APACHE KAFKA

Confluent Certified Developer for Apache Kafka [CCDAK]

GitHub Notes

Table of Contents

CHAPTER 1 : KAFKA BROKER

Kafka Broker Overview

A Kafka Broker is a term for a server in a Kafka cluster that hosts topics and processes clients' read and write requests. Brokers are responsible for preserving published data for a certain period or size. The Kafka broker makes the system scalable and fault-tolerant.

Points to Remember for CCDAK on Brokers

- **Scalability**: Brokers allow Kafka to scale out by adding more machines without downtime.
- **Fault Tolerance**: Replication across brokers ensures high availability and fault tolerance.
- **Configuration Management**: Broker configurations play a crucial role in Kafka performance and reliability.
- **Topic Management**: Brokers handle topic creation, partitioning, and replication.

Important Broker Properties

`default.replication.factor`

- **Default**: 1
- **Description**: Default number of replicas for a topic. Increasing this can enhance fault tolerance.
- **Trade-offs**: Higher replication can lead to increased latency and resource usage.

`num.partitions`

- **Default**: 1
- **Description**: Default number of partitions per topic. More partitions allow higher parallelism.
- **Trade-offs**: Too many partitions can increase overhead on broker and Zookeeper performance.

`min.insync.replicas`

- **Default**: 1
- **Description**: Minimum number of replicas that must acknowledge a write for it to be considered successful.
- **Trade-offs**: A higher value increases data durability but may impact write latency.

`unclean.leader.election.enable`

- **Default**: true
- **Description**: Allows a non-synced replica to become leader. Turning it off ensures data consistency.
- **Trade-offs**: Disabling may increase availability but risks data loss.

`num.replication.fetchers`

- **Default**: 1
- **Description**: Number of fetcher threads used to replicate messages.
- **Trade-offs**: Increasing can improve replication throughput at the cost of higher CPU usage.

`replica.fetch.max.bytes`

- **Default**: 1048576 (1MB)
- **Description**: Maximum bytes a replica can fetch in a single request. Higher values can improve replication throughput.
- **Trade-offs**: Larger fetch sizes may increase memory usage.

`advertised.listeners`

- **Default**: Depends on setup
- **Description**: Hostname and port the broker will advertise to producers and consumers.
- **Trade-offs**: Must be set correctly for clients to connect.

`logs.dirs`

- **Default**: /tmp/kafka-logs
- **Description**: Directories where the log data is stored.
- **Trade-offs**: Adequate disk space and IO performance are critical.

`auto.create.topic.enable`

- **Default**: true
- **Description**: Allows automatic topic creation on the server.
- **Trade-offs**: Convenient but may lead to unintentional topic creation.

`log.retention.hours`, `log.retention.bytes`, `log.segment.bytes`

- **Defaults**: 168 hours (7 days), -1 (unlimited), 1073741824 (1GB)
- **Description**: Control the retention policy for logs by time, size, and segment size.
- **Trade-offs**: Balancing disk usage with data availability needs.

`log.retention.check.interval.ms`

- **Default**: 300000 (5 minutes)
- **Description**: Frequency at which the log cleaner checks for logs to clean.
- **Trade-offs**: More frequent checks can slightly increase CPU usage but help in timely log cleanup.

Partition Management

- **Basics**: A Kafka topic consists of one or more partitions. This allows the data of a topic to be distributed across multiple brokers.
- **Replication Factor**: Each partition can have multiple replicas, with one being the leader. The replication factor determines the total number of these replicas.
- **Immutability and Ordering**: Once data is written to a partition, it is immutable. Kafka guarantees order within a partition but not across partitions.

Segment Management

- **Partition Segments**: Partitions are subdivided into segments, which are essentially log files where Kafka messages are stored.
- **Segment Size and Duration**: Configurable settings such as `log.segment.bytes` and `log.segment.ms` control the size and time before a new segment is rolled out.
- **Indexes for Efficiency**: Each segment comes with index files to efficiently locate messages either by offset or timestamp.

Log Retention and Cleanup

- **Policies**: Log segments are eligible for cleanup based on size or time, whichever is reached first.
- **Cleanup Types**: Kafka supports deletion or compaction as log cleanup policies. Deletion simply removes old segments, while compaction retains at least the latest value for each key.

Topic Replication

- **Leader and ISR**: For each partition, one replica is the leader that handles all read and write requests, while the others are in-sync replicas (ISR).
- **Replica Management**: Kafka manages replicas to ensure data is not lost and is accessible even if some brokers are down. The replication factor should not exceed the number of brokers in the cluster to ensure each partition has a unique set of brokers.

Producer Considerations

- **Reliability**: Producers can specify `acks` to control the number of replicas that must acknowledge a write for it to be considered successful, balancing between reliability and performance.
- **Message Size**: The `message.max.bytes` setting controls the maximum size of a message that can be sent. Large messages can impact broker performance and stability.
- **Message Keys**: Producers can send messages with a key to ensure messages with the same key are sent to the same partition, enabling ordering by key within a partition.

Consumer Considerations

- **Consumer Groups and Offset Management**: Consumers track their progress via offsets within each partition. Kafka stores these offsets in a special topic named `__consumer_offsets`.
- **Partition Assignment**: Consumers in a group are automatically assigned partitions by Kafka. This can be overridden with manual partition assignment if needed.

Broker Performance Tuning

- **Thread Management**: Configuring the number of I/O and network threads (`num.io.threads`, `num.network.threads`) can significantly affect broker performance, especially in high-throughput environments.
- **Log Flush Management**: Settings like `log.flush.interval.messages` and `log.flush.interval.ms` control the frequency of log flushes to disk, impacting durability vs performance.

Security Configurations

- **Encryption and Authentication**: Brokers can be configured to use SSL/TLS for encrypting client connections and SASL for client authentication, ensuring secure data transmission.
- **Authorization**: Kafka supports ACLs for authorizing client requests, allowing fine-grained control over who can read or write to topics.

Monitoring and Management

- **JMX Metrics**: Kafka exposes a wide range of metrics via JMX, which can be used to monitor broker health, performance, and resource usage.
- **Log Clean-Up**: Monitoring log segment sizes and cleanup policies is crucial to avoid running out of disk space. Adjusting `log.retention.hours`, `log.retention.bytes`, and related settings helps manage disk usage.
-

Broker Metric	**Explanation**
ACTIVE CONTROLLER COUNT	The broker with an Active Controller Count of 1 is the current controller of the Kafka cluster. The controller is responsible for managing the state of partitions and replicas, and electing leaders.
REQUEST HANDLER IDLE RATIO	This metric indicates the percentage of time the broker's request handlers are idle. A low idle ratio suggests the broker is under heavy load and may need additional resources or optimization.
ALL TOPICS BYTES IN	The All Topics Bytes In metric measures the incoming byte rate for all topics on the broker. High values indicate a need to scale out the Kafka cluster by adding more brokers.
ALL TOPICS BYTES OUT	This metric measures the outgoing byte rate for all topics on the broker, which reflects the consumer traffic. High consumer traffic may require increasing consumer instances or optimizing consumer configurations.

Broker Metric	Explanation
ALL TOPICS MESSAGES IN	The All Topics Messages In metric measures the incoming message rate for all topics on the broker.
PARTITION COUNT	This metric indicates the number of partitions assigned to a broker. Evenly distributing partitions across brokers is important for maintaining a balanced cluster.
LEADER COUNT	The Leader Count metric shows the number of partitions for which the broker is the leader. Leaders handle all read and write requests for partitions.
OFFLINE PARTITIONS	Offline partitions are partitions without an active leader. A high number of offline partitions can indicate issues with broker health or network connectivity.
REQUEST METRICS	Request metrics, such as request rate and request latency, provide insights into the performance of the broker and can help identify potential bottlenecks.

High Availability Considerations

- **Zookeeper Dependency**: Kafka brokers rely on Zookeeper for cluster metadata and coordination. Ensuring Zookeeper cluster's health is critical for Kafka's reliability.
- **Broker Failures**: Kafka's replication mechanism ensures that as long as a sufficient number of replicas are alive, brokers can fail without losing data. Properly configuring `min.insync.replicas` and `replication.factor` is key.

Zero Copy

Most traditional data systems use RAM for data storage due to its low latencies.

Kafka avoids using RAM for this purpose, achieving low latency message delivery through the Zero Copy Principle. This optimization eliminates intermediate data copies, transferring data directly from the read buffer to the socket buffer.

However, enabling SSL for data encryption in transit negates zero copy optimization. With SSL, the broker must decrypt and re-encrypt data, thus reintroducing intermediate data handling.

CHAPTER 2 : KAFKA CLI

Kafka CLI Tools Overview

Kafka provides a set of command-line tools that allow administrators and developers to interact with the Kafka cluster, topics, consumer groups, and other Kafka entities. These tools are fundamental for tasks such as creating topics, producing and consuming messages, and monitoring consumer group offsets.

Key CLI Tools

- **kafka-topics.sh**: Used for creating, deleting, describing, or altering topics on a Kafka broker.
- `# Create a topic`
- `kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic my-topic`
- **kafka-console-producer.sh**: A utility that allows you to produce messages to a Kafka topic from the command line.
- `# Produce a message`
- `kafka-console-producer.sh --broker-list localhost:9092 --topic my-topic`
- **kafka-console-consumer.sh**: Enables consuming messages from a Kafka topic and display them to standard output.
- `# Consume messages`
- `kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my-topic --from-beginning`
- **kafka-consumer-groups.sh**: Offers a way to manage consumer groups, including listing groups, describing group details, and resetting consumer group offsets.
- `# Describe consumer group`
- `kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --group my-group`
- **kafka-acls.sh**: Used for managing Access Control Lists (ACLs) for Kafka.
- **kafka-configs.sh**: Allows for modifying broker, topic, and client configs.
- **kafka-mirror-maker.sh**: A tool for mirroring data between Kafka clusters.

Command	**Explanation**
kafka-topics.sh	A tool for managing Kafka topics.
--bootstrap-server	Specifies a comma-separated list of Kafka broker addresses. This is the recommended option, replacing `--zookeeper`.
--create	Creates a new topic.
--delete	Deletes an existing topic.
--alter	Alters the configuration of an existing topic.
--list	Lists all available topics.
--describe	Describes the details of a specific topic, including partitions and replication factor.
--if-exists	Used with `--delete` or `--alter` to perform the operation only if the topic exists.
--if-not-exists	Used with `--create` to create the topic only if it does not already exist.

Command	**Explanation**
--topic	Specifies the name of the topic to create, modify, or delete.
--partitions	Specifies the number of partitions for the topic.
--replication-factor	Specifies the replication factor for each partition in the topic.
--config	Specifies topic-level configuration properties.
kafka-console-producer.sh	A command-line tool that allows you to produce messages to a Kafka topic.
--bootstrap-server	Specifies a comma-separated list of Kafka broker addresses.
--topic	Specifies the name of the topic to produce messages to.
--producer-property	Specifies a producer configuration property.
kafka-console-consumer.sh	A command-line tool that allows you to consume messages from a Kafka topic.
--bootstrap-server	Specifies a comma-separated list of Kafka broker addresses.
--topic	Specifies the name of the topic to consume messages from.
--from-beginning	Starts consuming messages from the beginning of the topic.
--group	Specifies the consumer group ID for the consumer.
--consumer-property	Specifies a consumer configuration property.
kafka-consumer-groups.sh	A tool for managing Kafka consumer groups.
--bootstrap-server	Specifies a comma-separated list of Kafka broker addresses. This is a required flag.
--group	Specifies the ID of the consumer group to describe or manage. This is a required flag for most operations.
--describe	Describes consumer group details, including the current offset and lag for each partition.
--list	Lists all consumer groups.
--reset-offsets	Resets the offset for a consumer group.
--to-earliest	Used with `--reset-offsets` to reset the offset to the earliest message in each partition.
--to-latest	Used with `--reset-offsets` to reset the offset to the latest message in each partition.
--to-offset	Used with `--reset-offsets` to reset the offset to a specific value for each partition.
--shift-by	Used with `--reset-offsets` to shift the offset by a relative value for each partition.

Command	Explanation
--execute	Executes the offset reset operation. Without this flag, the reset operation is only displayed but not executed.
--all-topics	Used with `--describe` to describe the group state for all topics.
--all-groups	Used with `--list` to list all consumer groups.

Note: While ZooKeeper is still a valid option for managing Kafka clusters, it's no longer the most up-to-date approach. The Kafka community is moving towards the self-managed KRaft mode, which simplifies the architecture and improves scalability.

Best Practices

- **Security**: When working with CLI tools in a production environment, always consider security implications, especially when configuring ACLs or SSL/TLS.
- **Performance Monitoring**: Use CLI tools like `kafka-consumer-groups.sh` for monitoring consumer lag and ensuring that your consumers are performing as expected.
- **Configuration Management**: Use `kafka-configs.sh` to adjust and fine-tune topic and broker configurations to optimize performance and resource usage.

Troubleshooting

- **Consumer Group Issues**: Utilize `kafka-consumer-groups.sh` to identify and resolve issues related to consumer group lag and partition assignment.
- **Topic Configuration**: Modify topic configurations dynamically using `kafka-configs.sh` to address performance bottlenecks.

Useful Commands

Here are a few commands that are particularly useful for managing a Kafka environment:

- Listing topics:
- `kafka-topics.sh --list --bootstrap-server localhost:9092`
- Describing topic details:
- `kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic my-topic`
- Resetting consumer group offsets:
- `kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group my-group --reset-offsets --to-earliest --execute --topic my-topic`

Default Ports

- **Broker**: 9092
- **Schema Registry**: 8081
- **REST Proxy**: 8082
- **KSQL**: 8088
- **Zookeeper Client Port**: 2181
- **Zookeeper Leader Port**: 3888
- **Zookeeper Election Port**: 2888

CHAPTER 3 : CLUSTER ADMINISTRATION

Kafka Cluster Administration Configurations

Effective Kafka cluster management involves tuning various configurations to ensure optimal operation, fault tolerance, and efficient resource utilization. Key administrative settings help in maintaining balance across the cluster, managing topic lifecycle, and ensuring leadership distribution for resilience.

Key Points for CCDAK on Cluster Administration

- **Cluster Balance and Efficiency**: Settings related to leader election and rebalance are crucial for evenly distributing the load across the cluster.
- **Fault Tolerance**: Proper configuration ensures the cluster remains operational even when certain nodes or partitions fail.
- **Topic Management**: Ability to dynamically manage topics, including deletion, is essential for operational flexibility.
- Kafka uses a binary protocol over TCP.

APIs

There are five key APIs in Kafka:

- **Producer API:** Enables applications to send streams of data to topics within the Kafka cluster.
- **Consumer API:** Allows applications to read streams of data from topics within the Kafka cluster.
- **Streams API:** Facilitates the transformation of data streams from input topics to output topics.
- **Connect API:** Provides the capability to implement connectors that continuously pull data from source systems into Kafka or push data from Kafka to sink systems.
- **Admin API:** Offers tools for managing and inspecting topics, brokers, and other Kafka objects.

Important Cluster Administration Properties

`controller.socket.timeout.ms`

- **Description**: Timeout for controller-to-broker sockets. This setting impacts how long the controller waits for a response from a broker.
- **Default Value**: Typically set to 30000 milliseconds (30 seconds).
- **Impact**: Affects the responsiveness and reliability of controller commands. Lower values may cause premature timeouts, while higher values can delay cluster reconfiguration in case of broker failures.

`leader.imbalance.check.interval.seconds`

- **Description**: Frequency at which the leader imbalance checker runs. This tool assesses if leaders are evenly distributed across the brokers.
- **Default Value**: Often set to 300 seconds (5 minutes).
- **Impact**: Adjusting this interval affects how quickly the cluster reacts to imbalances. More frequent checks can lead to a more balanced cluster but at the cost of increased control-plane traffic.

`leader.imbalance.per.broker.percentage`

- **Description**: The allowed percentage of leader imbalance on a broker before triggering a leader rebalance.
- **Default Value**: Usually configured around 10%.
- **Impact**: Defines the threshold for triggering rebalances. Lower thresholds aim for stricter balance but might lead to more frequent leadership changes, potentially impacting performance.

`auto.leader.rebalance.enable`

- **Description**: Enables automatic leader rebalancing to evenly distribute leader partitions across brokers in the cluster.
- **Default Value**: Typically true.
- **Impact**: When enabled, it helps in maintaining balance and can improve overall cluster performance and fault tolerance. However, frequent rebalancing can sometimes cause temporary performance dips.

`delete.topic.enable`

- **Description**: Allows topics to be deleted. When enabled, topics can be deleted, and their resources reclaimed.
- **Default Value**: True in newer versions of Kafka.
- **Impact**: Provides flexibility in managing topics but requires careful consideration to avoid accidental data loss. Ensuring this is enabled allows for better topic lifecycle management.

Internal Kafka Topics

Here are the important internal topics:

1. `__consumer_offsets`:

- Stores the offsets of consumer groups.
- Contains information about the last committed offset for each partition consumed by a consumer group.
- Helps in tracking the progress of consumers and enables them to resume from the last committed offset in case of failures or restarts.

2. `__transaction_state`:

- Stores the state of ongoing transactions in Kafka.
- Used by the Kafka transactional producer to ensure atomic writes across multiple partitions and topics.
- Enables transaction coordination and ensures data integrity in transactional messaging.

3. `__confluent.support.metrics`:

- A topic used by Confluent Control Center to store and retrieve metrics data.
- Contains metadata and information related to the health and performance of the Kafka cluster.
- Enables monitoring, alerting, and data visualization in Control Center.

4. `_schemas`:

- A topic used by the Confluent Schema Registry to store schema information.
- Contains versioned schemas associated with Kafka topics.
- Enables schema evolution and compatibility checks for Kafka producers and consumers.

5. `connect-configs`:

- A topic used by Kafka Connect to store connector and task configurations.
- Contains the configuration settings for Kafka Connect connectors and their associated tasks.
- Allows dynamic configuration updates and management of Kafka Connect deployments.

6. `connect-offsets`:

- A topic used by Kafka Connect to store the offsets of connector tasks.
- Contains information about the last processed offset for each partition by a connector task.
- Enables fault tolerance and resumption of connector tasks from the last processed offset.

7. `connect-status`:

- A topic used by Kafka Connect to store status information about connector tasks.
- Contains status updates and metadata related to the execution of connector tasks.
- Allows monitoring and tracking the health and progress of Kafka Connect tasks.

8. `_confluent-command`:

- A topic used by Confluent Control Center to store and manage command requests.
- Enables sending and receiving control commands between Control Center and other Confluent components.
- Facilitates actions like starting/stopping connectors, modifying configurations, and triggering rebalances.

9. `_confluent-monitoring`:

- A topic used by Confluent Control Center for monitoring purposes.
- Contains monitoring data and metrics related to the Confluent Platform components.
- Helps in collecting and analyzing health and performance data for the Confluent ecosystem.

10. `_confluent-secrets`:

- A topic used by Confluent Control Center to store and manage secrets securely.
- Enables secure storage and retrieval of sensitive configuration values, such as passwords and API keys.
- Provides a centralized and encrypted storage for secrets used across the Confluent Platform.

11. `__consumer_timestamps`:

- Stores the last consumed timestamp for each partition by a consumer group.
- Helps in monitoring consumer progress and identifying potential issues related to consumer lag.

12. `_confluent-metrics`:

- A topic used by Confluent Control Center to store and manage metrics data.
- Contains metrics related to the performance and health of Kafka clusters, topics, and clients.
- Enables data visualization, alerting, and monitoring in Control Center.

13. `_confluent-telemetry-metrics`:

- A topic used by Confluent Control Center to store and manage telemetry data.
- Contains usage and analytics data related to the Confluent Platform components.
- Helps in understanding platform usage, adoption, and performance trends.

14. `ksql-clusterksql_processing_log`:

- A topic used by Confluent ksqlDB for storing processing logs.
- Contains information about the processing of ksqlDB queries, including errors and status updates.
- Helps in monitoring and troubleshooting ksqlDB processing pipelines.

15. `_confluent-license`:

- A topic used by Confluent Control Center to store and manage license information.
- Contains data related to the Confluent Platform license, including expiration and feature entitlements.
- Helps in tracking and enforcing license compliance across the platform.

CHAPTER 4 : CONSUMER

Kafka Consumer Overview

Kafka Consumers read records from Kafka topics. They can subscribe to one or more topics and process the stream of records produced to them. Consumers are part of consumer groups to ensure scalable and fault-tolerant processing.

Points to Remember for CCDAK on Consumers

- **Consumer Groups**: Consumers within the same group divide topic partitions among themselves to ensure balanced processing.
- **Offset Management**: Consumers track their offset per partition to manage their position within the stream.
- **Rebalance Protocol**: Ensures partition ownership is balanced across all consumer instances in a group.
- **Fault Tolerance**: Consumers can recover from failures, continuing processing from their last committed offset.

When a consumer wants to join a group, it sends a JoinGroup request to the group coordinator. The first consumer to join becomes the group leader. The leader receives a list of all consumers in the group from the group coordinator (including those that sent a recent heartbeat and are considered alive) and is responsible for assigning a subset of partitions to each consumer.

Partition Assignors

A `PartitionAssignor` is a class that, given consumers and the topics they subscribed to, decides which partitions will be assigned to which consumer. Kafka has two default assignment strategies:

1. Range Assignor

- **How It Works**: Divides the sorted list of partitions into contiguous ranges and assigns each range to a consumer. The assignment is sequential based on the sorted order of consumer IDs.
- **Key Points**: Simple and predictable; however, it can lead to imbalanced workloads if the number of partitions is not a multiple of the number of consumers. It was the default partition assignor in Kafka versions prior to 2.4.
- **Example Scenario**: Imagine a topic with 12 partitions (P0 to P11) and 3 consumers (C0, C1, C2). The assignment might look like this: C0: P0, P1, P2, P3 - C1: P4, P5, P6, P7 - C2: P8, P9, P10, P11

2. Round Robin Assignor

- **How It Works**: Assigns partitions to consumers in a round-robin fashion across all consumers, ensuring each consumer gets a partition before any consumer gets a second one.

- **Key Points**: Provides better load balancing compared to the Range Assignor, especially when the number of partitions is not evenly divisible by the number of consumers. However, it can lead to rebalancing when consumers are added or removed.
- **Example Scenario**: Using the same topic with 12 partitions (P0 to P11) and 3 consumers (C0, C1, C2), the assignment would be: C0: P0, P3, P6, P9 - C1: P1, P4, P7, P10 - C2: P2, P5, P8, P11

3. Sticky Assignor

- **How It Works**: Aims to maintain a sticky relationship between consumers and partitions across rebalances, minimizing the number of partitions that get reassigned to different consumers.
- **Key Points**: Reduces the amount of data that needs to be re-fetched when partitions are reassigned; offers a good balance between fairness and stability. It was introduced in Kafka 2.4 and became the default assignor, replacing the Range Assignor.
- **Example Scenario**: After an initial assignment, if a new consumer (C3) joins, the Sticky Assignor would try to minimize partition movement. It might reassign only a few partitions from the existing consumers to the new one, like this: C0: P0, P3, P6 - C1: P1, P4, P7 - C2: P2, P5, P11 - C3: P8, P9, P10

4. Cooperative Sticky Assignor

- **How It Works**: Similar to the Sticky Assignor but allows for more cooperative rebalancing. It minimizes the impact on consumers by only reassigning partitions that need to be moved instead of revoking all partitions and redistributing them.
- **Key Points**: Designed to reduce the impact of rebalancing, allowing consumers to continue consuming while rebalance is in progress, and reducing the time it takes to complete a rebalance. It was introduced in Kafka 2.4 as an improvement over the Sticky Assignor.
- **Example Scenario**: When a consumer leaves the group, the Cooperative Sticky Assignor would only reassign the partitions that were consumed by the leaving consumer, without affecting the assignments of other consumers. This minimizes the rebalancing impact on the consumer group.

