Kafka Streams in Action - notes

**Chapter 2: Kafka Quickly**

* Kafka is a fault-tolerant, robust publish/subscribe system
* A single Kafka node is called a broker
* multiple Kafka servers make up a cluster
* Kafka stores messages written by producers in topics
* *Consumers subscribe to topics* and contact Kafka to see if messages are available in those subscribed topics

Kafka is a message broker:

Kafka is a publish/subscribe system, but it would be more precise to say that Kafka acts as a message broker. What is a broker? A broker is an intermediary that brings together *two parties that don’t necessarily know each other* for a mutually beneficial exchange or deal.

Kafka stores messages in topics and retrieves messages from topics. There’s *no direct connection between the producers and the consumers* of the messages.

Additionally, Kafka doesn’t keep any state regarding the producers or consumers. It acts solely as a

message clearinghouse.

The underlying technology of a Kafka topic is a log, which is a file that Kafka appends incoming records to.

To help manage the load of messages coming into a topic, Kafka uses partitions. One use of partitions is to bring data located on different machines together on the same server.

Kafka is a log:

The mechanism underlying Kafka is the log. In the context of Kafka (or any other distributed system), a log is "an append-only, totally ordered sequence of records *ordered by time*".

Topics in Kafka are logs that are segregated by topic name. You could almost think of topics as labeled logs. If the log is replicated among a cluster of machines, and a single machine goes down, it’s easy to bring that server back up: just replay the log file. The ability to recover from failure is precisely the role of a distributed commit log.

How logs work in Kafka:

When you install Kafka, one of the configuration settings is **log.dir** , which specifies where Kafka stores log data. *Each topic maps to a subdirectory under the specified log directory*. There will be as many subdirectories as there are topic partitions, with a format of **partition-name\_partition-number**.

Inside each directory is the log file where incoming messages are appended. Once the log files reach a certain size (either a number of records or size on disk), or when a configured time difference between message timestamps is reached, the log file is “rolled,” and Kafka appends incoming messages to a new log.

The logs directory is configured in the root at /logs.

/logs

/logs/topicA\_0 topicA has one partition.

/logs/topicB\_0 topicB has three partitions.

/logs/topicB\_1

/logs/topicB\_2

Logs and topics are highly connected concepts. You could say that *a topic is a log*, or that it represents a log. The topic name gives you a good handle on which log the messages sent to Kafka via producers will be stored in.

Kafka and partitions:

Partitions are a critical part of Kafka’s design. They’re essential for performance, and they guarantee that data with the same keys will be sent to the same consumer and in order.

As messages or records come in, they are written to a partition (assigned by producer) and appended in time order to the end of the that partition's log. NB Each partition's messages are in strictly increasing order, but there’s no order of messages across partitions.

Partitioning a topic essentially splits the data forwarded to a topic across parallel streams, and it’s key to how Kafka achieves its tremendous throughput.

A topic is a distributed log; each partition is similarly a log unto itself and follows the same rules. Kafka appends each incoming message to the end of the log, and all messages are strictly time-ordered. Each message has an offset number assigned to it. The order of messages across partitions isn’t guaranteed, but the order of messages within each partition is guaranteed.

Partitions group data by key:

Kafka works with data in key/value pairs. Message keys are used to determine which partition a message should go to. If the keys are null, the Kafka producer will write records to partitions chosen in a round-robin fashion.

The bytes of the key (if it is not null?) are used to calculate the hash. You obtain the partition by hashing the bytes of the key, modulus the number of partitions.

If the keys aren’t null, Kafka uses the following formula (shown in pseudocode) to determine which partition to send the key/value pair to:

HashCode.(key) % number of partitions

By using a deterministic approach to select a partition, records with the same key will always be sent to the same partition and in order. The default partitioner uses this approach; if you need a different strategy for selecting partitions, you can provide a custom partitioner.

Writing a custom partitioner:

Why would you want to write a custom partitioner? Of the several possible reasons, we’ll look at one simple case here—the use of composite keys.

