Kafka-Utils Documentation

Release 1.2.0

Yelp Inc.

Jul 18, 2017

Contents

CHAPTER 1

Description

Kafka-Utils is a library containing tools to interact with kafka clusters and manage them. The tool provides utilities like listing of all the clusters, balancing the partition distribution across brokers and replication-groups, managing consumer groups, rolling-restart of the cluster, cluster healthchecks.

For more information about Apache Kafka see the official Kafka documentation.

CHAPTER 2

How to install

```
$ pip install kafka-utils
```

List available clusters.

```
$ kafka-utils
Cluster type sample_type:
    Cluster name: cluster-1
    broker list: cluster-elb-1:9092
    zookeeper: 11.11.11.111:2181,11.11.11.112:2181,11.11.11.113:2181/kafka-1
```

Configuration

Kafka-Utils reads the cluster configuration needed to access Kafka clusters from yaml files. Each cluster is identified by *type* and *name*. Multiple clusters of the same type should be listed in the same *type.yaml* file. The yaml files are read from `$KAFKA_DISCOVERY_DIR`, `$HOME/.kafka_discovery` and `/etc/kafka_discovery`, the former overrides the latter.

Sample configuration for `sample_type` cluster at `/etc/kafka_discovery/sample_type.yaml`

```
---
  clusters:
    cluster-1:
      broker_list:
        - "cluster-elb-1:9092"
      zookeeper: "11.11.11.111:2181,11.11.11.112:2181,11.11.11.113:2181/kafka-1"
    cluster-2:
      broker_list:
        - "cluster-elb-2:9092"
      zookeeper: "11.11.11.211:2181,11.11.11.212:2181,11.11.11.213:2181/kafka-2"
  local_config:
    cluster: cluster-1
```

For example the kafka-cluster-manager command:

```
$ kafka-cluster-manager --cluster-type sample_type stats
```

will pick up default cluster *cluster-1* from the local_config at /etc/kafka_discovery/sample_type.yaml to display statistics of default kafka-configuration.

Cluster Manager

This tool provides a set of commands to manipulate and modify the cluster topology and get metrics for different states of the cluster. These include balancing the cluster-state, decommissioning brokers, evaluating metrics for the current state of the cluster. Each of these commands is as described below.

Replication group parser

The tool supports the grouping of brokers in replication groups. `kafka-cluster-manager` will try to distribute replicas of the same partition across different replication group. The user can use this feature to map replication groups to failure zones, so that a balanced cluster will be more resilient to zone failures.

By default all brokers are considered as part of a single replication group. Custom replication group parsers can be defined by extending the class `ReplicationGroupParser` as shown in the example below:

```
from kafka_utils.kafka_cluster_manager.cluster_info.replication_group_parser \
        import ReplicationGroupParser


class SampleGroupParser(ReplicationGroupParser):

        def get_replication_group(self, broker):
                """Extract the replication group from a Broker instance.
                Suppose each broker hostname is in the form broker-rack<n>, this
                function will return "rack<n>" as replication group
                """
                if broker.inactive:
                        # Can't extract replication group from inactive brokers␣
↪because they
                        # don't have metadata
                        return None
                hostname = broker.metadata['host']
                return hostname.rsplit('-', 1)[1]
```

Create a file named `sample_parser.py` into a directory containing the `__init__.py`.

Example:

```
$HOME/parser
  |-- __init__.py
  |-- sample_parser.py
```

To use the custom parser:

```
$ kafka-cluster-manager --cluster-type sample_type --group-parser $HOME/parser:sample_
↪parser rebalance --replication-groups
```

Cluster Balancers

Every command attempts to find a partition assignment that improves or maintains the balance of the cluster. This tool provides two different cluster balancers that implement different cluster balancing strategies. The *Partition Count Balancer* is the default cluster balancer and is recommended for most users. The *Genetic Balancer* is recommended for users that are able to provide partition measurements. See *partition measurement* for more information.

