Understand cluster best practices

Note: *In this reading you can see the best practices that should be following when working with Azure Databrick clusters.*

Finally, it is time to think about the workhorse of your Azure Databricks jobs; the cluster. As you recall, in this scenario, no one has reevaluated the types and sizes of clusters in the workspace for several years. To allocate the right amount and type of cluster resource for a job, we need to understand how different types of jobs demand different types of cluster resources.

- **Machine Learning** - To train machine learning models its required cache all of the data in memory. Consider using memory optimized VMs so that the cluster can take advantage of the RAM cache. You can also use storage optimized instances for large datasets. To size the cluster, take a % of the data set → cache it → see how much memory it used → extrapolate that to the rest of the data.
- **Streaming** - You need to make sure that the processing rate is just above the input rate at peak times of the day. Depending on peak input rate times, consider computing optimized VMs for the cluster to make sure processing rate is higher than your input rate.
- **Extract, Transform and Load (ETL)** - In this case, data size and deciding how fast a job needs to be will be a leading indicator. Spark doesn't always require data to be loaded into memory in order to execute transformations, but you'll at the least need to see how large the task sizes are on shuffles and compare that to the task throughput you'd like. To analyze the performance of these jobs start with basics and check if the job is by CPU, network, or local I/O, and go from there. Consider using a general purpose VM for these jobs.
- **Interactive / Development Workloads** - The ability for a cluster to auto scale is most important for these types of jobs. In this case taking advantage of the Autoscaling feature will be your best friend in managing the cost of the infrastructure.

While deciding on proper cluster configuration, you should keep these important design attributes of the Azure Databricks (ADB) service in mind:

- Cloud Optimized: Azure Databricks is a product built exclusively for cloud environments, like Azure. No on-prem deployments currently exist. It assumes certain features are provided by the Cloud, is designed keeping Cloud best practices, and conversely, provides Cloud-friendly features.
- Platform/Software as a Service Abstraction: ADB sits somewhere between the PaaS and SaaS ends of the spectrum, depending on how you use it. In either case ADB is designed to hide infrastructure details as much as possible so the user can focus on application development. It is not, for example, an IaaS offering exposing the guts of the OS Kernel to you.
- Managed service: ADB guarantees a 99.95% uptime SLA. There's a large team of dedicated staff members who monitor various aspects of its health and get alerted when something goes wrong. It is run like an always-on website and Microsoft and Databricks system operations team strives to minimize any downtime.

These three attributes make ADB different than other Spark platforms such as HDP, CDH, Mesos, etc. which are designed for on-prem datacenters and allow the user complete control over the hardware. The concept of a cluster is therefore unique in Azure Databricks. Unlike YARN or Mesos clusters, which are just a collection of worker machines waiting for an application to be scheduled on them, clusters in ADB come with a pre-configured Spark application. ADB submits all subsequent user requests like notebook commands, SQL queries, Java jar jobs, etc. to this primordial app for execution.

Under the covers Databricks clusters use the lightweight Spark Standalone resource allocator. When it comes to taxonomy, ADB clusters are divided along the notions of "type", and "mode." There are two ***types*** of ADB clusters, according to how they are created. Clusters created using UI and Clusters API are called Interactive Clusters, whereas those created using Jobs API are called Jobs Clusters. Further, each cluster can be of two ***modes***: Standard and High Concurrency. Regardless of types or mode, all clusters in Azure Databricks can automatically scale to match the workload, using a feature known as Autoscaling.

Note: *In addition to Autoscaling features, use auto-termination wherever applicable (e.g. auto-termination doesn't make sense if you need a cluster for data analysis by multiple users almost through the day, etc.).*

Choosing the VM type

Different Azure VM instance types

Compute Optimized	Memory Optimized	Storage Optimized	General Purpose
FS - Haswell processor (Skylake not supported yet) - 1 core ~ 2 GB RAM - SSD Storage: 1 core ~ 16 GB	**DSv2** - Haswell processor - 1 core ~ 7 GB RAM - SSD Storage: 1 core ~ 14 GB	**L** - 1 core ~ 8 GB RAM - SSD Storage: 1 core ~ 170 GB - Price: 0.156	**DSv2 and DSv3** - DSv2 - 1 core ~ 3.5 GB RAM - DSv3 - 1 core ~ 4 GB RAM - SSD Storage: >DSv2 - 1 core ~ 7 GB >DSv3 - 1 core ~ 8 GB
H - High-performance - 1 core ~ 7 GB RAM - SSD Storage: 1 core ~ 125 GB	**ESv3** - High-performance (Broadwell processor) - 1 core ~ 8 GB RAM - SSD Storage: 1 core ~ 16 GB		

