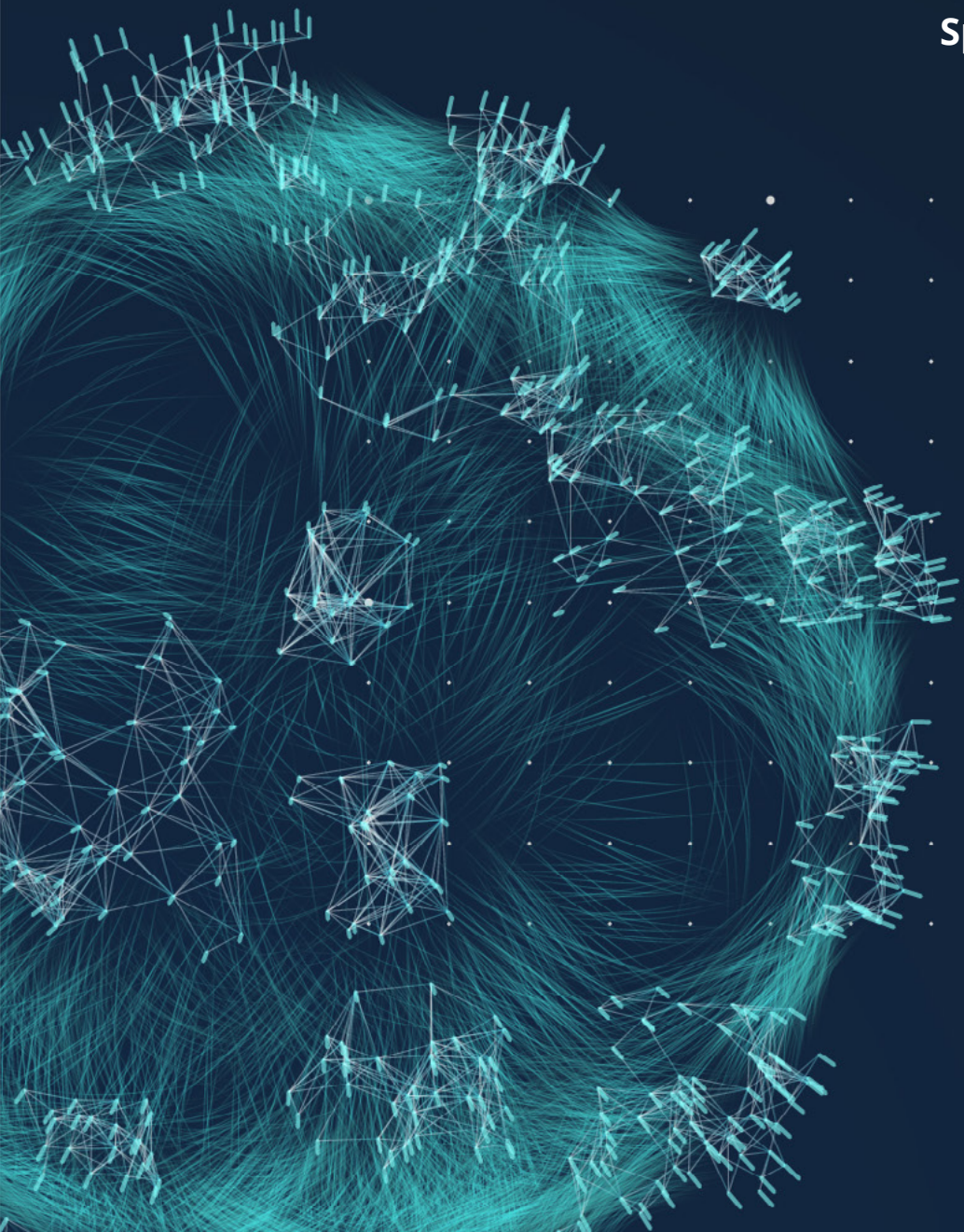
Qubole®

CO-EXISTENCE

BALANCING ACT OF MINIMIZING JOB FAILURES AND MAXIMIZING SPOT UTILIZATION

Customer's Journey of Running Spark jobs with Qubole



One of Qubole's customers—a large enterprise cloud content management company—runs several sophisticated machine learning (ML) predictive models daily for its retail clients. These models use large volumes of data, such as products, line items, purchase orders, among others. A necessary prerequisite is to prepare the data for training and running these ML models. Customer runs a daily data extract, transform, and load (ETL) job using Apache Spark on Qubole's Open Data Lake Platform. However, the job would run for over forty six minutes and fail intermittently, causing reliability issues, delays, and troubleshooting problems—aside from cost overruns and the business ramifications for its clients. So it was essential for the customer to resolve these issues and ensure the job ran reliably with the best performance possible moving forward on a regular basis.