Partition Count Balancer

This balancing strategy attempts to balance the number of partitions and leaders across replication groups and brokers. Balancing is done in four stages.

1. **Replica distribution**: Uniform distribution of partition replicas across replication groups.
2. **Partition distribution**: Uniform distribution of partitions across groups and brokers.
3. **Leader distribution**: Uniform distribution of preferred partition leaders across brokers.
4. **Topic-partition distribution**: Uniform distribution of partitions of the same topic across brokers.

Genetic Balancer

This balancing strategy considers not only the number of partitions on each broker, but the weight of each partition (see *partition measurement*). It uses a genetic algorithm to approximate the optimal assignment while also limiting the total size of the partitions moved.

The uniform distribution of replicas across replication groups is guaranteed by an initial stage that greedily reassigns replicas across replication groups.

The fitness function used by the genetic algorithm to score partition assignments considers the following:

1. **Broker weight balance**: The sum of the weights of the partitions on each broker should be as balanced as possible.
2. **Leader weight balance**: The sum of the weights of the preferred leader partitions on each broker should be as balanced as possible.
3. **Weighted topic-partition balance**: The distribution of partitions of the same topic across brokers weighted by the total weight of each topic.

The Genetic Balancer can be enabled by using the `--genetic-balancer` toggle.

Partition Measurement

Throughput can vary significantly across the topics of a cluster. To prevent placing too many high-throughput partitions on the same brokers, the cluster manager needs additional information about each partition. For the purposes of this tool, there are two values that need to be defined for each partition: weight and size.

The weight of a partition is a measure of how much load that partition places on the broker that it is assigned to. The weight can have any unit and should represent the relative weight of one partition compared to another. For example a partition with weight 2 is assumed to cause twice as much load on a broker as a partition with weight 1. In practice, a possible metric could be the average byte in/out rate over a certain time period.

The size of a partition is a measure of how expensive it is to move the partition. This value is also relative and can have any unit. The length of the partition's log in bytes is one possible metric.

Since Kafka doesn't keep detailed partition usage information, the task of collecting this information is left to the user. By default every partition is given an equal weight and size of 1. Custom partition measurement approaches can be implemented by extending the `PartitionMeasurer` class. Here is a sample measurer that pulls partition metrics from an external service.

```
import argparse
from requests import get

from kafka.kafka_utils.kafka_cluster_manager.cluster_info.partition_measurer \
        import PartitionMeasurer

class SampleMeasurer(PartitionMeasurer):

    def __init__(self, cluster_config, brokers, assignment, args):
        super(SampleMeasurer, self).__init__(
            cluster_config,
            brokers,
            assignment,
            args
        )
        self.metrics = {}
        for partition_name in assignment.keys():
            self.metrics[partition_name] = get(self.args.metric_url +
                "/{cluster_type}/{cluster_name}/{topic}/{partition}"
                .format(
                    cluster_type=cluster_config.type,
                    cluster_name=cluster_config.name,
                    topic=partition_name[0],
                    partition=partition_name[1],
                )
            ).json()

    def parse_args(self, measurer_args):
        parser = argparse.ArgumentParser(prog='SampleMeasurer')
        parser.add_argument(
            '--metric-url',
            type=string,
            required=True,
            help='URL of the metric service.',
        )
        return parser.parse_args(measurer_args, self.args)

    def get_weight(self, partition_name):
        return self.metrics[partition_name]['bytes_in_per_sec'] + \
            self.metrics[partition_name]['bytes_out_per_sec']

    def get_size(self, partition_name):
        return self.metrics[partition_name]['size']
```

Place this file in a file called `sample_measurer.py` and place it in a python module.

Example:

```
$HOME/measurer
  |-- __init__.py
  |-- sample_measurer.py
```

To use the partition measurer:

```
$ kafka-cluster-manager \
--cluster-type sample_type \
--partition-measurer $HOME/measurer:sample_measurer \
--measurer-args "--metric-url $METRIC_URL" \
stats
```

Cluster rebalance

This command provides the functionality to re-distribute partitions across the cluster to bring it into a more balanced state. The behavior of this command is determined by the choice of *cluster balancer*.

