Towards Plan-aware Resource Allocation in Serverless Query Processing

Malay Bag
malayb@microsoft.com
Microsoft

Alekh Jindal
alekh.jindal@microsoft.com
Microsoft

Hiren Patel
hirenp@microsoft.com
Microsoft

Abstract

Resource allocation for serverless query processing is a challenge. Unfortunately, prior approaches have treated queries as black boxes, thereby missing significant resource optimization opportunities. In this paper, we propose a plan-aware resource allocation approach where the resources are adaptively allocated based on the runtime characteristics of the query plan. We show the savings opportunity from such an allocation scheme over production SCOPE workloads at Microsoft. We present our current implementation of a greedy version that periodically estimates the peak resource for the remaining of the query as the query execution progresses. Our experimental evaluation shows that such an implementation could already save more than 8% resource usage over one of our production virtual clusters. We conclude by opening the discussion on various strategies for plan-aware resource allocation and their implications on the cloud computing stack.

1 Introduction

A new breed of *serverless* query processing is becoming increasing popular, where the system takes care of automatically provisioning the resources for analytical queries, e.g., Athena [2], Big Query [6], SCOPE [4]. This is attractive for the users since they do not have to worry about resource allocation and they pay-as-you-go at the query level, however, it is highly challenging for the cloud providers due to multiple reasons. First of all, it is very hard to estimate the fine-grained resource requirements for each query at compile time. This is because the relationship between resource and query performance is non-trivial, even for expert users [9]. The problem gets worse with the lack of accurate statistics over massive inputs sets in big data. Second, the resource requirements change over the course of query execution. For instance, analytical queries often have large scans and data shuffles early on, thereby requiring much more scale out in early parts of the query plan. The question then is whether to allocate for the peak, or the average, or something else. Finally, analytical

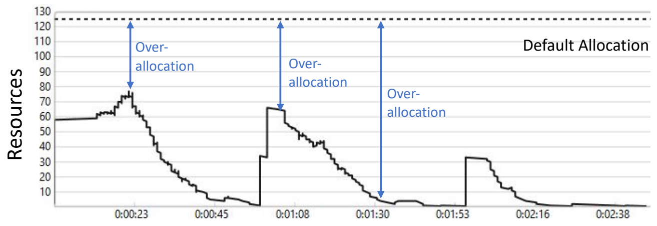


Figure 1: Resource usage in a typical SCOPE job over time.

queries are often long running and therefore over-allocation of resources could lead to significant inefficiencies in terms of resource overall utilization.

To illustrate, consider the SCOPE big data analytics platform at Microsoft, where the users specify the declarative SCOPE queries and the system takes care of running it in a massively distributed environment. The SCOPE query processor relies on a user-specified resource limit, i.e., the maximum number of containers (or *tokens*) that a query could use, and reserves them as guaranteed resources before starting the query execution. In case a query needs more resources, SCOPE can opportunistically use spare tokens from the cluster and thus improve the overall resource utilization [3]. However, the guaranteed resources once allocated are blocked for the entire duration of the SCOPE job. For instance, Figure 1 shows the token usage skyline in a typical SCOPE job. The dotted line shows the user-specified tokens while the solid line shows the actual tokens used over time. The gap between the two is indicated as over-allocation. We make the following key observations:

- The user-specified resource limit is more than 50% off than the actual peak resource. Indeed, it is very *hard for users to get the resource-limit right*.
- The actual resource consumption is not a straight line but rather a curve, with multiple peaks, and therefore a *fixed resource allocation may not be the best strategy*.
- The resource requirements peaks early on, with successively smaller peaks later on. Thus, the *query needs progressively less resources over time*.

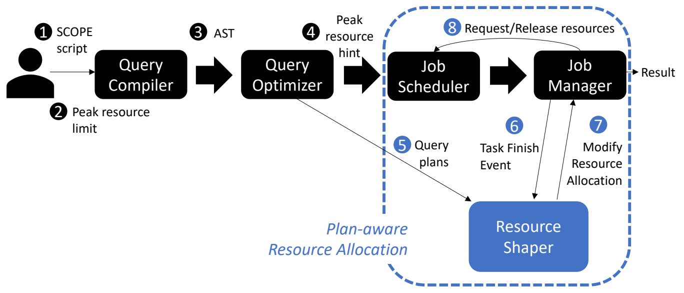


Figure 2: Plan-aware resource allocation highlighted in blue.

- Considering the end-to-end job execution, the area under the curved line is much smaller than the area under the dotted line, indicating *significant gains possible*.

