Troubleshoot performance bottlenecks in Azure Databricks

04/02/2019 • 6 minutes to read • Contributors

In this article

This article describes how to use monitoring dashboards to find performance bottlenecks in Spark jobs on Azure Databricks.

Azure Databricks is an Apache Spark–based analytics service that makes it easy to rapidly develop and deploy big data analytics. Monitoring and troubleshooting performance issues is a critical when operating production Azure Databricks workloads. To identify common performance issues, it's helpful to use monitoring visualizations based on telemetry data.

Prerequisites

To set up the Grafana dashboards shown in this article:

- Configure your Databricks cluster to send telemetry to a Log Analytics workspace, using the Azure Databricks Monitoring Library. For details, see Configure Azure Databricks to send metrics to Azure Monitor.
- Deploy Grafana in a virtual machine. See Use dashboards to visualize Azure Databricks metrics.

The Grafana dashboard that is deployed includes a set of time-series visualizations. Each graph is time-series plot of metrics related to an Apache Spark job, the stages of the job, and tasks that make up each stage.

Azure Databricks performance overview

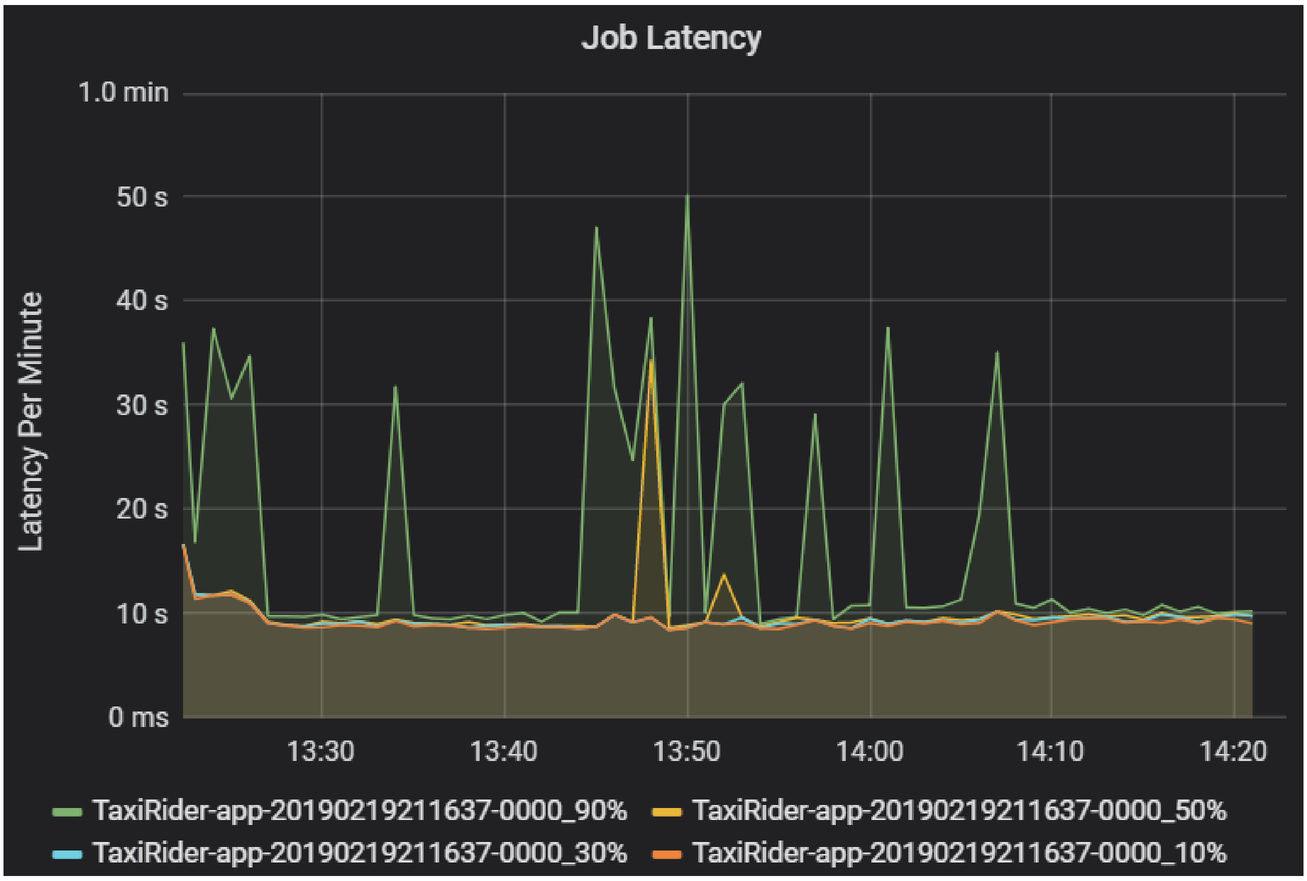
Azure Databricks is based on Apache Spark, a general-purpose distributed computing system. Application code, known as a **job**, executes on an Apache Spark cluster, coordinated by the cluster manager. In general, a job is the highest-level unit of computation. A job represents the complete operation performed by the Spark application. A typical operation includes reading data from a source, applying data transformations, and writing the results to storage or another destination.