The command provides three toggles to control how the cluster is rebalanced:

- `--replication-groups`: Rebalance partition replicas across replication groups.
- `--brokers`: Rebalance partitions across brokers.
- `--leaders`: Rebalance partition preferred leaders across brokers.

The command also provides toggles to control how many partitions are moved at once:

- `--max-partition-movements`: The maximum number of partition replicas that will be moved. Default: 1.
- `--max-leader-changes`: The maximum number of partition preferred leader changes. Default: 5.
- `--max-movement-size`: The maximum total size of the partition replicas that will be moved. Default: No limit.
- `--auto-max-movement-size`: Set `--max-movement-size` to the size of the largest partition in the cluster.
- `--score-improvement-threshold`: When the *Genetic Balancer* is being used, this option checks the *Genetic Balancer*'s scoring function and only applies the new assignment if the improvement in score is greater than this threshold.

```
$ kafka-cluster-manager --group-parser $HOME/parser:sample_parser --apply
--cluster-type sample_type rebalance --replication-groups --brokers --leaders
--max-partition-movements 10 --max-leader-changes 25
```

Or using the *Genetic Balancer*:

```
$ kafka-cluster-manager --group-parser $HOME/parser:sample_parser --apply
--cluster-type sample_type --genetic-balancer --partition-measurer
$HOME/measurer:sample_measurer rebalance --replication-groups --brokers
--leaders --max-partition-movements 10 --max-leader-changes 25
--auto-max-partition-size --score-improvement-threshold 0.01
```

Brokers decommissioning

This command provides functionalities to decommission a given list of brokers. The key idea is to move all partitions from brokers that are going to be decommissioned to other brokers in either their replication group (preferred) or others replication groups while keeping the cluster balanced as above.

Note: While decommissioning brokers we need to ensure that we have at least 'n' number of active brokers where n is the max replication-factor of a partition.

```
$ kafka-cluster-manager --cluster-type sample_type decommission 123456 123457 123458
```

Or using the *Genetic Balancer*:

```
$ kafka-cluster-manager --cluster-type sample_type --genetic-balancer
--partition-measurer $HOME/measurer:sample_measurer decommission
123456 123457 123458
```

Revoke Leadership

This command provides functionalities to revoke leadership for a particular given set of brokers. The key idea is to move leadership for all partitions on given brokers to other brokers while keeping the cluster balanced.

```
$ kafka-cluster-manager --cluster-type sample_type revoke-leadership 123456 123457␣
↪123458
```

Set Replication Factor

This command provides the ability to increase or decrease the replication-factor of a topic. Replicas are added or removed in such a way that the balance of the cluster is maintained. Additionally, when the replication-factor is decreased, any out-of-sync replicas will be removed first.

```
$ kafka-cluster-manager --cluster-type sample_type set_replication_factor --topic␣
↪sample_topic 3
```

Or using the *Genetic Balancer*:

```
$ kafka-cluster-manager --cluster-type sample_type --genetic-balancer
--partition-measurer $HOME/measurer:sample_measurer set_replication_factor
--topic sample_topic 3
```

Stats

This command provides statistics for the current imbalance state of the cluster. It also provides imbalance statistics of the cluster if a given partition-assignment plan were to be applied to the cluster. The details include the imbalance value of each of the above layers for the overall cluster, each broker and across each replication-group.

```
$ kafka-cluster-manager --group-parser $HOME/parser:sample_parser --cluster-type
sample_type stats
```

Store assignments

Dump the current cluster-topology in json format.

```
$ kafka-cluster-manager --group-parser $HOME/parser:sample_parser --cluster-type
sample_type store_assignments
```

