DUMPSBOSS.

Confluent Certified Developer for Apache Kafka Certification Examination

Confluent CCDAK

Version Demo

Total Demo Questions: 26

Total Premium Questions: 268

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Apache Kafka Fundamentals	111
Topic 2, Apache Kafka Development	69
Topic 3, Kafka Connect	29
Topic 4, Kafka Streams	42
Topic 5, Schema Registry	17
Total	268

QUESTION NO: 1

A Zookeeper ensemble contains 3 servers. Over which ports the members of the ensemble should be able to communicate in default configuration? (select three)

A. 2181

B. 3888

C. 2888

D. 9092

E. 80

ANSWER: A B C

Explanation:

The required ports are **2181**, **2888**, and **3888**. In the standard ZooKeeper configuration, port 2181 is the client port. Each ZooKeeper server listens on this port for client sessions, which can include applications and services that use ZooKeeper as well as administrative or health-check connections. It must therefore be reachable by the components that need to connect to the ensemble.

Port 2888 is the quorum peer communication port specified in each server entry. ZooKeeper ensemble members use it to exchange data and maintain the replicated quorum. Port 3888 is the election port, also specified in the server entries, and is used by ensemble members during leader election. A three-node ensemble needs peer-to-peer connectivity on both the quorum communication and leader-election ports so that it can establish a quorum and elect or replace a leader as needed.

These ports are commonly configured in ZooKeeper server definitions in the form `server.id=host:2888:3888`, with `clientPort=2181` defining client connectivity. See the Apache ZooKeeper Administrator's Guide and the Confluent Platform networking and ports documentation.

QUESTION NO: 2

In Avro, adding a field to a record without default is a __ schema evolution

A. forward

B. backward

C. full

D. breaking

ANSWER: A

Explanation:

forward is correct. Forward compatibility means that applications using the previous reader schema can read data produced using the newer writer schema. When a field is added to an Avro record, data written with the new schema contains that additional field. An older reader schema does not define the field, so Avro schema resolution simply ignores it while reading the record. Consequently, no default value is needed for this particular direction of compatibility: the old reader is not trying to populate a newly added field in its own schema.

This distinction is important when configuring Schema Registry compatibility. Under forward compatibility, Schema Registry verifies that data produced with the new schema remains consumable by clients using the prior schema. Adding a field without a default therefore fits this direction because the prior schema can safely disregard the unknown field. Avro's schema-resolution rules explicitly state that writer fields not present in the reader schema are ignored. See the Apache Avro specification and Confluent's Schema Evolution and Compatibility documentation.

QUESTION NO: 3

When using plain JSON data with Connect, you see the following error messageorg.apache.kafka.connect.errors.DataExceptionJsonDeserializer with schemas.enable requires "schema" and "payload" fields and may not contain additional fields. How will you fix the error?

A. Set key.converter, value.converter to JsonConverter and the schema registry url

B. Use Single Message Transforms to add schema and payload fields in the message

C. Set key.converter.schemas.enable and value.converter.schemas.enable to false

D. Set key.converter, value.converter to AvroConverter and the schema registry url

ANSWER: C

Explanation:

Set key.converter.schemas.enable and value.converter.schemas.enable to false is correct because Kafka Connect's JSON converter defaults to schema-enabled JSON handling. With schema support enabled, the converter expects each record to use Connect's JSON envelope structure, containing top-level `schema` and `payload` fields. A plain JSON event does not use that envelope; it is simply the payload itself. Consequently, attempting to deserialize such an event while schema support remains enabled produces the stated DataException.

Setting the `schemas.enable` setting to `false` for both the key and value converters instructs Connect's `JsonConverter` to process schemaless JSON. In many real deployments only values contain JSON and keys may be strings, but configuring both properties is the appropriate answer when both converters are configured as JSON converters and the input consists of plain JSON. These settings can be supplied in a connector configuration as converter-specific overrides, or in the Connect worker configuration when they apply globally. The Kafka documentation describes the JSON converter's schema-enabled envelope and the `schemas.enable` configuration property: Apache Kafka Connect configuration documentation. Confluent also documents JSON serialization formats and schema behavior in Confluent Connect converters documentation.

