Big Data (CSGY-6513)

Fall 2022

Project Report

on

Google-Borg Traces Analysis

by

Atmaja Raman **ar6871**

Nobel Dang **nd2100**

Priyadarshan Vijay **pv2109**

1. Introduction

A Google cluster is a set of machines, packed into physical enclosures, and connected by a high-bandwidth cluster network. A cell is a set of machines, typically all in a single cluster, that share a common cluster-management system that allocates work to machines. Borg supports two kinds of resource request: a job (made up of one or more tasks), which describes computations that a user wants to run, and an alloc set, which describes a resource reservation that jobs can be run in. A task represents a Linux program, possibly consisting of multiple processes, to be run on a single machine. A job may specify an alloc set in which it must run, in which case each of its tasks will receive resources from an alloc instance of that alloc set. If a job doesn't specify an alloc set, its tasks will consume resources directly from a machine. A single usage trace describes several days of the workload on a single Borg cell. A trace is made up of several tables, each indexed by a primary key that typically includes a timestamp. The data in the tables is derived from information provided by the cell's management system and the individual machines in the cell.

Analyzing the Google cluster manager Borg dataset helps us understand about the resource usage and requirements of a trace. Analyses of job duration, duration for each event/state transition, distribution of jobs, resource requests in terms of CPU and memory usage, type of resource request: job or alloc_set, job loads at each cluster etc. will lead to some very helpful insights. These insights can help us draw useful conclusions about machines, jobs, tasks and resource usage and in turn would aid in better allocation and management of resources.

Our objective is to analyze Borg Cluster data to observe:

1.1. Distribution of jobs across clusters
1.2. Time taken to transition states.
1.3. Frequency of jobs failing/succeeding at the end of their life cycles.
1.4. Memory consumption of jobs across clusters
1.5. Predict request success or failure and event or state of trace using applicable features.

Why is this a Big Data problem?
Our dataset size is 2.1 TiB in compressed format and this just spans 1 month (May 2019). As the number of years increases, the scale of the data increases. Moreover, to perform any kind of analysis with machine learning with a larger number of features, we would need Big Data infrastructure.

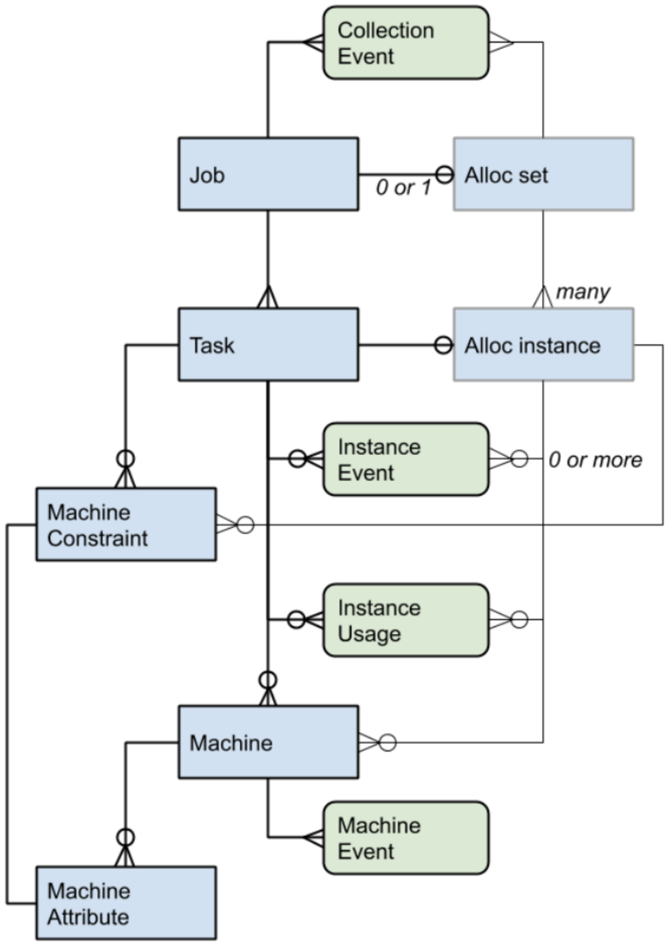
2. Methodology

2.1.Dataset Overview

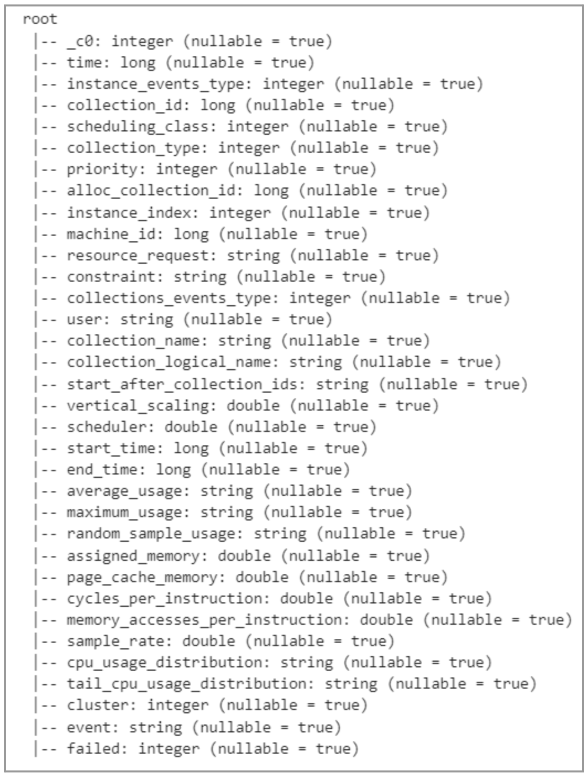
We referred to the traces from parts of the Google cluster management software and systems. The v3 of the dataset was captured in May 2019 and provides information about eight different borg cells. It is derived from information provided by the cell's management system and the individual machines in the cell. The total size of the dataset is 2.1TiB in compressed format. The dataset consisted of five different tables which could be joined together to provide useful records:

