KIP-441: Smooth Scaling Out for Kafka Streams

Status

Current state: *Accepted*

Vote thread: here

Discussion thread: here

JIRA:

key	summary	type	created	updated	due	assignee	reporter	priority	status	resolution

⚠ JQL and issue key arguments for this macro require at least one Jira application link to be configured

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

Motivation

This KIP is following on KIP-429 to improve Streams scaling out behavior. While KIP-429 is focused on decreasing the amount of time that progress is blocked on the rebalance itself, this KIP addresses a second phase of stoppage that happens today: upon receiving assignment of a stateful task, Streams has to catch it up to the head of its changelog before beginning to process it.

Currently, the Streams task assignment doesn't take into account how much work it will take to catch up a task, which leads to unfortunate cases in which, after a rebalance, some task can get hung up for hours while it rebuilds its state stores. The high level goal of this KIP is to allow the prior owner of that task to keep it *even if the assignment is now unbalanced* until the new owner gets caught up, and then to change ownership after the catch-up phase.

In short, the goals of this KIP are:

- Reduce unnecessary downtime due to task restoration and global application revocation.
- Better auto scaling experience for KStream applications.
- Stretch goal: better workload balance across KStream instances.

Background

Consumer Rebalance Protocol: Stop-The-World Effect

As mentioned in motivation, we also want to mitigate the stop-the-world effect of current global rebalance protocol. A quick recap of current rebalance semantics on KStream: when rebalance starts, all stream threads would

1. Join group with all currently assigned tasks revoked
2. Wait until group assignment finish to get assigned tasks and resume working
3. Replay the assigned tasks' states
4. Once all replay jobs finish, stream thread transits to running mode

After KIP-429, this changes to:

1. Joined the group, optimistically keeping all assigned tasks
2. Wait until group assignment finish to get tasks to revoke
3. Once the revocations are processed, get assigned tasks
4. Replay the assigned tasks' states
5. Once all replay jobs finish, stream thread transits to running mode

If you want to know more about details on protocol level, feel free to checkout KIP-429.

Constraints and Cost Function

While doing the assignment, we observe the following constraint:

- **Only one copy of a particular task can be hosted on a particular instance**. This means, a particular task, like `0_8` would have an "active" (i.e., primary) copy and some number of "standby" copies (i.e., replicas). A single *instance* (not thread) can only host one of those, either the active task or *one* of the standby tasks. We can't assign both an active and standby task with the same taskId to the same instance.

We also try to minimize the cost of the following cost function:

- **data parallel workload balance**. We make an assumption that all the partitions of a subtopology are roughly uniform effort. Therefore, we try to spread partitions across the cluster. For example, given two subtopologies X and Y, each with three partitions 0, 1, and 2, and three nodes A, B, and C, we'd prefer to allocate the tasks like (A: <X_0, Y_0>, B: <X_1, Y_1>, C: <X_2, Y_2>) instead of (eg) (A: <X_0, X_1>, B: <X_2, Y_0>, C: <Y_1, Y_2>).
- **even distribution of work**. We try to distribute tasks evenly across the cluster, instead of letting one thread have more tasks than another.
- **replay time**. We try to minimize the time it takes to replay a newly assigned task, which blocks further progress on that task. Note, although this is the focus of this KIP, this consideration was already part of Streams's assignment algorithm, as it's the reason we currently favor stickiness.

Note, this is a cost minimization problem, since it may not be possible to fully satisfy all three components of the cost function.

Proposed Assignment/Rebalance Algorithm

At a high level, we are proposing an iterative algorithm that prioritizes recovery time, while planning task movements to improve balance over consecutive rebalances. This proposal builds on KIP-429, so we assume the overhead of rebalancing is much less than it is today.

In each rebalance, the leader assigns active tasks only to instances that are ready to begin processing them immediately. If there is no such instance, then it assigns the active task to the instance that is closest to catching up.

The task movements are accomplished by assigning the task as a standby task to the destination instance, and then rebalancing once those tasks are up to date. The latter rebalance would find that a more balanced solution is available while still only assigning work to caught-up instances.

This means we need some mechanism to trigger a rebalance outside of group or topic changes (which are the only triggers currently). There are a number of ways to do this, but we propose to add a notion of "probing rebalance", discussed in detail below

If the leader determines that the current assignment is optimally balanced already, then it does not need to trigger any probing rebalances.

A note on the "moving" tasks. These are very similar to standby tasks, but we propose to give them a different name to make their role clear. This way, it's clear that they don't count against the "num.standby.replicas" config, and also that they represent an ephemeral increase in cluster storage requirements.

Parameters

- **acceptable_recovery_lag***: A scalar integer value indicating a task lag (number of offsets to catch up) that is acceptable for immediate assignment. Defaults to 10,000 (should be well under a minute and typically a few seconds, depending on workload). Must be at least 0.
- **num_standbys**: A scalar integer indicating the number of hot-standby task replicas to maintain in addition to the active processing tasks. Defaults to 0. Must be at least 0.
- **probing_rebalance_interval_ms***: A time interval representing the minimum amount of time (in milliseconds) that the leader should wait before triggering a "probing rebalance", assuming there is no intervening natural rebalance. Default is 10 minutes (because there would be effectively no cost for no-op rebalances). Must be at least 1 minute.
- **max_warmup_replicas***: A scalar integer representing the maximum number of *extra* replicas to assign for the purpose of moving a task to an instance that has neither the active nor any standby replicas of that task assigned. Used to throttle how much extra broker traffic and cluster state would be used for moving tasks. Defaults to 2. Minimum is 1.