QUESTION NO: 4

Which statements are true about Kafka ACL resource patterns? (select two)

A. A literal resource pattern matches one exact resource name

B. A prefixed resource pattern can match resources beginning with a specified prefix

C. An ACL can grant access only to consumer groups, not topics

D. A prefixed pattern matches resources based on a regular expression

E. ACLs are evaluated only when a topic is created

ANSWER: A B

Explanation:

An ACL can use a literal resource pattern to match one exact resource name. For example, an ACL for the literal topic `orders` applies to that topic and not to topics such as `orders-us`.

An ACL can also use a prefixed resource pattern to match resources whose names begin with a specified prefix. For example, an ACL with a prefixed topic resource named `payments-` can apply to topics such as `payments-us` and `payments-eu`. Prefix patterns are useful for authorizing controlled topic namespaces. See the Confluent Platform ACL overview and the Apache Kafka authorization documentation.

QUESTION NO: 5

If you enable an SSL endpoint in Kafka, what feature of Kafka will be lost?

A. Cross-cluster mirroring

B. Support for Avro format

C. Zero copy

D. Exactly-once delivery

ANSWER: C

Explanation:

Zero copy is correct. Kafka's high-throughput plaintext data path can use the operating system's zero-copy transfer capability, commonly implemented through the `sendfile` system call. With this optimization, record bytes can be transferred from the broker's log files through the operating-system page cache and to the network socket without requiring Kafka to copy the payload into JVM user-space buffers. This reduces memory copying, CPU overhead, and garbage-collection pressure, particularly when brokers serve large volumes of data to consumers.

An SSL/TLS listener must encrypt outbound record data before it is transmitted. To perform that cryptographic processing, the broker needs access to the record bytes in the JVM/TLS processing path rather than passing the file data directly from the operating system to the socket. As a result, the direct zero-copy file-to-network optimization is unavailable for SSL-enabled connections. Kafka still provides secure encrypted transport and continues to read, store, replicate, and serve records normally; the relevant trade-off is the loss of this particular performance optimization on the encrypted endpoint. Kafka's design documentation describes its use of zero-copy transfer for efficient consumer delivery, while Confluent's security documentation covers SSL/TLS as the mechanism for encrypting Kafka client-broker traffic. See Apache Kafka: Network Layer Design and Confluent Platform Security Tutorial.

QUESTION NO: 6

Suppose you have 6 brokers and you decide to create a topic with 10 partitions and a replication factor of 3. The brokers 0 and 1 are on rack A, the brokers 2 and 3 are on rack B, and the brokers 4 and 5 are on rack C. If the leader for partition 0 is on broker 4, and the first replica is on broker 2, which broker can host the last replica? (select two)

A. 6

B. 1

C. 2

D. 5

E. 0

F. 3

ANSWER: B E

Explanation:

The correct choices are **1** and **0**. Kafka's rack-aware replica assignment uses the broker rack configuration to spread replicas for a partition across distinct racks whenever the replication factor can be satisfied by the available rack count. Here, the leader is on broker 4 in rack C and another replica is on broker 2 in rack B. Because the topic has a replication factor of 3 and there are three racks, the remaining replica should be assigned to the unused fault domain, rack A.

Rack A contains brokers 0 and 1, so either broker can host the final replica. Placing one replica in each rack protects the partition against the loss of an individual rack: replicas in the two remaining racks can still be available if one rack becomes unavailable. This placement behavior depends on brokers being configured with their rack identity through the `broker.rack` setting. Kafka uses that topology information during replica assignment to improve resilience while distributing replicas across the cluster. See Apache Kafka's documentation for broker.rack and Confluent's guidance on broker rack awareness.

QUESTION NO: 7

(You are writing a producer application and need to ensure proper delivery.

You configure the producer with acks=all.

Which two actions should you take to ensure proper error handling?

Select two.)