1. Machine events table
2. Machine Attributes table
3. Collection Events table
4. Instance events table
5. Instance usage table

The relation between these tables could be better understood by the ER diagram below:



We took a simpler version of this dataset for our use which focuses on individual events of each job and is much smaller in size to carry out the analysis and develop the techniques which can be applied to the entire dataset. The dataset used contains the following features:



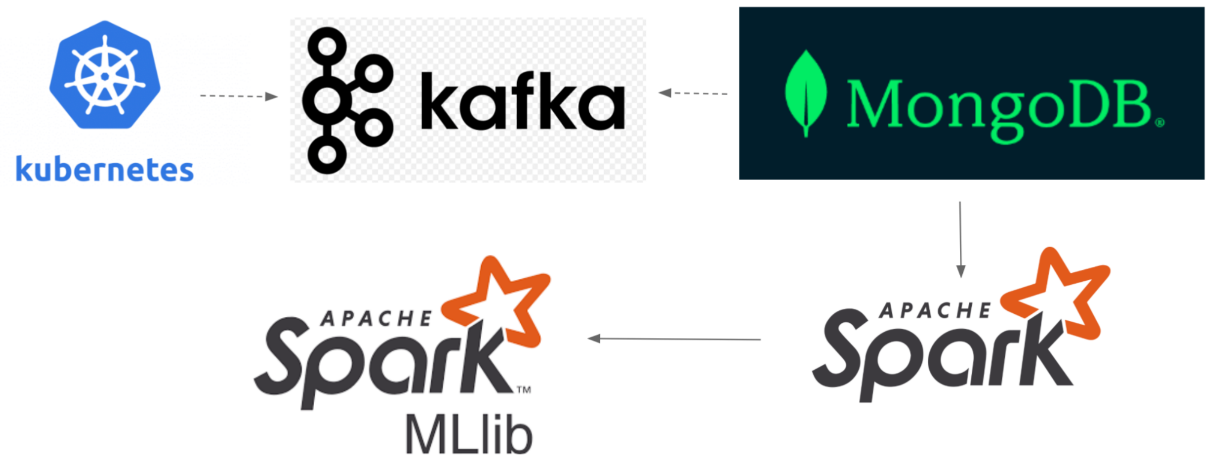
```
root
 |-- _c0: integer (nullable = true)
 |-- time: long (nullable = true)
 |-- instance_events_type: integer (nullable = true)
 |-- collection_id: long (nullable = true)
 |-- scheduling_class: integer (nullable = true)
 |-- collection_type: integer (nullable = true)
 |-- priority: integer (nullable = true)
 |-- alloc_collection_id: long (nullable = true)
 |-- instance_index: integer (nullable = true)
 |-- machine_id: long (nullable = true)
 |-- resource_request: string (nullable = true)
 |-- constraint: string (nullable = true)
 |-- collections_events_type: integer (nullable = true)
 |-- user: string (nullable = true)
 |-- collection_name: string (nullable = true)
 |-- collection_logical_name: string (nullable = true)
 |-- start_after_collection_ids: string (nullable = true)
 |-- vertical_scaling: double (nullable = true)
 |-- scheduler: double (nullable = true)
 |-- start_time: long (nullable = true)
 |-- end_time: long (nullable = true)
 |-- average_usage: string (nullable = true)
 |-- maximum_usage: string (nullable = true)
 |-- random_sample_usage: string (nullable = true)
 |-- assigned_memory: double (nullable = true)
 |-- page_cache_memory: double (nullable = true)
 |-- cycles_per_instruction: double (nullable = true)
 |-- memory_accesses_per_instruction: double (nullable = true)
 |-- sample_rate: double (nullable = true)
 |-- cpu_usage_distribution: string (nullable = true)
 |-- tail_cpu_usage_distribution: string (nullable = true)
 |-- cluster: integer (nullable = true)
 |-- event: string (nullable = true)
 |-- failed: integer (nullable = true)
```

2.2.Architecture

The architecture for the application should be able to support continuously generated trace data from a large number of clusters and events. The data is generated every second and in huge streams. We envision a data processing pipeline consisting of Borg (data generation source), Kafka with different topics for different types of data, MongoDB for storing data for later use, Spark and Spark ML for delivering real-time predictions and insights.

Borg can use any gRPC to publish data to Kafka topic, which can be consumed by MongoDB and stored in a specific collection based on the topic from which the data was read.

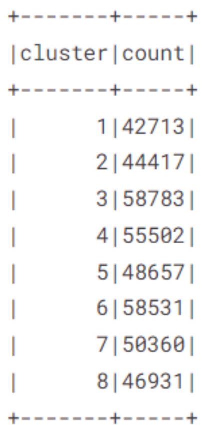
Now, as and when data is stored in a new collection, we can make use of mongo streams and events. Using readStream() and spark-connector we can push the data to our spark job in real time and can utilize spark mlLib there to generate the predictions and insights in real time.



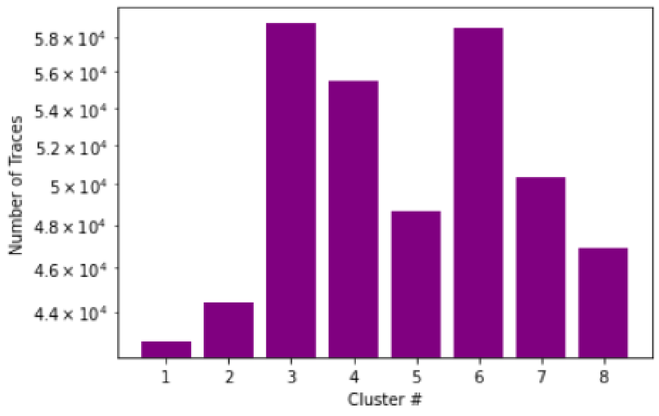
2.3.Analysis and Insights

2.3.1 Trace distribution across clusters

A cluster is a set of machines connected by a high bandwidth cluster network. There are a total of eight clusters present in the network. We are trying to find the distribution of traces across the eight clusters by grouping the traces by clusters. From the plot and table we know that the number of traces are highest in clusters 3 and 4 and it is the least in cluster 1.



