KIP-925: Rack aware task assignment in Kafka Streams

1. Status

Current state: *Accepted*

Discussion thread: here

JIRA: here

2. Motivation

Kafka `TopicPartitions` are replicated on the broker side to different racks. KIP-881: Rack-aware Partition Assignment for Kafka Consumers added rack information of clients in partition assignment interface so that the assignment logic can utilize this information to try to assign `TopicPartitions` to the clients in the same rack as much as possible so that it would avoid cross rack traffic (CRT) which has higher costs in terms of latency and money.

The client rack awareness additions in partition assignments also affects how Kafka Streams can assign its tasks to clients so that cross rack traffic can be avoided as much as possible. This design discusses how we can augment current assignment logic in Kafka Streams to utilize rack information to minimize cross rack traffic with other constraints. Note the design doesn't change the rebalance protocol itself and only changes the assignment logic (read path in group leader in clients).

3. Current Task Assignment Logic

There are two assignors implemented in Kafka Streams now: StickyTaskAssignor and HighAvailabilityTaskAssignor. `HighAvailabilityTaskAssignor` was created for KIP-441: Smooth Scaling Out for Kafka Streams and it’s the currently default assignor configured in Kafka Streams. `StickyTaskAssigno` can also be used by existing customer. We could put rack aware assignment logic in a separate module which be used by both `StickyTaskAssignor` and `HighAvailabilityTaskAssignor` . Below we mainly discuss how it works with `HighAvailabilityTaskAssignor. It works similar for StickyTaskAssignor.`

How HighAvailabilityTaskAssignor (HAAssignor) works

1. Assign active tasks to clients in a round-robin manner. Tasks and clients are both sorted when this assignment is performed so that this assignment is deterministic
2. Balance active tasks among clients so that the task load in clients is balanced. (taskLoad = (activeTaskCount + standbyTaskCount) / threadsNum)
3. Assign standby tasks to clients using `ClientTagAwareStandbyTaskAssignor` or `DefaultStandbyTaskAssignor`
4. Balance standby tasks among clients based on task load
5. Move active tasks around and create warm-up tasks if there are more caught-up clients (If a client previously has this task’s standby or active)
6. Move standby tasks around if there are more caught-up clients
7. Assign stateless tasks

8. Return current assignment and if there are tasks moved around in step 5 or 6, schedule a probing rebalance after some timeout hoping the stable assignment in step 1 and 3 can be achieved later

The main advantage of `HAAsssignor` is that it takes caught-up information on clients into account and will try to assign tasks to more caught-up clients so that those tasks can be run first. At the same time, the original assigned client will try to warm up its state and try to take over the task in following probing rebalances.

4. Design for rack aware assignment

There are several constraints for rack aware assignment:

1. Assignment should still balance tasks over clients based on task load even if this can cause higher cross rack traffic
2. Rack aware assignment should be computed in step 1 - 2 for active tasks and step 3 - 4 for standby tasks before we consider the caught-up information on clients to move tasks around in step 5 - 6. The reason is that the assignment computed in step 5 - 6 is temporary for task warmup. We should aim for minimizing task downtime over reduce CRT for this temporary assignment.
3. The computed assignment must converge which means that subsequent probing rebalancing can stop eventually if warm-up tasks are warmed up

Now suppose we have a set of tasks T = {t1, t2 … tn} and a set of clients C = {c1, c2 … cm}. We define the cost of assigning task ti to client cj as following:

```
int getCost(Task t, Client c) {
    int cost = 0;
    for TopicPartion tp : t.topicPartitions() {
        if (tp has no replica in same rack as c) {
            cost++;
        }
    }
}
```

Then our problem becomes to find out a mapping M: T -> C so that Cost[i][j] is minimized (for i in T and j in C) with the constraint that the mapping M should be as balanced as possible. For standby tasks, there should be other constraints such as a standby task shouldn't be assigned to a client which has the same active task.

