dremio

Dremio Software

Creating Multi-Engine Clusters

Introduction

When Dremio is installed, all executor nodes are assigned to a single default engine. This works well at the beginning of a project, but workloads change when the semantic layer is promoted to the production environment, and users query live datasets. This can cause resource contention even in large environments with 20-30 executor nodes.

This document recommends when, why, and how to create a multi-engine cluster.

Prerequisites

This document assumes Dremio has been installed, running, and configured for unlimited splits. This is set by default for Dremio 21+.

The "Configure WLM based on Query Analysis" section also assumes you have Dremio Query Analyzer running and its associated VDSs set up in Dremio. Please contact Dremio Professional Services for details on how to receive the Query Analyzer tool. Version 24.3+ also provides a system table, sys.jobs_recent, to get this information without using Query Analyzer.

This document also assumes that the user is familiar with Dremio WLM, queryCost, query types, etc.

Out of Scope

This document does not discuss query memory management or tuning through Dremio's support options.

Default Engine

When Dremio is installed, there is only one execution engine which comprises all the executor nodes. Advanced configuration for multi-engine setup cannot be delivered out of the box because each organization's use case and needs are very different. The workload which this single engine handles is:

- All the user queries (UI_RUN, JDBC/ODBC/Flight, REST API queries)
- All heartbeat query checks (e.g., "SELECT 1" from the reporting/BI tools)
- All catalog inquiries (queries from BI/reporting tools to get information about the table metadata or schema)
- All the reflection refreshes
- All the metadata refreshes

This document will mainly focus on handling the above minus the catalog inquiries. Catalog inquiries are generally very light, and no query cost is associated with them.

Working with the Default Engine

In a development environment, the default engine can handle all these jobs because of the following reasons:

- The data volume is generally smaller than the production environment.
- The number of datasets involved is lower than in the production environment.
- The data sets may not be updated with fresh data or not as frequently as in the production environment.

- The number of users is limited to the number of developers and not the actual users in the production environment.
- The load tests focus on volume testing or concurrency for user queries. During these tests, the same reflection and the same dataset are used without considering that they may be getting updated
- Intuitively, owners of the infrastructure want all nodes to be in the same engine so all nodes are used, and there is less chance of a node remaining idle.

User queries can slow down as usage increases in the production environment because of resource contention. Inconsistent long query response times are the first sign of resource contention. Metadata and Reflection refresh jobs are system jobs run in the background. They are not monitored as closely as user queries. Therefore, it is recommended to create multiple engines when setting up the production environment.

Resource Contention

Resource contention happens when many jobs are running simultaneously, and all are contending to get a portion of the server's resources (CPU, threads, memory, network bandwidth) to complete their work.

Dremio may be processing several queries and metadata refresh jobs as well as refreshing a reflection simultaneously. Depending on the size of the dataset, number of source files, and available memory, all these jobs could be fighting for limited resources.

With Resource contention, several things can be observed when you look at the query profile.

Long Sleep times		Long Block Times		Long Wait Times
Avg Sleep	**Max Sleep**	**Avg Blocked**	**Max Blocked**	**Wait Time**
2m52s	2m52s	34m28s	34m28s	58.151s
3m7s	43m15s	0.011s	0.113s	59.077s
				53.416s

Analyzing the Workload

To analyze the workload, we will make the following assumption in this section:

- We have a production environment, and many users now access Dremio using a BI or a reporting tool.
- Queries have been optimized to a satisfactory level.
- The workload on the server consists of user queries, reflection refresh jobs, and metadata refresh jobs
- We have Dremio Query Analyzer (DQA) installed or access to system tables.

The best tool to analyze the workload is Dremio Query Analyzer (DQA). It is also constructive to review some of the troubling query profiles. DQA utilizes Dremio to read the query history in the "queries.json" file(s).

Metadata Refresh Jobs

Start by focusing on the metadata refresh jobs (all jobs that start with REFRESH DATASET). Charting the data per day, we can see an example scenario below:

start Date	TotalQueryCount	AvgQueueTimeSec	MaxExecTimeSec	AvgExecTimeSec	80thPercentile
3/11/2023	1589	1	1139	48	72
3/12/2023	1871	91	740	45	74
3/13/2023	1761	220	3044	84	103
3/14/2023	1627	62	1410	73	98
3/15/2023	1776	20	1922	68	88
3/16/2023	1664	48	2839	82	96
3/17/2023	1688	3	2502	63	92
3/18/2023	1309	1	2183	49	68
3/19/2023	1072	3	701	41	66
3/20/2023	667	2	578	73	103
3/27/2023	1548	6	1410	61	74
3/28/2023	1451	6	766	62	83

Even when MD refresh jobs are not spending much time in the queue, we can see a massive execution time range with MaxExecTime above 2000 seconds, and 80% of jobs are completing well over 60 seconds.