A. Check the value of ProducerRecord.status().

B. Use a callback argument in producer.send() where you check delivery status.

C. Check that producer.send() returned a RecordMetadata object and is not null.

D. Surround the call to producer.send() with a try/catch block to catch KafkaException.

ANSWER: B D

Explanation:

Use a callback argument in `producer.send()` where you check delivery status, and surround the call to `producer.send()` with a try/catch block to catch `KafkaException`. Kafka producers send records asynchronously by default. Supplying a `Callback` to `send()` provides the definitive completion result for an individual produce request: on successful acknowledgment, the callback receives `RecordMetadata`; when asynchronous delivery fails, it receives an exception. The application can therefore log the failure, update application state, invoke its retry or recovery policy where appropriate, and avoid treating submission of a record as proof of delivery.

A try/catch around `send()` is also necessary because not every failure is reported asynchronously. Kafka's producer API documents synchronous exceptions that can be raised while preparing or submitting a send, including serialization-related failures and other `KafkaException` subclasses. Catching these failures at the call site lets the application distinguish an immediate inability to submit the record from a later broker-side delivery result handled by the callback. Together, these mechanisms cover both synchronous submission errors and asynchronous completion errors, which is essential even when `acks=all` is configured. See the KafkaProducer.send API documentation and Confluent Producer Configuration and Development documentation.

QUESTION NO: 8

We would like to be in an at-most once consuming scenario. Which offset commit strategy would you recommend?

A. Commit the offsets on disk, after processing the data

B. Do not commit any offsets and read from beginning

C. Commit the offsets in Kafka, after processing the data

D. Commit the offsets in Kafka, before processing the data

ANSWER: D

Explanation:

For an at-most-once consuming scenario, use **Commit the offsets in Kafka, before processing the data**. In this delivery model, the consumer records its progress before performing the business processing for the records returned by `poll()`. If the consumer crashes, is restarted, or otherwise fails after that commit but before processing completes, the next consumer instance starts after the committed offsets. Consequently, some records can be skipped rather than processed again. This intentional tradeoff prevents duplicate processing, which is the defining goal of at-most-once semantics, while accepting the possibility of data loss during a failure window.

Offsets should be committed to Kafka because Kafka stores consumer-group offsets in its internal `__consumer_offsets` topic and uses those committed positions when group members resume consumption or partitions are reassigned. In

practice, an application pursuing at-most-once behavior typically disables automatic commits and explicitly commits immediately after polling, before invoking its record-processing logic. The commit must cover the relevant returned batch before processing begins; committing later would instead permit replay after a failure. Confluent's consumer documentation describes the relationship between commit timing and delivery guarantees, including at-most-once behavior, and Apache Kafka documents committed offsets as the consumer group's saved position. See Confluent Consumer for Apache Kafka and Apache Kafka message delivery semantics.

QUESTION NO: 9

Your producer is producing at a very high rate and the batches are completely full each time. How can you improve the producer throughput? (select two)

A. Enable compression

B. Disable compression

C. Increase batch.size

D. Decrease batch.size

E. Decrease linger.ms
Increase linger.ms

ANSWER: A C

Explanation:

Enable compression and **Increase batch.size** are the appropriate throughput optimizations when a producer is already generating records quickly enough to fill batches. Kafka producers send records in batches, and compression is applied to those batches. Compressing a full batch reduces the number of bytes transmitted over the network and written to broker storage, which can improve effective producer throughput, particularly when network or disk bandwidth is a limiting resource. Kafka supports compression types including gzip, snappy, lz4, and zstd; the best choice depends on the desired balance between compression ratio and CPU use.

Increasing `batch.size` permits the producer to accumulate a larger amount of data for each partition before sending a request. Since the workload already fills batches, a larger batch capacity can produce larger requests and reduce per-request protocol, network, and broker processing overhead. This can increase the amount of data handled per unit of work, provided the producer has sufficient memory and the chosen batch size is suitable for the application's latency requirements. The Apache Kafka producer configuration documentation describes both `batch.size` and `compression.type`: Kafka producer batch.size configuration and Kafka producer compression.type configuration.

QUESTION NO: 10

A kafka topic has a replication factor of 3 and min.insync.replicas setting of 2. How many brokers can go down before a producer with acks=all can't produce?