An ideal solution would be that for active, standby and stateless tasks, we can find an assignment for them which minimizes the total cost while satisfying all the constraints. However, I couldn't find an efficient polynomial-time global optimization algorithm to do this. As a result, I propose the local optimizations that we still follow the structure of HAAssignor: first, we compute min cost assignment for active tasks. Then we compute it for standby tasks and lastly we compute assignments for stateless tasks.

A. Min-cost flow problem

The min-cost flow problem is defined as follows: given a directed graph, each edge has a capacity constraint which limits how much traffic it allows. Each edge also has a cost for each traffic unit. Each node could have a supply or demand of certain traffic. The problem is to find a max flow of traffic in which the total cost along the edges can be minimized.

Our assignment problem can be translated to a min-cost max flow problem as below.

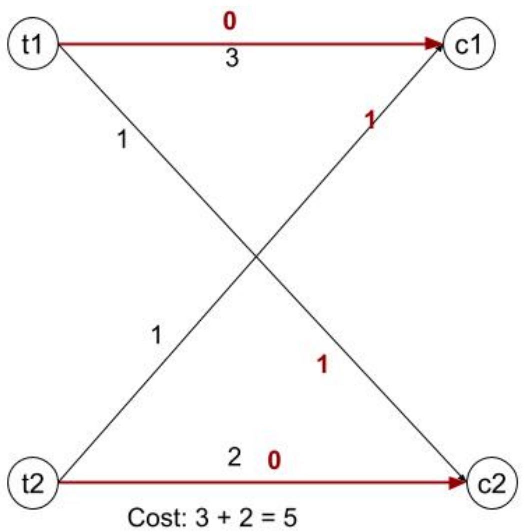
Tasks and Clients are graph nodes. Every task is connected to every client by an edge with capacity 1. The cost of an edge is the cost of cross rack traffic between the task and the client. If a task is assigned to a client, then there's one flow between them and vice versa. Then our assignment problem is translated to find the min-cost max flow in our graph with the constraint that the flow should be balanced between clients.

I. Algorithm

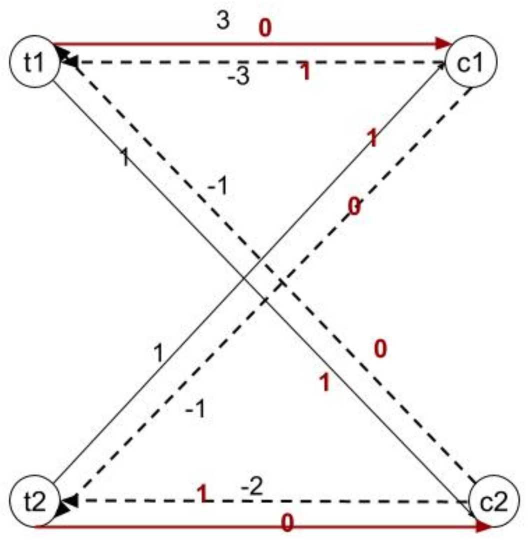
Klein's cycle canceling algorithm can solve the min-cost flow problem in O(E^2 * M * U) time where M is max cost and U is max capacity. In our particular case, U is 1 since a task can be assigned once to one client. M is related to the number of topic partitions a task can subscribe. So the algorithm runs in O (ME^2) time for our case. Note E is `|T| * |C|` where T are the tasks and C are the clients. This is because each task can be assigned to any client; so there is an edge because every task to every client. In another word, the time complexity of the algorithm is `O(M * |T|^2 * |N|^2)`. Below is a rough sketch of the algorithm:

```
find a feasible flow

while there's negative cylces in the residual graph:

    find a negative cycle

    update the flow with negative cycle's min capacity
```

