KIP-578: Add configuration to limit number of partitions

Author: Gokul Ramanan Subramanian

Contributors: Alexandre Dupriez, Tom Bentley, Colin McCabe, Ismael Juma, Boyang Chen, Stanislav Kozlovski

Status

Current state: Voting

Discussion thread: Here

JIRA: *Unable to render Jira issues macro, execution error.*

PR: https://github.com/apache/kafka/pull/8499 (currently only prototype, slightly out of date wrt KIP, but gets the idea across)

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

The number of partitions is a lever in controlling the performance of a Kafka cluster. Increasing the number of partitions can lead to higher performance. However, increasing the number beyond a point can lead to degraded performance on various fronts.

The current generic recommendation to have no more than 4000 partitions per broker and no more than 200000 partitions per cluster is not enforced by Kafka. We have seen multiple issues in production clusters where having a large number of partitions leads to live-locked clusters that are so busy that even topic deletion requests intended to alleviate the problem do not complete.

We did some performance experiments to understand the effect of increasing the number of partitions. See Appendix A1 for producer performance, and A2 for topic creation and deletion times. These consistently indicate that having a large number of partitions can lead to a malfunctioning cluster.

Topic creation policy plugins specified via the *create.topic.policy.class.name* configuration can partially help solve this problem by rejecting requests that result in a large number of partitions. However, these policies cannot produce a replica assignment that respects the partitions limits, instead they can only either accept or reject a request. Therefore, we need a more native solution for addressing the problem of partition limit-aware replica assignment. (See rejected alternatives for more details on why the policy approach does not work.)

We propose having two configurations (a) *max.broker.partitions* to limit the number of partitions per broker, and (b) *max.partitions* to limit the number of partitions in the cluster overall. These can act as guard rails ensuring that the cluster is never operating with a higher number of partitions than it can handle.

Goals

- These limits are cluster-wide. This is obviously true for *max.partitions* which is meant to apply at the cluster level. However, we choose this for *max.broker.partitions* too, instead of supporting different values for each broker. This is in alignment with the current recommendation to run homogenous Kafka clusters where all brokers have the same specifications (CPU, RAM, disk etc.).
- If both limits *max.partitions* and *max.broker.partitions* are specified, then the more restrictive of the two apply. It is possible that a request is rejected because it causes the *max.partitions* limit to be hit without causing any broker to hit the *max.broker.partitions* limit. The vice versa is true as well.
- These limits can be changed at runtime, without restarting brokers. This provides greater flexibility. See the "Rejected alternatives" section for why we did not go with read-only configuration.
- These limits won't apply to topics created via auto topic creation (currently possible via *Metadata* API requests) until KIP-590. With KIP-590, auto-topic creation will leverage the CreateTopics API, and will have same behavior as the creation of any other topic.

- These limits do not apply to internal topics (i.e. __consumer_offsets and __transaction_state), which usually are not configured with too many partitions. This ensures that any internal Kafka behaviors do not break because of partition limits. The topic partitions corresponding to these internal topics won't also count towards the limit.
- These limits do not apply when creating topics or partitions, or reassigning partitions via the ZooKeeper-based admin tools. This is unfortunate, because it does create a backdoor to bypass these limits. However, we leave this out of scope here given that ZooKeeper will eventually be deprecated from Kafka.

Public Interfaces

Configs

Config name	Type	Default	Update-mode
max.broker.partitions	int64	int64's max value (2^{63} - 1)	cluster-wide
max.partitions	int64	int64's max value (2^{63} - 1)	cluster-wide

Kafka administrators can specify these in the server.properties file.

They can also use the following to set/modify these configurations via the kafka-config.sh admin tool.

```
./kafka-config.sh --bootstrap-server $SERVERS --alter --add-config max.broker.partitions=4000 --entity-type
brokers --entity-default
./kafka-config.sh --bootstrap-server $SERVERS --alter --add-config max.partitions=200000 --entity-type brokers
--entity-default
```

It is also possible to set this value per broker via the following command, which applies the change to only a specific broker, for testing purposes.

```
./kafka-config.sh --bootstrap-server $SERVERS --alter --add-config max.broker.partitions=4000 --entity-type
brokers --entity-name 1
./kafka-config.sh --bootstrap-server $SERVERS --alter --add-config max.partitions=200000 --entity-type brokers
--entity-name 1
```

However, if different values are specified for different brokers, then only the value that applies to the broker handling the request will matter. This is the controller in most cases, but can be any broker in case of auto topic creation.