A. 0

B. 2

C. 1

D. 3

ANSWER: C

Explanation:

With a replication factor of three and `min.insync.replicas=2`, one broker can go down while production using `acks=all` remains available. Initially, the partition has three replicas. If one broker hosting a replica becomes unavailable,

two replicas can remain in the in-sync replica set, which still satisfies the configured minimum of two. A producer using `acks=all` requires the leader to receive acknowledgement from all replicas currently in the ISR, and Kafka enforces that the ISR count is at least `min.insync.replicas` for this kind of produce request. Therefore, the partition continues accepting these writes after a single broker failure, assuming the remaining replicas are healthy and in sync. This configuration provides tolerance for one replica or broker failure while retaining the durability guarantee associated with acknowledging writes to at least two in-sync replicas. Confluent documents that `min.insync.replicas`, together with producer `acks=all`, controls the minimum number of replicas that must acknowledge a write. See the Confluent topic configuration reference and the Confluent producer acks configuration reference.

QUESTION NO: 11

How much should be the heap size of a broker in a production setup on a machine with

256 GB of RAM, in PLAINTEXT mode?

A. 4 GB

B. 128 GB

C. 16 GB

D. 512 MB

ANSWER: A

Explanation:

4 GB is the appropriate heap-size choice for this production broker running in PLAINTEXT mode. Kafka is deliberately designed not to place all available machine memory in the JVM heap. Instead, it relies heavily on the operating system's filesystem page cache for efficient disk reads and writes. On a host with 256 GB of RAM, assigning only a modest heap leaves the overwhelming majority of memory available to that page cache, which is important for Kafka's log-segment access and throughput characteristics.

A heap allocation of 4 GB is a practical baseline for a PLAINTEXT broker when no additional JVM-memory demands, such as TLS/SSL encryption buffers and associated connection processing, are present. Heap requirements can also grow with broker-specific factors such as unusually high partition counts, connection counts, request sizes, or enabled features; those should be monitored and used to tune the baseline in a real deployment. Kafka's design guidance emphasizes reserving memory for the OS cache rather than over-allocating Java heap. See Confluent's Apache Kafka deployment guidance and Apache Kafka's filesystem design documentation.

QUESTION NO: 12

What is true about partitions? (select two)

A. A broker can have a partition and its replica on its disk

B. You cannot have more partitions than the number of brokers in your cluster

C. A broker can have different partitions numbers for the same topic on its disk

D. Only out of sync replicas are replicas, the remaining partitions that are in sync are also leader

E. A partition has one replica that is a leader, while the other replicas are followers

ANSWER: C E

Explanation:

A broker can have different partitions numbers for the same topic on its disk because a Kafka topic is divided into independently assigned, numbered partitions. Kafka distributes a topic's partitions across the available brokers for scalability

and fault tolerance, but this does not limit a broker to holding only one partition from that topic. For example, a topic with many partitions can have several of its partition logs assigned to the same broker, subject to the cluster's replica-assignment and balancing considerations. Each partition is stored as its own log on the broker.

A partition has one replica that is a leader, while the other replicas are followers because Kafka uses a single leader replica as the point that handles client reads and writes for that partition. The follower replicas copy the leader's log, allowing Kafka to maintain redundant copies and support recovery if the leader broker becomes unavailable. When needed, Kafka elects an eligible in-sync follower to become the new leader, preserving the one-leader-per-partition model. This leader-and-follower arrangement is central to Kafka's partition replication design. See the Confluent Platform replication documentation and the Apache Kafka replication documentation.

QUESTION NO: 13

CORRECT TEXT

If I want to send binary data through the REST proxy to topic "test_binary", it needs to be base64 encoded. A consumer connecting directly into the Kafka topic A. "test_binary" will receive

B. binary data

C. avro data

D. json data

E. base64 encoded data, it will need to decode it

Answer: B

Explanation:

:

On the producer side, after receiving base64 data, the REST Proxy will convert it into bytes and then send that bytes payload to Kafka. Therefore consumers reading directly from Kafka will receive binary data.