* new config

Probing Rebalances

As of this KIP, an assignment may result in an unbalanced distribution of tasks in favor of returning to processing as soon as possible, while under-loaded nodes are assigned standby tasks to warm up. Once the under-loaded instances are warm, a next rebalance would be able to produce a more balanced assignment. This proposal creates a new requirement: the ability to trigger rebalances in response to conditions other than cluster or topic changes.

Ideally, group members would be able to communicate their current status back to the group leader so that the leader could trigger rebalances when the members catch up, but there is currently no such communication channel available. In lieu of that, we propose to have the leader trigger rebalances periodically, on a configured "probing_rebalance_interval". As each member re-joins the group, they would report their current lag (as proposed below), and the leader would potentially be able to compute a more balanced assignment.

Note that "balance" in this KIP, as today, is purely a function of task distribution over the instances of the cluster. Therefore, an assignment's balance can only change in response to changes in the cluster, or changes to the topic metadata (which could change the number of tasks). Both of these conditions already trigger a rebalance, so once the leader makes a balanced assignment, it can stop performing probing rebalances, confident that any later change to the balance of the cluster would trigger a rebalance automatically.

See the "Rejected Alternatives" section for a discussion of alternatives to probing rebalances.

Iterative Balancing Assignments

As mentioned in the overall description of the algorithm, the assignment algorithm creates as balanced an assignment as possible while strictly assigning stateful tasks only among the most-caught-up instances. It also plans task movements to gradually improve assignment balance over time, stopping when it is able to create a balanced assignment.

As described above, to move tasks to fresh instances while still respecting num.standby.replicas, the assignor will assign "extra" standby replicas to the destination instances. Once these extra replicas are caught up, a subsequent assignment will make that the active (or a permanent standby) and unassign one of the prior active or standby tasks.

A consequence of this design is that the overall amount of state in the cluster will exceed the sum of active and configured standby tasks while tasks are being moved. Also, each "moving" task reads from the broker, resulting in increased broker load and network traffic. To mitigate these effects, we'll introduce the **max_task_migrations** configuration, which limits the number of extra tasks that can be assigned at a time (inspired by Elasticsearch). The default is set to 2, but can be decreased to 1 or increased arbitrarily.

The default setting is likely to result in needing multiple balance improvements during probing rebalances over time, but the tradeoff is less load on the cluster and the broker. At the other end of the spectrum, setting the number high would allow the algorithm to assign all the desired movements in one shot, resulting in the whole process needing just two rebalances: the initial one to assign the moving tasks, and the final one to realize the movements and drop the old, unbalanced tasks.

Assignment Algorithm

The overall balancing algorithm relies on an assignment algorithm, which is not specified in this KIP, but is left as an implementation detail. We do specify some important properties of any implementation:

1. The assignment algorithm is *required* to assign active stateful tasks to the most-caught-up-instances (see below for a definition of "most caught-up"). I.e., active stateful tasks have the highest assignment priority.
2. As a second priority, the algorithm must assign standby tasks to the next-most-caught-up instances. All computed assignments must have at least "num.standby.replicas" number of standby replicas for *every* task.
3. The assignment algorithm may assign extra standby tasks to warm up instances that it wants to move existing active or standby tasks to.
4. Of course, the algorithm must also assign stateless tasks.
5. The algorithm must converge: if the current assignment is already balanced, it must not alter the assignment.
6. The algorithm should produce stable assignments. E.g., it should try not to shuffle stateless tasks randomly among nodes. Depending on the implementation, this may not be easy to guarantee, but it would improve overall performance to provide stability if possible. Note that the convergence requirement (#5) at least guarantees that once the assignment is balanced, it won't change anymore at all.

Computing the most-caught-up instances.

The assignment algorithm is required to assign active and standby tasks to the most-caught-up-instances (see below), with priority given to the active tasks. To do this, we need to define what a "most-caught-up" task is.

First, we'll sort all the tasks and instances, to give the algorithms a stable starting ordering. Then we compute a "rank" for each instance on each task. The "rank" represents how far from "caught up" that instance is on the task. Lower is more caught up. Rank has a floor of **acceptable_recovery_lag** . That is, all instances that have a lag under acceptable_recovery_lag are considered to have a lag of 0. This allows instances that are "nearly caught up" to be considered for active assignment.

In the case where an instance has no state for a task, we just use the total number of offsets in that task's changelog as the lag. If this isn't convenient, we can also use "max long" as a proxy.

Something like this:

```
SortedTasks: List<task> := sort(Tasks) // define a stable ordering of tasks
SortedInstances := List<instance> := sort(Instances) // define a stable ordering of instances

StatefulTasksToRankedCandidates: Map<task, Map<rank, instance>> := {}

// Build StatefulTasksToRankedCandidates. After this block, all tasks map to a ranked collection of all
instances, where the rank corresponds to the instance's lag on that task (lower is better).
for task in SortedTasks if task is stateful:
  for instance in SortedInstances:
    if (instance does not contain task lag):
      // then the rank is equivalent to the full set of offsets in the topic, guaranteed to be greater than any
instance's actual lag on the task.
      StatefulTasksToRankedCandidates[task][task.offsets] := instance
    else if (instance[task].lag <= acceptable_recovery_lag:
      // then the rank is 0, because any instance within acceptable_recovery_lag is considered caught up.
      StatefulTasksToRankedCandidates[task][0] := instance
    else:
      // then the rank is the actual lag
      StatefulTasksToRankedCandidates[task][instance[task].lag] := instance
```

For a given task, all the instances with the minimum rank are "most-caught-up".