This is three part journey of how customer implemented it and now uses as part of daily routine:

Act 1

Maximize the chance of getting spot nodes by leveraging heterogeneous cluster configurations with Qubole Open Data Lake Platform .

Act 2

Optimize the job for lower costs while ensuring reliability with Qubole platform's Intelligent Spot Management.

Sparklens

Act 3

Visualize and optimize jobs for better performance in Spark by using Qubole Sparklens.

Act 1: Heterogeneous Cluster Configurations to Rescue

Consider a cluster with following configuration:

1. Minimum Worker Nodes = 2
2. Maximum Worker Nodes = 250
3. Master Instance Type = r5.4xlarge (16 cores, 128 GB mem)
4. Worker Instance Type = r5.2xlarge (8 cores, 64 GB mem)
5. Heterogeneous Configuration = Disabled
6. Fallback to On-demand
    - Option a: Enabled
    - Option b: Disabled

Mandatory Requirement: Ensure a cluster is running only when there are active workloads.

Upon arrival of the first workload or when a scheduled interval condition is met, Qubole cluster warms up to the minimum configuration. The minimum configuration in this example will comprise one master node of type r5.4xlarge and two minimum worker nodes of type r5.2xlarge.

Let's assume the worker instance type (r5.2xlarge) configured in the cluster is relatively new. This instance's demand is high due to its popularity and is either not available in the Spot market or is running low on availability.

Scenario 1:

1. The cluster is running at the minimum configuration of one master node of type r5.4xlarge and two minimum worker nodes of type r5.2xlarge.

2. Let's say that the hourly On-Demand price for r5.2xlarge is say $10 and Spot discount is, hypothetically, 90% or $1.

3. A giant workload arrives, which triggers an upscale event and requires an additional capacity of one hundred r5.2xlarge worker instance types. This equates to 800 cores, 6400 GB mem, adding $100 to the hourly cost.

4. Since r5.2xlarge is a popular instance type and is not available in the AWS Spot Market, we have two options:

    - Option #1: Acquire one hundred On-Demand nodes, with fallback to On-Demand Feature enabled, to ensure that the cluster has sufficient capacity to meet the workload's SLA. This, however, will shoot up the hourly EC2 Cost to $1000.

    - Option #2: Keep trying to acquire the desired spot nodes if fallback to on-demand is disabled. This option introduces delays and failures and results in unpredictable costs.

Qubole platform's Heterogeneous Cluster Configuration capability is designed to address these potential problems automatically. With heterogeneous cluster configuration you can configure the cluster to have additional worker instance types.

Repeat Scenario 1 with Heterogeneous Cluster Configuration:

Now, let's say we add heterogeneous configuration to the same cluster as follows:

1. Primary Worker Instance Type - r5.2xlarge
2. Secondary Worker Instance Types - r5.xlarge and r5.4xlarge
    - An r5.xlarge instance has 4 cores and 30.5 GB memory with an AWS EC2 cost of $0.5.
    - An r5.2xlarge instance has 8 cores and 61 GB memory with an EC2 cost of $1.
    - An r5.4xlarge instance has 16 cores and 122 GB memory with an EC2 cost of $2.

The scenario 1 will now be following:

1. Same giant workload arrives and triggers an upscale event. This requires an additional capacity of one hundred r5.2xlarge worker instance types, which equates to 800 cores, 6400 GB mem.
2. With the new cluster configuration with heterogeneous config enabled, Qubole's platform will meet this need with one of following alternatives:
    - 200 r5.xlarge instances; or
    - 100 r5.2xlarge instances; or
    - 50 r5.4xlarge instances; or
    - A combination of r5.xlarge, r5.2xlarge and r5.4xlarge instances equates to the same desired additional capacity of one hundred r5.2xlarge instances.
3. If the primary worker instance type, r5.2xlarge, is not available in the spot market, the spot request will be fulfilled with the secondary instance types r5.xlarge and r5.4xlarge. The more alternate worker instance types you configure, the higher are the chances of getting spot nodes.
4. Additionally, all of the above options will add the same capacity (in this example, 800 cores and 6400 GB memory) and cost the same (in this example, $100). All three amount to the equivalent of adding 100 nodes of the primary instance type (r5.2xlarge).

As you can see Qubole's native Heterogeneous Cluster Configuration helps the customer automatically maximize the chances of getting spot nodes, thus increasing reducing costs, while ensuring the system meets workloads SLAs.

Act 2: Optimize Jobs with Maximum Spot Utilization while Ensuring Reliability

Spot nodes come at a heavily discounted price compared to On Demand or Reserved nodes. Therefore higher spot utilization equates to greater cloud cost savings. Qubole Open Data Lake Platform with Intelligent Spot Management helps maximize the chances of getting spot nodes. It also ensures that the cluster always maintains the desired spot utilization. Additionally, the platform has built-in intelligence to reduce spot loss risk, and job failure risks associated with spot loss. Qubole platform does that in 4 easy steps.