Typically, we expect MD refresh jobs to have an average exec time of 10-15 seconds, and 75% of the jobs should be able to be completed in around 20 seconds max. Investigating job profiles should show that the job spends most of its time in Sleep Time, which means it competes for resources with other jobs running on the same engine.

We can also run this query

```
select dsName, avg(executionTime) avgExecTime,
PERCENTILE_DISC(0.75) WITHIN GROUP ( ORDER BY executionTime ASC ) UpperPercentile
from MDRefreshData where outcome='COMPLETED'
group by 1 order by 2 desc
```

And the results….

abc dsName	## avgExecTime	## UpperPercentile
[redacted]	202619.875	149425.0
[redacted]	152733.375	116430.0
[redacted]	143372.94736842104	40816.0
[redacted]	141837.76470588235	84570.0
[redacted]	134048.5873015873	165179.0
[redacted]	128139.97142857143	67521.0
[redacted]	116104.14035087719	106063.0
[redacted]	115481.22413793103	145013.0
[redacted]	114749.01587301587	122180.0
[redacted]	114450.7627118644	129367.0
[redacted]	110266.85714285714	139974.0

This shows a huge variance of MD refresh executionTime (excludes planning, queue, and pool time) for each of the datasets.

After configuring a separate engine for MD Refresh jobs, the results show that MD refresh jobs have a much more predictable and consistent execution time.

start Date	TotalQueryCount	AvgQueueTimeSec	MaxExecTimeSec	AvgExecTimeSec	80thPercentile
3/20/2023	667	2	578	73	103
3/27/2023	1548	6	1410	61	74
3/28/2023	1451	6	766	62	83
3/29/2023	1809	2	161	11	32
3/30/2023	1818	1	140	10	32
3/31/2023	1781	1	162	11	33
4/1/2023	1317	1	128	8	6
4/2/2023	1241	1	159	7	5
4/3/2023	1495	1	77	9	27
4/4/2023	1823	1	132	11	33
4/5/2023	2657	4	136	11	34

Running the same query to calculate the average and upper 75 percentile will give us the following results.

abc dsName	## avgExecTime	## UpperPercentile
[redacted]	54559.25	70495.0
[redacted]	49132.35	67156.0
[redacted]	46901.0625	42757.0
[redacted]	46459.17647058824	45352.0
[redacted]	45809.5	52337.0
[redacted]	45513.15789473684	62459.0
[redacted]	45180.77777777778	42414.0
[redacted]	43384.92227979274	43513.0
[redacted]	42712.15	64024.0
[redacted]	42089.31578947369	35456.0
[redacted]	40705.05882352941	37096.0

Another way is to calculate an index called WLMLoad.

```
select count(*) dsCount, sum(distance) totalDistance, sqrt (totalDistance/dsCount)
WLMLoad
from
(
select dsName,count(*) cnt,avg(executionTime) avgExec,
PERCENTILE_DISC(0.75) WITHIN GROUP ( ORDER BY executionTime ASC ) UpperPercentile,
(avgExec - UpperPercentile)*(avgExec - UpperPercentile) distance
from MDRefreshData
where outcome='COMPLETED'
group by 1
having stddev(executionTime) > 0
order by 1
) t1
```

WLMLoad calculation is as follows:

avgExec = avg execution time of MD Refresh of each dataset
UpperPercentile = 75% executionTime of MD Refresh of each dataset
dsCount = number of distinct datasets
Distance = (avgExec - UpperPercentile) ^ 2
WLMLoad = sqrt (sum of all distance) / dsCount
For Metadata refresh jobs, WLMLoad should be less than 15000.

Metadata refresh jobs do not require a massive engine. A single node of the same class as other nodes in the production environment can generally handle the task. If executor nodes have 16 cores/64GB memory, the MD refresh node should be the same caliber.

If the datasets are parquet files, then you can refresh them by partitions, which will lighten the load even further, as Dremio will only focus on the newly created partitions. Furthermore, if you have datasets based on DeltaLake or Iceberg tables, Then MD refresh jobs do not apply to them, and a separate engine may not be required at all. DeltaLake tables are refreshed by the coordinator.