Jobs are broken down into **stages**. The job advances through the stages sequentially, which means that later stages must wait for earlier stages to complete. Stages contain groups of identical **tasks** that can be executed in parallel on multiple nodes of the Spark cluster. Tasks are the most granular unit of execution taking place on a subset of the data.

The next sections describe some dashboard visualizations that are useful for performance troubleshooting.

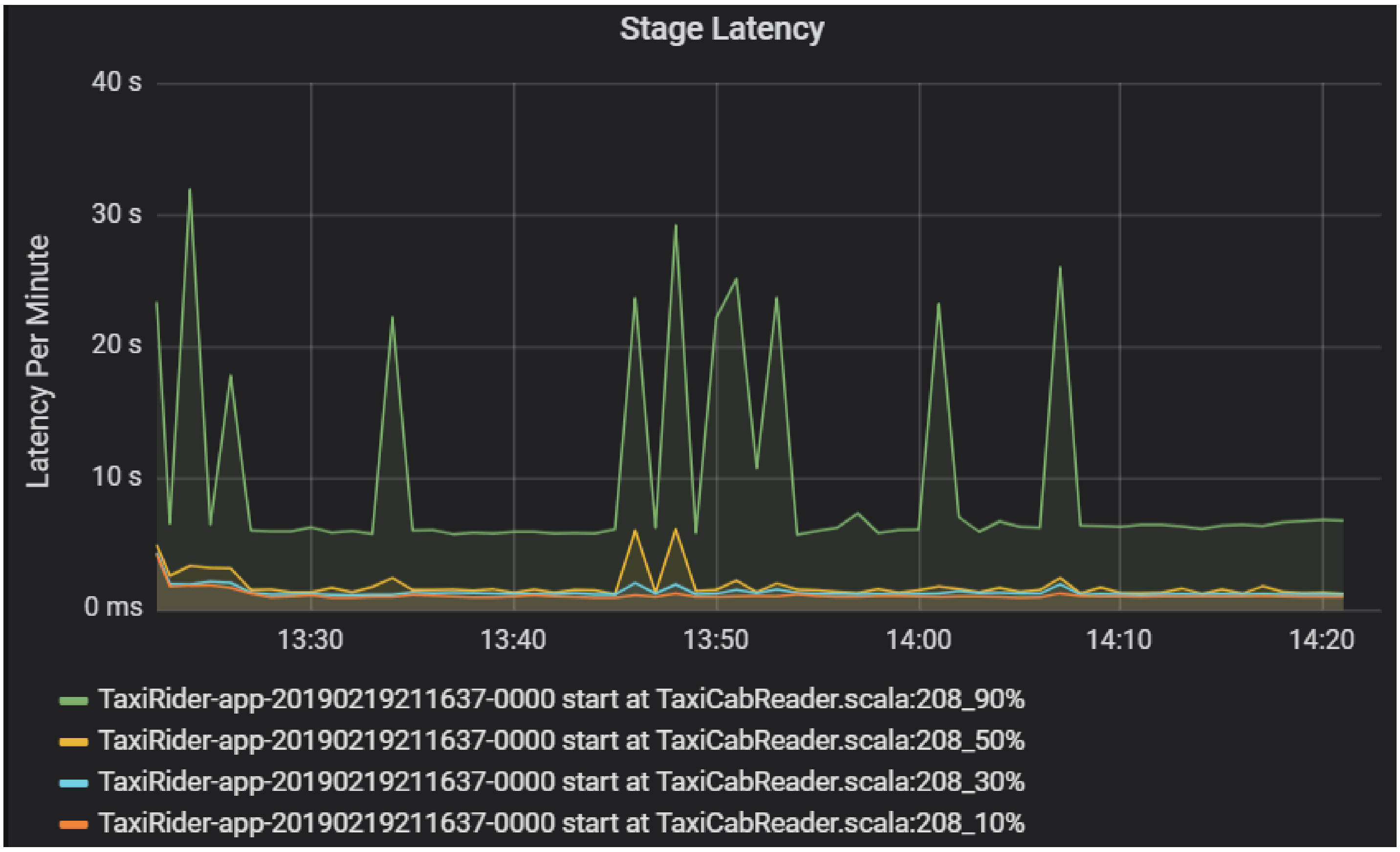
Job and stage latency