Autoscaling boundaries: Ensures that the cluster has optimal capacity to meet SLAs of ongoing workloads. The feature avoids job failures due to insufficient capacity in undersized clusters, and resource waste due to excessive capacity in oversized clusters in on-premises setups—thus helping data teams avoid unnecessary cloud costs. For the problem in hand, the cluster is configured to use 3 minimum worker nodes and autoscale upto a maximum of 250 worker nodes.

Desired Spot utilization: Maintains a certain percentage of Spot nodes in the cluster, which directly equates to the cloud cost savings. The cluster is configured to have a desired Spot utilization of 80%. This means that Qubole has to maintain 80% of total autoscaling nodes as Spot nodes. In a completely scaled out scenario for this cluster when the cluster has 250 worker nodes running, this means Qubole has to maintain about 197 Spot nodes.

Heterogenous configurations: Required spot capacity (cores and memory) is maintained even in cases where AWS is not able to fulfil the requested capacity for the configured primary worker instance type. Heterogeneous config is NOT enabled in the cluster.

Fallback to On-demand: Required capacity (cores and memory) is available via on-demand even in cases where AWS is not able to fulfil the requested spot capacity for the configured worker instance type from spot. If the cluster is configured to use this feature, Qubole fallback's to on-demand nodes in these circumstances. This configuration is enabled in the cluster.

Step 1: Optimize Spark Clusters for Spot Utilization

72 cluster instance runs happened with platform's Automated Cluster Life Cycle management that:

- Automatically starts clusters upon the arrival of a first workload or at a scheduled interval,
- Automatically scales up when the demand for more capacity increases either due to higher concurrency or the bursty nature of workloads
- Automatically scales down when workloads finish
- Auto-terminates a cluster when it is idle for a pre-configured period specified as "idle cluster timeout."

Following node metrics were collected and observations were made:

Observations

1. 33 out of 72 cluster instance runs experienced spot loss. Spot loss risk is higher with AWS instance types that are high in demand. In this particular case, all spot nodes that experienced Spot loss were of type r4.4xlarge.

2. Desired Spot utilization for the cluster was 80% and the actual (i.e.: realized) overall Spot utilization was 66.68%.

3. Spot utilization during the specific cluster run when command failure occurred was 56.92%. It hints that spot availability for the configured AWS instance type was low.

Metrics

Metrics	Cluster Run @ Failed Job (Before Optimization)	Aggregate (Before Optimization)
Node Metrics		
Total Cluster Instance Runs	1	72
Start Time	10/ 31/ 2019 17:46:00	9/ 17/ 2019 21 :00:12
End Time	10/ 31/ 2019 21:41 :00	11/ 5/ 2019 3:15:19
Total Workloads	32	675
Total Nodes Provisioned	842	25,305
Total Auto-scaling Nodes	838	25,017
Total Spot Nodes	477	16,682
Total Spot Loss Nodes	166	1,497
Average Spot Utilization - Desired	80%	80%
Spot Utilization - Actual	56.92%	66.68%
Spot Loss Rate	34.80%	8.97%

4. Spot loss rate for the cluster across all cluster instance runs was 8.97% and that rate shot up to 34.80% during the specific cluster run when command failure occurred.

From the metrics, it was evident that the cluster was experiencing performance problems due to an unusually high spot loss rate (34.80%), which was exacerbated by the spot utilization (56.92%) in the cluster.

Solution:

The goal of the solution that was implemented addressed this problem in two ways:

1. Eliminated spot loss risk while leveraging higher spot utilization in the cluster and

2. Leveraged engine level enhancements to effectively handle spot loss nodes and ensure reliability.

Higher Spot Utilization:

The following recommendations were based on the data points captured at the time when the incident occurred. Keep in mind that Spot availability in the AWS Spot market changes frequently.

1. Choose instance types with lower service interruption rates in the desired VPC region. A lower service interruption rate means higher reliability for tasks running on that instance and thus provides cost benefits associated with Spot instances (read: greater discount). Follow these guidelines to find lower service interruption instances:

    - You can check current service interruption rates here.
    - Instances with service interruption rate of <5% have lower Spot loss risk.
    - For example, r4.4xlarge has a service interruption rate of 5-10%. Whereas i3.4xlarge has a service interruption rate <5%.

2. Use heterogeneous configuration for the Spark on Qubole cluster, with at least 5 secondary worker instance types. A heterogeneous configuration with multiple secondary worker instance types maximizes the chances of getting Spot nodes in scenarios where a primary worker instance type is not available in the Spot market at the time of bidding due to high demand. This avoids the unnecessary spike in cloud costs during such periods, resulting in lower cloud costs.