Reflection Refresh Jobs

If after all metadata jobs have been assigned to their own engine and you are still experiencing long sleep times, you can do the same exercise for reflection refresh jobs.

```
Unset
select reflection_id, STDDEV(executionTime) stddev
from ReflectionRefreshData where outcome='COMPLETED'
group by 1 order by 2 desc
```

Will provide these results:

abc reflection_id	## stddev
fba5e92d-5a75-4456-bb5d-2cd0406fcd62	772320.0534403302
f9d7fa5a-a5b5-4065-94c1-3cfd8781b568	563690.4068049815
e59375d5-ad30-4162-a524-12c7c40658a6	592364.4573128094
dccceada-5059-4167-9865-d6f15c789493	1078501.0945542972
d33d78d8-969c-4cc6-b015-3c78663c2a9e	444509.7701582881
93e87e60-b97b-4f5a-a859-9b85918cc6de	548869.6349942597
82543c5c-c515-4e3f-a449-7f14f2456ac2	21100.117864865882
7c19a2c1-b2da-489b-87a8-f9c5a7574fff	26875.037307557523
7be35c2d-f21b-46a7-8df6-2f12df67b92b	793983.2390513498
5fd7d3da-a7a3-4e21-bbd2-1f7355a7b65d	545102.0229707725
52ae9b65-6b52-4724-ab09-c954d6879db1	46303.915320481385
2b5b6e02-40c1-4256-96e5-5e9a9e5ba4a6	307018.62991630496

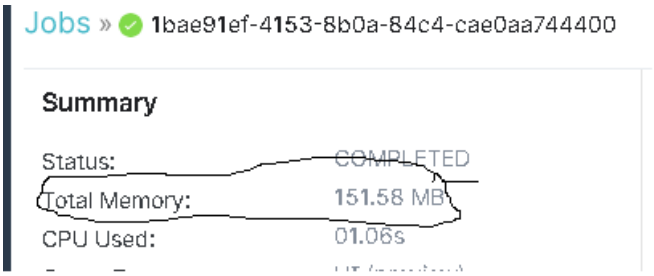
As you can see, the stddev is very high for individual reflections. There is solid evidence for separating an engine for Reflection refreshes jobs.

If stddev is larger than 200000(200 sec) in the top 10 or 15 records, then there is solid evidence that we need to separate an engine for Reflection refresh.

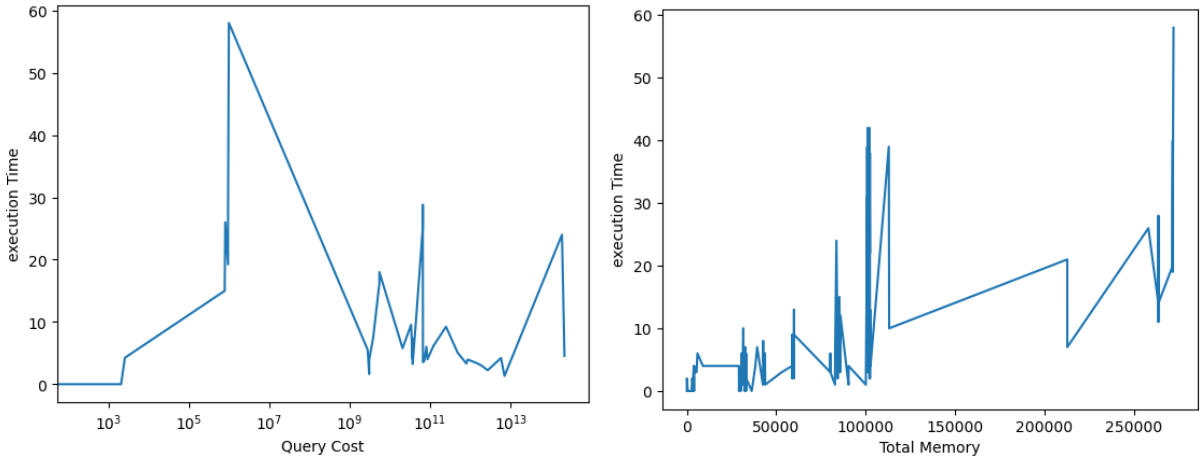
User Query Engines

Query cost is not directly related to the execution time of the query. It is possible for a query to have a query cost of ~ 4E+15, and it can get executed on an engine <5 sec without reflections. Alternatively, a query can have a cost of 4E+10 and run on a dedicated cluster for 4-5 minutes.

