CCDAK Introduction

Welcome to the Confluent Certified Developer for Apache Kafka (CCDAK) certification prep course!

We will cover the topics you will need to be aware of in order to earn your CCDAK certification.

Application Design

- Building a Practice Cluster
- Kafka Architecture Basics
- Kafka and Java
- Kafka Streams
- Advanced Application Design Concepts

CCDAK Introduction

Development

- Working with Kafka in Java
- Working with the Confluent Kafka REST APIs
- Confluent Schema Registry
- Kafka Connect

Deployment, Testing, and Monitoring

- Kafka Security
- Testing
- Working with Clients
- Confluent KSQL

Setting up Kafka Servers

To begin working with Kafka, we need to create a **Kafka cluster**. However, before we can construct a Kafka cluster, we must address the configuration for some servers!

Specifications:

- 3 Servers
- Distribution:

  Ubuntu 18.04 Bionic Beaver LTS
- Size: **1 Large, 2 Small**

Building a Kafka Cluster

The **Confluent Certified Developer for Apache Kafka (CCDAK)** exam covers the Confluent version of Apache Kafka.

Confluent is an enterprise platform built on Apache Kafka. Essentially, Confluent is Kafka with some additional enterprise features.

We will be using and installing **Confluent Community**, a free version of Confluent. It includes all of the features of Kafka and Confluent covered y CCDAK exam.

A note on sizing:

Kafka usually requires no more than **6 GB** of JVM heap space.

What is Apache Kafka?

The Kafka documentation describes Apache Kafka as a "*distributed streaming platform*."

This means that Kafka enables you to:

- Publish and Subscribe to streams of data records
- Store the records in a fault-tolerant and scalable fashion
- Process streams of records in real-time

Some background:

- Kafka is written in Java
- Originally created at LinkedIn
- Became open source in 2011

Use Cases

Messaging

“Building real-time streaming data pipelines that reliably get data between systems or applications”

Example:

An online store application. When a customer makes a purchase, the application sends the order data to a backend that handles fulfillment/shipping. You can use a messaging platform like Kafka to ensure that this data is passed to the backend application reliably and with no data loss, even if the backend application or one of your Kafka servers goes down.

Use Cases

Streaming

“Building real-time streaming applications that transform or react to the streams of data”

Example:

Log aggregation. You have multiple servers creating log data. You can feed the data to Kafka and use streams to transform the data into a standardized format, then combine the log data from all of the servers into a single feed for analysis.

Some Benefits of Kafka

- Strong reliability guarantees
- Fault tolerance
- Robust APIs

Kafka from the Command Line

Kafka includes a series of shell scripts that can be used to interact with the Kafka cluster from the command line.

These scripts can be found under the bin/ directory inside the main Kafka installation directory.

With a confluent package installation, the scripts are installed under /usr/bin and can be accessed as normal commands.

Kafka from the Command Line

Confluent also includes a command line tool, but it is recommended only for development purposes and not for a production environment.

If you have installed Confluent using the confluent packages, you can access this tool using the **confluent** command.

Topics

Topics are at the core of everything you can do in Kafka.

A topic is a data feed to which data records are published and from which they can be consumed.

Publishers send data to a topic, and subscribers read data from the topic. This is known as publisher/subscriber messaging, or simply pub/sub for short.

The Topic Log

Kafka topics each maintain a log.

The log is an ordered, immutable list of data records.

The log for a Kafka topic is usually divided into multiple partitions. This is what allows Kafka to process data efficiently and in a scalable fashion.

Each record in a partition has a sequential, unique ID called an offset.

Producers

A Producer is the publisher in the pub/sub model. The producer is simply an application that communicates with the cluster, and can be in a separate process or on a different server.

Producers write data records to the Kafka topic.

For each new record, the Producer determines which partition to write to, often in a simple round-robin fashion. You can customize this to use a more sophisticated algorithm for determining which topic to write a new record to.

Consumers

A Consumer is the subscriber in the pub/sub model. Like producers, consumers are external applications that can be in a separate process or on a different server from Kafka itself.