In the rest of the paper, we first argue that the above challenges are rooted in the fact that current approaches to resource allocation in big data environments are black-box in nature. We then propose to adopt a plan-aware approach where the resource manager is aware of the fine-grained structure (and resource needs) of the query plan, and present potential saving opportunities from the SCOPE workloads at Microsoft. As a first step towards plan-aware resource allocation, we present a greedy resource shaping algorithm for adaptively adjusting the resource allocation as the job execution progresses. We show an experimental evaluation of this scheme over our production SCOPE workloads. Finally, we seek discussion and feedback on several open topics.

2 Prior Art: Black Box Resource Allocation

We saw from Figure 1 that current resource allocation in SCOPE does not consider the query plans and rather considers a SCOPE job as a black box. Furthermore, the user-specified token counts aims to provision the peak resources that remains fixed throughout the job execution. Given the large gap between allocated and actual resource consumption, efficient resource provisioning has received a lot of attention in the context of cloud workloads. In particular, for big data systems, the typical approach is to determine the resource requirements of a job based on past history of resource consumption, e.g., Morpheus [8]. Alternate approach is to perform sample runs and build models using them, e.g., Ernest [11]. However, these approaches still treat jobs as black box models without considering the shape of the job or how the resources requirements change in different stages of the job (or even over time).

Other works on resource modeling and optimization builds resource cost models which could be then used to pick resources for a given query plan, e.g., Perforator [9]. However, Perforator [9] still relies on sample runs and is limited to static allocation and do not adapt over the course of job execution. Jockey [5] considers dynamic re-allocation of resources, however, they employ much heavier simulator that is very expensive to execute at runtime. As a result, Jockey creates an

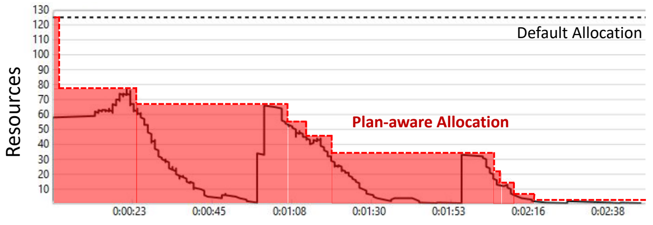


Figure 3: An instance of plan-aware resource allocation over the same typical SCOPE job as shown in Figure 1.

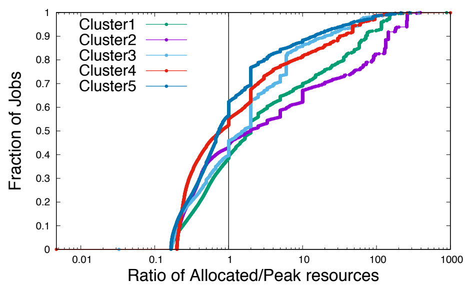
estimator based on previous runs, much similar to Morpheus, which the system can use to re-allocate resources. Instead, one could use a very lightweight simulator that uses cost estimates of each stage from the query optimizer and only replays the vertex scheduling strategy. In case the cost estimates are inaccurate, we could fix them separately using learned cost models [10]. Finally, Raqo [12] considers finding the optimal resources for each operator in the query plan as part of query optimization, however, it misses the transformation of operators into a stage graph and how cost varies with varying resources on this stage graph.

In summary, prior art considered analytical queries as black boxes, thereby missing the opportunity to tune resource allocation to the fine-grained behavior of the queries over time.

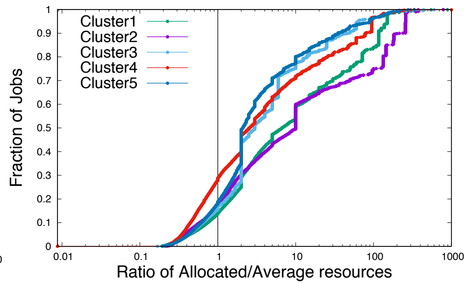
3 Plan-aware Resource Allocation

We now describe our plan-aware resource allocation approach. Figure 2 depicts the scheme for a typical big data processing system. The key addition here is a *resource shaper* component that adapts the resources to the query plan characteristics over time. To do this, the job manager invokes the resource shaper at periodic intervals, e.g., whenever a task finishes (Step 6). The resource shaper considers the query plan and the tasks finished so far to provide a possibly modified resource allocation (Step 7). Finally, the job manager coordinates with the job scheduler to either request or to release resources (Step 8). Note that the resource shaper could run in-process within the job manager, to handle dynamic changes to the execution graph, and the query plans could be passed on to the job manager anyways, thereby eliminating Step 5.