Defining "balance"

An assignment is "balanced" if it displays both:

- instance balance: The instance with the most tasks has no more than 1 more task than the instance with the least tasks.
- subtopology balance: The subtopology with the most instances executing its tasks has no more than 1 more instance executing it than the subtopology with the least instances executing it. I.e.: the subtopology is partitioned. Each partition of a subtopology is a "task", ideally, we want to spread the tasks for a subtopology evenly over the cluster, since we observe that workload over the partitions of a subtopology is typically even, whereas workload over different subtopologies may vary dramatically.

Rebalance Metadata

The Consumer rebalance protocol has a pluggable extension point, the `PartitionAssignor` interface, along with a black-box `SubscriptionInfo` field. These mechanisms can be used in conjunction to encode extra information from the group members when they join the group, and then make a custom assignment decision. Finally, the `PartitionAssignor` assignment response contains another black-box field to encode extra information from the leader to the members.

Today, Streams uses the `StreamsPartitionAssignor` to implement its custom "balanced stickiness" assignment strategy, which is supported by the metadata fields in the subscription and assignment protocols.

More specifically on subscription:

```
KafkaConsumer:


Subscription => TopicList SubscriptionInfo
   TopicList               => List<String>
   SubscriptionInfo        => Bytes

------------------


StreamsPartitionAssignor:

SubscriptionInfo (encoded in version 6) => VersionId LatestSupportVersionId ClientUUID PrevTasks StandbyTasks
EndPoint

   VersionId               => Int32
   LatestSupportVersionId  => Int32
   ClientUUID              => 128bit
   PrevTasks               => Set<TaskId>
   StandbyTasks            => Set<TaskId>
   EndPoint                => HostInfo
```

To support the proposed algorithms, we're proposing a new, version 7, format for `SubsriptionInfo`:

```
SubscriptionInfo (encoded in version 5) => VersionId LatestSupportVersionId ClientUUID TaskLags EndPoint

   VersionId               => Int32
   LatestSupportVersionId  => Int32
   ClientUUID              => 128bit
   EndPoint                => HostInfo
   Task Lags               => Map<TaskId, Int64>    // new change
```

The new field, `TaskLags` would encode the lag for every store that the instance hosts, summed to the task level. This subsumes both PrevTasks and StandbyTasks

We do not need any modifications to `AssignmentInfo` at this time.

State with logging disabled

There is a special case to consider: stateful stores with logging disabled. We have an *a priori* expectation that stateful tasks are more heavyweight than stateless tasks, so from the perspective of cluster balance, we should consider tasks containing non-logged stores to be stateful (and therefore heavy). On the other hand, if a store is not logged, then there is no notion of being "caught up" or of having a "lag" on it. So, we should consider all instances to be equally caught-up on such stores.

Rather than building this special case into the code, we can simply have the members not add any lag information for non-logged stores. If a task has both logged and non-logged stores, we'd wind up with a task lag that represents the sum of the lags on the logged stores only. If the task has only non-logged stores, the the task would be completely absent from the "task lags" map.

On the leader/assignment side, following the algorithm above for computing `StatefulTasksToRankedCandidates` (to determine the degree of caught-up-ness for each instance on each stateful task), we would fill in a lag of "number of offsets in the changelog" for any instance that doesn't report a lag for a stateful task. For each store in the stateful task that doesn't have a changelog, we would consider its "number of offsets" to be zero. This has the effect of presenting all instances as equal candidates to receive stateful but non-logged tasks.

Stateless tasks

Similar to stateful non-logged tasks, stateless tasks have no notion of "catching up", and therefore there's no concept of a "lag", and we can consider all instances in the cluster to be equal candidates to execute a stateless task.

Again similarly, there's no need to report a lag at all for these tasks. Stateless tasks are not represented at all in `StatefulTasksToRankedCandidates`, so there's no need for special handling on the assignment side, and the assignor just packs the stateless tasks into the cluster to achieve the best balance it can with respect to the assignments of stateful tasks (which are specified to have higher assignment priority).

Race between assignment and state cleanup

Streams starts a background thread to clean up (delete) any state directories that are currently unassigned and have not been updated in `state.cleanup.delay.ms` amount of time.

This KIP introduces a race by reporting the lag on an instance's stores, not just the assigned ones. It is possible that a member will report a lag on an unassigned store, then, while the rebalance is ongoing, the state directory could be deleted (invalidating the lag). If the leader assigns the task to that instance, based on the now-stale lag information, then the member would actually have to rebuild the entire state from the start of the changelog.

Currently, this race is prevented because the state cleaner only runs if the Streams instance is not in "REBALANCING" state. However, as part of KIP-429, the REBALANCING state would be removed from Streams (as it would no longer be necessary). Thus, we need to prevent the race condition without relying on the REBALANCING state.