    - Ensure that Spot fleet policy is in place for heterogeneous configuration for Spot requests to work - here.
    - In our example, for the customer's region, r4.8xlarge, i3.8xlarge, r4.2xlarge, i3.2xlarge were the better candidates with lower service interruption rate of < 5% at that time.

3. Use secondary worker instance types across different instance families.

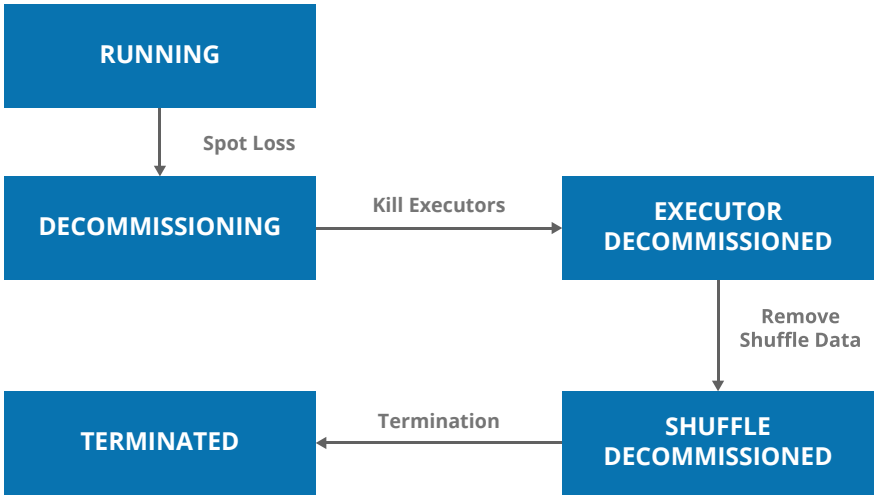
    - Generally, when an AWS instance type is experiencing Spot loss due to high demand and low availability, the availability of other instance types from the same family is also impacted. In such scenarios, having secondary worker instance types from different families ensures that the desired capacity is still fulfilled using Spot nodes belonging to secondary worker instance types. This avoids unnecessary spikes in cloud costs during such periods.

4. Reduce Spot Request Timeout. Link to cost saving and/or reliability

    - Generally, Spot loss risk for a given Spot node is directly proportional to the amount of time taken to acquire that node. A lower Spot request timeout helps acquire relatively stable Spot nodes in the cluster and ensures reliability in the cluster. Qubole's intelligent Spot rebalancer, which periodically runs as a background process in the cluster, monitors and corrects the Spot utilization in the cluster to the desired configuration.

Handle Spot Loss and Eliminate Reliability Risk

Apache Spark on Qubole effectively handles the Spot Node Loss in Engine. Enhancement makes Qubole Spark more reliable in the event of Spot Node Losses. Following state machine explains the algorithm used to handle Spot Node Loss effectively:



1. AWS sends the spot loss notification about two minutes prior to taking away the spot nodes. Upon receiving such Spot Node Loss notification, Qubole puts the impacted nodes into DECOMMISSIONING state. Additionally, Qubole ensures that no new tasks are assigned to nodes in this state.

2. Before the Spot node is lost, Apache Spark on Qubole kills all the executors running on the node. This is done to ensure fast failure of tasks, so that they can be retried on other surviving nodes. There might be tasks which could have finished within the time boundary AWS takes away the node. To avoid such scenarios, Qubole Spark waits for a configurable period of time before killing the executor. That waiting period can be configured by setting the following property - `spark.qubole.graceful.decommission.executor.leasetimePct`. After killing the executors, Node is put into EXECUTOR DECOMMISSIONED state.

3. After executors are killed, all the map outputs from the host are deleted from MapOutputTracker. This is to avoid other executors from reading the shuffle data from this node and failing due to FetchFailedExceptions. Node is put into SHUFFLE DECOMMISSIONED state.

4. After Node is terminated, move it to TERMINATED state.

5. Even when Map Outputs are cleared from the node (before moving it to SHUFFLE DECOMMISSIONED state), there will be stages that would have started reading shuffle data earlier. Those stages will fail due to FetchFailureException after Spot node is lost. Such FetchFailedException due to Spot Node Loss are ignored and stages are retried. Moreover, such retries are not counted towards failures bounded by spark.stage.maxConsecutiveAttempts (which is 4 by default). However, there is a threshold on the number of times they can experience failure due to SpotNode loss which can be configured via spark.qubole.graceful.decommission.fetchfailed.ignore.threshold.

Post Recommendation Implementation Impact Summary:

The table below depicts before and after cluster configurations and numbers of the cluster optimization recommendations that were implemented in the customer's cluster.

