MIFO: A Query-Semantic Aware Resource Allocation Policy

Prajakta Kalmegh
Duke University
Durham, North Carolina
pkalmegh@cs.duke.edu

Shivnath Babu
Unravel Data Systems
Palo Alto, California
shivnath@unraveldata.com

ABSTRACT

Data Analytics Frameworks encourage sharing of clusters for execution of mixed workloads by promising fairness and isolation along with high performance and resource utilization. However, concurrent query executions on such shared clusters result in increased queue and resource waiting times for queries affecting their overall performance. MIFO is a dataflow aware scheduling policy that mitigates the impacts due to queue and resource contentions by reducing the waiting times for queries near completion. We present heuristics that exploit query semantics to proactively trigger MIFO-based allocations in a workload. Our experiments on Apache Spark using TPCDS benchmark show that compared to a FAIR policy, MIFO provides an improved mean response time, reduced makespan of the workload and average speedup between 1.2x-2.7x in highly concurrent setting with only a momentary deviation in fairness.

KEYWORDS

Cluster scheduling; dynamic resource allocation; scheduler queue wait time; resource interferences

ACM Reference Format:

Prajakta Kalmegh and Shivnath Babu. 2019. MIFO: A Query-Semantic Aware Resource Allocation Policy. In *2019 International Conference on Management of Data (SIGMOD '19)*, June 30–July 5, 2019, Amsterdam, Netherlands. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3299869.3319902>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '19, June 30–July 5, 2019, Amsterdam, Netherlands

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-5643-5/19/06...\$15.00

<https://doi.org/10.1145/3299869.3319902>

Figure 1: Slowdown in different stage types (Initial, Intermediate, Later) of TPCDS Query Q_{75} .

1 INTRODUCTION

An increasing number of enterprises are hosting their business critical applications on a cluster shared between multiple tenants due to its flexibility and cost benefits. Users submit a mix of data analytical, machine learning or graph analytic queries to data analytics frameworks like Spark [42], Hive [37], Presto [6], etc. Since CPU, network, memory, and disk IO are shared among these applications, performance isolation has become an important requirement. Techniques like resource reservations [11], capacity constraints [2], etc. are useful but can lead to under-utilization or unexpected wait-times as: (i) users specifying the reservation requirement are often unaware of workload dynamics [16], and (ii) tenants have limited control over the resource usage of other's queries. This is further complicated by changing workload schedules due to online arrival of both planned (e.g. SLA or deadline-based queries) and unplanned (e.g. adhoc or best-effort) queries. From our interactions with multiple clients and from studies on unpredictable execution experiences [21] we observe that performance variability in query completion times is a typical problem seen today. In this paper, we explore an opportunity to improve predictability in the completion times of data analytical queries, which constitute an important share in a cluster workload [21, 34].

1.1 Motivation

Figure 1 shows the slowdown for different stages of TPCDS analytical query Q_{75} categorized by their depth from root stage in the physical execution plan (DAG). *Slowdown* of a stage is the ratio of its runtime to actual compute size (CPU time) [18]. Query Q_{75} was part of a workload of 50 randomly chosen TPCDS queries that were submitted to the cluster in a poisson arrival pattern. It is observed here that the later (smaller) stages have a slowdown of more than 15x (upto 35x) while initial (larger) stages have substantially smaller slowdown. This high slowdown for later stages, when the query has finished *most* of its work is undesirable, especially when it can lead to missed deadlines. Figures 2a and 2b show the variation in the Scheduling wait time (SWT) and resource wait time (RWT) values for tasks of different stage types in query Q_{75} . SWT is the time spent in the scheduler queue, and RWT is the total time that the tasks of these stages are waiting for resources (defined in Section 2). Note that the values of SWT and RWT for *later* stages, are comparable to those of *Intermediate* and even *Initial* stages while their actual compute times are much smaller. It is this disproportional wait time that results in very high slowdown for later stages observed in Figure 1.

One challenge is therefore to reduce such delays for later stages that would enable better performance and increased predictability of a query. Another challenge for any strategy to address is knowing when the later stages start. On a cluster with online query arrival this would require tracking the progress of a query along with its overall resource utilization. Most scheduling policies in use today like FIFO, FAIR [41] are dataflow agnostic. They do not know/use the structural properties of the query or its remaining resource requirements in scheduling decisions. . To the best of our knowledge, there is no resource-allocation policy that dynamically redistributes shares of resources among concurrent queries based on their dataflow semantics and progress. This forms the motivation of our work. It is important to optimize the overall runtime of these data analytical queries as they typically constitute a significant share of production workloads [17, 21, 33, 34], while not impacting other concurrent workloads.

Scope: MIFO assumes the following requirements for its use in a cluster scheduler: (a) a dataflow-based execution of SQL analytical queries, (b) a decreasing resource usage profile of queries selected for MIFO-based allocation, (c) availability of task runtime and wait-time metrics from its previous executions, and (c) availability of query progress information at runtime computed using the task CPU-times. MIFO based allocation can be used in many popular cluster schedulers like Spark [42], Hadoop Yarn [39], etc. which meet the above requirements.

Figure 2: Variation in Scheduling Wait Time (SWT) and Resource Wait Time (RWT) values for different stage types of TPCDS query Q_{75} .

1.2 MIFO Overview

We present MIFO, a query-semantics aware dynamic resource allocation policy that safeguards data analytical queries from two specific impacts of concurrency, namely online arrival and interferences. We discuss these problems - *Online Slot Hogging* (OSH) and *Multi-Resource Interferences* (MRI) in **Section 3**. In a mixed online workload, MIFO identifies queries that have a decreasing resource usage profiles, and that show vulnerability to slow down in their previous executions. It allows such queries to run with their configured share of resources (e.g. weighted fair) until they reach a certain age. This is called the MIFO Trigger Age, denoted as λ_{trig} . At this point it employs a combination of three techniques, described in **Section 4**, (i) *Dynamic Prioritization*, (ii) *Interference Avoidance*, and (iii) *Locality Ignorance*. In **Section 5**, we present some heuristics that use query-semantics and query-progress to determine the MIFO Trigger Age *i.e.* when a query becomes eligible for MIFO-based allocations during its execution. In **Section 6**, we discuss our implementation of MIFO in Spark [42] and compare it to max-min FAIR and FIFO policies implementations. We also compare MIFO with a query-progress aware version of Shortest-Remaining-Processing-Time (SRPT) policy that favors near-completed queries. We present our experimental results that evaluate MIFO in **Section 7**. Finally, we present related work in **Section 8** and conclude in **Section 9**.

2 BACKGROUND

In this section, we review the execution model of queries on data analytics frameworks and present some terminology.

2.1 Data Analytical Queries

This paper focuses on data-flow based query execution on data analytics frameworks like MapReduce [13], Spark [42], Presto [6], etc. Consider the physical execution plan for TPCDS query Q_7 shown in Figure 3; it consists of a Directed

Figure 3: TPCDS query Q_7 execution DAG. The blue labels on stage boxes are table names; the labels within the inner rectangles are plan operators.

Acyclic Graph (DAG) of low-level parallelizable units called *stages*. A stage consists of parallel tasks that perform same operator(s) logic on different blocks of input data. The edges represent data dependencies between the stages and their direction represents the data flow. We broadly classify all stages into three categories based on their operator types in the dataflow - *Initial*, *Intermediate* and *Later*. Figure 4 illustrates the decreasing resource usage pattern for TPCDS Q_7 . The initial stages are characterized by longer median task CPU time, while later stages exhibit low computation needs.

Initial Stages: We observe that analytical queries are characterized by heavy disk IO usage in the initial stages due to table scan operations on fact tables. These stages also demand more cpu (see Figure 4) owing to deserialization and record construction. Even if the table is partly/fully cached it only reduces the IO requirement but cpu usage is still relatively high as they process large input data.

Intermediate Stages: These stages typically perform joins and partial aggregations for grouping operations. They also tend to work on relatively larger data (especially in the case of bushy plans where multiple parallel initial stages merge into an intermediate join stage) with heavy requirement for all three resources cpu, memory and network io.

Later Stages: The later stages usually perform aggregations (for grouping and window operators) on partially aggregated data and/or order small resultset. The input data to these stages is much lower than earlier stages and even some costly aggregate operations do not add up to high usage.

2.2 Task Lifecycle

A stage is scheduled for execution when all its dependencies (previous stages) are completed. At every scheduling instance t , the scheduling queue is ordered by the scheduling policy using stage priorities. The priority P_j for stage j is:

$$P_j = \frac{w_j}{r_j} \quad (1)$$

Figure 4: Box-plot shows the decreasing resource requirements for tasks of Q_7 categorized by their stage type.

Figure 5: Task runtime components: Compute time, SWT: Scheduling Wait Time, RWT: Resources Wait Time

The value of r_j gives the current holdings for that stage. Tasks are launched for stages in the order of their priorities. Figure 5 shows the lifecycle of a task from the time its stage is submitted to the scheduler queue to its completion:

Scheduler Waiting Time (SWT): It is the time spent in the scheduler queue waiting for an execution slot. For a task, $SWT_{task} = task^{launch} - task^{submit}$ where $task^{launch}$ is the task's execution start time, and $task^{submit}$ is the time when its stage is submitted to scheduler queue. For a stage the SWT_{stage} is the cumulative SWT of all its tasks.

