Chapter 9. Administering Kafka

Kafka provides several command line utilities that are useful for making administrative changes to your clusters. The tools are implemented in Java classes, and a set of scripts are provided to call those classes properly. These tools provide basic functions, but you may find they are lacking for more complex operations. This chapter will describe the tools that are available as part of the Apache Kafka open source project. More information about advanced tools that have been developed in the community, outside of the core project, can be found in Appendix B

AUTHORIZING ADMIN OPERATIONS

While Apache Kafka implements authentication and authorization to control topic operations, most cluster operations are not yet supported. This means that these CLI tools can be used without any authentication required, which will allow operations such as topic changes to be executed with no security check or audit. This functionality is under development and should be added soon.

Topic Operations

The *kafka-topics.sh* tool provides easy access to most topic operations (configuration changes have been deprecated and moved to the *kafka-configs.sh* tool). It will allow you to create, modify, delete, and list information about topics in the cluster. For all usages of this command, you are required to provide the Zookeeper connect string for the cluster with the `--zookeeper` argument. In the examples that follow, the zookeeper connect string is assumed to be `zoo1.example.com:2181/kafka-cluster`

CHECK THE VERSION

Many of the command line tools for Kafka operate directly on the metadata stored in Zookeeper, rather than connecting to the brokers themselves. For this reason, it is important to make sure the version of the tools that you are using matches the version of the brokers in the cluster. The safest approach is to run the tools on the Kafka brokers themselves, using the deployed version.

Creating a New Topic

In order to create a new topic in the cluster, there are three required arguments. These arguments must be provided, even though some of them have broker-level defaults configured already: * **Topic name:** The name of the topic that you wish to create *

Replication Factor: The number of replicas of the topic to maintain within the cluster * **Partitions:** The number of partitions to create for the topic

SPECIFYING TOPIC CONFIGURATIONS

It is also possible to explicitly set the replicas for a topic during creation, or set configuration overrides for the topic. Neither of these operations will be covered here. Configuration overrides can be found later in this chapter, and they can be provided to *kafka-topics.sh* using the `--config` command line parameter. Partiton reassignment is also covered later in this chapter.

Topic names may contain alphanumeric characters, as well as underscores, dashes, and periods.

NAMING TOPICS

It is permitted, but not recommended, to have topic names that start with two underscores. Topics of this form are considered internal topics for the cluster (such as the `__consumer_offsets` topic for consumer group offset storage). It is also not recommended that the usage of both periods and underscores be mixed in a single cluster. This is because when the topic names are used in metric names inside Kafka, periods are changed to underscores (e.g. “topic.1” becomes “topic_1” in metrics).

Execute the command as follows:

```
kafka-topics.sh --zookeeper <zookeeper connect> --create --topic <string> --replication-factor <integer> --partitions
```

The command will cause the cluster to create a topic with the specified name and number of partitions. For each partition, the cluster will select the specified number of replicas appropriately. This means that if the cluster is set up for rack-aware replica assignment, the replicas for each partition will be in separate racks. If rack-aware assignment is not desired, specify the `--disable-rack-aware` command line argument.

Example: Create a topic named “my-topic” with 8 partitions that have 2 replicas each

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --create --topic my-topic --replication-factor 2 --p
Created topic "my-topic".
#
```

SKIPPING ERRORS FOR EXISTING TOPICS

When using this script in automation, you may want to use the `--if-not-exists` argument, which will not return an error if the topic already exists.

Adding Partitions

It is sometimes necessary to increase the number of partitions for a topic. Partitions are the way topics are scaled and replicated across a cluster, and the most common reason to increase the partition count is to spread out a topic further, or decrease the throughput for a single partition. Topics may also be increased if a consumer needs to expand to run more copies in a single group, as a partition can only be consumed by a single member in a group.

ADJUSTING KEYED TOPICS

Topics that are produced with keyed messages can be very difficult to add partitions to, from a consumer's point of view. This is because the mapping of keys to partitions will change when the number of partitions are changed. It is advisable to set the partition count for a keyed topic to be correct when it is created, in order to avoid having to resize it.

SKIPPING ERRORS FOR NON-EXISTANT TOPICS

While a `--if-exists` argument is provided for the `--alter` command, which will cause the command to not return an error if the topic being changed does not exist, it has less usefulness for automation as it can mask problems where a topic does not exist that should have been created.

Example: Increase the number of partitions for a topic named "my-topic" to 16

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --alter --topic my-topic --partitions 16
WARNING: If partitions are increased for a topic that has a key, the partition logic or ordering of the messages will
Adding partitions succeeded!
#
```

REDUCING PARTITION COUNTS

It is not possible to reduce the number of partitions for a topic. This is because it is unclear what the correct thing to do with the data from that partition is, and trying to keep the data and redistribute it to other partitions would be very complex in any implementation. Should you need to reduce the number of partitions, you will need to delete the topic and recreate it.

Deleting a Topic

If a topic in the cluster is no longer needed, it still uses some disk space, as well as open filehandles, as long as it exists. It can be deleted in order to free up these resources. In order to perform this action, the brokers in the cluster must have been configured with the delete.topic.enable option set to true. If this option has been set to false, then the request to delete the topic will be ignored.

WARNING

Deleting a topic will result in all messages that are in that topic being discarded. This is not a reversible operation, so make sure it executed carefully.

Example: Delete the topic named "my-topic"

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --delete --topic my-topic
Topic my-topic is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
#
```

Listing All Topics in a Cluster

The topics tool can list all topics in a cluster. The list is formatted with one topic per line, in no particular order.

Example: List topics in the cluster

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --list
my-topic - marked for deletion
other-topic
#
```

Describing Topic Details

It is also possible to get detailed information on one or more topics in the cluster. The output includes the partition count, topic configuration overrides, and a listing of each partition with its replica assignments. This can be limited to a single topic by providing a `--topic` argument to the command.