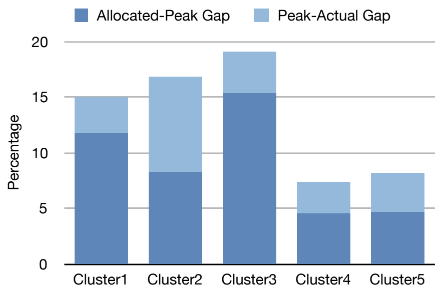
Figure 3 shows an instance of plan-aware resource allocation where the resource shaper simply re-computes the new peak for the remainder of the query plan as the query execution progresses. At any time, if the newer computed allocation is lower than the current one, the system releases the excess resources. We can see that compared to the default allocation, plan-aware allocation has significantly lower *resource usage*, defined as the area under the resource allocation curve. Furthermore, these savings come without sacrificing the query performance. Finally, the above resources optimizations are completely automatic and transparent for the users.



(a) CDF of Allocated Vs Peak Resources



(b) CDF of Allocated Vs Average Resources



(c) Allocation gap for recurring jobs

Figure 4: Resource optimization opportunity from different production clusters at Microsoft.

3.1 Saving Opportunities

We now present an analysis of production SCOPE workloads at Microsoft. SCOPE job consists of stages that are connected in a directed acyclic graph (DAG) with the data flow being from top to bottom. Each stage further contains one or more physical operators that could be processed locally in a single container, and instances of a stage (called vertices) can process different partitions of data in parallel. For the sake of presentation, we limit our discussion to maximum degree of parallelism (also referred to as *tokens* in SCOPE) as the unit of resource, however, one could easily extend it to other dimensions such as container size, VM type, etc. As mentioned earlier in Section 1, SCOPE jobs relies on a user provided maximum number of tokens to reserve (the allocated resources) before the job starts executing. Therefore, we first analyze the effectiveness of this resource allocation. Figure 4a shows the cumulative distribution of the ratio of the allocated and the peak consumed resources of SCOPE jobs from a single day in five different production clusters at Microsoft, comprising of 10s of thousands of jobs from each of the clusters. The vertical black line shows the ideal ratio when allocated resources are equal to the peak resources. Interestingly, we see from Figure 4a that 40% to 60% of the jobs are over-allocated in different clusters, by as much as $1000x$, indicating significant opportunities for down-sizing the resource allocation. The curve for $Cluster5$ is the best since the business unit on that cluster spent considerable effort to build client-side tools for resource allocation. However, $Cluster5$ still suffers from over-allocation in 40% of the jobs, with more than 10 times over-allocation in 15% of the jobs. Figure 4b shows the cumulative allocated and average resource consumption in each job. Now we see that 70% to 85% jobs are over-allocated with respect to their average resource consumption. This proves that there is a significant gap between the peak and the average resource consumption, and an adaptive allocation strategy could indeed be helpful. Finally, Figure 4c shows the aggregate over-allocation opportunity on each of the clusters. We break down the over-allocation into two components: (i) the gap between the allocated and the peak resource usage in each job, and (ii) the gap between the peak and the actual resource usage. Here, the resource usage is defined as the area under the resource curve. We can see from Figure 4c that the gap between allocated and peak resources ranges from $5-10\%$, while the gap between peak and actual resource usage adds another $3-9\%$. Together, they represent $8-19\%$ resource efficiency opportunity for $40-60\%$ of the workload, translating to tens of millions of dollars in operations costs in the exabyte-scale SCOPE infrastructure.

Reducing the above identified over-allocation has several implications. First of all, it improves the operational efficiency in a highly valuable business that powers big data analytics across the whole of Microsoft, including products such as Office, Windows, Bing, Xbox, etc. Second, it frees up guaranteed resources that could be used to submit more SCOPE jobs. Third, it reduces the queuing time of jobs by having them request for less resources. And finally, it improves the user experience by automating a mandatory parameter in SCOPE jobs. Note that apart from reducing the over-allocation, one could also increase the allocation for significantly under-allocated jobs. This could not only improve the job performance but also make job performance more predictable, since right-sizing the allocation reduces the dependence on opportunistic resource allocation [3].