Job latency is the duration of a job execution from when it starts until it completes. It is shown as percentiles of a job execution per cluster and application ID, to allow the visualization of outliers. The following graph shows a job history where the 90th percentile reached 50 seconds, even though the 50th percentile was consistently around 10 seconds.

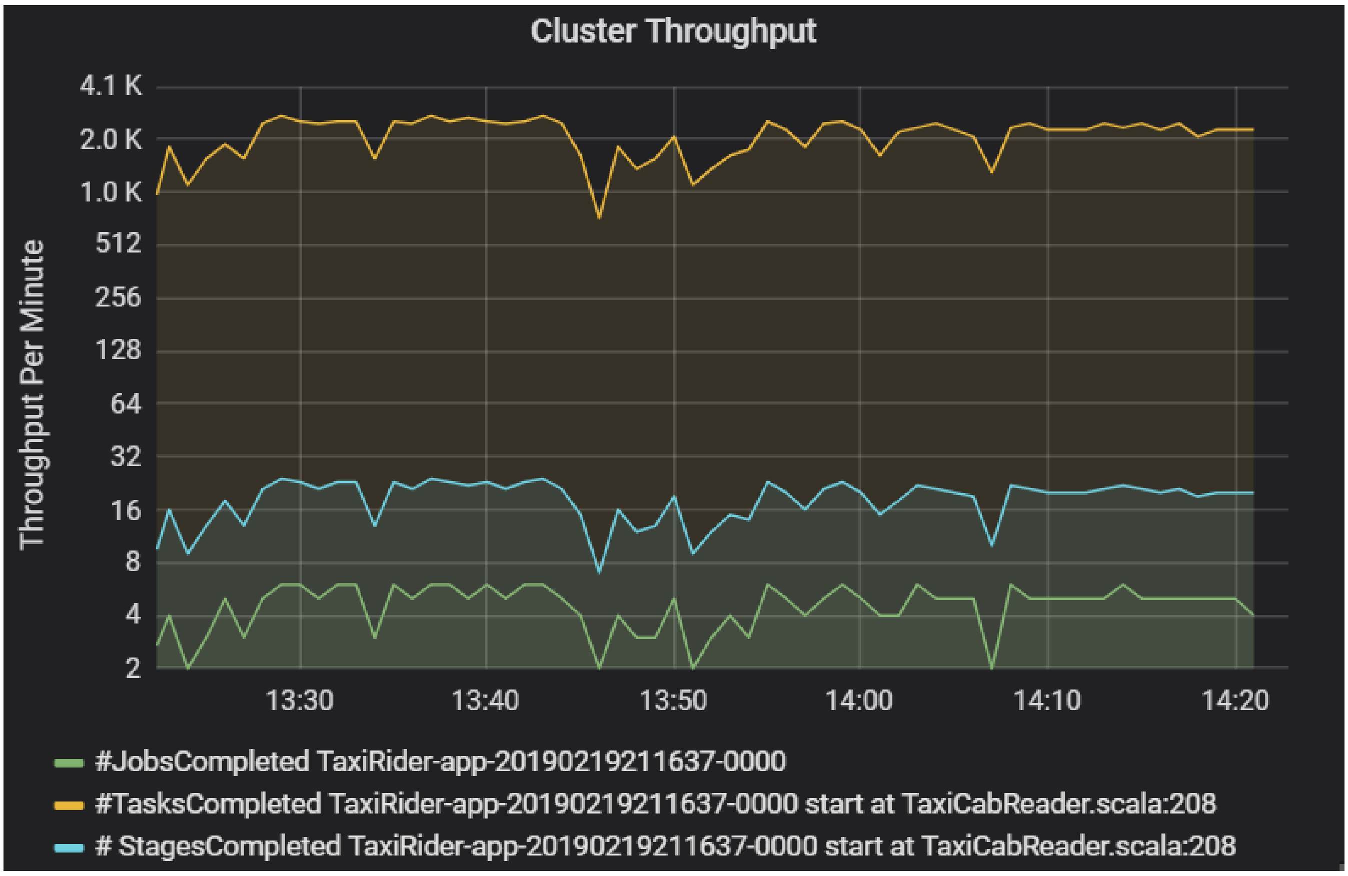


Investigate job execution by cluster and application, looking for spikes in latency. Once clusters and applications with high latency are identified, move on to investigate stage latency.