A. binary data

B. avro data

C. json data

D. base64 encoded data, it will need to decode it

ANSWER: A

Explanation:

binary data is correct because Base64 is used only as a text-safe representation for transporting arbitrary bytes in the REST Proxy request payload. Kafka REST Proxy accepts binary records through its binary media type, where record key and value data are represented as Base64 strings in JSON. When the proxy processes the produce request, it decodes that Base64 representation and produces the resulting byte array as the Kafka record value.

Kafka itself stores records as bytes; it does not retain an indication that a producer used Base64 when submitting a REST request. Therefore, a consumer connected directly to `test_binary`, and configured with an appropriate byte-array deserializer such as Kafka's `ByteArrayDeserializer`, receives the original binary byte payload. The consumer does not need to perform Base64 decoding merely because the record was written through REST Proxy. This distinction between the REST transport encoding and the bytes persisted in Kafka is fundamental when interoperating between HTTP-based producers and native Kafka clients.

See Confluent's Kafka REST Proxy API documentation and Apache Kafka's serializer and deserializer documentation.

QUESTION NO: 14

A consumer has successfully processed records through offset 27 in a partition. What offset should it commit so that it resumes at offset 28 after a restart?

A. 27

B. 28

C. 26

D. The latest offset in the partition

ANSWER: B

Explanation:

The correct offset to commit is `28`. Kafka stores a consumer group's committed offset as the next record position that the consumer should read, not as the offset of the last record that was processed. Therefore, after successfully processing the record at offset 27, the consumer commits offset 28.

If the application instead commits 27, it can receive the record at offset 27 again after a restart, resulting in duplicate processing. This next-offset convention allows a committed offset to represent all records before that position as processed. See the Apache Kafka consumer configuration documentation and the Confluent Consumer for Apache Kafka documentation.

QUESTION NO: 15

Your configuration parameters for a Source connector and Connect worker are:

offset.flush.interval.ms=60000

offset.flush.timeout.ms=500

offset.storage.topic=connect-offsets

offset.storage.replication.factor=-1**Which four statements match the expected behavior?**(Select four.)

A. The connector will wait 60000ms before trying to commit offsets for tasks.

B. The connector will wait 500ms for offset data to be committed.

C. The connector will commit offsets to a topic called connect-offsets.

D. The offsets topic will use the broker default replication factor.

ANSWER: A B C D

Explanation:

For a distributed Kafka Connect worker, source-task offsets are written to the configured internal offsets topic. `offset.flush.interval.ms=60000` schedules the worker's regular offset flush attempts at a 60-second interval, so the connector will wait 60,000 ms before its next scheduled attempt to commit task offsets. `offset.flush.timeout.ms=500` limits how long the worker may wait for the offset flush/commit operation to complete; therefore, it waits up to 500 ms for offset data to be committed before the flush times out. The setting `offset.storage.topic=connect-offsets` identifies `connect-offsets` as the Kafka topic where source connector offsets are stored. Finally, setting `offset.storage.replication.factor=-1` directs Connect to use the Kafka broker's default replication factor when it creates the internal offsets topic, rather than explicitly requesting a particular replication factor. Together, these settings define the timing, timeout, destination, and topic-replication behavior for persisted source offsets. Confluent documents these worker-level offset-storage and flush settings in its Kafka Connect worker configuration reference. The corresponding Apache Kafka configuration definitions are also available in the Kafka Connect configuration documentation.

QUESTION NO: 16

Which statements are true about Schema Registry transitive compatibility modes? (select two)

A. `BACKWARD_TRANSITIVE` checks compatibility against all previous schema versions

B. `FULL_TRANSITIVE` enforces backward and forward compatibility against all previous versions

C. `FORWARD_TRANSITIVE` checks only the latest schema version

D. Transitive compatibility is configured on a Kafka broker

E. `NONE` is a transitive compatibility mode

ANSWER: A B

Explanation:

`BACKWARD_TRANSITIVE` requires a new schema to be backward compatible with every previously registered version of the subject, not only the latest version. This is useful when consumers might use older schema versions and must still be able to read data produced with the new schema.

`FULL_TRANSITIVE` requires both backward and forward compatibility against all prior versions. It is the most restrictive of these common compatibility modes because it protects both old consumers reading new data and new consumers reading old data across the subject's complete schema history. See the Confluent Schema Registry compatibility documentation.