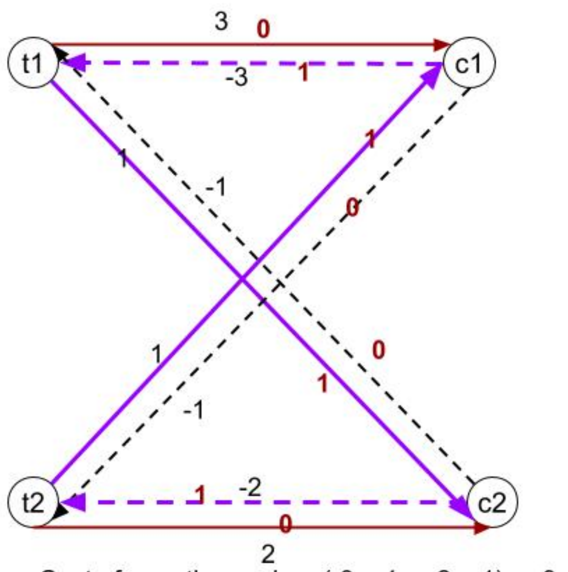
Negative cycles and residual graphs can be read here. Negative cycles can be found using Bellman-ford's algorithm. How to find a feasible flow? It can be found by using step `1 - 2` of `HAAssignor` . What's even nice about the cycle canceling algorithm is that since our graph is a bipartite graph, every time we find a negative cycle, the client nodes in the cycle satisfies that if there's an in edge, there must be an out edge which means the flow of the clients won't be changed. As a result, each round of the cycle canceling will reduce the cost and won't change the total assignments of a client. So eventually, when there's no more negative cycles, we find a min-cost balanced assignment. Below is an example of finding an min cost of 2 to assign task `t1, t2` to clients `c1, c2` .



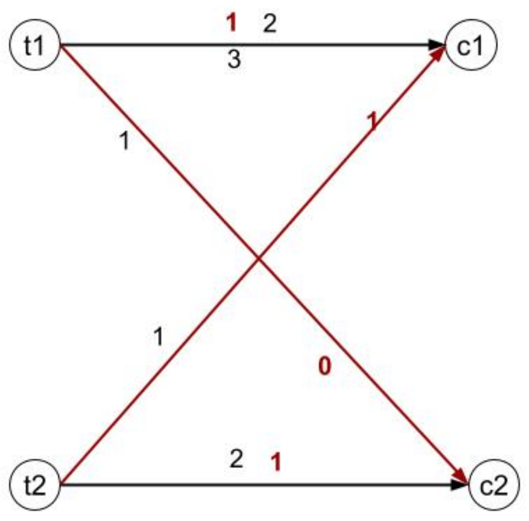
Cost: 3 + 2 = 5



Residual graph



Cost of negative cycle = (-3 + 1 + -2 + 1) = -3
Flow of cycle = 1



Cost: 1 + 1 = 2

B. Rack awareness assignment algorithm for active stateful tasks

I. Min cost with unbalanced sub-topology

```
1. Compute cost cross rack cost of each task/client pair
2. Find an assignment following step 1 - 2 of HAAssignor
Convert assignment to residual graph: if there's assignment, change capacity to 0 and flow to 1, if there's no
assignment, change flow to 0 and capacity to 1. Edge cost is task/client cost
3. while there's negative cycle in current graph:
       find negative cycle, update flow
4. Get assignment from final flow. If there's an flow in the edge, the two nodes of the edge which are task and
client have assignment relationship.
```

As analyzed, the result assignment will make each client have the same load as what's computed by step 1 - 2 from HAAssignor. So the assignment is balanced with minimal cost. Also the computation is deterministic as all the steps in this algorithm are deterministic.

One caveat is that while the load on each client is similar, this algorithm doesn't guarantee that different partitions of different sub-topology will be assigned to different clients as much as possible.

