KIP-435: Internal Partition Reassignment Batching

Status

Current state: Under Discussion

Discussion thread: *here*

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

Planned Release: 2.4

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

Motivation

If partition reassignment involves a lot of replicas, then it could put too much overhead on the brokers.

Say you have a replication factor of 4 and you trigger a reassignment which moves all replicas to new brokers. Now 8 replicas are fetching at the same time which means you need to account for 8 times the current producer load plus the catch-up replication. To make matters worse, the replicas won't all become in-sync at the same time; in the worst case, you could have 7 replicas in-sync while one is still catching up. Currently, the old replicas won't be disabled until all new replicas are in-sync. This makes configuring the throttle tricky since ISR traffic is not subject to it.

Rather than trying to bring all 4 new replicas online at the same time, a friendlier approach would be to do it incrementally: bring one replica online, bring it in-sync, then remove one of the old replicas. Repeat until all replicas have been changed. This would reduce the impact of a reassignment and make configuring the throttle easier at the cost of a slower overall reassignment.

Furthermore since the controller has a good knowledge about the cluster it makes sense to improve its reassignment feature to allow to internally batch the given reassignment. Therefore this KIP aims to change the controller to accommodate the internal batching rather than adding a new tool.

Public Interfaces

Config Changes

Three new configs will be added. All of these configs are cluster-wide which means they are global configs affecting the entire cluster.

Config name	Type	Default	Valid values	Importance	Dynamic update mode	Description

reassignment.max.concurrent.leader.movements	int	Int.MAX	[1,...]	medium	cluster-wide	This new configuration would tell how many replicas of a single partition can be moved at once.
reassignment.max.concurrent.partition.movements	int	Int.MAX	[1,...]	medium	cluster-wide	This configuration puts an upper limit on how many partition reassignments can be run concurrently. To calculate the sum of concurrent movements one can multiply this config by `reassignment.max.parallel.replica.count`.
reassignment.max.concurrent.replica.movements	int	Int.MAX	[1,...]	medium	per-partition	This one puts an upper limit on concurrent replica movements. It is useful to reduce the controller burden on big reassignments.

Protocol Changes

We will add a field to the `ListPartitionReassignmentsResponse` protocol (added by KIP-455) that will extend the response with the current reassignment batch. We'll increment the API version of this protocol with this change.

ListPartitionReassignmentsResponse

```
{
  "apiKey": 46,
  "type": "response",
  "name": "ListPartitionReassignmentsResponse",
  "validVersions": "0-1",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The top-level error code, or 0 on success." },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+",
      "about": "The top-level error message, or null on success." },
    { "name": "Topics", "type": "[]OngoingTopicReassignment", "versions": "0+",
      "about": "The ongoing reassignments for each topic.", "fields": [
        { "name": "Name", "type": "string", "versions": "0+",
          "about": "The topic name." },
        { "name": "Partitions", "type": "[]OngoingPartitionReassignment", "versions": "0+",
          "about": "The ongoing reassignments for each partition.", "fields": [
          { "name": "PartitionId", "type": "int32", "versions": "0+",
            "about": "The partition ID." },
          { "name": "CurrentBrokers", "type": "[]int32", "versions": "0+",
            "about": "The broker IDs which the partition is currently assigned to." },
          { "name": "TargetBrokers", "type": "[]int32", "versions": "0+",
            "about": "The broker IDs which the partition is being reassigned to." }
      ]}
    ]},
    { "name": "Topics", "type": "[]CurrentTopicReassignmentBatch", "versions": "1+",
      "about": "The currently executed reassignment batch.", "fields": [
        { "name": "Name", "type": "string", "versions": "1+",
          "about": "The topic name." },
        { "name": "Partitions", "type": "[]CurrentPartitionReassignmentBatch", "versions": "1+",
          "about": "The current reassignments for each partition.", "fields": [
          { "name": "PartitionId", "type": "int32", "versions": "1+",
            "about": "The partition ID." },
          { "name": "CurrentBrokers", "type": "[]int32", "versions": "1+",
            "about": "The broker IDs which the partition is currently assigned to." },
          { "name": "TargetBrokers", "type": "[]int32", "versions": "1+",
            "about": "The broker IDs which the partition is being reassigned to." }
      ]}
    ]}
  ]
}
```

Proposed Changes

As explained above, the goal would be to incrementally add new partitions, a batch at a time to avoid putting much pressure on the brokers. The only exception is the first step where (if needed) we add that many replicas that is enough to fulfil the min.insync.replicas requirement set on the broker, even if it exceeds the limit on parallel replica reassignments. For instance if there are 4 brokers, min.insync.replicas set to 3 but there are only 1 in-sync replica, then we immediately add 2 other in one step, so the producers are able to continue.