QUESTION NO: 17

How do Kafka brokers ensure great performance between the producers and consumers?

(select two)

A. It compresses the messages as it writes to the disk

B. It leverages zero-copy optimisations to send data straight from the page-cache

C. It buffers the messages on disk, and sends messages from the disk reads

D. It transforms the messages into a binary format

E. It does not transform the messages

ANSWER: B E

Explanation:

Kafka achieves high producer-to-consumer throughput by minimizing both data copying and unnecessary message processing in the broker. “It leverages zero-copy optimisations to send data straight from the page-cache” is correct because Kafka can use the operating system’s page cache and zero-copy file-transfer mechanisms when serving consumer fetch requests. This allows data already cached from the log files to be transferred to the network more efficiently, avoiding extra copies through application memory and reducing CPU overhead. The page cache also means recently written or frequently read log data can often be served from RAM while retaining Kafka’s durable append-only log design.

“It does not transform the messages” is also correct in the relevant broker data path. Kafka brokers persist records as producer-sent record batches and can transmit those batches to consumers without deserializing and reserializing each individual record. In particular, compressed record batches can remain compressed while stored and transferred, with decompression generally deferred to the consumer. Preserving the batch representation reduces broker CPU work and helps Kafka scale throughput efficiently. Confluent’s architecture overview describes Kafka’s use of the page cache and zero-copy transfer, while Apache Kafka’s design documentation explains the efficient transfer of log data to consumers. See Confluent Kafka introduction and Apache Kafka design documentation.

QUESTION NO: 18

What is the default port that the KSQL server listens on?

A. 9092

B. 8088

C. 8083

D. 2181

ANSWER: B

Explanation:

The default port is 8088. ksqlDB Server (formerly called KSQL Server) exposes its REST API on port 8088 by default. Clients, including the ksqlDB CLI and applications making HTTP requests to the server, use this endpoint to submit statements, execute queries, and access server metadata. The listener is configured through the `listeners` property; when no custom listener configuration is supplied, the standard local endpoint is `http://localhost:8088`. In a deployed environment, this port must be reachable from authorized clients, subject to the network, load-balancer, and security configuration used by the organization. Confluent's ksqlDB configuration documentation identifies `listeners` as the setting for the REST API listener and gives port 8088 as its default. The ksqlDB quickstart likewise uses `http://localhost:8088` as the server URL, confirming the expected default endpoint. See the ksqlDB Server configuration reference and the ksqlDB quickstart.

QUESTION NO: 19 - (SIMULATION)

You are creating a Kafka Streams application to process retail data.

Match the input data streams with the appropriate Kafka Streams object.

ANSWER: See the explanation for the answer

Explanation:

Correct matches:

`Product` → `KTable`
`Customers` → `KTable`
`Orders_Placed` → `KStream`
`Shipment_Of_Orders` → `KStream`

`KTable` represents the latest state for each key and is appropriate for reference/entity data such as products and customers. `KStream` represents an ordered sequence of independent events, making it appropriate for order-placement and shipment events.

QUESTION NO: 20

You are sending messages with keys to a topic. To increase throughput, you decide to increase the number of partitions of the topic. Select all that apply.

A. All the existing records will get rebalanced among the partitions to balance load B. New records with the same key will get written to the partition where old records with that key were written

B. New records may get written to a different partition

C. Old records will stay in their partitions

ANSWER: B C

Explanation:

"New records may get written to a different partition" and "Old records will stay in their partitions" are correct. For keyed producer records, Kafka's default partitioning behavior uses a hash of the key and selects a partition from the topic's currently available partitions. When the partition count changes, the calculation can yield a different destination partition for subsequently produced records having the same key. Therefore, increasing partitions can disrupt the practical expectation that all records for one key will remain in one partition over the lifetime of a topic. Applications that depend on per-key ordering across the change must account for this before altering the partition count.