Say a your key is: PurchaseKey(String customerId, Date transactionDate)

You need to ensure you send all transactions with the same customer ID to the same partition. The only way to do that is to only use the customer ID when determining the partition.

Example"

**public class** PurchaseKeyPartitioner **extends** DefaultPartitioner {  
 @Override  
 **public int** partition(String topic, Object key,  
 **byte**[] keyBytes, Object value,  
 **byte**[] valueBytes, Cluster cluster) {  
 Object newKey = **null**;  
 **if** (key != **null**) {  
 PurchaseKey purchaseKey = (PurchaseKey) key;  
 newKey = purchaseKey.getCustomerId();  
 keyBytes = ((String) newKey).getBytes(); }  
 return super.partition(topic, newKey, keyBytes, value,  
 valueBytes, cluster);  
 }  
}

This custom partitioner extends **DefaultPartitioner** . You could implement the Partitioner interface directly, but there’s existing logic in DefaultPartitioner that we’re using in this case. Keep in mind that when creating a custom partitioner, you aren’t limited to using only the key. Using the value alone, or the value in combination with the key, is valid

as well.

NOTE The Kafka API provides a Partitioner interface that you can use to write a custom partitioner. We won’t be covering writing a partitioner from scratch, but the principles are the same as those in the listing above.

Specifying a custom partitioner:

Now that you’ve written a custom partitioner, **you need to tell Kafka you want to use it** instead of the default partitioner. You specify a different partitioner when configuring the Kafka producer:

partitioner.class=bbejeck\_2.partitioner.PurchaseKeyPartitioner

By setting **a partitioner per producer instance**, you’re free to use any partitioner class for any producer.

WARNING You must exercise some caution when choosing the keys you use and when selecting parts of a key/value pair to partition on. **Make sure the key you choose has a fair distribution across all of your data.** Otherwise, you’ll end up with a data-skew problem, because most of your data will be located on just a few of your partitions.

Determining the correct number of partitions:

Choosing the number of partitions to use when creating a topic is part art and part science. One of the key considerations is *the amount of data flowing into a given topic*. More data implies more partitions for higher throughput. But as with anything in life, there are trade-offs.

**Increasing the number of partitions increases the number of TCP connections and open file handles**. Additionally, how long it takes to process an incoming record in a consumer will also determine throughput. If you have heavyweight processing in your consumer, adding more partitions may help, but ultimately the slower processing will hinder performance.

The distributed log:

Typically a Kafka production cluster environment includes several machines - in practice, you’ll always be working with a cluster of machines in Kafka.

When a topic is partitioned, Kafka doesn’t allocate those partitions on one machine—Kafka spreads those topics across several machines in the cluster. As Kafka appends records to a log, Kafka is distributing those records across several machines by partition.

Remember:

A single topic represents messages sent by a producer, and consisting of key/value pairs.

That topic will be partitioned using a partitioning strategy into multiple partitions: *a topic consists of multiple partitions*.

Those partitions will be allocated/spread across several machines in a cluster.

A single kafka node is called a broker, but there are muliple partitions per broker; so effectively multiple partitions per node; each node has a leader broker.

Bottom line: partitions aren’t all located on one machine but are spread out on brokers throughout the cluster.

ZooKeeper: leaders, followers, and replication:

How Kafka provides data availability in the face of machine failures:

Kafka has the notion of **leader and follower brokers**. In Kafka, *for each topic partition, one broker is chosen as the leader for the other brokers (the followers)*.

One of the chief duties of the leader is to **assign replication of topic partitions** to the follower brokers. Just as Kafka allocates partitions for a topic across the cluster, Kafka also replicates the partitions across machines. Before we go into the details of how leaders, followers, and replication work, we need to discuss the technology Kafka uses to

achieve this.

Apache ZooKeeper:

Apache ZooKeeper is integral to Kafka’s architecture, and it’s ZooKeeper that enables Kafka to have leader brokers and to do such things as track the replication of topics.