Consumer Manager

This kafka tool provides the ability to view and manipulate consumer offsets for a specific consumer group. For a given cluster, this tool provides us with the following functionalities:

- **Manipulating consumer-groups**: Listing consumer-groups subscribed to the cluster. Copying, deleting and renaming of the group.
- **Manipulating offsets**: For a given consumer-group, fetching current offsets, low and high watermarks for topics and partitions subscribed to the group. Setting, advancing, rewinding, saving and restoring of current-offsets.
- **Manipulating topics**: For a given consumer-group and cluster, listing and unsubscribing topics.
- **Offset storage choice**: Supports Kafka 0.8.2 and 0.9.0, using offsets stored in either Zookeeper or Kafka. Version 0 and 2 of the Kafka Protocol are supported for committing offsets.

Subcommands

- copy_group
- delete_group
- list_groups
- list_topics
- offset_advance
- offset_get
- offset_restore
- offset_rewind
- offset_save
- offset_set
- rename_group
- unsubscribe_topics

Listing consumer groups

The `list_groups` command shows all of the consumer groups that exist in the cluster.

```
$ kafka-consumer-manager --cluster-type=test list_groups
  Consumer Groups:
      group1
      group2
      group3
```

If `list_groups` is called with the `--storage` option, then the groups will only be fetched from Zookeeper or Kafka.

Listing topics

For information about the topics subscribed by a consumer group, the *list_topics* subcommand can be used.

```
$ kafka-consumer-manager --cluster-type=test list_topics group3
  Consumer Group ID: group3
      Topic: topic_foo
             Partitions: [0, 1, 2, 3, 4, 5]
      Topic: topic_bar
             Partitions: [0, 1, 2]
```

Getting consumer offsets

The `offset_get` subcommand gets information about a specific consumer group.

The most basic usage is to call `offset_get` with a consumer group id.

```
$ kafka-consumer-manager --cluster-type test --cluster-name my_cluster offset_get my_
↪group
  Cluster name: my_cluster, consumer group: my_group
  Topic Name: topic1
     Partition ID: 0
             High Watermark: 787656
             Low Watermark: 787089
             Current Offset: 787645
```

The offsets for all topics in the consumer group will be shown by default. A single topic can be specified using the `--topic` option. If a topic is specified, then a list of partitions can also be specified using the `--partitions` option.

By default, the offsets will be fetched from both Zookeeper and Kafka's internal offset storage. A specific offset storage location can be speficied using the `--storage` option.

Manipulating consumer offsets

The offsets for a consumer group can also be saved into a json file.

```
$ kafka-consumer-manager --cluster-type test --cluster-name my_cluster offset_save my_
↪group my_offsets.json
  Cluster name: my_cluster, consumer group: my_group
  Consumer offset data saved in json-file my_offsets.json
```

The save offsets file can then be used to restore the consumer group.

```
$ kafka-consumer-manager --cluster-type test --cluster-name my_cluster offset_restore␣
↪my_offsets.json
  Restored to new offsets {u'topic1': {0: 425447}}
```

The offsets can also be set directly using the `offset_set` command. This command takes a group id, and a set of topics, partitions, and offsets.

```
$ kafka-consumer-manager --cluster-type test --cluster-name my_cluster offset_set my_
↪group topic1.0.38531
```

There is also an `offset_advance` command, which will advance the current offset to the same value as the high watermark of a topic, and an `offset_rewind` command, which will rewind to the low watermark.

If the offset needs to be modified for a consumer group does not already exist, then the `--force` option can be used. This option can be used with `offset_set`, `offset_rewind`, and `offset_advance`.

Copying or renaming consumer group

Consumer groups can have metadata copied into a new group using the `copy_group` subcommand.

```
$ kafka-consumer-manager --cluster-type=test copy_group my_group1 my_group2
```

They can be renamed using `rename_group`.

```
$ kafka-consumer-manager --cluster-type=test rename_group my_group1 my_group2
```

When the group is copied, if a topic is specified using the `--topic` option, then only the offsets for that topic will be copied. If a topic is specified, then a set of partitions of that topic can also be specified using the `--partitions` option.