Example: Describe all topics in the cluster

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --describe
Topic:other-topic       PartitionCount:8        ReplicationFactor:2     Configs:
        Topic: other-topic      Partition: 0    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: other-topic      Partition: 1    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: other-topic      Partition: 2    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: other-topic      Partition: 3    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: other-topic      Partition: 4    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: other-topic      Partition: 5    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: other-topic      Partition: 6    Leader: 1       Replicas: 1,0   Isr: 1,0
        Topic: other-topic      Partition: 7    Leader: 0       Replicas: 0,1   Isr: 0,1
#
```

The describe command also has several useful options for filtering the output. These can be helpful for diagnosing cluster issues. For each of these, do not specify the `--topic` argument (as the intention is to find all topics or partitions in a cluster that match the criteria). These options will not work with the list command (detailed in the last section).

In order to find all topics that have configuration overrides, use the `--topics-with-overrides` argument. This will describe only the topics that have configurations set that differ from the cluster defaults.

To find partitions that have problems, there are two filters. The `--under-replicated-partitions` argument will show all partitions where one or more of the replicas for the partition are not in-sync with the leader. The `--unavailable-partitions` argument shows all partitions where there is no leader at all. This is a more serious situation that means that the partition is currently offline and unavailable for produce or consume clients.

Example: Show Under-replicated Partitions:

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --describe --under-replicated-partitions
        Topic: other-topic      Partition: 2    Leader: 0       Replicas: 1,0   Isr: 0
        Topic: other-topic      Partition: 4    Leader: 0       Replicas: 1,0   Isr: 0
#
```

Consumer Groups

Consumer groups in Kafka are managed in two places: for older consumers the information is maintained in Zookeeper, while for the new consumer it is maintained within the Kafka brokers. The *kafka-consumer-groups.sh* tool can be used to list and describe both types of groups. It can also be used to delete consumer groups and offset information, but only for groups running under the old consumer (maintained in Zookeeper). When working with older consumer groups, you will access the Kafka cluster by specifying the `--zookeeper` command line parameter to the tool. For new consumer groups, you will need to use the `--bootstrap-server` parameter with the hostname and port number of the Kafka broker to connect to instead.

List and Describe Groups

To list consumer groups using the older consumer clients, execute with the `--zookeeper` and `--list` parameters. For the new consumer, use the `--bootstrap-server`, `--list`, and `--new-consumer` parameters.

Example: List old consumer groups

```
# kafka-consumer-groups.sh --zookeeper zoo1.example.com:2181/kafka-cluster --list
console-consumer-79697
myconsumer
#
```

Example: List new consumer groups

```
# kafka-consumer-groups.sh --new-consumer --bootstrap-server kafka1.example.com:9092/kafka-cluster --list
kafka-python-test
my-new-consumer
#
```

For any group listed, you can also get more details on that group by changing the `--list` parameter to `--describe` and adding the `--group` parameter. This will list all the topics that the group is consuming, as well as the offsets for each topic partition.

Example: Get consumer group details for the old consumer group named “testgroup”

```
# kafka-consumer-groups.sh --zookeeper zoo1.example.com:2181/kafka-cluster --describe --group testgroup
GROUP                          TOPIC                          PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
myconsumer                     my-topic                       0          1688            1688            0
myconsumer                     my-topic                       1          1418            1418            0
myconsumer                     my-topic                       2          1314            1315            1
myconsumer                     my-topic                       3          2012            2012            0
myconsumer                     my-topic                       4          1089            1089            0
myconsumer                     my-topic                       5          1429            1432            3
myconsumer                     my-topic                       6          1634            1634            0
myconsumer                     my-topic                       7          2261            2261            0
#
```

In this output, the following fields are provided:

Field	Description
GROUP	the name of the consumer group
TOPIC	the name of the topic being consumed
PARTITION	the ID number of the partition being consumed
CURRENT-OFFSET	the last offset committed by the consumer group for this topic partition. This is the position of the consumer within the partition.
LOG-END-OFFSET	the current high water mark offset from the broker for the topic partition. This is the offset of the last message produced and committed to the cluster.
LAG	the difference between the consumer Current-Offset and the broker Log-End-Offset for this topic partition
OWNER	the member of the consumer group that is currently consuming this topic partition. This is an arbitrary ID provided by the group member, and does not necessarily contain the hostname of the consumer.

Delete Group

Deletion of consumer groups is only supported for old consumer clients. This will remove the entire group from Zookeeper, including all stored offsets for all topics that the group is consuming. In order to perform this action, all consumers in the group should be shut down. If this step is not performed first, you may have undefined behavior from the consumer as the Zookeeper metadata for the group will be removed while it is using it.

Example: Delete the consumer group named “testgroup”

```
# kafka-consumer-groups.sh --zookeeper zoo1.example.com:2181/kafka-cluster --delete --group testgroup
Deleted all consumer group information for group testgroup in zookeeper.
#
```

It is also possible to use the same command to delete offsets for a single topic that the group is consuming without deleting the entire group. Again, it is recommended that the consumer group be stopped, or configured to not consume the topic to be deleted, before performing this action.

Example: Delete the offsets for the topic “my-topic” from the consumer group named “testgroup”

```
# kafka-consumer-groups.sh --zookeeper zoo1.example.com:2181/kafka-cluster --delete --group testgroup --topic my-topic
Deleted consumer group information for group testgroup topic my-topic in zookeeper.
#
```

Offset Management

In addition to displaying and deleting the offsets for a consumer group using the old consumer client, it is also possible to retrieve the offsets and store new offsets in a batch. This is useful for resetting the offsets for a consumer when there is a problem that requires messages be reread, or for advancing offsets past a message that the consumer is having a problem with (such as if there is a badly formatted message that the consumer cannot handle).