Consumers read data from Kafka topics. Each consumer controls the offset it is currently reading for each partition, and consumers normally read records in order based on the offset.

You can have any number of consumers for a topic, and they can all process the same records.

Records are not deleted when they are consumed. They are only deleted based upon a configurable retention period.

Consumer Groups

By default, all consumers will process all records, but what if you want to scale your record processing so that multiple instances can process the data without two instances processing the same record?

You can place consumers into Consumer Groups. Each record will be consumed by exactly one consumer per consumer group.

With consumer groups, Kafka dynamically assigns each partition to exactly one consumer in the groups. If you have more consumers than partitions, some of the consumers will be idle and will not process records.

Terms

Topic: A named data feed that data can be written to and read from.

Log: The data structure used to store a topic's data. The log is a partitioned, immutable sequence of data records.

Partition: A section of a topic's log.

Offset: The sequential and unique ID of a data record within a partition.

Producer: Something that writes data to a topic.

Consumer: Something that reads data from a topic.

Consumer group: A group of multiple consumers. Normally, multiple consumers can all consume the same record from a topic, but only one consumer in a consumer group will consume reach record.

Brokers

The central component of Kafka architecture is the broker.

Brokers are the servers that make up a Kafka cluster (one or more brokers).

Producers and consumers communicate with brokers in order to publish and consume messages.

Zookeeper

Kafka depends on an underlying technology called Zookeeper.

Zookeeper is a generalized cluster management tool. It manages the cluster and provides a consistent, distributed place to store cluster configuration.

Zookeeper coordinates communication throughout the cluster, adds and removes brokers, and monitors the status of nodes in the cluster. It is often installed alongside Kafka, but can be maintained on a completely separate set of servers.

Networking

Kafka uses a simple TCP protocol to handle messaging communication.

The Controller

In a Kafka cluster, one broker is dynamically designated as the Controller. The controller coordinates the process of assigning partitions and data replicas to nodes in the cluster.

Every cluster has exactly one controller. If the controller goes down, another node will automatically become the controller.

Replication

Kafka is designed with fault tolerance in mind. As a result, it includes built-in support for replication.

Replication means storing multiple copies of any given piece of data.

In Kafka, every topic is given a configurable replication factor.

The replication factor is the number of replicas that will be kept on different brokers for each partition in the topic.

Leaders

In order to ensure that messages in a partition are kept in a consistent order across all replicas, Kafka chooses a **leader** for each partition.

The leader handles all reads and writes for the partition. The leader is dynamically selected and if the leader goes down, the cluster attempts to choose a new leader through a process called leader election.

In-Sync Replicas

Kafka maintains a list of In-Sync Replicas (ISR) for each partition.

ISRs are replicas that are up-to-date with the leader. If a leader dies, the new leader is elected from among the ISRs.

By default, if there are no remaining ISRs when a leader dies, Kafka waits until one becomes available. This means that producers will be on hold until a new leader can be elected.

You can turn on unclean leader election, allowing the cluster to elect a non-in-sync replica in this scenario.

The Life of a Message

- Producer publishes a message to a partition within a topic.
- The message is added to the partition on the leader.
- The message is copied to the replicas of that partition on other brokers.
- Consumers read the message and process it.
- When the retention period for the message is reached, the message is deleted.

The Kafka Java APIs

Kafka provides a series of Application Programming Interfaces (APIs). These make it easier to write applications that use Kafka.

Kafka maintains a set of client libraries for Java, although there are open-source projects providing similar support for a variety of other languages.

Include these client libraries in your application to easily interact with Kafka!

The Kafka Java APIs

- Producer API: Allows you to build producers that publish messages to Kafka.
- Consumer API: Allows you to build consumers that read Kafka messages.
- Streams API: Allows you to read from input topics, transform data, and output it to output topics.
- Connect API: Allows you to build custom connectors, which pull from or push to specific external systems.
- AdminClient API: Allows you to manage and inspect higher-level objects like topics and brokers.

What are Streams?

So far, we have discussed using Kafka for messaging (reliably passing data between two applications).

Kafka Streams allows us to build applications that process Kafka data in real-time with ease.