Mnemonic to Remember

"RRSC" - Round, Range, Sticky, Cooperative

- **Round** for **Round Robin**: Think of a round table where everyone gets a slice of cake one by one.
- **Range** for **Range Assignor**: Imagine dividing a chocolate bar into contiguous pieces where each person gets a range of pieces.
- **Sticky** for **Sticky Assignor**: Like sticky notes, once a partition is assigned to a consumer, it tries to "stick" with them across rebalances.
- **Cooperative** for **Cooperative Sticky Assignor**: Think of a team project where everyone cooperates, making changes only when necessary, thus minimizing disruption.

Important Consumer Properties

`group.initial.rebalance.delay.ms`

- **Default**: 3000 (3 seconds)
- **Description**: Delay time before initial rebalance. A higher value can reduce rebalance storms in large clusters.
- **Trade-offs**: Longer delays may slow initial startup time for consumer groups.

`max.poll.intervals.ms`

- **Default**: 300000 (5 minutes)
- **Description**: Maximum delay between invocations of poll() when using consumer group management. Exceeding this will cause the consumer to leave the group.
- **Trade-offs**: Higher values provide more leeway for processing, but risk slower recovery from failures.

`max.poll.records`

- **Default**: 500
- **Description**: Maximum number of records returned in a single call to poll().
- **Trade-offs**: Lower values improve latency at the cost of throughput, and vice versa.

`enable.auto.commits`

- **Default**: true
- **Description**: If true, offsets are committed automatically at intervals.
- **Trade-offs**: Manual offset control can improve accuracy at the cost of convenience.

`auto.offset.reset`

- **Default**: latest
- **Description**: Controls the action when no initial offset is found for a consumer's group. Options are "earliest", "latest", or "none".
- **Trade-offs**: "earliest" may result in more data being processed, "latest" might miss messages if the consumer is down.

`fetch.min.bytes`

- **Default**: 1
- **Description**: Minimum data the server should return for a fetch request. Helps in controlling the number of updates.
- **Trade-offs**: Larger values can improve throughput but increase latency.

`max.partition.fetch.bytes`

- **Default**: 1048576 (1MB)

- **Description**: Maximum amount of data per partition the server will return.
- **Trade-offs**: Larger values increase throughput but can lead to more memory use.

`fetch.max.bytes`

- **Default**: 52428800 (50MB)
- **Description**: Maximum amount of data the server will return for a fetch request across all partitions.
- **Trade-offs**: Higher values allow more data to be fetched in a single request, affecting memory usage.

`session.timeout.ms`

- **Default**: 10000 (10 seconds)
- **Description**: Time used to detect consumer failures. If no heartbeat is received within this time, the consumer is considered dead.
- **Trade-offs**: Lower values make detection faster but may cause unnecessary rebalances.

`heartbeat.interval.ms`

- **Default**: 3000 (3 seconds)
- **Description**: Expected time between heartbeats to the group coordinator.
- **Trade-offs**: Lower values may lead to more frequent rebalances.

`isolation.level`

- **Default**: read_uncommitted
- **Description**: Controls visibility of transactional messages. "read_committed" filters out transactions not yet committed.
- **Trade-offs**: "read_committed" ensures cleaner data at the cost of potential latency.

`partition.assignment.strategy`

- **Default**: [org.apache.kafka.clients.consumer.RangeAssignor]
- **Description**: Determines the strategy for partition assignment among consumers.
- **Trade-offs**: Different strategies can optimize for fairness or throughput.

`client.rack`

- **Default**: Not set
- **Description**: Specifies the client's rack to enable rack-aware partition assignment.
- **Trade-offs**: Can reduce cross-rack traffic at the cost of potential imbalance in local traffic.

Additional Consumer Configurations and Practices

Understanding Consumer Behavior

- **Subscription Types**: Kafka consumers can subscribe to topics in different ways, such as by specific topic names, pattern matching on topic names, or direct partition assignment. (e.g. `consumer.subscribe(Pattern.compile(".*topic"));` and `consumer.subscribe(Arrays.asList("orders-topic", "customers-topic", "sales-topic", "stocks-topic"));`)
- **Polling Process**: Inside the consumer's `poll()` call, several checks and operations occur, including heartbeat checks, subscription updates, and potentially triggering rebalance if necessary.

Consumer Rebalancing Triggers

Rebalancing, a critical aspect of Kafka consumer groups for ensuring even data processing distribution, can be triggered by:

- Changes in the consumer group (e.g., new consumer joining, existing consumer leaving).
- Changes in topic partition counts.
- Manual rebalance triggers like calling `unsubscribe()`.

Consumer group rebalances are managed by the Kafka consumers themselves, not by the Controller. When a consumer joins or leaves a consumer group, the consumers coordinate among themselves to redistribute the partitions they are consuming from.

Managing Consumer Offsets

- **Auto-Commit Strategy**: By default, Kafka consumers automatically commit offsets at regular intervals, but manual offset management can provide finer control over when and how offsets are committed, which is essential for precise processing records management.
- **Offset Reset Behavior**: Determines how consumers behave when no initial offset is found. `auto.offset.reset` can significantly impact consumer startup behavior, especially in scenarios where precise data replay or skipping is required.
- The consumer is responsible for committing the offsets to the `__consumer_offsets` topic. However, **the consumer does not directly write the offset information** to the topic. Instead, it sends the offset information to the **group coordinator broker**, which then writes the offsets to the `__consumer_offsets` topic on behalf of the consumer.

Multi-threading and Consumer Safety

- **Single-threaded Nature**: Each Kafka consumer instance is not thread-safe. The recommended practice is one consumer per thread to avoid concurrency issues.
- **Consumer Thread Safety**: While the consumer itself is not thread-safe, certain operations like `wakeup()` can be safely called from another thread to interrupt an ongoing operation, such as a long `poll()`. (`WakeupException` will be thrown)

High Availability and Fault Tolerance

- **Consumer Group Coordination**: Kafka uses the GroupCoordinator and ConsumerCoordinator components to manage consumer groups' state and rebalancing, ensuring consumers within a group efficiently distribute workload among themselves.

- **Heartbeats and Session Management**: By adjusting `heartbeat.interval.ms` and `session.timeout.ms`, you can fine-tune how quickly Kafka detects failed consumers and rebalances partitions, balancing between responsiveness and stability.

Consumer Polling and Processing Patterns

- **Long Processing Times**: If your consumer processing time might exceed `max.poll.interval.ms`, consider batching records and processing them more efficiently or adjusting the `max.poll.interval.ms` to allow more time for processing while avoiding unintended group rebalancing.
- **Batch Processing**: Consumers can adjust `max.poll.records` to control the number of records fetched in each poll call, allowing for more predictable processing times and better throughput control.

Offset and Partition Management

- **Manual Partition Assignment**: Using the `assign()` method allows for direct control over which partitions a consumer processes, bypassing Kafka's consumer group management and rebalance protocol. This approach is suitable for scenarios requiring static partition assignments.
- **Custom Offset Storage**: While Kafka provides built-in offset management through the `__consumer_offsets` topic, consumers can implement custom offset storage mechanisms if needed, allowing for greater flexibility in offset management strategies.
- Kafka allows specifying the position using `seek(TopicPartition, long)` to specify the new position (offset) a consumer can read from. Also `seekToBeginning(TopicPartition tp)` and `seekToEnd(TopicPartition tp)`.

Optimizing Consumer Performance

- **Tuning Fetch Parameters**: Adjust `fetch.min.bytes`, `fetch.max.bytes`, and `max.partition.fetch.bytes` to optimize the balance between latency and throughput based on your application's specific needs.
- **Consumer Lag Monitoring**: Keeping an eye on `records-lag-max` can help identify when a consumer is not keeping up with the producers, allowing for timely adjustments to consumer configurations or scaling out consumers.

High Watermark in Kafka

The **high watermark (HW)** is a critical concept in Kafka, ensuring data consistency and reliability. It represents the offset of the last message that has been successfully replicated to all **In-Sync Replicas (ISR)** of a partition.

Group Coordinator and Group Leader

The **group coordinator** is a designated **broker** that receives heartbeats from all `consumers` within a `consumer group`. Each `consumer group` is assigned a single group coordinator. If a consumer fails to send heartbeats, the coordinator initiates a rebalance process.

When a consumer wishes to join a `consumer group`, it sends a `JoinGroup` request to the group coordinator. The first consumer to join the group is appointed as the group leader.

Metadata request

When a consumer has the wrong partition leader, it sends a metadata request to any Kafka broker. The response includes metadata for each partition, grouped by topic:

- **Leader**: The current leader broker's node ID for the partition (-1 if no leader exists).
- **Replicas**: The alive slave brokers for the partition's leader.
- **ISR**: The subset of caught-up replicas.
- **Broker**: The node ID, hostname, and port of a Kafka broker.

Key Points:

1. **Definition**:
    - The high watermark is the offset of the last message replicated to all ISR.
    - It ensures that consumers only read fully replicated messages.
2. **In-Sync Replicas (ISR)**:
    - ISR are replicas that are fully caught up with the leader replica.
    - The leader handles all reads and writes for a partition.
3. **Replication and Acknowledgements**:
    - When a producer sends a message, it is written to the leader replica and then replicated to all ISR.
    - Producers specify the acknowledgment level (`acks`):
        - `acks=0`: No acknowledgment needed.
        - `acks=1`: Leader acknowledgment only (default).
        - `acks=-1` or `acks=all`: All ISR acknowledgment.
4. **Message Visibility**:
    - Consumers can only read messages up to the high watermark.
    - This ensures that only fully replicated messages are read, maintaining data consistency.
    - If `acks=1`, the highest offset (latest message) can be greater than the high watermark if the message is not yet replicated.

Example Scenario:

- A producer sends a message with offset 10.
- With `acks=1`, the leader acknowledges the write before replication.
- The high watermark may remain at offset 9 until the message at offset 10 is replicated to all ISR.
- Consumers will read messages up to offset 9 until offset 10 is fully replicated.

Security Considerations

- **Encryption and Authentication**: Secure consumer connections to Kafka brokers using SSL/TLS for encryption and SASL for authentication to protect data in transit and ensure that only authorized consumers can access topic data.

CHAPTER 5 : KSQL

KSQL Overview for CCDAK

KSQL, known as ksqlDB in its more recent versions, is a stream processing framework that enables real-time data processing against Apache Kafka. It uses a SQL-like language, making it accessible to developers familiar with SQL.

Key Concepts

- **Stream Processing**: Understand how KSQL processes data in real-time, transforming, enriching, and querying data streams.
- **KSQL Syntax**: Familiarity with the SQL-like syntax for creating streams, tables, and performing queries.
- **Streams vs. Tables**: Grasp the difference between a stream (an immutable sequence of records) and a table (a mutable, stateful entity).

ksqlDB supports exactly-once processing, ensuring correct results even in the event of failures such as machine crashes. Developers can configure this behavior using the `processing.guarantee` setting.

Idle servers in a ksqlDB cluster consume minimal resources. When running a query against a Kafka topic with multiple partitions, only the servers corresponding to the number of partitions perform the actual work.

Essential Components

- **KSQL Server**: The server that runs KSQL queries. It interacts with Kafka to execute stream processing applications.
- **KSQL CLI**: The command-line interface to interact with KSQL Server, useful for executing commands and queries.

Configuration Details

- **`auto.offset.reset`**: By default, it is set to "latest", meaning that streams start reading messages that arrive after their creation. Can be changed to "earliest" to read from the beginning.
- **`ksql.streams.state.dir`**: Defines the directory for RocksDB storage used by stateful operations.

Advanced Stream Creation

- **Existing Topic**: Streams can be created from existing Kafka topics, with support for various data formats.
- **Stream Count/Join**: Creating streams through aggregations or joins on existing streams.

- **Persistent Streaming Query**: Continuously running queries that write results to a new stream or table.

Additional Stream Features

- **ROWTIME and ROWKEY**: Every KSQL stream includes `ROWTIME` and `ROWKEY` fields by default, representing the message timestamp and the key of the topic, respectively.
- **Re-Keying and Materializing**: Techniques for re-keying a stream and creating a materialized view to support efficient querying.

Query Types

- **Pull Queries**: Execute against materialized views with limitations on the WHERE clause to key column constraints.
- **Push Queries**: Subscribe to future query results with `EMIT CHANGES`, allowing real-time streaming of query results.

Merging Streams

- Demonstrates how to combine data from multiple streams into a single stream, useful for aggregating similar data from different sources.

Windowing Functions

- **TOPK/TOPKDISTINCT**: Functions for retrieving the top K values within a window, useful for analytics and leaderboards.
- **GEO_DISTANCE**: A function for calculating geospatial distances, enabling location-based filtering and analysis.

Important Commands and Concepts

- **Creating Streams and Tables**: How to define streams and tables over Kafka topics to start processing data.

  ```
  CREATE STREAM stream_name (column_name data_type, ...) WITH
  (kafka_topic='topic_name', value_format='format', ...);
  ```
- **Data Processing**: Understanding of how to perform data transformations, aggregations, and joins using KSQL.
- **Windowing**: Knowledge of windowing functions to process data in time-bound chunks.
- **User-Defined Functions (UDFs)**: Familiarity with using and possibly creating custom functions for specialized processing needs.

The `DESCRIBE EXTENDED` statement in ksqlDB provides detailed information about a stream or table, including its serialization format. Checking the compatibility between the `VALUE_FORMAT` of a stream and the format of records in a topic can help diagnose deserialization issues.

Best Practices

- **Query Optimization**: Tips for optimizing query performance, such as minimizing the use of repartitioning and choosing appropriate keys for joins.
- **Monitoring and Troubleshooting**: Basic practices for monitoring KSQL performance and troubleshooting common issues.

Real-world Application Scenarios

- **Real-Time Analytics**: Use KSQL for instant analytics on streaming data, such as calculating average values, counts, or processing time-series data.
- **Anomaly Detection**: Implement anomaly detection systems by processing and analyzing data in real-time, identifying outliers or unusual patterns.
- **Data Enrichment**: Enrich streaming data by joining it with static datasets or additional streams, adding value and context to the data.

Headless mode

ksqlDB offers two deployment modes:

1. **Headless ksqlDB deployment (Application Mode):**
    - This is the recommended mode for production deployments.
    - You write your queries in a SQL file and start ksqlDB server instances with this file as an argument.
    - Each server instance reads the SQL file, compiles the ksqlDB statements into Kafka Streams applications, and starts executing the generated applications.
2. **Interactive ksqlDB deployment:**
    - In this mode, you interact with the ksqlDB servers through a REST API.
    - You can interact directly using REST clients, the ksqlDB CLI, or Confluent Control Center.
    - This mode is suitable for development, testing, and exploratory purposes.

`ksqlDB` supports a locked-down, "headless" deployment mode that disables interactive access to the `ksqlDB` cluster. This mode is useful in production environments where you want to ensure that only predefined queries are executed and prevent direct user interaction with the cluster.

In a typical workflow:

1. A team of users develops and tests their queries interactively on a shared testing `ksqlDB` cluster using the CLI.
2. When ready for production deployment, the verified queries are version-controlled and stored in a `.sql` file.
3. The `.sql` file is provided to the `ksqlDB` servers during startup using either:
    - The `--queries-file` command-line argument, or
    - The `ksql.queries.file` setting in the `ksqlDB` configuration file.

When a `ksqlDB` server is running with a predefined script (`.sql` file), it automatically disables its REST endpoint and interactive use. This lockdown mechanism ensures that the server only executes the predefined queries and prevents any direct interaction with the production `ksqlDB` cluster. In headless mode, ksqlDB stores metadata in the config topic. The config topic stores the ksqlDB properties provided to ksqlDB when the application was first started. ksqlDB uses these configs to ensure that your ksqlDB queries are built compatibly on every restart of the server.

ksqlDB In-Depth

1. ksqlDB Overview

- ksqlDB is an event streaming database that allows building event streaming applications using SQL-like syntax.
- It uses Kafka Streams under the hood, so understanding Kafka Streams concepts is beneficial when using ksqlDB.
- ksqlDB runs continuous queries that constantly evaluate incoming events from Kafka topics and can persist results back to Kafka or return them to clients.
- ksqlDB offers a powerful way to build event streaming applications using familiar SQL syntax, making it accessible to a wider audience beyond Java developers.
- Question: How does ksqlDB handle state management and fault tolerance?
    - ksqlDB leverages Kafka's fault-tolerant and distributed architecture for state management and fault tolerance.
    - It uses Kafka topics to store intermediate state and supports exactly-once semantics through transaction support.

2. Streams and Tables

- ksqlDB supports creating `STREAM`s and `TABLE`s, which have similar semantics to KStream and KTable in Kafka Streams.
    - A `STREAM` represents an unbounded sequence of independent events, where records with the same key are not related.
    - A `TABLE` is an update stream, where records with the same key are considered updates to previous records with that key.
- Streams and tables can be created from a Kafka topic using a `CREATE STREAM` or `CREATE TABLE` statement.
- The `WITH` clause is used to specify properties like the Kafka topic, number of partitions, and serialization format.
- Key columns are optional for streams but required for tables. They are defined using the `KEY` keyword.
- **For tables, null keys are not allowed and will cause the record to be dropped**. For streams, null keys are allowed.
- **In tables, null values are considered tombstones and mark the record for deletion** . In streams, null values have no special meaning.
- Question: Can you create a stream or table without an underlying Kafka topic?
    - No, every stream or table in ksqlDB must be backed by a Kafka topic.

- The Kafka topic can be pre-existing, or ksqlDB can create it automatically based on the stream or table definition.

3. Query Types

1. **Source queries**:
    - Create a `STREAM` or `TABLE` backed by a Kafka topic.
    - Defined using `CREATE STREAM` or `CREATE TABLE` statements.
    - Bring data from a Kafka topic into ksqlDB for further processing.
2. **Persistent queries**:
    - Select columns from a source query and persist results to a Kafka topic.
    - Defined using `CREATE STREAM AS SELECT` or `CREATE TABLE AS SELECT` statements.
    - Results are stored in a Kafka topic and can be shared by multiple clients.
    - Continuously process incoming events and update the result topic.
3. **Push queries**:
    - Select a subset of columns from a persistent query and stream results directly to the client.
    - Defined using a `SELECT` statement with the `EMIT CHANGES` clause.
    - Run indefinitely until terminated by the client.
    - Results are not persisted to a topic; they are returned to the client in real-time.
4. **Pull queries**:
    - Execute once and terminate, returning a result set to the client.
    - Defined using a regular `SELECT` statement without the `EMIT CHANGES` clause.
    - Have some limitations on supported SQL statements (e.g., no `JOINs`, `GROUP BY`, or `WINDOW` clauses).
    - Results are not persisted to a topic and are not shared; identical queries from different clients are executed independently.
    - Question: Can pull queries be executed on any stream or table?
        - Pull queries **can only be executed on materialized streams or tables that have a defined primary key**.
        - They cannot be executed on non-materialized streams or tables without a primary key.

- Question: What happens if a persistent query fails or is terminated unexpectedly?
    - If a persistent query fails, ksqlDB will automatically restart it from the last committed offset.
    - The query will resume processing events from where it left off, ensuring no data loss.
    - If a query is terminated unexpectedly (e.g., due to server failure), it will be automatically restarted on another ksqlDB server in the cluster.

4. Schema Registry Integration

- ksqlDB integrates with Schema Registry for Avro, Protobuf, and JSON Schema serialization formats.

- Streams and tables inherit key and value formats from their backing topics or source streams/tables.
- Formats can be changed using a `WITH` clause when defining a stream or table.
- When using a schema-based format, column names and types can be omitted in the definition, as they are inferred from the schema.
- ksqlDB automatically registers schemas with Schema Registry when creating streams or tables based on persistent queries.
- The `ksql.schema.registry.url` property must be set to the Schema Registry URL to enable schema integration.
- Question: What happens if the schema of a topic changes and doesn't match the schema registered in ksqlDB?
    - If the schema of a topic changes and doesn't match the registered schema, ksqlDB will fail to deserialize the data and will report an error.
    - To handle schema changes, you need to update the stream or table definition in ksqlDB to match the new schema.
    - ksqlDB supports schema evolution using Confluent Schema Registry's compatibility rules (e.g., backward, forward compatibility).

5. Aggregations and Functions

- ksqlDB provides built-in aggregation functions like `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`.
- Aggregations are performed using the `GROUP BY` clause in a persistent query.
- Aggregation results are stored in a table, which is backed by a Kafka topic.
- ksqlDB supports user-defined functions (UDFs) and user-defined aggregation functions (UDAFs) for custom processing logic.
- ksqlDB has a limitation in that it doesn't support structured keys, preventing the creation of a stream from a windowed aggregate.
- Question: Can you perform aggregations on windowed data in ksqlDB?
    - Yes, ksqlDB supports windowed aggregations using the `WINDOW` clause.
    - Windows can be defined based on time (e.g., tumbling, hopping, session windows) or custom window boundaries.
    - Windowed aggregations allow grouping and aggregating data within specific time intervals.

6. Joins

- **Stream-Stream joins**:
    - Create new streams by joining two streams.
    - Requires co-partitioning (same key type and number of partitions).
    - Supports `INNER`, `LEFT OUTER`, `RIGHT OUTER`, and `FULL OUTER` joins.
    - Defined using a `CREATE STREAM AS SELECT` statement with a `JOIN` clause.
- **Stream-Table joins**:
    - Enrich a stream with data from a table.
    - Requires co-partitioning.

    - Performs a lookup in the table for each record in the stream.
    - Defined using a `CREATE STREAM AS SELECT` statement with a `JOIN` clause.
- **Table-Table joins**:
    - Perform joins between tables using primary keys or foreign keys.
    - Primary key joins require co-partitioning.
    - Foreign key joins allow joining on a non-primary key field in the value of one table.
    - Defined using a `CREATE TABLE AS SELECT` statement with a `JOIN` clause.
- While Stream-KGlobalTable offers significant convenience and flexibility for join operations by eliminating the co-partitioning requirement, it also means that each application instance requires enough memory to hold the entire dataset.
- Foreign Key Joins: Do not require co-partitioning based on the primary key of each table.
- Question: What is the default join window size for stream-stream joins in ksqlDB?
    - The default join window size is 24 hours.
    - You can specify a custom join window size using the `WITHIN` clause in the `JOIN` statement.
    - The join window determines the maximum time difference between the joining records.
- Question: Can you perform a left join between a stream and a table in ksqlDB?
    - Yes, you can perform a left join between a stream and a table using the `LEFT JOIN` syntax.
    - The stream is the left side of the join, and the table is the right side.
    - For each record in the stream, ksqlDB will perform a lookup in the table based on the join key.
    - If a matching record is found in the table, the join result will include fields from both the stream and the table.
    - If no matching record is found in the table, the join result will include fields from the stream and NULL values for the fields from the table.