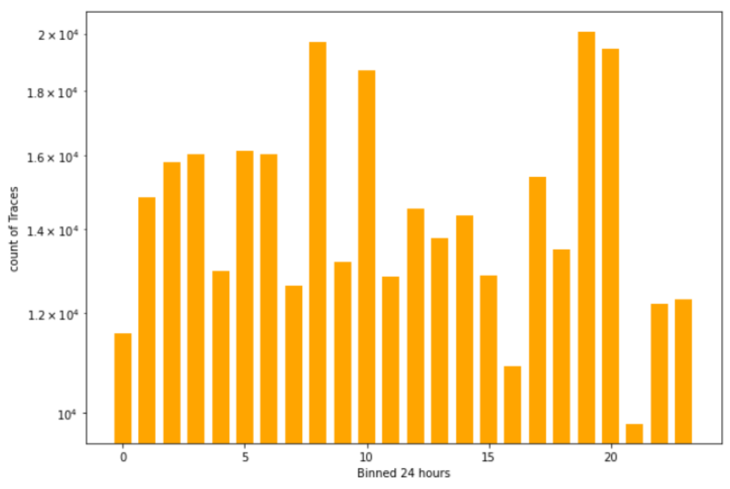
```
+-------+-----+
|cluster|count|
+-------+-----+
|      1|42713|
|      2|44417|
|      3|58783|
|      4|55502|
|      5|48657|
|      6|58531|
|      7|50360|
|      8|46931|
+-------+-----+
```



2.3.2 Trace requests by the hour

Finding the hours in the day when trace requests are high and the hours when trace requests are lower would help in better resource allocation and management. In order to find this, we subtract the offset of 600 seconds from time, convert the time to hours from milliseconds, bin it into 24 hours, and count the number of traces in each hour bin. From the table and plot below, we can see that the number of traces is more in the 8th,10th hour (morning) and also in the 19th and 20th hour (evening) of the day.

```
+-------+-----+
|hourBin|count|      |     11|12818|
+-------+-----+      |     12|14512|
|      0|11562|      |     13|13751|
|      1|14826|      |     14|14337|
|      2|15796|      |     15|12851|
|      3|16031|      |     16|10878|
|      4|12963|      |     17|15404|
|      5|16145|      |     18|13477|
|      6|16033|      |     19|20083|
|      7|12618|      |     20|19445|
|      8|19711|      |     21| 9807|
|      9|13180|      |     22|12212|
|     10|18690|      |     23|12312|
```



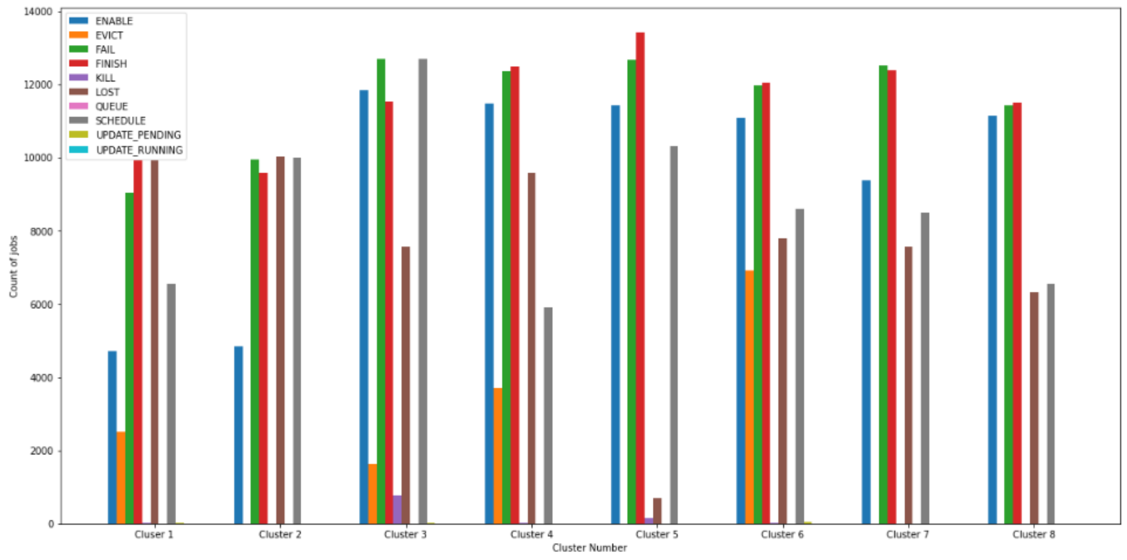
2.3.3 Distribution of different trace events across clusters

There are basically two types of events: ones that affect the scheduling state, and ones that reflect state changes of a task. There are eleven different possible values across these two types of events as follows:

- Submit - when a task was submitted to the cluster manager.
- Queue - a task is queued until the scheduler is ready to act on it.
- Enable - a task became eligible for scheduling.
- Schedule - a task was scheduled on a machine.
- Evict - a task was descheduled because of a higher priority task, because the scheduler over-scheduled
- Fail - a task was descheduled because of a failure or used more than requested memory.
- Finish - a task completed normally.
- Kill - a task was canceled due to failure or exit of dependent job.
- Lost - a task was ended, but a record indicating its termination was missing.