Compute Time: It is the time that a task spends running on the processor represented as $task^{compute}$.

Resource Waiting Time (RWT): It is the time when the task waits for system resources like network, cpu, disk and memory. It is computed as $RWT_{task} = (task^{finish} - task^{launch}) - task^{compute}$. RWT_{stage} is the total RWT of all its tasks.

3 THE PROBLEM SETTING

A max-min implementation of the FAIR scheduling policy is most widely used in today's data analytics frameworks. As a slot becomes free it is allocated to a stage that is *starving* the most (i.e., its current share or holdings is lowest from fair share) [19, 41]. Example 3.1 shows how online arrival of queries can delay the execution of already running queries that are towards their completion:

Figure 6: A timeline showing online arrival of Q_2 and Q_3 when Q_1 is in its last stage

EXAMPLE 3.1. Figure 6 shows an online arrival of queries Q_1 , Q_2 and Q_3 on a shared cluster with total 90 slots in a FAIR allocation policy. Queries Q_2 and Q_3 arrive when Q_1 with a decreasing resource profile is executing its stage S_{13} .

Observe that at time t' when Q_2 arrives, S_{21} just started and is starved with no allocated slots out of the eligible fair share of 45 (90 split between 2 stages); whereas S_{13} uses all 90 slots. As tasks of S_{13} complete they are allocated to stage S_{21} until both stages run with their fair share. At t'' when Q_3 arrives, S_{31} just started is most starved with no allocated slots out of the eligible 30 (90 split into 3 stages), and thereby receives the next 30 free slots further stalling S_{13} . Figure 7 illustrates the redistribution of slots in this scenario. We call this effect as **Online Slot Hogging (OSH)** which directly impacts the SWT of a stage. Since the tasks of initial stages S_{21} and S_{31} have higher median task runtimes, they hold on to the slots longer delaying the launch of the remainder tasks of S_{13} .

Consider another problem we refer to as **Multi-Resource Interferences (MRI)**. Tasks of different queries execute concurrently on a host and contend for various low-level resources. The primary reason for this interference is that resources are mostly partitioned only based on CPU slots (e.g. Spark [42]) or based on memory and virtual cores (e.g. YARN [39]). However, this partitioning does not provide any isolation guarantees on other resources like network, and disk. In practice even cpu cores are oversubscribed to increase utilization. It is thus challenging to shield the tasks of a query from such conflicts especially in a cluster that hosts adhoc queries. If the tasks of a later stage are placed on hosts with high resource contentions, they experience RWT that is disproportional to their compute needs.

Figure 7: Resource re-distribution for Example 3.1.

3.1 Contribution of SWT and RWT

On a cluster, let S be the total number of slots and T be the average task runtime. A slot becomes free every T/S seconds [41]. A query q with N tasks in the scheduler queue needs to wait for $SWT_q = (NT)/S$ seconds before all tasks are launched. Data analytical queries have multiple stages with many parallel tasks. For example, in our experiments, a query typically had 4-10 stages with 800-1500 tasks. In private clusters the number of slots is typically much less than the number of tasks when the contention for resources is high, i.e. $S \ll N$. As such, from above since $\frac{SWT_q}{T} \propto \frac{N}{S}$, the contribution of SWT to query runtime is significant. In our workload SWT constituted on an average about 40% of the overall query runtimes. For this workload, cumulative RWT contributed to about 45% of the overall delay whereas the actual CPU execution time was just 15%.

The contribution of SWT to query runtime can be reduced by either increasing S or decreasing N above. An increase in S can be accomplished by launching more tasks per host but will lead to higher resource contentions (increased RWT) and reduced utilization. In declarative workloads like sql queries it is not possible to control N at runtime as it depends on the generated physical plan and input data, both of which vary per query and also across its execution. These limitations reduce the administrator's ability to minimize both SWT and RWT for data analytical queries. Next, we discuss how MIFO-based allocations can help mitigate the impact due to the OSH and MRI problems.

4 TECHNIQUES IN MIFO

Recall from Section 2.2, the scheduler calculates a query's scheduling priority based on its weight and instantaneous slot allocation. In MIFO, once a query crosses a certain age, termed as its MIFO Trigger Age ($\lambda_{q, trig}$), we employ the following three techniques to advance the query and to shield its remaining tasks from slowdown:

- (1) **Dynamic Prioritization (DP):** The query's scheduling priority is increased by increasing its weight $\mathcal{W}$ dynamically such that it receives a larger share of resources than its static configuration. In Section 4.1, we show how this helps us to tackle the impact due to OSH.
- (2) **Interference Avoidance (IA):** Offers from hosts that can *potentially* result in increased RWT due to resource interferences are rejected. We describe this technique in Section 4.2.
- (3) **Locality Ignorance (LI):** Finally, once query reaches age $\lambda_{q, trig}$, we turn off delay scheduling for it and let the remainder stages of q accept offers irrespective of locality preference. We discuss locality ignorance in Section 4.3.

We defer the discussion on computing MIFO Trigger Age ($\lambda_{q, trig}$) to next Section 5.

4.1 Dynamic Prioritization

The weight $\mathcal{W}_{q,t}$ of a query q at any scheduling instant t after $\lambda_{q, trig}$ is computed using a dynamic scaling factor $\alpha_{q,t}$ as:

$$\text{where, } \mathcal{W}_{q,t} = \mathcal{W}_{q, conf} + (\alpha_{q,t} * \lambda_{q,t}) \quad (2)$$

- $\mathcal{W}_{q, conf}$ = weight defined by a user or system by static policies
- $\lambda_{q,t}$ = age of query q at time t
- $\alpha_{q,t}$ = scaling factor for query q at time t

We present our dynamic prioritization logic in Algorithm 1. While making a scheduling decision the scheduler uses the *priorityComparator()* method (line 4) to compare priorities of query pairs in the scheduler queue. This method retrieves previously recorded weights $\mathcal{W}_{q_1, prev}$ and $\mathcal{W}_{q_2, prev}$ of the queries q_1, q_2 for stages s_1, s_2 respectively and computes their new weights $\mathcal{W}_{q_1, t}$ and $\mathcal{W}_{q_2, t}$ (lines 5,6) using the *computeNewWeight()* method (lines 11-18). These weights are used to compute query priorities (lines 7,8). The dynamic weight of a query is computed in the method *computeNewWeight()* (lines 11-18) using Equation 2. It checks if the query has crossed its MIFO trigger age in *isMIFOTriggered()* (lines 19-24). The MIFO Scaling Factor is computed in method *computeScalingFactor()* (lines 25-31).

4.1.1 MIFO Scaling Factor ($\alpha_{q,t}$). The MIFO Scaling Factor $\alpha_{q,t}$ controls the extent of benefit desired compared to a FAIR allocation share. We now derive the bounds for $\alpha_{q,t}$ in a cluster with total N slots and k concurrently running stages.

Weight for Minimum Benefit ($\mathcal{W}^{min}$): A lower-bound for the weight *i.e.* $\mathcal{W}^{min}$ of a stage S_j of query q is when it gets minimum benefit (*i.e.*, at least 1 slot more) than its fair share. The priority $\hat{P}_{j,q}$ should be strictly greater than its equivalent FAIR priority. At time t , let stage S_j be running with its fair share of tasks (N/k) when another stage $S_{j,q'}$ frees a slot. Thus $S_{j,q'}$ now has a total of $r_j = (N/k) - 1$ running tasks. If the weight of query q' with stage $S_{j,q'}$ is $\mathcal{W}_{q'}$, the minimum benefit is achieved if $\hat{P}_{j,q} > P_{j,q'}$, this slot should go to S_j instead of $S_{j,q'}$. Substituting the values for r_j and $r_{j,q'}$ in Equation 1, the lower bound on the weight for stage S_j is:

$$\mathcal{W}^{min} > \frac{\mathcal{W}_{q'} (N/k)}{(N/k) - 1} \quad (3)$$

Weight for Maximum Benefit: An upper-bound *i.e.* $\mathcal{W}^{max}$ gives the weight required to get the maximum benefit over FAIR at time t . An allocation using $\mathcal{W}^{max}$ implies that stage S_j should get all possible slots in the cluster once q reaches age λ^q . Note, in order for each of the $k - 1$ concurrent stages