It's important to remember that query cost is an "estimate" calculated as the sum of the number of rows (records) processed by each operator in each phase. Dremio's planner cannot know how much data would be processed in join clauses (hash join or nested loop join). However, after a query is executed, we have a metric that we can use to get a sense of how complex it is. That metric is **Total Memory**, which is the total amount of memory used on all nodes to process the query.



Total Memory can be found in queries.json files. Therefore, it can be viewed and analyzed using DQA. We can compare the relationship between query cost, Total memory, and execution time.



As you can see, Total Memory is more directly related to execution time than query cost.

You can run this query to see what the range of allocatedMemory for queries in a queue

queueName	minMem(MB)	90PctMem(MB)	maxMem(MB)
High Cost User Queries	30110	82890	102185
Mid Cost User Queries	30110	43683	43825
Low Cost User Queries	11	101077	272099

As you can see, in the scenario above, mid-cost user queries are the most consistent when it comes to memory allocation, ranging from 30 to 48GB. High-Cost User Queries memory allocation ranges from 30 to 102GB. Considering that they are allocated to the High-Cost User Query queue and the concurrency is generally low (1-5), this is okay. However, the Low-Cost User queries range from 11MB-272GB. This is a vast range. If several large memory-consuming queries (270GB) run simultaneously, it could significantly impact the health of the executor nodes as they may run out of memory for other queries.

If query tuning is not possible to reduce the memory consumption, then breaking the default engine into 2 or more engines is recommended to help the system be more stable.

- Use DQA to identify queries that are extremely memory intensive and identify their query cost. We hope that only 2 or 3 queries have such profiles.
- Write a WLM rule that selects those queries identified in the previous step and forwards them to the High-Cost User Query queue. Since the HCUQ queue has a low concurrency, we ensure that only a few of those queries are executed simultaneously. For example, this will be very specific, but it is what we need.
- Create a SMALL-ENGINE to handle all Mid to High-cost user queries; the default engine can handle all other queries.
- Create a High-Cost Engine to handle all high-cost queries with a low concurrency and identify high memory usage queries to run with JDBC tags or different user id and use rules to assign them to the High-Cost Engine.

This approach ensures that these memory-intensive queries are isolated and will not affect other queries.

If having a separate engine for a few memory-intensive queries is not desirable, another option is the redistribution of the workload by query cost. As a best practice, it is recommended that Low and High-cost user queries should be split the user queries by 75% and 25%, respectively. If some memory-intensive queries fall into the LCUQ queue, the high concurrency (e.g., 20) in LCUQ can allow multiple queries to run simultaneously. You can distribute it by 70/30% or 65/35%. Let's assume the boundary query cost is 170M. You can run this query to see if any memory-intensive job would fall in the LCUQ.

```
Unset
select min(memoryAllocated) minMem, avg(memoryAllocated) avgMem,
PERCENTILE_DISC(0.80) WITHIN GROUP ( ORDER BY memoryAllocated ASC ) "80thPercentileMem",
max(memoryAllocated) maxMem, count(*) cnt
from SelectQUeryData
where queryCost < 170000000 and outcome='COMPLETED'
```

# ▮inMe▮ ···	## avgMe▮ ···	## 80thPercentileMe▮ ···	# ▮axMe▮ ···	# cnt ···
0	4.9822346845640 49E8	1.00275968E9	15793472832	5769

This shows that no query will consume more than 16GB of memory in the LCUQ queue. Since 80% of queries only consume 1GB of memory, and if you have a cluster of 10 nodes with Direct memory set to 110GB, you can easily allow a concurrency of 15-20 for LCUQ.

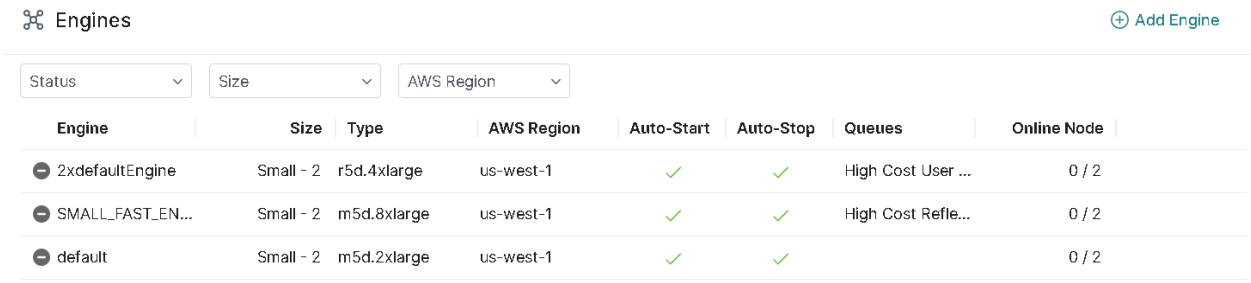
Creating a Multi-Engine cluster

This section shows how to create a multi-engine cluster. Creating a multi-engine cluster can be different based on the type of deployment that you have. We will cover AWSE, Standalone, and K8s.

