SPARK-12133: Dynamic Allocation in Spark Streaming

Andrew Or, Tathagata Das
Last edited: 12/5/15

Dynamic allocation is a feature in Spark where an application scales the number of executors up and down based on the workload. This feature does not currently work well with Spark Streaming applications for several reasons:

- Executors may never be idle since batches run every N seconds
- Heuristics do not take into account the batch queue
- Executors running receivers should be treated differently

The goal of this issue is to take into account these considerations and integrate dynamic allocation more tightly with Spark Streaming. This will happen in multiple steps:

Version 1: Basic functionality
Version 2: Receiver-aware scheduling
Version 3: Variable number of receivers

Version 1: Basic functionality

Every minute, we make a decision of whether to add or remove executors. The metric we use is the ratio of *batch processing time / batch duration* (R). If the average R over the past minute (R_{avg}) is > 0.9, we add executors. If R_{avg}< 0.5, we remove executors. When we add executors, we request `round`(R_{avg}) executors with a minimum of 1. When we remove executors, we always remove 1 executor. Executors with receivers are never removed in this version.

This design has the following properties:

- **Proactive scaling.** The batch queue builds up when processing time is consistently higher than the batch duration, i.e. R_{avg} > 1. This approach preemptively requests executors before the batch queue starts building up, i.e. as soon as R_{avg} > 0.9.

- **Gradual adjustment.** The idea is to adjust the number of executors slowly over time. Unlike the batch case, cluster usage in streaming should be relatively steady; it is uncommon for the cluster to be completely idle or completely busy at any given time. For this reason, we add/remove 1 executor at a time rather than reusing the existing heuristics in Spark Core.

- **Burst mitigation.** If the R of a single batch spikes, the system remains stable because we take the average of all R's within the past minute. If the R_{avg} within a minute spikes, then we will request number of executors proportional to the R_{avg}. This ensures that we request more resources when the persistent burst is high.

The only user-facing configuration that will be introduced is `spark.streaming.dynamicAllocation.enabled` (default false). We do not reuse the existing `spark.dynamicAllocation.enabled` flag because the two sets of heuristics have very confusing semantics if used together. If both are enabled, an appropriate error will be thrown.

Additional considerations:

- ***Why not reuse the existing heuristics but on the batch queue?*** This approach works for scaling up, but not for scaling down. Currently, we remove executors that have been idle for some time. This does not work in streaming because executors may never be idle.

- ***Why 1 minute scaling interval? Why not use a multiple of batch duration?*** The batch duration can be arbitrarily small, e.g. 100ms, in which case even if we take 10x the batch duration we will potentially launch an executor every second. This leads to a lot of churn. Additionally, any multiple of long batch durations, e.g. 10 minutes, will be too long for the system to adjust quickly.

- **How did you determine the 0.5 and 0.9 thresholds?** The upper threshold must be < 1 for the scaling to be proactive. Then, along with the upper threshold, the lower threshold specifies what the average processing time should be, i.e. 0.7 of the batch duration. We decided to make the lower threshold 0.5 because anything lower essentially means the cluster is idle half the time, an indication that we should release some resources.

- **What if adding executors doesn't lower the processing time?** In other words, if the job the user is running inherently takes more than the batch duration, then the system will always try to request executors in vain. To avoid having the requests pile up, the application must keep track of the target number of executors and refresh this every minute, overriding the target on any outstanding requests that have not been serviced.

Version 2: Receiver-aware scheduling

This builds on top of version 1, relaxing the constraint that the system never removes executors running receivers. This is important for cases where there is a receiver running on every node in the cluster, a common setup.

In this version, executors running receivers *can* be killed, but on the condition that the number of receivers (N) in the cluster must remain the same. This means if there would not have been enough cores to run all N receivers after killing a particular executor, then the executor would not be killed. All killed receivers are automatically relaunched on a different executor using an existing placement strategy that spreads out receivers across executors.

Additionally, every minute, receivers are rebalanced across the existing executors if necessary. This is crucial for scaling back up after scaling down because the receivers may no longer be evenly distributed across executors. In particular, if there is an executor with >= 1 + N_{avg} receivers, where N_{avg} is the average number of receivers per executor across the cluster, then the system kills one of the receivers on this executor to relaunch it on a different one.

Example: Suppose we start with 5 executors with 4 cores and 1 receiver each. If we scale down all the way, we will end up with 2 executors where the receiver allocation is [3, 2] (i.e. 3 receivers on the first executor, 2 receivers on the second). Now, if we scale back up by adding 1 executor, then the new receiver allocation is [3, 2, 0]. In this allocation, N_{avg} = (3 + 2 + 0) / 3 = 1.67, and 3 >= 1 + N_{avg}, so the system kills a receiver on the first executor. The final allocation will be [2, 2, 1], which is stable because no executor has more than 1 + N_{avg} receivers.

Additional considerations:

- ***Why should the number of receivers be constant?*** Otherwise, we need to adjust the number of receivers using another set of heuristics that takes into account the input rate. This is additional complexity that can be added later.

- ***Why 1 + N_{avg}?*** This essentially means an executor has at least 1 executor above the average, so we can safely remove a receiver from the executor.

- **Why 1 minute again?** No particular reason. The rebalancing just has to be run sufficiently often for the receiver allocation to adjust quickly. Note that nothing is done if there is no need to rebalance receivers.

Version 3: Variable number of receivers

Coming soon.