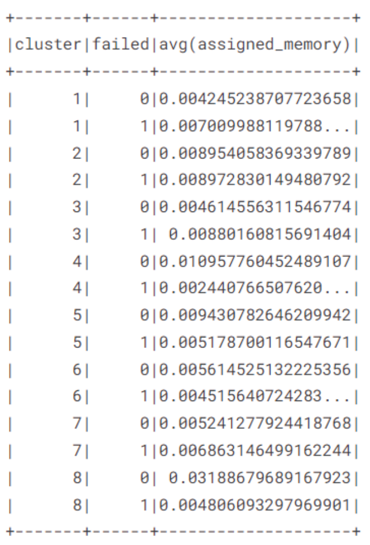
- Update_pending - a task's scheduling class, resource requirements, or constraints were updated while it was waiting to be scheduled.
- Update_running - a task's scheduling class, resource requirements, or constraints were updated while it was running.

We find the distribution of these events across the clusters and plot the results using a grouped bar chart where each bar represents an event, and they are grouped by the cluster, and bar length represents the number of traces. Comparing the number of finished and failed tasks, we can see clusters 5 and 1 are more successful. In cluster 3 and 2, the number of failed jobs are higher. In all other clusters have a comparable number of finished and failed jobs

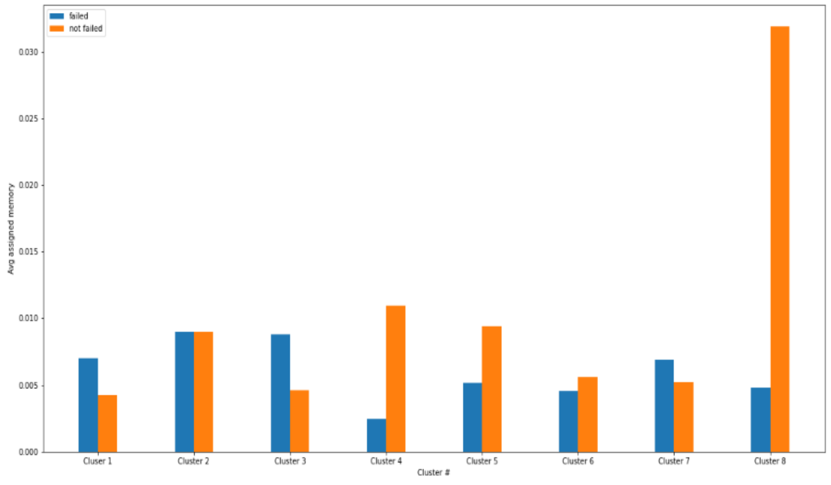


2.3.4 Average assigned memory for successful and failed tasks by cluster

The assigned memory is the average memory limit for an instance given to the OS kernel by the Borglet. To find the assigned memory for successful and failed tasks of each cluster, we group the traces by cluster and failed and get the count. The table and graph below show that the assigned memory per successful request is higher for clusters 8, 4, and 5. The assigned memory per failed request is higher for clusters 1,3, and 7, and the assigned memory per failed or successful request is the same for cluster 2.

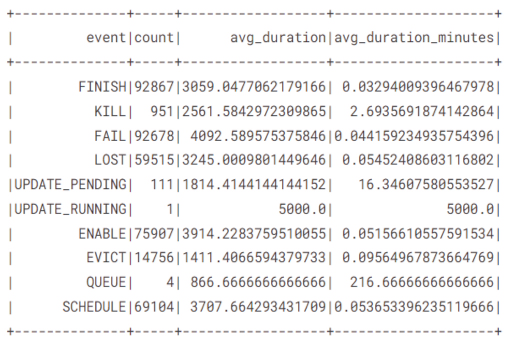


```
+-------+------+--------------------+
|cluster|failed|avg(assigned_memory)|
+-------+------+--------------------+
|      1|     0|0.004245238707723658|
|      1|     1|0.007009988119788...|
|      2|     0|0.008954058369339789|
|      2|     1|0.008972830149480792|
|      3|     0|0.004614556311546774|
|      3|     1| 0.00880160815691404|
|      4|     0|0.010957760452489107|
|      4|     1|0.002440766507620...|
|      5|     0|0.009430782646209942|
|      5|     1|0.005178700116547671|
|      6|     0|0.005614525132225356|
|      6|     1|0.004515640724283...|
|      7|     0|0.005241277924418768|
|      7|     1|0.006863146499162244|
|      8|     0| 0.03188679689167923|
|      8|     1|0.004806093297969901|
+-------+------+--------------------+
```

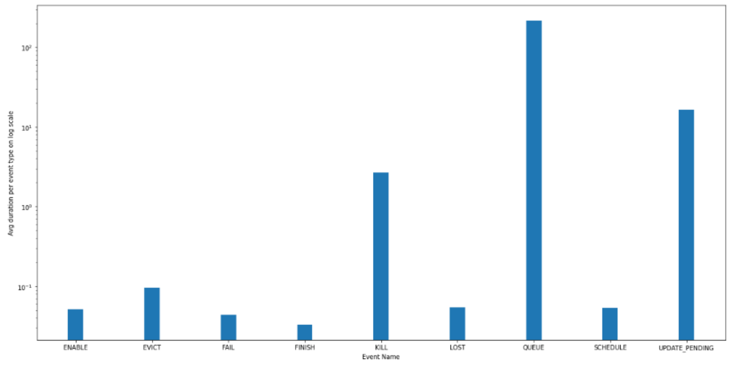


2.3.5 Average time taken for each event

In order to find the average duration of each event, we grouped the traces by event and found the average time in minutes. We also displayed the count of traces for each event for a fair comparison. The average duration in minutes per job is highest for update running, followed by queue event. Update running is the event classes for jobs whose resource requirements and constraints are being updated while the job is running. As expected, the average time per job is higher for killed jobs as well. The average duration in minutes per job is the least for finish event.

