Analyzing System Tables in Databricks: A Final Project Report

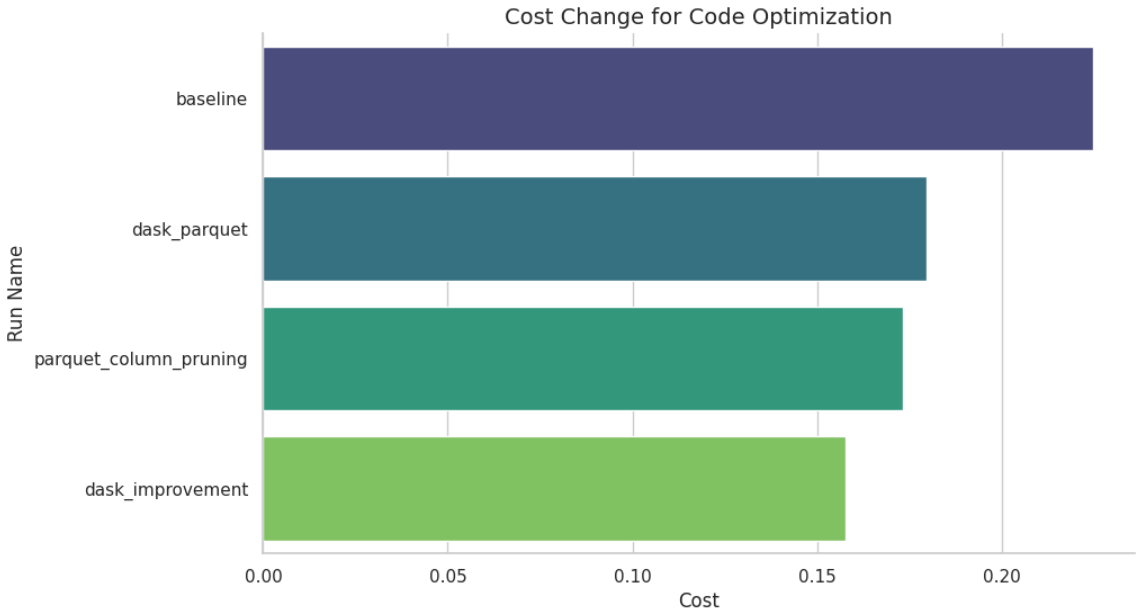
This report outlines the key elements of my final project, which focused on exploring system tables within Databricks. The project's primary objective was to enable and experiment with system tables to better understand their utility in analyzing operational efficiency and costs.

Baseline Job and Data Preparation

The baseline job for this project involved using the Pandas package to perform a group by operation on a large dataset, approximately 5GB in size. This dataset, which is publicly available, was first downloaded to my local environment and then uploaded to my S3 bucket. In Databricks, I employed a mount operation to facilitate its accessibility and use in subsequent processes.

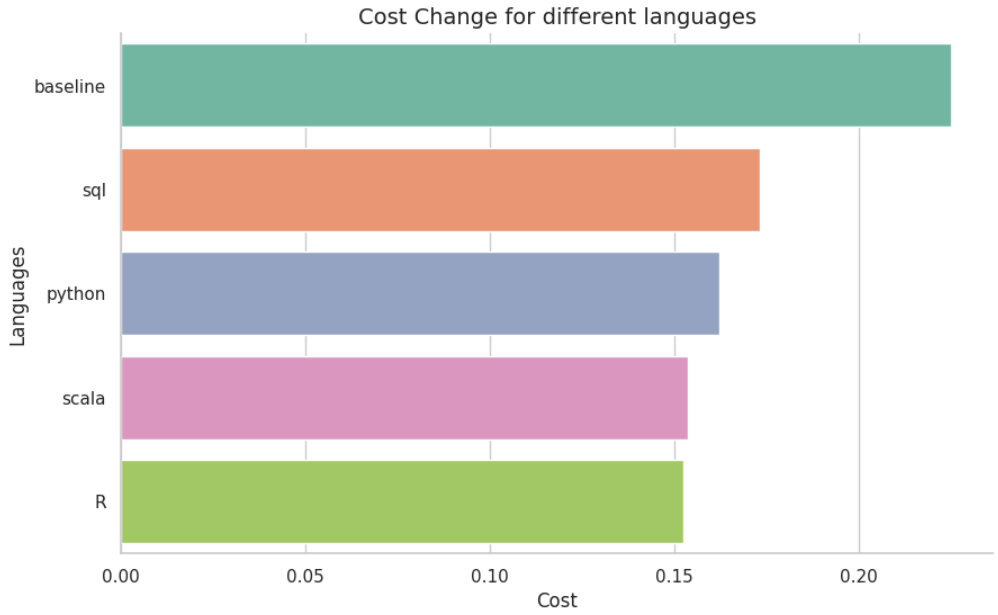
Part I Code Optimization and Comparative Analysis