*ZooKeeper is a centralized service for maintaining configuration information, naming,providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.*

In a Kafka cluster, one of the brokers is “elected” as the controller. Kafka spreads partitions across different machines in the cluster. Topic partitions have a leader and follower(s) (the level of replication determines the degree of replication). When producing messages, Kafka sends the record to the broker that is the leader for the record’s partition.

https://www.quora.com/What-is-the-actual-role-of-Zookeeper-in-Kafka-What-benefits-will-I-miss-out-on-if-I-don%E2%80%99t-use-Zookeeper-and-Kafka-together/answer/Gwen-Shapira

Kafka uses Zookeeper for the following:

1. **Electing a controller**. The controller is one of the brokers and is responsible for maintaining the leader/follower relationship for all the partitions. When a node shuts down, it is the controller that tells other replicas to become partition leaders to replace the partition leaders on the node that is going away. Zookeeper is used to elect a controller, make sure there is only one and elect a new one it if it crashes.
2. **Cluster membership** - which brokers are alive and part of the cluster? this is also managed through ZooKeeper.
3. **Topic configuration** - which topics exist, how many partitions each has, where are the replicas, who is the preferred leader, what configuration overrides are set for each topic
4. (0.9.0) - Quotas - how much data is each client allowed to read and write
5. (0.9.0) - ACLs - who is allowed to read and write to which topic
6. (old high level consumer) - Which consumer groups exist, who are their members and what is the latest offset each group got from each partition.

Electing a controller:

Kafka uses ZooKeeper to elect the controller broker.

If the controlling broker fails or becomes unavailable for any reason, ZooKeeper elects a new controller from a set of brokers that are considered to be caught up with the leader (an in-sync replica [ISR]). The brokers that make up this set are dynamic, and ZooKeeper recognizes only brokers in this set for election as leader.

Replication:

Kafka replicates records among brokers to ensure data availability, should a broker in the cluster fail. You can set the level of replication for each topic or for all topics in the cluster.

The Kafka replication process is straightforward. Brokers that follow a topic partition consume messages from the topic-partition leader and append those records to their log. As discussed in the previous section, follower brokers that are caught up with their leader broker are considered to be ISRs. ISR brokers are eligible to be elected leader, should the current leader fail or become unavailable.

**Producers write records to the leader of a partition, and the followers read (copy data) from the leader.**

Controller responsibilities:

The controller broker is responsible for setting up leader/follower relationships for all partitions of a topic. If a Kafka node dies or is unresponsive (to ZooKeeper heart-beats), all of its assigned partitions (both leader and follower) are reassigned by the controller broker.

ZooKeeper is also involved in the following aspects of Kafka operations:

* Cluster membership—Joining a cluster and maintaining membership in a cluster. If a broker becomes unavailable, ZooKeeper removes the broker from cluster membership.
* Topic configuration—Keeping track of the topics in a cluster, which broker is the leader for a topic, how many partitions there are for a topic, and any specific configuration overrides for a topic.
* Access control—Identifying who can read from and write to particular topics.

This is why Kafka has a dependency on Apache ZooKeeper. It’s ZooKeeper that enables Kafka to have a leader broker with followers. The head broker has the critical role of assigning topic partitions for replication to the followers, as well as reassigning them when a member broker fails.

Log management:

The amount of space on spinning disks in a cluster is a finite resource, so it’s important for Kafka to remove messages over time. When it comes to removing old data in Kafka, there are two approaches: the traditional log-deletion approach, and compaction.

Deleting logs:

The log-deletion strategy is a two-phased approach: first, the logs are rolled into segments, and then the oldest segments are deleted. To manage the increasing size of the logs, Kafka rolls them into **segments**. The timing of log rolling is based on timestamps embedded in the messages. Kafka rolls a log when a new message arrives, and its timestamp is greater than the timestamp of the first message in the log plus the **log.roll.ms** (or log.roll.hours) configuration value. At that point, the log is rolled and a new segment is created as the new active log. The previous active segment is still used to retrieve messages for consumers.