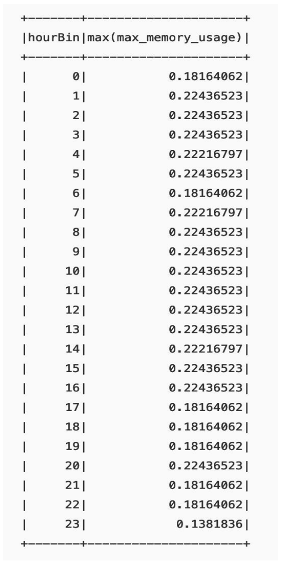


```
+-------------+-----+-----------------+-------------------+
|        event|count|     avg_duration|avg_duration_minutes|
+-------------+-----+-----------------+-------------------+
|       FINISH|92867|3059.0477062179166| 0.03294009396467978|
|         KILL|  951|2561.5842972309865|  2.6935691874142864|
|         FAIL|92678| 4092.589575375846|0.044159234935754396|
|         LOST|59515|3245.0009801449646| 0.05452408603116802|
|UPDATE_PENDING|  111|1814.4144144144152|   16.34607580553527|
|UPDATE_RUNNING|    1|            5000.0|              5000.0|
|       ENABLE|75907|3914.2283759510055| 0.05156610557591534|
|        EVICT|14756|1411.4066594379733| 0.09564967873664769|
|        QUEUE|    4| 866.6666666666666|  216.66666666666666|
|     SCHEDULE|69104| 3707.664293431709|0.053653396235119666|
+-------------+-----+-----------------+-------------------+
```

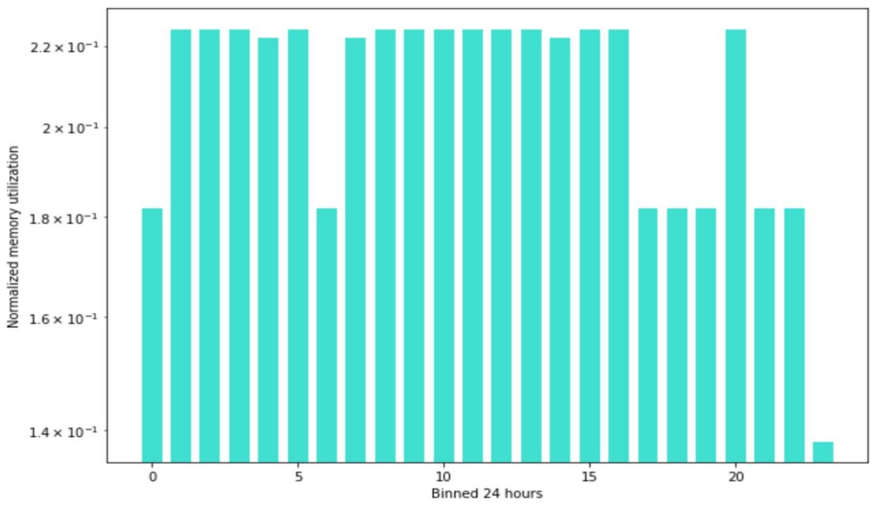


2.3.6 Memory utilization with time

Maximum memory usage is the largest observed memory usage during the window. Here we split time into 24-hour bins and then find the peak memory utilization during that time period. From the plot below, we see that memory is most utilized during the day till 4 pm in the evening, and at 8 pm also it peaks after which until midnight it is less.

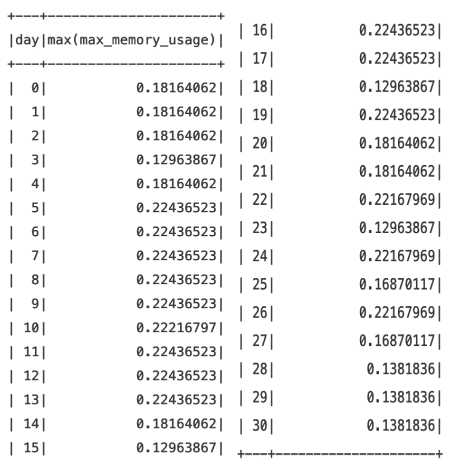


```
+-------+---------------------+
|hourBin|max(max_memory_usage)|
+-------+---------------------+
|      0|           0.18164062|
|      1|           0.22436523|
|      2|           0.22436523|
|      3|           0.22436523|
|      4|           0.22216797|
|      5|           0.22436523|
|      6|           0.18164062|
|      7|           0.22216797|
|      8|           0.22436523|
|      9|           0.22436523|
|     10|           0.22436523|
|     11|           0.22436523|
|     12|           0.22436523|
|     13|           0.22436523|
|     14|           0.22216797|
|     15|           0.22436523|
|     16|           0.22436523|
|     17|           0.18164062|
|     18|           0.18164062|
|     19|           0.18164062|
|     20|           0.22436523|
|     21|           0.18164062|
|     22|           0.18164062|
|     23|            0.1381836|
+-------+---------------------+
```

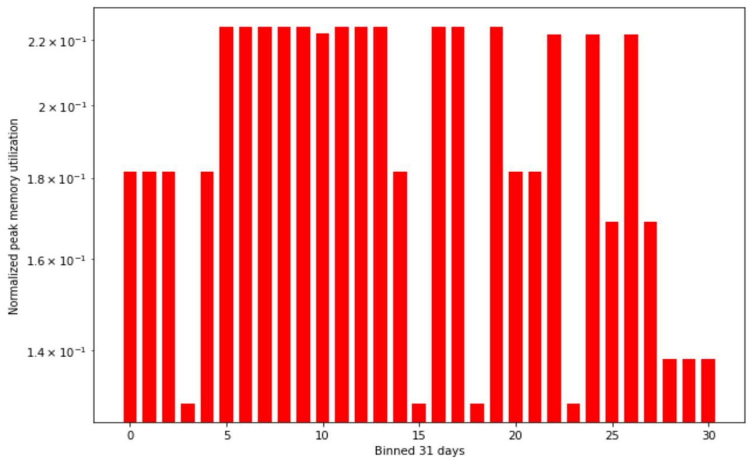


2.3.7 Memory utilization by day

Similar to the analysis above we bin the time by days of the month and find the maximum memory usage by day. We find that memory utilization is higher during the weekdays rather than the weekends.