Algorithm 1 Algorithm to dynamically prioritize tasks of query q at scheduling instance t using heuristic H – KNEE. For simplicity, we omit the use of *minshare* of a query in computing the final priorities. Similar to FAIR, in practice *minshare* is used allocate minimum slots to a query.

```

1: Input: head stages  $s_1, s_2$  in scheduler queue;
    $\forall_{q \in Q} (\mathcal{W}_{q, conf})$ ;  $r_{s_1}$  : running tasks of stage  $s_1$ ;  $r_{s_2}$ 
   : running tasks of stage  $s_2$ ;  $\mathcal{W}_{q_1, prev}$  : last recorded
   weight of query  $q_1$ ;  $\mathcal{W}_{q_2, prev}$  : last recorded weight
   of query  $q_2$ ;  $\forall_{q \in Q} (\lambda_{q, trig})$  = MIFO Trigger Age for
   query  $q$ ;  $T_{q, obs}$  = observed (in this execution) CPU
   Time for all tasks of  $q$ ;  $T_{q, ideal}$  = ideal (from  $k$  pre-
   vious executions) CPU Time for all tasks of  $q$ ;
2: Output: best-fit stage with higher priority  $P$ 
3: Scheduling method: priorityComparator( $s_1, s_2$ )

4: function PRIORITYCOMPARATOR( $s_1, s_2$ )
5:   Get  $\mathcal{W}_{q_1, t}$  : computeNewWeight( $q_1, \mathcal{W}_{q_2, prev}$ )
6:   Get  $\mathcal{W}_{q_2, t}$  : computeNewWeight( $q_2, \mathcal{W}_{q_1, prev}$ )
7:    $P_{s_1} = \frac{\mathcal{W}_{q_1, t}}{r_{s_1}}$ 
8:    $P_{s_2} = \frac{\mathcal{W}_{q_2, t}}{r_{s_2}}$ 
9:   return  $P_{s_1} \geq P_{s_2}$ 
10: end function

11: function COMPUTENEWWEIGHT( $q, \mathcal{W}_{q', prev}$ )
12:   if isMIFOTriggered( $q$ ) then
13:     Set  $\alpha_{q,t} = \text{computeScalingFactor}(q, \mathcal{W}_{q', prev})$ 
14:      $\mathcal{W}_{q,t} = \mathcal{W}_{q, conf} + (\alpha_{q,t} * \lambda_{q,t})$ 
15:   else  $\mathcal{W}_{q,t} = \mathcal{W}_{q, conf}$ 
16:   end if
17:   return  $\mathcal{W}_{q,t}$ 
18: end function

19: function ISMIFOTRIGGERED( $q$ )
20:    $\lambda_{q,t} = \frac{T_{q, obs}}{T_{q, ideal}}$ 
21:   if  $\lambda_{q,t} > \lambda_{q, trig}$  then return true
22:   else return false
23:   end if
24: end function

25: function COMPUTESCALINGFACTOR( $q, \mathcal{W}_{q', prev}$ )
26:   Get  $N$  = total number of slots in the cluster
27:   Get  $k$  = total number of concurrent queries at time  $t$ 
28:   Set  $\mathcal{W}^{max} = \mathcal{W}_{q', prev} * (N - k + 1)$ 
29:    $\alpha_{q,t} = \mathcal{W}^{max}$ 
30:   return  $\alpha_{q,t}$ 
31: end function

```

to proceed without starvation, we need to set a non-zero but minimum priority $P_{j,q'}$ for each stage to allow them to proceed. Thus, we allocate a maximum of 1 slot each to the $k - 1$ other stages, and allocate the rest of the $N - (k - 1)$ slots to stage S_j . Thus, when a slot becomes available at time t , the number of tasks running for stage S_j and $S_{j,q'}$ will be $r_j = (N - (k - 1))$ and $r_{j,q'} = 1$ respectively. Substituting for these values, the weight for maximum benefit is:

$$\mathcal{W}^{max} > \mathcal{W}_{q'}(N - k + 1) \quad (4)$$

Summary: Since the value of N remains constant, the value of $\alpha_{q,t}$ depends on the concurrency k and the weight of the other query q' being compared with $(\mathcal{W}_{q'})$ at time t . Thus, use of a scaling factor in computing new priority for a query ensures that newly arriving stages do not delay its time in the scheduler queue (SWT) significantly *i.e.* the OSH impact due to increased k is mitigated. We show that MIFO is starvation-free in Appendix A. Next, we discuss how MIFO addresses the issue of *Multi-Resource Interferences* MRI (defined in Section 3).

4.2 Interference Avoidance (IA)

Our goal is to reduce the RWT (defined in Section 2.2) for later stages of a query q by minimizing the impact from the interferences caused by concurrent queries *i.e.* we relinquish any offers from hosts that can *potentially* result in any resource interferences for q . Suppose query q was executing its stage j when MIFO was triggered for it. For each host h on which tasks of stage j are running, we compute the $RWT_{j,h}^{mean}$ as the moving mean RWT faced by tasks of stage j on host h . This is maintained at the scheduler based on wait time metrics received in heartbeats. When a slot on host h is offered to stage j , the scheduler checks if $RWT_{j,h}^{mean}$ is greater than the mean RWT across *all hosts* by a margin. If yes, it is a *potential* interference; the scheduler relinquishes the offer and waits for one from a less contentious host (or until a configured time threshold).

This simplistic technique enables us to address impacts due to MRI by rejecting offers only if there is a higher wait time (a case of potential slowdown) on some host irrespective of task runtimes on that host. For example, even if there are long running tasks on some other host h' but they are not causing any interference or resource wait times to the remaining tasks of q , in MIFO we still accept offers from them. As we demonstrate in Experiment 7.3, applying only IA may not show benefits as relinquishing offers can further add to the SWT of the queries. However, when applied along with DP, the query gets more slots offered quickly and a better chance to launch its tasks on machines with less potential conflicts. .

4.3 Locality Ignorance (LI)

In data analytics frameworks, locality preference is a major consideration for task placement to improve cluster throughput. In the *delay scheduling* [41] approach a stage that cannot launch a data-local task relinquishes the offer to other stages while it waits for another offer where it gets an opportunity to launch a data-local task. We argue that while this is a good strategy for the initial IO and CPU intensive stages of a query, the later stages with minimal resource requirements spend undue time waiting for offers from preferred hosts resulting in their increased SWT. Additionally, they are also subjected to MRI if the concurrent tasks on the preferred hosts are contentious on the same resources. This concern is also discussed recently in [4]. In MIFO, we modify the default delay-scheduling implementation. Specifically, after MIFO is triggered for a query, we turn off delay scheduling and let the stages accept any offer that comes their way unless it is from a *potentially* contentious host (see previous section). As we show in our experiments in Section 7, this contributes significantly towards the reduction in the SWT and RWT values of later stages of a query. Next, we describe the heuristics used to compute MIFO Trigger Age for queries.

5 TRIGGERING MIFO

Before we present our heuristics, we first define how we compute the age $\lambda_{q,t}$ of a query q at any time t .

5.1 Computing age of a query

We compute the age of a query using the task compute times as the primary measure of ‘*work-done*’ estimation. We define $\lambda_{q,t}$ of query q at time t as:

$$\lambda_{q,t} = \frac{\tau_{completed}}{\tau_q} \quad (5)$$

where,

- $\tau_{completed}$ = cumulative CPU time of all *completed* tasks of query q at time t in current execution;
- τ_q = cumulative CPU time of all tasks in the DAG of query q from a previous execution.

In recent approaches that employ re-ordering policies [14, 22, 34], task duration estimates are collected from previous executions for recurring jobs to compute their progress. However, query progress estimation using historical task runtimes can lead to sub-optimal scheduling decisions [15]. Figure 8 shows that query Q_7 ’s runtime varies significantly while there is little deviation in its CPU time in different concurrent settings. Our choice to compute the age of a query based on task CPU times and not task runtimes provides a rather simplistic approach for our context of highly diverse concurrent settings.

Figure 8: Plot showing variation in runtime and CPU Time for TPCDS query Q_7 in 7 different executions when run concurrently with 10 other TPCDS queries chosen randomly in each run. The dotted lines plot the deviation of its runtime (red dotted) and its CPU time (black dotted) from an unconstrained (run in isolation) execution. Users see an unpredictable execution ranging from -1.5% in Run-3 to about 80% higher runtime in Run-1 compared to its unconstrained execution. But the variation in task CPU times is relatively much lower and limited to at most 7.4% in this example.

5.2 Heuristics to trigger MIFO

For each query q , in order to determine when to trigger our techniques, we explore the following heuristics:

- (1) **Operator Statistics (H – OPS):** when all of its stages consisting of scan operations have completed execution. Use of this heuristic does not require any historical executions. With H – OPS, once any query has crossed its resource-intensive scan stages, we enable all three MIFO techniques of dynamic prioritization (DP), interference avoidance (IA) and locality ignorance (LI).
- (2) **Knee-Point Detection (H – KNEE):** We keep track of two distinct MIFO Trigger Age values for a query - (i) $\lambda_{q, trig-DP}$: This is the age after which the query is vulnerable to the impacts due to OSH. We compute it by running a *Knee-point detection* algorithm on a query's previous execution data, and apply dynamic prioritization to queries annotated with this age in subsequent executions. (ii) $\lambda_{q, trig-IA}$: Similarly, we use the *Knee-point detection* algorithm to determine an age for applying interference avoidance. For our third technique of locality ignorance (LI), we use the former age (*i.e.* $\lambda_{q, trig-DP}$). For example, if we determine from a previous execution of q that its $\lambda_{q, trig-DP} = 0.7$, we apply DP and LI techniques when it exceeds 70% of its total work in the current execution.