7. Nested Data Handling

- ksqlDB supports querying nested data structures using the `STRUCT` data type.
- `STRUCT` is used to model nested schemas in streams or tables.
- Nested fields are accessed using the `->` operator to dereference objects and drill down to specific fields.
- Array elements can be accessed using array indexing, e.g., `field->array[index]`.
- Map values can be accessed using `field->map['key']`.
- Nested data can be flattened using the `EXPLODE` function to create separate records for each element in an array or map.
- Question: Can you perform joins on nested fields in ksqlDB?
    - Yes, you can perform joins on nested fields by accessing them using the `->` operator.
    - For example, you can join two streams based on a nested field using `JOIN ON stream1->nested->field = stream2->nested->field`.

8. Deployment and Scaling

- ksqlDB can be deployed in standalone mode or as a cluster of servers.
- In standalone mode, a single ksqlDB server is responsible for processing all queries and interacting with the Kafka cluster.
- In cluster mode, multiple ksqlDB servers work together to distribute the processing load and provide fault tolerance.
- ksqlDB leverages Kafka's distributed architecture to scale horizontally by adding more servers to the cluster.
- Persistent queries are automatically distributed across the available ksqlDB servers based on the partitioning of the input topics.
- ksqlDB supports interactive and headless (non-interactive) deployment modes.
    - Interactive mode allows users to submit queries and receive results through a REST API or command-line interface (CLI).
    - Headless mode is suitable for production deployments, where queries are predefined and executed automatically without user interaction.
- Question: How does ksqlDB handle service discovery in a clustered deployment?
    - In a clustered deployment, ksqlDB servers use Apache ZooKeeper for service discovery and coordination.
    - Each ksqlDB server registers itself with ZooKeeper and maintains a persistent session.
    - Clients (e.g., CLI, REST API) discover the available ksqlDB servers by querying ZooKeeper.
    - If a ksqlDB server fails or becomes unavailable, ZooKeeper notifies the other servers, and the workload is redistributed among the remaining servers.

9. Monitoring and Troubleshooting

- ksqlDB exposes metrics through JMX (Java Management Extensions) for monitoring the health and performance of the system.
- Key metrics to monitor include:
    - `messages-consumed-per-sec`: The number of messages consumed per second by ksqlDB.
    - `messages-produced-per-sec`: The number of messages produced per second by ksqlDB.
    - `error-rate`: The number of errors occurred per second in ksqlDB.
    - `num-active-queries`: The number of active queries running in ksqlDB.
- ksqlDB provides a REST API for managing and monitoring the system, including retrieving cluster information, listing streams and tables, and executing queries.
- The ksqlDB CLI can be used to interact with the system, execute queries, and view query results and logs.
- If ksqlDB doesn't clean up its internal topics make sure that your Kafka cluster is configured with the property delete.topic.enable=true.
- Logs can be configured and monitored to troubleshoot issues and track the execution of queries.
- ksqlDB supports configurable error handling, such as deserialization errors and production exceptions, through the use of dead letter queues (DLQs).

- Question: How can you monitor the resource utilization of ksqlDB servers?
    - ksqlDB exposes metrics for CPU, memory, and network utilization through JMX.
    - You can use tools like JConsole, VisualVM, or Prometheus to scrape and visualize these metrics.
    - Monitoring resource utilization helps in identifying performance bottlenecks and capacity planning.
- Question: What are the best practices for troubleshooting ksqlDB queries?
    - Use the `DESCRIBE` and `EXPLAIN` statements to understand the schema and execution plan of a query.
    - Monitor the ksqlDB logs for error messages and stack traces.
    - Use the `SHOW QUERIES` statement to view the status and metrics of running queries.
    - Enable query logging by setting the `ksql.logging.processing.topic.auto.create` and `ksql.logging.processing.stream.auto.create` properties to `true`.
    - Investigate the Kafka consumer group lag for the input topics to identify if the queries are falling behind in processing.
    - Use the ksqlDB REST API or CLI to execute `SELECT` statements and verify the query results.

CHAPTER 6 : KAFKA-CONNECT

Kafka Connect Overview

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

- A common framework for Kafka connectors.
- Distributed and standalone modes.
- REST interface.
- Automatic offset management.
- Distributed and scalable by default.
- Streaming/batch integration.

Key Concepts

- **Connectors**: The components in Kafka Connect that implement the connection to other systems. There are two types of connectors: Source connectors for importing data into Kafka, and Sink connectors for exporting data from Kafka.
- **Tasks**: The actual workers that perform the data import or export. Each connector could be split into multiple tasks that can be distributed over a cluster for scalability and fault tolerance.
- **Converters**: Responsible for converting data between Kafka Connect's internal data format and the format required by external systems.

Core Components

- **Worker**: The runtime environment where connectors and tasks are executed. Workers can be standalone (single process) or distributed across multiple nodes.
- **REST API**: Kafka Connect provides a REST API for managing connectors and tasks, making it easy to deploy and monitor without writing any code.

Key Connector configurations

Connector configurations are simple key-value mappings. In standalone mode, these are defined in a properties file and passed to the Connect process on the command line. In distributed mode, they are included in the JSON payload for the request that creates or modifies the connector.

Most configurations are connector-dependent, so they can't be outlined here. However, there are a few common options:

- **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:** The maximum number of tasks that should be created for this connector. The connector may create fewer tasks if it cannot 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.

Important Operations

- **Configuring Connectors**: Understand how to configure source and sink connectors, including specifying the name of the Kafka topics to use, the key and value converters, and any connector-specific settings.

```
{
  "name": "example-connector",
  "config": {
    "connector.class": "org.example.MyConnector",
    "tasks.max": "1",
    "topics": "my-topic",
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",
    "value.converter": "org.apache.kafka.connect.json.JsonConverter",
    "value.converter.schemas.enable": "false",
    ...
  }
}
```

- **Deploying and Scaling**: Knowledge on deploying connectors in a distributed mode to leverage Kafka Connect's scalability and fault tolerance.
- **Monitoring and Managing Connectors**: Using the REST API to monitor the health and performance of connectors and tasks, and to dynamically adjust their configuration.

Best Practices

- **Error Handling and Retry Policies**: Implement robust error handling and retry mechanisms to ensure resilience and reliability.
- **Offset Management**: Proper management of offsets to ensure accurate and reliable data processing.
- **Securing Kafka Connect**: Apply security configurations, including encryption and authentication, to protect data in transit and at rest.

REST API Operations

Kafka Connect's REST API provides comprehensive control over connectors and tasks, allowing for dynamic management without direct code intervention.

- **Worker Information**: Retrieve details about Kafka Connect workers.
- **Connectors Management**: List, create, update, pause, resume, and delete connectors.
- **Tasks Management**: Monitor and manage the individual tasks of each connector.
- **Configuration Management**: View and update connector configurations.

Internal Topics of Kafka Connect

Kafka Connect utilizes several internal topics for maintaining its state, configuration, and progress:

- **Connector Config**: Stores connector configurations.
- **Task Config**: Contains task-specific configurations.
- **Offset**: Tracks the progress of source connectors to ensure data consistency.
- **Status**: Records the state and status of connectors and tasks.

These internal topics enable Kafka Connect to recover from failures, ensuring robust and reliable data streaming.

Configuration Options

Common Worker Configurations

- **`bootstrap.servers`**: The Kafka cluster to connect to.
- **`key.converter`** and **`value.converter`**: Converters for record keys and values.
- **`offset.flush.interval.ms`**: Frequency at which to save offsets.
- **`plugin.path`**: Path to directory containing Kafka Connect plugins.

Additional Configurations

- **Standalone mode**:
- `offset.storage.file.filename`: File to store offset data
- **Distributed mode**:
- `group.id`: Unique name for the cluster
- `config.storage.topic`: Topic for storing connector and task configurations
- `offset.storage.topic`: Topic for storing offsets
- `status.storage.topic`: Topic for storing statuses

Distributed Worker Configurations

- **`group.id`**: Unique identifier for the Kafka Connect cluster.
- **`config.storage.topic`** and **`offset.storage.topic`**: Kafka topics for storing connector and task configurations and offsets.

In distributed mode for Kafka Connect, key features include automatic work balancing, dynamic scaling, and fault tolerance. While configuration parameters are similar to standalone mode, the storage locations and classes differ. In distributed mode, Kafka Connect stores offsets, configurations, and task statuses in Kafka topics. It is recommended to manually create these topics to achieve the desired number of partitions and replication factors.

Execution is similar to standalone mode:

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

JDBC Sink Configuration Options

- `connection.url`, `connection.user`, `connection.password`: Database connection details.
- `insert.mode`: Mode of inserting data into the database.
- `auto.create` and `auto.evolve`: Automatically create and modify database schemas.

Connector Types

- **Source Connectors**: Import data from external systems into Kafka.
- **Sink Connectors**: Export data from Kafka to external systems.

Source Connectors

These connectors ingest data from external systems into Kafka topics.

- **JDBC Source Connector**: Ingests data from relational databases into Kafka.
- **File Source Connector**: Reads files from a file system and ingests their contents into Kafka topics.
- **MQTT Source Connector**: Integrates IoT device data communicated over MQTT protocol into Kafka.
- **Twitter Source Connector**: Streams live tweets directly into Kafka topics.

Sink Connectors

These connectors export data from Kafka topics to external systems.

- **JDBC Sink Connector**: Exports data from Kafka topics to relational databases.
- **HDFS Sink Connector**: Moves data from Kafka topics into HDFS (Hadoop Distributed File System), commonly used in big data environments.
- **Elasticsearch Sink Connector**: Exports data from Kafka topics into Elasticsearch for search and analytics.
- **S3 Sink Connector**: Stores data from Kafka topics into Amazon S3 for durable, long-term storage.

MQTT Proxy and Connector

- **MQTT (Message Queuing Telemetry Transport)** is a lightweight messaging protocol for small sensors and mobile devices optimized for high-latency or unreliable networks.
- An **MQTT proxy** allows Kafka to directly ingest data from IoT devices using the MQTT protocol.
- An **MQTT connector** (typically a Kafka Connect connector) serves a similar purpose, facilitating data flow from MQTT-enabled devices into Kafka topics.

JDBC Connectors

- **JDBC (Java Database Connectivity)** connectors are used within Kafka Connect for integrating Kafka with relational databases.

    - A **JDBC source connector** pulls data from a database into Kafka topics.
    - A **JDBC sink connector** moves data from Kafka topics into a database.

Single Message Transforms (SMT)

- **SMTs** are Kafka Connect transformations that are applied to data as it passes through a connector, allowing for modification of the data (such as enrichment, filtering, or transformation) before it lands in the destination.

Data Enrichment

- **Data enrichment** in the context of Kafka involves augmenting messages in a stream with additional information. This can be achieved by joining streams with other data sources (e.g., tables or databases) to add context or additional details to the original data.

Error Handling

Kafka Connect provides error reporting to handle errors encountered at various stages of processing. By default, any error during conversion or transformations causes the connector to fail. Each connector configuration can also enable tolerating such errors by skipping them and optionally writing each error and details of the failed operation and problematic record to the Connect application log.

To report errors within a connector's converter, transforms, or within the sink connector itself to the log, set `errors.log.enable=true` in the connector configuration to log details of each error and the problem record's topic, partition, and offset. For additional debugging purposes, set `errors.log.include.messages=true` to also log the problem record key, value, and headers (note this may log sensitive information).

To report errors within a connector's converter, transforms, or within the sink connector itself to a dead letter queue topic, set `errors.deadletterqueue.topic.name`, and optionally `errors.deadletterqueue.context.headers.enable=true`.

REST API

- Kafka Connect provides a REST API for managing connectors
- Key endpoints include viewing connector configurations, status, and restarting connectors and tasks
- Secured clusters require additional configurations for the REST API

Connector Development

- Connectors are responsible for breaking down a data copying job into tasks and monitoring for changes

- `SourceConnector`s import data from systems into Kafka, `SinkConnector`s export data from Kafka to systems
- Developers must implement the `Connector` and `Task` interfaces
- Tasks should have consistent data schemas for the input/output streams and records

Configuration Validation

- Kafka Connect allows configuration validation using `ConfigDef` and the `validate()` method
- Provides feedback about errors and recommended values before running connectors

Schemas and the Kafka Connect Data API

- More complex data requires using the data API to define schemas using `Schema` and `Struct` classes
- Source connectors may have static or dynamic schemas and should avoid recomputing them frequently
- Sink connectors should validate that consumed data has the expected schema

Kafka Connect In-Depth

1. Connect Architecture

- Kafka Connect runs as a separate process from the Kafka brokers.
- It can run in standalone mode (single process) or distributed mode (multiple instances).
- In distributed mode, Connect distributes the work of connectors across multiple worker instances.
- Workers coordinate through Kafka topics (`config`, `offset`, and `status`) for configuration, offset storage, and status updates.
- Connectors are responsible for breaking down data copying tasks and generating task configurations.
- Tasks are the actual units of work that perform data copying.
- The `max.tasks` parameter determines the maximum number of tasks a connector can create.

To understand Kafka Connect's internals, let's define its key concepts:

- **Connectors**: Manage tasks to coordinate data streaming.
- **Tasks**: Define how data is moved between Kafka and external systems.
- **Workers**: Running processes that execute connectors and tasks.
- **Converters**: Translate data between Connect and external systems.
- **Transforms**: Modify messages produced by or sent to a connector.
- **Dead Letter Queue**: Handle connector errors.

Connectors in Kafka Connect are responsible for breaking down the data copying job into tasks that can be distributed to workers.

2. Connector Development

- Connectors are implemented as Java classes that extend either `SourceConnector` or `SinkConnector`.
- The `taskClass()` method returns the class of the `Task` implementation (`SourceTask` or `SinkTask`).
- The `start()` method is called when the connector is started and initializes any necessary resources.
- The `stop()` method is called when the connector is stopped and cleans up any resources.
- The `config()` method returns the configuration definition for the connector.
- The `taskConfigs()` method generates task configurations based on the connector's configuration and the maximum number of tasks.
- `SourceTask` implementations define the `poll()` method to fetch data from the external system and return a list of source records.
- `SinkTask` implementations define the `put()` method to receive a list of sink records and write them to the external system.

3. Converters and Serialization

- Converters are used to translate between the Connect data format and the serialized form stored in Kafka.
- The `key.converter` and `value.converter` properties specify the converter classes for record keys and values, respectively.
- Common converters include `JsonConverter`, `AvroConverter`, and `StringConverter`.
- Converters can be configured with additional properties, such as `schemas.enable` to include schema information in the serialized data.

4. Single Message Transforms (SMTs)

- SMTs perform lightweight modifications to records as they flow through a connector.
- They operate on individual records and should not perform complex operations like joins or aggregations.
- SMTs are configured as part of the connector configuration using the `transforms` property.
- The `transforms` property specifies a comma-separated list of transform aliases.
- Each transform alias is configured with additional properties, such as `transforms.<alias>.type` to specify the transform class.
- Common SMTs include `MaskField`, `ReplaceField`, `ExtractField`, and `InsertField`.

5. Error Handling and Dead Letter Queues

- Connectors can define error handling behavior using the `errors.tolerance` property.
- The `errors.tolerance` property can be set to `none` (fail on any error), `all` (continue on all errors), or `transient` (continue on transient errors).

- Dead letter queue (DLQ) can be configured to capture failed records using the `errors.deadletterqueue.topic.name` property.
- The `errors.deadletterqueue.context.headers.enable` property can be set to `true` to include error context in the DLQ record headers.
- Monitoring and handling records in the DLQ is important to identify and resolve issues with connector configuration or data compatibility.

6. Offset Management

- Connectors manage offsets to track the progress of data copying.
- Source connectors use offsets to determine where to resume reading from the external system in case of failures or restarts.
- Sink connectors use offsets to track the progress of writing records to the external system.
- Offsets are stored in a Kafka topic specified by the `offset.storage.topic` property.
- The `offset.flush.interval.ms` property controls how frequently offsets are flushed to the offset storage topic.
- Offset management ensures exactly-once semantics and enables fault tolerance in Kafka Connect.

7. Exactly-Once Semantics (EOS)

- Kafka Connect supports exactly-once semantics for both source and sink connectors.
- EOS ensures that records are processed exactly once, preventing duplicates or missing data.
- To enable EOS, the following properties need to be configured:
- `enable.idempotence=true` for the Kafka producer used by the connector.
- `max.in.flight.requests.per.connection=1` to ensure records are sent in order.
- `acks=all` to require acknowledgment from all in-sync replicas.
- `transaction.timeout.ms` to set the transaction timeout for the connector.
- EOS requires careful configuration and compatibility between the connector and the external system to handle failures and ensure data consistency.

8. Schema Evolution

- Kafka Connect supports schema evolution for connectors that use schema-based serialization formats like Avro or Protobuf.
- Schema evolution allows the schema of records to change over time while maintaining compatibility with existing data.
- The Confluent Schema Registry is often used in conjunction with Kafka Connect to manage and evolve schemas.
- Connectors can be configured with the `value.converter.schema.registry.url` property to specify the URL of the Schema Registry.
- When consuming records with a schema, the connector retrieves the schema from the Schema Registry based on the schema ID stored with the record.

- Schema evolution strategies, such as backward and forward compatibility, need to be considered when making changes to record schemas.

9. Monitoring and Metrics

- Kafka Connect exposes metrics for monitoring the performance and health of connectors and tasks.
- Metrics are exposed through JMX (Java Management Extensions) and can be collected using monitoring tools like JMX exporters or Prometheus.
- Key metrics to monitor include:
- `connector-destroyed-task-count`: The number of destroyed tasks for a connector.
- `connector-failed-task-count`: The number of failed tasks for a connector.
- `connector-paused-task-count`: The number of paused tasks for a connector.
- `connector-running-task-count`: The number of running tasks for a connector.
- `connector-unassigned-task-count`: The number of unassigned tasks for a connector.
- `source-record-poll-total`: The total number of records polled by a source connector.
- `sink-record-read-total`: The total number of records read by a sink connector.
- `sink-record-send-total`: The total number of records sent by a sink connector.
- Monitoring metrics helps identify performance bottlenecks, connectivity issues, and resource utilization of Kafka Connect.

CHAPTER 7 :KAFKA STREAMS

Kafka Streams: Building Real-Time Applications and Microservices

Kafka Streams is a client library designed to build real-time applications and microservices, with data both inputted and outputted to Kafka clusters. It marries the simplicity of developing and deploying standard Java and Scala applications with the power of Kafka's server-side capabilities.

Core Concepts of Kafka Streams

- **Stream Processing**: Fundamentals of processing records in real-time from Kafka topics.
- **Topology**: A graph of stream processors (nodes) connected by streams (edges), defining the flow of data.
- **KStreams and KTables**: KStreams represent a continuous stream of data, while KTables represent a changelog stream, akin to a table in a database.

Know the Differences between KStream and KTable

- KStream processes each record as an independent event, while KTable treats each record as an update.
- KStream is useful for processing individual events, while KTable is suitable for maintaining the latest value for a given key.
- KStream supports record-by-record transformations, while KTable allows aggregations and joins based on a key.

Identify the Appropriate Use Cases

- Use KStream when you need to process each record independently, perform stateless transformations, or handle unbounded data.
- Use KTable when you need to perform aggregations, joins, or maintain a materialized view of the latest values for each key.

Stream-table duality:

- **Stream as Table:** A stream can be viewed as a changelog of a table, where each data record in the stream captures a state change of the table. A stream can be turned into a 'real' table by replaying the changelog from the beginning to reconstruct the table.
- **Table as Stream:** A table can be viewed as a snapshot of the latest value for each key in a stream. A table can be turned into a 'real' stream by iterating over each key-value entry in the table.

Fundamental Components

- **Stream Processor**: A fundamental unit that processes each incoming record, capable of transforming, filtering, or aggregating data streams.
- **State Stores**: Facilitate storing state for stateful operations, enabling functionalities like windowing and interactive queries.

Permanent State Stores in Kafka Streams:

- **Definition:** Permanent state stores are used in Kafka Streams to persistently store and manage stateful data across reboots and restarts.
- **Types:** Includes key-value stores, window stores, and session stores.
- **Usage:** Essential for operations like aggregations, joins, and windowing which require maintaining state over time.
- **Configuration:** Defined in the Kafka Streams API using `Stores.persistentKeyValueStore()`, `Stores.persistentWindowStore()`, etc.
- **Resource Management:** Always close Iterators after use to prevent resource leaks and Out-Of-Memory (OOM) issues.
- **Recovery:** State stores support fault tolerance by restoring state from changelog topics on restart.
- **Performance:** Performance can be optimized by configuring RocksDB, the default storage engine for state stores.
- **Scaling:** State stores can be scaled across multiple instances using partitioning and sharding.
- **Monitoring:** Monitor metrics related to state stores, such as cache hit rates and store sizes, to ensure efficient operation.

Key Features

- **Time Windowing**: Execute operations on data within specific time frames, crucial for temporal data analysis.
- **Stateful Operations**: Perform computations that require maintaining a state, such as joins, aggregations, and windowed computations.
- **Exactly-Once Semantics**: Ensure each record is processed exactly once, crucial for fault-tolerant processing.

Developing with Kafka Streams

Building a Stream Processing Application

Steps include setting up the project, defining the topology, and integrating stream processors with Kafka topics.