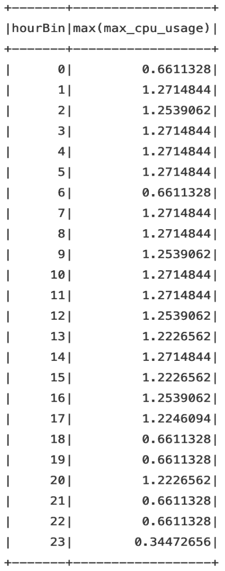


```
+---+---------------------+   | 16|           0.22436523|
|day|max(max_memory_usage)|   | 17|           0.22436523|
+---+---------------------+   | 18|           0.12963867|
|  0|           0.18164062|   | 19|           0.22436523|
|  1|           0.18164062|   | 20|           0.18164062|
|  2|           0.18164062|   | 21|           0.18164062|
|  3|           0.12963867|   | 22|           0.22167969|
|  4|           0.18164062|   | 23|           0.12963867|
|  5|           0.22436523|   | 24|           0.22167969|
|  6|           0.22436523|   | 25|           0.16870117|
|  7|           0.22436523|   | 26|           0.22167969|
|  8|           0.22436523|   | 27|           0.16870117|
|  9|           0.22436523|   | 28|            0.1381836|
| 10|           0.22216797|   | 29|            0.1381836|
| 11|           0.22436523|   | 30|            0.1381836|
| 12|           0.22436523|   +---+---------------------+
| 13|           0.22436523|
| 14|           0.18164062|
| 15|           0.12963867|
```

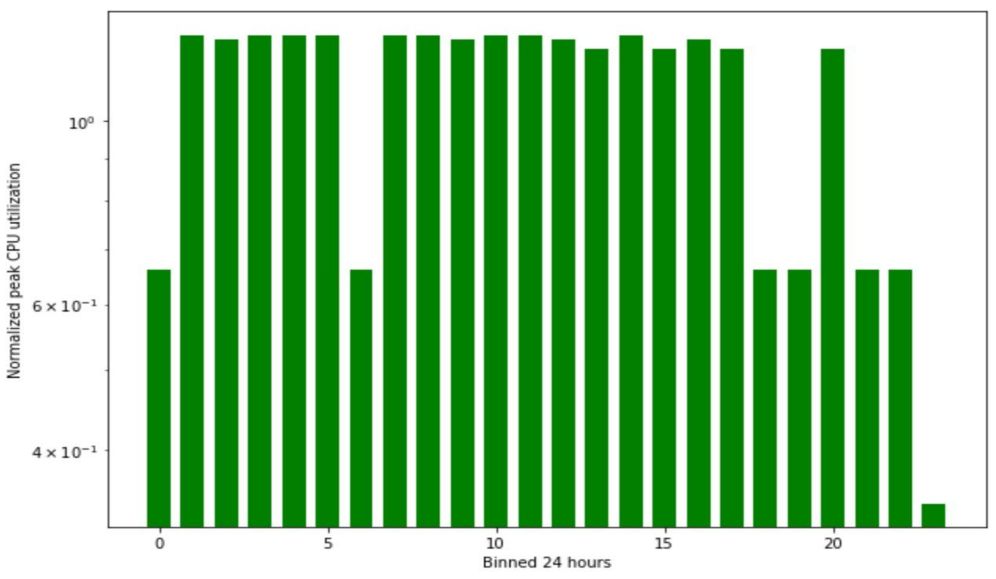


2.3.8 CPU peak utilization with time

Maximum memory usage is the largest observed CPU usage during the window. Here we split time into 24-hour bins and then find the peak CPU utilization during that time period. From the plot below, we see that the CPU is most utilized during the day till 5 pm in the evening, and at 8pm also it peaks after which until midnight it is less.

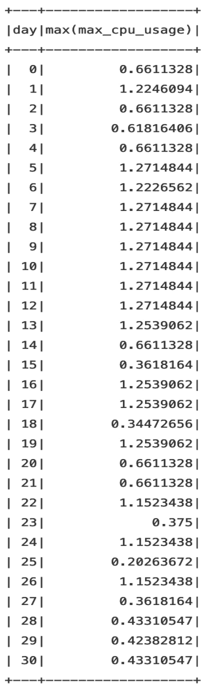


```
+-------+-----------------+
|hourBin|max(max_cpu_usage)|
+-------+-----------------+
|      0|        0.6611328|
|      1|        1.2714844|
|      2|        1.2539062|
|      3|        1.2714844|
|      4|        1.2714844|
|      5|        1.2714844|
|      6|        0.6611328|
|      7|        1.2714844|
|      8|        1.2714844|
|      9|        1.2539062|
|     10|        1.2714844|
|     11|        1.2714844|
|     12|        1.2539062|
|     13|        1.2226562|
|     14|        1.2714844|
|     15|        1.2226562|
|     16|        1.2539062|
|     17|        1.2246094|
|     18|        0.6611328|
|     19|        0.6611328|
|     20|        1.2226562|
|     21|        0.6611328|
|     22|        0.6611328|
|     23|       0.34472656|
+-------+-----------------+
```