It is sufficient to create a mutex that the state cleaner must acquire before running (and hold for the duration of its run). If we also acquire the mutex before we report store lags (as part of the JoinGroup), and hold it until we get an assignment back (when the rebalance completes), then we can ensure the race cannot occur.

Therefore, we propose to create such a mutex, and acquire it during org.apache.kafka.clients.consumer.ConsumerPartitionAssignor#subscriptionUserData (when we would compute the store lags), and release it in org.apache.kafka.clients.consumer.ConsumerRebalanceListener#onPartitionsAssigned (when we know that the state directory locks have already been acquired for any newly assigned tasks). Also, we will hold the lock for the entire duration of any call to `org.apache.kafka.streams.processor.internals.StateDirectory#cleanRemovedTasks(long)`.

This builds an unfortunate three-way dependency between the state cleanup, Streams's ConsumerPartitionAssignor, and Streams's ConsumerRebalanceListener. Any cleaner solution would require changing the Consumer APIs, though.

Example Scenarios

Scaling Out

Stateful (active) Tasks := {T1, T2, T3}
Standby Replicas := 1
Instances := {**I1**, **I2**}

Initial State

	I1	**I2**
active	T1, T3	T2
standby	T2	T1, T3

-- New instance (**I3**) joins --

First Rebalance (catching-up)

	I1	**I2**	**I3**
active	T1, T3	T2	
standby	T2	T1, T3	T1, T3

-- **I3** finishes catching up, another rebalance is triggered --

Second Rebalance (stable state)

	I1	**I2**	**I3**
active	T1	T2	T3
standby	T2	T3	T1

Note that following KIP-429, the second rebalance will technically itself be composed of two rebalance since we are revoking the active task T3 from **I1.**

Scaling In (Pt. 1 – standbys in sync)

Stateful (active) Tasks := {T1, T2, T3, T4}
Standby Replicas := 1
Instances := {**I1**, **I2, I3**}

The standby tasks in this scenario are assumed to be in sync with the active tasks, that is, caught up and within *acceptable_recovery_lag.*

Initial State

	I1	**I2**	**I3**
active	T1, T4	T2	T3
standby	T3	T1, T4	T2

-- One instance (**I1**) is brought down --

First Rebalance

	I2	**I3**
active	T1, T4	T2, T3
standby	T2, T3	T1, T4

In this case, no subsequent rebalance is necessary (note that as of KIP-429 the first rebalance technically consists of two rebalances as an active task is being revoked). Processing resumes immediately as the standby tasks are ready to take over as active tasks.

Scaling In (Pt. 2 – standbys lag nontrivially)

Stateful (active) Tasks := {T1, T2, T3, T4}
Standby Replicas := 1
Instances := {**I1**, **I2, I3**}

In this scenario, the standby tasks are lagging the active tasks by some nontrivial amount, that is, by more than the *acceptable_recovery_lag.*

Initial State

	I1	I2	I3
active	T1, T4	T2	T3
standby	T3	T1, T4	T2

-- One instance (**I1**) is brought down --

First Rebalance (catching-up)

	I2	I3
active	T1, T2, T4	T3
standby	T3	T1, T2, T4

-- **I3** finishes restoring T4 --

Second Rebalance (stable state)

	I2	I3
active	T1, T2	T3, T4
standby	T3, T4	T1, T2

Unlike the previous example, **I3** cannot immediately take over T2 as an active task since it is not sufficiently caught up. In this case the first rebalance would produce an imbalanced assignment where **I2** bears the major processing load while **I3** focuses on restoring the task (T4) it will eventually be assigned as active.

Related Work

Note that the main concern of this KIP is how to allocate and re-allocate sharded stateful tasks, of which the state itself is the difficult part. Thus, although other stream processing systems are of prime interest, we can also look to the balancing algorithms employed by distributed databases, as long as those dbs follow the Primary/Replica model. This is advantageous both for the diversity of perspective it lends, but also because some of these database systems are more mature than any modern Stream Processing system.

One thing to note when considering other SP and DB systems is that, unlike most of them, Kafka Streams achieves durability via changelog topics. That is, in Streams, the purpose of a replica is *purely* a hot standby, and it's perfectly safe to run with no replicas at all. In contrast, most other systems use the replicas for durability, so they potentially need extra machinery to ensure that at all times a certain number of replicas is available, or active, or consistent.

As an example of the degrees of freedom that are unique to Streams, we would be perfectly safe to assign the active task to the most caught-up node and assign the standby task to an empty node and completely discard any other existing replicas. In any other distributed data system, this would result in a dangerous loss of durability.

Kafka Consumer StickyAssignor

The Consumer's StickyAssignor implementation is interesting. It has many of the same goals as Streams' assignor, although it only has to deal with one class of partitions. In contrast, Streams' assignor has to consider: (0) partitions that must be grouped together for tasks, (1) partitions for Stateful tasks, (2) partitions for Stateless tasks, and (3) partitions for Standby tasks. Nevertheless, we could consider generalizing the StickyAssignor algorithm for multiple classes of partitions, as well as the grouping constraint and the standby/active constraint.