Observe that in H – OPS, MIFO is triggered at stage boundaries; for H – KNEE each query may be in the middle of a stage when MIFO is triggered. As such, we dynamically increase the weight of the remaining tasks of that stage, and for the rest of the stages in the query DAG.

5.3 Knee-point detection Algorithm

In this section, we elaborate on our approach to compute $\lambda_{q, trig-DP}$ using the elbow or knee-point detection algorithm. For simplicity and due to space constraints, we skip the computation of $\lambda_{q, trig-IA}$. The steps are identical, except that we need to replace the use of SWT metric with RWT throughout the approach explained below.

The knee-point on a curve is defined as the point of maximum curvature. A curvature gives the rate at which a curve denoted by a set of discrete points changes its direction. To illustrate, consider the curve of SWT of tasks of a query q shown in Figure 9a. For all tasks of q , we plot their normalized cumulative scheduling wait time SWT (y-axis) against the normalized cumulative work done on x-axis. Each point on the curve denotes the proportion of SWT experienced by the query q (*i.e.* y-axis value) when it had reached the corresponding age denoted by its x-axis value. For detecting a knee-point on this curve, we use the Kneedle algorithm [35] that detects the point of maximum curvature in a dataset consisting of discrete points. The point where the dotted line crosses the task-curve in Figure 9a represents the knee-point

Figure 9: Knee-Point detection in a query q : For (a) and (b), Y-axis shows normalized cumulative scheduling wait time SWT for q (or normalized cumulative RWT for (c) and (d)); X-axis shows normalized cumulative work done or age of query. The vertical dotted line represents the knee-point of the curve. The color of the points represent the type of stage each task belongs to. Figures (a) and (c) denote a disproportional observation in SWT and RWT respectively for TPCDS query Q_7 , whereas (b) and (d) denote an ideal execution for another TPCDS query Q_5 .

for this curve, and is represented by: $\{x - value, y - value\}$. That is, the $x - value$ gives the age of the query, and $y - value$ gives the normalized SWT of the query at this knee-point.

Intuitively, a knee-point helps us detect the disproportionality in the SWT experienced by a query during its execution. For example, at the knee-point ($x = 0.69, y = 0.26$) in Figure 9a, query Q_7 experiences only approx. 26% of its overall SWT when it had finished about 70% of its work. The remaining tasks of Q_7 performed 30% of the total work while facing 74% of the SWT faced by Q_7 during its lifetime. Ideally, every stage of a query irrespective of whether it is an initial stage doing more work or a later stage should face a waiting time in the scheduler queue proportional to its compute needs. Figure 9b depicts such an example of query Q_5 where the total SWT faced by Q_5 when it finished 48% of its work is also approx 46%.

5.4 Compute MIFO Trigger Age

We use the Knee-point detection algorithm to pre-calculate the SWT and RWT based knee-points for every recurrent query from its previous executions. In order to flag queries with disproportional SWT or RWT, *i.e.* to distinguish between ideal (Figures 9b and 9d) and unacceptable (Figures 9a and 9c) executions, we compute the $\frac{y-value}{x-value}$ ratio, *i.e.* SWT-Work Done and RWT-Work Done Ratios. We maintain a system-wide user-configured threshold for disproportionality, called Th_{disp} . For example, in Figure 2a, plot for query Q_7 shown in (a) gives a value of $\frac{0.26}{0.69} = 0.37$ which is $<$ the configured value of say $Th_{disp} = 0.7$ is unacceptable. Whereas, for Q_5 , the SWT-Work Done Ratio is $\frac{0.46}{0.48} = 0.95$ which is $> Th_{disp}$ is ideal.

Note, some queries which are not characterized by a decreasing resource usage pattern do not exhibit any vulnerability to OSH even in multiple recurrent executions. The Kneedle algorithm tends to detect a knee-point for such queries at very early phases in their execution. In order to avoid flagging a query to be MIFO-eligible very early on, we use a system-wide MIFO trigger threshold, denoted by λ_{conf} , for considering a minimum age of any query after which it is vulnerable to impacts due to SWT and RWT. That is, the knee-points for these queries occur after a minimum age of λ_{conf} . Users can configure how many executions to consider; then, a query is qualified to be MIFO-eligible only if it has knee-points beyond λ_{conf} in, say, η previous executions. In our example, if $\lambda_{conf} = 0.6$ and since $Th_{disp} = 0.7$, Q_7 is marked as MIFO-eligible with its MIFO Trigger Ages $\lambda_{q, trig-DP}$ set to 0.69 and $\lambda_{q, trig-IA}$ set to 0.93. From our experiments, we noticed that some queries show vulnerability to either only disproportional SWT or RWT and some to both.

Next, we discuss how we have implemented these heuristics and our three techniques in Apache Spark [42] and its data processing engine SparksQL [8].

Figure 10: MIFO Architecture

6 IMPLEMENTATION

We have implemented MIFO in Spark [42] by extending its scheduling algorithm API. We now describe the MIFO architecture shown in Figure 10. The **Query Profile Generator** uses the Knee-point detection algorithm offline on η previous executions to identify MIFO-eligible queries (Step 0). A user inputs MIFO configuration parameters (Step 1) and submits the workload. At runtime, the **Query Execution Monitor** tracks the age of each query q (Step 2a) and feeds its progress information along with its stored profile (Step 2b) to the **MIFO Schedule Tracker** module (Step 3). The schedule tracker applies the configured heuristics (Step 4) in order to determine if a query has reached its MIFO Trigger Ages ($\lambda_{q, trig-DP}$ and $\lambda_{q, trig-IA}$) at time t . Once a query crosses this age, it is marked for reallocation. When resource offers are received by the Spark scheduler, if a query is marked for reallocation, it retrieves the new weight (Step 5) from the **Query Weight Calculator** and selects the *best-fit* stage.

An important contribution of MIFO is to make the task-level cluster scheduler aware of the query-level dataflow semantics enabling it to make effective resource allocation decisions. To enable this, we extended the interface between the data processing engine of SparksQL [8] and its compute engine (or Spark core) so that the DAG scheduler is made aware of dataflow dependencies in the query. Additionally, we have instrumented Spark to publish our performance evaluation metrics (see Section 7) to their REST API (Step 6). These metrics are parsed and processed to generate visualization plots (*e.g.* most plots in our experiments in Section 7 are generated through our UI using Plotly [5] library) for users to compare performance and wait-time data (Step 7) for individual resources (slots, cpu, network, memory, io).

6.1 Tuning MIFO parameters

The list of all configurable parameters in MIFO is shown in Table 1. MIFO has three main configurable parameters: λ_{conf} ,

Notation	Parameter	Range	Default
η	Number of prev executions for profiling	> 1	5
λ_{conf}	Knee-point Threshold	0 to 1	0.5
Th_{disp}	Unacceptable Disproportionality threshold	0 to 1	0.7
α_{bound}	Desired benefit from MIFO Scaling Factor	min, max, large	max

Table 1: MIFO Configuration Parameters

Th_{disp} and bounds of MIFO Scaling Factor. The values of λ_{conf} and Th_{disp} are workload dependent. In our implementation, we adopted the following approach to set the default values: (i) we calculate the knee-points of all the queries and plot a histogram of these knee-point values. We then pick an age threshold value beyond which we have a majority of queries with a knee-point. This forms the default value of λ_{conf} . (ii) For the set of queries filtered in step (i), we again plot a histogram of the disproportionality values (SWT/work_done and RWT/work_done) for these queries. We then select a $Th_{disp} > 0.5$ (since any value lower than this indicates no disproportionality) such that majority of queries are beyond this threshold. While this approach worked reasonably good for our experiments, it is time-consuming for an administrator. As a result, we are currently developing an auto-configurator that can use this methodology to use workload execution metrics as input and output the default values to be used by the administrator. With this methodology of tuning MIFO, we ensure that we don't select any queries which don't exhibit disproportionality for any wait-time metric in the past. A mis-configuration for these parameters, thus, will only result in more or less queries being selected for MIFO-eligibility in subsequent executions.

The bounds of MIFO Scaling Factor provides a trade-off between reducing *mttr* or increasing fairness of the workload. In our implementation, users can configure $\alpha_{q,t}$ value to be either $\mathcal{W}^{min}$ or $\mathcal{W}^{max}$, or configure it to be an extremely large value (we call this configuration as MIFO_{large}) such that all available slots in the cluster go to query q once it crosses MIFO Trigger age. To clarify, in MIFO_{large}, we do not scale a query's weight based on k or weight of the other queries, but set it to a very high fixed constant. Intuitively, MIFO_{large} assigns all available slots to a query once MIFO is triggered for it until it finishes execution. By default, we set it to $\mathcal{W}^{max}$. A configuration with $\mathcal{W}^{min}$ is desirable in scenarios where fairness is really important while some gains are still useful with MIFO-based allocations. Note, our implementation also allows dynamically changing the bounds at runtime; for example, if the user desires fairness for a certain duration of workload, then can set the bound to $\mathcal{W}^{min}$, and then reset it to $\mathcal{W}^{max}$.