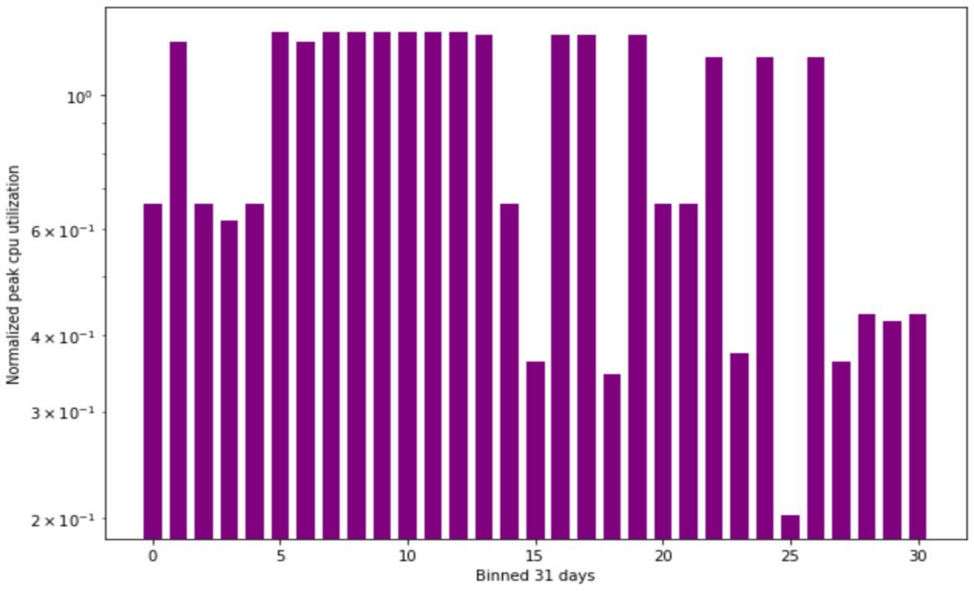


2.3.9 CPU peak utilization by day

Similar to the analysis above we bin the time by days of the month and find the maximum CPU usage by day. We find that the CPU utilization is higher during the weekdays rather than the weekends.

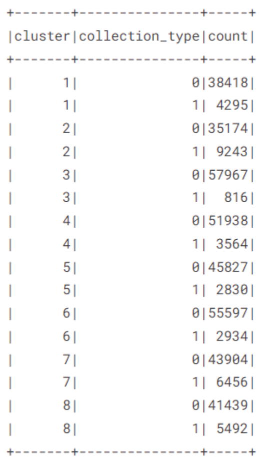


```
+---+-----------------+
|day|max(max_cpu_usage)|
+---+-----------------+
|  0|        0.6611328|
|  1|        1.2246094|
|  2|        0.6611328|
|  3|       0.61816406|
|  4|        0.6611328|
|  5|        1.2714844|
|  6|        1.2226562|
|  7|        1.2714844|
|  8|        1.2714844|
|  9|        1.2714844|
| 10|        1.2714844|
| 11|        1.2714844|
| 12|        1.2714844|
| 13|        1.2539062|
| 14|        0.6611328|
| 15|        0.3618164|
| 16|        1.2539062|
| 17|        1.2539062|
| 18|       0.34472656|
| 19|        1.2539062|
| 20|        0.6611328|
| 21|        0.6611328|
| 22|        1.1523438|
| 23|            0.375|
| 24|        1.1523438|
| 25|       0.20263672|
| 26|        1.1523438|
| 27|        0.3618164|
| 28|       0.43310547|
| 29|       0.42382812|
| 30|       0.43310547|
+---+-----------------+
```



2.3.10 Distribution of requests based on collection type across clusters

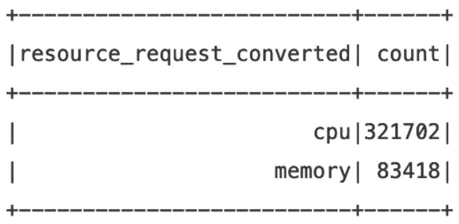
Borg supports two kinds of resource requests, a job which has one or more tasks, which describes computations that a user wants to run, and an alloc set which is made up of one or more allocs, or alloc instances, which describes a resource reservation that jobs can be run in. A job may specify an alloc set in which it must run, in which case each of its tasks will get resources from an alloc instance of that alloc set. If a job doesn't specify an alloc set, its tasks will get resources directly from a machine. Here collection types refer to either jobs or alloc_sets. Finding the number of job or alloc_set requests by cluster helps with resource allocation. We group by cluster and collection_type and find count, and here 0 represents a job, and 1 represents alloc_set. From our results, we notice that the job requests on all the clusters are mostly for the collection type job.



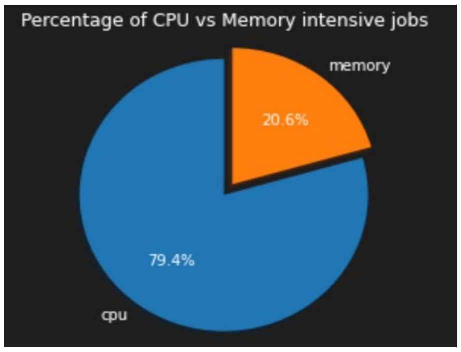
```
+-------+---------------+-----+
|cluster|collection_type|count|
+-------+---------------+-----+
|      1|              0|38418|
|      1|              1| 4295|
|      2|              0|35174|
|      2|              1| 9243|
|      3|              0|57967|
|      3|              1|  816|
|      4|              0|51938|
|      4|              1| 3564|
|      5|              0|45827|
|      5|              1| 2830|
|      6|              0|55597|
|      6|              1| 2934|
|      7|              0|43904|
|      7|              1| 6456|
|      8|              0|41439|
|      8|              1| 5492|
+-------+---------------+-----+
```

2.3.11 CPU vs Memory based requests

Knowing if the traces are more CPU or memory-intensive aids in the management of said CPU and memory resources and allocation. From the plot and table below we can see that most requests are CPU intensive and more requests are present under CPU.



```
+--------------------------+------+
|resource_request_converted| count|
+--------------------------+------+
|                       cpu|321702|
|                    memory| 83418|
+--------------------------+------+
```



2.4.Machine Learning Model

2.4.1 Data Preprocessing

Data preprocessing involves assessing the quality of data, and cleaning and transforming the data so that the machine learning model learns well. This involves steps like handling missing data, handling data type mismatch, scaling or normalizing, etc.