The assignment algorithm begins by copying the prior assignment and then removing any assignments that have become invalid (consumer has left or partition no longer exists). Thus, we start with the valid sub-set of the prior assignment and a list of all the partitions that need to be assigned. Then we iterate over the unassigned partitions and assign each one to the consumer that *can* host it and has that has the smallest current assignment. This is a greedy assignment that should produce an assignment that is as balanced as possible while maintaining all current assignments. Then, we enter the balancing phase.

The balancing phase is an iterative algorithm. In each pass, it attempts to move each partition to a better place, and it continues with more passes until no more improvements are possible, or until the cluster as a whole is fully balanced. (Due to the assignment constraints, full balance may not be possible).

When considering the best move for a partition, it first checks to see if that partition is currently hosted on a consumer that is unbalanced with respect to the prior host of that partition. In this case, it just moves the partition back to the prior host. This is essentially a short-circuit for the case where a partition has become "unstuck" and restoring stickiness could actually improve balance. If we get past that short-circuit, then we just propose to move the partition to the consumer that can host it and has the smallest current assignment.

As mentioned, we keep "shuffling" all partitions in this way until we get an optimal balance, given the constraints.

reference: https://github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/clients/consumer/StickyAssignor.java

Cruise Control

Cruse Control is a LinkedIn project to automate some aspects of Kafka broker cluster administration. One thing it does is dynamically rebalance the partition assignment over the broker instances based on a large number of metrics it collects including CPU load, disk usage, etc. It structures the assignment optimization task as a Goal-Based Optimization problem. See https://en.wikipedia.org/wiki/Goal_programming for more information about this paradigm. Note that "goal programming" often implies that you represent the goals as a system of linear equations, and then solve the system to maximize some variables (aka Linear Programming), but that's not how Cruise Control is implemented. It just expresses the optimization problem as a system of goals and seeks a satisfactory (not optimal) solution by sequentially satisfying each goal.

The structure of the implementation is that you get a list of Goal implementations, each one corresponding to an optimization goal, like disk usage. The overall optimizer sorts the goals by priority, and then passes in the cluster assignment to the highest priority goal. That goal proposes modifications to the assignment (typically using an iterative algorithm similar to the StickyAssignor's). Once it's happy with the proposal, it returns. Then the optimizer passes in the new proposed assignment to the next goal, and so on. Each goal is responsible for ensuring that its proposals do not violate any of the prior, higher priority, goals. The API provides a hook that the goal can call during its optimization passes to ensure it doesn't violate the higher priority goals.

I don't think it would be very straightforward to turn Cruise Control into a general framework for allocating resources, because an awareness of the structure of the task is built in at every level (the optimization and the goals all know that they are dealing with Kafka brokers and partitions). But there are of course off-the-shelf optimization frameworks we could consider using if we want to go down a generalization path.

It would be straightforward to implement our allocation algorithm following a similar pattern, though. This might be a good choice if we want to add more optimization goals in the future. The main benefit of the goal-based orientation is that it scales naturally with adding more optimization goals (you just plug them in). As well, it's pretty easy to configure/reconfigure the optimizer to include or remove different goals.

source: https://github.com/linkedin/cruise-control

Redis Cluster

Redis is a high-query-performance database. The main use case for it is as a caching layer. Partly due to this fact, data durability isn't of tremendous importance, and the main operating mode is single-node. However, a clustered version was released in 2015. Reading the documentation, it sounds like the intent is more to provide a simple mechanism for transcending single-node mode than to provide a true distributed database with the kinds of guarantees one would expect (like consistency). Accordingly, the data distribution and primary/replica handling are quite simplistic. This is not meant to be disparaging. Redis Cluster is designed to serve specific use cases well at the expense of other use cases.

In Redis Cluster, keys are hashed into one of 16384 buckets (called slots). The hashing algorithm does not need to be consistent because the slots are fixed in number. Nodes in the cluster have unique and permanent IDs. Cluster integrity is maintained via a gossip protocol, and each node ultimately maintains a connection to every other node. Nodes don't proxy queries to non-local keys. Instead they respond with the hash slot that contains the key, along with the node that currently holds that hash slot (this is similar to Streams). Every node in the cluster is assigned some subset of the slots, and the slots can be manually assigned, unassigned, and moved to, from, and between nodes.

Redis Cluster doesn't have primary and replicas for each slot (the way that Streams has primary and standbys for each task). Rather, it has primary and replicas for each *node* . Thus, if there are two nodes, A and B with replicas Ar and Br, the node Ar is responsible for replicating the exact same set of slots that are assigned to A, and likewise with B and Br. This simplifies the assignment considerations somewhat, as a given node *only* needs to host active (primary) copies *or* passive (replica) copies. The replicas can serve queries if the query specifies that it accepts stale data, but most likely this arrangement results in the replica nodes being underutilized. So, my quick judgement is that the simplicity of this arrangement is enviable, but we probably don't want to follow suit.

reference: https://redis.io/topics/cluster-spec

Elasticsearch

An Elasticsearch cluster is a collection of *nodes*, an individual instance of Elasticsearch. At the cluster scope, there is a *master node* that is responsible for coordinating cluster changes such as adding or removing nodes, indices, etc. Each node has one or more *shards*, which correspond to a certain (Lucene) *index* that the cluster is fetching data from. An index is divided up into shards across one or more nodes, where the work for that index is distributed across the shards. Each shard has a primary shard responsible for writes, and one or more replica shards that can receive reads.