6.2 Heuristics Library

In our implementation, we use the combination of both heuristics - H - OPS, and H - KNEE, in that order. That is, for example if $\lambda_{conf} = 0.5$, we trigger MIFO when a query has finished all its stages with resource-intensive scan operations, and the query has reached at least 50% of its compute requirements. If the knee-point age of the query is greater than this age, we mark the query at runtime as MIFO-eligible. Our implementation also provides a pluggable interface to add and apply more heuristics¹, and also specify their execution order. For example, in our on-going research we are exploring: (a) **Data Statistics** (H - DATA): when the size of input data for a stage in the query DAG reduces to $d\%$ (configurable) of the total input data size of its initial scan stages, (b) **Bushy Plan** (H - BUSH): here MIFO advances struggling stages with shorter task duration in a bushy plan that are awaiting their execution slots and blocking the dataflow from graduating to next level due to dataflow dependencies, and (c) **Structural Properties** (H - DAG): where we trigger MIFO based on the DAG structural properties (depth) of the query.

7 EXPERIMENTS

We present our evaluation of MIFO with experiments conducted on our local cluster that help answer the following questions:

- (1) **Exp-Policies:** How does MIFO compare to FAIR, FIFO and SRPT for our metrics of evaluation?
- (2) **Exp-Bounds:** What is the impact of setting MIFO Scaling Factor to different bounds (minimum, maximum and large benefit)?
- (3) **Exp-Techniques:** What is the contribution of individual techniques of MIFO (DP, IA, LI) in overall benefit?
- (4) **Exp-Load:** How does MIFO perform compared to FAIR for varying contention conditions, *i.e.* increasing arrival rate and concurrency of queries?
- (5) **Exp-Heterogeneity/Adhoc:** What is the impact of adding more heterogeneous queries with different resource profiles or adhoc queries with no historical execution data?
- (6) **Exp-Scalability:** What is the overhead of MIFO scheduling algorithm, and how does it scale to a larger concurrent query setting?(details in Appendix B)
- (7) **Exp-Mimic:** What is the impact of triggering MIFO earlier vs later? That is, how can we balance the optimization objectives by tuning MIFO? (details in Appendix C)

Experimental setup: We conducted our experiments on a local cluster consisting of Apache Spark 2.3 deployed on 20-nodes (each machine has 8 cores, 16GB RAM, and 1 TB storage). Spark was configured to run in standalone mode

¹we defer the discussion of these other three heuristics as it is a focus of our on-going research to analyze use of different dataflow-based heuristics with MIFO for different workload types (ML, graph analytics, etc.)

Figure 11: Exp-Policies: Comparison of (a) Mean-time-to-response (*mttr*), (b) makespan, (c) average speedup (the three bars here denote the speedup obtained in MIFO_5 over FAIR_5 (blue), FIFO_5 (skyblue), and SRPT_5 (green)), and (d) Resource Fairness Index (RFI) in FAIR vs MIFO vs FIFO vs SRPT.

(modified for MIFO) with other default configurations. Our query set consists of the TPCDS queries [7] working on a 100GB data stored on HDFS in Parquet [3] data format. The SQLs for these queries were taken from [36] without any modifications. The queries are submitted through Thrift-server in parallel streams from 5 concurrent users. Each user submits 10 queries to the same queue with a configurable Poisson arrival rate. Thus, our workload consists of a total of 50 random queries arriving from five concurrent users in a poisson distribution. Note, although we have only 5 concurrent users, the total number of concurrent queries at any time was observed to peak at 37. For each scheduling configuration, we ran five executions. The results presented in this section report the runs with median runtimes for each configuration to enable ease of comparison among individual queries in two different configurations.

Identifying MIFO-Eligible Queries: We executed our workload of 50 queries with a FAIR policy for $\eta = 5$ times, and used these executions to run our knee-point detection algorithm described in Section 5.3 to identify MIFO-eligible queries. Our algorithm returned 31 queries eligible for dynamic prioritization (DP) and 23 queries eligible for Interference Avoidance (IA) with MIFO. Only 18 of these queries were common. Thus, we had 14 queries not annotated for MIFO. These queries run with their FAIR share of resources until completion. We set $\lambda_{conf} = 0.5$ and $Th_{disp} = 0.7$. Same configurations were used for knee-point detection for both SWT and RWT disproportionality. Recall, this implies that we selected only those TPCDS queries which showed a ≤ 0.7 SWT-to-Work Done disproportionality for OSH and/or RWT-to-Work Done disproportionality for MRI after 50% of their compute time requirement in five previous executions.

Metrics for evaluation: We evaluate MIFO using the following metrics: (1) **mean-time-to-response** (*mttr*) is the mean latency for all the queries in the workload; (2) **makespan** [18]

gives the end-to-end response time of the workload; (3) **average speedup** is the mean of speedup (latency of query in baseline policy by latency in MIFO) obtained for all queries in the workload; (4) we maintain a time-series of **task throughput**, where each 60sec window captures the number of tasks completing in that interval; (5) **CPU utilization**, and (6) we calculate the **resource-fairness-index** (RFI) using the Jain's Fairness Index [20] for every 60sec interval of the workload. For every i^{th} query running in this interval, we calculate its deficit as $x_i = numRunningTasks_i / fairEligibility_i$. The RFI in a window for N concurrent queries is then calculated as:

$$RFI = \frac{(\sum_{i \in N} x_i)^2}{N * \sum_{i \in N} x_i^2} \quad (6)$$

Table 2 compares MIFO with other popular approaches to cluster scheduling policies for various metrics. We demonstrate the findings summarized in this table next.

7.1 Exp-Policies: MIFO vs FAIR/FIFO/SRPT

In this experiment, we evaluate MIFO against other policies for the same schedule of queries. We first created a workload for FAIR scheduling policy with a poisson arrival rate of $\rho = 5$ queries/min. We serialized this schedule and used it to run the workloads for MIFO (all three techniques, $\alpha_{bound} = large$), FIFO, and SRPT policies. Note, for SRPT, we used the query runtimes averaged over five previous executions as their estimated execution times. This was used to calculate the remaining time for the query at any scheduling instance. We term each of the runs as FAIR_5, MIFO_5, FIFO_5 and SRPT_5 where '_5' denotes the poisson arrival rate.

Observations: Figure 11 compares MIFO with other policies for our five metrics. Figure 11a shows an improved *mttr* of about 48% with MIFO_5 compared to FAIR_5. When we tried this experiment with different set (randomized) of 50 queries, we saw *mttr* gains ranging from 11% to upto 67%. We also obtained about 1.8x speedup (see Figure 11c) in MIFO_5 over FAIR_5. SRPT and FIFO have lower *mttr* values compared

Metric	FIFO	FAIR	SRPT	MIFO
MTTR	Low	High	V low if more short	Lower than FAIR
Makespan	High for later	High for all	High for long	Lower than FAIR
Throughput	Depends on length of jobs	Depends on concurrency	High	Higher than FAIR
Fairness	Favors early	Fair for all	Favors short	Favors work done
Starvation	Later starved	Not possible	Longer starved	Not possible
Overhead	Low	Depends on arrival rate	High if multiple short jobs	Depends on arrival rate
Predictability	Low for short jobs	Depends on workload	Low for long jobs	High for MIFO-eligible

Table 2: Comparison of MIFO with Other Policies

Figure 12: *Exp-Policies*: (a) CPU Utilization and (b) Task Throughput computed in every 60sec intervals of the workload’s execution.

to MIFO_5; even the speedup in MIFO_5 over these policies was observed to be worse at 0.78x and 0.83x respectively. However, MIFO_5 is seen to have a lower *makespan* (see Figure 11b) and a much better RFI (see Figure 11d) compared to these policies. MIFO_5’s maximum RFI is only slightly lesser than the maximum RFI obtained with FAIR, and the median value is seen to be higher. Figure 12a shows an improved CPU utilization over FAIR_5; but lower than FIFO_5 and SRPT_5. Compared to a median CPU utilization of 30% in FAIR, we observed a median utilization of 48% in MIFO. Note, the CPU utilization was calculated using the cumulative executor CPU time for all tasks in every 60sec interval of the workload’s execution. Our slot utilization remained high throughout for both FAIR and MIFO as we had $S \ll N$ for our cluster (see Section 3.1). The median task throughput across all intervals in Figure 12b is seen to be much higher than FAIR_5, and almost similar to SRPT_5.

Figure 13 shows the breakdown for the overall improvement observed in *CPUTime*, *SWT* and *RWT* values in MIFO compared to FAIR. We plot the cumulative time per query in *sec* on the y-axis on log scale. The cumulative time for all tasks of all queries was summed up and then averaged over the the number of queries. *CPUTime* does not show any variability; the other two components of the task run-time - *SWT* and *RWT* showed about 48% and 38% reduction respectively in MIFO. We instrumented Spark to capture per task CPU Waiting Time, and used existing Spark metrics to

Figure 13: Y-axis shows the cumulative time in *sec* (log scale) spent on an average per query in *CPUTime*, *SWT* and *RWT* in FAIR vs MIFO. *CWT*, *NWT* and *IOWT* denote the CPU, Network and IO waiting times observed respectively. We did not capture the memory waiting times for these workloads.

analyze the network (*fetchWaitTime*), and IO (*scanTime*, *shuffleWriteTime*) waiting times. These components contribute to the *RWT*, but do not sum up to it as multiple other factors like JVM GC Time, OS-scheduling wait, etc. contribute to this value as well. Our per-resource wait-time analysis was intended to help us understand the resources for which the *RWT* improved for MIFO. We observed about 48% and 47% reduction in the network and CPU waiting times respectively, whereas a mere 4% lesser *IOWT*. This lower value shows that since we do not trigger MIFO during the early IO-read and shuffle-write phases of a query’s DAG, the gains for disk IO are minimal, and are contributed by only the reduction in *shuffleWriteTime* (IO write Wait) for later stages.