A Kafka Streams application is an application where both the input and the output are stored in Kafka topics.

Kafka Streams is a client library (API) that makes it easy to build these applications.

Kafka Streams Transformations

Kafka Streams provides a robust set of tools for processing and transforming data. The Kafka cluster itself serves as the backend for data management and storage.

There are two types of data transformations in Kafka Streams:

- Stateless transformations do not require any additional storage to manage the state.
- Stateful transformations require a state store to manage the state.

Stateless Transformations

Branch

Splits a stream into multiple streams based on a predicate.

Filter

Removes messages from the stream based on a condition.

FlatMap

Takes input records and turns them into a different set of records.

Foreach

Performs an arbitrary stateless operation on each record. This is a terminal operation and stops further processing.

Stateless Transformations

GroupBy/GroupByKey

Groups records by their key. This is required to perform stateful transformations.

Map

Allows you to read a record and produce a new, modified record.

Merge

Merges two streams into one stream.

Peek

Similar to `Foreach`, but does not stop processing.

Kafka Streams Aggregations

Stateless transformations, such as `groupByKey` and `groupBy` can be used to group records that share the same key.

Aggregations are stateful transformations that always operate on these groups of records sharing the same key.

Aggregations

Aggregate

Generates a new record from a calculation involving the grouped records.

Count

Counts the number of records for each grouped key.

Reduce

Combines the grouped records into a single record.

Kafka Streams Joins

Joins are used to combine streams into one new stream.

Co-Partitioning

When joining streams, the data must be co-partitioned:

- Same number of partitions for input topics.
- Same partitioning strategies for producers.

You can avoid the need for co-partitioning by using a GlobalKTable. **With GlobalKTables, all instances of your streams application will populate the local table with data from all partitions.**

Kafka Streams Joins

Inner Join

The new stream will contain only records that have a match in both joined streams.

Left Join

The new stream will contain all records from the first stream, but only matching records from the joined stream.

Outer Join

The new stream will contain all records from both streams.

Kafka Streams Windowing

Windows are similar to groups in that they deal with a set of records with the same key. However, windows further subdivide groups into “time buckets.”

Tumbling Time Windows

Windows are based on time periods that never overlap or have gaps between them.

Hopping Time Windows

Time-based, but can have overlaps or gaps between windows.

Kafka Streams Windowing

Sliding Time Windows

These windows are dynamically based on the timestamps of records rather than a fixed point in time. They are only used in joins.

Session Windows

Creates windows based on periods of activity. A group of records around the same timestamp will form a session window, whereas a period of "idle time" with no records in the group will not have a window.

Late-Arriving Records

In real-world scenarios, it is always possible to receive out-of-order data.

When records fall into a time window received after the end of that window's grace period, they become known as late-arriving records.

You can specify a retention period for a window. Kafka Streams will retain old window buckets during this period so that late-arriving records can still be processed.

Any records that arrive after the retention period has expired will not be processed.

Streams vs. Tables

Kafka Streams models data in two primary ways: streams and tables.

Streams: Each record is a self-contained piece of data in an unbounded set of data. New records do not replace an existing piece of data with a new value.

Tables: Records represent a current state that can be overwritten/ updated.

Streams vs. Tables

Here are some example use cases:

Stream:

- Credit card transactions in real time.
- A real-time log of attendees checking in to a conference.
- A log of customer purchases which represent the removal of items from a store’s inventory.

Table:

- A user’s current available credit card balance.
- A list of conference attendee names with a value indicating whether or not they have checked in.
- A set of data containing the quantity of each item in a store’s inventory.

Kafka Configuration

Kafka objects such as brokers, topics, producers, and consumers can all be customized using configuration.

Kafka uses the property file format for configuration. All configuration options are provided using key-value pairs, for example:

```
broker.id=1
```

With Kafka, you can configure the following:

- Brokers
- Topics
- Clients (Producers, Consumers, Streams Applications, etc.)

Broker Configuration

You can configure your Kafka broker using `server.properties`, the command line, or even programmatically using the AdminClient API.