Further, the stricter of the two configurations *max.broker.partitions* and *max.partitions* will apply.

An illustrative (toy) example - CreateTopic

For example, in a 3 broker cluster, say that *max.broker.partitions* is configured equal to 10. If the brokers already host 8, 6 and 9 partitions respectively, then a request to create a new topic with 1 partition and a replication factor 3 can be satisfied, resulting in partition counts of 9, 7 and 10 respectively. However, the next topic creation request for 1 partition with a replication factor of 3 will fail because broker 3 has already reached the limit. A similar request with a replication factor of 2 can however, succeed, because 2 of the brokers have still not reached the limit. If the original request for a topic with 1 partition was actually a request for a topic with 2 partitions, with a replication factor of 3, then the request would have failed in entirety.

API Exceptions

CreateTopics, *CreatePartitions,* and AlterPartitionReassignments APIs will throw *PolicyViolationException* and correspondingly the *POLICY_VIOLATION (44)* error code if it is not possible to satisfy the request while respecting the *max.broker.partitions* or *max.partitions* limits. This applies to *Metadata* request s only in case auto-topic creation is enabled post KIP-590, which will modify the Metadata API to call CreateTopics. We will bump up the version of these APIs by one for new clients.

The actual exception will contain the values of *max.broker.partitions* and *max.partitions* in order to make it easy for users to understand why their request got rejected.

Proposed Changes

The following table shows the list of methods that will need to change in order to support the *max.broker.partitions* and *max.partitions* configurations. (We skip a few internal methods for the sake of simplicity.)

Method name	Description of what the method does currently	Context in which used	Relevant methods which directly depend on this one	Relevant methods on which this one is directly dependent
`AdminUtils.				

assignReplicasToBrokers`	• Encapsulates the algorithm specified in KIP-36 to assign partitions to brokers on as many racks as possible. This also handles the case when rack-awareness is disabled.
	• This is a pure function without any state or side effects.
		• API
		• ZooKeeper-based admin tools
			`AdminZkClient.createTopic`
			
			`AdminZkClient.addPartitions`
			
			`AdminManager.createTopics`
			
			`ReassignPartitionsCommand.generateAssignment`	

`AdminZkClient.createTopicWithAssignment`	Creates the ZooKeeper znodes required for topic-specific configuration and replica assignments for the partitions of the topic.	• API
		• ZooKeeper-based admin tools
			`AdminZkClient.createTopic`
			
			`AdminManager.createTopics`
			
			`ZookeeperTopicService.createTopic`	

`AdminZkClient.createTopic`	Computes replica assignment using `AdminUtils.assignReplicasToBrokers` and then reuses `AdminZkClient.createTopicWithAssignment`.	• API
		• ZooKeeper-based admin tools
			`KafkaApis.createTopic`
			
			`ZookeeperTopicService.createTopic`
				`AdminUtils.assignReplicasToBrokers`
				
				`AdminZkClient.createTopicWithAssignment`

`AdminZkClient.addPartitions`	• Computes replica assignment using `AdminUtils.assignReplicasToBrokers` when replica assignments are not specified.
	• When replica assignments are specified, uses them as is.
	• Creates the ZooKeeper znodes required for the new partitions with the corresponding replica assignments.
		• API
		• ZooKeeper-based admin tools
			`AdminManager.createPartitions`
			
			`ZookeeperTopicService.alterTopic`
				`AdminUtils.assignReplicasToBrokers`

`AdminManager.createTopics`	• Used exclusively by `KafkaApis.handleCreateTopicsRequest` to create topics.
	• Reuses `AdminUtils.assignReplicasToBrokers` when replica assignments are not specified.
	• When replica assignments are specified, uses them as is.
		• API	`KafkaApis.handleCreateTopicsRequest`	`AdminUtils.assignReplicasToBrokers`
				