The first part of my experiment involved code optimization to compare the performance enhancements achieved through various coding approaches. I experimented with the Dask package and partitioned the dataset, as well as converting it into different packet formats, to evaluate the running efficiency.



Part II Experimenting with Different Programming Languages

The second part of my experiment in Databricks involved using different programming languages, namely Python, SQL, R, and Scala, to drive Spark data processing operations. The aim was to compare the efficiency of operations across these different languages.

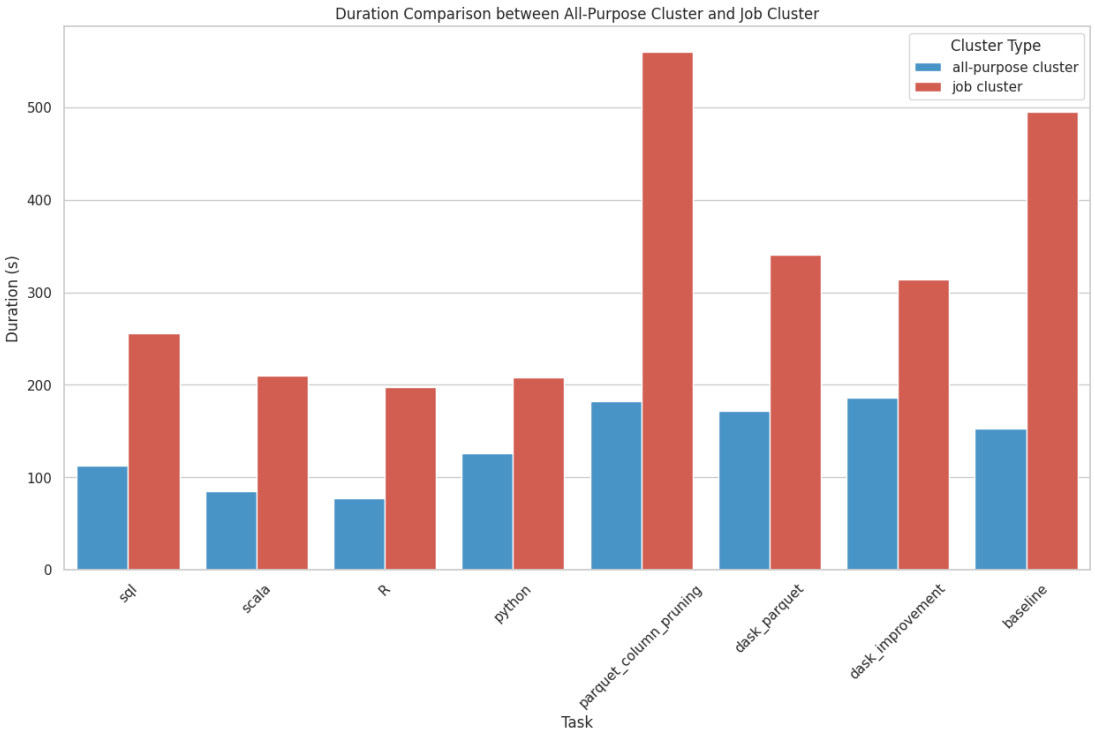


Part III: Comparing Cluster Performance in Databricks

In the third part of my project, I compared the performance of two types of clusters in Databricks: the all-purpose cluster and the job cluster. For this comparison, I ran the various tasks mentioned earlier on both types of clusters.