Some broker configs can be changed dynamically (without a broker restart). Check the documentation to see which configs can be dynamically updated.

- read-only: These configs require a broker restart in order to be updated.
- per-broker: These can be dynamically updated for each individual broker.
- cluster-wide: These configs can be updated per-broker, but the cluster-wide default can also be dynamically updated.

Topic Configuration

Topics can be configured using the command line tools (i.e. `kafka-topics` or `kafka-configs`), as well as programmatically.

All topic configurations have a broker-wide default. The default values will be used unless an override is specified for a particular topic.

Use the `--config` argument with `kafka-topics` to override default configuration values when creating topics from the command line.

Client Configuration

All Kafka clients can be configured:

- Producers
- Consumers
- Kafka Streams Applications
- Kafka Connect Applications
- AdminClient Applications

Usually, these applications are configured programmatically in the client code.

Topic Design

When using Kafka to address a given use case, it is important to design your topics for maximum performance.

The two main considerations when designing topics are partitions and replication factor.

Some questions to consider when designing your topics:

- How many brokers do you have?

  The number of brokers limits the number of replicas.

- What is your need for fault tolerance?

  A higher replication factor means greater fault tolerance.

Topic Design

Some questions to consider when designing your topics:

- How many consumers do you want to place in a consumer group for parallel processing?

  You will need at least as many partitions as the number of consumers you expect to have on a single group.

- How much memory is available on each broker?

  Kafka requires memory to process messages. The configuration setting `replica.fetch.max.bytes` (default ~1 MB) determines the rough amount of memory you will need for each partition on a broker.

Metrics and Monitoring

Both Kafka and ZooKeeper offer metrics through JMX, a built-in Java technology for providing and accessing performance metric data.

You can access metrics from Kafka brokers as well as your Kafka client applications.

You can find a full list of available metrics in the Kafka documentation.

Java Producers

Producers write data to Kafka topics.

You can create your own producer in Java using the Producer API.

Let’s create a producer to:

- Count from 0 to 99 and publish the counts as records to a topic.
- Print the record metadata to the console using a callback.

You can configure a producer with acks=all. This will cause the producer to receive an acknowledgement for a record only after all in-sync replicas have acknowledged the record.

Java Consumers

Consumers read data from Kafka topics.

You can create your own consumer in Java using the Consumer API.

Let's create a consumer to:

- Consume data from two topics.
- Print the key and value to the console, along with some metadata such as the message partition and offset.

The Confluent REST Proxy

The Confluent REST Proxy provides a “RESTful interface” for Kafka. It is part of Confluent and does not come with a standard non-Confluent Kafka installation.

REpresentational State Transfer (REST)

An architectural style for creating web services that uses normal HTTP patterns as a standard for interacting with web resources.

The Confluent REST Proxy allows you to interact with Kafka using simple HTTP requests.

Producing Messages with REST

To produce messages with REST, send a POST request to:

```
http://localhost:8082/topics/<topic_name>
```

For the POST body, enter a JSON representation of the message(s) you want to publish:

```
{
  "records":[
    {
      "key": "<key>",
      "value": "<value>"
    },
    {
      "key": "<key>",
      "value": "<value>"
    }
  ]
}
```

Consuming Messages with REST

Consuming messages with REST requires a few steps.

First, create a consumer and consumer instance.

POST request to `http://localhost:8082/consumers/<consumer_name>`

```
{
  "name":"<consumer instance name>",
  "format":"json",
  "auto.offset.reset":"earliest"
}
```

Here `auto.offset.reset=earliest` will cause this consumer to consume from the beginning of the log.

Consuming Messages with REST

Subscribe the consumer instance to a topic.

POST request to
`http://localhost:8082/consumers/<consumer name>/instances/<consumer instance name>/subscription`

```
{
  "topics":[
    "<topic name>"
  ]
}
```

You can subscribe to any number of topics with a single request.

Consuming Messages with REST

Consume messages.

GET request to
`http://localhost:8082/consumers/<consumer name>/instances/<consumer instance name>/records`

This request will return all the records in the subscribed topic.