MANAGING OFFSETS COMMITTED TO KAFKA

There is currently no tool available to manage the offsets for a consumer client that is committing offsets to Kafka. This function is only available for consumers that are committing offsets to Zookeeper. In order to manage offsets for a group that is committing to Kafka, you must use the APIs available in the client to commit offsets for the group.

EXPORT OFFSETS

There is no named script to export offsets, but we are able to use the *kafka-run-class.sh* script to execute the underlying Java class for the tool in the proper environment. Exporting offsets will produce a file that contains each topic partition for the group and it’s offsets in a defined format that the import tool can read. The file that is created will have one topic partiton per line, with the following format: /consumers/GROUPNAME/offsets/topic/TOPICNAME/PARTITIONID-0:OFFSET

Example: Export the offsets for the consumer group named “testgroup” to a file named “offsets”

```
# kafka-run-class.sh kafka.tools.ExportZkOffsets --zkconnect zoo1.example.com:2181/kafka-cluster --group testgroup --o
# cat offsets
/consumers/testgroup/offsets/my-topic/0:8905
/consumers/testgroup/offsets/my-topic/1:8915
/consumers/testgroup/offsets/my-topic/2:9845
/consumers/testgroup/offsets/my-topic/3:8072
/consumers/testgroup/offsets/my-topic/4:8008
/consumers/testgroup/offsets/my-topic/5:8319
/consumers/testgroup/offsets/my-topic/6:8102
/consumers/testgroup/offsets/my-topic/7:12739
#
```

IMPORT OFFSETS

The import offset tool is the opposite of exporting. It takes the file produced by exporting offsets in the previous section and uses it to set the current offsets for the consumer group. A common practice is to export the current offsets for the consumer group, make a copy of the file (such that you preserve a backup), and edit the copy to replace the offsets with the desired values. Note that for the import command, the `--group` option is not used. This is because the consumer group name is embedded in the file to be imported.

STOP CONSUMERS FIRST

Before performing this step, it is important the all consumers in the group are stopped. They will not read the new offsets if they are written while the consumer group is active. The consumers will just overwrite the imported offsets.

Example: Import the offses for the consumer group named “testgroup” from a file named “offsets”

```
# kafka-run-class.sh kafka.tools.ImportZkOffsets --zkconnect zoo1.example.com:2181/kafka-cluster --input-file offsets
#
```

Dynamic Configuration Changes

Configurations can be overridden while the cluster is running for topics and for client quotas. There is the intention to add more dynamic configurations in the future, which is why these changes have been put in a separate CLI tool, *kafka-configs.sh*. This allows you to set configurations for specific topics and client IDs. Once set, these configurations are permanent for the cluster. They are stored in Zookeeper, and they are read by each broker when it starts. In tools and documentation, dynamic configurations like this are referred to as “per-topic” or “per-client” configurations, as well as “overrides”.

As with the previous tools, you are required to provide the Zookeeper connect string for the cluster with the `--zookeeper` argument. In the examples that follow, the zookeeper connect string is assumed to be “zoo1.example.com:2181/kafka-cluster”.

Overriding Topic Configuration Defaults

There are many configurations that apply to topics that can be changed for individual topics in order to accommodate different use cases within a single cluster. Most of these configurations have a default specified in the broker configuration, which will apply unless an override is set.

The format of the command to change a topic configuration is:

```
kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster --alter --entity-type topics --entity-name <topic nam
```

The valid configurations (keys) for topics are:

| ============================== | Configuration Key | Description | cleanup.policy | If set to “compact”, this messages in this topic will be discarded such that only the most recent message with a given key is retained (log compacted) | compression.type | The compression type used by the broker when writing message batches for this topic to disk. Current values are “gzip”, “snappy”, and “lz4” | delete.retention.ms | How long, in milliseconds, delete tombstones will be retained for this topic. Only valid for log compacted topics | file.delete.delay.ms | How long, in milliseconds, to wait before deleting log segments and indices for this topic from disk | flush.messages | How many messages are received before forcing a flush of this topic’s messages to disk | flush.ms | How long, in milliseconds, before forcing a flush of this topic’s messages to disk | index.interval.bytes | How many bytes of messages can be produced between entries in the log segment’s index | max.message.bytes | The maximum size of a single message for this topic, in bytes | message.format.version | The message format version that the broker will use when writing messages to disk. Must be a valid API version number (e.g. “0.10.0”) | message.timestamp.difference.max.ms | The maximum allowed difference, in milliseconds, between the message timestamp and the broker timestamp when the message is received. This is only valid if the `message.timestamp.type` is set to “CreateTime” | message.timestamp.type | Which timestamp to use when writing messages to disk. Current values are “CreateTime” for the timestamp specified by the client and “LogAppendTime” for the time when the message is written to the partition by the broker | min.cleanable.dirty.ratio | How frequently the log compactor will attempt to compact partitions for this topic, expressed as a ratio of the number of uncompacted log segments to the total number of log segments. Only valid for log compacted topics | min.insync.replicas | The minimum number of replicas that must be in sync for a partition of the topic to be considered available | preallocate | If set to “true”, log segments for this topic should be preallocated when a new segment is rolled | retention.bytes | The amount of messages, in bytes, to retain for this topic | retention.ms | How long messages should be retained for this topic, in milliseconds | segment.bytes | The amount of messages, in bytes, that should be written to a single log segment in a partition | segment.index.bytes | The maximum size, in bytes, of a single log segment index | segment.jitter.ms | A maximum number of milliseconds that is randomized and added to `segment.ms` when rolling log segments | segment.ms | How frequently, in milliseconds, the log segment for each partition should be rotated | unclean.leader.election.enable | If set to false, unclean leader elections will not be permitted for this topic | ==============================

Example: Set the retention for the topic named “my-topic” to 1 hour (3600000 ms)