				`AdminZkClient.createTopicWithAssignment`

`AdminManager.createPartitions`	Used exclusively by `KafkaApis.handleCreatePartitionsRequest` to create partitions on an existing topic.	• API	`KafkaApis.handleCreatePartitionsRequest`	`AdminZkClient.addPartitions`
`KafkaController.onPartitionReassignment`	Handles all the modifications required on ZooKeeper znodes and sending API requests required for moving partitions from some brokers to others.	• API	`KafkaApis.handleAlterPartitionReassignmentsRequest` (not quite directly, but the stack trace in the middle is not relevant)	
`KafkaApis.handleCreateTopicsRequest`	Handles the *CreateTopics* API request sent to a broker, if that broker is the controller.	• API		`AdminManager.createTopics`
`KafkaApis.handleCreatePartitionsRequest`	Handles the *CreatePartitions* API request sent to a broker, if that broker is the controller.	• API		`AdminManager.createPartitions`
`KafkaApis.handleAlterPartitionReassignmentsRequest`	Handles the *AlterPartitionReassignments* API request sent to a broker, if that broker is the controller.	• API		`KafkaController.onPartitionReassignment` (not quite directly, but the stack trace in the middle is not relevant)
`KafkaApis.createTopic`	• Creates internal topics for storing consumer offsets (*__consumer_offsets*), and transaction state (*__transaction_state*). • Also used to auto-create topics when topic auto-creation is enabled.	• API	`KafkaApis.handleTopicMetadataRequest` (not quite directly, but the stack trace in the middle is not relevant)	`AdminZkClient.createTopic`
`KafkaApis.handleTopicMetadataRequest`	Handles the *Metadata* API request sent to a broker.	• API		`KafkaApis.createTopic` (not quite directly, but the stack trace in the middle is not relevant)
`ZookeeperTopicService.createTopic`				`AdminZkClient.createTopic`

	<ul><li>Used by the *./kafka-topics.sh* admin tool to create topics when --*zookeeper* is specified.</li><li>Reuses `AdminZkClient.createTopic` when no replica assignments are specified.</li><li>Reuses `AdminZkClient.createTopicWithAssignment` when replica assignments are specified.</li></ul>	<ul><li>ZooKeeper-based admin tools</li></ul>		`AdminZkClient.createTopicWithAssignment`
`ZookeeperTopicService.alterTopic`	<ul><li>Used by the *./kafka-topics.sh* admin tool to alter topics when --*zookeeper* is specified.</li><li>Calls `AdminZkClient.addPartitions` if topic alteration involves a different number of partitions than what the topic currently has.</li></ul>	<ul><li>ZooKeeper-based admin tools</li></ul>		`AdminZkClient.addPartitions`
`ReassignPartitionsCommand.generateAssignment`	Used by the *./kafka-reassign-partitions.sh* admin tool to generate a replica assignment of partitions for the specified topics onto the set of specified brokers.	<ul><li>ZooKeeper-based admin tools</li></ul>		`AdminUtils.assignReplicasToBrokers`

For all the methods in the above table that are used in the context of both Kafka API request handling paths and ZooKeeper-based admin tools (`AdminUtils.assignReplicasToBrokers`, `AdminZkClient.createTopicWithAssignment`, `AdminZkClient.createTopic` and `AdminZkClient.addPartitions`), we will pass the values for maximum number of partitions per broker, maximum number of partitions overall, and the current number of partitions for each broker as arguments.

We will modify the core algorithm for replica assignment in the `AdminUtils.assignReplicasToBrokers` method. The modified algorithm will ensure that as replicas are being assigned to brokers iteratively one partition at a time, if assigning the next partition to a broker causes the broker to exceed the *max.broker.partitions* limit, then the broker is skipped. If all brokers are skipped successively in a row, then the algorithm will terminate and throw *PolicyViolationException*. The check for *max.partitions* is much simpler and based purely on the total number of partitions that exist across all brokers.