Over time, the number of segments will continue to grow, and older segments will need to be deleted to make room for incoming data. To handle the deletion, you can specify how long to retain the segments - configurable by log.retention.ms, log.retention.minutes, log.retention.hours, log.retention.bytes.

The deletion of logs works well for non-keyed records, or records that stand alone. But if you have keyed data and expected updates, there’s another method that will suit your needs better - compaction.

Compacting logs:

Consider the case where you have keyed data, and you’re receiving updates for that data over time, meaning a new record with the same key will update the previous value.

Updating records by key is the behavior that compacted topics (logs) deliver.

Instead of taking a coarse-grained approach and deleting entire segments based on time or size, compaction is more fine-grained and **deletes old records per key** in a log. At a very high level, a log cleaner (a pool of threads) runs in the background, recopying log-segment files and removing records if there’s an occurrence later in the log with the same key.

So in the log after compactiont, the latest value for each key is retained, and the log is smaller in size.

This approach guarantees that the last record for a given key is in the log. You can specify log retention per topic, so it’s entirely possible to have some topics using time-based retention and other topics using compaction.

By default, the log cleaner is enabled. To use compaction for a topic, you’ll need to set the log.cleanup.policy=compact property when creating the topic.

Log compaction is a broad subject, and we’ve only touched on it here. For more information, see the Kafka documenta-

tion: <http://kafka.apache.org/documentation/#compaction>.

NOTE With a cleanup.policy of compact , you might wonder how you can remove a record from the log. With a compacted topic, deletion provides a null value for the given key, setting a tombstone marker. Any key with a null

value ensures that any prior record with the same key is removed, and the tombstone marker itself is removed after a period of time.

The key takeaway from this section is that if you have independent, standalone events or messages, use log deletion. If you have updates to events or messages, you’ll want to use log compaction.

Sending messages with producers

In Kafka,

* the **producer is the client used for sending messages**.
* Producers are used to send messages to Kafka.
* Producers don’t know which consumer will read the messages or when.
* A cluster of ZooKeeper nodes communicates with Kafka to maintain topic info and keep track of brokers in the cluster.

Properties properties = **new** Properties();  
 properties.put(**"bootstrap.servers"**, **"localhost:9092"**);  
 properties.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  
 properties.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  
 properties.put(**"acks"**, **"1"**);  
 properties.put(**"retries"**, **"3"**);  
 properties.put(**"compression.type"**, **"snappy"**);  
 properties.put(**"partitioner.class"**, PurchaseKeyPartitioner.**class**.getName());  
 PurchaseKey key = **new** PurchaseKey(**"12334568"**, **new** Date());  
 **try** (Producer<PurchaseKey, String> producer = **new** KafkaProducer<>(properties)) {  
 ProducerRecord<PurchaseKey, String> record =  
 **new** ProducerRecord<>(**"transactions"**, key, **"{\"item\":\"book\", \"price\":10.99}"**);  
 Callback callback = (metadata, exception) -> {  
 **if** (exception != **null**) {  
 System.***out***.println(**"Encountered exception "** + exception);  
 }  
 };  
 Future<RecordMetadata> sendFuture = producer.send(record, callback);  
 }

Kafka producers are thread-safe. All sends to Kafka are asynchronous— Producer.send returns immediately once the producer places the record in an internal buffer. The buffer sends records in batches. Depending on your configuration, there could be some blocking if you attempt to send a message while a producer’s buffer is full.

The Producer.send method depicted here takes a **Callback** instance. Once the leader broker acknowledges the record, the producer fires the **Callback.onComplete** method. Only one of the arguments will be non-null in the Callback.onComplete method. In this case, you’re only concerned with printing out the stacktrace in the event of error, so you check if the exception object is non-null. The **returned Future yields a RecordMetadata object** once the server acknowledges the record.