The rough translations to Streams are shown in the table below. Note that the comparisons are drawn as relates to load balancing, rather than literal definition (for example, an index is really more like a store. However for our purposes it is more useful to think of it as an entire subtopology, in that each index/subtopology is an independent job, that has some inherent "weight" – such as the number of stores for a subtopology – and its work is partitioned and distributed independently, into some number of shards/tasks. The analogies are fairly close, and Elasticsearch has to solve a load balancing problem similar to the one that Streams faces – one main high level difference to point out is that the replica shards are presumed in sync with the active shards, removing the complexity of "restore completeness" from their challenge.

Elasticsearch	Streams
index	subtopology
master node	group leader
node	instance
primary shard	active task
replica shard	standby task
shard	task

Elasticsearch actually breaks down the problem into two separate processes: allocation and rebalancing. Allocation refers to the assignment of (unallocated) shards to nodes, while rebalancing occurs separately and involves moving allocated shards around. By default, rebalancing can only occur when all shards are allocated (can be configured to be allowed only when active shards, or always). Multiple rebalances can take place concurrently, up to some configurable max (defaults to 2) – note that this limit applies only to "load balancing" rebalances and not those forced by environmental (user-defined) constraints. You can also dynamically disable/enable rebalancing either type of shard.

The allocation algorithm is as follows – note that this applies only to placement of unassigned shards, but nodes may have other shards already assigned to them.

1. Group shards by index, then sort by shard ID to get the order of shard allocation. First all primary shards are allocated, then one replica for each shard of each index, and repeat if number of replicas is greater than one.
2. For each shard, build a list of possible nodes. Nodes may be eliminated from consideration based on user config (eg allocation filtering) or various constraints (no copies of a shard should be allocated to the same node, adequate remaining disk space, forced awareness, max retries)
3. If step 2 returns no nodes, the shard will be retried later (possibly after a rebalance). Otherwise, we calculate the weight of each node if given the shard, and allocate it to the one with the lowest weight. The weighting function depends on two settings: indexBalance (0.55 by default) and shardBalance (0.45 by default). The total weight is the weighted average of the shard and index weights, weighted by the fractional shard and index balance respectively. This is computed as

```
private float weight(Balancer balancer, ModelNode node, String index, int numAdditionalShards) {
    final float weightShard = node.numShards() + numAdditionalShards - balancer.avgShardsPerNode();
    final float weightIndex = node.numShards(index) + numAdditionalShards - balancer.avgShardsPerNode(index);
    return theta0 * weightShard + theta1 * weightIndex;
}
```

where

```
theta0 = shardBalance / (indexBalance + shardBalance);
theta1 = indexBalance / (indexBalance + shardBalance);
```

The shardBalance and indexBalance can be thought of as controlling the tendency to equalize the number of shards (shardBalance) or number of shards per index (indexBalance) across nodes. This is analogous to Streams assigning a balanced number of tasks to each StreamThread for the former, and spreading the tasks of a given subtopology across StreamThreads for the latter. The weighting formula used by Elasticsearch indicates there is some tradeoff between the two, but that applies only to Elasticsearch's allocation, where some shards have already been allocated to some subset of nodes. For example, if you add a new node and a new index, you could allocate all the new shards to the empty node for shard number balance, or give each node a shard for shard index balance, but likely not both.

If you start with a "clean assignment" where all shards must be (re)allocated, both can be balanced at once. You can simply group the shards by index, then loop through the nodes assigning a shard from the list in round-robin fashion. This would be the case when first starting up in Streams. However, the more "sticky" the assignment in Streams the better, so we tend to avoid this kind of simple round-robin assignment on subsequent rebalances. So the tradeoff in Streams is between balance (however that is defined) and stickiness – whereas in Elasticsearch, the tradeoff is first between types of balance where stickiness is effectively a constraint (of allocation), and the stickiness is then (potentially) traded for further overall balance by the rebalancing operation. One interesting thing to note is that, while Streams effectively enforces "total balance" (defined as number of tasks across threads), this is actually configurable in Elasticsearch. You can tune the cluster to be more or less aggressive about balancing the shards across nodes. This might be useful if users just want to get up and running again as fast as possible, and would be satisfied with a slightly imbalance workload.

Of course, the workload in Streams is already potentially unbalanced because our definition of balance is simple and does not account for degree of statefulness. But maybe assigning weights that account for "stickiness" – how much would this instance have to catch up? – and "load" – some measure of the number of stores – would be useful for distributing tasks.

YARN

“Yet Another Resource Negotiator” is a cluster management technology introduced in Hadoop 2.0 that decouples resource management from job scheduling. The YARN framework consists of a global master daemon called the *ResourceManager*, a per-application *ApplicationMaster*, and per-node *NodeManagers*. NodeManagers are responsible for allocating containers and running/managing processes within them, and monitoring/reporting their resource usage to the ResourceManager, who in turn is responsible for keeping track of live NodeManagers and arbitrating resources among competing applications. The ApplicationMaster manages the application life cycle, negotiating resources from the ResourceManager and triggering NodeManagers to begin executing tasks for that application.

The ResourceManager itself has two main components, the *Scheduler* and the *ApplicationsManager*. The ApplicationManager accepts jobs and is responsible for managing the application's ApplicationMaster (including starting & restarting it if necessary). The Scheduler is then free to allocate resources to applications without concerning itself with monitoring applications or restarting failed tasks. The actual scheduling policy can be plugged in to partition the cluster resources.