Before and After Configurations:

Cluster Configurations	Before Optimization	After Optimization
Master Instance Type	r4.2xlarge (8 cores, 64 GB mem)	r4.2xlarge (8 cores, 64 GB mem)
Worker Instance Type	r4.4xlarge (16 cores, 128 GB mem)	i3.4xlarge (16 cores, 128 GB mem)
Minimum Worker Nodes	2	2
Maximum Worker Nodes	250	250
Heterogeneous Configuration	No	Yes
Secondary Worker Instance Types	N/A	r4.8xlarge (32 cores, 256 GB mem), weight = 2
		i3.8xlarge (32 cores, 256 GB mem),weight = 2
		r4.2xlarge (8 cores, 64 GB mem), weight = 0.5
		i3.2xlarge (8 cores, 64 GB mem), weight = 0.5
Desired Spot Utilization	80%	80%
Fallback to Ondemand	Enabled	Enabled
Spot Request Timeout	10m	3m

Before and After Metrics:

Metrics	Cluster Run @ Failed Job (Before Optimization)	Cluster Run @ Optimized Job (After Optimization)	Aggregate (Before Optimization)	Aggregate (After Optimization)
Node Metrics				
Total Cluster Instance Runs	1	1	72	290
Start Time	10/31/2019 17:46:00	11/25/2019 17:07:00	9/17/2019 21:00:12	11/05/19 17:23:00
End Time	10/31/2019 21:41:00	11/25/2019 17:44:00	11/05/19 3:15:19	03/07/20 0:45:00
Total Workloads	32	3	675	1,732
Total Nodes Provisioned	842	346	25,305	17,287
Total Auto-scaling Nodes	838	342	25,017	16,369
Total Spot Nodes	477	304	16,682	13,054
Total Spot Loss Nodes	166	0	1,497	194
Average Spot Utilization - Desired	80%	80%	80%	80%
Spot Utilization - Actual	**56.92%** →	**88.00%**	**66.68%** →	**79.75%**
Spot Loss Rate	**34.80%** →	**0.00%**	**8.97%** →	**1.49%**

1. 15.63% increase in Average Spot Utilization (from 66.68% to 79.75%) leading to greater cloud cost savings.
2. 83.39% drop in Average Spot Loss Rate (from 8.97% to 1.49%) leading to greater cloud cost savings and greater performance.
3. Additionally, we did not observe spot loss related failures after the engine optimizations were rolled out.

Act 2 before and after scenario shows that Qubole helped the customer improve spot utilization and reduce spot loss rate while ensuring that the reliability was not impacted.

ACT 3: Optimize Spark Applications for Performance using Qubole Sparklens

Let's start with a glossary of metrics that Qubole Sparklens offers. Qubole Sparklens, when enabled, provides these metrics out-of-the-box.

Sparklens Metrics - Application Level:

Driver WallClock	Total time taken by driver to complete the execution.
Executors WallClock	Total time taken by all the executors to complete their execution.
Total WallClock	Driver WallClock + Executors WallClock
Critical Path	Minimum possible time for the app assuming unlimited resources.
Ideal Application	Minimum possible time for the app assuming perfect parallelism and no data skews.

Sparklens Metrics - Stage Level:

WallClock Stage%	Total time taken by driver to complete the execution.
PRatio	Degree of parallelism = number of tasks in the stage / number of cores. PRatio > 1 => Too many tasks PRatio < 1 => Too many cores
TaskSkew	Minimum possible time for the app assuming unlimited resources.
TaskStageSkew	Minimum possible time for the app assuming perfect parallelism and no data skews.
OIRatio	Total output of the stage (results + shuffle write) / total input (input data + shuffle read)

As mentioned in Act 1, the customer has a Spark ETL job that has reliability and performance issues. The job still ran for over forty six minutes and required optimization to reduce that time. Note that performance optimization is an iterative process.

Iteration #1:

To start with, configure the customer's spark ETL job to use Sparklens for generating the metrics by passing following arguments to spark-submit.

```
--packages qubole:sparklens:0.3.1-s_2.11
--conf spark.extraListeners=com.qubole.sparklens.QuboleJobListener
```

Iteration #1 - Qubole Sparklens Output:

1. Application Metrics

```
Time spent in Driver vs Executors
1 Driver WallClock Time     00m 26s    0.95%
2 Executor WallClock Time   45m 52s    99.05%
3 Total WallClock Time      46m 18s


4 Minimum possible time for the app based on the critical path (with infinite resources)    07m 32s
5 Minimum possible time for the app with same executors, perfect parallelism and zero skew 41m 19s
  If we were to run this app with single executor and single core                           40h 53m
```