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster --alter --entity-type topics --entity-name my-topic
Updated config for topic: "my-topic".
#
```

Overriding Client Configuration Defaults

For Kafka clients, the only configurations that can be overridden are the producer and consumer quotas. These are both a rate, in bytes per second, that all clients with the specified client ID are allowed to either produce or consume on a per-broker basis. This means that if you have 5 brokers in a cluster, and you specify a producer quota of 10 MB/sec for a client, that client will be allowed to produce 10 MB/sec on each broker at the same time for a total of 50 MB/sec.

CLIENT ID VS. CONSUMER GROUP

The client ID is not necessarily the same as the consumer group name. Consumers can set their own client ID, and you may have many consumers that are in different groups that specify the same client ID. It is considered a best practice to set the client ID for each consumer group to something unique that identifies that group. This allows a single consumer group to share a quota, and it makes it easier to identify in logs what group is responsible for requests.

The format of the command to change a client configuration is:

```
kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster --alter --entity-type clients --entity-name <client I
```

The configurations (keys) for clients are:

| ============================ | Configuration Key | Description | producer_bytes_rate | The amount of messages, in bytes, that a singe client ID is allowed to produce to a single broker in one second | consumer_bytes_rate | The amount of messages, in bytes, that a single client ID is allowed to consume from a single broker in one second | ============================

Describing Configurations Overrides

All configuration overrides can be listed using the command line tool. This will allow you to examine the specific configuration for a topic or client. Similar to other tools, this is done using the `--describe` command.

Example: Show all configuration overrides for the topic named “my-topic”

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster --describe --entity-type topics --entity-name my-to
Configs for topics:my-topic are retention.ms=3600000,segment.ms=3600000
#
```

TOPIC OVERRIDES ONLY

The configuration description will only show overrides - it does not include the cluster default configurations. Currently, there is no way to dynamically discover, either via Zookeeper or the Kafka protocol, the configuration of the brokers themselves. This means that when using this tool to discover topic or client settings in automation, the tool must have separate knowledge of the cluster default configuration.

Removing Configuration Overrides

Dynamic configurations can be removed entirely, which will cause the entity to revert back to the cluster defaults. To delete a configuration override, use the `--alter` command along with the `--delete-config` parameter.

Example: Delete a configuration override for `retention.ms` for a topic named “my-topic”

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster --alter --entity-type topics --entity-name my-topic
Updated config for topic: "my-topic".
#
```

Partition Management

The Kafka tools contain two scripts for working with the management of partitions - one allows for the reelection of leader replicas, and the other is a low-level utility for assigning partitions to brokers. Together, these tools can assist with the proper balancing of message traffic within a cluster of Kafka brokers.

Preferred Replica Election

As described in Chapter 6, partitions can have multiple replicas for reliability. However, only one of these replicas can be the leader for the partition, and all produce and consume operations happen on that broker. This is defined by Kafka internals as the first in sync replica in the replica list, but when a broker is stopped and restarted, it does not resume leadership of any partitions automatically.

AUTOMATIC LEADER REBALANCING

There is a broker configuration for automatic leader rebalancing, but it is not recommended for production use at the time of publication. There are significant performance impacts caused by the automatic balancing module, and it can cause a lengthy pause in client traffic for larger clusters.

One way to cause brokers to resume leadership is to trigger a preferred replica election. This tells the cluster controller to select the ideal leader for partitions. The operation is generally non-impacting, as the clients can track leadership changes automatically. This can be manually ordered using the *kafka-preferred-replica-election.sh* utility.

Example: Start a preferred replica election for all topics in a cluster with 1 topic that has 8 partitions

```
# kafka-preferred-replica-election.sh --zookeeper zoo1.example.com:2181/kafka-cluster
Successfully started preferred replica election for partitions Set([my-topic,5], [my-topic,0], [my-topic,7], [my-topic
#
```

For clusters with a large number of partitions, it is possible that a single preferred replica election will not be able to run. The request must be written to a Zookeeper znode within the cluster metadata, and if the request is larger than the size for a znode (by default, 1 MB), it will fail. In this case, you will need to create a file that contains a JSON object listing the partitions to elect for and break the request into multiple steps. The format for the JSON file is:

```
{
    "partitions": [
        {
            "partition": 1,
            "topic": "foo"
        },
        {
            "partition": 2,
            "topic": "foobar"
        }
    ]
}
```

Example: Start a preferred replica election with a specified list of partitions in a file named “partitions.json”

```
# kafka-preferred-replica-election.sh --zookeeper zoo1.example.com:2181/kafka-cluster --path-to-json-file partitions.j
Successfully started preferred replica election for partitions Set([my-topic,1], [my-topic,2], [my-topic,3])
#
```

Changing a Partition’s Replicas

From time to time it may be necessary to change the replica assignments for a partition. Some examples of when this might be needed are * if a topic's partitions are not balanced across the cluster, causing uneven load on brokers * if a broker is taken offline and the partition is under replicated * if a new broker is added and needs to receive a share of the cluster load.

The *kafka-reassign-partitions.sh* can be used to perform this operation. This tool must be used in at least two steps. The first step uses a broker list and a topic list to generate a set of moves. The second step executes the moves that were generated. There is an optional third step that uses the generated list to verify the progress or completion of the partition reassignments.

To generate a set of partition moves, you must create a file that contains a JSON object listing the topics. The JSON object is formatted in this way (the version number is currently always 1):

```
{
    "topics": [
        {
            "topic": "foo"
        },
        {
            "topic": "foo1"
        }
    ],
    "version": 1
}
```

Example: Generate a set of partition moves to move the topics listed in the file "topics.json" to the brokers with IDs 0 and 1

```
# kafka-reassign-partitions.sh --zookeeper zoo1.example.com:2181/kafka-cluster --generate --topics-to-move-json-file t
Current partition replica assignment