4 Greedy Resource Shaper

We now describe a greedy implementation of resource shaper that dynamically shapes the resource allocation based on the query execution graph, as illustrated in Figure 3. The idea is to estimate the peak resource usage in the remaining of the job execution, and release any excess resources. We have purposefully chosen to be more conservative by only releasing resources, rather than being more aggressive in following the resource curve and both releasing and requesting resources. This is because releasing resources is a more lightweight operation without incurring the request overheads on the resource manager or the queuing overheads on the job execution. It is therefore pragmatic to passively inform the resource manager of the spare resources which could be recycled at any time. To detect the peak for the remaining query, we present a novel

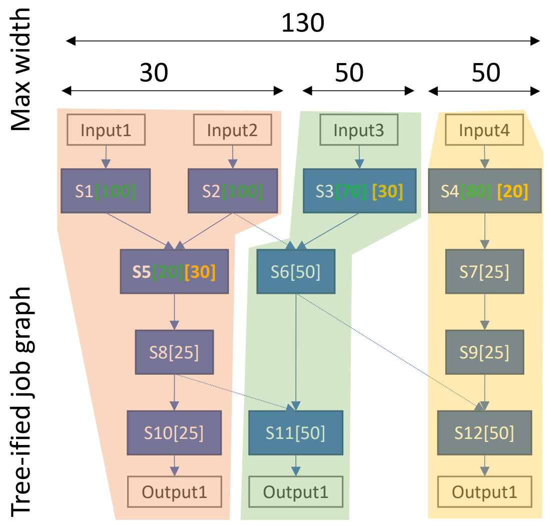


Figure 5: Shaping the resources for a typical SCOPE job at a particular point in execution.

query graph based peak resource estimator that could be invoked at any point duration the course of query execution, e.g., at the end of each task execution. Excess resources are released via communication between the job manager and the job scheduler.

Below we discuss the two steps, namely *tree-ification* and *max-cut*, for estimating the peak resource in the remaining of the job execution.

1. Tree-ification. A key techniques that we use for estimating the peak resources for the remaining of the job is to convert a job graph into one or more trees. This is possible by removing one of the output edges of the *Spool* operators in the job graph, since Spool is the only operator in SCOPE that could have more than one consumers[1]. The intuition behind removing one of the output Spool edges is that the stage containing the Spool operator could not run concurrently with its consumer stages. We remove the edge with the consumer that has the maximum in-degree. In case of a tie we pick any consumer at random. Furthermore, in the trivial case, if the maximum in-degree of spool consumers is one then we do not remove any edge, since the subgraph is already a tree.

2. Max-cut. Figure 5 illustrates the resource shaper on a typical SCOPE job. The stage graph of this job consists of twelve stages over four inputs and produces three outputs. Note that stages $S2$, $S6$, and $S8$ have spool operators, since they have two downstream consumers each. To convert this DAG into a set of trees, we consider removing one of the outgoing edges of these spools. For $S8$, we remove the edge with $S11$, since it has a higher in-degree than $S10$. For $S2$ and $S6$, we pick an edge at random since their consumers have equal in-degrees of 2. This results in three trees that

[1]A Spool operator represents a common subexpression that is consumed by one or more downstream operators [4].

Algorithm 1: `Resource Shaper`

```
Input   : stage graph $G$, stage vertices $V$, current resources $R$,
          completion state $W$
Output : updated peak $P$
T = Treeify($G$)
maxRemaining = Empty
foreach $root \in T.roots$ do
    maxRemaining.Add(RemainingPeak($root, V, W$))
if $maxRemaining < R$ then
    GiveUpResources($R -$ $maxRemaining$)
```

Algorithm 2: `RemainingPeak`

```
Input   : root stage $s$, stage vertices $V$, completion state $W$
Output : updated peak $P$
if $W[s] \geq V[s]$ then
    return Empty;
childResources = Empty;
foreach $child \in s.ChildStages$ do
    childResources.Add(RemainingPeak(child, $V$, $W$));
return Max(Resources($s$), $childResources$)
```

are highlighted in red, green, and yellow in Figure 5. Now consider the vertices shown in square brackets for each of the stage in Figure 5. Also consider a point in execution where the green numbers in brackets denote the completed vertices, the yellow numbers denote the running vertices, and the white numbers denote vertices that are yet to be scheduled. Given this particular point in execution, we can compute the maximum remaining peak by computing the maximum width of each of the trees, which turns out to be 30, 50, and 50 respectively, and then taking the sum of the individual tree widths, i.e., 130. If the job started with say 200 tokens then we can release 70 tokens at this point in execution.