2. Qubole Sparklens: Per Stage Metrics

Stage-ID	WallClock Stage%	OneCore ComputeHours	Task Count	PRatio	-----Task------ Skew	StageSkew	OIRatio	|* ShuffleWrite%	ReadFetch%	GC% *|
0	44.75	17h 10m	1633	27.22	2.74	0.08	2.00	|* 2.12	0.00	0.59 *|
1	17.08	07h 08m	447	7.45	1.45	0.18	4.74	|* 7.97	0.00	0.55 *|
2	38.16	16h 34m	20000	333.33	152.19	0.30	0.00	|* 0.00	0.22	0.66 *|

```
PRatio:        Number of tasks in stage divided by number of cores. Represents degree of
               parallelism in the stage
TaskSkew:      Duration of largest task in stage divided by duration of median task.
               Represents degree of skew in the stage
TaskStageSkew: Duration of largest task in stage divided by total duration of the stage.
               Represents the impact of the largest task on stage time.
OIRatio:       Output to input ration. Total output of the stage (results + shuffle write)
               divided by total input (input data + shuffle read)
```

3. Qubole Sparklens: Simulation Model Metrics

```
Real App Duration 46m 18s
Model Estimation  42m 57s
Model Error       7%

 Executor count     2  ( 10%) estimated time 409m 31s and estimated cluster utilization 99.83%
 Executor count     4  ( 20%) estimated time 205m 17s and estimated cluster utilization 99.57%
 Executor count    10  ( 50%) estimated time 83m 07s and estimated cluster utilization 98.37%
 Executor count    16  ( 80%) estimated time 52m 47s and estimated cluster utilization 96.80%
 Executor count    20  (100%) estimated time 42m 57s and estimated cluster utilization 95.16%
 Executor count    22  (110%) estimated time 39m 20s and estimated cluster utilization 94.45%
 Executor count    24  (120%) estimated time 36m 23s and estimated cluster utilization 93.60%
 Executor count    30  (150%) estimated time 29m 56s and estimated cluster utilization 91.05%
 Executor count    40  (200%) estimated time 23m 37s and estimated cluster utilization 86.51%
 Executor count    60  (300%) estimated time 17m 45s and estimated cluster utilization 76.70%
 Executor count    80  (400%) estimated time 14m 46s and estimated cluster utilization 69.17%
 Executor count   100  (500%) estimated time 13m 04s and estimated cluster utilization 62.51%
```

Iteration #1: Consolidated view of Application Configuration and Sparklens Metrics

Configurations and Metrics	278774
Spark Driver Memory	32g
Cores per Executor	3
Max Number of Executors	20
Min Number of Executors	20
Executor Memory Overhead (G)	4
Executor Memory (G)	50
# of Shuffle Partitions	20,000
Sparklens: Job Metrics	
Driver Wall Clock	00m 26s
Executor Wall Clock	45m 52s
Total Wall Clock	46m 18s
Critical Path	07m 32s
Ideal Application	41m 19s
Maximum Cores Available - Cluster Config	2000
Maximum Cores - App Config	60
"Total cores available to the app"	60
Executors Count (Actual)	20
Sparklens: Stage Metrics	
Total # of stages	3
Least Number of tasks across all stages	447
Most Number of tasks across all stages	20000
WallClock% (Max)	44.75%
PRatio (Max)	333.33
TaskSkew (Max)	152.19
StageTaskSkew (Max)	0.3
OIRatio (Max)	4.74
Sparklens: Simulation Model	
Executors Count (Actual)	20
Real App Duration	46m 18s
Model Estimated Duration	42m 57s
Model Error	7%
Model Estimation: Executors Count	100
Model Estimation: Estimated Duration	42m 57s

Iteration #1 Sparklens Metrics:

1. Total Wall Clock = ~ 46m
2. Critical Path = ~ 7m
3. Maximum available cores as per App Config = 60
4. Least number of tasks across all stages = 447
5. PRatio = 333.33

Observations:

1. There was a huge disparity in Critical Path and Total Wall Clock which confirmed that there was a lot of room to optimize resources.
2. PRatio > 1 => Too many tasks, but not enough cores!
3. Min Tasks across all stages = 447. => Adding more cores (one core per task) should increase parallelism and improve performance.
4. 447 cores = 447/4 = ~ 112 Executors.

Next Step / Action:

Increase Max Executors from 20 to 100.

For the next iteration, the customer increased parallelism in the executor stage by adding more executors.

Iteration #2: Consolidated view of Application Configuration and Sparklens Metrics

Configurations and Metrics	279108
Application configuration	
Spark Driver Memory	32g
Cores per Executor	4
Max Number of Executors	100
Min Number of Executors	20
Executor Memory Overhead (G)	4
Executor Memory (G)	50
# of Shuffle Partitions	20,000
Sparklens: Job Metrics	
Driver Wall Clock	01m 01s
Executor Wall Clock	15m 09s
Total Wall Clock	16m 11s
Critical Path	07m 15s
Ideal Application	07m 38s
Maximum Cores Available - Cluster Config	2000
Maximum Cores - App Config	400
"Total cores available to the app"	400
Executors Count (Actual)	100
Sparklens: Stage Metrics	
Total # of stages	3
Least Number of tasks across all stages	447
Most Number of tasks across all stages	20000
WallClock% (Max)	38.52%
PRatio (Max)	50
TaskSkew (Max)	123.98
StageTaskSkew (Max)	0.74
OIRatio (Max)	4.73
Sparklens: Simulation Model	
Executors Count (Actual)	100
Real App Duration	16m 11s
Model Estimated Duration	11m 36s
Model Error	28%
Model Estimation: Executors Count	400
Model Estimation: Estimated Duration	07m 36s