{"version":1,"partitions":[{"topic":"my-topic","partition":5,"replicas":[0,1]},{"topic":"my-topic","partition":10,"rep
Proposed partition reassignment configuration

{"version":1,"partitions":[{"topic":"my-topic","partition":5,"replicas":[0,1]},{"topic":"my-topic","partition":10,"rep
#
```

The broker list is provided on the tool command line as a comma-separated list of broker IDs. The tool will output, on standard out, two JSON objects describing the current partition assignment for the topics and the proposed partition assignment. The format of these JSON objects is: {"partitions": [{"topic": "my-topic", "partition": 0, "replicas": [1,2] }], "version":1}

The first JSON object can be saved in case the reassignment needs to be reverted. The second JSON object, the one that shows the proposed assignment, should be saved to a new file. This file is then provided back to the *kafka-reassign-partitions.sh* tool for the second step.

Example: Execute a proposed partition reassignment from the file "reassign.json"

```
# kafka-reassign-partitions.sh --zookeeper zoo1.example.com:2181/kafka-cluster --execute --reassignment-json-file reas
Current partition replica assignment

{"version":1,"partitions":[{"topic":"my-topic","partition":5,"replicas":[0,1]},{"topic":"my-topic","partition":10,"rep

Save this to use as the --reassignment-json-file option during rollback
Successfully started reassignment of partitions {"version":1,"partitions":[{"topic":"my-topic","partition":5,"replicas
#
```

This will start the reassignment of the specified partition replicas to the new brokers. The cluster controller performs this action by adding the new replicas to the replica list for each partition (increasing the replication factor). The new replicas will then copy all existing messages for each partition from the current leader. Depending on the size of the partitions on disk, this can take a significant amount of time as the data is copied across the network to the new replicas. Once replication is complete, the controller removes the old replicas from the replica list (reducing the replication factor to the original size).

IMPROVING NETWORK UTILIZATION WHEN REASSIGNING REPLICAS

When removing many partitions from a single broker, such as if that broker is being removed from the cluster, it is a best practice to shut down and restart the broker before starting the reassignment. This will cause that broker to not be the leader for any partitions, distributing them into the rest of the cluster (as long as automatic leader elections are not enabled). This can significantly increase the performance of reassignments and reduce the impact on the cluster as the replication traffic will be distributed to many brokers.

While the reassignment is running, and after it is complete, the *kafka-reassign-partitions.sh* tool can be used to verify the status of the reassignment. This will show what reassignments are currently in progress, what reassignments have completed, and, if there was an error, what reassignments have failed. In order to do this, you must have the file with the JSON object that was used in the execute step.

Example: Verify a running partition reassignment from the file "reassign.json"

```
# kafka-reassign-partitions.sh --zookeeper zoo1.example.com:2181/kafka-cluster --verify --reassignment-json-file reass
Status of partition reassignment:
Reassignment of partition [my-topic,5] completed successfully
Reassignment of partition [my-topic,0] completed successfully
Reassignment of partition [my-topic,7] completed successfully
Reassignment of partition [my-topic,13] completed successfully
Reassignment of partition [my-topic,4] completed successfully
Reassignment of partition [my-topic,12] completed successfully
Reassignment of partition [my-topic,6] completed successfully
Reassignment of partition [my-topic,11] completed successfully
Reassignment of partition [my-topic,10] completed successfully
Reassignment of partition [my-topic,9] completed successfully
Reassignment of partition [my-topic,2] completed successfully
Reassignment of partition [my-topic,14] completed successfully
Reassignment of partition [my-topic,3] completed successfully
Reassignment of partition [my-topic,1] completed successfully
Reassignment of partition [my-topic,15] completed successfully
Reassignment of partition [my-topic,8] completed successfully
#
```

BATCHING REASSIGNMENTS

Partition reassignments have a big impact on the performance of your cluster, as they will cause changes to the consistency of the memory page cache and use network and disk I/O. Breaking reassignments into many small steps is a good idea to keep this to a minimum.

Changing Replication Factor

There is an undocumented feature of the partition reassignment tool which will allow you to increase or decrease the replication factor for a partition. This may be necessary in situations where a partition was created with the wrong replication factor (such as if there were not enough brokers available when the topic was created). This can be done by created a JSON object with the format used in the execute step of partition reassignment that has an explicit list of replicas with the number of replicas needed for the partition. The cluster will complete the reassignment and keep the replication factor at the new size.

For example, consider the current assignment for a topic named "my-topic" with one partition has a replication factor of 1:

```
{
    "partitions": [
        {
            "topic": "my-topic",
            "partition": 0,
            "replicas": [
                1
            ]
        }
    ],
    "version": 1
}
```

Providing the following JSON object in the execute step of partition reassignment will result in the replication factor being increased to 2:

```
    {
        "partitions": [
            {
                "partition": 0,
                "replicas": [
                    1,
                    2
                ],
                "topic": "my-topic"
            }
        ],
        "version": 1
    }
```

Similarly, the replication factor for a partition can be reduced by providing a JSON object with a smaller list of replicas.

Dumping Log Segments

If you have to go looking for the specific content of a message, perhaps because you ended up with a “poison pill” message in your topic that the consumer cannot handle, there is a helper tool to decode the log segments for a partition. This will allow you to view individual messages without needing to consume and decode them. The tool takes a comma-separated list of log segment files as an argument and can print out either message summary information or detailed message data.

Example: Decode the log segment file named 00000000000052368601.log, showing message summaries

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files 00000000000052368601.log
Dumping 00000000000052368601.log
Starting offset: 52368601
offset: 52368601 position: 0 NoTimestampType: -1 isvalid: true payloadsize: 661 magic: 0 compresscodec: GZIPCompressio
offset: 52368603 position: 687 NoTimestampType: -1 isvalid: true payloadsize: 895 magic: 0 compresscodec: GZIPCompress
offset: 52368604 position: 1608 NoTimestampType: -1 isvalid: true payloadsize: 665 magic: 0 compresscodec: GZIPCompres
offset: 52368606 position: 2299 NoTimestampType: -1 isvalid: true payloadsize: 932 magic: 0 compresscodec: GZIPCompres
...
```

