CCDAK Preparation Notes by Henry Naftulin

Note: These notes are not enough to pass the exam – these are only review notes. The information here is not guaranteed to be right, I tried to make it correct, but I cannot guarantee it. What helped me pass the exam, might not help other people. This document is still somewhat disorganized. Please e-mail me with suggestions or corrections: henryn73 at gmail.com

Broker- Kafka server is called a broker. The broker receives messages from producers, assigns offsets to them, and commits the messages to storage on disk and sends them to consumers. Could be part of a cluster, one is a controller (elected by zookeepers) - assigning partitions, monitoring for failures. Producers and consumers deal with partition leaders. Message retention – 7 days default: time period, size could be controlled: durable, per topic.

Zookeeper cluster - an ensemble, recommended odd # (3,5,..) server.X=hostname:peerPort:leaderPort (host1:2888:3888) and client port 2181.

Broker properties: broker.id - unique within a single Kafka cluster. **port** – listens on, 9092. **zookeeper.connect** - storing the broker metadata hostname:port/path (localhost:2181/idk). logs.dir, auto.create.topic.enable. Topic properties: num.partitions, log.retention.ms/log.retention.bites. Size of cluster – disks on brokers, bandwidth.

Producing messages -> creating a ProducerRecord, it must include the **topic** and a **value**; it could optionally include a key and/or a partition. We serialize the key and value objects to ByteArrays over the network. Next partitioner uses specified partition on prod, or murmur2 algo to decide partitions to send the message based on key or nothing if pation specified on Producer record. Producer has to have: bootstrap.servers, key.serializer, value.serializer. Fire-and-forget: just send message to server. **Synchronous** - send() and then get() to see if send was successful. **Asynchronous** - send(), examine the future async. Retriable messages can be configured to be retried automatically. Aks: 0 – we sent a message ok, 1 – leader replica got the message, all – all in-sync replicas got the message (at least min servers), or generic class. Partitioner interface includes configure, **partition**, and close methods. **compression.type** - compression type for a given topic (none, 'gzip', 'snappy', 'lz4', 'zstd'): producer compresses messages, sends compressed message to broker, consumer decompresses the message. In general, Kafka transfers data with zero copy on the transformation, so producer and consumer, not broker, has to do message transformation work; though if **SSL** is used, then zero copy optimization on the broker is lost.

Consumer: consumer in the group will receive messages from a different subset of the partitions in the topic. Consumer group share ownership of the subscribed topics: rebalances when add consumer or consumer shuts down or crashes. Consumer sends heartbeats to coordinator with every poll(), commit or programmatically. mandatory properties: **bootstrap.servers**, **key.deserializer**, and **value.deserializer.** *Example* consumer.subscribe(Collections.singletonList("cs.100")) or regex “cs*. During polling the method automatically handles partition rebalances, sends heartbeats, fetches data, and returns available data from the assigned partitions. close() - closes the network connections, sockets, immediately triggers rebalance. Calling wakeup() from external thread cleanly exists from the poll() loop,

but close() still needs to be called after. One consumer per thread is the rule. To run multiple consumers in the same group in one application, you will need to run each in its own thread. **fetch.min.bytes** - minimum amount of data from the broker when fetching. **fetch.max.wait.ms** wait until it has enough data to send before responding to the consumer. **fetch.max.wait.ms** how long to wait. **session.timeout.ms** – how long consumer is considered alive usually 3 times **heartbeat.interval.ms**. If it is lower, failed consumers detection and recovery sooner, but extra rebalances, possibly longer poll() and more garbage collection. **auto.offset.reset** where to starts reading a partition at "start": default is "latest" e.g. last committed, can be "earliest" and "none". **enable.auto.commit** - commit offsets automatically, and defaults to true. If true consider frequency parameter **auto.commit.interval.ms**. Offsets committed to topic __consumer_offsets with the committed offset for each partition. Auto commit, commitSync() but the application is blocked until the broker responds to the commit request limiting the throughput of the application; alternative consumer.commitAsync(). It is common to use the callback to log commit errors or to count them for metrics, but not for retry commits because of commit order. A simple pattern to get commit order right for asynchronous retries is to use a monotonically increasing sequence number (disregard lower seq failures). A consumer can either subscribe to topics (and be part of a consumer group), or assign itself to partitions, but not both at the same time. Consumer that assigns to partition is not part of consumer group. If manually requested partitions assignment – consumer is not notified of partitions that are added after the fact, so check consumer.partitionsFor() periodically, or bouncing the app whenever partitions are added. (Topic,partition,offset) triple uniquely identifies messages in Kafka.