Iteration #2 Sparklens Metrics:

1. Total Wall Clock = ~ 16m
2. Critical Path = ~ 7m
3. Maximum available cores as per App Config = 400
4. Least number of tasks across all stages = 447
5. PRatio = 50
6. Simulation Model confirmed that adding more executors would help improve performance.

Observations:

1. Total Wall Clock dropped drastically so increasing max executors count from 20 to 100 helped. There is still a reasonable disparity between Total Wall Clock and Critical Path.
2. PRatio dropped to 50 but it's still high enough which indicates that there are more tasks than available cores.

Next Step / Action:

Action: Increase Max Executors from 100 to 400.

Iteration #3: Consolidated view of Application Configuration and Sparklens Metrics

Configurations and Metrics	279332
Spark Driver Memory	32g
Cores per Executor	4
Max Number of Executors	400
Min Number of Executors	20
Executor Memory Overhead (G)	4
Executor Memory (G)	50
# of Shuffle Partitions	20,000
Sparklens: Job Metrics	
Driver Wall Clock	01m 23s
Executor Wall Clock	15m 36s
Total Wall Clock	17m 00s
Critical Path	07m 37s
Ideal Application	03m 37s
Maximum Cores Available - Cluster Config	2000
Maximum Cores - App Config	1600
"Total cores available to the app"	1252
Executors Count (Actual)	313
Sparklens: Stage Metrics	
Total # of stages	3
Least Number of tasks across all stages	447
Most Number of tasks across all stages	20000
WallClock% (Max)	43.16%
PRatio (Max)	15.97
TaskSkew (Max)	117.79
StageTaskSkew (Max)	0.76
OIRatio (Max)	4.73
Sparklens: Simulation Model	
Executors Count (Actual)	313
Real App Duration	17m 00s
Model Estimated Duration	08m 06s
Model Error	52%
Model Estimation: Executors Count	1565
Model Estimation: Estimated Duration	07m 41s

Iteration #3 Sparklens Metrics:

1. Total Wall Clock = ~ 9m
2. Critical Path = ~ 7m
3. Ideal Application = ~ 3m
4. PRatio = 17.61
5. Model Estimation: Executors Count = 1420
6. Model Estimation: Estimated Duration = ~7m

Observations:

1. Disparity in Total Wall Clock and Critical Path has reduced significantly.
2. Disparity in Critical Path and Ideal Application means that there is data skew.
3. The simulation model confirms that adding more executors would NOT help improve performance.

Next Step / Action:

Action: Keep Max Executors to 400 but Increase Min Executors from 20 to 100 so that nodes are acquired beforehand

Iteration #4: Consolidated view of Application Configuration and Sparklens Metrics

Configurations and Metrics	279368
Spark Driver Memory	32g
Cores per Executor	4
Max Number of Executors	400
Min Number of Executors	100
Executor Memory Overhead (G)	4
Executor Memory (G)	50
# of Shuffle Partitions	20,000
Sparklens: Job Metrics	
Driver Wall Clock	01m 03s
Executor Wall Clock	08m 06s
Total Wall Clock	09m 10s
Critical Path	07m 08s
Ideal Application	03m 29s
Maximum Cores Available - Cluster Config	2000
Maximum Cores - App Config	1600
"Total cores available to the app"	1136
Executors Count (Actual)	284
Sparklens: Stage Metrics	
Total # of stages	3
Least Number of tasks across all stages	447
Most Number of tasks across all stages	20000
WallClock% (Max)	58.44%
PRatio (Max)	17.61
TaskSkew (Max)	114.6
StageTaskSkew (Max)	0.99
OIRatio (Max)	4.73
Sparklens: Simulation Model	
Executors Count (Actual)	284
Real App Duration	09m 10s
Model Estimated Duration	08m 24s
Model Error	8%
Model Estimation: Executors Count	1420
Model Estimation: Estimated Duration	07m 13s

Iteration #4 Sparklens Metrics:

1. Total Wall Clock = ~ 17m
2. Critical Path = ~ 7m
3. PRatio = 15.97
4. Model Error = 52%
5. Model Estimation: Executors Count = 1420
6. Model Estimation: Estimated Duration = ~7m

Observations:

1. Increasing max executors from 100 to 400 did NOT help this time!
    - Reason: Upscaling from 20 Min Executors to 400 Max Executors required the cluster to be upscaled which added some delays.
2. Disparity in Critical Path and Total Wall Clock confirms that there is still room to optimize resources.
3. PRatio has dropped significantly from 50 to 15.97.
4. The simulation model confirms that adding more executors would NOT help improve performance.