Furthermore in the first step we'll elect the new preferred leader (if the reassignment requires it) to unload pressure from the current leader.

The configs are aiming to control batching on partition and topic levels. We practically default to the current behaviour to remain backward compatible, although as a future work it might make sense to lower the defaults based on feedback.

For instance in case of a reassignment for a single partition from (0, 1, 2, 3, 4) to (5, 6, 7, 8, 9) we would form the batches (0, 1) (5, 6); (2, 3) (7, 8) and 4 9 and would execute the reassignment in these increments, depending on how many parallel replica reassignments do we allow. For multiple partitions it would work in a similar fashion but the reassignment.parallel.replica.count would control how many replicas of that partition can be reassigned concurrently. On top of these we would control how many leaders could be reassigned in parallel. That means that after we calculated the possible reassignment steps we disallow those which would involve leader movement over the limit and instead if possible add reassignments that involve no leader movement. It might be possible that we can't fill their place and we won't fill reassignment.parallel.partition.count. In this case we fill up the batch limit with reassignments on partitions that don't require leader movement.

As an addition these values could be changed dynamically to somewhat "throttle" the reassignment. This kind of throttling would only affect the next reassignment step calculation and would leave the currently running one as it is. It might be better to throttle certain reassignment on a much more advanced way but it could also exceed the scope of this KIP.

Algorithm

Calculating a Reassignment Step (partition level)

1. For calculating a reassignment step, always the final target replica (FTR) set and the current replica (CR) set is used.
2. Calculate the replicas to be dropped (DR):
    a. Calculate `n = max(reassignment.parallel.replica.count, size(FTR) - size(CR))`
    b. Filter those replicas from CR which are not in FTR, this is the excess replica (ER) set
    c. Take the first *reassignment.parallel.replica.count* replicas of ER, that will be the set of dropped replicas
3. Calculate the new replicas (NR) to be added
    a. Calculate that if the partition has enough online replicas to fulfil the min.insync.replicas config so the producers are able to continue.
    b. If the preferred leader is different in FTR and it is not yet reassigned, then add it to the NR
    c. If `size(NR) < min.insync.replicas` then take `min(min.insync.replicas, reassignment.parallel.replica.count) - size(NR)` replicas from FTR
    d. Otherwise take as many replica as *reassignment.parallel.replica.count* allows
4. Create the target replica (TR) set: CR + NR - DR
5. If this is the last step, then order the replicas as specified by FTR. This means that the last step is always equals to FTR

Collecting a reassignment batch

Collecting a reassignment batch

```
val R = reassignment.parallel.replica.count
val P = reassignment.parallel.partition.count
val L = reassignment.parallel.leader.movements

val batchSize = P
// split the individual partition reassignments whether they involve leader movement or not
val partitionMovements = calculateReassignmentStepsFor(partitionsToReassign).partition(partitionReassignment.
involvesLeaderReassignment)
// fill the batch with as much leader movements as possible and take the rest from other reassignments
val currentBatch = if (partitionMovements.leaderMovements.size < batchSize)
  partitionMovements.leaderMovements ++ partitionsToReassign.otherPartitionMovements.take(batchSize -
partitionMovements.leaderMovements.size)
else
 partitionMovements.leaderMovements.take(batchSize)
executeReassignmentOnBatch(currentBatch)
```

The algorithm basically calculates the next step for all the partitions to be reassigned (note that it isn't compute heavy operation) and then separates the leader movements from the rest. It will then tries to fill the batch with reassignments involving leader movement and the rest with other reassignments.

Performing reassignment for a single partition

New algorithm

Performing a reassignment step is somewhat similar in big picture to the currently existing algorithm. There will be `reassignment.parallel.partition.count` such algorithm running in parallel.