Stage latency is also shown as percentiles to allow the visualization of outliers. Stage latency is broken out by cluster, application, and stage name. Identify spikes in task latency in the graph to determine which tasks are holding back completion of the stage.

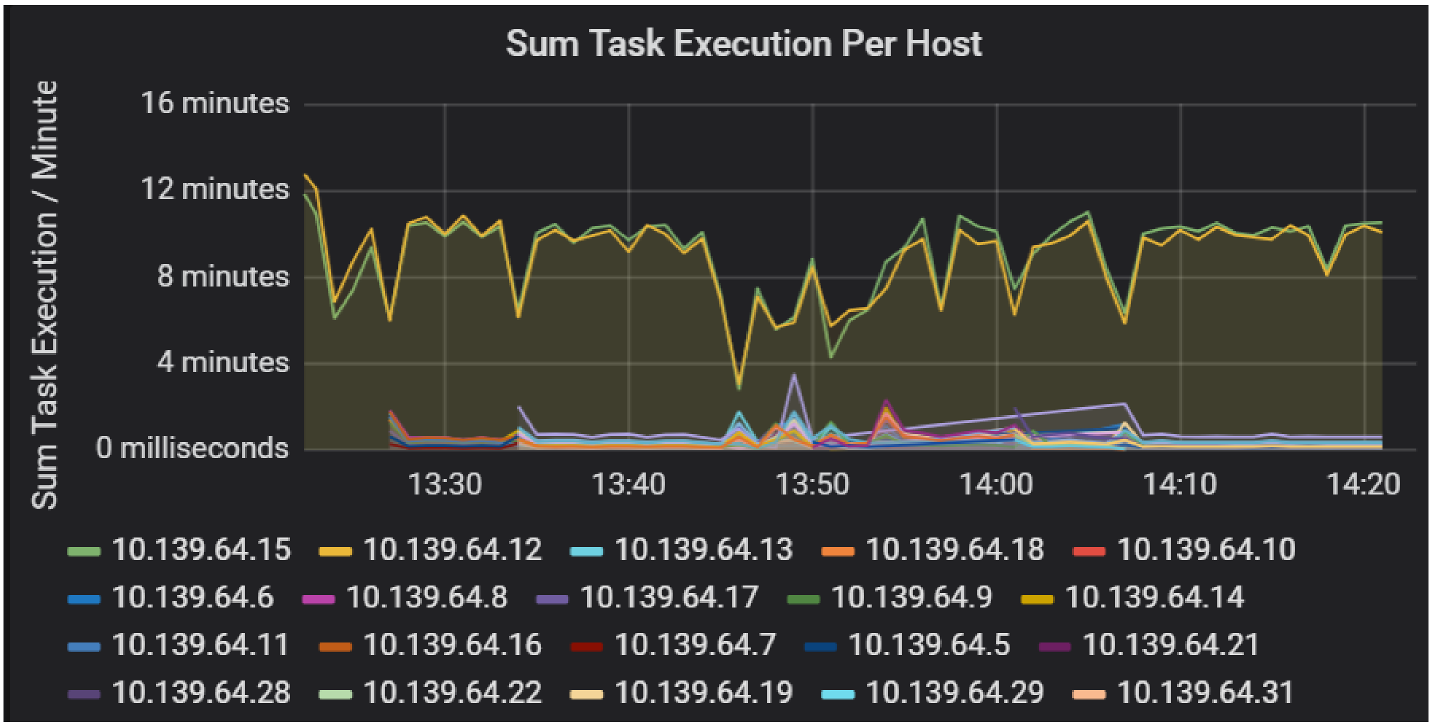


The cluster throughput graph shows the number of jobs, stages, and tasks completed per minute. This helps you to understand the workload in terms of the relative number of stages and tasks per job. Here you can see that the number of jobs per minute ranges between 2 and 6, while the number of stages is about 12 – 24 per minute.