Producer properties

When you created the KafkaProducer instance, you passed a java.util.Properties parameter containing the configuration for the producer. The configuration of a KafkaProducer isn’t complicated, but there are key properties to consider when setting it up. These settings are where you’d specify a custom partitioner, for example.

* Bootstrap servers— bootstrap.servers is a comma-separated list of host:port values. Eventually the producer will use all the brokers in the cluster; this list is used for initially connecting to the cluster.
* Serialization— key.serializer and value.serializer instruct Kafka how to convert the keys and values into byte arrays. Internally, Kafka uses byte arrays for keys and values, so you need to provide Kafka with the correct serializers to convert objects to byte arrays before them sending across the wire.
* acks — acks specifies the minimum number of acknowledgments from a broker that the producer will wait for before considering a record send completed. Valid values for acks are all , 0 , and 1 . With a value of all , the producer will wait for a broker to receive confirmation that all followers have committed the record. When set to 1 , the broker writes the record to its log but doesn’t wait for any followers to acknowledge committing the record. A value of 0 means the producer won’t wait for any acknowledgments—this is mostly fire-and-forget.
* Retries—If sending a batch results in a failure, retries specifies the number of times to attempt to resend. If record order is important, you should consider setting max.in.flight.requests.per.connection to 1 to prevent the scenario of a second batch being sent successfully before a failed record being sent as the result a retry.
* Compression type— compression.type specifies what compression algorithm to apply, if any. If set, compression.type instructs the producer to compress a batch before sending. Note that it’s the entire batch that’s compressed, not individual records.
* Partitioner class— partitioner.class specifies the name of the class implementing the Partitioner interface.

For more information about producer configuration, see the Kafka documentation:

http://kafka.apache.org/documentation/#producerconfigs

Specifying partitions and timestamps

When you create a ProducerRecord , you have the option of specifying a partition, a timestamp, or both.

There are a number of overloaded constructors eg:

ProducerRecord(String topic, Integer partition, String key, String value)  
ProducerRecord(String topic, Integer partition, Long timestamp, String key, String value)

Specifying a partition

Why would you explicitly set the partition? There are a variety of business reasons why you might do so. Here’s one example. Suppose you have keyed data coming in, but it doesn’t matter which partition the records go to, because the consumers have logic to handle any value that the key might contain. Additionally, the distribution of the keys might not be even, and you want to ensure that all partitions receive roughly the same amount of data. Here’s a

rough implementation that would do this.

AtomicInteger partitionIndex = **new** AtomicInteger(0);  
**int** currentPartition = Math.abs(partitionIndex.getAndIncrement()) % numberPartitions;  
ProducerRecord<String, String> record = **new** ProducerRecord<>(**"topic"**, currentPartition, **"key"**, **"value"**);

Here, you use the Math.abs call, so you don’t have to keep track of the value of the

integer if it goes beyond Integer.MAX\_VALUE

Timestamps in Kafka

Kafka version 0.10 added timestamps to records. You set the timestamp when you create a ProducerRecord via this overloaded constructor call:

ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)

If you don’t set a timestamp, the producer will (using the current clock time) before sending the record to the Kafka broker. Timestamps are also affected by the log.message.timestamp.type broker configuration setting, which can be set to either **CreateTime** (the default) or **LogAppendTime** . Like many other broker settings, the value configured on the broker applies to all topics as a default value, but when you create a topic, you can specify a different value for that topic. If you specify LogAppendTime and the topic doesn’t override the broker’s configuration, the broker will overwrite the timestamp with the current time when it appends the record to the log. Otherwise, the timestamp from ProducerRecord is used.

Why would you choose one setting over another? **LogAppendTime is considered to be “processing time,”** and **CreateTime is considered to be “event time.”** Which you should use depends on your business requirements. You’ll need to decide whether you need to know when Kafka processed the record, or when the actual event occurred. In later chapters, you’ll see the important role timestamps have in controlling data flow in Kafka Streams.