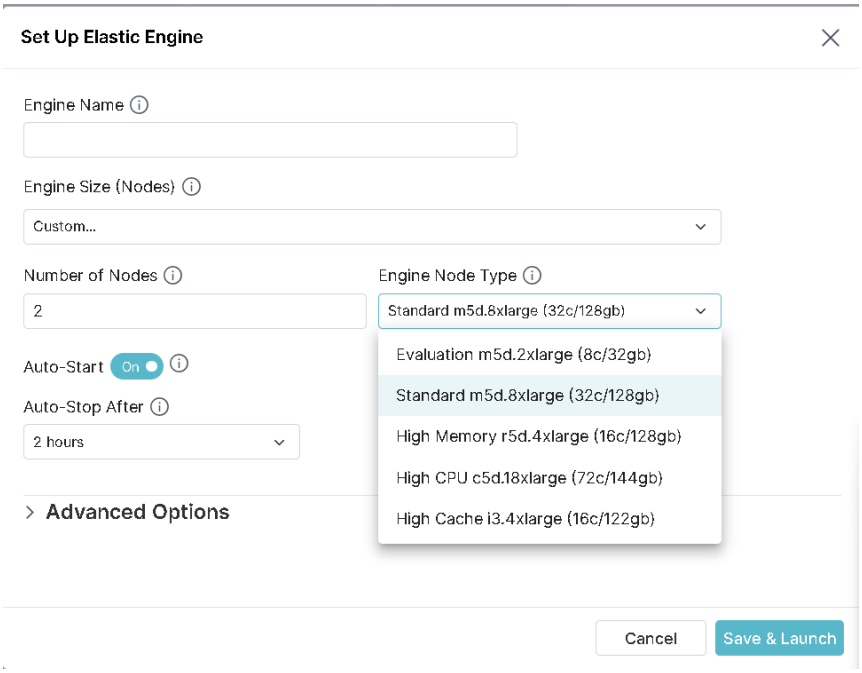
AWSE

In AWSE, creating a multi-engine cluster is very easy, and does not require a restart of the cluster.

Go to Setting → Engines and click on Add Engine



In the Set Up Elastic Engine page, specify a name (e.g., Reflection Engine), the number of nodes, and the type of the node. Ensure Auto-Start is set to ON. and Auto-Stop is also set to a reasonable value. (e.g. 2hrs). This way, the engine will shut itself down after 2 hours when not in use, and the cost of ownership will be lowered.



After you create the engine, use queues and rules to route the reflection refresh jobs to the new engine. This way, reflection refresh jobs will not take any resources from the engine running user queries.

Standalone VMs (Non-K8s)

In a standalone VM deployment, you must decide which nodes in your cluster should be part of the new or the default engine.

For each of the nodes that you want to move to your new engine, login and modify the dremio.conf file by adding this line in the services section….

```
node-tag: "ReflectionEngine"
```

For all other nodes,

```
node-tag: "default"
```

You do not need to restart the coordinator, but you need to restart each of the executors after you have modified the dremio.conf file. Remember that the new engine will not appear on the Engines page of Dremio Settings. However, on the Node Activity page, each node will have the new Engine name.

After you create the engine, use queues and rules to route the reflection refresh jobs to the new engine. This way, reflection refresh jobs will not take any resources from the engine running user queries.

Kubernetes (k8s)

For creating different engines in K8s deployment, please refer to the helm charts provided by Dremio.
https://github.com/dremio/dremio-cloud-tools/tree/master/charts/dremio_v2
There are multiple sections in values.yaml file, which needs to be modified.