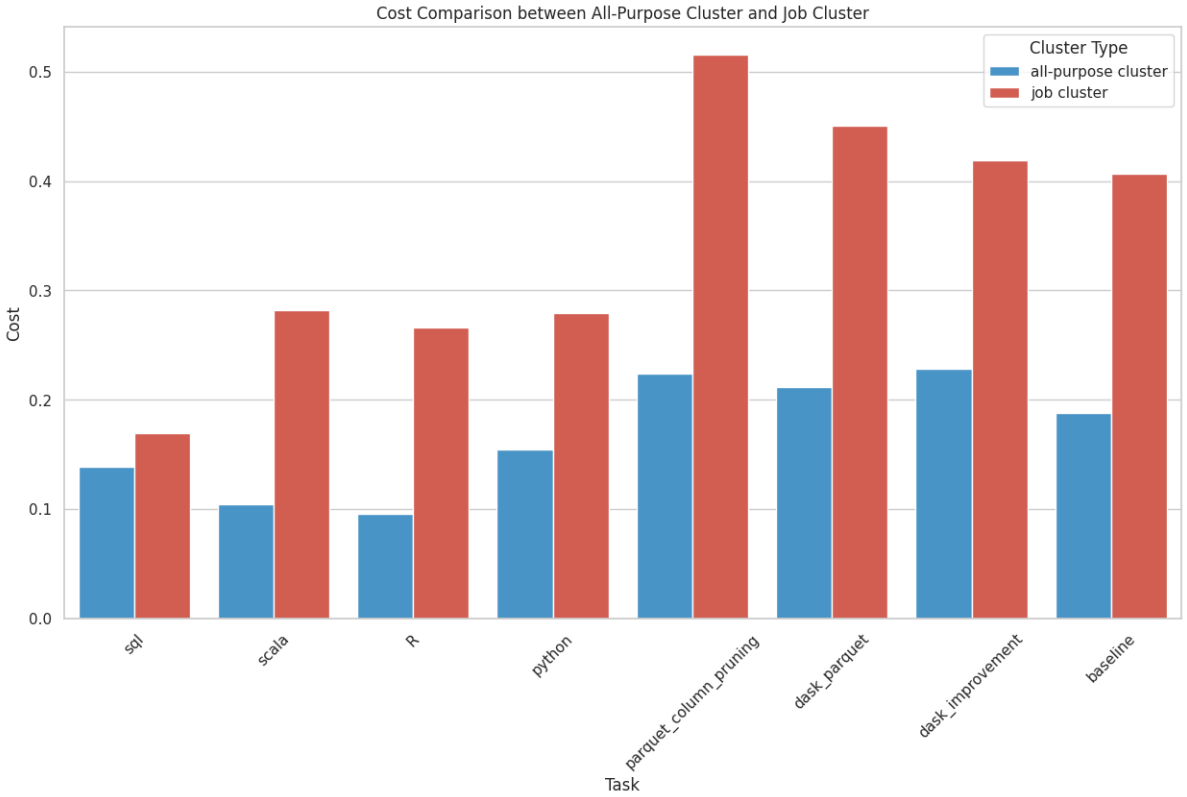
Findings on Cluster Performance

The initial findings indicated that the all-purpose job cluster had a higher operational efficiency with shorter execution times. However, a further examination revealed that while the job cluster took longer, most of the time was spent in the 'pending' phase, which involved preparing the cluster. This delay is attributed to the job cluster not being always on standby, unlike the all-purpose cluster, which is ready to operate at any time. Consequently, the actual execution time of the job cluster was similar to that of the all-purpose cluster.



Cost Analysis and Interesting Observations

An interesting phenomenon I observed was that despite the lower per-unit cost of the job cluster, its overall expenditure was higher. I hypothesized that this could be due to the initialization cost of the job cluster, especially since many small tasks require initialization, leading to higher overall costs.



Comprehensive Experiment on Task Aggregation

To investigate further, I conducted a comprehensive experiment where I combined eight small tasks into one large task and executed it on both the all-purpose and job clusters for comparison. The results aligned with my expectations: in scenarios involving large, consolidated tasks, the job cluster was more cost-effective.

	sku_name	avg(usage_quantity)	Cost
1	PREMIUM_JOBS_COMPUTE_(PHOTON)	8.4561889444444450000000	1.268428
2	PREMIUM_ALL_PURPOSE_COMPUTE_(PHOTON)	3.3428450000000000000000	1.838565

Conclusion

Based on these findings, I conclude that the all-purpose cluster is more suitable for regular, interactive applications. In contrast, the job cluster is more economical for large, pre-defined computational tasks, particularly when the task volume and computational requirements are significantly high. And code optimizations and correct languages are worth taking into account.