Reading messages with consumers

**KafkaConsumer** is the client you’ll use to consume messages from Kafka. The KafkaConsumer class is straightforward to use, but there are a few operational considerations to take into account.

Remember that just as producers have no knowledge of the consumers, consumers read messages from Kafka with no knowledge of who produced the messages.

Managing offsets

**KafkaProducer is essentially stateless, but KafkaConsumer manages some state** by periodically committing the offsets of messages consumed from Kafka. **Offsets uniquely identify messages and represent the starting positions of messages in the log**. Consumers periodically need to commit the offsets of messages they have received.

Committing an offset has two implications for a consumer:

* Committing implies the consumer has fully processed the message.
* Committing also represents the starting point for that consumer in the case of failure or a restart.

If you have a new consumer instance or some failure has occurred, and the last committed offset isn’t available, where the consumer starts from will depend on your configuration:

* auto.offset.reset="earliest" —Messages will be retrieved starting at the earliest available offset. Any messages that haven’t yet been removed by the log-management process will be retrieved.
* auto.offset.reset="latest" —Messages will be retrieved from the latest offset, essentially only consuming messages from the point of joining the cluster.
* auto.offset.reset="none" —No reset strategy is specified. The broker throws an exception to the consumer.

Options for commiting offsets - Automatic offset commits

Automatic offset commits are enabled by default, and they’re represented by the enable.auto.commit property. There’s a companion configuration option, auto.commit.interval.ms , which specifies how often the consumer will commit offsets (the default value is 5 seconds). You should take care when adjusting this value. If it’s too small, it will increase network traffic; if it’s too large, it could result in the consumer receiving large amounts of repeated data in the event of a failure or restart.

Options for commiting offsets - Manual offset commits

There are two types of manually committed offsets—synchronous and asynchronous.

These are the synchronous commits:

consumer.commitSync()

consumer.commitSync(Map<TopicPartition, OffsetAndMetadata>)

The no-arg commitSync() method blocks until all offsets returned from the last retrieval (poll) succeed. This call applies to all subscribed topics and partitions. The other version takes a Map<TopicPartition, OffsetAndMetadata> parameter, and it commits only the offsets, partitions, and topics specified in the map. There are analogous consumer.commitAsync() methods that are completely asynchronous and return immediately. One of the overloaded methods accepts no arguments, and two of the consumer.commitAsync methods have an option to provide an

OffsetCommitCallback object, which is called when the commit has concluded either successfully or with an error. Providing a callback instance allows for asynchronous processing and error handling. The advantage of using manual commits is that it gives you direct control over when a record is considered processed.

Creating the consumer

Creating a consumer is similar to creating a producer. You supply a configuration in the form of a Java java.util.Properties object, and you get back a KafkaConsumer instance. This instance then subscribes to topics from a supplied list of topic names or by specifying a regular expression. Typically, you’ll run the consumer in a loop, where

you poll for a period specified in milliseconds.

A **ConsumerRecords<K, V>** object is the result of the polling. ConsumerRecords implements the Iterable interface, and each call to next() returns a ConsumerRecord object containing metadata about the message, in addition to the actual key and value.

After you’ve exhausted all of the ConsumerRecord objects returned from the last call to poll , you return to the top of the loop, polling again for the specified period. In practice, consumers are expected to run indefinitely in this manner, unless an error occurs or the application needs to be shut down and restarted (this is where committed offsets come into play—on reboot, the consumer will pick up where it left off).

Consumers and partitions

You’ll generally want multiple consumer instances—**one for each partition of a topic**. It’s possible to have one consumer read from multiple partitions, but it’s not uncommon to have a **thread pool with as many threads as there are partitions**, and with **each thread running a consumer that’s assigned to one partition**.

This **consumer-per-partition pattern** maximizes throughput, but if you spread your consumers across multiple applications or machines, the total thread count across all instances shouldn’t exceed the total number of partitions in the topic. Any threads in excess of the total partition count will be idle. If a consumer fails, the leader broker assigns its partitions to another active consumer.