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> textLines = builder.stream("inputTopic");
KTable<String, Long> wordCounts = textLines
    .flatMapValues(textLine -> Arrays.asList(textLine.toLowerCase().split("\\W+")))
    .groupBy((key, word) -> word)
    .count(Materialized.as("countsStore"));
```

```
wordCounts.toStream().to("outputTopic", Produced.with(Serdes.String(), Serdes.Long()));
```

Application Configuration

Critical configurations for Kafka Streams applications include the application ID, bootstrap servers, and key-value serializers/deserializers.

Advanced Concepts and Operations

Stateless and Stateful Operators

- **Stateless**: Operations like `map`, `filter`, and `foreach` that don't maintain state.
- **Stateful**: Operations such as `join`, `aggregate`, and `count`, which maintain state over time or across keys.

Window Types

- **Tumbling Window**: Fixed-size, non-overlapping, time-based windows.
- **Hopping Window**: Fixed-size, overlapping time windows.
- **Sliding Window**: Overlapping windows based on time difference between records.
- **Session Window**: Dynamically sized, non-overlapping windows based on activity sessions.

SerDes and Streams DSL

- **SerDes**: Serialization and Deserialization frameworks essential for data interpretation in Kafka Streams.
- **Streams DSL**: High-level domain-specific language to define stream processing topologies, including `KStream`, `KTable`, and `GlobalKTable`.

Join Operations

Using the Streams API, the following join operations are supported, with joins being either windowed or non-windowed depending on the operands:

- **KStream-to-KStream (windowed):** Supports inner join, left join, and outer join.
- **KTable-to-KTable (non-windowed):** Supports inner join, left join, and outer join.
- **KStream-to-KTable (non-windowed):** Supports inner join and left join. Outer join is not supported.
- **KStream-to-GlobalKTable (non-windowed):** Supports inner join and left join. Outer join is not supported.

Message Delivery Guarantees

Kafka supports three types of message delivery guarantees:

- **At-most-once:** Every message is persisted in Kafka at most once. Message loss is possible if the producer doesn't retry on failures.

- **At-least-once:** Every message is guaranteed to be persisted in Kafka at least once. There is no chance of message loss, but messages can be duplicated if the producer retries after the message is already persisted.
- **Exactly-once:** Every message is guaranteed to be persisted in Kafka exactly once without any duplicates or data loss, even in the event of broker failures or producer retries.

Exactly-once processing can be achieved for Kafka-to-Kafka workflows using the Kafka Streams API. For Kafka-to-sink workflows, an idempotent consumer is required.

Stream processing applications written with the Kafka Streams library can enable exactly-once semantics by setting the `processing.guarantee` configuration to `exactly_once` (the default value is `at_least_once`). This change requires no code modifications.

Co-Partitioning

Co-partitioning is a concept in Kafka where two or more topics have their partitions aligned in such a way that the same partition numbers across these topics contain related data. This alignment is crucial for operations that involve joining data streams, ensuring that related data from different topics can be processed together efficiently. Here are the key rules and considerations for achieving co-partitioning in Kafka:

1. **Same Number of Partitions**: Co-partitioned topics must have the same number of partitions. This ensures that data which is logically related and keyed similarly ends up in the corresponding partition across these topics.
2. **Consistent Partitioning Strategy**: To ensure that related messages from different topics land in the corresponding partitions, a consistent partitioning strategy must be used. This often involves ensuring that messages are produced with the same key and that the default partitioner or a custom partitioner is used consistently across these topics.
3. **Parallel Consumer Processing**: When consuming from co-partitioned topics, ensure that the consumer groups are configured to consume in parallel from all the related partitions. This is typically handled naturally by Kafka's consumer group mechanism when the topics are consumed together.
4. **Kafka Streams and KTable**: In Kafka Streams, co-partitioning is essential when performing join operations between KStreams and KTables or between KStreams themselves. Kafka Streams applications will automatically co-partition the data of joined streams by repartitioning them as needed, which may introduce additional processing steps and topics.
5. **Use Case for Co-Partitioning**: Co-partitioning is often used in scenarios requiring data aggregation or join operations across different data streams that share a logical relationship, such as aggregating user click events with user profile information for real-time analytics.
6. **Management and Maintenance**: When scaling out (adding more partitions to topics), ensure that all co-partitioned topics are scaled consistently. This might require manual intervention or custom tooling, as Kafka does not automatically manage the partition count across topics to maintain co-partitioning.
7. **Producing to Co-Partitioned Topics**: When producing messages to co-partitioned topics, ensure that the keys used for partitioning are consistent and that messages destined to be joined together use the same keys.

Parallel processing

Kafka Streams scales by allowing multiple threads of execution within one instance of the application and by supporting load balancing between distributed instances of the application. You can run the Streams application on one machine with multiple threads or on multiple machines; in either case, all active threads in the application will balance the data processing work.

The Streams engine parallelizes the execution of a topology by splitting it into tasks. The number of tasks is determined by the Streams engine and depends on the number of partitions in the topics that the application processes. Each task is responsible for a subset of the partitions: the task subscribes to those partitions and consumes events from them. For every event it consumes, the task executes all the processing steps that apply to this partition in order before eventually writing the result to the sink. These tasks are the basic unit of parallelism in Kafka Streams, as each task can execute independently of others.

To achieve the maximum parallelism in a Kafka Streams application, you can set the `num.stream.threads` configuration parameter to the desired parallelism, **not necessarily the number of partitions**.

Repartitioning

When Kafka Streams requires repartitioning, it:

1. Writes the repartitioned data to a new topic with new keys and partitions.
2. Creates a new set of tasks to read and process events from the new topic.

This process divides the topology into two subtopologies, each with its own tasks. The second subtopology depends on the output of the first.

The two sets of tasks can still run independently and in parallel because:

- The first set writes data to the new topic at its own pace.
- The second set consumes and processes events from the new topic independently.

There is no communication or shared resources between the tasks, allowing them to run on separate threads or servers.

Understand the Available Operations

- KStream operations:
- `map`: Applies a one-to-one transformation to each record in the stream. Example:

```
KStream<String, Integer> doubled = stream.map((key, value) -> new KeyValue<>(key,
value * 2));
```

- `filter`: Filters records based on a predicate. Example:

```
KStream<String, Integer> filtered = stream.filter((key, value) -> value > 100);
```

- `flatMap`: Applies a one-to-many transformation to each record, flattening the result into a new stream. Example:

  ```
  KStream<String, String> words = stream.flatMapValues(value -> Arrays.asList(value.split("\\s+")));
  ```
- `branch`: Splits a stream into multiple streams based on predicates. Example:

  ```
  KStream<String, Integer>[] branches = stream.branch((key, value) -> value > 100, (key, value) -> value <= 100);
  ```
- `merge`: Merges multiple streams into a single stream. Example:

  ```
  KStream<String, Integer> merged = stream1.merge(stream2);
  ```
- KTable operations:
- `aggregate`: Performs an aggregation on the records of a KTable. Example:

  ```
  KTable<String, Long> counts = table.groupByKey().aggregate(() -> 0L, (aggKey, newValue, aggValue) -> aggValue + newValue);
  ```
- `reduce`: Combines the records of a KTable based on a reducer function. Example:

  ```
  KTable<String, Integer> reduced = table.groupByKey().reduce((value1, value2) -> value1 + value2);
  ```
- `join`: Joins two KTables based on their keys. Example:

  ```
  KTable<String, String> joined = table1.join(table2, (value1, value2) -> value1 + "-" + value2);
  ```
- `groupBy`: Groups the records of a KTable based on a new key. Example:

  ```
  KTable<String, Integer> grouped = table.groupBy((key, value) -> value % 10);
  ```
- `count`: Counts the number of records in a KTable. Example:

  ```
  KTable<String, Long> counts = table.groupByKey().count();
  ```

Be Familiar with the Syntax and APIs

- Creating a KStream:

```
KStream<String, String> stream = builder.stream("input-topic");
```

- Creating a KTable:

```
KTable<String, String> table = builder.table("input-topic");
```

- Performing transformations:

```
KStream<String, Integer> transformed = stream.mapValues(value -> value.length());
```

- Aggregating data:

```
KTable<String, Long> aggregated = stream.groupByKey().count();
```

- Joining streams and tables:

```
KStream<String, String> joined = stream.leftJoin(table, (streamValue, tableValue) ->
streamValue + "-" + tableValue);
```

Best Practices for Kafka Streams

- **State Management**: Leverage state stores and interactive queries efficiently to manage state.
- **Scaling and Performance**: Apply strategies to scale applications horizontally and optimize performance through tuning and configuration adjustments.
- It is recommended to store Kafka Streams application state on a **persistent volume** to minimize recovery time by only restoring the missing state upon failure or restart.

Kafka Streams offers a comprehensive set of tools and functionalities for building robust, real-time streaming applications, leveraging Kafka's strengths in handling vast streams of data efficiently and reliably.

CHAPTER 8 : MONITORING METRICS

Monitoring and Metrics in Kafka

Monitoring and Metrics in Kafka Kafka exposes rich metrics to monitor the health, performance, and utilization metrics of Kafka brokers, producers, and consumers. The ability to obtain these metrics quickly and at scale is vital for troubleshooting and optimizing Kafka cluster .

Points to Remember for CCDAK on Monitoring and Metrics

- **JMX Integration**: Kafka metrics can be exposed via JMX, allowing integration with monitoring tools.
- **Custom Reporters**: Kafka supports pluggable metrics reporters for flexibility in how metrics are reported and consumed.
- **Performance Tuning**: Metrics are key to identifying bottlenecks and tuning Kafka for optimal performance.
- **Operational Insight**: Monitoring provides insights into Kafka's operational status, including throughput, latency, and resource utilization.

Important Monitoring and Metrics Properties

`kafka.metrics.reporters`

- **Default**: ""
- **Description**: Comma-separated list of classes to report Kafka metrics. These reporters must implement the `kafka.metrics.KafkaMetricsReporter` interface.
- **Trade-offs**: Adding custom reporters can enhance monitoring capabilities but might introduce additional overhead.

`metric.reporters`

- **Default**: ""
- **Description**: Comma-separated list of classes to report metrics from producers or consumers. These reporters implement the `org.apache.kafka.common.metrics.MetricsReporter` interface.
- **Trade-offs**: Enables detailed monitoring of producer and consumer metrics, with the trade-off of potential performance impact due to monitoring overhead.

`metrics.num.samples`

- **Default**: 2
- **Description**: The number of samples maintained to compute metrics.
- **Trade-offs**: Higher values provide more accurate metrics but can consume more memory.

`metrics.sample.window.ms`

- **Default**: 30000 (30 seconds)
- **Description**: The time window associated with each sample. Affects how metrics are calculated over time.
- **Trade-offs**: Larger windows provide smoother metrics over time but may delay detection of spikes or drops in metric values.

`metrics.recording.level`

- **Default**: INFO
- **Description**: The highest recording level for metrics in clients. Options are INFO or DEBUG.
- **Trade-offs**: DEBUG provides more granular metrics but can significantly increase the amount of data collected, impacting performance and storage.

Broker Metrics:

- `UnderReplicatedPartitions`: Number of partitions that are under-replicated. Use case: Identifying replication issues. Alert if greater than 0.
- `ActiveControllerCount`: Number of active controllers in the cluster. Use case: Ensuring controller failover is working correctly. Should be 1. Alert if not 1. (The **unique** controller takes care of Leader Election, Cluster Metadata, Rebalancing Partitions and Handling Broker Failures)
- `OfflinePartitionsCount`: Number of partitions without an active leader. Use case: Detecting availability issues. Should be 0. Alert if greater than 0.
- `RequestHandlerAvgIdlePercent`: Average idle percentage of the request handler thread pool. Use case: Monitoring request handling capacity. Low values may indicate resource contention.
- `NetworkProcessorAvgIdlePercent`: Average idle percentage of the network thread pool. Use case: Monitoring network processing capacity. Low values suggest potential network bottlenecks.
- `IsrExpandsPerSec`: Rate at which the in-sync replica set is expanding. Use case: Tracking replica synchronization.
- `IsrShrinksPerSec`: Monitors the rate at which replicas are removed from the ISR list per second. Use case: Tracking replica synchronization issues.
- `BytesInPerSec` and `BytesOutPerSec`: Inbound and outbound byte rate of brokers. Use case: Monitoring data throughput.
- `MessagesInPerSec`: Rate at which messages are being produced to brokers. Use case: Tracking message production rate.
- `UnderMinIsrPartitionCount`: Number of partitions with insufficient in-sync replicas. Use case: Identifying replication issues.
- `PartitionCount`: Total number of partitions across all topics in the cluster. Use case: Monitoring cluster capacity.
- `LeaderCount`: Number of partitions for which a broker is the leader. Use case: Monitoring leadership distribution.
- `RequestQueueSize` and `ResponseQueueSize`: Size of request and response queues. Use case: Ensuring brokers can keep up with client requests.
- `ProduceTotalTimeMs` and `FetchTotalTimeMs`: Total time taken for produce and fetch requests. Use case: Monitoring request latency.

- `TotalTimeMs`: Tracks the total time taken for request processing. High values may indicate performance issues. Use case: Monitoring overall request latency.
- `FetchMessageConversionsPerSec`: Rate of message format conversions during fetch requests. Use case: Tracking message format compatibility.
- `ZooKeeperRequestLatencyMs`: Latency of ZooKeeper requests made by Kafka. Use case: Monitoring ZooKeeper performance.
- `ControllerState`: Indicates whether a broker is currently the active controller (1 if active, 0 if not). Use case: Monitoring controller health.
- `LogFlushRateAndTimeMs`: Monitors the rate and time of log flushes. High values can indicate I/O bottlenecks. Use case: Identifying disk performance issues.
- `LeaderElectionRateAndTimeMs`: Monitors the rate and time taken for leader election. High values may indicate issues with the controller or network. Use case: Tracking controller performance and stability.
- `JvmMemoryUsage`: Tracks the memory usage of the JVM, helping to identify potential memory leaks or insufficient heap sizes. Use case: Monitoring broker memory utilization.
- `GarbageCollectionTime`: Monitors the time spent in garbage collection. High values can indicate potential performance issues. Use case: Identifying JVM garbage collection overhead.
- `ZooKeeperDisconnectsPerSec`: Tracks the rate of disconnects from ZooKeeper. Use case: Identifying ZooKeeper connection issues.

Consumer Metrics:

- `records-lag-max`: Maximum lag in terms of number of records for any partition in this window. Use case: Monitoring consumer lag.
- `fetch-latency-avg`: Average time taken for a fetch request. Use case: Monitoring consumer performance.
- `records-consumed-rate`: The average number of records consumed per second. Use case: Tracking consumer throughput.
- `FetchRequestsPerSec`: Tracks the number of fetch requests from consumers per second. Use case: Monitoring consumer activity.
- `ConsumerLag`: Monitors the lag of consumers to track how many messages the consumer is behind.

Producer Metrics:

- `record-send-rate`: The average number of records sent per second. Use case: Monitoring producer throughput.
- `request-latency-avg`: The average request latency in ms. Use case: Monitoring producer latency.
- `outgoing-byte-rate`: The average number of outgoing bytes sent per second to all servers. Use case: Monitoring producer network utilization.
- `ProduceRequestsPerSec`: Tracks the number of produce requests from producers per second. Use case: Monitoring producer activity.

Topic Metrics:

- `BrokerTopicMetrics`: Monitors topic-specific metrics such as bytes in/out, messages in/out, etc. Use case: Analyzing performance at the topic level.

Confluent-Specific Metrics:

- Schema Registry: `schema-registry.registered.count`: Number of registered schemas. Use case: Monitoring schema usage.
- Kafka Connect: `connect-worker.connector-started-task-count`: Number of tasks started for a connector. Use case: Monitoring connector health.
- ksqlDB: `ksql-server.query-stats.bytes-consumed-total`: Total bytes consumed by ksqlDB queries. Use case: Monitoring ksqlDB resource usage.

Important Kafka Metrics

`UnderReplicatedPartitions`

- **Description**: Represents the number of partitions that are under-replicated, meaning they have fewer in-sync replicas (ISRs) than the desired replication factor.
- **Importance**: This metric is crucial for monitoring the health and reliability of the Kafka cluster. A non-zero value indicates that some partitions are at risk of data loss if a broker fails.
- **Monitoring Recommendations**:
- Alert if the value is consistently greater than 0.
- Investigate the root cause of under-replication, such as broker failures, network issues, or insufficient disk space.
- Ensure that the replication factor is set appropriately based on the desired fault tolerance.

`ActiveControllerCount`

- **Description**: Indicates the number of active controllers in the Kafka cluster. In a healthy cluster, there should be exactly one active controller.
- **Importance**: The controller is responsible for managing partition leader elections and maintaining the overall state of the cluster. Having multiple active controllers can lead to conflicts and instability.
- **Monitoring Recommendations**:
- Alert if the value deviates from 1.
- If multiple controllers are detected, investigate the root cause, such as network partitions or issues with ZooKeeper.

`OfflinePartitionsCount`

- **Description**: Represents the number of partitions without an active leader. These partitions are unavailable for producing or consuming messages.
- **Importance**: Offline partitions indicate a problem with the Kafka cluster and can impact data availability and processing.
- **Monitoring Recommendations**:

- Alert if the value is greater than 0.
- Investigate the reasons for offline partitions, such as broker failures, network issues, or insufficient replicas.

`BytesInPerSec` and `BytesOutPerSec`

- **Description**: Measures the inbound and outbound byte rate of Kafka brokers, indicating the amount of data being produced to and consumed from the cluster.
- **Importance**: Monitoring these metrics helps in understanding the data throughput and identifying potential bottlenecks or performance issues.
- **Monitoring Recommendations**:
- Set appropriate thresholds based on the expected data volume and network capacity.
- Monitor trends over time to detect anomalies or sudden spikes in data throughput.
- Correlate with other metrics like `MessagesInPerSec` and `ProduceRequestsPerSec` to gain insights into producer and consumer behavior.

`ConsumerLag`

- **Description**: Represents the lag of consumers, indicating how many messages they are behind the latest offset in the partition they are consuming from.
- **Importance**: Consumer lag is a critical metric for monitoring the health and performance of Kafka consumers. High consumer lag can indicate issues like slow processing, insufficient consumer resources, or problems with the consumer application.
- **Monitoring Recommendations**:
- Set appropriate lag thresholds based on the acceptable delay in message processing.
- Alert if the lag exceeds the defined thresholds consistently.
- Investigate the root cause of high consumer lag, such as slow processing logic, resource contention, or network issues.

`ProduceTotalTimeMs` and `FetchTotalTimeMs`

- **Description**: Measures the total time taken for produce and fetch requests, respectively. These metrics provide insights into the latency of producer and consumer operations.
- **Importance**: Monitoring these metrics helps in identifying performance bottlenecks and ensuring that produce and fetch operations are within acceptable latency ranges.
- **Monitoring Recommendations**:
- Set appropriate latency thresholds based on the requirements of the application.
- Alert if the latency consistently exceeds the defined thresholds.
- Analyze trends over time to detect degradation in performance and investigate the underlying causes.

CHAPTER 9 : PRODUCER

Kafka Producer Configurations

Kafka producers send records to topics. The efficiency, reliability, and performance of data production can be significantly influenced by how producers are configured.

Key Points for CCDAK on Producers

- **Efficiency and Throughput**: Configurations like batch size and compression type can greatly affect the throughput.
- **Data Integrity**: Settings such as `acks` and `retries` ensure data integrity and delivery guarantees.
- **Latency**: Configurations like `linger.ms` can be tuned to balance between latency and throughput.
- **Reliability**: Features like idempotence and transactions ensure reliable data delivery.
- **Simplicity:** Producers are conceptually simpler than consumers since they do not require group coordination.
- **Broker and Partition Awareness:** Producers automatically know which broker and partition to write to.
- **Partitioning:** A producer partitioner maps each message to a topic partition, and the producer sends a produce request to the leader of that partition.
- **Message Key Guarantee:** The partitioners shipped with Kafka ensure that all messages with the same non-empty key are sent to the same partition.
- **Round-Robin Sending:** If producers send data without a key, the data is sent round-robin to all available brokers.
- **Acknowledgements:** Producers can choose to receive acknowledgements of data writes.

Important Producer Properties

`acks`

- **Default**: 1
- **Description**: The number of acknowledgments the producer requires from brokers. Options are 0, 1, or 'all'.
- **Trade-offs**: More `acks` increase reliability but may reduce throughput.

`linger.ms`

- **Default**: 0
- **Description**: The delay to aggregate batches of records. Higher values allow larger and more efficient batches.
- **Trade-offs**: Increases latency but improves throughput and efficiency.

`batch.size`

- **Default**: 16384 (16KB)
- **Description**: Maximum batch size in bytes. Larger batches are more efficient but take more memory and may delay transmission.
- **Trade-offs**: Balances memory usage with throughput.

`max.inflight.requests.per.connection`

- **Default**: 5
- **Description**: The maximum number of unacknowledged requests the client will send on a single connection before blocking.
- **Trade-offs**: Higher values can improve throughput but may compromise ordering guarantees, especially with retries.

`enable.idempotence`

- **Default**: false
- **Description**: Ensures exactly-once delivery by preventing duplicate records.
- **Trade-offs**: May slightly impact throughput but significantly increases reliability.

`compression.type`

- **Default**: none
- **Description**: The compression type for all data generated by the producer. Options are 'gzip', 'snappy', 'lz4', and 'zstd'.
- **Trade-offs**: Compression reduces the size of data sent over the network at the cost of CPU usage.

`retries`

- **Default**: 2147483647 (Integer.MAX_VALUE)
- **Description**: Setting a high value allows Kafka to retry indefinitely on transient errors.
- **Trade-offs**: Ensures delivery at the potential cost of ordering if `max.inflight.requests.per.connection` is greater than 1.

`transaction.timeout.ms`

- **Default**: 60000 (1 minute)
- **Description**: Maximum time a transaction can remain open. Relevant for exactly-once semantics.
- **Trade-offs**: Longer transactions can hold resources but ensure completeness of operations.

`max.request.size`

- **Default**: 1048576 (1MB)
- **Description**: The maximum size of a request. Limits the size of a message that can be sent.
- **Trade-offs**: Larger sizes can improve throughput but risk overwhelming brokers or dropping connections on large messages.

`buffer.memory`

- **Default**: 33554432 (32MB)
- **Description**: Total memory available to the producer for buffering.
- **Trade-offs**: More memory allows more batching and in-flight messages, improving throughput but increasing memory usage.

`transactional.id`

- **Default**: null
- **Description**: Unique identifier for transactional messages. Necessary for exactly-once semantics.
- **Trade-offs**: Enables transaction support at the cost of additional overhead for maintaining state.

Handling Errors in Kafka Producers

Kafka classifies errors as *retriable* or *non-retriable*, affecting how producers react to them.

Retriable Errors

Retriable errors are temporary and often resolved through retries. Kafka producers automatically retry these errors up to the `retries` limit. Common retriable errors include:

- **Leader Not Available**: Occurs when a new leader is being elected. The producer automatically recovers.
- **Not Leader for Partition**: The targeted leader is not the current leader for the partition. The producer automatically recovers.
- **Not Enough Replicas**: Not enough replicas are available for the producer to satisfy the `acks` configuration.

Producers handle these errors by:

- Retrying with a delay specified by `retry.backoff.ms`.
- Continuing retries up to the `retries` limit.

Non-Retriable Errors

Non-retriable errors indicate issues not resolved by retrying and often require configuration changes or other interventions:

- **Message Too Large**: The message size exceeds `max.request.size`. Requires adjusting the message size or configuration.
- **Invalid Topic Exception**: The specified topic is invalid or does not exist.
- **Topic Authorization Failed**: The producer lacks permission to publish to the specified topic.
- **Offset Out of Range**: The specified offset is not within the range of the topic.

- **Broker Not Available**: The broker is not available for connections.
- **Invalid Required Acks**: The `acks` configuration is set to an invalid value.

Non-retriable errors demand an intervention or change in configuration or permissions. They point to conditions that retries alone cannot overcome, often requiring administrative action or code changes.

Handling non-retriable errors requires:

- Catching and logging the exception.
- Reviewing the error to understand its cause.
- Adjusting the configuration or addressing the permission issues as needed.

Best Practices for Error Handling

- **Monitoring**: Continuously monitor log files for error patterns to proactively address emerging issues.
- **Smart Retrying**: Use the `retry.backoff.ms` setting wisely to avoid overwhelming the system.
- **Idempotent Producers**: Enable `enable.idempotence` to ensure that messages are not duplicated in the event of retries.

CHAPTER 10 : REST PROXY

Kafka REST Proxy Overview

The Kafka REST Proxy provides a RESTful interface to a Kafka cluster, enabling the production and consumption of Kafka messages over HTTP. It's a critical component for integrating non-JVM applications with Kafka.

Key Points for CCDAK on REST Proxy

- **Accessibility**: Allows applications that cannot use Kafka's native client libraries to interact with Kafka topics via HTTP requests.
- **Integration**: Simplifies integration with web-based applications and services, enabling a broader range of technologies to connect with Kafka.

Important Features of Kafka REST Proxy

HTTP-Based Interaction

- **Scope**: Client-Server Communication
- **Description**: Facilitates the production and consumption of messages through standard HTTP methods, making Kafka accessible from any language that can make HTTP requests.
- **Impact**: Broadens Kafka's usability beyond traditional back-end systems to front-end and external systems without native Kafka support.

Consumer Group Management

- **Scope**: Consumption
- **Description**: Supports the creation and management of consumer groups via RESTful endpoints, allowing detailed control over message consumption.
- **Impact**: Enables efficient message processing and load balancing across multiple consumers using standard web technologies.

Essential REST Proxy Configuration Parameters

Proxy Server Settings

- `listeners`: Configures the network address the REST Proxy listens on, defining how clients connect to the proxy.
- `host.name`: Specifies the host name advertised in the producer and consumer configurations, important for ensuring that the REST Proxy is reachable.

Security and Authentication

- `authentication.method`: Determines the authentication mechanism, such as basic or OAuth, enhancing security by controlling access to the Kafka REST Proxy.
- `ssl.enabled`: Enables SSL/TLS for the REST Proxy, securing data in transit between clients and the proxy.

Advanced REST Proxy Configurations for Fine-Tuning

Request and Consumer Settings

- `consumer.request.timeout.ms`: Sets the timeout for consumer instances to close automatically if no further requests are received. Helps manage server load and resource allocation.
- `consumer.instance.max.age.ms`: Configures the maximum lifespan of a consumer instance, ensuring that resources are freed if the consumer is no longer active.

Producing and Consuming Enhancements

- `simple.consumer.pool.size.max`: Determines the maximum number of simple consumer instances that can be created, balancing load and performance in the REST Proxy.
- `fetch.min.bytes`: Sets the minimum amount of data that the server should return for fetch requests, optimizing network and application performance.

Key Considerations for Scalability and Reliability

Load Balancing

- Utilizing multiple REST Proxy instances behind a load balancer can help distribute client requests evenly, improving throughput and reducing latency.

High Availability

- Deploying REST Proxy instances in different physical locations or availability zones can increase fault tolerance and ensure continuous availability of services.

Monitoring and Management

- Monitoring the REST Proxy's performance and logs is crucial for maintaining an efficient and reliable system, enabling proactive management of potential issues.

Additional Points for CCDAK on REST Proxy

Data Formats

- The REST Proxy supports various data formats for producing and consuming messages, including JSON, Avro, and binary.
- It integrates with the Confluent Schema Registry for seamless Avro serialization and deserialization.

- The REST Proxy requires to receive data over REST that is already base64 encoded, hence it is the responsibility of the producer to encode the data.

Authentication and Authorization

- The REST Proxy supports multiple authentication mechanisms, such as Basic Auth, OAuth, and SSL/TLS, for secure access control.
- It integrates with Kafka's ACLs (Access Control Lists) for fine-grained authorization control over topics and operations.

Error Handling and Response Codes

- The REST Proxy handles errors and returns appropriate HTTP response codes for different scenarios, such as invalid requests, authentication failures, or Kafka-related errors.
- Clients can handle and interpret these response codes to effectively deal with errors and take appropriate actions.

Performance Considerations

- Using the REST Proxy introduces some performance overhead compared to using Kafka's native client libraries directly.
- It's important to consider the trade-offs between ease of use and performance when deciding to use the REST Proxy.

Monitoring and Metrics

- The REST Proxy provides metrics and monitoring endpoints for tracking its health and performance.
- These metrics can be integrated with monitoring tools and dashboards for effective monitoring and alerting.

Integration with Confluent Platform

- The REST Proxy seamlessly integrates with other components of the Confluent Platform, such as Confluent Control Center, for centralized management and monitoring.
- The Confluent Platform offers additional features and benefits when using the REST Proxy, enhancing the overall Kafka experience.

CHAPTER 11 : SCHEMA REGISTRY

Schema Registry and Avro Overview

Confluent Schema Registry and Avro serialization provide a robust way to ensure that Kafka messages conform to a schema, enabling schema evolution and ensuring data compatibility. Understanding these components is essential for building resilient and forward-compatible Kafka applications.

Schema Registry Key Concepts

- **Centralized Schema Management**: Schema Registry stores and manages Avro, JSON Schema, and Protobuf schemas, ensuring that producers and consumers agree on schema formats.
- **Schema Evolution**: Supports schema evolution with backward, forward, and full compatibility checks, preventing breaking changes.
- **Integration with Kafka**: Iintegrates with Kafka, enabling schemas to be applied to Kafka messages, ensuring data consistency across the system.

The following schema formats are supported out-of-the-box with Confluent Platform, with serializers, deserializers, and command line tools available for each format:

- **Avro:** Relies on schemas. When Avro data is read, the schema used when writing it is always present.
- **JSON:** A lightweight data-interchange format. It's easy for humans to read and write, and it's easy for machines to parse and generate. JSON does not rely on schemas.
- **ProtoBuf:** Protocol Buffers (ProtoBuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.

Avro

- **Binary Format**: Avro is a compact binary format that enables efficient serialization and deserialization of data.
- **Schema Evolution**: Avro supports schema evolution, allowing schemas to be updated without breaking existing applications.

The set of primitive types:

- null: no value
- boolean: a binary value
- int: 32-bit signed integer
- long: 64-bit signed integer
- float: single precision (32-bit) IEEE 754 floating-point number

- double: double precision (64-bit) IEEE 754 floating-point number
- bytes: sequence of 8-bit unsigned bytes
- string: unicode character sequence

Avro supports six types of complex data structures: records, enums, arrays, maps, unions, and fixed.

Records are defined with the type name "record" and include the following attributes:

- **name:** A required JSON string that specifies the record's name.
- **namespace:** A JSON string that qualifies the record's name.
- **doc:** An optional JSON string that provides documentation for the schema.
- **aliases:** An optional JSON array of strings that provides alternate names for the record.
- **fields:** A required JSON array that lists the fields of the record.

Key Advantages of Schema Registry and Avro

Centralized Schema Management

- The Schema Registry serves as a centralized repository for storing Avro, JSON Schema, and Protobuf schemas, facilitating schema sharing and enforcement across producers and consumers.

Schema Evolution Support

- Both Schema Registry and Avro support schema evolution, allowing schemas to be updated while maintaining compatibility, thus preventing breaking changes.

Backward compatibility: Focus on what the **new system can handle from the old**. It’s about not breaking old producers or data. Forward compatibility: Concentrate on what the **old system can handle from the new**. It’s about not breaking old consumers with new data.

- **Backward**: New schema can read data written in old schema. This means your applications can read old data even after the schema has been updated. Adding optional fields with defaults, compatible with existing data and code.

1. Removing a field
2. Making a required field optional
3. Widening the range of an integer or long field

- **Forward**: Old schema can read data written in new schema. This allows applications using an older schema to read data that was produced with a newer schema. Removing optional fields, compatible with existing data and code that doesn't use the removed fields.

1. Adding a new optional field
2. Adding a new required field with a default value

- **Breaking**: Such as removing required fields, changing field types incompatibly, or adding new required fields.

Mnemonic: "CF-PS" (Consumer First, Producer Second)

In most cases, it's safer to update the consumer first and then the producer. This approach ensures that the consumer can handle both the old and new schema versions, minimizing the risk of disruption. To remember the general rule for updating consumers and producers, use the mnemonic "CF-PS":

- **C (Consumer)**: Update the Consumer first
- **F (Forward)**: Consumer can handle forward compatible changes
- **P (Producer)**: Update the Producer second
- **S (Special)**: In Special cases, like **backward compatible** changes or urgent fixes, update the producer first

**Enums: ** Adding enum symbols is backward-compatible. Removing enum symbols is a breaking change. Reordering enum symbols is a breaking change. "Add, Remove (break), Reorder (break)" (ARR) Since Avro 1.9.1 (>= 1.9.1): Default value for enums. Writer's symbol not in reader's enum => Use default value if specified, else error.

Compatibility	Change
Backward	Delete fields
Backward	Add optional fields
Forward	Add fields
Forward	Delete optional fields
Full	Add optional fields
Full	Delete optional fields
Transitive	Backward/Forward/Full compatibility

Backward: Deleting fields, Adding optional fields - Upgrade first: Consumers. (Default for topics, for Streams only Backward is allowed) Forward: Adding fields, Deleting optional fields - Upgrade first: Producers.

Efficient Data Serialization

- Avro provides a compact, fast, binary data format, significantly reducing the payload size and improving data serialization and deserialization efficiency.

Implementing Schema Registry and Avro

Schema Registration

- Register schemas via the Schema Registry's REST API to enforce schema consistency across messages.
- `# Command to register a schema`
- `curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \`
- `    --data '{ "schema": "{\"type\":\"record\",\"name\":\"Test\",\"fields\":[{\"name\":\"field1\",\"type\":\"string\"}]}" }' \`
- `    http://localhost:8081/subjects/test-topic-value/versions`

Avro in Kafka Producers and Consumers

- Utilize Avro serializers and deserializers in Kafka clients, integrating seamlessly with the Schema Registry for schema management.

Essential Schema Registry and Avro Operations

Viewing and Managing Schemas

- List registered schemas or test schema compatibility using the Schema Registry's REST API.
- `# List all subjects`
- `curl -X GET http://localhost:8081/subjects`
- `# Test schema compatibility`
- `curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \`
- `    --data '{ "schema": "{\"type\":\"record\",\"name\":\"TestNew\",\"fields\":[{\"name\":\"field1\",\"type\":\"string\"}]}" }' \`
- `    http://localhost:8081/compatibility/subjects/test-topic-value/versions/latest`

Avro Serialization Details

- **Data Types**: Supports both primitive (e.g., int, string) and complex types (e.g., enums, arrays, maps), enabling diverse data representation.
- **Schema Evolution**: Avro allows for backward, forward, and full compatibility modes, facilitating flexible schema updates.

Schema Registry Integration with Kafka

- **Schema Storage**: Schemas are stored in a Kafka topic (`_schemas`), managed by the Schema Registry, separating schema management from data storage.
- **Security and Efficiency**: Reduces message size by sending schema IDs instead of full schemas with each message, while also supporting secure client communication protocols (HTTP, HTTPS).

Important Operations and Concepts

- **Registering Schemas**:
- `# Example command to register a schema using the Schema Registry REST API`
- `curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \`

- `  --data '{ "schema": "{\"type\":\"record\",\"name\":\"Test\",\"fields\":[{\"name\":\"field1\",\"type\":\"string\"}]}" }' \`
- `  http://localhost:8081/subjects/test-topic-value/versions`
- **Schema Compatibility**: Configuring compatibility settings in Schema Registry to ensure that schema updates are compatible with consumer expectations.
- **Using Avro with Kafka Producers and Consumers**: Implementing Kafka producers and consumers using Avro serialization requires using specific serializers and deserializers that integrate with Schema Registry.

Useful Commands

- **Viewing Registered Schemas**:
- `# List all subjects`
- `curl -X GET http://localhost:8081/subjects`
- **Checking Schema Compatibility**:
- `# Test compatibility of a schema with the latest schema under a subject`
- `curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \`
- `  --data '{ "schema": "{\"type\":\"record\",\"name\":\"TestNew\",\"fields\":[{\"name\":\"field1\",\"type\":\"string\"}]}" }' \`
- `  http://localhost:8081/compatibility/subjects/test-topic-value/versions/latest`

Schema Registry In-Depth

1. Overview

- Schema Registry is a centralized service for storing and managing schemas used for serializing and deserializing data in Kafka.
- It provides a RESTful API for storing and retrieving schemas, as well as enforcing schema compatibility rules.
- Schema Registry integrates with Kafka clients, allowing them to retrieve schemas for serialization and deserialization.
- It supports multiple serialization formats, including Avro, Protobuf, and JSON Schema.
- Schema Registry ensures data compatibility between producers and consumers by:
- Storing a schema for each unique topic-key or topic-value combination.
- Allowing producers and consumers to retrieve schemas based on the topic and key/value type.
- Enforcing compatibility rules to ensure that producers and consumers use compatible schemas.
- Rejecting incompatible schemas and ensuring that data can be deserialized correctly by consumers.

2. Schema Compatibility

- Schema compatibility ensures that data serialized with an older schema can be deserialized using a newer schema.

- Schema Registry supports four compatibility types:
- `BACKWARD`: Consumers using the new schema can read data produced with the last schema.
  - Allows adding new fields to the schema.
  - Allows making a previously required field optional.
  - Allows removing a field with a default value.
- `FORWARD`: Consumers using the last schema can read data produced with the new schema.
  - Allows removing fields from the schema.
  - Allows making a previously optional field required.
- `FULL`: Both `BACKWARD` and `FORWARD` compatibilities are maintained.
  - Allows adding new optional fields.
  - Allows removing fields with default values.
- `NONE`: No compatibility checks are performed.
- Compatibility can be configured globally or per subject (topic-key or topic-value).
- Schema evolution is supported by allowing compatible schema changes, such as adding optional fields or removing fields with default values.
- When a producer tries to register an incompatible schema:
- Schema Registry rejects the registration request.
- The producer receives an error indicating that the schema is incompatible.
- The producer must update its schema to be compatible with the existing schema before successfully registering the schema and producing data.

Think of "BA" in "BAckward" as standing for "Adding". (Adding, optional, defaults.) `FORWARD`: "FORward is for FOllowing defaults" (Adding, defaults.)

3. Subject Naming Strategies

- Subjects in Schema Registry represent a unique combination of a topic and a key or value schema.
- Schema Registry supports three subject naming strategies:
- `TopicNameStrategy` (default): The subject name is the topic name suffixed with `-key` or `-value`.
  - All messages in a topic must have the same schema.
  - Suitable when all messages in a topic have the same schema.
- `RecordNameStrategy`: The subject name is the fully-qualified name of the record schema.
  - Allows different topics to use the same schema.
  - All topics with the same record name must have the same schema version.
  - Useful when multiple topics share the same schema.
- `TopicRecordNameStrategy`: The subject name is the topic name concatenated with the record name.
  - Allows different topics to have different schemas for the same record name.
  - Provides flexibility for schema evolution per topic.
- The subject naming strategy determines how schemas are grouped and evolved within Schema Registry.

4. Schema Registry REST API

- Schema Registry exposes a RESTful API for interacting with schemas and subjects.
- Key endpoints include:
- `POST /subjects/(string: subject)/versions`: Register a new schema version under the specified subject.
- `GET /subjects/(string: subject)/versions/(versionId: version)`: Retrieve a specific version of the schema registered under the given subject.
- `POST /compatibility/subjects/(string: subject)/versions/(versionId: version)`: Test schema compatibility against a specified version.
- `GET /schemas/ids/(int: id)`: Retrieve the schema string associated with the given schema ID.
- `GET /subjects/(string: subject)/versions/latest`: Retrieve the latest version of a schema for a given subject.
- `GET /subjects/(string: subject)/versions`: Retrieve the version number of the latest schema for a given subject.
- The API allows registering schemas, retrieving schemas by ID or version, and testing schema compatibility.

5. Serializers and Deserializers

- Schema Registry provides serializers and deserializers for Avro, Protobuf, and JSON Schema.
- Serializers and deserializers are used by Kafka producers and consumers to convert between Kafka Connect data formats and Kafka's byte[].
- Key classes include:
- `KafkaAvroSerializer` and `KafkaAvroDeserializer` for Avro.
- `KafkaProtobufSerializer` and `KafkaProtobufDeserializer` for Protobuf.
- `KafkaJsonSchemaSerializer` and `KafkaJsonSchemaDeserializer` for JSON Schema.
- Serializers and deserializers are configured with the Schema Registry URL and handle schema registration and retrieval automatically.
- When a serializer tries to serialize data with an unregistered schema:
- The serializer automatically registers the schema with Schema Registry before serializing the data.
- The serializer makes a call to the Schema Registry API to register the schema and retrieve the schema ID.
- If the schema registration is successful, the serializer proceeds with serializing the data using the registered schema.
- If the schema registration fails (e.g., due to incompatibility), the serializer throws an exception, and the data is not serialized.

6. Multi-Datacenter Setup

- Schema Registry supports multi-datacenter deployments for high availability and disaster recovery.

- In a multi-datacenter setup, each datacenter has its own Schema Registry cluster, and the clusters are configured to replicate schemas asynchronously.
- Schema changes are propagated from the primary cluster to the secondary clusters using Kafka's internal replication mechanism.
- Producers and consumers in each datacenter interact with their local Schema Registry cluster for schema registration and retrieval.
- Schema replication in a multi-datacenter setup works as follows:
- One datacenter is designated as the primary (or master) datacenter, and the others are secondary (or slave) datacenters.
- When a schema is registered or updated in the primary datacenter, the Schema Registry cluster in the primary datacenter writes the schema to a special Kafka topic (`_schemas` by default).
- The Schema Registry clusters in the secondary datacenters consume the schemas from the `_schemas` topic and apply the changes locally.
- Schema replication ensures that all datacenters have a consistent view of the schemas, even in the presence of network partitions or datacenter failures.

7. Schema Registry Security

- Schema Registry supports authentication and authorization to secure access to schemas and the API.
- Authentication mechanisms include:
- Basic Auth: Username and password-based authentication.
- SSL/TLS: Encryption and authentication using certificates.
- SASL: Authentication using Kerberos or other SASL mechanisms.
- Authorization can be configured to control access to specific subjects and API endpoints based on user roles and permissions.
- Schema Registry can integrate with external authentication and authorization systems, such as LDAP or OAuth.
- To enable authentication in Schema Registry:
- Set the `kafkastore.security.protocol` configuration property to the desired security protocol (e.g., `SSL`, `SASL_SSL`).
- Configure the appropriate authentication settings based on the chosen protocol. For example, for basic auth, set `kafkastore.basic.auth.user.info` to the username and password.
- Configure authorization by setting `kafkastore.acl.authorizer.class` to the fully-qualified class name of the authorizer implementation.
- Clients accessing the Schema Registry API must provide the necessary credentials or certificates to authenticate and authorize their requests.

8. Confluent Control Center Integration

- Schema Registry integrates with Confluent Control Center, a web-based user interface for managing and monitoring Kafka clusters.

- Control Center provides a graphical interface for viewing and managing schemas, subject compatibility, and schema evolution.
- It allows searching for schemas, viewing schema details, and testing schema compatibility.
- Control Center also provides monitoring and alerting capabilities for Schema Registry, including metrics on schema registration, retrieval, and compatibility checks.
- To view the schema history for a subject in Confluent Control Center:
- Navigate to the "Schema Registry" section.
- Select the desired subject from the list of available subjects.
- In the subject details page, the schema history is displayed, including the version number, schema, and timestamp of each registered schema version.
- Different versions of the schema can be compared to see the changes between them.
- Control Center provides a user-friendly interface to explore the schema history and track the evolution of schemas over time.

9. Best Practices

- Use a meaningful and consistent subject naming convention based on your use case and subject naming strategy.
- Define schemas with appropriate compatibility rules to allow for schema evolution while maintaining compatibility.
- Use Avro, Protobuf, or JSON Schema for strong typing and schema validation.
- Ensure that producers and consumers are configured with the correct serializers and deserializers that match the schemas.
- Monitor Schema Registry metrics and logs to identify any issues or performance bottlenecks.
- Implement proper security measures, such as authentication and authorization, to protect sensitive schema information.
- Use Schema Registry in conjunction with Kafka Connect and ksqlDB to enable seamless data integration and stream processing with schema validation.
- Common schema evolution use cases include:
- Adding a new optional field to a schema: Producers can start including the new field in the data, while consumers can handle data with or without the field.
- Removing an existing field from a schema: Producers can stop including the field in the data, and consumers can handle data with or without the field (assuming a default value is provided).
- Renaming a field: Add a new field with the new name and mark the old field as deprecated. Consumers can handle both the old and new field names until the old field is completely removed.
- Changing the data type of a field: This requires careful consideration and may involve a multi-step process, such as adding a new field with the new data type and gradually migrating producers and consumers to use the new field.
- Evolving schemas in a backward or forward compatible manner allows for smooth updates and reduces the risk of data incompatibility between producers and consumers.

Additional Key Concepts

Schema Validation

- **Validation**: Ensure that the schema conforms to the expected structure and data types.
  - The Schema Registry provides a way to validate schemas before registering them.
  - Validation helps to catch errors early and enforce schema correctness.
- **Schema References**: Use schema references to manage shared schemas and reduce redundancy.
  - Particularly useful for complex schema structures.
  - References can help maintain consistency and ease of schema updates.

Advanced Avro Features

- **Logical Types**: Utilize logical types in Avro for precise representations of specific data types.
  - Examples: dates, times, decimal values.
  - Improves readability and interoperability of data.
- **Default Values**: Define default values for fields to handle missing data gracefully.
  - Important for schema evolution and backward compatibility.
  - Helps ensure that older consumers can still process newer data.

Implementing Schema Registry and Avro

Configuration Tips

- **Global Compatibility Setting**: Configure a global compatibility level for the Schema Registry.
  - Ensures consistent schema evolution policies across all subjects.
  - Reduces the risk of introducing breaking changes.
- **Caching**: Enable caching for schema lookups to improve performance.
  - Particularly beneficial in high-throughput environments.
  - Reduces latency in schema retrievals.

Monitoring and Metrics

- **Metrics Collection**: Use monitoring tools to track metrics related to schema operations.
  - Confluent Control Center is a recommended tool.
  - Metrics to track: schema registrations, retrievals, compatibility checks.
- **Alerting**: Set up alerts for common issues.
  - Schema compatibility violations.
  - High latency in schema retrievals.

Kafka Streams and Schema Registry Integration

- **Kafka Streams**: Integrate Schema Registry with Kafka Streams for stream processing.

    - Use `SpecificAvroSerde` or `GenericAvroSerde` for serialization and deserialization.
    - Ensure schema compatibility to avoid runtime errors.

Avro Serialization Details

Custom Avro Conversions

- **Custom Conversions**: Implement custom conversions for complex data types.
    - Useful for types not natively supported by Avro.
    - Can include certain Java objects or third-party library types.
    - Enhances flexibility in handling diverse data formats.

Security and Governance

Data Governance

- **Schema Governance**: Implement schema governance policies.
    - Manage schema lifecycle, access control, and audit logging.
    - Maintains data quality and ensures compliance with regulations.
- **Schema Evolution Strategies**: Plan and document schema evolution strategies.
    - Include versioning conventions and deprecation policies.
    - Ensures smooth transitions and minimizes disruptions.

Common Issues and Troubleshooting

Debugging Tips

- **Schema Not Found**: Ensure correct configuration for producer and consumer.
    - Verify the Schema Registry URL.
    - Check that schemas are correctly registered.
- **Serialization Errors**: Common issues and resolutions.
    - Missing schemas: Ensure schemas are registered before producing data.
    - Incompatible schema changes: Review schema compatibility settings.
    - Incorrect data types: Validate data against the schema.

Additional Useful Commands

Schema Deletion

- **Deleting Schemas**: Use Schema Registry API for schema deletion.
    - Ensure deletion does not impact existing consumers.