II. Min cost with balanced sub-topology

Above algorithm doesn't balance tasks for same sub-topology which may lead to processing skewness. We could try to balance it by constructing a more complex graph by limiting how many tasks of a sub-topology we can assign to a client.

```
1. follow HAAssignor's step 1 - 2 to get a balance assignment for each client.
2. Suppose all the clients have n tasks in total, client i have C[i] tasks assigned which is balanced
assignment, subtopology j have S[j] tasks (partitions). Now we can limit the number of tasks subtopology j can
assign to client i to ceiling(S[j] * C[i] / n).
3. With above limitation, we construct a min-cost flow graph and compute the min-cost with balanced sub-
topology assignment as well.
```

Graph construction

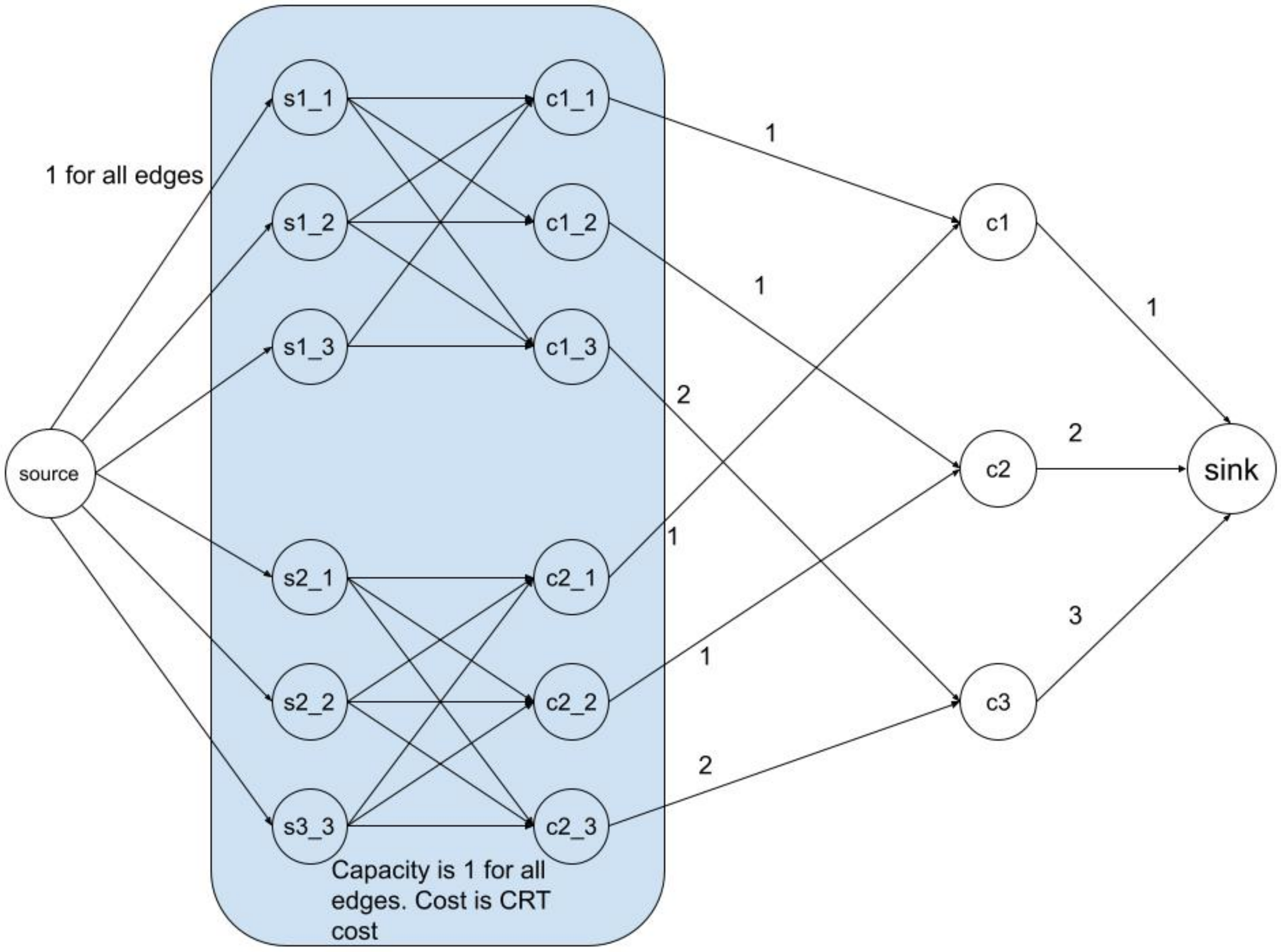
1. For each sub-topology, do this:
    a. Create new set of nodes which has same number as clients
    b. Add an edge between task node and each new client nodes with capacity 1 and cost as CRT cost
    c. Add an edge between each new client node and corresponding original client nodes with capacity as the limit of tasks this sub-topology can assign to the client. The cost of the edge is 0
2. Add a sink node connecting all client nodes with capacity as what we computed in HAAssignor's step 1 - 2 and cost 0