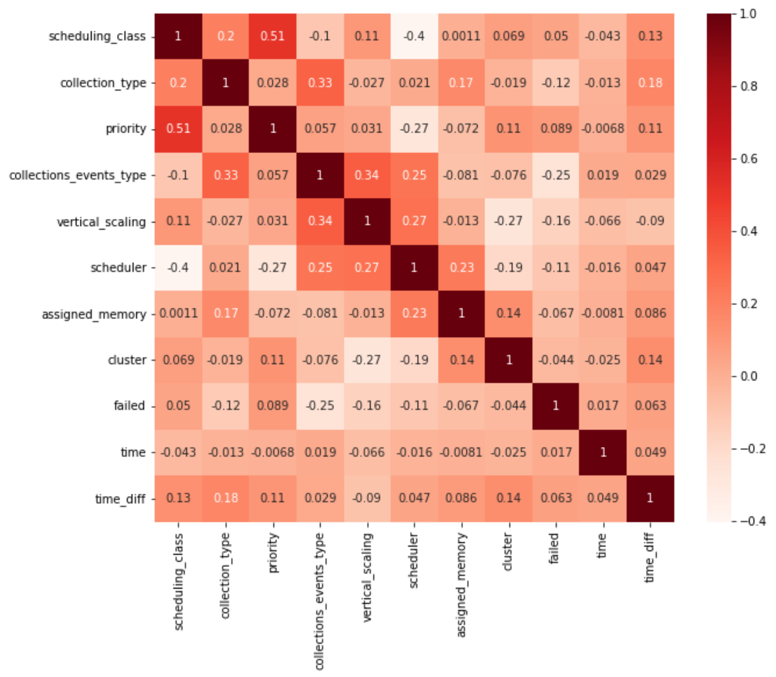
The following preprocessing was done on our dataset before inputting it to the model:

- Spark Dataframe columns (time_diff, assigned_memory, and priority) were scaled between 0 to 1
- Encoding categorical one hot.(vertical_scaling, scheduling_class, scheduler, collection_type, binned time, cluster)
- Correlation between variables was also performed to check dependent column features.
- The offset of 600s in time was removed and time was binned to 24 hours.

2.4.2 Binary Classification

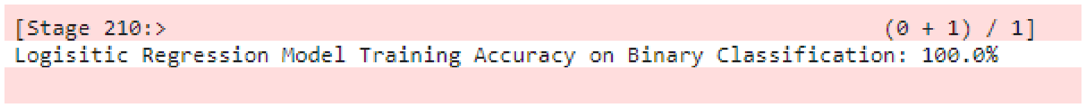
In this we aim to predict if a trace of a request from a cluster would fail or succeed. So our target column attribute is “failed,” and a value of 1 means that the request will fail whereas 0 means that it will not fail.

After data preprocessing, the correlation matrix was plotted and observed to check the dependency of features on our target variable and on each other. We observed that none of the variables are highly correlated to each other, and it seems that they are independent.

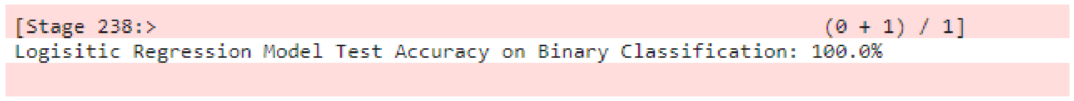


For this problem, we are using an Apache Pyspark **logistic regression** model from pyspark.ml.classification API. Following this, we split out data into **0.75:0.25 ratio** of train and test data respectively.

After training, our model was able to achieve 100 percent of accuracy on training and testing.



```
[Stage 210:>                                                        (0 + 1) / 1]
Logisitic Regression Model Training Accuracy on Binary Classification: 100.0%
```

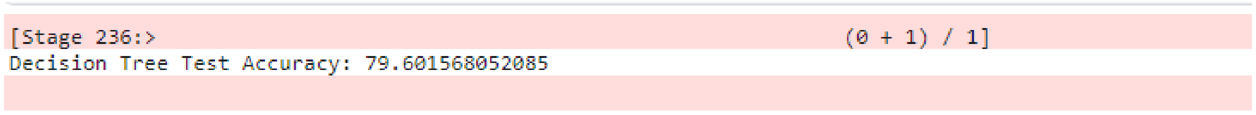


```
[Stage 238:>                                                        (0 + 1) / 1]
Logisitic Regression Model Test Accuracy on Binary Classification: 100.0%
```

2.4.3 Multiclass Classification

In Multiclass classification, we aim to predict that given a trace of request from the cluster which state of its life cycle is the request most likely to be in. For this problem statement, we used **Decision Tree Classifier** from pyspark.ml.classification API. To train and test our model, we split our dataset into **0.75:0.25 ratio** of train and test data respectively.

On the test dataset, our decision tree classifier was able to achieve around **80% accuracy.**



```
[Stage 236:>                                                        (0 + 1) / 1]
Decision Tree Test Accuracy: 79.601568052085
```

We were able to generate these metrics from pyspark.ml.evaluation package's MulticlassClassificationEvaluator and BinaryClassificationEvaluator.

One thing that we observed that pyspark's machine learning model does differently from the machine learning model's fit on pandas is that the machine learning models in pyspark API take in a column that is of vector type where each vector consists of multiple feature values and a target variable column.

3. Future Work

1) The work done in the analysis of Borg cluster data can be very beneficial in reducing the overall cost of operating data centers while improving the efficiency of cluster-based operations.

2) The analysis can help in choosing the correct/most successful Hardware SKUs based on the completion/failure rate. This data can be further analyzed to optimize the cost of the operations team in big data centers depending on the future/anticipated resource load and job failure rates.