CAFxX / **kafkabalancer**

Automatically rebalance your kafka topics, partitions, replicas across your cluster

53 commits	3 branches	0 releases	1 contributor	MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

Carlo Alberto Ferraris Use pure RMS to compute unbalance — Latest commit 917a869 on Nov 14, 2017

logbuf	gofmt logbug	2 years ago
test	add simple json	3 years ago
.gitignore	Remove the build output	3 years ago
LICENSE	Initial commit	3 years ago
README.md	README	a year ago
balancer.go	move leaders first, if enabled	3 years ago
balancer_test.go	Fix MoveDisallowedReplicas suboptimal broker seelction	a year ago
codecs.go	remove a bit of duplicated code	2 years ago
codecs_test.go	Implement reading from ZK, add tests	2 years ago
kafkabalancer.go	more tests	2 years ago
kafkabalancer_test.go	unit tests: 92.2%	2 years ago
steps.go	Fix MoveDisallowedReplicas suboptimal broker seelction	a year ago
utils.go	Use pure RMS to compute unbalance	a year ago
xbuild.sh	xbuild.sh	2 years ago

README.md

kafkabalancer

Rebalance your kafka topics, partitions, replicas across your cluster

Purpose

`kafkabalancer` allows you to compute the set of rebalancing operations yielding a minimally-unbalanced kafka cluster, given a set of constraints:

- set of allowed brokers (globally, or per partition)
- number of desired replicas (per partition)
- current distribution of replicas (per partition)
- leader reassignment enabled/disabled (globally)
- partition weight (per partition)

The goal is to minimize the workload difference between brokers in the cluster, where the workload of a broker is measured by the sum of the weights of each partition having a replica on that broker. Additionally leaders have to do more work (producers and consumers only operate on the leader, followers fetch from the leaders as well) so the weight applied to leader partitions is assumed to be proportional to the sum of the number of replicas and consumer groups.

The tool is designed to be used iteratively: at each iteration only a single reassignment operation is returned by kafkabalancer. This is useful in a automated framework like the following:

```
forever:
  if !kafka_cluster_is_nominal:
    continue
  state = get_current_state()
  change = kafkabalancer(state)
  if change:
    apply(change)
```

Installation

`go get github.com/cafxx/kafkabalancer`

Usage

Run `kafkabalancer -help` for usage instructions.

```
Usage of ./kafkabalancer:
  -allow-leader
        Consider the partition leader eligible for rebalancing
  -broker-ids string
        Comma-separated list of broker IDs (default "auto")
  -from-zk string
        Zookeeper connection string (can not be used with -input)
  -full-output
        Output the full partition list: by default only the changes are printed
  -help
        Display usage
  -input string
        Name of the file to read (if no file is specified read from stdin, can not be used with -from-zk)
  -input-json
        Parse the input as JSON
  -max-reassign int
        Maximum number of reassignments to generate (default 1)
  -min-replicas int
        Minimum number of replicas for a partition to be eligible for rebalancing (default 2)
  -min-unbalance float
        Minimum unbalance value required to perform rebalancing (default 1e-05)
  -pprof
        Enable CPU profiling
```

How to perform rebalancing

Getting the Kafka cluster state from zookeeper

To emit the first suggested rebalancing operation for your Kafka cluster (`$ZK` is the comma-separated list of your zookeeper brokers):

```
kafkabalancer -from-zk $ZK > reassignment.json
```

To perform the suggested change, run the following command:

```
kafka-reassign-partitions.sh --zookeeper $ZK --reassignment-json-file reassignment.json --execute
```

If you want to generate/run more than a single rebalancing operation, specify a value greater than `1` for `-max-reassign`.

Getting the Kafka cluster state from a dump of the partition list

First dump the list of partitions from your Kafka broker (`$ZK` is the comma-separated list of your zookeeper brokers):

```
kafka-topics.sh --zookeeper $ZK --describe > kafka-topics.txt
```

Next run `kafkabalancer` on the list (note: this assumes that all partitions have the same weight and no consumers; this is functionally OK but could lead to suboptimal load distribution). `kafkabalancer` will analyze the list and suggest one or more reassignments:

```
kafkabalancer -input kafka-topics.txt > reassignment.json
```

To perform the suggested change(s), run the following command:

```
kafka-reassign-partitions.sh --zookeeper $ZK --reassignment-json-file reassignment.json --execute
```

If you want to generate/run more than a single rebalancing operation, specify a value greater than `1` for `-max-reassign`.