Solution

1. Given the above graph, we could use the max-flow algorithm to find a solution first (Why must a solution exist? Intuitively, we if we need to assign C[i] tasks to Client i and there are totally N tasks, for sub-topology [j], we need to assign S[j] * C[i] / N. If we take the ceiling of this and denote this as the max this sub-topology can donate to this client, within this bound, we can find a solution. But how to prove this mathematically?)
2. Then we apply cycle canceling algorithm to minimize the cost
3. When the algorithm finishes, we have a solution which minimizes the CRT cost while limits the number of tasks each sub-topology can assign to each client (Balance tasks of same sub-topology)

Example

Suppose there are 3 clients C1, C2 and C3 with 1, 2 and 3 threads respectively. There are 6 tasks total with 2 sub-topologies S1 and S2. So each sub-topology has 3 tasks. It's easy to see C1, C2 and C3 should have 1, 2 and 3 tasks assigned to each respectively since this will make them have equal load of 1. Then the limit of tasks within a sub-topology for each client would be ceiling of 3*, 3*2/6 and 3*3/6 which are 1, 1, and 2. Then our graph can be constructed as



Based on the cost information, one possible solution could be {s1_1}, {s1_2, s2_1}, {s1_3, s2_2, s2_3}. Note this solution could satisfy all capacity requirements and if this isn't the optimal solution, cycle canceling would find us one which would also satisfy the capacity requirements which balances sub-topologies.

One drawback of this solution is that the final solution is based on the number of tasks which can be assigned to each client computed from HAAssignor's step 1 - 2. If every client has an equal number of threads but the number of tasks couldn't be divided equally, then there's one client with a different number of tasks. Ideally, this client could be anyone when we find our min cost assignment, but we fix it during our computation, which may result in a slight sub-optimal solution.

The runtime complexity of this algorithm is also higher since there are more edges in the graph. Suppose we have `S` sub-topologies, then we have `(|T| + |T|*|C| + S*|C| + |C|)` edges, which is still bounded by `O(|T|*|C|)` edges. So the time complexity is still `O(M * T^2 * N^2)`.

Since 4.B.I and 4.B.II have different tradeoffs and work better in different workloads etc, we could add a config to choose one of them at runtime.

III. How to make the assignment as stable as possible if there are small changes in the CRT cost

While the min-cost flow algorithm can find us assignments with min CRT cost, it's also possible that if a `TopicPartition` 's rack changes, the newly computed assignment could drastically change which can cause task shuffling and new warmups which involves more network and computation cost. So it's preferable to make the new assignment as close to the old assignment as possible.

While it's tempting to factor in previous assignment to make the new task assignment sticky, it poses some challenges to store assignment information across rebalances and make sure the assignment can converge. Alternatively, we can choose to overlap more with target assignment computed by HAAssignor step 1 - 2. The idea is that if we always try to make it overlap as much with HAAssignor's target assignment, at least there's a higher chance that tasks won't be shuffled a lot if the clients remain the same across rebalances even if some traffic cost changes. In this way, we can maintain a stateless assignment but at the same time try to overlap with some fixed assignment as much as possible. Below is the adjusted cost function suggested by John Roesler

```
int getCost(Task t, Client c) {
    final int TRAFFIC_COST = 10;
    final int NON_OVERLAP_COST = 1;
    int cost = 0;

    // add weights for cross-AZ traffic
    for TopicPartion tp : t.topicPartitions() {
        if (tp has no replica in same rack as c) {
            cost += TRAFFIC_COST;
        }
    }

    // add weights for overlap with current assignment
    if (c is NOT currently target assignment) {
        cost += NON_OVERLAP_COST;
    }
}
```