When the methods are invoked in the context of a Kafka API call, we will get the values for the maximum number of partitions per broker by reading the *max.broker.partitions* configuration from the `KafkaConfig` object (which holds the current value after applying precedence rules on configuration supplied via server.properties and those set via ZooKeeper). Similarly, we will get the maximum number of partitions overall by reading the *max.partitions* configuration from the `KafkaConfig` object. We will fetch the current number of partitions for each broker from either the `AdminManager` or `KafkaControllerContext` depending on the method.

When the methods are invoked in the context of ZooKeeper-based admin tools, we will set these limits equal to the maximum *int64* value that Java can represent. This is basically because it is not easy (and we don't want to make it easy) to get a reference to the broker-specific `KafkaConfig` object in this context. We will also set the object representing the current number of partitions for each broker to *None*, since it is not relevant when the limits are not specified.

Compatibility, Deprecation, and Migration Plan

This change is backwards-compatible in practice because we will set the default values for *max.broker.partitions* and *max.partitions* equal to the maximum *int64* value that Java can represent, which is quite large (2^{63} - 1). Users will anyway run into system issues far before hitting these limits.

In order to ease migration, a broker that already has more than *max.broker.partitions* number of partitions at the time at which *max.broker.partitions* configuration is set for that broker, will continue to function just fine for the existing partitions although it will be unable to host any further partitions. The Kafka administrator can later reassign partitions from this broker to another in order to get the broker to respect the *max.broker.partitions* limit.

Similarly, a cluster that already has more than *max.partitions* number of partitions at the time at which *max.partitions* configuration is set, will continue to function just fine. It will however, fail any further requests to create topics or partitions. Any reassignment of partitions should work fine.

These soft behaviors are also necessary because (even with this KIP), users can bypass the limit checks by using ZooKeeper-based admin tools.

Rejected Alternatives

Add configuration to limit number of partitions per topic

Having a limit at the topic level does not help address the problem discussed in this KIP if there is large number of topics. While each topic may only have a limited number of partitions, it is possible that there will be many more partitions than on a broker than it can handle efficiently.

Add configuration to limit the number of topics

Having a limit on the number of topics does not help address the problem discussed in this KIP if there is a large number of partitions on the topic.

Make these configurations read-only

This approach makes administration a bit easier because once the limits are set, the users of the cluster cannot accidentally change the value without administrator privileges and without restarting the brokers. This can provide an additional safety net.

However, in environments such as Docker where it is possible to allocate more resources to the broker on the fly, it would be restrictive to not be able to modify the *max.broker.partitions* and *max.partitions* configurations on the fly as well.

Further having these configurations be read-only is not flexible. The number of partitions that a broker can handle depends on the replication factor of these partitions. Smaller the replication factor, lower is the incoming traffic due to replication *Fetch* requests that the broker has to handle. This allows the broker to use its resources more efficiently to handle the client requests such as produce / consume. Therefore, a Kafka administrator may want to set different values on a broker as the workload changes without disrupting operations by restarting the brokers.

Support max.broker.partitions as a per-broker configuration

This is in general a more flexible approach than the one described in this KIP and allows having different brokers with different resources, each have its own *max.broker.partitions* configuration. However, this would require sending the broker-specific configuration to the controller, which needs this while creating topics and partitions or reassigning partitions. One approach would be to put this information into the broker's ZooKeeper znode and have the controller rely on that. And the other would be to create a new API request-response that brokers can use to share this information with the controller. Both of these approaches introduce complexity for little gain. We are not aware of any clusters that are running with heterogenous configurations where having different *max.broker.partitions* configuration for each broker would help. Therefore, in this KIP, we do not take this approach.

Use configurable topic policies for limiting number of partitions

Kafka allows plugging in custom topic creation policies via the *create.topic.policy.class.name* configuration. This allows administrators to install a policy that can limit number of partitions. There are a few downsides to using this approach

- such a configuration is not available for partition increase or reassignment, which even if we can address do not fix the next problem.
- partition limits are not "yet another" policy configuration. Instead, they are fundamental to partition assignment. i.e. the partition assignment algorithm needs to be aware of the partition limits. To illustrate this, imagine that you have 3 brokers (1, 2 and 3), with 10, 20 and 30 partitions each respectively, and a limit of 40 partitions on each broker enforced via the configurable policy class. This leaves extra leg room for 30, 20 and 10 partitions respectively on the 3 brokers. This adds up to a total legroom of 60 partitions. It should be possible to create a topic with 30 partitions and replication factor of 2 with this configuration. Assign the first 10 partitions to brokers 1 and 3; then assign the next 20 partitions to brokers 1 and 2. While the configurable policy class may accept a topic creation request for 30 partitions with a replication factor of 2 each (because it is satisfiable), the non-pluggable partition assignment algorithm (in AdminUtils.assignReplicasToBrokers) has to do the assignment in such a way as to not violate the partition limits.