7.2 Exp-Bounds: Varying Scaling Factor

We now present results for setting the MIFO Scaling Factor to different values for getting minimum benefit (MIFO_min), maximum benefit without completely blocking concurrent queries (MIFO_max), and maximum benefit by blocking concurrent queries (MIFO_large). We discussed these configurations in Section 4.1.1.

Observations: In Figure 14, MIFO_large is observed to give the maximum gain across all six metrics (we don’t show

Figure 14: Exp-Bounds: Comparison of (a) makespan, (b) average speedup, (c) task throughput and (d) Resource Fairness Index (RFI) of different bounds of MIFO i.e. MIFO_min, MIFO_max, and MIFO_large plotted against FAIR.

mttr and CPU utilization here due to space constraints), it can lead to head-of-line blocking or starvation of other concurrent queries. As a result, an administrators may prefer a configuration with MIFO_max as it is starvation-free (see discussion in Section 6.1). Observe that MIFO_max performs better than MIFO_min for all metrics, and also has a higher deviation from a desirable RFI compared to MIFO_min. In our experiment, MIFO_large was observed to give the highest RFI among the three MIFO configurations.

7.3 Exp-Techniques: Comparing DP, IA, LI

We analyze the contribution of individual techniques in MIFO i.e. dynamic prioritization (DP), interference avoidance (IA) and locality ignorance (LI) towards the overall benefit in isolation, and pairwise. We use the same baseline FAIR_5 from *Exp-Policies* experiment above. Figure 15 compares *makespan* of these executions. The suffix in the name denotes the technique that was switched on for that execution. For example, MIFO_DP_LI run denotes an execution with DP and LI turned on, but IA was configured to be off.

Observations: Observe that all of our techniques except MIFO_LI_IA gave some benefit in *makespan* over FAIR when used individually or pairwise. When the techniques were applied in isolation, dynamic prioritization (MIFO_DP) gave the highest benefit, whereas for pairwise combinations MIFO_DP_IA gave best makespan. In our experience, the benefits obtained from applying IA and LI either individually or pair-wise were highly dependant on the contention conditions on the cluster. For example, for some other workloads we observed applying only IA worsened the makespan by 43%. Even with LI, any benefit obtained in reduced SWT was offset if the tasks were launched on more contentious hosts instead. As a result, we recommend applying all three techniques together to get maximum benefit compared to FAIR as shown in MIFO_5 configuration which had all three techniques applied.

7.4 Exp-Load: Increasing Concurrency

To study the impact of increased contention, we vary the concurrency in the system by varying the poisson arrival rate of queries from 4 queries/min to 8 queries/min. We observed a maximum of 37 concurrent queries for our workload with 8 queries/min with FAIR. Note, for this experiment, the set of 50 queries chosen were same as the baseline, but their schedule was different owing to the changing arrival rate. The set of MIFO-eligible queries also remained the same. We present our evaluation to compare only the *makespan* and average speedup due to space constraints. The configuration chosen was MIFO_large to analyze the maximum benefit we get with MIFO. Figure 16 compares the performance of 5

Figure 15: We compare the benefit obtained in MIFO from our three techniques when used in isolation (MIFO_DP, MIFO_IA, MIFO_LI), pairwise (MIFO_DP_LI, MIFO_DP_IA, MIFO_LI_IA), or combined (MIFO_DP_LI_IA). We also plot our baseline workload FAIR_5.

Figure 16: (a) compares the *makespan* of each workload for increasing concurrency (poisson arrival rate ranging from 4-8 queries/min). (b) shows the average speedup obtained in MIFO over the corresponding FAIR run. (c) shows the CDF of the query completion times, and (d) compares the query run-times histogram for FAIR_8 and MIFO_8 runs.

different workloads of FAIR with MIFO where the suffixes (e.g., _5) denotes the poisson arrival rate.

Observations: Before we reason for MIFO benefits shown in Figure 16a, observe that the makespan of the workload with FAIR does not show a steady decline with increasing contentions (lower for FAIR_5 with higher load than FAIR_4). The primary reason for this unpredictability is that in a cluster scheduler that is dataflow agnostic, the arrival rate of stages within the scheduler queue affects the starvation of other concurrent stages (OSH). Moreover, since there is no interference-aware placement of tasks on hosts, some co-location patterns can result in less interferences, while others in more conflicts (MRI). The alarming spike in FAIR for $\rho = 8$ was due to increasing wait-times in the scheduler queue as more stages of concurrent queries entered the workload while waiting for others to complete on the same number of slots. Given this, MIFO gives steady improvement over FAIR for all concurrency levels (see Figure 16a for improvements in makespan, and Figure 16b for average speedup obtained over corresponding FAIR workload). Especially for $\rho = 8$, MIFO extenuates the impact due to OSH by pushing queries towards completion faster. This can be seen from Figure 16c and Figure 16d. For ease of presentation, we plot only the extreme load conditions ($\rho = 4, 8$ queries/min) in Figure 16c.

7.5 Exp-Heterogeneous/Adhoc

We could not evaluate MIFO for other workload types (machine learning or graph analytics) as we instrumented Spark-SQL's plans to make the cluster scheduler dataflow-aware. As a result, to test for heterogeneity we created 5 custom SQL queries with a strictly non-decreasing resource usage profiles. To analyze the impact of inducing adhoc queries on a workload, we selected another 10 queries randomly from TPCDS benchmark where the algorithm assumes no prior information about them. In our implementation, we do not consider adhoc queries for MIFO-based allocations, and let them run with their default FAIR-share of resources. As such, we created a set of 15 queries to be induced to a *base* run of 20 MIFO-eligible queries (see 'Identifying MIFO-Eligible Queries' earlier in this Section) arriving in a poisson rate of 5 queries/min. We generated three runs each for both FAIR and MIFO by adding 5, 10 and 15 of these adhoc and heterogeneous profile queries.

Observations: Figure 17a compares the *mttr* values for the FAIR and corresponding MIFO runs. With the addition of new queries, this value increases for both FAIR and MIFO, but we see some variability in FAIR executions. This is because performance of queries in FAIR is highly dependant on the arrival rate of stages and resource consumption profiles of other concurrent queries. Since MIFO shields the *base* queries from the impacts due to adhoc and other non-decreasing profile queries, it is seen to give consistent improvements over FAIR. This becomes more clear from Figure 17b which compares the *mttr* for only the 20 MIFO-eligible queries used in *base*. Note for these queries, the *mttr* in MIFO showed an average of 29% gain over 3 runs; whereas, for the total workload's *mttr* we observed 26% improvement.

Figure 17: Exp-Heterogeneous/Adhoc: (a) *mttr* of FAIR increases with addition of more queries to the workload. MIFO is seen to give steady improvement over FAIR. (b) The *mttr* value for only the original 20 MIFO-eligible queries increases significantly in FAIR, while we see a consistent and more predictable *mttr* value for these 20 queries with MIFO policy.

8 RELATED WORK

Interference-aware resource allocation: In cloud-based research, this is a well-studied topic but is mostly geared towards interference-management or interference-aware resource allocations amongst different co-located applications or VMs (virtual machines)[31, 38]. They provide empirical rules to manage contentions, but rely heavily on detailed application characterization[38] or dynamic provisioning of underutilized resources [31] upon detecting interferences. MIFO provides a proactive resource re-allocation scheme for shielding queries from possible interference effects.

Minimizing SWT or RWT: SWT has previously been addressed by designing efficient **admission control** schemes [1] that control the number of concurrent queries at any time. However, this can lead to under utilization of the cluster. Approaches to reduce RWT have been geared towards reducing bottlenecks for each resource type in isolation, for example improving IO performance [27], optimizing shuffle performance [12], reducing memory and GC overheads [40], reduce network bottlenecks [10], etc.

Query-Semantics Awareness: One approach to shield stages with small tasks from the impact due to OSH is to reduce the parallelism by combining their input keys. While this reduces the delays in the queue, it can potentially impact cluster responsiveness by reducing the opportunity for tiny tasks of interactive jobs [32]. It is also difficult to track and specify this option for component stages of declarative queries submitted through systems like SparkSQL [8] and Hive [37]. A recent study [17] stresses the importance of DAG-aware allocations and uses structural properties of DAGs to schedule heterogeneous DAGs; specifically, they place long-running tough-to-pack troublesome tasks first in a virtual resource-time space, and then schedule the remaining tasks without violating job dependencies. While MIFO shares the same goal of optimizing *mttr*, it also aims to provide a balanced per-queue tunable alternative for scheduling.