Algorithm 1 shows the control loop of the resource shaper that first converts the job graph (G) into a set of trees, and then recursively computes the remaining peak in each of the tree root nodes (note that the trees are inverted here, so the output operators are the root nodes). If the total remaining peak is less than the current resources, then the job manager makes the call to give up excess resources. Algorithm 2 find the remaining peak. The algorithm returns empty if all vertices in a stage s have been completed, i.e., the count of completed vertices in s (denoted as W[s]) matches or exceeds the total vertices in s (denoted as V[s]). Otherwise, it iteratively adds the peak resources of its children, and returns the max of the children and parent peak resources. Function $Resources(s)$ gives the count of vertices that are yet to be executed in stage s. Essentially, the peak resource estimation algorithm finds the max-cut in each of the trees generated from the job graph and takes the sum.

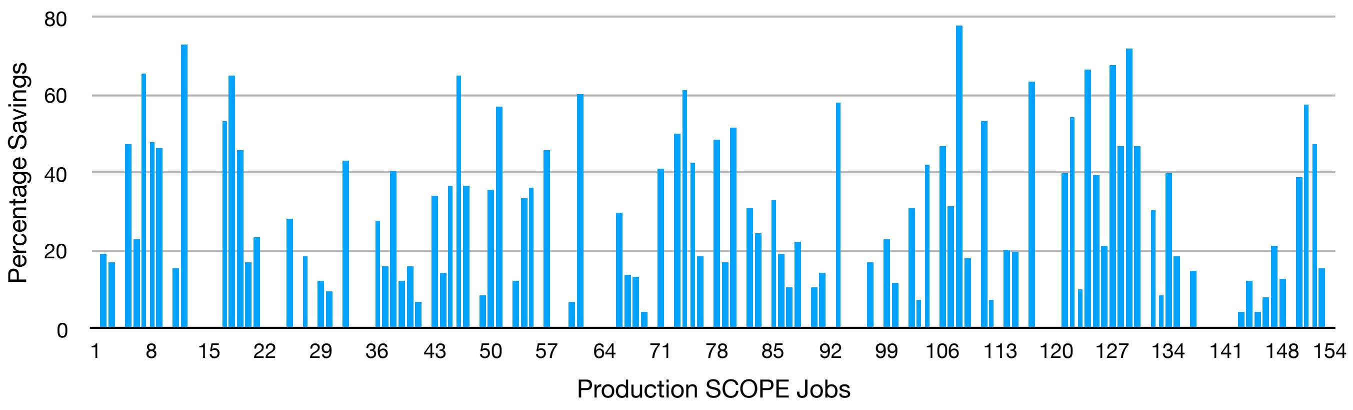


Figure 6: Performance results from one of the production virtual cluster. Each bar denotes the percentage saving in resource usage by each of the production SCOPE queries on that virtual cluster.

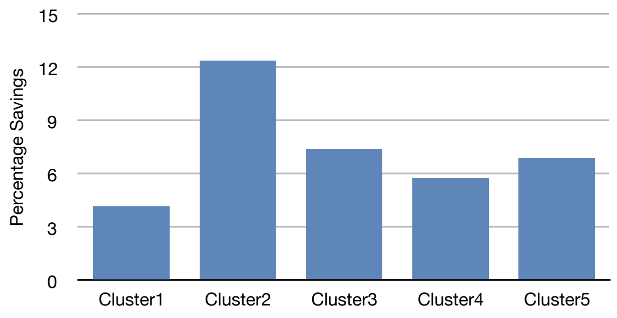


Figure 7: Resource usage savings over the same workload as in Figure 4. (Compare with the opportunity from Figure 4c)

5 Evaluation

We now present an evaluation of greedy our resource shaper implementation in two parts: (1) to show the estimated benefits from the entire production workload fleet, and (2) to show the performance results over one of the virtual clusters, i.e., a smaller subset of the overall workload. For both experiments we use resource usage savings as the metrics. Recall that resource usage is defined as the area under the resource allocation curve.

5.1 Overall Estimated Savings

We first present estimated gains over the same workload fleet as used for the motivating analysis from Figure 4, i.e., a single day workload consisting of 10s of thousands of SCOPE jobs from each of five different production clusters. For this experiment, we replayed the resource skyline history for each of the jobs and invoked the resource shaper module to estimate the savings. Figure 7 shows the percentage guaranteed resources that gets freed up when the resource shaper estimates the peak resource usage in the remaining of the job execution, and release any excess resources. The percentage savings varies for different clusters. Cluster2 shows the highest benefit since it contains jobs processing very large datasets that could use very high peak resources but then their resource requirements taper very fast. These are ideal examples where resources could be quickly freed up as the job execution proceeds.