Consuming Messages with REST

Clean up by deleting the consumer.

DELETE request to

```
http://localhost:8082/consumers/<consumer name>/instances/<consumer instance name>
```

What is Confluent Schema Registry?

Confluent Schema Registry is a versioned, distributed storage for Apache Avro schemas.

These schemas define an expected format for your data and can be used to serialize and deserialize complex data formats when interacting with Kafka.

Avro schemas allow producers to specify a complex format for published data, and consumers can use the schema to interpret this data. Both communicate with the schema registry to store and retrieve these schemas.

Schemas can be applied to both keys and values in messages.

Creating an Avro Schema

Avro schemas are represented in JSON.

```
{
  "namespace":"<namespace>",
  "type": "record",
  "name": "<schema name>",
  "fields": [
    {
      "name": ”<field name>",
      "type": ”<field type>"
    }
  ]
}
```

Using Schema Registry with a Producer

To build a Kafka producer that uses Schema Registry, we need to perform the following steps:

1. Add the Avro plugin to the project.
2. Add the Confluent Maven repository to the project.
3. Add the Schema Registry and Avro dependencies to the project.
4. Implement a producer that is set up to use an Avro serializer.
5. Publish some records using the auto-generated Avro Java object.

Using Schema Registry with a Consumer

To build a Kafka consumer that uses Schema Registry, we need to perform the following steps:

1. Implement a consumer that is set up to use an Avro deserializer.
2. Consume the records and use them to populate the auto-generated Avro Java object.

Making Changes to an Avro Schema

When developing an application, you may sometimes need to make changes to your schema.

This creates a problem because there may be messages in your topic(s) that were serialized using the old schema. How will consumers be able to deserialize them once the schema has changed?

Confluent Schema Registry implements a schema compatibility checking mechanism to help you evolve your schemas over time.

Making Changes to an Avro Schema

Here's how schema compatibility checking works:

- Select a compatibility type for your schema.
- Schema Registry will prevent you from making changes that would break compatibility based on your chosen compatibility type.
- If a producer attempts to register an incompatible schema, an error will occur.

Making Changes to an Avro Schema

Compatibility types:

- `BACKWARD`: (Default) consumers using an updated schema can read data that was serialized via the current schema in the registry.
- `BACKWARD_TRANSITIVE`: A consumer using an updated schema can use data serialized via all previously registered schemas.
- `FORWARD`: Consumers using the current schema in the registry can read data serialized via the updated schema.
- `FORWARD_TRANSITIVE`: Consumers using any previous schema in the registry can read data serialized via the updated schema.

Making Changes to an Avro Schema

Compatibility types:

- `FULL`: The new schema is forward and backward compatible with the current schema in the registry.
- `FULL_TRANSITIVE`: The new schema is forward and backward compatible with all previous schemas in the registry.
- `NONE`: All compatibility checks are disabled.

What is Kafka Connect?

Kafka Connect is a tool for streaming data between Kafka and other systems.

Source Connectors import data from external systems into Kafka.

Sink Connectors export data from Kafka into external systems.

These connectors are designed to be scalable and reusable.

Connectors are available for a variety of external systems, and the Connect API allows you to build your own.

Using Kafka Connect

To use Kafka Connect, create connectors with the Kafka Connect REST API:

POST

```
https://localhost:8083/connectors
```

```
{
  "name": ”<connector name>",
  "config": {
    "connector.class": ”<class
name>",
    <other configuration options>
    …
  }
}
```

Using Kafka Connect

Some additional API calls:

View the connector’s configuration:

GET

```
https://localhost:8083/connectors/connector name>
```

Get the connector’s status, including error messages:

GET

```
https://localhost:8083/connectors/connector name>/status
```

Delete a connector:

DELETE

```
https://localhost:8083/connectors/connector name>
```

TLS Encryption

If you plan to have external clients (such as producers and consumers) connect to Kafka, it may be a good idea to confirm that they use TLS to do so.

Transport Layer Security (TLS) prevents man-in-the-middle (MITM) attacks, plus ensures that communication between clients and Kafka servers is encrypted.