```
# Delete a specific schema version
curl -X DELETE http://localhost:8081/subjects/test-topic-value/versions/1
```

Bulk Operations

- **Bulk Schema Retrieval**: Retrieve all versions of a schema.
    - Useful for auditing or migration purposes.

-
  ```
  # Retrieve all versions of a schema
  curl -X GET http://localhost:8081/subjects/test-topic-value/versions
  ```

CHAPTER 12 : SECURITY

Kafka Security Configurations

Securing Kafka involves configuring encryption, authentication, and authorization mechanisms to protect data in transit and at rest. Properly configuring security settings is crucial for safeguarding sensitive information and ensuring compliance with security standards.

Key Points for CCDAK on Security

- **Encryption**: SSL/TLS encryption prevents data eavesdropping between clients and brokers.
- **Authentication**: SASL and SSL can be used to authenticate clients and brokers.
- **Authorization**: Controls access to Kafka resources, ensuring only authorized users can produce or consume data.
- **Comprehensive Security**: A combination of encryption, authentication, and authorization provides robust security.

Important Security Properties

`ssl.key.password`

- **Description**: Password for the private key in the key store file. This is optional and used for keys.
- **Security Impact**: Protects the private key integrity, ensuring that only authorized users can access it.

`ssl.keystore.location`

- **Description**: Location of the keystore file containing the server certificate and private key.
- **Security Impact**: Enables SSL encryption for data in transit. Proper management ensures identity integrity.

`ssl.keystore.password`

- **Description**: Password to access the keystore file.
- **Security Impact**: Secures the keystore content against unauthorized access.

`ssl.truststore.location`

- **Description**: Location of the truststore file containing certificates of trusted CAs.
- **Security Impact**: Essential for establishing trust relationships and validating certificate chains during SSL/TLS handshakes.

`ssl.truststore.password`

- **Description**: Password to unlock the truststore file.

- **Security Impact**: Protects the integrity of trusted certificates, ensuring they are not tampered with.

`sasl.mechanism`

- **Description**: SASL mechanism used for client-server authentication. Examples include PLAIN, GSSAPI (Kerberos), SCRAM, etc.
- **Security Impact**: Defines the authentication protocol, directly affecting security strength and compatibility.

`sasl.jaas.config`

- **Description**: JAAS configuration for SASL mechanisms specifying login modules and relevant options.
- **Security Impact**: Central to SASL authentication setup, ensuring secure and flexible authentication configurations.

`security.protocol`

- **Description**: Protocol used to communicate with brokers. Options include PLAINTEXT, SSL, SASL_PLAINTEXT, and SASL_SSL.
- **Security Impact**: Determines the level of security for data in transit between clients and brokers.

`ssl.endpoint.identification.algorithm`

- **Description**: The algorithm to use for host name verification of server certificates. Typically, this is set to HTTPS.
- **Security Impact**: Protects against man-in-the-middle attacks by verifying the server's hostname matches its certificate.

`sasl.login.callback.handler.class`

- **Description**: Specifies the callback handler class for SASL login if not using the default.
- **Security Impact**: Allows custom handling of authentication challenges and responses, enhancing flexibility in security configurations.

`sasl.kerberos.service.name`

- **Description**: Service name to use for Kerberos authentication.
- **Security Impact**: Critical for integrating Kafka with Kerberos for strong authentication, ensuring that services communicate securely under mutual authentication.

Core Principles for Robust Kafka Security : Encryption

- Implements SSL/TLS to safeguard data as it moves between Kafka clients and brokers, thwarting eavesdropping attempts.

Authentication

- Utilizes SASL and SSL for verifying client and broker identities, ensuring that only legitimate participants engage in communication.

Authorization

Apache Kafka ships with a pluggable, out-of-the-box Authorizer implementation that uses ZooKeeper to store all the ACLs. Setting ACLs is crucial because, without them, access to resources is limited to super users when an Authorizer is configured. By default, if a resource has no associated ACLs, no one except super users is allowed to access it.

Access Control Lists (ACLs) provide essential authorization controls for your enterprise's Kafka cluster data.

To set the required ACLs to allow a specific user "Andoni" to read and write to the topic "test-topic" from host "test-host," use the following command:

```
bin/kafka-acls.sh  --authorizer-properties  zookeeper.connect=localhost:2181  --add  --
allow-principal User:Andoni --allow-host test-host --operation read --operation write -
-topic test-topic
```

Comprehensive Security Model

- A layered approach combining encryption, authentication, and authorization fortifies Kafka's security posture.

Critical Security Configuration Parameters

SSL Configuration

- **`ssl.key.password`**: Secures the private key, preventing unauthorized access to critical encryption assets.
- **`ssl.keystore.location`**: Specifies the keystore's path, a fundamental component for SSL encryption, containing server certificates and keys.
- **`ssl.keystore.password`**: Protects the keystore's integrity, ensuring its contents remain confidential.
- **`ssl.truststore.location`**: Designates the truststore's path, essential for establishing trust chains during SSL handshakes.
- **`ssl.truststore.password`**: Guards the truststore against unauthorized alterations, maintaining the trust integrity.

SASL Configuration for Authentication

- **`sasl.mechanism`**: Defines the SASL mechanism (e.g., PLAIN, GSSAPI/Kerberos, SCRAM) for client-server authentication, influencing security level and protocol compatibility.
- **`sasl.jaas.config`**: Central to configuring SASL authentication, specifying login modules and options for various SASL mechanisms.

- **`sasl.login.callback.handler.class`**: Facilitates custom authentication challenge handling, offering flexibility in security setups.
- **`sasl.kerberos.service.name`**: Essential for Kerberos integration, ensuring secure mutual authentication between services.

Communication and Protocol Security

- **`security.protocol`**: Sets the communication protocol with brokers (e.g., PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL), directly affecting in-transit data security.
- **`ssl.endpoint.identification.algorithm`**: Enables hostname verification in server certificates to protect against man-in-the-middle attacks.

Kafka JAAS Configuration Example

```
KafkaClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/tmp/reader.user.keytab"
    principal="reader@KAFKA.SECURE";
};
```

Access Control with ACLs

Kafka provides ACLs for fine-grained access control, managed via:

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
super.users=User:admin;User:kafka
allow.everyone.if.no.acl.found=false
security.inter.broker.protocol=SASL_SSL
```

Managing ACLs

- **Listing ACLs**: `kafka-acls.sh --list --authorizer-properties zookeeper.connect=zookeeper:2181`
- **Adding ACLs**: `kafka-acls.sh --add --allow-principal User:Bob --allow-principal User:Alice --allow-hosts Host1,Host2 --operations Read,Write --topic Test-topic`

Zookeeper Security Configurations

For enhanced security, Zookeeper supports configurations like:

```
authProvider=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
jaasLoginRenew=3600000
kerberos.removeHostFromPrincipal=true
kerberos.removeRealmFromPrincipal=true
```

Migrating to Secure Topic Configurations

Use the `zookeeper-security-migration` tool for upgrading existing topics to secure configurations, ensuring all aspects of Kafka security are addressed.

CHAPTER 13 : TOPIC

Kafka Topic Configurations

Kafka topics are categorized into partitions for scalability and replicated across brokers for fault tolerance. Topic configuration parameters play a critical role in determining the behavior of topics in terms of performance, durability, and availability.

Key Points for CCDAK on Topics

- **Durability and Fault Tolerance**: Configurations like `replication.factor` determine how data is replicated to ensure availability in case of broker failures.
- **Scalability**: The `partition` count of a topic influences how data is distributed across the cluster and impacts parallel processing capabilities.
- **Data Consistency**: The `acks` setting affects how producers receive acknowledgments from brokers, impacting data consistency guarantees.
- **Order Guarantee:** Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.
    - If maintaining strict order within a partition is critical, set max.in.flight.requests.per.connection to 1. This ensures that while a message is retrying, no other messages can be sent that might overtake the retried message. (Unless `enable.idempotence` is enabled, as it prevents message reordering caused by retries.)
- **Partition-Specific Order:** Order is guaranteed only within a partition, not across partitions.
    - Repartitioning can disrupt the order of messages in Kafka, primarily because it changes the way records are assigned to partitions.
- **Variable Message Count:** Partitions do not need to have the same number of messages.
- **Partition-Specific Offsets:** Offsets only have meaning within a specific partition.
- **Data Retention:** Messages are kept only for a limited time in Kafka, with the default being one week.
- **Immutability:** Once data is written to a partition, it cannot be changed.
- **Partition Assignment:** Data is assigned randomly to a partition unless a key is provided.

After adding partitions later, it cannot be guaranteed that old messages will be on the same partition as new messages with the same key.

Important Topic Properties

`acks`

- **Scope**: Producer
- **Description**: Determines the number of acknowledgments the producer requires from brokers before considering a request complete. Valid values are `0`, `1`, or `all`.
- **Impact**: This setting impacts data durability and producer throughput. Setting `acks=all` ensures higher data safety as it waits for all in-sync replicas to acknowledge.

However, this may lower throughput compared to `acks=1` or `acks=0`, where the latter offers the highest throughput but with potential data loss.

`replication.factor`

- **Scope**: Topic
- **Description**: Specifies the number of copies (replicas) of a topic to maintain across the cluster.
- **Default Value**: This is a topic-level configuration set at the time of topic creation and doesn't have a universal default. It often defaults to the cluster's `default.replication.factor`.
- **Impact**: A higher replication factor increases data availability and fault tolerance but requires more disk space and network bandwidth. It is critical for ensuring that even if some brokers are down, your data remains accessible.

`partitions`

- **Scope**: Topic
- **Description**: Determines the number of partitions within a topic. Partitions are the basic unit of parallelism in Kafka, with each partition being independently consumed.
- **Default Value**: This is set at the time of topic creation and typically defaults to the cluster's `num.partitions` setting.
- **Impact**: More partitions allow greater parallel processing of data but can increase the overhead on the Kafka cluster and Zookeeper. Finding the right balance is key to optimizing performance and resource utilization.

Essential Kafka Topic Configuration Parameters

Durability and Fault Tolerance

- **`replication.factor`**: Defines the number of replicated copies of a topic across the cluster, enhancing data availability and fault tolerance. The actual number can be set per topic and is crucial for maintaining data integrity in the event of broker failures.

Scalability Through Partitioning

- **`partitions`**: Controls the partition count for a topic, directly influencing data distribution and parallel processing capabilities across consumers. More partitions support higher parallelism but may increase cluster management overhead.

Data Consistency and Throughput

- **`acks`**: Configures acknowledgment requirements from brokers to producers, balancing between data consistency and throughput. Values range from `0` (fire and forget), `1` (leader acknowledgment), to `all` (full ISR acknowledgment).

Advanced Topic Configurations for Fine-Tuning

Log Compaction and Retention

- **`cleanup.policy`**: Supports log compaction (`"compact"`) to retain at least the last known value for each key within a topic, crucial for stateful applications.
    - The default `cleanup.policy` for Kafka topics is "delete". (Kafka will delete records that are older than the retention period specified by `retention.ms`.)
    - Use `kafka-topics.sh` CLI tool or Admin API to change the policy for existing topics.
    - The `delete` policy might also be used in tandem to prevent the state store from growing indefinitely, especially when windowed operations are involved. (Use `"compact,delete"`)
    - Kafka Streams often requires that the latest state be recoverable even after restarts or rebalancing, making compaction a necessity.
- **`min.cleanable.dirty.ratio`**: Determines the ratio of log segments eligible for compaction. Lower values trigger compaction sooner, helping maintain a cleaner log with less overhead.

Segment Management and Efficiency

- **`segment.ms`**: Configures the time Kafka waits before closing the current log segment and starting a new one. Segment management affects storage and can impact log compaction and retention behavior.

Timestamps and Ordering

- **`message.timestamp.type`**: Defines whether Kafka should use the message creation time (`CreateTime`) or the time of log append (`LogAppendTime`). This setting can impact message ordering and log retention policies.

Key Considerations for Message Key and Size

Message Key Usage

- Utilizing a message key (`key!=null`) ensures that messages with the same key are always sent to the same partition, facilitating message ordering per key. However, modifying the partition count can disrupt this consistency.

Managing Message Sizes

- Kafka's default message size limit is 1MB. Exceeding this without proper configuration adjustments (`message.max.bytes`, `replica.fetch.max.bytes`, `fetch.message.max.bytes`) will result in a `MessageSizeTooLargeException`.

CHAPTER 14: ZOOKEEPER

ZooKeeper Overview

Apache ZooKeeper used to play a critical role in Kafka's ecosystem, acting as a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. In newer versions of Kafka Zookeeper is not required anymore.

Key Concepts

- **Cluster Coordination**: ZooKeeper is used by Kafka for leader election, cluster membership, and topic configuration management.
- **Configuration Management**: Stores configuration data and metadata for Kafka brokers, topics, and other entities.

Zookeeper's internal data structure is a tree:

- Each node is called a zNode
- Each zNode has a path
- zNodes can be persistent or ephemeral
- Each zNode can store data
- You cannot rename a zNode
- Each zNode can be watched for changes

ZooKeeper Configuration Parameters

Understanding key ZooKeeper configuration parameters is essential for setting up and managing a robust Kafka environment:

- **`data.dir`**:
  - **Description**: Location of the directory where ZooKeeper stores its data.
  - **Impact**: Affects how ZooKeeper stores and retrieves cluster state and metadata.
- **`maxClientCnxns`**:
  - **Description**: Maximum number of client connections that can be concurrently handled by a ZooKeeper server. Zero means unlimited.
  - **Impact**: Balances load handling capacity with resource utilization.
- **`tickTime`**:
  - **Description**: The length of a single tick in milliseconds, which is the basic time unit in ZooKeeper's time management. Commonly set to 2000.
  - **Impact**: Influences session timeouts and heartbeat intervals for cluster management.
- **`initLimit`**:
  - **Description**: Specifies the amount of time, in ticks (as defined by `tickTime`), to allow followers to connect and sync to a leader. Commonly set to 10.
  - **Impact**: Affects startup and recovery times for ZooKeeper quorums.

- `syncLimit`:
    - **Description**: Defines how many ticks can pass between sending a request and receiving an acknowledgment. Often set to 5.
    - **Impact**: Impacts the tolerance for network latency between ZooKeeper nodes.
- `autopurge.snapRetainCount`:
    - **Description**: The number of most recent snapshots and the corresponding transaction logs in the dataDir and dataLogDir to retain. Additional files are deleted during purge. Often set to 3.
    - **Impact**: Controls disk space usage by managing the retention of snapshot files.
- `autopurge.purgeInterval`:
    - **Description**: The interval, in hours, at which the purge task is triggered. Set to 24 hours to purge once a day.
    - **Impact**: Automates the cleanup process to manage disk space effectively.

Important ZooKeeper Operations

- **zkCli.sh**: The ZooKeeper command-line interface to interact with the ZooKeeper ensemble.
- `# List the brokers registered in ZooKeeper`
- `zkCli.sh -server localhost:2181 ls /brokers/ids`
- **Configuration**: Understanding the ZooKeeper configuration file (`zoo.cfg`) is essential for setting up and managing a ZooKeeper ensemble.
- **Monitoring and Maintenance**: Familiarity with ZooKeeper's metrics and logs for monitoring the health and performance of the ensemble.

ZooKeeper Ensemble

- **Definition**: A ZooKeeper ensemble is a cluster of ZooKeeper servers deployed across multiple nodes to ensure fault tolerance and high availability.
- **Quorum Calculation**: The ensemble follows a 2n + 1 formula, where n is the number of allowed failed servers. This odd number configuration enables majority-based leader elections and operational quorum maintenance.
- **Example**: To tolerate the failure of 2 servers, an ensemble requires 5 servers (2*2+1 = 5).

High Availability and Fault Tolerance

- The ensemble allows for up to `n` failed servers while still maintaining the quorum. Losing the quorum (more than `n` failures in a 2n + 1 configuration) renders the ZooKeeper cluster non-operational.
- **Quorum Loss Impact**: If quorum is lost, the ZooKeeper service will cease to function, affecting the dependent Kafka cluster's metadata management and overall operation.

Multi-Node Configuration Parameters

- **`initLimit`** and **`syncLimit`**: These parameters control the time allowed for server synchronization with the leader during startup (`initLimit`) and operation (`syncLimit`).

  - Given `tickTime=2000`, `initLimit=5`, and `syncLimit=2`, a follower server has up to 10000ms (5 * 2000ms) to initialize with the leader and can be out of sync for up to 4000ms (2 * 2000ms) without being considered down.

Ensemble Membership Configuration

- **`server.<myid>`**: Defines the ensemble membership and communication ports.
  - **Format**: `server.<myid>=<hostname>:<leaderport>:<electionport>`
  - **`myid`**: A unique server identification number within the ensemble. It's specified in a `myid` file located in the ZooKeeper's `dataDir`. This ID correlates with one of the ensemble configurations.
  - **`leaderport`**: Used by following servers to connect to the current leader.
  - **`electionport`**: Facilitates leader election among ensemble members.

Four Letter Words (4LW) Commands

ZooKeeper supports various "Four Letter Words" commands for monitoring and administration purposes. To enable these commands, the following property must be set in `zoo.cfg`:

- **`4lw.commands.whitelist=*`**: This configuration whitelists all available 4LW commands for use.

Key 4LW Commands

- **`ruok`**: Checks if ZooKeeper server is running without errors.
- `echo "ruok" | nc localhost 2181; echo`
- **`conf`**: Prints the server's current configuration.
- **`cons`**: Lists details about client connections to the server.
- **`crst`**: Resets connection and session statistics.
- **`envi`**: Displays the server's environment variables.
- **`srst`**: Resets server statistics.
- **`mntr`**: Outputs a list of variables that can be used for monitoring the health of the cluster.
- **`dump`**: Lists all sessions and ephemeral nodes. This command only works on the leader node.

Znodes in ZooKeeper

ZooKeeper manages data in a hierarchical namespace, similar to a filesystem, which consists of znodes. There are two main types of znodes:

Persistent Znodes

- **Characteristics**: Permanent and must be explicitly deleted by the client. They remain available even after the session that created them ends.
- **Use Cases**: Useful for storing configuration information or metadata that must persist beyond the lifespan of the creating session.

Ephemeral Znodes

- **Characteristics**: Temporary and automatically deleted when the session that created them ends. These znodes are crucial for detecting client disconnections.
- **Use Cases**: Often used for maintaining cluster membership and leader election, as they provide a straightforward way to detect when a node has failed or left the cluster.

Best Practices

- **Security**: Secure ZooKeeper ensemble using ACLs and by enabling SSL for client-server communications.
- **Backup and Recovery**: Regularly backup ZooKeeper data and understand the procedures for recovery in case of data loss.

Troubleshooting

- **Connectivity Issues**: Solving common connectivity problems between Kafka and ZooKeeper, such as ensuring the correct port and host are specified in Kafka's `zookeeper.connect` configuration.
- **Performance Tuning**: Adjusting ZooKeeper settings for optimal performance, especially in larger Kafka deployments.

Useful Commands

Here are some useful ZooKeeper commands for managing Kafka's metadata:

- Checking the health of ZooKeeper:
- `echo ruok | nc localhost 2181`

  If ZooKeeper is running fine, it responds with `imok`.

- Listing topics (via ZooKeeper):
- `zkCli.sh -server localhost:2181 ls /brokers/topics`

KRaft Overview

- KRaft (pronounced craft) mode is generally available starting with Confluent Platform version 7.4
- KRaft is the consensus protocol introduced to remove Kafka's dependency on ZooKeeper for metadata management
- Simplifies Kafka's architecture by consolidating responsibility for metadata into Kafka itself

Why move to KRaft?

- Enables right-sized clusters to scale up to millions of partitions
- **Near-instantaneous metadata failover**
- Single management model for authentication, configuration, protocols, and networking

The Controller Quorum

- KRaft controller nodes comprise a Raft quorum which manages the Kafka metadata log
- The leader of the metadata log is called the **active controller**
- Follower controllers replicate data and serve as **hot standbys**
- Requires a **majority** of nodes to be running (e.g., 3-node cluster can survive 1 failure)
- Controllers periodically write out a **snapshot** of the metadata to disk

Scaling Kafka with KRaft

- KRaft mode is designed to handle a **much larger number of partitions** per cluster compared to ZooKeeper
- Experiment shows KRaft can handle **10 times** the maximum number of partitions for a cluster running ZooKeeper
- Controlled **shutdown time and recovery time** after uncontrolled shutdown are greatly improved with a quorum controller versus ZooKeeper

Configure Confluent Platform with KRaft

- Client configurations are not impacted by Confluent Platform moving to KRaft

Limitations and Known Issues

- Combined mode (Kafka node acting as a broker and KRaft controller) is not currently supported for production workloads
- JBOD (just a bunch of disks) is not supported in KRaft mode
- Source-initiated Cluster Linking for Confluent Platform between a source cluster running Confluent Platform 7.0.x or earlier and a destination Confluent Platform cluster running in KRaft mode is not supported
- No support for quorum reconfiguration (adding or removing KRaft controllers)
- Cannot currently use Schema Registry Topic ACL Authorizer for Confluent Platform for Schema Registry with Confluent Platform in KRaft mode
- Health+ reports KRaft controllers as brokers, which may cause alerts to function unexpectedly

See the Kraft section on the Last minute review

CHAPTER 15 : LAST MINUTE REVIEW

LAST MINUTE REVIEW

Kafka Overview:

- **Kafka relies on ZooKeeper** for maintaining metadata, configuration data, and providing synchronization within distributed systems.
- **Auto topic creation** depends on the broker setting `auto.create.topics.enable`.
- **In Kafka, ordering is guaranteed** only within a partition, not across partitions.
- **A topic partition consists of segments** with two indexes: offset to position and timestamp to offset.
- **Topic, Partition and Offset** uniquely identifies a message in Kafka.
- **Replicas are passive**, they don't handle produce or consume request. Produce and consume requests get sent to the node hosting partition leader.
- **Point-to-point (request-response)** style will couple client to the server.