`TRAFFIC_WEIGHT` and `OVERLAP_WEIGHT` indicate how much we favor minimizing traffic cost compared to maximizing overlap. In the extreme case, we can put `OVERLAP_WEIGHT` to 0 which means we always compute the lowest CRT assignment. We can expose these weights as internal configs. Note that we don't want to put a very high value for the weight because the cost upper bound appears in the algorithm's time complexity `O(E^2 * (MU))` .

Summary

In summary, for active task assignment, we can add public configs such as rack.aware.assignment.strategy with value min_cost or balanced_min_cost to choose algorithm I or algorithm II. We can make algorithm I as default to minimize total CRT cost. For cost computation, we can use Option 2 in III and expose configs to adjust the weight such as rack.aware.assignment.traffic_cost and rack.aware.assignment.non_overlap_cost . For `StickyTaskAssigno`
, we could run the same algorithm follow the active task assignment for stateful active tasks by adjusting the `traffic_cost` and `non_overlap_cost` t
o balance stickiness and traffic cost. Note that for `StickyTaskAssignor` , we would increase `NON_OVERLAP_COST` if t is not assigned to c in previous assignment.

C. Rack awareness assignment for standby tasks

Assigning standby tasks comes with even more constraints or preference:

1. Standby task shouldn't be assigned to client which already have the active task
2. Ideally, standby tasks should be in different rack from corresponding active tasks
3. Ideally, standby tasks themselves should be in different racks
4. Ideally, cross rack traffic cost should be minimized
5. Assignment should be deterministic

While it's possible to come up very complex min-cost flow algorithms for standby assignment, I think it's OK to use an easy greedy algorithm for standby assignment initially:

I. Balance reliability over cost

In this case, we should first optimize constraints 1 - 3. After we get an assignment from ClientTagAwareStandbyTaskAssignor for example, we use two loops to check if we could swap any standby task assignment between different clients which can result in smaller traffic cost. This is a greedy algorithm which may not give the optimal solution.

```
for (ClientState client : ClientStates) {
    for (Task standby task : client.standbyTasks) {
        for (ClientState otherClient : ClientState) {
            if (client.equals(otherClient) || otherClient.contains(task)) {
                continue;
            }
            for (Task otherStandbyTask : otherClient.standbyTask) {
                if (swap task and otherStandbyTask is feasible and have smaller cost) {
                    swap(standbyTask, otherStandbyTask);
                }
            }
      }
}
```

D. What if both rack information and rack aware tags are present

KIP-708: Rack aware StandbyTask assignment for Kafka Streams added rack awareness configurations for Kafka Streams with `rack.aware.assignment.tags` and `client.tag`. The goal is to use the tags to identify locations of clients and try to put tasks in clients of different locations as much as possible. This configuration could possibly be a superset of clients' rack configuration in consumer which is `client.rack`. There are several options on how to assign standby tasks when both configurations are present.

I. Choose a default one

If both rack aware tags and client rack information in subscription are present, we could default to use one for standby assignment. Both can be used to compute if two clients are in the same rack. Rack aware tags may have more information since it can support more keys. So we default to rack aware tags.

II. Enforce client.rack information must be present in rack.aware.assignment.tags

For example, if a client configures consumer rack information as `client.rack: az1`. Then rack aware tags must contain `rack.aware.assignment.tags: rack` and `client.tag.rack: az1`. We can fail the assignment if above is not satisfied.

III. Check client.rack information must be present in rack.aware.assignment.tags but don't enforce it

Option II is too strict. We could just check and warn the users in logs if both `client.rack` and `rack.aware.assignment.tags` are configured but rac doesn't exist in `rack.aware.assignment.tags` or `client.tag.rack` doesn't have the same value as `client.rack`. However, we continue processing the assignment using option I by defaulting to one of the configurations.