To set up TLS, we will need to:

- Create a certificate authority.
- Create signed certificates for our Kafka brokers.
- Configure brokers to enable TLS and use the certificates.
- Configure a client to connect securely and trust the certificates.

Client Authentication

For additional security, you may want to use client authentication. There are multiple ways to authenticate with Kafka. Check the documentation for a full list.

One way to authenticate is to use mutual TLS with client certificates. To set this up, we can:

- Generate a client certificate.
- Sign it with the Certificate Authority.
- Enable SSL client authentication on the brokers.
- Configure a client to use the client certificate.

ACL Authorization

Once you have the ability to authenticate clients, you may consider exercising more granular control over what each client can do in your cluster.

Kafka uses Access Control Lists (ACLs) that let you control what users can do. You can allow or disallow granular operations such as reading and writing, plus do so against a variety of resources like topics, cluster configurations, etc.

ACL Authorization

An ACL contains the following components:

- Principal – This is the user.
- Allow/Deny
- Operation – What the user can do, for example, read or write.
- Host – The IP address(es) of hosts connecting to the cluster to perform this action.
- Resource Pattern – A pattern that matches one or more resources, such as topics.

You can manage ACLs using the `kafka-acls` command line tool.

Testing Producers

Unit Testing – Testing a piece of code in isolation. Usually, unit testing is performed by writing code that is designed to execute the code being tested and verify whether its outputs are correct.

If you want to write tests for your producer code, you will need to simulate the behavior of the producer itself in order to see how your code responds.

Luckily, the Producer API includes a class called MockProducer to help you accomplish this.

Testing Consumers

Just like with producers, you may want to create unit tests for your Kafka consumers.

You can simulate consumer behavior using the MockConsumer mock object.

Like the MockProducer, the MockConsumer can be injected into your consumer processing code and used to simulate the behavior of a consumer for testing purposes.

Testing Streams Applications

Kafka Streams applications can be more complex than producers and consumers, particularly as they often involve more asynchronous logic.

Testing Kafka Streams code can be challenging.

Luckily, Kafka provides the `kafka-streams-test-utils` library to make this process significantly easier.

Testing Streams Applications

Some of the features offered by `kafka-streams-test-utils` include:

`TopologyTestDriver` – Allows you to feed test records in, simulates your topology, and returns output records.

`ConsumerRecordFactory` – Helps you convert consumer record data into byte arrays that can be processed by the `TopologyTestDriver`.

`OutputVerifier` – Provides helper methods for verifying output records in your tests.

Monitoring Clients

You can monitor Kafka clients using JMX just like brokers.

- Ensure the proper JVM arguments are set when running your client.
- Connect to the client JVM with a JMX client, for example, JConsole.

This applies to clients such as the console producer and consumer, as well as your own custom Java clients.

Producer Metrics

Some important producer metrics:

- `response-rate` (global and per broker): Responses (acks) received per second. Sudden changes in this value could signal a problem, though what the problem could be depends on your configuration.
- `request-rate` (global and per broker): Average requests sent per second. Requests can contain multiple records, so this is not the number of records. It does give you part of the overall throughput picture.
- `request-latency-avg` (per broker): Average request latency in `ms`. High latency could be a sign of performance issues, or just large batches.

Producer Metrics

Some important producer metrics:

- `outgoing-byte-rate` (global and per broker): Bytes sent per second. Good picture of your network throughput. Helps with network planning.
- `io-wait-time-ns-avg` (global only): Average time spent waiting for a socket ready for reads/writes in nanoseconds. High wait times might mean your producers are producing more data than the cluster can accept and process.

Consumer Metrics

Some important consumer metrics:

- `records-lag-max`: Maximum record lag. How far the consumer is behind producers. In a situation where real-time processing is important, high lag might mean you need more consumers.
- `bytes-consumed-rate`: Rate of bytes consumed per second. Gives a good idea of throughput.
- `records-consumed-rate`: Rate of records consumed per second.
- `fetch-rate`: Fetch requests per second. If this falls suddenly or goes to zero, it may be an indication of problems with the consumer.