One broker is a **controller**, in addition to the usual broker functionality, is responsible for electing partition leaders. Controller creates an ephemeral node in ZooKeeper called /controller. It uses higher epoch number to prevent a "split brain" scenario. If the brokers have rack information then assign the replicas for each partition to different racks if possible. Data retention policy is per topic. Compaction is for topics with key and values. If topic has null key, compaction will fail. For null value we will delete key completely. Messages are eligible for compaction only on inactive segments.

Kafka provides order guarantee of messages in a partition. Messages that are committed will not be lost if as at least one in-synch replica is alive. Consumers can only read messages that are committed. The topic-level configuration is **replication.factor**. At the broker level, you control the **default.replication.factor** for automatically created topics. Use parameter **unclean.leader.election.enable** (true) to select leader that might not be in-sync replica. Most important for reliability are **error-rate** and **retry-rate** per record (aggregated). Keep an eye on those, since error or retry rates going up can indicate an issue with the system. Also monitor the producer logs for errors that occur while sending events. On the consumer side, the most important metric is **consumer lag**.

ELT stands for Extract-Load-Transform - data pipeline does only minimal transformation (mostly around data type conversion). You will use Connect to connect Kafka to datastores that you did not write and whose code you cannot or will not modify. Connect **source** pulls data from the external datastore into Kafka; **sink** - push data from Kafka to an external store. Prepackaged Connect – no development, just configuration. If no connect exists Connect API - provides out-of-the-box features like configuration management, offset storage, parallelization, error handling, support for different data types, and

standard management REST APIs. Kafka Connect is a part of Apache Kafka. Kafka Connect runs as a cluster of worker processes. Install the connector plugins on the workers and then REST API to configure and manage connectors. Connectors start additional tasks - move large amounts of data in parallel and use the available resources on the worker nodes more efficiently. Source connector tasks just need to read data from the source system and provide Connect data objects to the worker processes. Sink connector tasks get connector data objects from the workers and are responsible for writing them to the target data system. Kafka Connect convertors – JSON, Avro converters. This allows users to choose the format in which data is stored in Kafka independent of the connectors they use. Connect stars on separate servers: • Determining how many tasks will run for the connector Deciding how to split the data-copying work between the tasks • Getting configurations for the tasks from the workers and passing it along(JDBC – lower of max.tasks, # of tables. Once it decides how many tasks will run, it will generate a configuration for each task—using both the connector configuration (e.g., connec tion.url) and a list of tables it assigns for each task to copy. The taskConfigs() method returns a list of maps (i.e., a configuration for each task we want to run). The workers are then responsible for starting the tasks and giving each one its own unique configuration so that it will copy a unique subset of tables from the database. Note that when you start the connector via the REST API, it may start on any node and subsequently the tasks it starts may also execute on any node. Tasks are responsible for actually getting the data in and out of Kafka. Source context includes an object that allows the source task to store the offsets of source records (e.g., in the file connector, the offsets are positions in the file; in the JDBC source connector, the offsets can be primary key IDs in a table). Kafka Connect's worker processes are the "container" processes that execute the connectors and tasks. They are responsible for handling the HTTP requests that define connectors and their configuration, as well as for storing the connector configuration, starting the connectors and their tasks, and passing the appropriate configurations along. If a worker process is stopped or crashes, other workers in a Connect cluster will recognize. JDBC connector allows 1 task per table.