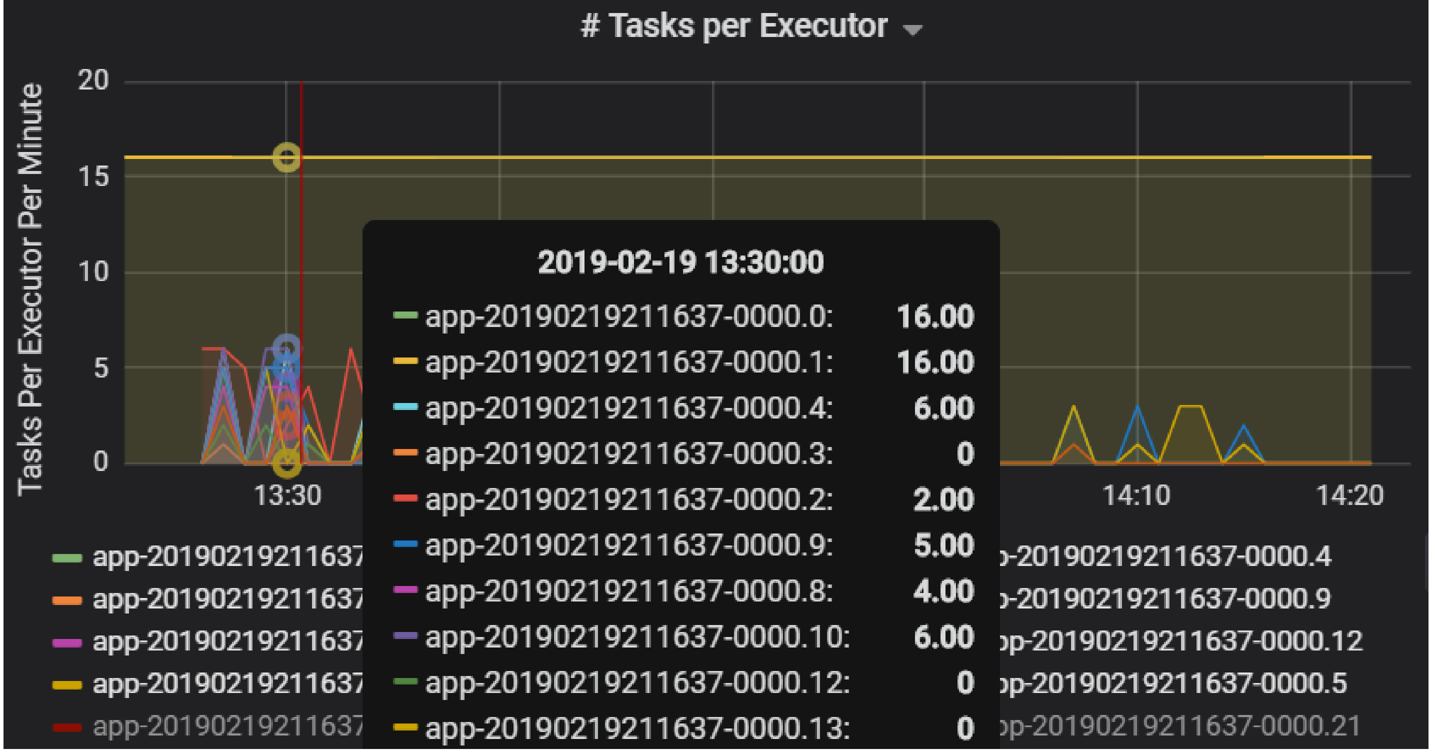


Sum of task execution latency

This visualization shows the sum of task execution latency per host running on a cluster. Use this graph to detect tasks that run slowly due to the host slowing down on a cluster, or a misallocation of tasks per executor. In the following graph, most of the hosts have a sum of about 30 seconds. However, two of the hosts have sums that hover around 10 minutes. Either the hosts are running slow or the number of tasks per executor is misallocated.



The number of tasks per executor shows that two executors are assigned a disproportionate number of tasks, causing a bottleneck.