Hybrid Schedulers: Recent research in hybrid scheduling [14, 34] has focused on the benefits of stage-aware scheduling at worker queues. A central scheduler in *Eagle* [14] uses a Least-Work-Left (LWL) policy that matches tasks of long stages to nodes with the least queuing time for the tasks. The distributed schedulers employ a stage-progress aware Shortest-Remaining-Time-First (or SRPT) policy at individual worker nodes. SRPT queuing combined with techniques like Sticky Probes (probes for short stages keep launching their own tasks) results in reduced queuing delays and improved cluster utilization for short latency-sensitive stages. Since the work requirements of stages within a query DAG may vary significantly, it is possible that long and short stages of a query get queued and handled by separate scheduling

entities in such hybrid schedulers. In this setting, the later short stages of data analytical queries will be handled by the distributed schedulers with SRPT as their queuing policy. Additionally, the effectiveness of an SRPT policy relies heavily on accurate estimation of task runtimes. In the absence of such estimates, or due to unpredictable resource contentions, an SRPT-based allocation can prioritize wrong stages to completion while blocking others.

Approaches for progress indicators: As part of MIFO, we have studied the literature on query progress estimation [9, 23–25, 28, 29]. The predominant approach used to estimate the progress of SQL queries in database systems and the progress of DAGs of parallel jobs (e.g., MapReduce jobs) [30] in systems like Hadoop is based on estimating the percentage of tuples that remain to be processed by the query plan. Unfortunately, we found that tuple cardinality estimates are notoriously unreliable in modern systems like Hadoop and Spark. Hence, we followed the recommendation in [26], namely, to develop custom query progress estimators that are good enough for the problem to be solved. This direction led us to the approach discussed in Section 5.1 where MIFO uses task CPU time measured from past runs of a query to estimate whether the current query execution is past the knee of the curve and in the region where the query execution will take an adverse hit in SWT and RWT compared to the work remaining in the query plan. This approach worked reasonably well in our experiments as shown in the paper. In future work, we will continue to investigate how alternative query progress estimators can improve MIFO.

9 CONCLUSION

Data analytical queries constitute a significant portion of the mixed workloads running on cluster computing frameworks. The increasing heterogeneity of resource usage patterns on a shared cluster penalizes the data analytical workloads in multiple ways as we have seen in this paper. This causes undue delays in query execution times and also under-utilization of cluster. To that end, we have shown that query-semantic information can be used to make effective low-level resource allocations that reduce the resource waiting times for queries thereby improving their performance. We explored various techniques that can be triggered during a query's execution to safeguard its progress from concurrency related problems in a shared cluster.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their insightful comments that helped us improve the paper. This work was supported in part by NSF awards IIS-1408846, IIS-1423124 and IIS-1552538.

REFERENCES

- [1] 2019. Admission Control and Query Queuing. https://www.cloudera.com/documentation/enterprise/5-5-x/topics/impala_admission.html.
- [2] 2019. Apache Hadoop Capacity Scheduler. <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html>.
- [3] 2019. Apache Parquet. <https://parquet.apache.org>.
- [4] 2019. Delay scheduling should not delay some executors indefinitely if one task is scheduled before delay timeout. <https://issues.apache.org/jira/browse/SPARK-18886>.
- [5] 2019. Plotly: Modern Visualization for the Data Era. <https://plot.ly>.
- [6] 2019. Presto: Distributed SQL Query Engine for Big Data. <https://prestodb.github.io>.
- [7] 2019. TPC Benchmark™DS. <http://www.tpc.org/tpcds/>.
- [8] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. ACM, 1383–1394.
- [9] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. 2004. Estimating progress of execution for SQL queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 803–814.
- [10] Mosharaf Chowdhury and Ion Stoica. 2012. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 31–36.
- [11] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.
- [12] Aaron Davidson and Andrew Or. 2013. Optimizing shuffle performance in spark. *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep* (2013).
- [13] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
- [14] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*.
- [15] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2018. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. 135–148. <https://doi.org/10.1145/3267809.3267838>
- [16] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.
- [17] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. of the USENIX OSDI*.
- [18] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.
- [19] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 261–276. <https://doi.org/10.1145/1629575.1629601>
- [20] Raj Jain, Dah-Ming Chiu, and W. Hawe. 1998. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR* cs.NI/9809099 (1998). <http://arxiv.org/abs/cs.NI/9809099>
- [21] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: towards automated SLOs for enterprise clusters. In *Proceedings of OSDI 2016: 12th USENIX Symposium on Operating Systems Design and Implementation*. 117.
- [22] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters.. In *USENIX Annual Technical Conference*. 485–497.
- [23] Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. 2011. A statistical approach towards robust progress estimation. *Proceedings of the VLDB Endowment* 5, 4 (2011), 382–393.
- [24] Kukjin Lee, Arnd Christian König, Vivek Narasayya, Bolin Ding, Surajit Chaudhuri, Brent Ellwein, Alexey Eksarevskiy, Manbeen Kohli, Jacob Wyant, Praneeta Prakash, et al. 2016. Operator and query progress estimation in microsoft sql server live query statistics. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1753–1764.
- [25] Jiexing Li, Rimma V Nehme, and Jeffrey Naughton. 2012. GSLPI: A cost-based query progress indicator. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 678–689.
- [26] Jiexing Li, Rimma V Nehme, and Jeffrey F Naughton. 2013. Toward Progress Indicators on Steroids for Big Data Systems.. In *CIDR*.
- [27] Jialin Liu, Evan Racah, Quincey Koziol, and Richard Shane Canon. 2016. H5Spark: Bridging the I/O Gap between Spark and Scientific Data Formats on HPC Systems. *Cray User Group* (2016).
- [28] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. 2004. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 791–802.
- [29] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. 2005. Increasing the accuracy and coverage of SQL progress indicators. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 853–864.
- [30] Kristi Morton, Magdalena Balazinska, and Dan Grossman. 2010. Para-Timer: a progress indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 507–518.
- [31] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*. ACM, 237–250.
- [32] Kay Ousterhout, Aurojit Panda, Josh Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. 2013. The Case for Tiny Tasks in Compute Clusters.. In *HotOS*, Vol. 13. 14–14.
- [33] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307. <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout>
- [34] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. 2016. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 36.
- [35] V Satopaa, J Albrecht, D Irwin, and B Raghavan. 2011. Finding a knee in a haystack: Detecting knee points in system behavior. 166–171. In *31-st International Conference on Distributed Computing Systems*. See: <http://www1.icsi.berkeley.edu/barath/papers/kneede-simplex11.pdf>.
- [36] spark-sql-perf team. 2018. Spark SQL Performance. <https://github.com/databricks/spark-sql-perf>.

- [37] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *PVLDB* 2, 2 (2009), 1626–1629.
- [38] Nedeljko Vasić, Dejan Novaković, Svetozar Miućin, Dejan Kostić, and Ricardo Bianchini. 2012. Dejavu: accelerating resource allocation in virtualized environments. In *ACM SIGARCH computer architecture news*, Vol. 40. ACM, 423–436.
- [39] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.
- [40] Luna Xu, Min Li, Li Zhang, Ali R Butt, Yandong Wang, and Zane Zhenhua Hu. 2016. MEMTUNE: Dynamic Memory Management for In-memory Data Analytic Platforms. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 383–392.
- [41] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleggy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 265–278. <https://doi.org/10.1145/1755913.1755940>
- [42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. <http://dl.acm.org/citation.cfm?id=1863103.1863113>

A MIFO IS STARVATION-FREE

In a mixed workload where long running batch queries execute concurrently with short interactive queries, non-starvation is an important consideration. While a $MIFO_{large}$ configuration can benefit a query, it will block other queries till its completion. Thus, a lesser value that is still greater than the lower-bound may be desired in practice to prevent complete starvation of other queries. A value of $\alpha_{q,t}$ set to η^{max} achieves this result. It gives the maximum benefit for query q while ensuring at least one task of all other concurrent queries is running. This avoids head-of-line blocking - a problem typically seen in FIFO and SRPT.