Existing data is not redistributed when partitions are added. Kafka partitions are immutable, append-only logs, and adding partitions creates additional empty logs for future appends; it does not move old records out of their original partition logs. Consumers can continue to read the historical records from those original partitions according to their offsets and retention settings. Confluent specifically cautions that increasing a topic's partition count changes key-to-partition mapping and can affect ordering for keyed messages. See Confluent documentation on increasing topic partitions and Apache Kafka partitioning documentation.

QUESTION NO: 21

A producer is sending messages with null key to a topic with 6 partitions using the

DefaultPartitioner. Where will the messages be stored?

A. Partition 5

B. Any of the topic partitions

C. The partition for the null key

D. Partition 0

ANSWER: B

Explanation:

Any of the topic partitions is correct. With a null message key, the producer does not have a key value to hash into one deterministic partition. The default producer partitioning behavior therefore selects from the topic's available partitions. In modern Apache Kafka clients, this is implemented using sticky partitioning for keyless records: the producer selects a partition and continues sending records there to efficiently fill a batch, then selects another available partition when appropriate, such as after the batch is completed. Consequently, no single numbered partition is guaranteed for all null-key messages; over time, records may be written to any of the six partitions. This batching-oriented behavior improves producer throughput while still allowing keyless records to be distributed across the topic partitions. The Apache Kafka DefaultPartitioner API describes that when neither an explicit partition nor a key is supplied, a sticky partition is chosen and changed when the batch is full. Confluent's producer configuration documentation also identifies the partitioner as the component that determines the destination partition for producer records. See the Apache Kafka DefaultPartitioner API documentation and Confluent producer partitioner configuration documentation.

QUESTION NO: 22

You are building a consumer application that processes events from a Kafka topic. What is the most important metric to monitor to ensure real-time processing?

A. UnderReplicatedPartitions

B. records-lag-max

C. MessagesInPerSec

D. BytesInPerSec

ANSWER: B

Explanation:

`records-lag-max` is the most important metric for assessing whether a consumer application is keeping up with real-time event production. It reports the largest current record lag across all partitions assigned to that consumer: in practical terms, it identifies the partition for which the consumer is furthest behind the latest available data. A consistently low value indicates that the consumer is processing records at approximately the rate they are produced. A growing value indicates that processing capacity, downstream dependencies, polling behavior, or consumer-group assignment should be investigated before latency becomes unacceptable.

Using the maximum rather than only an aggregate is especially important because a consumer can appear healthy overall while one busy or problematic partition accumulates substantial delay. Since Kafka ordering is guaranteed within a partition, lag on an individual partition directly represents the time-sensitive processing backlog for records in that partition. Monitoring this metric with alert thresholds and trends helps teams detect degraded real-time behavior early and scale or remediate the consumer application appropriately. Confluent describes consumer lag as the difference between the latest produced offset and the consumer's current offset, and Kafka exposes `records-lag-max` as a consumer metric. See Confluent consumer lag monitoring and Apache Kafka consumer documentation.

QUESTION NO: 23

(A consumer application runs once every two weeks and reads from a Kafka topic.

The last time the application ran, the last offset processed was 217.

The application is configured with auto.offset.reset=latest.

The current offsets in the topic start at 318 and end at 588.

Which offset will the application start reading from when it starts up for its next run?)

A. 0

B. 218

C. 318

D. 588

ANSWER: D

Explanation:

The correct offset is **588**. The consumer group's previously committed position is no longer valid because the retained log now begins at offset 318, while its prior processed position was 217. This situation can occur when Kafka's retention policy deletes older records. When a consumer cannot use its committed offset because it is out of range, it applies the policy configured by `auto.offset.reset`.

With `auto.offset.reset=latest`, the consumer is assigned the partition's current log-end offset. Kafka's end offset is an exclusive position: it is the offset at which the next produced record would be written, rather than the offset of the last existing record. Therefore, when the topic's current offset range ends at 588, seeking to `latest` positions the consumer at 588. It will not read the currently retained records before that position, and it will receive records subsequently produced beginning at offset 588.

This offset convention is used by Kafka consumer APIs, including `endOffsets`, which returns the offset of the next record after the last available record. See the Apache Kafka consumer configuration documentation and the KafkaConsumer endOffsets API documentation.