This scheduling however is fairly orthogonal to the balancing/restoring problem we face, since they involve the question of how to distribute new/freed resources to existing jobs rather than how to distribute all existing "jobs" (tasks) to all available "resources" (instances/threads). The interesting difference here is that in Streams, a new resource (fresh instance) necessitates revoking tasks from another instance, whereas in YARN a new resource can simply be given away. Maybe the right way to look at it is to consider the YARN resources as Stream tasks, and YARN jobs as Streams instances (or threads, just using instance for now) – one resource/task can only ever run one job/be run on one instance, while new instances/jobs are started and stopped, and require some (re)allocation of resources/tasks.

Flink

Flink jobs are composed of *operators,* a chain of which forms a *task,* which can be executed on threads as a one or more *subtasks*. The total number of subtasks is the parallelism of that operator. A Flink task corresponds to a Streams subtopology, and a Flink subtask corresponds to a Stream task. In the following section, task will refer to a Flink task and any references to a Streams task will be labelled as such.

The Flink runtime consists of two processes, the *JobManager* which schedules tasks and coordinates checkpoints, failure recovery, etc. and one or more *TaskManagers* which execute the subtasks and buffer/exchange streams of data. TaskManagers, similar to a Streams app instance, are brought up and connect to the JobManager to announce themselves as available, then are assigned work. Each TaskManager has one or more *task slots* corresponding to a fixed subset of the TaskManager's resources – they share network connections similar to how Streams tasks share a StreamThread consumer.

Also similarly to StreamThreads executing one or more Streams tasks, Flink allows subtasks of the same job to share task slots (note that a single task slot may use one or more threads, though Flink does recommend matching the number of task slots to the number of CPU cores). This can be controlled to a hard or soft degree by the user defined CoLocationGroup (which subtasks **must** share a slot) and SlotSharingGroup (which subtasks **can** share a slot).

Flink integrates with other cluster resource managers including YARN, but can also be run as a stand-alone cluster. Unlike YARN, in general Flink jobs are continuous (like Streams or Elasticsearch) so the "load balancing" aspect is more important than the "scheduling/distributing transient resources" aspect. This brings it closer to Streams, but unlike Streams, Flink has no notion of standby tasks – instead, for high availability some distributed storage is required for checkpoints to be saved/recovered from. This greatly simplifies the assignment logic relative to the Streams/KIP-441 case.

Samza

Out of all the other systems we've looked at Samza is the closest in comparison to Kafka Streams.

Tasks and Containers

Like Streams, the unit of parallelism is a task. Samza assigns tasks to a partition from an input stream. Tasks in Samza also perform periodic checkpointing. The checkpointing allows the resuming of processing from the latest offset on a different worker in the case of failures. Tasks themselves are hosted in a Container, which is the physical unit of work, compared with tasks which are a logical unit of work. A Container will have one or more tasks and are distributed across different hosts.

Coordinator

To handle the distribution of tasks to containers, Samza uses a Coordinator. The Coordinator also monitors the containers, and when a failed container is detected, the tasks of the failed container are distributed out to the remaining healthy containers. The Coordinator is pluggable, giving Samza the unique ability to run either in standalone/embedded mode or cluster mode with a cluster-manager such a YARN.

State

Samza also offers stateful operation and uses RocksDB for the implementation of the persistent storage. Each task has its own state-store. Storage engines are pluggable as well. For fault tolerance, state-stores use changelog (compacted) topics as well.

Host Affinity

To deal with possible long startup times necessary when reading an entire changelog, Samza persists metadata containing the host that each task is running on. When users elect to enable the job .host-affinity.enabled configuration, Samza will attempt to place a container for a given task(s) on the same host every time a job is deployed. The host affinity feature is done on a best-effort basis, and in some cases, a container will get assigned to another available resource.

Given the closeness of Samzas architecture to Kafka Streams, IMHO it makes sense to take a quick look into the algorithm used by the Coordinator concerning load distribution and stateful tasks.

Heron

Twitter developed Heron in 2011 to overcome the shortcomings of Apache Storm attempting to handle the production load of data.
Specifically, the items Twitter needed to address are

1. Resource isolation
2. Resource efficiency
3. Throughput

Heron claims to be compatible with Storm, but there are a few steps developers need to take https://apache.github.io/incubator-heron/docs/migrate-storm-to-heron/, so it's not entirely backward compatible out of the box.

Heron API

Heron is similar to Kafka Streams in that it offers a high-level DSL (https://apache.github.io/incubator-heron/docs/concepts/streamlet-api/) and a lower level API (https://apache.github.io/incubator-heron/docs/concepts/topologies/#the-topology-api) based on the original Storm API.

Topology

Heron topologies are composed of two main components Spouts and Bolts.

Spouts

Spouts inject data into Heron from external sources Kafka, Pulsar. So a spout is analogous to an input topic in Kafka Streams.

Bolts

Bolts apply the processing logic defined by developers on the data supplied by Spouts.

Streams of data connect spouts and Bolts.