Features

- parse the output of kafka-topic.sh --describe or the Kafka cluster state in Zookeeper
- parse the reassignment JSON format
- output the reassignment JSON format
- minimize leader unbalance (maximize global throughput)

Planned

- parse the output of kafka-offset.sh to get the per-partiton weights (number of messages)
- fetch elsewhere additional metrics to refine the weights (e.g. number of consumers, size of messages)
- minimize same-broker colocation of partitions of the same topic (maximize per-topic throughput)
- proactively minimize unbalance caused by broker failure (i.e. minimize unbalance caused by one or more brokers failing) keeping into considerations how followers are elected to leaders when the leader fails
- consider N-way rebalancing plans (e.g. swap two replicas) to avoid local minima
- prefer to relocate "small" partitions to minimize the additional load due to moving data between brokers
- use something like https://github.com/wvanbergen/kazoo-go to query state directly
- use something like https://github.com/wvanbergen/kazoo-go to apply changes directly

Scenarios

This section lists some examples of how `kafkabalancer` operates.

Adding brokers

Setting `broker-ids=1,2,3` will move partition 1 from broker 2 to broker 3 to equalize the load:

Part	Original	Output
1	1,2	1,3
2	2,1	2,1

Setting `broker-ids=1,2,3,4` will move partition 1 from brokers 1,2 to brokers 4,3 to equalize the load:

Part	Original	Output
1	1,2	4,3
2	2,1	2,1

Removing brokers

Setting `broker-ids=1,2` will move partition 3 from broker 3 to broker 2 to equalize the load:

Part	Original	Output
1	1,2	1,2
2	1	1
3	3	2

Setting `broker-ids=1` will return error because the partition 1 requires 2 replicas.

Add replicas

Setting `NumReplicas=2` for partition 3 will add a replica on broker 2 to equalize the load.

Part	Original	Output
1	1,2	1,2

Part	Original	Output
2	1,3	1,3
3	3	2,3

Remove replicas

Setting `NumReplicas=1` for partition 1 will remove the replica from broker 1 to equalize the load.

Part	Original	Output
1	1,2	2
2	1	1

Automated rebalancing

If no changes need to be made, `kafkabalancer` will simply seek to equalize the load between brokers:

Part	Original	Output
1	1,2,3	1,4,3
2	1,2,4	1,2,4
3	1,2,3	1,2,3

If leader moving is enabled, also the leaders are eligible for rebalancing:

Part	Original	Output
1	1,2,3	4,2,1
2	1,2,4	3,2,4
3	1,2,3	1,2,3

How is rebalancing done

`kafkabalancer` rebalancing capabilities are split in a series of step executed in order. The order of steps is chosen to prioritize constraints application first and then performance optimization.

The steps are defined in `steps.go` and the ordering of the steps in `balancer.go`.

When the steps need to identify the relative load of the cluster nodes, they use each partition weight as a relative measure of the workload the cluster has to sustain for that particular partition. The partition weight is then scaled by a multiplier and added to the total load of each node; the multiplier depends on the role of the node for that partition and is defined as:

Leader	Follower
`(Replicas)+(Consumers)`	`1`

Where `(Replicas)` and `(Consumers)` are, respectively, the number of replicas and consumers of the partition.

`ValidateWeights`, `ValidateReplicas` and `FillDefaults`

These steps simply validate that the input data is consistent and they fill in any default value that is not explicitely defined.

`RemoveExtraReplicas` and `AddMissingReplicas`

These steps deal with any changes in the desired number of replicas by either removing replicas from the highest-loaded cluster nodes or by adding replicas to the lowest-loaded cluster nodes.

`MoveDisallowedReplicas`

This step detects if any replica is currently on a broker that is not in the list of allowed brokers and, if so, it moves those replicas to the lowest-loaded allowed brokers.

`MoveLeaders` and `MoveNonLeaders`

These steps attempt to redistribute replicas to minimize the load difference between brokers (see the section above to understand the metric used to measure load on each broker).

`MoveLeaders` is a no-op if you don't specify `-allow-leader`. Leaders are moved before followers because the weight of leader partitions is normally greater than the one of follower partitions.

Author

Carlo Alberto Ferraris (@cafxx)

License