Broker:

- **A Kafka Broker** is a JVM process that runs on a machine and hosts topics.
- Any Kafka broker can provide cluster **metadata to clients**.
- **In production, set the Java heap size to 6 GB** for optimal Kafka performance.
- **In a multi-broker Kafka cluster**, only the `broker.id` property needs to be unique for each node.
- **Garbage Collection** is used by Kafka brokers to handle the deletion of unused objects and tune performance.
- **Kafka brokers** handle all read and write requests for partitions and manage replication between partitions.
- **The Controller** is one of the brokers in a Kafka cluster responsible for managing the states of partitions and replicas.
- **Kafka brokers** communicate with each other using the Kafka protocol over TCP.
- **If a broker fails**, Kafka automatically fails over to one of the replicas, promoting it to be the new leader.
- **Broker configuration** is defined in the `server.properties` file, which includes settings like `broker.id`, `port`, `log.dirs`, `zookeeper.connect`, etc.
- **Kafka brokers** rely on ZooKeeper for maintaining their cluster state, like topics, partitions, replicas, leaders, ISRs, and more.
- **Each broker** has a unique integer identifier, the `broker.id`, which is used in various Kafka protocols and requests.
- `min.insync.replica`s does not impact producers when `acks=1` only when `acks=all`
- **Kafka brokers** handle authentication and authorization of clients based on the configured security protocols and ACLs.

- **Kafka brokers** expose metrics via JMX for monitoring their performance and health, like request rates, byte rates, CPU usage, etc.
- **Kafka brokers** can be configured to use rack awareness for improved fault tolerance, by spreading replicas across different racks.
- **Kafka brokers** can be horizontally scaled by adding more broker nodes to the cluster to increase storage and processing capacity.
- **OS page cache** is heavily used by Kafka brokers for caching log segments, which allows for efficient reads and writes.
- **Kafka brokers** persist all messages to disk and maintain an in-memory cache of messages to serve faster reads.
- **Quotas** can be configured on Kafka brokers to enforce limits on produce and fetch requests to control resource utilization.
- **Kafka brokers** support multiple listener configurations to allow different security protocols on different ports, like PLAINTEXT, SSL, SASL, etc.
- **Kafka brokers** can be configured with different log cleanup policies, like deletion or compaction, to manage disk space usage.
- **Log Segments:** Kafka brokers split log data into segments. Each segment is a large file, which is flushed to disk periodically.
- **Replica Fetchers:** Brokers have fetcher threads to pull data from leader replicas and replicate it to follower replicas.
- **Inter-Broker Communication:** Brokers use ZooKeeper to elect the controller and maintain metadata about the cluster.
- **Broker Metrics:** Commonly monitored metrics include `UnderReplicatedPartitions`, `OfflinePartitionsCount`, `RequestHandlerAvgIdlePercent`, and `NetworkProcessorAvgIdlePercent`.
- **Leader and Follower:** Each partition has a leader broker and follower brokers. The leader handles all reads and writes, while followers replicate the data.
- **Broker Startup:** During startup, brokers register themselves with ZooKeeper and load topic metadata to prepare for handling requests.
- **Kafka Protocol:** Brokers use a binary protocol for communication, which clients and other brokers utilize to interact with the cluster.
- **Broker Upgrades:** Brokers can be upgraded with zero downtime using a rolling upgrade process, ensuring continuous availability.
- In a 5 node ZooKeeper ensemble, **up to 2 servers can fail while still maintaining a quorum**. In a 9 broker, 4 can fail and still keep voting majority.
- ZooKeeper ensemble members communicate on **ports 2181, 2888, 3888** by default.
- The **Metadata request can be handled by any node**, enabling clients to discover the designated leader for topic partitions.
- With `replication.factor=3`, `min.insync.replicas=2`, `acks=all`, a `NOT_ENOUGH_REPLICAS` exception is thrown if **2 brokers are down**.
- Setting `unclean.leader.election.enable=true` allows **non-ISR replicas to become leader**, ensuring availability but risking data loss.

- ZooKeeper's behavior is governed by the ZooKeeper configuration file, which includes keywords like **clientPort, dataDir, and tickTime**.
- Kafka supports rack awareness for **fault tolerance** by configuring the `broker.rack` property to spread replicas across racks.
- Garbage collection in Kafka brokers is a **JVM process that automatically frees up memory and removes unused pointers to objects**, helping to improve overall performance.

CLI:

- **Creating Topics:** Use `bin/kafka-topics.sh --create` to create topics.
- **Describing Topics:** Use `bin/kafka-topics.sh --describe` to get details about topics.
- **Listing Topics:** Use `bin/kafka-topics.sh --list` to list all topics in the cluster.
- **Deleting Topics:** Use `bin/kafka-topics.sh --delete` to delete a topic.
- **Consuming Messages:** Use `bin/kafka-console-consumer.sh` to consume messages from a topic.
- **Producing Messages:** Use `bin/kafka-console-producer.sh` to produce messages to a topic.
- **Managing Consumer Groups:** Use `bin/kafka-consumer-groups.sh` to manage consumer groups, including viewing offsets and lag.
- **Resetting Offsets:** Use `bin/kafka-consumer-groups.sh --reset-offsets` to reset the offsets for consumer groups.
- **Configuring Topics:** Use `bin/kafka-configs.sh` to alter topic configurations.
- **Viewing Configs:** Use `bin/kafka-configs.sh --describe` to view configurations of brokers, topics, or clients.
- **Running a Simple Producer Performance Test:** Use `bin/kafka-producer-perf-test.sh` to test producer performance.
- **Running a Simple Consumer Performance Test:** Use `bin/kafka-consumer-perf-test.sh` to test consumer performance.
- **Setting ACLs:** Use `bin/kafka-acls.sh --add` to set ACLs for users or hosts.
- **Listing ACLs:** Use `bin/kafka-acls.sh --list` to view the ACLs configured.
- **Removing ACLs:** Use `bin/kafka-acls.sh --remove` to remove ACLs.
- **Starting Kafka Connect:** Use `bin/connect-standalone.sh` for standalone mode or `bin/connect-distributed.sh` for distributed mode.
- **Monitoring Kafka Connect:** Use `bin/connect-distributed.sh --status` to check the status of Kafka Connect tasks and connectors.
- **Rebalancing Partitions:** Use `bin/kafka-reassign-partitions.sh` to perform partition reassignments.
- **Checking Reassignments:** Use `bin/kafka-reassign-partitions.sh --verify` to verify the status of partition reassignments.
- **Viewing Partition Reassignment Plans:** Use `bin/kafka-reassign-partitions.sh --generate` to generate a reassignment plan.
- **Running a Simple Consumer Group Test:** Use `bin/kafka-consumer-groups.sh --describe --group <group-id>` to check the status of consumer groups.
- **Listing Consumer Groups:** Use `bin/kafka-consumer-groups.sh --list` to list all consumer groups.
- **Decommissioning Brokers:** Use `bin/kafka-preferred-replica-election.sh` to initiate a preferred replica election.

- **Migrating ZooKeeper:** Use `bin/kafka-migration.sh` to help with migrating metadata from ZooKeeper to KRaft (Kafka's Raft-based metadata quorum).
- The Kafka CLI command to display the Kafka version is: `bin/kafka-console-producer.sh --version` or any `bin/kafka-*.sh` script with the `--version` parameter.
- To create a topic only if it does not already exist, use the `--if-not-exists` parameter with the `kafka-topics.sh` command.

Administration:

- **For a safe data pipeline without message loss**, use `acks=all`, `replication.factor=3`, `min.insync.replicas=2`.
- With `replication.factor=3`, `min.insync.replicas=2`, `acks=all`, a `NOT_ENOUGH_REPLICAS` exception is thrown if 2 brokers are down.
- **Set `unclean.leader.election.enable=false`** to prevent data loss. ("Consistency-Availability" trade-off)
- **Partitions with out-of-sync replicas** are considered under-replicated.
- **Use `min.insync.replicas`** to define the minimum number of replicas that must acknowledge a write for it to be considered successful.
- **Monitor under-replicated partitions** to ensure data durability and prevent data loss.
- **Use `auto.leader.rebalance.enable=true`** to automatically balance leadership across the cluster.
- **Configure `num.partitions` per topic** based on the desired parallelism and throughput.
- **Use `log.retention.hours` or `log.retention.bytes`** to control the data retention period.
- **Monitor disk usage** on brokers and add more disks or brokers when necessary.
- **Use `replica.lag.time.max.ms`** to control the maximum time a replica can lag behind the leader.
- **Upgrade Kafka brokers** one at a time, starting with the controller, to ensure a smooth rolling upgrade.
- **Use Kafka ACLs** to control access to topics and consumer groups.
- **Configure `max.message.bytes`** to control the maximum size of a message that can be produced.
- **Use `compression.type`** to enable compression for efficient storage and network usage.
- **Monitor the Kafka cluster** using tools like Kafka Manager, Prometheus, and Grafana.
- **Use Kafka Connect** for integrating Kafka with external systems like databases and filesystems.
- **Configure `zookeeper.connection.timeout.ms`** to control the maximum time to wait for a ZooKeeper connection.
- **Use Kafka Streams** for real-time data processing and aggregation.
- **Configure `producer.buffer.memory` and `consumer.fetch.max.bytes`** to tune producer and consumer performance.
- **Implement quotas** using `quota.producer.byte-rate` and `quota.consumer.byte-rate` to manage resource usage.
- **Set `log.cleanup.policy`** to either `delete` or `compact` to manage log retention policies. (`delete` is the default)

- **Use `controlled.shutdown.enable=true`** to ensure brokers shut down cleanly without data loss.
- **Monitor the `UnderReplicatedPartitions` metric** to track replication health.
- **Review and adjust JVM heap settings** to optimize broker performance (`-Xmx` and `-Xms` settings).
- **Implement rack awareness** by setting `broker.rack` to improve fault tolerance.
- **Regularly back up ZooKeeper data** to protect against data loss.
- **Enable TLS/SSL** for secure communication between brokers and clients.
- **Use `security.inter.broker.protocol`** to configure secure communication between brokers.
- **Configure `zookeeper.session.timeout.ms`** to manage ZooKeeper session timeouts.
- **Use `advertised.listeners`** to ensure brokers can be reached by clients.
- **Enable and monitor JMX** for broker metrics and performance data.

Consumer:

- Consumers can **manually assign partitions using `assign()`** and **seek to specific offsets using `seek()`**.
- **The first consumer to join a group** becomes the group leader.
- **Consumer group rebalancing** reassigns partitions for a proportional share when members join/leave.
- **Kafka consumer partition assignment strategies** include `RangeAssignor`, `RoundRobinAssignor`, `StickyAssignor`, and `CooperativeStickyAssignor`.
- **Committed consumer group offsets** take precedence over `auto.offset.reset`.
- **For at-most-once**, commit offsets before processing messages.
- For at-least-once processing, **commit offsets after processing messages**.
- **Consumers can subscribe to multiple topics** via regex or a topic list.
- To achieve scaling out and scaling in for a ksqlDB cluster, you can start additional ksqlDB servers with the same configuration during live operations or stop the desired running servers, keeping at least one server running. The **maximum parallelism depends on the number of partitions**.
- In a multi-consumer setup, the **consumer group leader is responsible for assigning a subset of topic partitions to each consumer** in the group. The **heartbeat thread** is used to detect if a consumer has failed.
- If two consumers have the same group ID and subscribe to the same topic, **each consumer will be assigned a subset of the partitions** in the topic.
- To commit offsets, **consumers produce messages to a special `__consumer_offsets` topic** with the committed offset for each partition.
- For **at-most-once processing**, offsets should be committed **before** processing the data.
- **Adding more consumers to a consumer group** is the main way to scale data consumption from a Kafka topic.
- **Consumer offsets are stored** in the `consumer_offsets` topic, schemas are stored in the `_schemas` topic.
- **Use `max.poll.records`** to limit the number of records returned in a single call to `poll()`.

- **Use `enable.auto.commit=false`** for manual offset control.
- **Consumers with the same `group.id` form a consumer group** and share the partitions of subscribed topics.
- **Kafka consumers are not thread-safe** and should not be shared across threads.
- **Use `isolation.level=read_committed`** to only read transactional messages that have been committed.
- **Set `fetch.max.bytes` and `max.partition.fetch.bytes`** to control the amount of data fetched per partition.
- **Use `ConsumerRebalanceListener` to get notifications** when partitions are revoked or assigned during rebalancing.
- **Tune `fetch.min.bytes` and `fetch.max.wait.ms`** to balance between latency and throughput.
- **Set `max.poll.interval.ms`** to control the maximum time between poll invocations before the consumer is considered dead.
- **Use `seek()` to reset the consumer's position** to a specific offset.
- **Commit offsets asynchronously** using `commitAsync()` for better performance.
- **Handle `CommitFailedException`** when offset commit fails due to rebalancing or other errors.
- **Use `auto.offset.reset`** to control the behavior when there are no initial offsets or the current offset does not exist.
- **Configure `heartbeat.interval.ms`** to control the interval at which the consumer sends heartbeats to the group coordinator.
- **Use `session.timeout.ms`** to set the timeout for detecting consumer failures.
- **Monitor consumer lag** to ensure that the consumer is keeping up with the latest records.
- In a multi-consumer setup, the **consumer group leader is responsible for assigning a subset of topic partitions to each consumer** in the group.
- The **heartbeat thread** in a Kafka consumer is used to detect if a consumer has failed by sending periodic heartbeats to the broker.
- Consumer **fetch size** is the most important configuration for performance, controlled by `fetch.min.bytes` and `fetch.max.bytes`.
- The `wakeup()` method can be safely used from an external thread to **interrupt an active consumer operation**.
- Consumers **read data in order within each topic partition** and can **read from multiple partitions**.
- **Enable `client.id`** to provide a logical application name in logs and metrics for easier monitoring.
- **Set `interceptor.classes`** to add custom interceptors for additional logging, monitoring, or modifications to records.
- **Use `poll()` in a loop** to continuously consume messages from the Kafka broker.
- **Handle rebalances gracefully** by using `RebalanceListener` to perform necessary actions during partition assignment and revocation.
- **Calling `close()`** on consumer immediately triggers a partition rebalance as the consumer will not be available anymore.
- If there are no initial offsets or the current offset does not exist, setting `auto.offset.reset=none` will make the consumer **throw an exception**.
- When used with default options and no specified group ID, the `kafka-console-consumer` generates a **random consumer group**.

Kafka Connect:

- Kafka Connect uses **workers** to execute connectors and tasks for reading data from or writing data to external systems.
- Kafka Connect provides a **REST API for managing connectors and tasks** in both standalone and distributed modes. The REST API server can be configured using the `listeners` configuration option.
- **Connectors** in Kafka Connect are responsible for breaking down the data copying job into **tasks** that can be distributed to workers. There are two types of connectors: **SourceConnectors for importing data and SinkConnectors for exporting data**.
- Confluent Cloud offers **fully managed connectors**, which can be leveraged by using the `ccloud-stack` utility to create a new Confluent Cloud instance.
- **Kafka Connect simplifies connector development**, deployment, and management, supporting distributed and standalone modes.
- **Distributed mode in Kafka Connect** handles automatic work balancing, scaling, and fault tolerance. Topics for offsets, configs, and statuses should be manually created.
- **Connector configurations** must include a unique name, max tasks, and the connector class.
- **Kafka Connect Source** is used to import data from external systems into Kafka.
- **The number of sink tasks** for a Kafka Connect connector should not exceed the number of partitions in the input topic(s).
- **max.tasks limits the number of tasks** per connector.
- **Kafka Connect Sink** is used to export data from Kafka to external systems.
- **The Kafka Connect REST API** allows managing and deploying connectors, and checking their status.
- **Use Single Message Transformations (SMTs)** to modify the data in-flight between the source and sink.
- **The Kafka Connect `key` and `value` converters** are used to serialize and deserialize data between Kafka and external systems.
- **Kafka Connect supports schema evolution** through the use of a schema registry like the Confluent Schema Registry.
- **Use the `tasks.max` parameter** to set the maximum number of tasks that should be created for the connector.
- **The Kafka Connect `offset` topic** stores the offsets for each connector, allowing for fault tolerance and recovery.
- **The Kafka Connect `status` topic** stores the current status of connectors and tasks.
- **Kafka Connect `dead letter queue`** can be used to handle connector errors and problematic records.
- **Kafka Connect supports custom connectors** through the implementation of the `SourceConnector` or `SinkConnector` interfaces.
- **Use the `connector.class` parameter** to specify the Java class for the connector.
- **Kafka Connect can be used for Change Data Capture (CDC)** to capture changes from databases and stream them into Kafka.
- **The Kafka Connect `config` topic** stores the configuration for each connector and task.

- **Kafka Connect can be scaled horizontally** by adding more worker nodes to the cluster.
- **Standalone mode in Kafka Connect** is simpler to set up but **lacks the fault tolerance and scalability of distributed mode**.
- Kafka Connect **supports exactly-once semantics** for **sink and source connectors** if they are designed to leverage the framework's capabilities.
- **Plugin discovery** in Kafka Connect is controlled by the `plugin.path` worker configuration, and the `service_load` strategy is fastest but requires verifying plugin compatibility.
- **Standalone mode in Kafka Connect** is simpler and requires less setup but lacks the fault tolerance and scalability of distributed mode.
- **Monitor Kafka Connect** using JMX metrics and integrate with monitoring tools like Prometheus and Grafana.
- **Restart connectors and tasks** through the REST API to recover from transient errors or apply configuration changes.
- **Ensure proper configuration** of `offset.storage.topic`, `config.storage.topic`, and `status.storage.topic` for reliable operation.
- **Use `value.converter.schemas.enable`** to control whether the schema is included with each message for Avro, JSON, and Protobuf.
- **Configure `key.converter` and `value.converter`** to specify the serialization format used by the connectors.
- **Leverage Kafka Connect transformations** to perform lightweight message modifications without needing custom code.
- Connector configurations are defined in the worker configuration file or REST requests as **key-value mappings**.
- Kafka Connect can provide **exactly-once semantics for sink and source connectors** if they are designed to take advantage of the framework's capabilities.

Streams:

- **Tumbling time windows in Kafka Streams** are fixed-size, non-overlapping, and gap-less.
- **Kafka Streams achieves parallelism** by splitting the topology into tasks that handle a subset of partitions independently.
- The **maximum parallelism** of a Kafka Streams application is determined by the **number of partitions of the input topic(s)** being read.
- **In Kafka Streams, `map` creates an output stream**, `foreach` is a terminal operation, and `peek` is for side effects.
- **The Kafka Streams API** supports exactly-once delivery.
- **Stream-table duality:** A stream is a changelog of a table, and a table is a snapshot of a stream at a point in time.
- **When using a persistent state store** in Kafka Streams, close iterators to prevent memory issues.
- **The Kafka Streams API** supports KStream-to-KStream windowed joins and KTable-to-KTable non-windowed joins.

- **Stateless operations** don't require state, while stateful operations depend on state for processing: Branch, Filter, FlatMap, Foreach, GroupByKey, GroupBy, Map, Map (values only), Peek, Print, SelectKey, Table to Stream.
- **Stateful operations** : Aggregate, Count and Reduce. (Joins, windowing and custom pressesors)
- **Mounting a persistent volume for RocksDB** can dramatically speed up Kafka Streams recovery on restart.
- **reduce, join, count and aggregate** are stateful Kafka Streams operations.
- **KStream-KTable joins** output a KStream.
- Kafka Streams supports **at-least-once and exactly-once processing guarantees**.
- Kafka Streams applications run on **client machines at the periphery of a Kafka cluster**, not within the Kafka brokers or the cluster itself.
- To speed up recovery for a Kafka Streams application, the state can be stored on a **persistent volume**.
- A benefit of using `GlobalKTable` when joining input data is that the **input data does not need to be co-partitioned**.
- Kafka Streams **achieves parallelism by splitting the topology into tasks** that are executed by threads across application instances. The **maximum parallelism is determined by the number of partitions** of the input topic(s).
- When repartitioning in Kafka Streams, the framework **writes the repartitioned data to a new topic with new keys and partitions**, creating two independent subtopologies.
- **Kafka Streams exactly-once semantics** apply to Kafka-to-Kafka flows only.
- **KStream-GlobalKTable join** does not require input topics to have the same number of partitions, unlike other join types.
- **Kafka Streams uses the `application.id` config** as a prefix for internal topics like repartition and state topics.
- **Kafka Streams supports Interactive Queries** to query the state of a running application.
- **Hopping time windows in Kafka Streams** have a fixed size but can overlap, emitting results at a specified interval.
- **Sliding time windows in Kafka Streams** are similar to hopping windows but emit results every time a new event enters or exits the window.
- **Session windows in Kafka Streams** group events by key and session, with sessions having a defined inactivity gap.
- Kafka Streams should be used for **transforming data from one Kafka topic to another**, as it is simpler and more powerful than using a consumer and producer directly.
- To effectively **manage the size of state stores in Kafka Streams**, consider using **windowed stores, producing tombstones, or implementing a custom TTL-based cleanup mechanism**.
- **Kafka Streams supports custom SerDes** for serializing and deserializing data.
- **Kafka Streams requires at least one input topic and one output topic** for processing data.
- **The Kafka Streams DSL** provides a high-level API for common stream processing operations.

- **The Kafka Streams Processor API** provides a low-level API for custom stream processing logic.
- **Kafka Streams supports fault-tolerant state** through the use of changelog topics and state stores.
- **The Kafka Streams `KStream` abstraction** represents an unbounded stream of records.
- **The Kafka Streams `KTable` abstraction** represents a changelog stream, where each record represents an update.
- **The Kafka Streams `GlobalKTable` abstraction** represents a fully replicated, non-partitioned changelog stream.
- Use KStream when you need to process each record independently, perform stateless transformations, or handle unbounded data.
- Use KTable when you need to perform aggregations, joins, or maintain a materialized view of the latest values for each key.
- When joining streams/tables, **input data must be co-partitioned with the same number of partitions**. (Except for GlobalKTable)
- **Branch, FlatMapValues, GroupBy** are examples of stateless operations in Kafka Streams.
- To convert a KStream to a KTable, options include `KStream.toTable()`, **writing to Kafka and reading as KTable**, or performing a dummy aggregation.
- For Kafka Streams output topics, it's recommended to set `cleanup.policy=compact` to align with **KTable semantics**.

KSQL:

- ksqlDB is a dialect inspired by ANSI SQL but introduces differences like **"windowing" for streaming data processing**.
- ksqlDB **doesn't support structured keys**, making it impossible to create a stream from a windowed aggregate.
- **ksqlDB simplifies stream processing applications** on Kafka by allowing developers to write SQL queries. Key use cases include materialized caches, streaming ETL, and event-driven microservices.
- **ksqlDB supports windowing operations** like `TUMBLING`, `HOPPING`, and `SESSION` windows for aggregations over time.
- **SHOW STREAMS and EXPLAIN statements** in ksqlDB communicate directly with the ksqlDB server, not Kafka.
- **Idle ksqlDB servers** consume few resources. Only servers corresponding to the number of partitions actively process a query.
- **ksqlDB supports exactly-once processing**, configurable with the `processing.guarantee` setting.
- **Adding a ksqlDB server to an existing cluster** requires matching `bootstrap.servers` and `ksql.service.id` settings.
- **The DESCRIBE EXTENDED statement in ksqlDB** helps check compatibility between the `VALUE_FORMAT` of a stream and the format of topic records.
- **The default port for the KSQL server is 8088**.
- **KSQL CREATE STREAM/TABLE AS SELECT queries** write to Kafka topics.