Streams:

Data stream is an abstraction representing an unbounded dataset (infinite and ever growing).Event streams are ordered, Immutable data records, Event streams are replayable

Size of the window; widow advance interval. (tumbling window: advance = window size; move with ever record - sliding window). • How long the window remains updatable. **Processing of each event in isolation**. This is also known as a map/filter pattern because it is commonly used. Local store – same partition, persistence done with RocksDB. Mutli-phase, global state done by running local states and then writing results to new topic with single partition. External table lookup to be fast is done via cache and CDC stream (change data capture) that is applied to cache. **Streaming applications scale**: you will have as many tasks as you have partitions in the topics you are processing. If you want to process faster, add more threads. If you run out of resources on the server, start another instance on another server. Kafka will automatically coordinate work.

Local store: Kafka Streams currently requires that all topics that participate in a join operation will have the same number of partitions and be partitioned based on the join key. Kafka Streams repartitions by writing the events to a new topic with new keys and partitions. In multi-stage pipeline, the first and second sets of tasks can still run independently and in parallel because the first set of tasks writes data

into a topic at its own rate and the second set consumes from the topic and processes the events on its own. Stream windows

• If you are trying to solve an ingest problem use Kafka Connect. • If you are trying to solve a problem that requires low milliseconds actions use request-response patterns. • If you are building asynchronous microservices, you need a stream processing system. • If you are building a complex analytics engine, you also need a stream-processing system with great support for a local store—this time, not for maintenance of local caches and materialized views but rather to support advanced aggregations, windows, and joins that are otherwise difficult to implement. The APIs should include support for custom aggregations, window operations, and multiple join type.

Window name	Behavior	Short description
Tumbling	Time-based	Fixed-size, non-overlapping, gap-less windows […][…][…]
Hopping	Time-based	Fixed-size, overlapping windows [.[.[.].].]
Sliding	Time-based	Fixed-size, overlapping windows that work on differences between record timestamps. Mostly for joins
Session	Session-based	[…] [….] […. ..]

Join between stream & stream = stream; stream & table = stream; table & table = table

Schema:

Store schemas in the registry. Then store the identifier for the schema in the produced record. The consumers can then use the identifier to pull the record out of the schema registry and deserialize the data. Storing the schema in the registry and pulling it up—is done in the serializers and deserializers. The code that produces data to Kafka simply uses the Avro serializer just like it would any other. Two classes: specific – generated by Java, has types; Shema stored in _schemas topic. Avro schema is in JSON format that looks like:

```
{
   "namespace": "example.avro",
   "type": "record",
   "name": "User",
   "fields": [
      {"name": "name", "type": "string"},
      {"name": "favorite_number",  "type": ["null", "int"]},
      {"name": "favorite_color", "type": ["null", "string"]}
   ]
}
```

Avro messages are binary (most often used) or could be switched to JSON for debugging purposes.

Compatibility Type	Changes allowed	Check against which schemas	Upgrade first
BACKWARD (new consumer can read old data)	Delete fields Add optional fields	Last version	Consumers (new consumer ignores filed; new consumer uses default for new field)

Compatibility Type	Changes allowed	Check against which schemas	Upgrade first
FORWARD (old consumer can read new schema)	Add fields Delete optional fields	Last version	Producers (old consumer ignores new filed; old consumer gets default value for deleted field)
FULL	Add optional fields Delete optional fields	Last version	Any order
NONE	All changes are accepted	checking disabled	Depends
Breaking change	Enum		Changing e-num breaks the schema compatibility

REST

To send binary data to REST end-point producer needs to base 64 encode it, and consumer needs to decode it.

Odds and Ends:

Zookeeper math: tickTime – unit of time in miliseconds, initLimit – how many tickTimes for followers to connect to leader, syncLimit – how many tickTimes to sync with leader (otherwise zookeeper node will be dropped) Dynamic topic configurations are maintained in Zookeeper.

4GB is heap size of a broker in a production setup on a machine with 256 GB of RAM, in PLAINTEXT mode – small heap size, will be slightly larger with SSL.

Dynamic topic configurations are maintained in Zookeeper.

Broker will automatically create a topic with broker settings num.partitions and default.replication.factor

References: Kafaka The Difinitive Guide by Neha Narkhede at el, Apache Kafka CCDAK Exam Notes by Ashish Lahoti, Confluent-Kafka-Certification, and Practice tests for CCDAK on Udemy by Stephane Maarek.