Next Step / Action:

Action: Keep Max Executors to 400 but Increase Min Executors from 20 to 100 so that the required nodes are acquired immediately after job submission instead of waiting for spark auto-scaling to kick in.

Summary: Consolidated view across iterations

Configurations and Metrics	Description	278774	279108	279332	279368
Application configuration					
Spark Driver Memory	spark.driver.memory	32g	32g	32g	32g
Cores per Executor	spark.executor.cores	3 →	4	4	4
Max Number of Executors	spark.dynamicAllocation.maxExecutors	20 →	100 →	400	400
Min Number of Executors	spark.executor.instances	20	20	20 →	100
Executor Memory Overhead (G)	spark.yarn.executor.memoryOverhead	4	4	4	4
Executor Memory (G)	spark.executor.memory	50	50	50	50
# of Shuffle Partitions	spark.sql.shuffle.partitions (default = 200)	20,000	20,000	20,000	20,000
Sparklens: Job Metrics					
Driver Wall Clock	Total time utilized by driver	00m 26s	01m 01s	01m 23s	01m 03s
Executor Wall Clock	Total time utilized by executors	45m 52s	15m 09s	15m 36s	08m 06s
Total Wall Clock	Total time utilized by driver + executors	46m 18s	16m 11s	17m 00s	09m 10s
Critical Path	Total Time with maximum parallelism	07m 32s	07m 15s	07m 37s	07m 08s
Ideal Application	Total Time with no data/task skews	41m 19s	07m 38s	03m 37s	03m 29s
Maximum Cores Available - Cluster Config	Derived from instance type and max worker nodes	2000	2000	2000	2000
Maximum Cores - App Config	Derived from Max Executors config	60	400	1600	1600
"Total cores available to the app"	Actual Executors that were allocated to the appl	60	400	1252	1136
Executors Count (Actual)		20	100	313	284
Sparklens: Stage Metrics					
Total # of stages		3	3	3	3
Least Number of tasks across all stages		447	447	447	447
Most Number of tasks across all stages		20000	20000	20000	20000
WallClock% (Max)		44.75%	38.52%	43.16%	58.44%
PRatio (Max)		333.33	50	15.97	17.61
TaskSkew (Max)		152.19	123.98	117.79	114.6
StageTaskSkew (Max)		0.3	0.74	0.76	0.99
OIRatio (Max)		4.74	4.73	4.73	4.73
Sparklens: Simulation Model					
Executors Count (Actual)		20	100	313	284
Real App Duration		46m 18s	16m 11s	17m 00s	09m 10s
Model Estimated Duration		42m 57s	11m 36s	08m 06s	08m 24s
Model Error		7%	28%	52%	8%
Model Estimation: Executors Count		100	400	1565	1420
Model Estimation: Estimated Duration		42m 57s	07m 36s	07m 41s	07m 13s

Impact Summary:

1. Increasing max executors count from 20 to 100 increased parallelism significantly. The PRatio came down from ~333 to 50 and Total Wall Clock reduced from ~46m to 16m.
2. Increasing max executors count from 100 to 400 increased parallelism further up. PRatio came down from 50 to 15.97. Total Wall Clock remained flat.
3. Increasing min executors from 20 to 100 brought down Total Wall Clock Time from ~17m to ~9m.
4. 80% drop in latency: The job that originally ran for ~46 minutes was optimized to complete in ~9 minutes.
5. The disparity between Total WallClock and Critical Path dropped down significantly to a satisfactory level.

Thus, the customer optimized the spark application performance by increasing the degree of parallelism using cues from sparklens metrics of the spark application.

KEY TAKEAWAY

Important considerations for sizing your spark applications:

1. **Concurrency and Cluster auto-scaling limits (min, max worker nodes):** The number of applications that may run concurrently and cluster autoscaling limits need to be considered while sizing the spark application.
2. **Workload scaling limits (min, max executors):** The degree of parallelism of the spark application is limited to workload scaling limits configured for the application.
3. Beyond a certain threshold, adding more resources (executors and cores) will NOT help. Identifying that threshold for your spark application is extremely important to avoid unnecessary resource wastage and cloud cost.
4. **Performance vs cost trade-off:** Adding more resources will help complete the long running stages faster. However, these additional resources, if acquired upfront for the entire duration of the job, can remain idle (unutilized) during the execution of smaller stages. This could drop cluster utilization and hence increase overall cost.

For more information:

Contact:
sales@qubole.com

Try QDS for Free:
https://www.qubole.com/products/pricing/

469 El Camino Real, Suite 205
Santa Clara, CA 95050
(855) 423-6674 | info@qubole.com

WWW.QUBOLE.COM