consumer in charge of working with the Group Coordinator to manage the consumer group
* Topic Subscription - Kafka consumers indicate to the Kafka Cluster that they would like to consume from one or more topics by specifying one or more topics that they wish to subscribe to.
* Consumer Lag - The difference between the offset of a consumer group and the latest message offset in Kafka itself
* CCPA - California Consumer Privacy Act
* GDPR - General Data Protection Regulation

**Lesson Recap**

In this lesson, you learned what a data schema is, and how they can help us better deal with change as it occurs within our data streams. Specifically we learned how Apache Avro works and how it integrates with tools like Schema Registry to provide producers and consumers information about the data they are consuming. Finally, we learned how to decide if a change is backward, forward, or full compatible.

**Glossary of Key Terms in Lesson (same as presented at beginning of lesson)**

* Data Schema - Define the shape of a particular kind of data. Specifically, data schemas define the expected fields, their names, and value types for those fields. Data schemas may also indicate whether fields are required or optional.
* Apache Avro - A data serialization framework which includes facilities for defining and communicating data schemas. Avro is widely used in the Kafka ecosystem and data engineering generally.
* Record (Avro) - A single encoded record in the defined Avro format
* Primitive Type (Avro) - In Avro, a primitive type is a type which requires no additional specification - null, boolean, int, long, float, double, bytes, string.
* Complex Type (Avro) - In Avro, a complex type models data structures which may involve nesting or other advanced functionality: records, enums, maps, arrays, unions, fixed.
* Schema Evolution - The process of modifying an existing schema with new, deleted, or modified fields.
* Schema Compatibility - Determines whether or not two given versions of a schema are usable by a given client
* Backward Compatibility - means that consumer code developed against the most recent version of an Avro Schema can use data using the prior version of a schema without modification.
* Forward Compatibility - means that consumer code developed against the previous version of an Avro Schema can consume data using the newest version of a schema without modification.
* Full Compatibility - means that consumers developed against the latest schema can consume data using the previous schema, and that consumers developed against the previous schema can consume data from the latest schema as well. In other words, full compatibility means that a schema change is both forward and backward compatible.
* None Compatibility - disables compatibility checking by Schema Registry.

**Lesson Recap**

In this lesson you learned how to use Kafka Connect to quickly integrate Kafka into a number of your existing data stores and workflows. We went hands-on with the JDBC and FileStream connectors and saw how to configure and deploy them. Next, we saw how REST Proxy can be used to bring Kafka to applications that can’t integrate native clients, but do have REST capabilities.

**Glossary of Key Terms in this Lesson: (same as provided in beginning of lesson)**

* Kafka Connect - A web server and framework for integrating Kafka with external data sources such as SQL databases, log files, and HTTP endpoints.
* JAR - **J**ava **AR**chive. Used to distribute Java code reusably in a library format under a single file.
* Connector - A JAR built on the Kafka Connect framework which integrates to an external system to either source or sink data from Kafka
* Source - A Kafka client putting data into Kafka from an external location, such as a data store
* Sink - A Kafka client remove data from Kafka into an external location, such as a data store
* JDBC - Java Database Connectivity. A Java programming abstraction over SQL database interactions.
* Task - Responsible for actually interacting with and moving data within a Kafka connector. One or more tasks make up a connector.
* Kafka REST Proxy - A web server providing APIs for producing and consuming from Kafka, as well as fetching cluster metadata.