Example: Decode the log segment file named 00000000000052368601.log, showing message data

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files 00000000000052368601.log --print-data-log
offset: 52368601 position: 0 NoTimestampType: -1 isvalid: true payloadsize: 661 magic: 0 compresscodec: GZIPCompressio
offset: 52368603 position: 687 NoTimestampType: -1 isvalid: true payloadsize: 895 magic: 0 compresscodec: GZIPCompress
offset: 52368604 position: 1608 NoTimestampType: -1 isvalid: true payloadsize: 665 magic: 0 compresscodec: GZIPCompres
offset: 52368606 position: 2299 NoTimestampType: -1 isvalid: true payloadsize: 932 magic: 0 compresscodec: GZIPCompres
...
```

It is also possible to use this tool to validate the index file that goes along with a log segment. The index is used for finding messages within a log segment, and if corrupted will cause errors in consumption. Validation is performed whenever a broker starts up in an unclean state (i.e. it was not stopped normally), but it can be performed manually as well. There are two options for checking indices, depending on how much checking you want to do. The option `--index-sanity-check` will just check that the index is in a useable state, while `--verify-index-only` will check for mismatches in the index without printing out all the index entries.

Example: Validate that the index file for the log segment file named 00000000000052368601.log is sane

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files 00000000000052368601.index,00000000000052368601.log --index-s
Dumping 00000000000052368601.index
00000000000052368601.index passed sanity check.
Dumping 00000000000052368601.log
Starting offset: 52368601
offset: 52368601 position: 0 NoTimestampType: -1 isvalid: true payloadsize: 661 magic: 0 compresscodec: GZIPCompressio
offset: 52368603 position: 687 NoTimestampType: -1 isvalid: true payloadsize: 895 magic: 0 compresscodec: GZIPCompress
offset: 52368604 position: 1608 NoTimestampType: -1 isvalid: true payloadsize: 665 magic: 0 compresscodec: GZIPCompres
...
```

Replica Verification

Partition replication works similar to a regular Kafka consumer client: the follower broker starts replicating at the oldest offset and checkpoints the current offset to disk periodically. When replication stops and restarts, it picks up from the last checkpoint. It is possible for previously replicated log segments to get deleted from a broker and the follower will not fill in the gaps in this case.

To validate that the replicas for a topic's partitions are the same across the cluster, you can use the *kafka-replica-verification.sh* tool for verification. This tool will fetch messages from all the replicas for a given set of topic partitions and check that all messages exist on all replicas. You must provide the tool with a regular expression that matches the topics you wish to validate. If none is provided, all topics are validated. You must also provide an explicit list of brokers to connect to.

CAUTION: CLUSTER IMPACT AHEAD

The replica verification tool will have an impact on your cluster similar to reassigning partitions, as it must read all messages from the oldest offset in order to verify the replica. In addition, it reads from all replicas for a partition in parallel, so it should be used with caution.

Example: Verify the replicas for the topics starting with "my-" on brokers 1 and 2

```
# kafka-replica-verification.sh --broker-list kafka1.example.com:9092,kafka2.example.com:9092 --topic-white-list 'my-.
2016-11-23 18:42:08,838: verification process is started.
2016-11-23 18:42:38,789: max lag is 0 for partition [my-topic,7] at offset 53827844 among 10 partitions
2016-11-23 18:43:08,790: max lag is 0 for partition [my-topic,7] at offset 53827878 among 10 partitions
```

Consuming and Producing

While working with Apache Kafka, you will often find it is necessary to manually consume messages, or produce some sample messages, in order to validate what's going on with your applications. There are two utilities provided to help with this, *kafka-console-consumer.sh* and *kafka-console-producer.sh*. These are wrappers around the Java client libraries that allow you to interact with Kafka topics without having to write an entire application to do it.

PIPING OUTPUT TO ANOTHER APPLICATION

While it is possible to write applications that wrap around the console consumer or producer, for example to consume messages and pipe them to another application for processing, this type of application is quite fragile and should be avoided. It is difficult to interact with the console consumer in a way that does not lose messages. Likewise, the console producer does not allow for using all features and properly sending bytes is tricky. It is best to use either the Java client libraries directly, or a 3rd party client library for other languages that uses the Kafka protocol directly.

Console Consumer

The *kafka-console-consumer.sh* tool provides a means to consume messages out of one or more topics in your Kafka cluster. The messages are printed on standard out, delimited by a newline. By default, it outputs the raw bytes in the message with no formatting (using the `DefaultFormatter`). There are numerous options available, but certain basics that must be provided.

WARNING

It is very important to use a consumer that is the same version as your Kafka cluster. Older console consumers can potentially damage the cluster by interacting with Zookeeper in incorrect ways.

The first step is to specify whether or not to use the new consumer, and the configuration to point to the Kafka cluster itself. When using the older consumer, the only argument required for this is the `--zookeeper` option followed by the connect string for the cluster. From the above examples, this might be `--zookeeper zoo1.example.com:2181/kafka-cluster`. If you are using the new consumer, you must specify both the `--new-consumer` flag as well as the `--broker-list` option followed by a comma separated broker list, such as `--broker-list kafka1.example.com:9092,kafka2.example.com:9092`

Next, you must specify the topics to consume. There are three options provided for this, `--topic`, `--whitelist`, and `--blacklist`. One, and only one, may be provided. The `--topic` option specifies a single topic to consume. The `--whitelist` and `--blacklist` options each are followed by a regular expression (remember to escape it properly for the shell command line). The whitelist will consume all topics that match the regular expression, while the blacklist will consume all topics *except* those matched by the regular expression.