Producer Tuning

You can tune your producers by changing their configurations to suit your needs.

Some important configurations:

- `acks`: Determines when the broker will acknowledge the record.
  - `0`: Producer will not wait for acknowledgement from the server. The record is considered acknowledged as soon as it is sent. The producer will not receive certain metadata such as the record offset.
  - `1`: Record will be acknowledged when the leader writes the record to disk. Note that this creates a single point of failure. If the leader fails before followers replicate the record, data loss will occur.
  - `all / -1`: Record will be acknowledged only when the leader and all replicas have written the record to their disks. The acks may take longer, but this provides the maximum data integrity guarantee.

Producer Tuning

Some important configurations:

- `retries`: Number of times to retry a record if there is a transient error. If `max.in.flight.requests.per.connection` is not set to 1, the retry could cause records to appear in a different order.
- `batch.size`: Producers batch records sharing the same partition into a single request to create fewer requests. This specifies the maximum number of bytes in a batch. Messages larger than this size will not be batched. Requests can contain multiple batches (one for each partition) if data is going to more than one partition.

Consumer Tuning

You can tune your consumers by changing their configurations to suit your needs.

Some important configurations:

- `fetch.min.bytes`: The minimum amount of data to fetch in a request. If there is not enough data to satisfy this requirement, the request will wait for more data before responding. Set this higher to get better throughput in some situations at the cost of some latency.
- `heartbeat.interval.ms`: How often to send heartbeats to the consumer coordinator. Set this lower to allow a quicker rebalance response when a consumer joins or leaves the consumer group.

Producer Tuning

Some important configurations:

- `auto.offset.reset`: What to do when the consumer has no initial offset.
  - `Latest`: Start at the latest record.
  - `Earliest`: Start with the earliest record.
  - `none`: Throw an exception when there is no existing offset data.
- `enable.auto.commit`: Periodically commit the current offset in the background. Use this to determine whether you want to handle offsets manually or automatically.

What is KSQL?

Confluent KSQL is a streaming SQL engine for Kafka.

KSQL is very similar to Kafka Streams. It allows you to perform stream processing operations using SQL-like queries.

Although not compliant with ANSI SQL, its syntax is inspired by and similar to ANSI SQL.

What is KSQL?

With KSQL, you can do almost anything that you can do with Kafka Streams, such as:

- Data transformations
- Aggregations
- Joins
- Windowing
- Modeling data with streams and tables

KSQL Architecture

KSQL runs as a separate service from Kafka. You can run multiple KSQL servers in a KSQL cluster.

Clients make requests to the KSQL server, and the server communicates with the Kafka cluster.

Confluent comes with a command-line KSQL client that you can access using the `ksql` command.

Using KSQL

KSQL queries look similar to SQL queries, but they query data from Kafka streams and tables.

List topics:

```
SHOW TOPICS;
```

Print records in a topic:

```
PRINT ‘<topic name>’;
```

Create a stream:

```
CREATE STREAM <name> (<fields>)
WITH (kafka_topic=‘<topic>’,
value_format=’<format>’);
```

Create a table:

```
CREATE TABLE <name> (<fields>) WITH
(kafka_topic=‘<topic>’,
value_format=’<format>’,
key=‘<key>’);
```

Using KSQL

Select from a table or stream:

```
SELECT <fields> FROM <table or
stream>;
```

Aggregate data:

```
SELECT sum(<field>) FROM <table or
stream> GROUP BY <field>;
```

Preparing for the Exam

Confluent Certified Developer for Apache Kafka Exam

- 55 questions
- Multiple choice
- 90 minutes
- No documentation

Study Tips

- Use the study guide
- Take the practice exam
- Use flash cards for memorization
- Repeat hands-on labs
- Scan the official documentation

What's Next?

Here are some other Linux Academy big data courses to check out:

- Apache Kafka Deep Dive
- Big Data Essentials
- Big Data Fundamentals
- Splunk Deep Dive
- Hadoop Quick Start
- Elastic Stack Essentials
- Elasticsearch Deep Dive
- Linux Academy's Elastic Certification Preparation Course