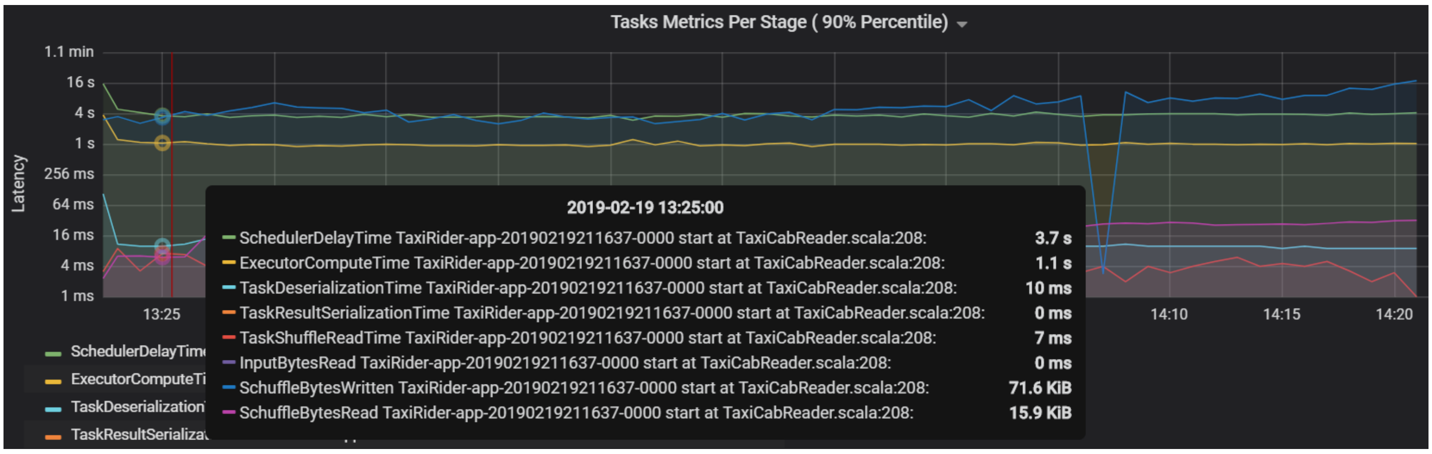


Task metrics per stage

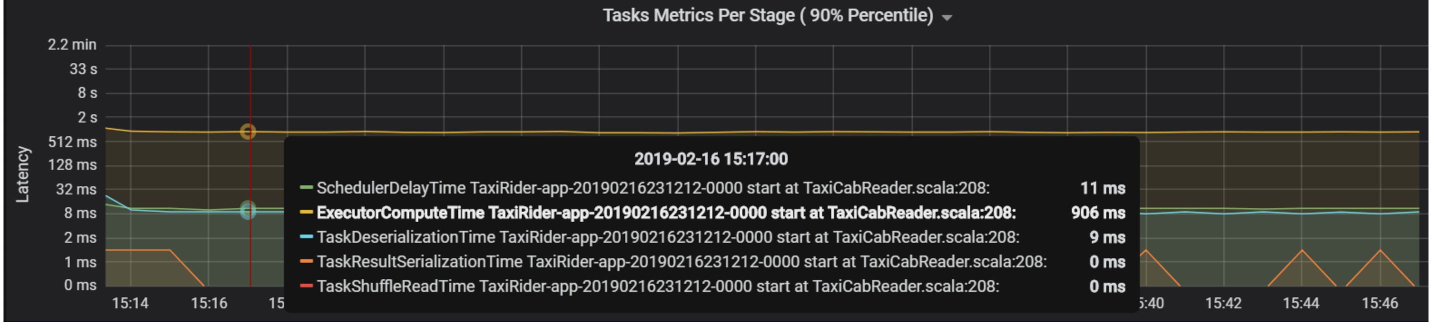
The task metrics visualization gives the cost breakdown for a task execution. You can use it see the relative time spent on tasks such as serialization and deserialization. This data might show opportunities to optimize — for example, by using broadcast variables to avoid shipping data. The task metrics also show the shuffle data size for a task, and the shuffle read and write times. If these values are high, it means that a lot of data is moving across the network.

Another task metric is the scheduler delay, which measures how long it takes to schedule a task. Ideally, this value should be low compared to the executor compute time, which is the time spent actually executing the task.

The following graph shows a scheduler delay time (3.7 s) that exceeds the executor compute time (1.1 s). That means more time is spent waiting for tasks to be scheduled than doing the actual work.



In this case, the problem was caused by having too many partitions, which caused a lot of overhead. Reducing the number of partitions lowered the scheduler delay time. The next graph shows that most of the time is spent executing the task.