Example: Consume a single topic named “my-topic” using the old consumer

```
# kafka-console-consumer.sh --zookeeper zoo1.example.com:2181/kafka-cluster --topic my-topic
sample message 1
sample message 2
^CProcessed a total of 2 messages
#
```

In addition to the basic command line options, it is possible to pass any normal consumer configuration options to the console consumer as well. This can be done in two ways, depending on how many options you need to pass and how you prefer to do it. The first is to provide a consumer configuration file by specifying `--consumer.config CONFIGFILE`, where *CONFIGFILE* is the full path to a file that contains the configuration options. The other way is to specify the options on the command line with one or more arguments of the form `--consumer-property KEY=VALUE`, where *KEY* is the configuration option name, and *VALUE* is the value to set it to. This can be useful for consumer options like setting the consumer group ID.

CONFUSING COMMAND LINE OPTIONS

There is a `--property` command line option for both the console consumer and the console producer, but this should not be confused with the `--consumer-property` and `--producer-property` options. The `--property` option is only used for passing configurations to the message formatter, and not the client itself.

There are a few other commonly used options for the console consumer that you should know: * `--formatter CLASSNAME` - Specifies a message formatter class to be used to decode the messages. This defaults to *kafka.tools.DefaultFormatter* * `--from-beginning` - Consume messages in the topic(s) specified from the oldest offset. Otherwise, consumption starts from the latest offset * `--max-messages NUM` - Consume at most *NUM* messages before exiting * `--partition NUM` - Consume only from the partition with ID *NUM* (requires the new consumer)

MESSAGE FORMATTER OPTIONS

There are 3 message formatters available to use besides the default: * *kafka.tools.LoggingMessageFormatter* - Outputs messages using the logger, rather than standard out. Messages are printed at the INFO level, and include the timestamp, key, and value * *kafka.tools.ChecksumMessageFormatter* - Prints only message checksums * *kafka.tools.NoOpMessageFormatter* - Consumes messages but does not output them at all

The *kafka.tools.DefaultMessageFormatter* also has several useful options that can be passed using the `--property` command line option: * *print.timestamp* - Set to "true" to display the timestamp of each message (if available) * *print.key* - Set to "true" to display the message key in addition to the value * *key.separator* - Specify the delimiter character to use between the message key and message value when printing * *line.separator* - Specify the delimiter character to use between messages * *key.deserializer* - Provide a class name that is used to deserialize the message key before printing * *value.deserializer* - Provide a class name that is used to deserialize the message value before printing

The deserializer classes must implement *org.apache.kafka.common.serialization.Deserializer* and the console consumer will call the *toString* method on them to get the output to display. Typically, you would implement these deserializers as a Java class that you would insert into the classpath for the console consumer by setting the *CLASSPATH* environment variable before executing *kafka_console_consumer.sh*.

CONSUMING THE OFFSETS TOPIC

It is sometimes useful to see what offsets are being committed for the cluster's consumer groups. You may want to see if a particular group is committing offsets at all, or how often offsets are being committed. This can be done by using the console consumer to consume the special internal topic named "__consumer_offsets". All consumer offsets are written as messages to this topic. In order to decode the messages in this topic, you must use the formatter class _kafka.coordinator.GroupMetadataManager$OffsetsMessageFormatter.

Example: Consume a single message from the offsets topic

```
# kafka-console-consumer.sh --zookeeper zoo1.example.com:2181/kafka-cluster --topic __consumer_offsets --formatter 'ka
[my-group-name,my-topic,0]::[OffsetMetadata[481690879,NO_METADATA],CommitTime 1479708539051,ExpirationTime 14803133390
Processed a total of 1 messages
#
```

Console Producer

Similar to the console consumer, the *kakfa-console-producer.sh* tool can be used to write messages into a Kafka topic in your cluster. By default, messages are read one per line, with a tab character separating the key and the value (if no tab character is present, the key is null).

CHANGING LINE READING BEHAVIOR

You can provide your own class for reading lines in order to do custom things if you really would like to. The class that you create must extend *kafka.common.MessageReader* and will be responsible for creating the *ProducerRecord*. Specify your class on the command line with the `--line-reader` option, and make sure the JAR containing your class is in the classpath.

The console producer requires that two arguments are provided at a minimum. The parameter `--broker-list` specifies one or more brokers, as a comma-separated list of `hostname:port` entries for your cluster. The other required parameter is the `--topic` option to specify the topic that you are producing messages to. When you are done producing, send an EOF character to close the client.

Example: Produce two messages to a topic named "my-topic"

```
# kafka-console-producer.sh --broker-list kafka1.example.com:9092,kafka2.example.com:9092 --topic my-topic
sample message 1
sample message 2
^D
#
```

Just like the console consumer, it is possible to pass any normal producer configuration options to the console producer as well. This can be done in two ways, depending on how many options you need to pass and how you prefer to do it. The first is to provide a producer configuration file by specifying `--producer.config CONFIGFILE`, where *CONFIGFILE* is the full path to a file that contains the configuration options. The other way is to specify the options on the command line with one or more arguments of the form `--producer-property KEY=VALUE`, where *KEY* is the configuration option name, and *VALUE* is the value to set it to. This can be useful for producer options like message batching configurations (such as `linger.ms` or `batch.size`).