Basically, partition limits cannot be viewed as a policy on top of topic creation. They are integral to topic creation / partition increase and reassignment.

Add configuration to limit number of partitions that a specific user can create

A large number of partitions can cause performance issues for a Kafka cluster irrespective of which user created those partitions. The focus of the KIP is to prevent the Kafka cluster from entering into a bad state when having a large number of partitions. Therefore, it does not focus on addressing the orthogonal use case of having partition quotas per user in a multi-tenant environment.

Appendix A: Performance with a large number of partitions

Our setup had had 3 m5.large EC2 broker instances on 3 different AZs within the same AWS region us-east-1, running Kafka version 2.3.1. Each broker had an EBS GP2 volume attached to it for data storage. All communication was plaintext and records were not compressed. The brokers each had 8 IO threads (*num.io.threads*), 2 replica fetcher threads (*num.replica.fetchers*) and 5 network threads (*num.network.threads*).

A1: Producer performance

We did a performance test (using kafka-producer-perf-test.sh from a single m5.4xlarge EC2 instance). On the producer side, each record was 1 KB in size. The batch size (*batch.size*) and artificial delay (*linger.ms*) were left at their default values.

The following table shows results from a producer performance test. The table indicates that throughput with 3-way replication improves from 52Mbps to 101Mbps in going from 10 to 100 partitions, but then degrades beyond that. Also, the throughput with 1-way replication is better compared to that with 3-way replication. This is because of the number of replication *Fetch* requests that a broker receives increases with the number of partitions on the broker, and having 1-way replication means that the broker does not have to deal with *Fetch* requests. But even so, the performance is much worse with 10000 partitions than with 10 partitions.

Produce throughput (Mbps)						
Replication	**Number of partitions on a broker**					
	10	**100**	**500**	**1000**	**5000**	**10000**
3-way	52	101	86	12	2.5	0.9
1-way	142	113	104	99	24	16

We also stress tested the system with a single topic consisting of 30000 partitions each with 3-way replication. The producer performance test provided basically 0 throughput. Most requests returned NOT_ENOUGH_REPLICAS error, indicating that replication is backlogged, and brokers are also unable to handle the *Produce* requests. The UnderMinIsrPartitionCount was equal to the LeaderCount on all brokers, confirming this hypothesis.

A2: Topic creation and deletion times

For stress testing the system with a large number of partitions, we created a single topic with 30000 partitions with 3-way replication. It took 30 minutes for the necessary (30000) log directories to get created on the brokers after the point in time when the PartitionCount metric indicated that the brokers have 30000 partitions each.

We then tried to delete the topic with no *Produce* / *Fetch* traffic ongoing. However, topic deletion did not complete. From the GC logs, we noticed that all brokers had almost maxed out their allocated heap (Xmx) of 2GB, and the processes were busy in G1 garbage collection, causing the ZooKeeper sessions to repeatedly timeout (with session timeout of 6 seconds). Restarting the brokers reduced heap usage to 800MB, but did not cause the partitions to get deleted.

Although deletion did not happen, the LeaderCount did drop to 0 in 1 hour 40 minutes. We noticed that leadership resignation process was happening more rapidly as the number of partitions with leadership decreased. The following table shows how long it took for the last N partitions (out of the total 30000 partitions) took to not have any leader.

We can see that leadership resignation times are exponential in the number of partitions.

Leadership resignation time (minutes)						
Number of partitions left						
30000	**20000**	**10000**	**5000**	**1000**	**100**	**10**
100	43	9	3	< 1	< 1	< 1