1. Calculate the next reassignment batch.
2. Wait until this step is not identical to the current assignment and there is at least one replica ISR.
    a. The calculation would result in identical steps if it's not able to add new replicas to the TR set. This could be because target brokers might be offline.
    b. We are not able to reassign partitions if the partition is offline
3. Update CR in Zookeeper (/brokers/topics/[topic]/partitions/[partitionId]/state) with TR for the given partitions.
4. Send LeaderAndIsr requests to all replicas in CR+NR.
5. Start new replicas in NR by moving them into the NewReplica state.
6. Set CR to TR in memory.
7. Send LeaderAndIsr request with a potential new leader (if current leader not in TR) and a new CR (using TR) and same isr to every broker in TR
8. Replicas in DR -> Offline (force those replicas out of isr)
9. Replicas in DR -> NonExistentReplica (force those replicas to be deleted)
10. Update the /admin/reassign_partitions path in ZK to remove this partition if the reassignment is completed.

Existing algorithm

RAR = Reassigned replicas
OAR = Original list of replicas for partition
AR = current assigned replicas

1. Update AR in ZK with OAR + RAR.
2. Send LeaderAndIsr request to every replica in OAR + RAR (with AR as OAR + RAR). We do this by forcing an update of the leader epoch in zookeeper.
3. Start new replicas RAR - OAR by moving replicas in RAR - OAR to NewReplica state.
4. Wait until all replicas in RAR are in sync with the leader.
5. Move all replicas in RAR to OnlineReplica state.
6. Set AR to RAR in memory.
7. If the leader is not in RAR, elect a new leader from RAR. If new leader needs to be elected from RAR, a LeaderAndIsr will be sent. If not, then leader epoch will be incremented in zookeeper and a LeaderAndIsr request will be sent. In any case, the LeaderAndIsr request will have AR = RAR. This will prevent the leader from adding any replica in RAR - OAR back in the isr.
8. Move all replicas in OAR - RAR to OfflineReplica state. As part of OfflineReplica state change, we shrink the isr to remove OAR - RAR in zookeeper and send a LeaderAndIsr ONLY to the Leader to notify it of the shrunk isr. After that, we send a StopReplica (delete = false) to the replicas in OAR - RAR.
9. Move all replicas in OAR - RAR to NonExistentReplica state. This will send a StopReplica (delete = true) to the replicas in OAR - RAR to physically delete the replicas on disk.
10. Update AR in ZK with RAR.
11. Update the /admin/reassign_partitions path in ZK to remove this partition.
12. After electing leader, the replicas and isr information changes. So resend the update metadata request to every broker.

Example

The following code block shows how a transition happens from `(0, 1, 2, 3, 4)` into `(5, 6, 7, 8, 9)` where the initial leader is 0.

Reassignment Example

```
 (0, 1, 2, 3, 4)     // starting assignment
        |
(5, 0, 1, 2, 3, 4)   // +5, new leader (5) is elected
        |
 (5, 6, 2, 3, 4)   // -[0,1] +[6]
        |
 (5, 6, 7, 8, 4)   // -[2,3] +[7,8]
        |
 (5, 6, 7, 8, 9)   // -[4] +[9], requested order is matched, reassignment finished
```

Compatibility, Deprecation, and Migration Plan

Since these changes won't affect any public interfaces, neither Zookeeper, there will be no compatibility issues.

Test Plan

- The existing unit tests will be parameterized so they would run with both modes
- Extra unit tests will be added to cover those cases that are not covered with the existing tests
- Ducktape tests would be parameterized to run with both modes
- If needed, extra ducktapes could be added to cover cases that are needed

Future Work

Replication Throttling

It would be useful to give an upper cap on the bandwidth of the replication so users won't overwhelm their cluster. This throttle could be controlled overall for all partition and perhaps it would make sense to do it on a per partition basis and only specify a max capacity at the overall level. KIP-73 covers some related tasks but that isn't specifically tailored strictly to reassignment but a bit more general.

Rejected Alternatives

If there are alternative ways of accomplishing the same thing, what were they? The purpose of this section is to motivate why the design is the way it is and not some other way.