Note that the savings shown in Figure 7 are modest compared to the opportunity illustrated in Figure 4c. This is because our current resource shaper only computes the peak in the remaining of the query plan, since releasing excess resources is much cheaper than requesting additional ones. Still, overall, we see that $4-13\%$ of guaranteed resources in over-allocated jobs could be freed up in different clusters. This translates to significant operational cost savings in a hundreds of millions of dollar business.

5.2 Performance Results

Given that production resources are scarce, running performance experiments over the entire production fleet is not feasible. Therefore, we pick a subset by selecting one of the virtual clusters[2] with 154 SCOPE jobs. We re-ran each of these jobs over the production inputs but redirected their outputs to a dummy location, similar as in prior works [1, 7, 13]. In each of the executions, the job manager invokes the resource shaper and measures the actual savings in resource usage. Figure 6 shows the savings for each of the jobs. We see that 99 out of 154 jobs, i.e., 64%, having savings. The savings could range up to 78%, with average savings being 20.5% across all 154 jobs. Overall, we see a savings of 8.3% for the cumulative resource usage of these 154 jobs.

The above results demonstrate the utility of plan-aware resource allocation for our production workloads. Note that there are no side effects of our approach, i.e., there is no performance penalty and the overheads for running the resource shaper algorithm are negligible, since we perform a single post-order traversal over each of the trees.

[2]A virtual cluster is a sub-cluster roughly, roughly corresponding to a business unit, having a dedicated set of resources for its workload.

6 Discussion Topics

We believe that adding plan-awareness to resource allocation opens up several new topics, and we seek active discussion and feedback. We summarize some of these topics below.

Beyond conservative resource shaper? The resource shaper presented in this paper was super conservative, and yet substantial gains were seen. The question therefore is whether more aggressive strategies could do better. For instance, instead of only finding the next peaks, could we also follow the resource curve down the "valley" and come back up? This would require to release and request resources over the course of query execution, and the question is whether this is going to be practical without hurting performance (e.g., waiting in the queue for acquiring resources). Could efficient queue implementations help here? Being plan-aware not only allows us to react to the query execution behavior but also see and reason about what lies ahead in the remaining of the plan. Having said that, so far we have only considered a conservative set of improvements by looking at the query plan structure, and it is still unclear how far we are from the optimal.

Modeling task scheduler? We made several simplifying assumptions regarding how the tasks would be scheduled. And even though we could still see gains in performance experiments, we believe there is still a huge room for modeling the task scheduler more carefully. For instance, we assumed that all stages run equally fast, but what if we were to consider the stage execution costs from the query optimizer? Likewise, the assumption that the total max-width in each of the trees will give the next peak is again very conservative. The question is whether we could mimic the task scheduler more accurately and whether that matters in practice.

Re-planning the query plan? Just as resource allocation is impacted by the query plan, vice-versa is true as well. Especially if resources are constrained or if there is a lot of contention, then an alternate query plan could be chosen. Likewise, just as the resource allocation changes over time, so could the query plan. For instance, instead of requesting more resources, adjusting the query plan could be cheaper.

Merits and demerits of plan-aware resource allocation. Finally, it would be interesting to get feedback on the merits and loopholes in our approach. Specifically, since we are making resource allocation dynamic and depending on the query plan, are there concerns about the *robustness* of the job manager? Do we see other problems with the dynamic behavior? Is the plan-aware approach more likely to cope with changing or mixed workload scenarios, where the black box approaches need more time, or data, or manual adjustments to adapt to? Discussing some of these tradeoffs would be interesting.

References

[1] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *NSDI*, pages 21–21. USENIX, 2012.

[2] AWS Athena. https://aws.amazon.com/athena/.

[3] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, pages 285–300, 2014.

[4] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[5] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, pages 99–112, 2012.

[6] Google BigQuery. https://cloud.google.com/bigquery.

[7] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*, 2018.

[8] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[9] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. Perforator: eloquent performance models for resource optimization. In *Symposium on Cloud Computing*, pages 415–427. ACM, 2016.

[10] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. Cost models for big data query processing: Learning, retrofitting, and our findings. In *SIGMOD*, 2020.

[11] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.

[12] L. Viswanathan, A. Jindal, and K. Karanasos. Query and Resource Optimization: Bridging the Gap. In *ICDE*, pages 1384–1387, 2018.

[13] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. Towards a Learning Optimizer for Shared Clouds. *PVLDB*, 12(3):210–222, 2018.