Architecture

Heron needs to run on a cluster of machines. Once developers build a topology, they submit the topology to cluster, much like Spark, Flink in that you develop the application then distributed to the worker nodes

Resource Management

When using the high-level Streamlet API Heron allows you to specify resources for the topology:

1. The number of containers into which the topology's physical plan divided into.
2. The total number of CPUs allocated to be used by the topology
3. The total amount of RAM allocated to be used by the topology

Operations

Heron offers stateless and stateful operations. There are windowing operations similar to Streams Sliding, Tumbling, and Time based windows. Heron also offers the concept of "counting" windows.

State Management

Heron uses either Zookeeper (https://apache.github.io/incubator-heron/docs/operators/deployment/statemanagers/zookeeper/) or the local file system (https://apache.github.io/incubator-heron/docs/operators/deployment/statemanagers/localfs/) for managing state. Details about state management are relatively sparse, but IMHO, I think the approach to state management is far enough from Kafka Streams that there is nothing we can gain Heron's state management.

The overall recommendation from the Heron documentation is that the local file system state management is for local development only, and Zookeeper is the preferred approach in a production environment.

Compatibility, Deprecation, and Migration Plan

We will implement the new algorithm as a new PartitionAssignor implementation without removing the existing StreamsPartitionAssignor. Thus, the existing upgrade/downgrade procedure makes moving in both directions seamless. As long as any member does not support the new assignment protocol, all members will use the existing assignment strategy. As soon as all members indicate they support the new strategy, the group will automatically encode the SubscriptionInfo in the new format, and the leader will automatically start using the new PartitionAssignor, including the new algorithm and the probing rebalances.

Thus, the upgrade should be seamless.

Ultimately, we will want to remove the old StreamsPartitionAssignor. For example, we may remove it in version 3.0. In that case, we will document that operators must upgrade the whole cluster to version [2.4, 3.0) first, before upgrading to version 3.0+

Rejected Alternatives

Triggering a probing rebalance when Moving tasks reach the acceptable lag

One idea was, instead of triggering a probing rebalance on an interval, to trigger it when Moving tasks actually catch up. One advantage of this is that you don't get rebalances when none of the Moving tasks have caught up yet. In the interval-based approach, you are guaranteed not to trigger probing rebalances more frequently than the interval, because there is just one node (the leader) that triggers the rebalance. On the other hand, if you want to trigger when members catch up, the members themselves have to do the triggering. Since they are all independent, they will cause "storms" or rebalances when first one member catches up, and the cluster rebalances, then the next catches up, and we rebalance again, and so forth.

Ideally, the members would have a way to inform the leader that they have caught up, and the leader could then make a global decision about whether to rebalance now, or wait a bit. Unfortunately, there is currently no channel for this communication. The only one is for a member to join the group again, which causes a full rebalance; there's no way to avoid it. Instead, we've proposed the interval approach for now, to ensure that the frequency of rebalances can be controlled, and we plan to add this informational channel in the future.

Adding a new kind of task ("moving", "recovering", "learner") for task movements

During the KIP discussion, an idea came up, instead of assigning standbys to accomplish task movements, to create a new kind of task with a name that doesn't suggest the idea of a "hot standby". One rationale is that it might be confusing as a user to configure `num.standby.replicas:=0`, but then still see standbys in the logs, etc. Another thought is that it seems more straightforward to have two different names for what seems like two different roles.

However, it turns out to cause more conflicts that it solves. To start off, here's a "happy path" scenario:

Happy path scenarios

Looking at a cluster with two nodes and two tasks, with **no** standbys, immediately after having lost and replaced a node, the KIP would assign active tasks like this:

```
num.standby.replicas:0

Node1: [Task1(active), Task2(active)]
Node2: []

(no standbys reported in logs, etc.)
```

To achieve balance, we need to move one of the tasks over, so we'd create a "moving" task for it to catch up:

```
num.standby.replicas:0

Node1: [Task1(active), Task2(active)]
Node2: [Task2(moving)]

(no standbys reported in logs, etc.)
```

Trouble

However, what should we do if we *did* have standbys configured?

Looking at a cluster with two nodes and two tasks, with **no** standbys, immediately after having lost and replaced a node, the KIP would assign active tasks like this:

```
num.standby.replicas:1

Node1: [Task1(active), Task2(active)]
Node2: [Task1(standby, not caught up), Task2(standby, not caught up)]

(standbys ARE reported in logs, etc.)
```

To achieve balance, we need to move one of the tasks over, but now we cannot create our "moving" task because Node2 already has a standby task for Task2, and it can't host two different "types" of the same task. We have several choices, and none of them are satisfying.

- We could make Task2 a "moving" task *instead* of a "standby" task. This is true to the idea of adding a "moving" type, because it tells us that Task2 is intended to become active after it catches up. But the bummer is that we would *not* report it as a standby task. So operators would be confused and probably concerned to see that Task2 has no standby task in this scenario.
- We could somehow mark the task as both "moving" and "standby", so that it's a "moving" task for the purpose of assignment, but it's also reported as a "standby" task in the logs, etc. This results in increases the code complexity, since we have to handle the task properly all over Streams, whenever we log it or store it in a collection, to decide if we're going to treat it as a standby or a moving task.

In retrospect, if we had called these standbys "replicas", it would have helped the situation. Then, it would seem more ok to have an extra replica for moving a shard *sometimes* , and you'd always expect to see at least "num.standby.replicas" number of replicas in your cluster. But it's a little hard at this point to consider completely renaming "StandbyTask" to "ReplicaTask", etc.