The leader broker assigns topic partitions to all available consumers with the same **group.id** . The group.id is a configuration setting that **identifies the consumer as belonging to a consumer group**—that way, consumers don’t need to reside on the same machine. In fact, it’s **probably preferable to have your consumers spread out across a few machines**. That way, in the case of one machine failing, the leader broker can assign topic partitions to consumers on good machines.

Rebalancing

The **process of adding and removing topic-partition assignments to consumers** described in the previous section is called rebalancing. **Topic-partition assignments to a consumer aren’t static—they’re dynamic.** As you add consumers with the same group ID, some of the current topic-partition assignments are taken from active consumers and given to the new consumers. This reassignment process continues until every partition has been assigned to a consumer that’s reading data. After that equilibrium point, any additional consumers will remain idle. When consumers leave the group for whatever reason, their topic-partition assignments are reassigned to other consumers.

Finer-grained consumer assignment

Although Kafka will balance the load of topic-partitions across all consumers, the assignment of the topic and partition isn’t deterministic. You won’t know what topic-partition pairings each consumer will receive.

KafkaConsumer has methods that allow you to subscribe to a particular topic and partition:

TopicPartition fooTopicPartition\_0 = new TopicPartition("foo", 0);

TopicPartition barTopicPartition\_0 = new TopicPartition("bar", 0);

consumer.assign(Arrays.asList(fooTopicPartition\_0, barTopicPartition\_0));

There are trade-offs to consider when using manual topic-partition assignment:

* Failures won’t result in topic partitions being reassigned, even for consumers with the same group ID. Any changes in assignments will require another call to consumer.assign .
* The group specified by the consumer is used for committing, but because each consumer will be acting independently, it’s a good idea to give each consumer a unique group ID.

Consumer example

**class** ThreadedConsumerExample {  
 **public void** startConsuming() {  
 executorService = Executors.newFixedThreadPool(numberPartitions);  
 Properties properties = getConsumerProps();  
 **for** (**int** i = 0; i < numberPartitions; i++) {  
 Runnable consumerThread = getConsumerThread(properties);  
 executorService.submit(consumerThread);  
 }  
 }  
  
 **private** Runnable getConsumerThread(Properties properties) {  
 **return** () -> {  
 Consumer<String, String> consumer = **null**;  
 **try** {  
 consumer = **new KafkaConsumer**<>(properties);  
 consumer.subscribe(Collections.singletonList(**"test-topic"**));  
 **while** (!doneConsuming) {  
 ConsumerRecords<String, String> records = consumer.poll(5000);  
 **for** (ConsumerRecord<String, String> record : records) {  
 String message = String.format(**"Consumed: key = %s value = %s with offset = %d partition = %d"**, record.key(), record.value(), record.offset(), record.partition());  
 System.***out***.println(message);  
 }  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **if** (consumer != **null**) {  
 consumer.close();  
 }  
 }  
 };  
 }  
}

Installing and running Kafka

Because Kafka is a Scala project, each release comes in two versions: one for Scala 2.11 and another for Scala 2.12.

To install Kafka, extract the .tgz file found in the book’s source code repo (source code can be found on the

book’s website here: https://manning.com/books/kafka-streams-in-action), to somewhere in the libs folder on your machine.

NOTE The binary distribution of Kafka includes Apache ZooKeeper, so no extra installation work is required.

Kafka local configuration

Running Kafka locally on your machine requires minimal configuration if you accept the default values. By default, Kafka uses port 9092, and ZooKeeper uses port 2181. Assuming you have no applications already using those ports, you’re all set.

Kafka by default writes its logs to /tmp/kafka-logs, and ZooKeeper uses /tmp/zookeeper for its log storage.

To change the Kafka logs directory, cd into <kafka-install-dir>/config and open the server.properties file. Find the log.dirs entry, and change the value to what you’d rather use. In the same directory, open the zookeeper.properties file and change the dataDir entry.