The console producer has many command line arguments available for adjusting its behavior. Some of the more useful options are: * `--key-serializer CLASSNAME` - Specifies a message encoder class to be used to serialize the message key. This defaults to *kafka.serializer.DefaultEncoder* * `--value-serializer CLASSNAME` - Specifies a message encoder class to be used to serialize the message value. This defaults to *kafka.serializer.DefaultEncoder* * `--compression-codec STRING` - Specify the type of compression to be used when producing messages. This can be one of: none, gzip, snappy, or lz4. The default value is gzip * `--sync` - Produce messages synchronously, waiting for each message to be acknowledged before sending the next one

CREATING A CUSTOM SERIALIZER

Custom serializers must extend *kafka.serializer.Encoder*. This can be used for performing operations such as taking a string from standard input that is JSON formatter and coverting it to an encoded format, such as Avro, that is suitable for the data encoding in your topic.

LINE READER OPTIONS

The *kafka.tools.LineMessageReader* class, which is responsible for reading standard input and creating producer records, also has several useful options that can be passed to the console producer using the `--property` command line option: * *ignore.error* - Set to “false” to throw an exception when `parse.key` is set to “true” and a key separator is not present. Defaults to “true”. * *parse.key* - Set to “false” to always set the key to null. Defaults to “true”. * *key.separator* - Specify the delimiter character to use between the message key and message value when reading. Defaults to a tab character.

When producing messages, the *LineMessageReader* will split the input on the first instance of the `key.separator`. If there are no characters remaining after that, the value of the message will be empty. If no key separator character is present on the line, or if `parse.key` is false, the key will be null.

Client ACLs:

There is a command line tool, *kafka-acls.sh*, provided for interacting with access controls for Kafka clients. The usage of this tool is covered in [Link to Come].

Unsafe Operations

There are some administrative tasks that are technically possible to do, but should not be attempted except in the most extreme situations. Often this is when you are diagnosing a problem and have run out of options, or you have found a specific bug that you need to work around temporarily. These tasks are usually undocumented, unsupported, and pose some amount of risk to your application.

Several of the more common of these tasks are documented here so that in an emergency situation, there is a potential option for recovery. Their use is not recommended under normal cluster operations, and should be considered carefully before being executed.

DANGER: HERE BE DRAGONS

The operations in this section involve working with the cluster metadata stored in Zookeeper directly. This can be a very dangerous operation, so you must be very careful to not modify the information in Zookeeper directly, except as noted.

Moving the Cluster Controller

Every Kafka cluster has a controller, which is a thread running within one of the brokers. The controller is responsible for overseeing cluster operations, and from time to time it is desirable to forcibly move the controller to a different broker. One such example is when the controller has suffered an exception or other problem that has left it running, but not functional. Moving the controller in these situations does not have a high risk, but it is not a normal task and should not be performed regularly.

The broker that is currently the controller registers itself using a Zookeeper node at the top level of the cluster path that is named `/controller`. Deleting this Zookeeper node manually will cause the current controller to resign and the cluster will select a new controller.

Killing a Partition Move

The normal operational flow for a partition reassignment is: # Reassignment is requested (Zookeeper node is created) # Cluster controller adds partitions to new brokers being added # New brokers begin replicating each partition until it is in-sync # Cluster controller removes the old brokers from the partition replica list

As all the reassignments are started in parallel when requested, there is normally no reason to attempt to cancel an in-progress reassignment. One of the exceptions is when a broker fails in the middle of a reassignment and cannot immediately be restarted. This results in a reassignment that will never finish, which precludes starting any additional reassignments (such as to remove partitions from the failed broker and assign them to other brokers). In a case such as this, it is possible to make the cluster forget about the existing reassignment.

To remove an in-progress partition reassignment: # Remove the `/admin/reassign_partitions` Zookeeper node from the Kafka cluster path # Force a controller move (see the previous section for more detail)

CHECKING REPLICATION FACTORS

When removing an in-progress partition move, any partitions that have not yet completed will not go through the step of having the old brokers removed from the replica list. This means that the replication factor for some partitions may be greater than intended. The broker will not allow some admin operations for topics that have partitions with inconsistent replication factors (such as increasing partitions). It is advisable to review the partitions that were still in progress and make sure their replication factor is correct with another partition reassignment.

Removing Topics to be Deleted

When using the command line tools to delete a topic, this is done by creating a Zookeeper node that requests the deletion. Under normal circumstances, this is executed by the cluster immediately. However, the command line tool has no way to know whether or not deletion of topics is enabled in the cluster. As a result, it will request deletion of topics regardless, and this can leave a surprise waiting if deletion is disabled. It is possible to delete the requests pending for deletion of topics to avoid this.

Topics are requested for deletion by creating a Zookeeper node as a child under `/admin/delete_topic` that is named with the topic name. Deleting these Zookeeper nodes (but not the parent `/admin/delete_topic` node) will remove the pending requests.

Deleting Topics Manually

If you are running a cluster with delete topic disabled, or if you find yourself in need of deleting some topics outside of the normal flow of operations, it is possible to manually delete them from the cluster. This requires a full shutdown of all brokers in the cluster, however, and cannot be done online.

WARNING

Modifying the cluster metadata in Zookeeper when the cluster is online is a very dangerous operation, and can put the cluster into an unstable state. Never attempt to delete or modify topic metadata in Zookeeper while the cluster is online.

To delete a topic from the cluster: # Shut down all brokers in the cluster # Remove the Zookeeper path `/brokers/topics/TOPICNAME` from the Kafka cluster path. Note that this node has child nodes that must be deleted first # Remove the partition directories from the log directories on each broker. These will be named `TOPICNAME-NUM`, where `NUM` is the partition ID # Restart all brokers

Monitoring Brokers and Clients

This chapter covered using the administrative tools to manage your Kafka cluster, but managing the cluster is impossible without proper monitoring in place. The next chapter, Chapter 10, will discuss ways to monitor the broker and cluster health and operations so you will have assurances that Kafka is working well (and that you know about it when it isn't). We will also offer best practices for monitoring your clients, both producers and consumers.

PREV
8. Cross-Cluster Data Mirroring

NEXT
10. Monitoring Kafka