Next, we provide approximate bounds on the length of a query q such that it does not make the performance of other concurrent queries worse than FAIR. Consider two data analytical queries q_s and q_l as the shortest and the longest queries under consideration at time t . Let W_s and W_l be their respective total work (or compute) requirements, and N is the number of total slots in the cluster. Let us consider a worst case scenario for q_s getting blocked behind q_l until q_l finishes execution. That is, q_s starts at the same instance t when q_l crosses its MIFO trigger age λ_l . For q_s to still benefit in MIFO compared to FAIR, the time for which q_s will be blocked by q_l after t plus its runtime in MIFO should be strictly less than the runtime of q_s in a FAIR allocation. That is,

$$\tau_s^{fair} > \tau_s^{blocked_mifo} + \tau_s^{mifo} \quad (7)$$

Let $\tau_s^{fair} = \frac{W_s}{\frac{1}{k} * N}$ be the time q_s takes to finish work W_s in FAIR with k concurrently running queries; the value $\frac{1}{k} * N$ gives the share of resources q_s will get in FAIR. Let MIFO be triggered for the shorter query at age λ_s . The value of τ_s^{mifo} is then computed as:

$$\tau_s^{mifo} = \frac{W_s * \lambda_s}{\frac{1}{k} * N} + \frac{W_s * (1 - \lambda_s)}{N - (k - 1)} \quad (8)$$

In MIFO, after q_l crosses λ_l fraction, the amount of work remaining is $(1 - \lambda_l) W_l$; this gives $\tau_s^{blocked_mifo} = \frac{(1 - \lambda_l) W_l}{N - (k - 1)}$ as query q_l gets $N - (k - 1)$ resources in our scenario leaving 1 slot for q_s . Substituting for the values of τ_s^{fair} , τ_s^{mifo} and $\tau_s^{blocked_mifo}$, and solving further, we get the following bound on the length of any concurrent query q_s such that query q_l does not starve it:

$$W_s > \frac{W_l * (1 - \lambda_l)}{(k - 1) * (1 - \lambda_s)} \quad (9)$$

For example, assuming we trigger MIFO at age 0.7 for both q_s and q_l , if the length of long-running query $W_l = 1000sec$ and the number of concurrent queries is 10 then all shorter queries whose work requirement is greater than 100 will benefit. A concurrent query whose remainder work requirement is shorter than this value will be delayed, but not starved. In production environments, where schedulers typically partition long and short jobs to execute on separate partitions of the cluster, MIFO can be used to schedule among long jobs where there is not much deviation in query lengths.

B EXP-SCALABILITY: SCHEDULING OVERHEAD

MIFO needs minimal instrumentation to capture query progress information at runtime, and to make the cluster scheduler DAG structure aware. The actual logic to compare stages and select a *best-fit* stage, and to launch the tasks on the offered slots is inherent to any scheduler irrespective of the scheduling policy. We profiled our instrumentation in Spark to output overhead incurred in different phases of FAIR and MIFO execution - (1) *mifoLogicTime* denotes the time required to annotate the SQL plans with query information, and the time taken by the *Query Execution Monitor* (see architecture in Section 6) to retrieve this information at runtime. For FAIR runs, this value is zero. (2) *schedulerCompareStagesTime* gives the time required to compare the stages in the queue and output the *best-fit* stage based on their priorities. For MIFO runs, this includes the combined time taken to check if a query has crossed MIFO Trigger age, to compute the MIFO Scaling Factor, and to compute a new weight during DP. (3) *timeTakenToChooseBestOffer* is incremented when resource offers are received by the scheduler to output a schedule. For MIFO runs, this includes the time taken to generate an

Figure 18: Each bar denotes the overhead time spent in different phases of code as a percentage of the workload’s *makespan*. The suffixes in x-axis labels denote the poisson arrival rates for each run. The numbers on the bars show the maximum number of concurrent queries that were running at any point in each of the runs.

alternate schedule by applying IA technique. (4) Finally, *taskMetricsCaptureTime* is the overhead added by our code to send query progress data on heartbeats, and to aggregate the information for maintaining per-query resource waiting time values used in IA. Again, this value is zero for all FAIR runs. We present results for the runs for increasing poisson arrival rates in Figure 18 to demonstrate how increasing concurrency impacts these values.

Observations: The overhead of *taskMetricsCaptureTime* does not add significant overhead since we piggyback our data on the Spark default heartbeats. The *Query Execution Monitor* maintains tree-maps for different execution metrics per query; as a result, the retrieval in *mifoLogicTime* is of $O(\log n)$ complexity with a $O(n)$ storage overhead. MIFO comparator (*schedulerCompareStagesTime*) incurs the highest overhead, but it is comparable to the time spent by FAIR in this phase. It is slightly more in MIFO as we check eligibility of query for MIFO-allocations, apply our techniques logic and compute new weights. Since the comparator is invoked for every pair of stages pending in the scheduler queue to select a *best-fit* stage, the number of invocations is same in FAIR and MIFO. As a result, the overhead in MIFO will always be a proportional to the number of pending stages in the queue. From Figure 18, it can be seen that the total overhead does not exceed beyond 0.4% even with increase in concurrent queries from 15 to 31.

C BALANCED OPTIMIZATION

In this section, we see how MIFO can provide a balanced alternative to both FAIR and FIFO as it can be tuned to mimic

either of these policies based on the queue optimization requirements.

C.1 Problems with FIFO

While FIFO provides a good ordering strategy for low-level processes, we observe that it is not the best option in a cluster where delays due to multiple-resource interferences can go undetected. To illustrate, consider three scenarios: (a) If multiple queries are submitted to a queue in a short window (e.g. poisson arrivals), a FIFO-based scheduling can result in longer queue wait times. Since the FIFO allocations happen at stage-level based on stage-ids, it is possible that for queries that were submitted simultaneously their stage-ids are interleaved. In such a scenario, these queries will proceed alternately while blocking each other. (b) If these queries have a distinct complimentary resource usage requirements, it can result in enormous under-utilization of the resources. (c) Another problem with FIFO is that since all tasks of a single stage are launched when slots become available, more tasks of similar resource utilization pattern execute on the cluster. For example, this can result in enormous CPU and IO waiting times if all tasks of some initial stages of the query are launched.

Despite these shortcomings, FIFO is often desired as the default policy as it gives lower mean response times compared to FAIR. Next, we show how MIFO can provide a balanced alternative to both FAIR and FIFO to achieve lower mean response times along with improved cluster utilizations depending on the queue-optimization objective.

C.2 Tune Between FAIR and FIFO

Users often submit workloads on shared clusters are often partitioned in different queues, each dedicated for a user. In such a setting, it is often desirable to achieve high fairness amongst the users. However, the users who are submitting their queries to a queue often have different optimization objectives. For example, lower *mttr* than what is provided by FAIR. Today, the only alternative is to use FIFO, thus making the users compromise on their fairness within a queue requirement. MIFO can provide a balance alternative in such a scenario, where it can be configured to achieve high fairness at the cost of higher *mttr* for inter-queue scheduling; and configured to achieve a lower *mttr*, but comparable fairness for queries submitted to the same queue. In this section, we review how tuning the parameters in MIFO can achieve this objective.

An admin can specify an optimization objective for a workload using the *mttr* and RFI system-wide metrics. Users today are not able to attain a good balance between the two extreme optimization objectives of FAIR and FIFO based on the

changing workloads. MIFO aims to provide this balance with a tunable framework that helps regulate the optimization goal for each queue. To get the least *mttr* comparable to FIFO users can trigger MIFO very early using the H – KNEE heuristic only. Ideally, a value of $\lambda^q = 0$ results in FIFO like behavior. However, these settings will result in a higher RFI value. As we increase the value of λ^q , the *mttr* and RFI values increase, thus, approaching a FAIR behavior. A value of $\lambda^q = 1$ (we do not trigger MIFO at all) mimics FAIR. Auto-configuring the MIFO Trigger Age (λ^q) for each query based on the optimization objective per queue (desired balance levels of *mttr*, RFI) is a subject of our on-going research.

C.3 Experiment *Exp-Mimic*: Varying Trigger Age and Scaling Factor

In this experiment, we use a smaller workload consisting of 20 TPCSDS queries submitted in a poisson arrival rate of 5 queries/minute. Here, we report the actual values recorded from each run. In order to demonstrate how MIFO can be used to reproduce the objective functions of FAIR or FIFO, we do not use the knee-detection algorithm for filtering queries for MIFO-eligibility in this experiment. Instead, we use a system-wide MIFO trigger age threshold λ_q (all queries in the workload are accelerated with MIFO techniques) and vary it from 0 to 1. The purpose of this experiment is to evaluate the impact of triggering MIFO early ($\lambda_q = 0$) vs late ($\lambda_q = 1$ where we don't trigger MIFO at all). We evaluate this impact for varying bounds of the MIFO Scaling Factor.

Figure 19: MIFO gives low *mttr* (mimics FIFO) for a very large MIFO Scaling Factor and low MIFO Trigger Age threshold (λ_{conf}) values. Whereas, it gives *mttr* (mimics FAIR) for the workload for a low Scaling Factor and a high λ_{conf} .

Observations: We plot the *mttr* and RFI values in surface plots shown in Figures 19 and 20 respectively, where MIFO can be seen to mimic the behavior of FIFO and FAIR policies. For example, when MIFO_{large} is run with low threshold values of trigger age, MIFO behaves similar to FIFO where it advances queries on first-come basis as they reach there work done thresholds early on. For any other age value it gives a balanced optimization between *mttr* and RFI. We found this approach useful to help determine the value of λ_{conf} for a workload based on the resource usage profiles of its queries. In practice, while this is an expensive exercise, it can be useful to sample the workload offline to derive λ_{conf} .

Figure 20: MIFO gives low RFI value (mimics FIFO) for a very large MIFO Scaling Factor and low MIFO Trigger Age threshold (λ_{conf}) values. Whereas, it gives high RFI (mimics FAIR) for the workload for a lower Scaling factor and a high λ_{conf} .