QUESTION NO: 24

If I want to have an extremely high confidence that leaders and replicas have my data, I should use

A. acks=all, replication factor=2, min.insync.replicas=1

B. acks=all, replication factor=3, min.insync.replicas=2

C. acks=all, replication factor=3, min.insync.replicas=1

ANSWER: B

Explanation:

acks=all, replication factor=3, min.insync.replicas=2 provides the intended durable-write configuration. With `acks=all` (also called `acks=-1`), a producer receives a successful acknowledgement only after the leader and every replica currently in the in-sync replica set have acknowledged the record. This ensures that a successful write has been replicated beyond the leader rather than residing only on a single broker.

A replication factor of three stores each partition on three brokers: one leader and two followers. Setting `min.insync.replicas=2` requires at least two of those replicas to be in the ISR before the leader can accept writes using `acks=all`. Therefore, at the point the producer receives success, the record is confirmed on at least two replicas. The topic can tolerate one replica becoming unavailable while continuing to accept strongly acknowledged writes, while retaining a further replica for resiliency and recovery. This combination is a standard production-oriented durability setting when high confidence in replicated data is required.

Confluent documents the interaction between producer acknowledgements and the minimum ISR requirement in its min.insync.replicas topic configuration reference and producer acknowledgements documentation.

QUESTION NO: 25

(You need to set alerts on key broker metrics to trigger notifications when a Kafka cluster is unhealthy.

What are three minimum broker metrics to monitor for cluster health?

Select three.)

A. kafka.controller:type=KafkaController,name=LastCommittedRecordOffset

B. kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec

C. kafka.controller:type=KafkaController,name=ActiveControllerCount

D. kafka.controller:type=KafkaController,name=OfflinePartitionsCount

E. kafka.controller:type=KafkaController,name=TopicsToDeleteCount

ANSWER: B C D

Explanation:

`kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec`, `kafka.controller:type=KafkaController,name=ActiveControllerCount`, and `kafka.controller:type=KafkaController,name=OfflinePartitionsCount` are core controller metrics for detecting serious Kafka cluster-health conditions. `ActiveControllerCount` verifies that exactly one broker is acting as the active controller. Alerting on a value other than one identifies a controller-election or controller-availability problem that can prevent normal cluster administration and partition leadership management.

`OfflinePartitionsCount` identifies partitions for which Kafka cannot provide a leader. A value above zero means affected partitions are unavailable for normal produce and consume operations, so it is an immediate availability signal. `UncleanLeaderElectionsPerSec` identifies leadership changes where an out-of-sync replica may have been elected because no in-sync replica was available. Such elections can discard records that had not replicated, making this an important data-durability and broker-failure alert.

Together, these metrics provide a minimum operational view across controller health, partition availability, and risk of data loss. Confluent's monitoring guidance documents these controller metrics and their expected healthy values; Apache Kafka also documents the controller JMX metrics: Confluent Platform Kafka Monitoring and Apache Kafka Monitoring.

QUESTION NO: 26

Which of the following Kafka Streams operators are stateful? (select all that apply)

A. flatmap

B. reduce

C. joining

D. count

E. peek

F. aggregate

ANSWER: B C D F

Explanation:

`reduce`, `joining`, `count`, and `aggregate` are stateful Kafka Streams operations. A stateful operation needs to retain data from records processed previously so it can calculate the result for a newly arriving record. For example, `count` maintains a running count for each key, while `reduce` repeatedly combines a new value with the current accumulated value for that key. `aggregate` similarly maintains an accumulator, allowing applications to construct arbitrary incremental results such as totals, collections, or domain-specific summary objects.

Kafka Streams joins also require state because records from one input stream or table must be correlated with records from another input. Kafka Streams stores the relevant records in local state stores, allowing it to look up matching keys or retain windowed records until corresponding records arrive. This local state is backed by Kafka changelog topics for fault tolerance and restoration, so state can be recovered after an application instance fails or is restarted. The Kafka Streams DSL documentation identifies aggregations and joins among the stateful transformation categories. See the Confluent Kafka Streams stateful transformations documentation and the Apache Kafka Streams DSL guide.