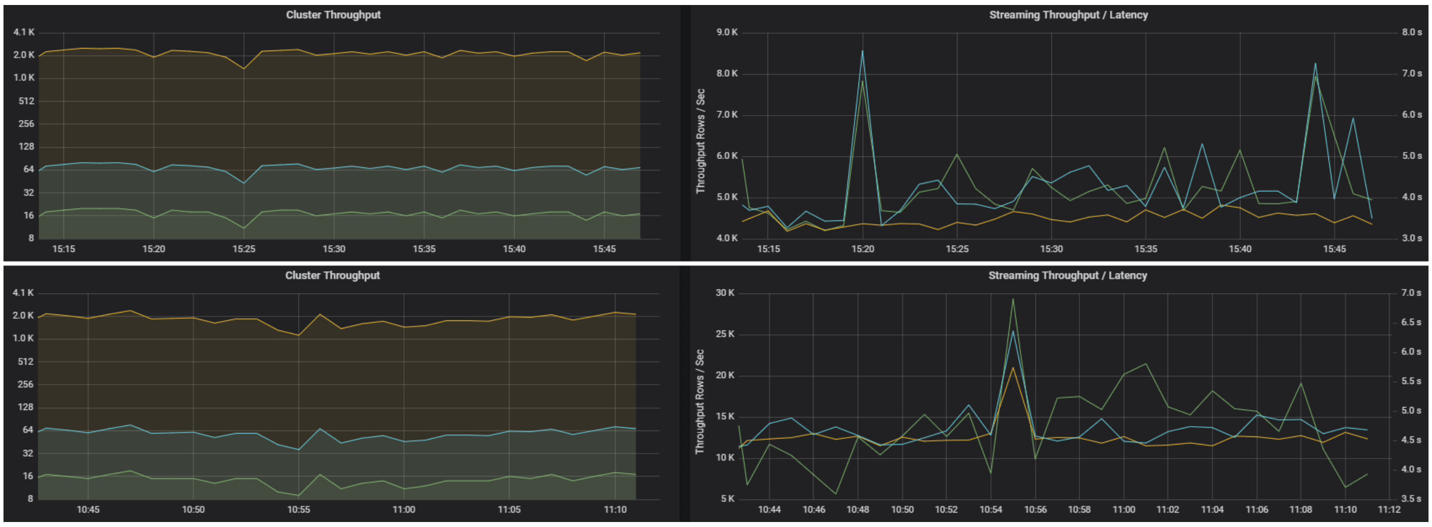


2019-02-16 15:17:00
SchedulerDelayTime TaxiRider-app-20190216231212-0000 start at TaxiCabReader.scala:208: 11 ms
ExecutorComputeTime TaxiRider-app-20190216231212-0000 start at TaxiCabReader.scala:208: 906 ms
TaskDeserializationTime TaxiRider-app-20190216231212-0000 start at TaxiCabReader.scala:208: 9 ms
TaskResultSerializationTime TaxiRider-app-20190216231212-0000 start at TaxiCabReader.scala:208: 0 ms
TaskShuffleReadTime TaxiRider-app-20190216231212-0000 start at TaxiCabReader.scala:208: 0 ms

Streaming throughput and latency

Streaming throughput is directly related to structured streaming. There are two important metrics associated with streaming throughput: Input rows per second and processed rows per second. If input rows per second outpaces processed rows per second, it means the stream processing system is falling behind. Also, if the input data comes from Event Hubs or Kafka, then input rows per second should keep up with the data ingestion rate at the front end.

Two jobs can have similar cluster throughput but very different streaming metrics. The following screenshot shows two different workloads. They are similar in terms of cluster throughput (jobs, stages, and tasks per minute). But the second run processes 12,000 rows/sec versus 4,000 rows/sec.



Streaming throughput is often a better business metric than cluster throughput, because it measures the number of data records that are processed.

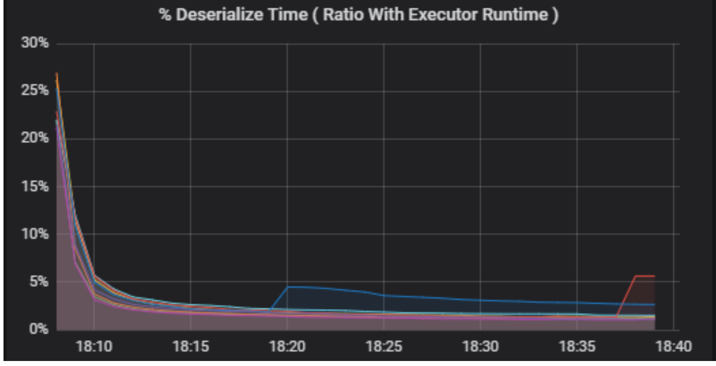
Resource consumption per executor

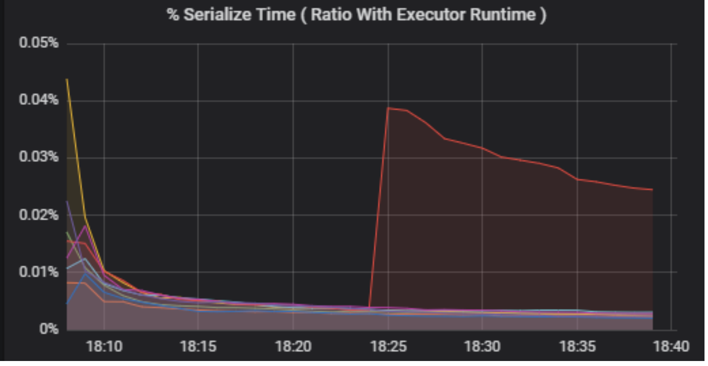
These metrics help to understand the work that each executor performs.