Keep in mind that these “logs” are the actual data used by Kafka and ZooKeeper, and not application-level logs that track the application’s behavior. The application logs are found in the <kafka-install-dir>/logs directory.

Running Kafka

Kafka is simple to get started. Because ZooKeeper is essential for the Kafka cluster to function properly (ZooKeeper determines the leader broker, holds topic information, performs health checks on cluster members, and so on), you’ll need to start ZooKeeper before starting Kafka.

Starting ZooKeeper

To start ZooKeeper, open a command prompt and enter the following command:

bin/zookeeper-server-start.sh config/zookeeper.properties

Starting Kafka

To start Kafka, open another command prompt and type this command:

bin/Kafka-server-start.sh config/server.properties

NB ZooKeeper is essential for Kafka to run, so it’s important to reverse the order when shutting down: stop Kafka first, and then stop ZooKeeper. To stop Kafka, you can press Ctrl-C from the terminal Kafka is running in, or run kafka-server-stop.sh from another terminal. The same goes for ZooKeeper, except the shutdown script is zookeeper-server-stop.sh

Sending your first message

Before you send a message, you’ll need to define a topic for a producer to send a message to.

YOUR FIRST TOPIC

Creating a topic in Kafka is simple. It’s just a matter of running a script with some configuration parameters. The configuration is easy, but the settings you provide have broad performance implications.

By default, Kafka is configured to autocreate topics, meaning that if you attempt to send to or read from a nonexistent topic, the Kafka broker will create one for you (using default configurations in the server.properties file). It’s rarely a good practice to rely on the broker to create topics, even in development, because the first produce/consume attempt will fail, as it takes time for the metadata about the topic’s

existence to propagate. Be sure to always proactively to create your topics.

CREATING A TOPIC

To create a topic, you need to run the kafka-topics.sh script. Open a terminal window and run this command:

bin/kafka-topics.sh --create --topic first-topic --replication-factor 1 --partitions 1 --zookeeper localhost:2181

* replication-factor —This flag determines how many copies of the message the leader broker distributes in the cluster. In this case, with a replication factor of 1 , no copies will be made. Just the original message will reside in Kafka. A replication factor of 1 is fine for a quick demo or prototype, but in practice you’ll almost always want a replication factor of 2 or 3 to provide data availability in the case of machine failures.
* partitions —This flag specifies the number of partitions that the topic will use. Again, just one partition is fine here, but if you have greater load, you’ll certainly want more partitions. Determining the correct number of partitions is not an exact science

SENDING A MESSAGE

Sending a message in Kafka generally involves writing a producer client, but Kafka also comes with a handy script called kafka-console-producer that allows you to send a message from a terminal window.

To send your first message, run the following command:

# command assumes running from bin directory

./kafka-console-producer.sh --topic first-topic --broker-list localhost:9092

There are several options for configuring the console producer, but for now we’ll only use the required ones: the topic to send the message to, and a list of Kafka brokers to connect to (in this case, just the one on your local machine).

**Starting a console producer is a “blocking script,”** so after executing the preceding command, you enter some text and press Enter. You can send as many messages as you like, but for our demo purposes you can type a single message, “the quick brown fox jumped over the lazy dog,” press Enter, and then press Ctrl-C to exit the producer.

READING A MESSAGE

Kafka also provides a console consumer for reading messages from the command line. The console consumer is similar to the console producer: once started, it will keep reading messages from the topic until the script is stopped by you (with Ctrl-C).

To launch the console consumer, run this command:

bin/kafka-console-consumer.sh --topic first-topic --bootstrap-server localhost:9092 --from-beginning

After starting the console consumer, you should see something like figure 2.21 in your

terminal.

The --from-beginning parameter specifies that you’ll receive any message not deleted from that topic. The console consumer won’t have any committed offsets, so if you didn’t have the --from-beginning setting, you’d only get messages sent after the console consumer had started.