Cluster sizing starting points

Arrive at the correct cluster size by iterative performance testing

It is impossible to predict the correct cluster size without developing the application because Spark and Azure Databricks use numerous techniques to improve cluster utilization. The broad approach you should follow for sizing is:

1. Develop on a medium-sized cluster of 2-8 nodes, with VMs matched to workload class as explained earlier.

2. After meeting functional requirements, run end to end test on larger representative data while measuring CPU, memory and I/O used by the cluster at an aggregate level.

3. Optimize cluster to remove bottlenecks found in step 2

- **CPU bound**: add more cores by adding more nodes
- **Network bound**: use fewer, bigger SSD backed machines to reduce network size and improve remote read performance
- **Disk I/O bound**: if jobs are spilling to disk, use VMs with more memory.

Repeat steps 2 and 3 by adding nodes and/or evaluating different VMs until all obvious bottlenecks have been addressed.

Performing these steps will help you to arrive at a baseline cluster size which can meet SLA on a subset of data. In theory, Spark jobs, like jobs on other Data Intensive frameworks (Hadoop) exhibit linear scaling. For example, if it takes 5 nodes to meet SLA on a 100 TB dataset, and the production data is around 1PB, then prod cluster is likely going to be around 50 nodes in size. You can use this back of the envelope calculation as a first guess to do capacity planning. However, there are scenarios where Spark jobs don't scale linearly. In some cases this is due to large amounts of shuffle adding an exponential synchronization cost (explained next), but there could be other reasons as well. Hence, to refine the first estimate and arrive at a more accurate node count we recommend repeating this process 3-4 times on increasingly larger data set sizes, say 5%, 10%, 15%, 30%, etc. The overall accuracy of the process depends on how closely the test data matches the live workload both in type and size.

Rules of thumb

- Fewer big instances > more small instances - Reduce network shuffle; Databricks has 1 executor / machine - Applies to batch ETL mainly (for streaming, one could start with smaller instances depending on complexity of transformation) - Not set in stone, and reverse would make sense in many cases - so sizing exercise matters
- Size based on the number of tasks initially, tweak later - Run the job with a small cluster to get idea of # of tasks (use 2-3x tasks per core for base sizing)
- Choose based on workload (Probably start with F-series or DSv2): - ETL with full file scans and no data reuse - F / DSv2 - ML workload with data caching - DSv2 / F - Data Analysis - L - Streaming - F

Workload requires caching (like machine learning)

- Look at the Storage tab in Spark UI to see if the entirety of the training dataset is cached - Fully cached with room to spare -> fewer instances - Partially cached > Almost cached? -> Increase the cluster size > Not even close to cached -> Consider L series or DSv2 memory-optimized - Check to see if persist is MEMORY_ONLY, or MEMORY_AND_DISK - Spill to disk with SSD isn't so bad
- Still not good enough? Follow the steps in the next section

ETL and analytic workloads

- Are we compute-bound? - Check CPU Usage (Ganglia metrics to come to Azure Databricks soon) - Only way to make faster is more cores
- Are we network-bound? - Check for high spikes before compute heavy steps - Use bigger/fewer machines to reduce the shuffle - Use an SSD backed instance for faster remote reads

- Are we spilling a ton? - Check Spark SQL tab for spill (pre-aggregate before shuffles are common to spill)
- Use L-series
- Or use more memory

Other considerations

- Use latest Databricks Runtime version to take advantage of latest performance & other optimizations (applicable in most cases, though not all).
- Use High-concurrency cluster mode for data analysis by a team of users via notebooks or a BI tool, or if you want to enforce data protection via Table ACLs or ADLS Passthrough.
- Use cluster tags for project / team based chargeback.
- Use the Spark config tab if certain tuning would make sense for a specific workload (like config to use broadcast join).
- Use Event Log and Spark UI to see how different queries / workload executions perform, and what effect those have on a cluster's health.
- Configure Cluster Log Delivery
- Use Cluster ACLs to configure what each user or a group of users are allowed to do.