1. In the executor section, list the engines you want for your cluster…. For example..

```
# Engines
# Engine names be 47 characters or less and be lowercase alphanumber characters or '-'.
# Note: The number of executor pods will be the length of the array below * count.
engines: ["default","mdrefresh","refrefresh"]
```

2. In the engineOverride section, you need to define the configuration of each engine… for example… 2 nodes in the ***default*** engine

```
  default:
    cpu: 6
    memory: 28000
#
    count: 2
#
#     annotations: {}
#     podAnnotations: {}
#     labels: {}
#     podLabels: {}
#     nodeSelector: {}
#     tolerations: []
#
#     serviceAccount: ""
#
#     extraStartParams: >-
#       -DsomeCustomKey=someCustomValue
#
#     extraInitContainers: |
#       - name: extra-init-container
#         image: {{ $.Values.image }}:{{ $.Values.imageTag }}
#         command: ["echo", "Hello World"]
#
#
#     extraVolumes: []
#     extraVolumeMounts: []
#
    volumeSize: 50Gi
#     storageClass: managed-premium
#     volumeClaimName: dremio-default-executor-volume
#
    cloudCache:
      enabled: true
#
#       storageClass: ""
#
      volumes:
      - name: "default-c3"
        size: 100Gi
#         storageClass: ""
```

3. If you have different node groups for different engines, you can specify them in the nodeSelector.

```
volumeSize: 50Gi
nodeSelector:
  eks.amazonaws.com/nodegroup: dremio-ps-eks-exec-eng2-ng
  node.kubernetes.io/instance-type: m5d.8xlarge
```

4. Save your helm chart and run *helm install [deployment_name]* .

Setting up Queues for Heartbeat queries

When working with BI or reporting tools, setting up connection pool properties that check the connection before issuing the query is common. With this setting, the server will submit a simple query, e.g., “SELECT 1” to Dremio to ensure the connection is still alive before submitting the user query. A good way to isolate these queries is to assign them to a Heartbeat Queue.

Queue and Job memory limits should be 30 and 10MB, respectively.

Queues

Name	CPU Priority	Concurrency Limit	Queue Memory Limit per Node	Job Memory Limit per Node
Heartbeat Queries	High	100	30 MB	10 MB

The cost of the “SELECT 1” query is 7. To assign these queries to the Heartbeat Queue, have a WLM rule similar to

```
query_cost() < 7 and query_type() in ('JDBC','ODBC','Flight')
```

Conclusion

Let's summarize our findings and recommendations here.

Separating out engines becomes mandatory when there is solid evidence of

1. Resource contentions between user queries and/or Dremio's internal tasks (MD refresh and reflection refresh jobs). This is evident in large sleep times when reviewing query profiles.
2. The standard deviation for metadata refreshes is higher than 50,000 for individual datasets.
3. The standard deviation for reflection refreshes is higher than 200000 for individual reflections.
4. Large Memory consumption (200GB+) by few queries.

These issues will affect the performance of queries which is considered the primary purpose of a data lake engine. You can proactively set up multiple engines from the beginning to avoid the calculation and risk of contention on your cluster.

On the first days of going live with your application, you may not observe any solid or consistent issues. For example, a memory-intensive query that runs only once a day in the morning, may cause OUT_OF_MEMORY errors for a few other queries which end users may just refresh their dashboards and ignore the problem. With a proper load test or analysis of the cluster using Dremio Query Analyzer, you may find evidence of poor performance due to resource contention.

Here are our recommendations if you have a cluster with 10-20 nodes.

- Separate Dremio's internal jobs to their own engines.
- We dedicated 1 to 4 nodes for the metadata refresh engine. Start with 1 node, and if the volume of REFRESH DATASETS is high, then you can increase the size of the engine to 2 or 3 if required. **This is only recommended if your datasets are parquet files. Deltalake tables, JSON or CVS files use the coordinator to refresh the metadata. Iceberg tables do not need metadata refreshes.**
- Set up 2 MD refresh queues for Low and High-Cost MD refresh jobs. The majority of MD refresh jobs have a query cost of ~50M. Find the 75% boundary for query cost and configure the rules to assign them to different queues.
- 5-10% of your cluster should be assigned to MD refresh jobs.
- If after the MD refresh jobs have been isolated to their own engine and you are still observing poor performance and resource contention, it is time to separate out an engine for Reflection refreshes jobs. Start with 25% of the number of nodes in your cluster and dedicate it to the Reflection Engine. If you have a cluster of 20 nodes, allocate 5 for the Reflection refresh engine. Create Low and high-cost reflection queues and assign them to the new engine. You can check the improvement by running the sample stddev query provided above.
- 20-30% of your cluster should be assigned to reflection refresh jobs.
- The remaining 50-60%