IV. Summary

I think option III is good enough for us and we can default to `rack.aware.assignment.tags` configuration since it can contain more information than `client.rack`.

E. Assignment for stateless tasks

We can use a min-cost flow algorithm to assign stateless tasks. First, we can find a feasible solution based on task load. Then we can use the cycle canceling algorithm to compute minimum cost assignment as we do for active tasks. We can reuse the configuration for active tasks to control if we merely compute the min-cost assignment or we also balance tasks for the same sub-topology.

5. Public Interfaces

There will be a few new public configs added:

- **rack.aware.assignment.strategy**: this config controls if we should use 4.B.I or 4.B.II to compute min-cost with or without balanced sub-topology. If set to `NONE`, rack aware assignment will be disabled
- **rack.aware.assignment.traffic_cost**: this config controls the cost we add to cross rack traffic
- **rack.aware.assignment.non_overlap_cost**: this config controls the cost we add to non overlap assignment with targeted assignment computed by HAAssignor

Note `client.rack` consumer config needs to be configured by customer to enable rack aware assignment.

6. Compatibility, Deprecation, and Migration Plan

We will implement the new rack aware assignment logic in `HighAvailabilityTaskAssignor` and `StickTaskAssignor. Rack aware assignment will be` used only when

- client rack information appears in Subscriptions by configuring `client.rack` on client side. If any client doesn't have rack information configured, a warning will be given and rack aware assignment will be disabled.
- at least one of the TopicPartitions in some tasks have different cost compared to other TopicPartitions. This is because if all of them have the same cost, there's no point to use rack aware assignment logic to compute min cost since assigning tasks to any clients doesn't make a difference.
- **rack.aware.assignment.enabled** config is enabled

If users want to use rack aware assignment, they need to upgrade Kafka Streams to at least the earliest supported version. It's better to stop all the instances and start them all with the latest version with `client.rack` config, but it's also OK to do a rolling bounce with the new version. Rolling bounce with new version may cause the assignment changing between rack aware assignment and no rack aware assignment but eventually the assignment will be computed using rack awareness after every instance uses the new version.

7. Test Plan

- Unit test will be added for min-cost flow graph algorithm and rack aware assignment
- Existing integration test will be updated to ensure it works with or without rack configuration as well as with both `client.rack` and `rack.aware.assignment.tags` configurations
- New integration test will be added to verify rack aware assignment
- Existing system test will be used to verify compatibility with old version

- `TaskAssignorConvergenceTest` will be updated to verify the new assignor will converge and it produces no worse assignment in terms of cross AZ cost
- Performance test will be done manually to verify the efficiency of min-cost flow computation with large number of tasks and clients as well as different number of subscribed topic partitions

8. Rejected Alternatives

A. The follow algorithm for standby assignment is rejected

Prefer reliability and then find optimal cost

We could also first find a more reliable standby assignment and then if we have a certain number of reliable standby assignments. We switch to prefer to minimize cost. For example, if we would like at least 1 standby to be in a different rack from the active, when we assign the first standby, we can make clients which are in a different rack from the active client have less cost. We can adjust the cost function as below:

```
int getCost(Task t, Client c) {
    final int TRAFFIC_WEIGHT = 1;
    final int SAME_RACK_WEIGHT = 10;
    int cost = 0;

    // add weights for cross-AZ traffic
    for TopicPartion tp : t.topicPartitions() {
        if (tp has no replica in same rack as c) {
            cost += TRAFFIC_WEIGHT;
        }
    }

    // add weights for assignment with different racks
    for (Client c1 which has t assigned) {
        if (c is in same rack as other clients which has t) {
            cost += SAME_RACK_WEIGHT;
        }
    }
}
```

After we assign the first standby to different racks as much as possible, we adjust the cost function to give more weight for `TRAFFIC_WEIGHT` so that we prefer low traffic assignments. This option is rejected because it's too complex. We need to run min-cost flow several times and it's not clear we could get a valid assignment solution each time.