ADDING PRIORITY TO RESERVATION SYSTEM

Carlo Curino, Sean Po, Subramaniam Venkatraman Krishnan, Vin Wang

MOTIVATION

YARN-1051 introduced reservation mechanism for jobs. Currently, Yarn RM does not keep track of priority in the reservation system. This means that when new reservations are submitted to a queue that is full during the reservation time window, it will always be rejected. This behavior is not ideal for reservations that are late-arriving, but high priority. If priority is added to the reservation system, Yarn RM can reason about reservation preemption instead of always rejecting new reservations.

PROPOSED APPROACH

We propose to support priority in three stages:

1. Add an extra field in the ReservationSubmission API to support priority.
2. Add a configurable component called the PriorityAgent will be added.
    a. At a high level, the PriorityAgent orchestrates the removal and addition of reservations tracked by the plan.
3. Add a priority-aware re-planner to ensure that reservations are removed in priority order when the re-planner is kicked.

Adding priority to the reservation system will have no effect on the failover mechanism added by YARN-4985.

IMPLEMENTATION DETAILS

Reservation priority needs to be incorporated in all stages of the reservation system.

1. Any user which wants to submit reservations with priority will add an extra field to their ReservationDefinition. If the priority is not provided, a configurable default will be used.
2. The PriorityAgent will only take effect if the ReservationAgent cannot make a reservation.
3. In this case, the PriorityAgent will remove appropriate reservations in order to make room for the incoming reservation. The following describe the steps taken specifically for the SimplePriorityAgent.
4. The SimplePriorityAgent will remove all reservations that is lower priority than the incoming reservation.
5. These reservations will then be ordered by decreasing priority and increasing arrival time.
6. The removed reservations will then be re-added into the plan by invoking the ReservationAgent.
7. In addition to the SimplePriorityAgent, a priority aware replanner also needs to be implemented.
8. The SimplePriorityAwareReplanner will account for a cluster capacity change by removing reservations in increasing priority, and decreasing arrival time order.

FUTURE WORK

1. Removing all reservations that are lower priority than an incoming reservation can be very expensive if the incoming reservation is very high priority.

   Performance can be improved by pruning the list of reservations to be deleted and then re-added. Instead of taking all reservations that are lower priority than the incoming reservation, we can start off by looking for reservations that are lower priority in a certain time range. If we simply remove all reservations strictly overlapping the reservation time window of the incoming reservation, there may be a scenario where a higher priority reservation gets deleted in place of a lower priority one. Take the following table as an example.

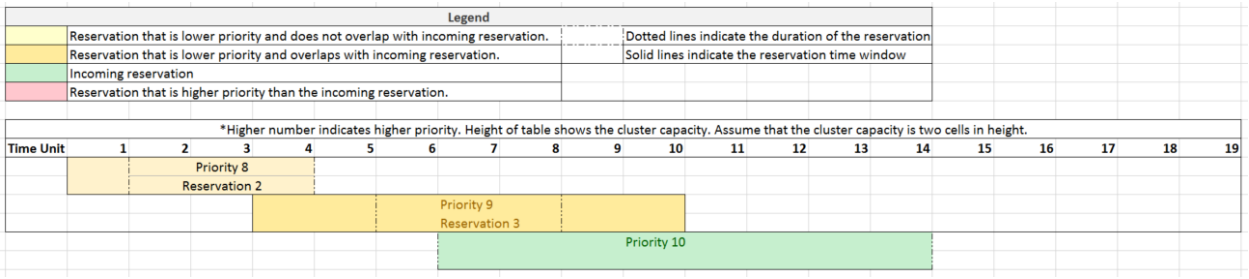

Figure 1: Image depicting the problem of simply deleting reservations overlapping the incoming reservation. **Assume that the max capacity is two cells in height.**

When the incoming reservation arrives, Reservation 3 will need to be deleted and re-scheduled. Since Reservation 2 is not overlapping the incoming reservation window, it will not be effected. Since Reservation 3 cannot fit after the incoming reservation is added, it will be deleted. This behavior is incorrect since Reservation 2 could have been moved / deleted to make room for Reservation 3 which is higher priority. This problem becomes recursive due to this behavior. That is, in order for the algorithm to do the right thing, it must remove all overlapping reservations that are lower priority, as well as all overlapping reservations that are lower priority than any of the previously removed reservations.

The proposed algorithm is the following:

1. Define a recursive method with the signature

   HashSet<ReservationId> getSortedOverlappingReservations(
   SortedMap<ReservationInterval, Set<InMemoryReservationAllocation>>
   allReservations,
   ReservationId reservationId) :

   a. Create HashSet<ReservationId> called *reservations*.
   b. Add *reservationId* to *reservations*.
   c. Find all overlapping reservations in *allReservations* that are lower priority that do not exist in *reservations*. Complexity details:

Search: O(N + E) where N is the number of reservations, and E is the sum of overlapping reservations for each reservation. (Note if allReservations is very dense, for example if all reservations are made starting and ending at the same time, O(E) = O(N^2)).

d. Call getSortedOverlappingReservations on each reservation found in step b, and merge the results with HashSet<ReservationId> *reservations*. Complexity details:

Merge: O(1) most cases, O(N) on hash collision or adding beyond capacity.

e. Return *reservations*.

2. Get all reservations that are lower in priority than the incoming reservation called *allReservations.* Note that this step is optional and that in some circumstances, it may be better not to pre-prune.
3. Call getSortedOverlappingReservations(allReservations, incomingReservationId). Complexity details: O(N + E) where E is the sum of overlapping reservations for each reservation
4. Sort the output of calling getSortedOverlappingReservations. Complexity details: O(N*log(N))

The total average complexity of the algorithm is O(N*log(N)) since the sorting algorithm is the bottleneck.

2. When support for recurring reservations is incorporated to the reservation system (YARN-5326), recurring reservations will need to bet set with strictly higher priority than any other reservation.