Percentage metrics measure how much time an executor spends on various things, expressed as a ratio of time spent versus the overall executor compute time. The metrics are:

- % Serialize time
- % Deserialize time
- % CPU executor time
- % JVM time

These visualizations show how much each of these metrics contributes to overall executor processing.





Shuffle metrics are metrics related to data shuffling across the executors.

- Shuffle I/O
- Shuffle memory
- File system usage
- Disk usage

Common performance bottlenecks

Two common performance bottlenecks in Spark are *task stragglers* and a *non-optimal shuffle partition count*.

Task stragglers

The stages in a job are executed sequentially, with earlier stages blocking later stages. If one task executes a shuffle partition more slowly than other tasks, all tasks in the cluster must wait for the slow task to catch up before the stage can end. This can happen for the following reasons:

1. A host or group of hosts are running slow. Symptoms: High task, stage, or job latency and low cluster throughput. The summation of tasks latencies per host won't be evenly distributed. However, resource consumption will be evenly distributed across executors.

2. Tasks have an expensive aggregation to execute (data skewing). Symptoms: High task, stage, or job and low cluster throughput, but the summation of latencies per host is evenly distributed. Resource consumption will be evenly distributed across executors.

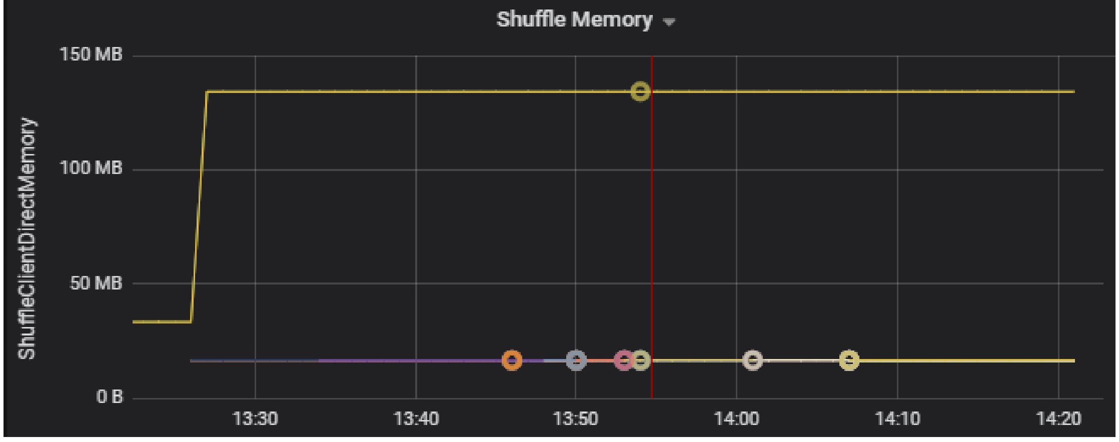
3. If partitions are of unequal size, a larger partition may cause unbalanced task execution (partition skewing). Symptoms: Executor resource consumption is high compared to other executors running on the cluster. All tasks running on that executor will run slow and hold the stage execution in the pipeline. Those stages are said to be *stage barriers*.

Non-optimal shuffle partition count

During a structured streaming query, the assignment of a task to an executor is a resource-intensive operation for the cluster. If the shuffle data isn't the optimal size, the amount of delay for a task will negatively impact throughput and latency. If there are too few partitions, the cores in the cluster will be underutilized which can result in processing inefficiency. Conversely, if there are too many partitions, there's a great deal of management overhead for a small number of tasks.

Use the resource consumption metrics to troubleshoot partition skewing and misallocation of executors on the cluster. If a partition is skewed, executor resources will be elevated in comparison to other executors running on the cluster.

For example, the following graph shows that the memory used by shuffling on the first two executors is 90X bigger than the other executors:


Shuffle Memory
ShuffleClientDirectMemory
150 MB
100 MB
50 MB
0 B
13:30
13:40
13:50
14:00
14:10
14:20