Deleting or unsubscribing consumer groups

A consumer group can be deleted using the `delete_group` subcommand.

```
$ kafka-consumer-manager --cluster-type=test delete_group my_group
```

A consumer group be unsubscribed from topics using the `unsubscribe_topics` subcommand. If a single topic is specified using the `--topic` option, then the group will be unsubscribed from only that topic.

Rolling Restart

The kafka-rolling-restart script can be used to safely restart an entire cluster, one server at a time. The script finds all the servers in a cluster, checks their health status and executes the restart.

Cluster health

The health of the cluster is defined in terms of broker availability and under replicated partitions. Kafka-rolling-restart will check that all brokers are answering to JMX requests, and that the total numer of under replicated partitions is zero. If both conditions are fulfilled, the cluster is considered healthy and the next broker will be restarted.

The JMX metrics are accessed via Jolokia, which must be running on all brokers.

Note: If a broker is not registered in Zookeeper when the tool is executed, it will not appear in the list of known brokers and it will be ignored.

Parameters

The parameters specific for kafka-rolling-restart are:

- `--check-interval INTERVAL`: the number of seconds between each check. Default 10.
- `--check-count COUNT`: the number of consecutive checks that must result in cluster healthy before restarting the next server. Default 12.

- `--unhealthy-time-limit LIMIT`: the maximum time in seconds that a cluster can be unhealthy for. If the limit is reached, the script will terminate with an error. Default 600.
- `--jolokia-port PORT`: The Jolokia port. Default 8778.
- `--jolokia-prefix PREFIX`: The Jolokia prefix. Default "jolokia/".
- `--no-confirm`: If specified, the script will not ask for confirmation.
- `--skip N`: Skip the first N servers. Useful to recover from a partial rolling restart. Default 0.
- `--verbose`: Turn on verbose output.

Examples

Restart the generic dev cluster, checking the JMX metrics every 30 seconds, and restarting the next broker after 5 consecutive checks have confirmed the health of the cluster:

```
$ kafka-rolling-restart --cluster-type generic --cluster-name dev --check-interval 30␣
↪--check-count 5
```

Check the generic prod cluster. It will report an error if the cluster is unhealthy for more than 900 seconds:

```
$ kafka-rolling-restart --cluster-type generic --cluster-name prod --unhealthy-time-
↪limit 900
```

Kafka Check

The kafka-check command performs multiple checks on the health of the cluster. Each subcommand will run a different check. The tool can run on the broker itself or on any other machine, and it will check the health of the entire cluster.

One possible way to deploy the tool is to install the kafka-utils package on every broker, and schedule kafka-check to run periodically on each machine with cron. Kafka-check provides two simple coordination mechanisms to make sure that the check only runs on a single broker per cluster.

Coordination strategies: * First broker only: the script will only run on the broker with lowest

  broker id.

- Controller only: the script will only run on the controller of the cluster.

Coordination parameters: * `--broker-id`: the id of the broker where the script is running.

  Set it to -1 if automatic broker ids are used.

- `--data-path DATA_PATH`: Path to the Kafka data folder, used in case of automatic broker ids to find the assigned id.
- `--controller-only`: if is specified, the script will only run on the controller. The execution on other brokers won't perform any check and it will always succeed.
- `--first-broker-only`: if specified, the command will only perform the check if broker_id is the lowest broker id in the cluster. If it is not the ' lowest, it will not perform any check and succeed immediately.

Checking in-sync replicas

The `min_isr` subcommand checks if the number of in-sync replicas for a partition is equal or greater than the minimum number of in-sync replicas configured for the topic the partition belongs to. A topic specific `min.insync.replicas` overrides the given default.