- The `bootstrap.servers` configuration in ksqlDB specifies the **initial connection point to the Kafka cluster** as a list of host and port pairs.
- ksqlDB allows you to **query, read, write, and process data in Kafka** in real-time and at scale using **intuitive SQL-like syntax** without requiring proficiency in Java or Scala or installing a separate processing cluster.
- ksqlDB can be deployed in **headless mode** (recommended for production) or **interactive mode**.
- ksqlDB stores **metadata in the config topic** in headless mode, containing the ksqlDB properties provided during the initial startup of the application.
- In interactive mode, securing the **ksqlDB command topic is crucial for protecting sensitive metadata** related to DDL statements and TERMINATE queries.
- ksqlDB **supports Avro, Protobuf, JSON, and delimited formats** by specifying the desired format and configuring `ksql.schema.registry.url` to point to Schema Registry.
- **Use SET `auto.offset.reset`='earliest'** for KSQL to read a topic from the start.
- **KSQL-related data and metadata are stored** in Kafka topics, not in Zookeeper, databases, or Schema Registry.
- In interactive mode, **securing the ksqlDB command topic is crucial** for protecting sensitive metadata related to **DDL statements and TERMINATE queries**.
- ksqlDB **supports Avro, Protobuf, and JSON formats** by specifying the desired format (e.g., `VALUE_FORMAT='AVRO'`) and configuring `ksql.schema.registry.url` to point to Schema Registry.
- **ksqlDB supports Pull and Push queries**. Pull queries are one-time queries that return a result immediately, while Push queries are continuous queries that output results to a new stream or table.
- **The ksqlDB REST API** allows interacting with ksqlDB servers programmatically.
- **ksqlDB supports User Defined Functions (UDFs)** for custom processing logic.
- **ksqlDB queries are scalable and fault-tolerant** since they leverage the underlying Kafka infrastructure.
- **The `CREATE STREAM` statement** in ksqlDB creates a new stream from a Kafka topic or another stream.
- **The `CREATE TABLE` statement** in ksqlDB creates a new table from a Kafka topic or another table.
- **The `INSERT INTO` statement** in ksqlDB writes records into an existing stream or table.
- **The `CREATE CONNECTOR` statement** in ksqlDB creates a new Kafka Connect connector.
- **ksqlDB supports JOIN operations** between streams and tables, including `INNER JOIN`, `LEFT JOIN`, and `OUTER JOIN`.
- To use Avro data with ksqlDB, **Schema Registry must be installed** and `ksql.schema.registry.url` configured to point to it.
- The ksqlDB command topic stores metadata for **persistent queries based on `CREATE STREAM AS SELECT` and `CREATE TABLE AS SELECT`**.
- Maximum parallelism in ksqlDB depends on the **number of topic partitions**.

Metrics:

- **Confluent Control Center** enables centralized management and monitoring of Kafka components.
- **Important metrics** to monitor include `UnderReplicatedPartitions`, `ActiveControllerCount`, `OfflinePartitionsCount`, `RequestHandlerAvgIdlePercent`, `NetworkProcessorAvgIdlePercent`, and `IsrExpandsPerSec`.
- **Monitor consumer lag** using `records-lag-max` to track how many messages the consumer is behind.
- **Kafka exposes metrics via JMX** (Java Management Extensions) for monitoring.
- **Use Kafka's built-in JmxTool** to dump JMX metrics periodically for analysis.
- **Monitor the `BytesInPerSec` and `BytesOutPerSec` metrics** to track the inbound and outbound byte rate of brokers.
- **The `MessagesInPerSec` metric** indicates the rate at which messages are being produced to brokers.
- **Monitor the `UnderMinIsrPartitionCount` metric** to track partitions with insufficient in-sync replicas.
- **The `PartitionCount` metric** shows the total number of partitions across all topics in the cluster.
- **The `LeaderCount` metric** indicates the number of partitions for which a broker is the leader.
- **Monitor the `RequestQueueSize` and `ResponseQueueSize` metrics** to ensure brokers can keep up with client requests.
- **The `ProduceTotalTimeMs` and `FetchTotalTimeMs` metrics** track the total time taken for produce and fetch requests, respectively.
- **Use the `FetchMessageConversionsPerSec` metric** to monitor the rate of message format conversions during fetch requests.
- **The `ZooKeeperRequestLatencyMs` metric** tracks the latency of ZooKeeper requests made by Kafka.
- **Monitor the `NetworkProcessorAvgIdlePercent` metric** to ensure the network thread is not a bottleneck.
- **The `ControllerState` metric** indicates whether a broker is currently the active controller.
- **Use Prometheus and Grafana** for collecting, storing, and visualizing Kafka metrics.
- **The Confluent Platform** includes built-in dashboards for monitoring Kafka clusters, Connect, ksqlDB, and Schema Registry.
- **Datadogs Kafka integration** provides pre-built dashboards and alerts for monitoring Kafka performance and health.
- **Cloudwatch and CloudTrail** are used for monitoring Kafka clusters running on AWS.
- **Azure Monitor** is used for monitoring Kafka clusters running on Azure.

Producer:

- **The most important producer configurations** are `acks`, `compression`, and `batch.size`.
- **Producers send messages to the leader** of a topic partition. Same keys go to the same partition.
- **The Kafka Producer workflow** includes `ProducerRecord`, `Serializer`, and `Partitioner`, but not `Deserializer`.
- **To manually commit offsets**, use the `commitSync()` API.

- **Set `bootstrap.servers` to multiple host:port pairs** for Kafka broker fault tolerance.
- **bootstrap.servers, key.serializer and value.serializer** are mandatory for producers.
- **Setting `max.in.flight.requests.per.connection > 1`** with retries enabled can lead to out-of-order messages.
- **linger.ms increases the chance of batching** in producers.
- The `send()` method returns a **Future object with RecordMetadata**. Using `Future.get()` **waits for a reply from Kafka** before continuing and **throws an exception if the record is not sent successfully**.
- Setting `enable.idempotence=true` is a producer configuration that can be used to **guarantee a stable and safe pipeline without processing duplicate messages**.
- To send binary data through the REST Proxy, the **producer** needs to encode the data into base64.
- When creating a `ProducerRecord`, the **topic and value are mandatory**, while the key and partition are optional.
- **Enabling compression and increasing batch.size** can improve producer throughput.
- **`Message_TOO_LARGE`, `INVALID_REQUIRED_ACKS`, and `TOPIC_AUTHORIZATION_FAILED`** are examples of non-retriable errors for Kafka producers, while `NOT_ENOUGH_REPLICAS` and `NOT_LEADER_FOR_PARTITION` are retriable.
- **Using producer.send(record).get()** will decrease throughput by waiting for a reply from Kafka.
- **Setting compression.type=snappy** for a producer means consumers have to decompress the data. Kafka brokers transfer compressed data as-is.
- **KafkaProducer may throw `SerializationException` or `BufferExhaustedException`** before sending the record. It handles large messages by throwing `MessageSizeTooLargeException` without retries if exceeding max size.
- **Setting `enable.idempotence=true`** helps prevent duplicate messages caused by network issues between the producer and broker.
- **The `acks` configuration** determines the level of acknowledgement required from the broker before considering a write successful.
- **The `buffer.memory` configuration** sets the total amount of memory the producer can use to buffer records waiting to be sent to the server.
- **The `retries` configuration** sets the number of times the producer will retry a failed request before giving up.
- **The `max.block.ms` configuration** sets the maximum amount of time the producer will block when calling `send()` or `partitionsFor()`.
- **The `request.timeout.ms` configuration** sets the maximum amount of time the producer will wait for a response from the server when sending a request.
- **The `max.request.size` configuration** sets the maximum size of a request in bytes.
- **The `transactional.id` configuration** enables idempotent and transactional delivery.
- **Setting `compression.type` to `lz4`** provides a good balance of compression ratio and CPU usage.
- **The `partitioner.class` configuration** allows specifying a custom partitioner for determining which partition a record should be sent to.
- **Producers can be created using the `KafkaProducer` class** in the Kafka Java API.

- `RecordTooLargeException` is a **non-retriable exception**, meaning the message will not be sent again.
- The `send()` method returns a **Future object with RecordMetadata**.
- To receive leader acknowledgment of writes, set `acks=1`.
- **Increasing batch size (`batch.size`) and enabling compression (`compression.type`)** can improve producer throughput when the batches are completely full.
- Setting **`linger.ms`** to a higher value (e.g., 10 ms) **increases the chances of batching messages** by allowing the producer to wait before sending a batch.
- **Custom partitioning** can be implemented to optimize message distribution across partitions based on specific requirements, such as dedicating a partition to a high-volume customer.

Schema Registry:

- **Supported schema formats in Confluent Platform** include Avro, JSON, and Protobuf.
- **For Avro BACKWARD compatibility**, allowed changes are deleting fields and adding optional fields.
- **For Avro FORWARD compatibility**, allowed changes are adding fields with defaults and deleting optional fields.
- **For Avro FULL compatibility**, only adding optional fields is allowed.
- **Avro primitive types** include null, boolean, int, long, float, double, bytes, and string.
- **Avro logical types** include date, time, timestamp, and decimal.
- **Schema Registry supports HTTP and HTTPS client protocols**.
- **The Schema Registry stores schemas in the internal `_schemas` Kafka topic**, not in an embedded database or Zookeeper.
- **Avro deserializer fetches missing schemas from the registry**.
- **Adding a field without a default is a forward compatible Avro schema change**.
- **Avro SpecificRecord classes** are generated from an Avro schema plus a Maven/Gradle plugin, not just the schema alone.
- The **Confluent Schema Registry default compatibility type** is `BACKWARD`, and `FULL` compatibility allows adding or removing **optional fields only**.
- **Queries** in Schema Registry can be tested using **cURL**, such as `curl -X GET http://localhost:8081/schemas/ids/1` to fetch a schema by its global ID.
- **The Schema Registry provides a REST API** for registering, retrieving, and managing schemas.
- **The Schema Registry uses a unique subject name per topic** to identify the schema associated with the data.
- **The Schema Registry supports schema evolution** by allowing compatible schema changes.
- **The Schema Registry checks compatibility** based on the compatibility type set for the subject (e.g., BACKWARD, FORWARD, FULL).
- **The Schema Registry can be deployed in a multi-node setup** for high availability and fault tolerance.
- **The Schema Registry integrates with Kafka Connect** for automatic schema management in source and sink connectors.

- **The Schema Registry integrates with ksqlDB** for schema inference and schema evolution in ksqlDB applications.
- **The Schema Registry UI** provides a web-based interface for viewing, searching, and managing schemas.
- **The Schema Registry Client** is a Java library for interacting with the Schema Registry from Kafka clients.
- **The Schema Registry Maven Plugin** generates Java classes from Avro schemas for use in Kafka producers and consumers.
- **The Schema Registry supports multiple subjects per topic**, allowing different schemas for keys and values.
- **Confluent Replicator** can be used to replicate schemas between Schema Registry clusters.
- **Avro deserializer fetches missing schemas from the registry**.
- Adding a field without a default is a **forward compatible Avro schema change**.

Security:

- **Encryption is configured** using listener configuration and managing truststores/keystores to protect data in transit.
- **Kafka security features are optional**, supporting a mix of authenticated, unauthenticated, encrypted, and non-encrypted clients.
- **To allow a user to read/write a topic**, add an ACL with `--allow-principal` and `--allow-host` for read/write operations.
- **SASL can be used with PLAINTEXT or SSL**. If `SASL_SSL`, SSL must also be configured.
- **ACLs are stored in the Zookeeper node /kafka-acls/ by default**.
- **SAML is not a valid authentication mechanism in Kafka**. Valid options are SASL/GSSAPI, SASL/SCRAM, and SSL.
- **Kafka clients use HTTPS (SSL/TLS) to securely connect to the Confluent REST Proxy**.
- **Enabling SSL encryption in Kafka disables the zero-copy optimization** since data must be loaded into the JVM to encrypt/decrypt.
- Kafka supports authentication using **SSL certificates, SASL/PLAIN, SASL/SCRAM, SASL/GSSAPI (Kerberos)**.
- **SASL/PLAIN and SASL/SCRAM** provide username/password-based authentication, while **SASL/GSSAPI** integrates with Kerberos.
- The security protocol of each listener is defined in the `listener.security.protocol.map` configuration, with options like **PLAINTEXT, SSL, SASL_PLAINTEXT, SASL_SSL**.
- **Kafka supports authorization using ACLs (Access Control Lists)** to control access to resources like topics and consumer groups.
- **Encryption of data at rest can be achieved using disk encryption** or by configuring Kafka to encrypt its data files.
- **Kafka supports SSL/TLS encryption for client-broker and broker-broker communication**.

- **ACLs in Kafka are defined using a combination of principal, host, operation, and permission**.
- **Kafka supports HTTPS for securing communication with the Kafka Connect REST API and Schema Registry API**.
- **Confluent Control Center supports HTTPS** for securing its web interface and API endpoints.
- **Kafka supports delegating authentication and authorization to external systems** like LDAP or Active Directory using plugins.
- **Quotas can be used to limit the resource consumption** of clients, such as the amount of data they can produce or consume.
- **Kafka supports encrypting sensitive configuration values** using the `kafka-configs` command-line tool.
- **The Confluent Platform Security Plugin** provides additional security features like Audit Logs, Quotas, and LDAP integration.
- When securing a running Kafka cluster, the **recommended approach** is to enable security in phases: **client-broker connection first, then broker-broker, and finally closing the PLAINTEXT port**.
- **Host name verification** in Kafka is the process of checking the certificate presented by the server against the actual hostname or IP address to prevent man-in-the-middle attacks. It can be disabled by setting `ssl.endpoint.identification.algorithm` to an empty string.
- Kafka supports authenticating to ZooKeeper with **SASL and mTLS individually or both together**. When using mTLS alone, every broker and CLI tool should identify itself with the **same Distinguished Name (DN)** for proper ACL'ing.
- The Confluent Server **Authorizer** can be used to capture, protect, and preserve authorization activity into topics in a Kafka cluster, helping to track user and application access across the platform through **audit logs**.

ZooKeeper: (Being replaced by KRaft, less important now)

- **ZooKeeper keeps track of znodes**, which have a path, can store data, and be persistent or ephemeral. Renaming znodes is not supported.
- **In a 5 node ZooKeeper ensemble, up to 2 servers can fail while still maintaining a quorum**.
- **ZooKeeper ensemble members communicate on ports 2181, 2888, 3888 by default**.
- **For ZooKeeper config tickTime=2000, initLimit=20, syncLimit=5**, follower connection timeout is 40 sec (2000ms * 20 ticks).
- **ZooKeeper uses a leader-follower architecture**, where one node is elected as the leader and the others are followers.
- **ZooKeeper uses a consensus algorithm called Zab (ZooKeeper Atomic Broadcast)** to ensure consistency across the ensemble.
- **ZooKeeper stores its data in memory** and writes transaction logs and snapshots to disk for persistence.
- **ZooKeeper clients connect to a single ZooKeeper server** and can failover to another server if the connection is lost.

- **ZooKeeper is used by Kafka for storing metadata** like topics, partitions, replicas, and controller information.
- **The `zookeeper.connect` parameter in Kafka** specifies the connection string for the ZooKeeper ensemble.
- **The `zookeeper.session.timeout.ms` parameter in Kafka** sets the timeout for ZooKeeper sessions.
- **The `zookeeper.connection.timeout.ms` parameter in Kafka** sets the maximum time to wait for a ZooKeeper connection.
- **ZooKeeper uses a hierarchical namespace** similar to a file system, with paths separated by slashes (/).
- **ZooKeeper supports watches** that allow clients to receive notifications when a znode changes.
- **ZooKeeper has a command-line interface (CLI)** for interacting with the ZooKeeper ensemble.
- **The `zkCli.sh` script** is used to start the ZooKeeper CLI and execute commands.
- **The `zoo.cfg` file** is used to configure ZooKeeper, including settings like `tickTime`, `dataDir`, and `clientPort`.
- **ZooKeeper can be used for distributed locking, leader election, and configuration management** in distributed systems.
- **Confluent Kafka distributions include a bundled ZooKeeper** for convenience, but a separate ZooKeeper ensemble is recommended for production.
- **ZooKeeper requires a majority (quorum) of servers to be available** for the ensemble to be operational.
- Running Kafka in production, a machine should have at least **32 GB of RAM** as a decent choice for storing and caching messages.
- In a ZooKeeper ensemble with 9 servers, **up to 4 servers can fail** while still maintaining a quorum.
- The command to start the ZooKeeper service is: `bin/zookeeper-server-start.sh config/zookeeper.properties`.

KRaft Configuration in Confluent Platform

Hardware and JVM Requirements

- **Minimum of 4 GB of RAM**
- **Dedicated CPU core** should be considered when the server is shared
- An **SSD disk at least 64 GB** in size is highly recommended
- **JVM heap size of at least 1 GB** is recommended

Configuration Files

- Example KRaft configuration files are located in `/etc/kafka/kraft/` after installing Confluent Platform
- `broker.properties`: Settings for broker-only servers

- `controller.properties`: Settings for controller-only servers
- `server.properties`: Settings for combined broker and controller servers (not supported for production)

Metrics Reporter

- The metrics reporter must be **enabled on each broker and controller** in KRaft mode to see broker metrics in Confluent Control Center
- Uncomment the relevant lines in the properties file

Socket Server Settings

- `listeners`: Must be configured for controllers, consistent with `controller.quorum.voters` value
- `controller.listener.names`: Required for KRaft mode, specifying listeners used by the controller

Log Settings

- `log.dirs`: Should list only one log directory for KRaft mode, as JBOD is not currently supported
- `num.partitions`: Sets the default number of log partitions per topic for brokers (ignored by controllers)

Metadata Retention Settings

- `metadata.log.dir`: Specifies the location of the metadata log for KRaft clusters (defaults to the first `log.dirs` directory if not set)
- `metadata.max.idle.interval.ms`: Sets how often the active controller writes no-op records to the metadata partition (default 500 ms)

Settings for Other Components

- When using KRaft instead of ZooKeeper, current, non-deprecated configuration settings must be used for:
- Clients and services
- Schema Registry
- Administrative tools
- Retrieving the Kafka cluster ID

Generating and Formatting IDs

- Before starting Kafka, use the `kafka-storage` tool to:
- Generate a cluster ID with `random-uuid`
- Format each node with the `format` command

Debugging Tools

- Kafka provides tools for debugging KRaft mode clusters:
- `kafka-metadata-quorum`: Describe runtime status
- `kafka-dump-log`: Debug log segments
- `kafka-metadata-shell`: Inspect the metadata partition

Monitoring Metrics

- Key metrics to monitor for KRaft mode:
- KRaft quorum metrics (e.g., `append-records-rate`, `commit-latency-avg`)
- Controller metrics (e.g., `ActiveBrokerCount`, `LastAppliedRecordOffset`)
- Broker metadata metrics (e.g., `last-applied-record-offset`, `metadata-load-error-count`)

Topic:

- **Auto topic creation** uses broker config for `num.partitions` and `default.replication.factor`.
- **Dynamic topic configs are stored in Zookeeper**.
- **Kafka Connect supports up to 1 task per input topic partition**.
- **Producing with keys** allows influencing the partitioning of messages to ensure ordering within each partition.
- **To produce data to a topic**, a producer must provide any broker from the cluster and the topic name. The partitions list is not required.
- **Create a topic with** `bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 3 --partitions 3 --topic test`.
- **The number of partitions in a Kafka topic can only be increased**, not decreased, using the kafka-topics.sh command.
- **Configure topic-level retention.ms**, not just broker-level log.retention.ms, to set retention for a specific topic to 1 hour (3600000 ms).
- **Only the leader replica handles produce and consume requests** for a partition. Follower replicas do not actively serve clients.
- **To find partitions without a leader**, use: `kafka-topics.sh --bootstrap-server localhost:9092 --describe --unavailable-partitions`.
- **Partition rebalance for a consumer group** is triggered by increasing partitions, adding/removing consumers, or consumer shutting down.
- **If multiple log retention configs are set**, the smaller unit size takes precedence.
- **Adding partitions causes new records with the same key to potentially go to different partitions**. Existing records stay put.
- **Tombstone records with null value will remove all messages with that key from the log on compaction**.
- **Log compaction is triggered when a segment closes if the dirty ratio threshold is met**. Not on every message or flush.
- **Kafka messages are immutable and cannot be modified after they are written to the log**.

- **The "same key to same partition" rule breaks down if the number of partitions for the topic changes**, as this changes the key hashing.
- **For Kafka Streams output topics, set `cleanup.policy=compact`** to align with KTable semantics that require log compaction.
- **With fewer topic partitions than consumer group members**, some consumers will remain idle, limited by the number of partitions.
- **Active-passive mirroring** is when a topic is replicated from an active region to a passive region used only for read-only purposes like analytics.
- **With 5 brokers, 10 partitions, 3 replicas, a client is allowed 5 MB/s maximum throughput** if each broker has a 1 MB/s quota.
- **Any broker can handle metadata requests**.
- **Consumers commit offsets by interacting with the Group Coordinator broker**, not by writing directly to the `consumer_offsets` topic.
- **The Controller is a broker elected by ZooKeeper** to be responsible for partition leadership assignment (in addition to normal tasks).
- **Producers automatically handle broker leader changes** by requesting new leaders from any broker and resuming production.
- `unclean.leader.election.enable=true` allows non ISR replicas to become leader, ensuring availability but losing consistency as data loss might occur
- A Kafka broker automatically creates a topic if a producer sends a message, a consumer reads a message, or a client requests metadata for the topic, when `auto.create.topics.enable` is set to **true**.
- Adding partitions causes **new records with the same key to potentially go to different partitions**, while existing records stay put.
- Kafka topics are always **multi-producer and multi-subscriber**, and you can have **as many topics as you want**.
- To configure a **topic-level retention period**, set the `retention.ms` property for the specific topic (**10 days = 864000000 ms**).
- Log compaction in Kafka ensures that at least the **last known value for each message key** is retained within a topic partition, which is useful for restoring state after crashes or rebuilding caches.
- With log compaction, **tombstone records with null values** will remove all messages with that key from the log.

Other:

- **Serialization converts objects to byte streams** for transmission, while deserialization does the opposite.
- **Unit tests verify** isolated code blocks work as expected.
- **After adding partitions**, there's no guarantee that old and new messages with the same key will be on the same partition.
- **Kafka uses TCP** for high-performance communication between servers and clients.

- **In point-to-point messaging**, messages are exchanged between senders and receivers using a queue.
- **Active-passive mirroring** is when a topic is replicated from an active region to a passive region used only for read-only purposes like analytics.
- **To reduce consumer rebalance time from 10s to 3s with 12 consumers**, decrease `session.timeout.ms` to 3s and decrease `heartbeat.interval.ms`.
- **To guarantee at-least-once processing**, do not commit offsets until successfully processing the failed record, rather than committing earlier offsets.
- **Consumers in the same group read mutually exclusive partitions**, not mutually exclusive offsets or all data from all partitions.
- **The Kafka Producer is thread-safe and can be shared across threads**, but the Consumer is not thread-safe and should be used in one thread.
- Kafka uses **Java and Scala** as its primary programming languages.
- When evaluating a stream processing system, global considerations include the **availability of clean APIs and abstractions** and the system's **community support**, in addition to use-case-specific considerations.