The parameters for min_isr check are:

- `--default_min_isr DEFAULT_MIN_ISR`: Default min.isr value for cases without settings in Zookeeper for some topics.

```
$ kafka-check --cluster-type=sample_type min_isr
OK: All replicas in sync.
```

In case of min isr violations:

```
$ kafka-check --cluster-type=sample_type min_isr --default_min_isr 3

 isr=2 is lower than min_isr=3 for sample_topic:0
 CRITICAL: 1 partition(s) have the number of replicas in sync that is lower
 than the specified min ISR.
```

Checking replicas available

The `replica_unavailability` subcommand checks if the number of replicas not available for communication is equal to zero. It will report the aggregated result of unavailable replicas of each broker if any.

The parameters specific to replica_unavailability check are:

```
$ kafka-check --cluster-type=sample_type replica_unavailability
OK: All replicas available for communication.
```

In case of not first broker in the broker list in Zookeeper:

```
$ kafka-check --cluster-type=sample_type --broker-id 3 replica_unavailability --first-
↪broker-only
OK: Provided broker is not the first in broker-list.
```

In case where some partitions replicas not available for communication.

```
$ kafka-check --cluster-type=sample_type replica_unavailability
CRITICAL: 2 replica(s) unavailable for communication.
```

Checking offline partitions

The `offline` subcommand checks if there are any offline partitions in the cluster. If any offline partition is found, it will terminate with an error, indicating the number of offline partitions.

```
$ kafka-check --cluster-type=sample_type offline
CRITICAL: 64 offline partitions.
```

Corruption Check

The kafka-corruption-check script performs a check on the log files stored on the Kafka brokers. This tool finds all the log files modified in the specified time range and runs DumpLogSegments on them. The output is collected and filtered, and all information related to corrupted messages will be reported to the user.

Even though this tool executes the log check with a low ionice priority, it can slow down the cluster given the high number of io operations required. Consider decreasing the batch size to reduce the additional load.

Parameters

The parameters specific for kafka-corruption-check are:

- `--minutes N`: check the log files modified in the last `N` minutes.
- `--start-time START_TIME`: check the log files modified after `START_TIME`. Example format: `--start-time "2015-11-26 11:00:00"`
- `--end-time END_TIME`: check the log files modified before `END_TIME`. Example format: `--end-time "2015-11-26 12:00:00"`
- `--data-path`: the path to the data files on the Kafka broker.
- `--java-home`: the JAVA_HOME on the Kafka broker.
- `--batch-size BATCH_SIZE`: the number of files that will be checked in parallel on each broker. Default: 5.
- `--check-replicas`: if set it will also check the data on replicas. Default: false.
- `--verbose`: enable verbose output.

Examples

Check all the files (leaders only) in the generic dev cluster and which were modified in the last 30 minutes:

```
$ kafka-corruption-check --cluster-type generic --cluster-name dev --data-path /var/
↪kafka-logs --minutes 30
Filtering leaders
Broker: 0, leader of 9 over 13 files
Broker: 1, leader of 4 over 11 files
Starting 2 parallel processes
  Broker: broker0.example.org, 9 files to check
  Broker: broker1.example.org, 4 files to check
Processes running:
  broker0.example.org: file 0 of 9
  broker0.example.org: file 5 of 9
ERROR Host: broker0.example.org: /var/kafka-logs/test_topic-0/00000000000000003363.log
ERROR Output: offset: 3371 position: 247 isvalid: false payloadsize: 22 magic: 0␣
↪compresscodec: NoCompressionCodec crc: 2230473982
  broker1.example.org: file 0 of 4
```

In this example, one corrupted file was found in broker 0.

Check all the files modified after the specified date, in both leaders and replicas:

```
$ kafka-corruption-check [...] --start-time "2015-11-26 11:00:00" --check-replicas
```

Check all the files that were modified in the specified range:

```
$ kafka-corruption-check [...] --start-time "2015-11-26 11:00:00" --end-time "2015-11-
↪26 12:00:00"
```

Indices and tables

- genindex
- modindex
- search