RESEARCH-ARTICLE

# Balancing Fairness and Performance in Multi-User Spark Workloads with Dynamic Scheduling



**Published:** 19 November 2025



SoCC '25: ACM Symposium on Cloud Computing
*November 19 - 21, 2025*
*Online, USA*

**Conference Sponsors:**
SIGOPS
SIGMOD

SoCC '25: Proceedings of the 2025 ACM Symposium on Cloud Computing (November 2025)
https://doi.org/10.1145/3772052.3772214
ISBN: 9798400722769

# Balancing Fairness and Performance in Multi-User Spark Workloads with Dynamic Scheduling

Dāvis Kažemaks
davis.kazemaks@gmail.com
Delft University of Technology

Laurens Versluis
laurens.versluis@asml.com
ASML

Burcu Kulahcioglu Ozkan
b.ozkan@tudelft.nl
Delft University of Technology

Jérémie Decouchant
j.decouchant@tudelft.nl
Delft University of Technology

## Abstract

Apache Spark is a widely adopted framework for large-scale data processing. However, in industrial analytics environments, Spark's built-in schedulers, such as FIFO and fair scheduling, struggle to maintain both user-level fairness and low mean response time, particularly in long-running shared applications. Existing solutions typically focus on job-level fairness which unintentionally favors users who submit more jobs. Although Spark offers a built-in fair scheduler, it lacks adaptability to dynamic user workloads and may degrade overall job performance. We present the User Weighted Fair Queuing (UWFQ) scheduler, designed to minimize job response times while ensuring equitable resource distribution across users and their respective jobs. UWFQ simulates a virtual fair queuing system and schedules jobs based on their estimated finish times under a bounded fairness model. To further address task skew and reduce priority inversions, which are common in Spark workloads, we introduce runtime partitioning, a method that dynamically refines task granularity based on expected runtime. We implement UWFQ within the Spark framework and evaluate its performance using multi-user synthetic workloads and Google cluster traces. We show that UWFQ reduces the average response time of small jobs by up to 74% compared to existing built-in Spark schedulers and to state-of-the-art fair scheduling algorithms.

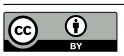
## CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Scheduling**.

## Keywords

Multi-user, Scheduling, Fairness, Spark

**ACM Reference Format:**
Dāvis Kažemaks, Laurens Versluis, Burcu Kulahcioglu Ozkan, and Jérémie Decouchant. 2025. Balancing Fairness and Performance in Multi-User Spark Workloads with Dynamic Scheduling . In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772214




*SoCC '25, Online, USA*

ACM ISBN 979-8-4007-2276-9/2025/11
https://doi.org/10.1145/3772052.3772214

## 1 Introduction

The exponential growth of online data [1] has driven an increasing demand for scalable, high performance batch processing frameworks. Among these, Apache Spark, or simply Spark, has emerged as a leading platform [2] due to its in-memory computation model, rich ecosystem of libraries (e.g., SQL engine [3]) and integration with major cloud providers such as Databricks and Amazon Web Services. In industrial analytics environments, Spark is commonly deployed as a shared, multi-user system, where users concurrently submit batch jobs for execution. These environments rely on fair scheduling mechanisms to manage constrained computing resources and deliver acceptable job response times to all users.

In most systems, such a fair scheduler is typically implemented through cluster managers such as YARN or Apache Mesos to control the job flow before it reaches Spark's scheduler. Users insert their main execution code within a Spark application, which additionally interfaces with the cluster manager to request resources. Cluster managers assign resources across multiple Spark applications, each launched by a different user. Prior works have explored such approaches, focusing on equalizing resource distribution [4] or meeting job deadlines through application-level scheduling [5]. Additionally, Spark now offers dynamic resource allocation (DRA), which adjusts the amount of resources based on the application workload. However, DRA does not enforce strict fairness policies nor explicitly aim to minimize execution time, in particular when the system is congested.

However, launching an application for each workload induces a significant performance overhead in environments where many small jobs need to be executed. This overhead comes from to the time required to start the driver program, allocate workers, or launch executors [6]. Furthermore, if the expected workloads are highly parallelizable, all applications benefit from having more executors.

Industrial analytics platforms therefore typically rely on a single long-running Spark application that continuously listens for incoming user jobs and executes them for performance. While this solution reduces setup delays, it centralizes scheduling responsibilities within Spark's scheduler, which is ill-equipped to handle complex fairness and performance trade-offs. Spark's built-in scheduling options, i.e., first-in-first-out (FIFO), fair, and fairness pools, fall short in achieving both fairness across users and low job response times, particularly in multi-user long-running Spark applications where job arrival patterns and workloads are highly dynamic [7, 8]. Alternative schedulers have been proposed to ensure

fairness among jobs [8–14]. However, by focusing on job fairness, these schedulers allow users who schedule more jobs to be allocated more resources than those with fewer jobs, which is unfit for our multi-user and multi-job batch processing environment.

Task skew and priority inversions accentuate fairness issues in Spark environments. Both problems stem from Spark's default task partitioning strategy, which decomposes a job's input data into multiple partitions, each executed in parallel as an individual task. Task skew occurs when one or more partitions take significantly longer to execute than others. This leads to underutilization of resources, as the straggling partition may use a single core while other cores remain idle, effectively reducing the job's parallelism and extending its completion time. Priority inversions occur when long-running tasks from lower-priority jobs occupy executors, thereby blocking higher-priority jobs from making progress. Such inversions have been observed in Spark deployments [15] and are particularly problematic when poor partitioning results in jobs that cannot be preempted or rescheduled efficiently [8].

We propose User Weighted Fair Queuing (UWFQ), a novel Spark scheduler that ensures bounded user-job fairness while significantly reducing job response times. UWFQ builds upon the virtual time concept introduced by Cluster Fair Queuing (CFQ) [8] by incorporating both user and job context awareness. Additionally, UWFQ dynamically adjusts task granularity based on estimated runtimes to minimize task skew and mitigate priority inversions.

As a summary, we make the following **contributions**:

We introduce UWFQ, a new scheduler that ensures bounded user-job fairness while reducing the average response time of jobs. UWFQ operates by simulating a virtual fair scheduler on currently active jobs in the system, and assigning the highest priority to jobs that would complete the earliest in the virtual scheduler.

We propose a dynamic partitioning method that utilizes the estimated runtime of jobs to determine the optimal number of input partitions to use, thereby avoiding priority inversion caused by limited preemption in Spark and preventing job response time extension due to task runtime skew. This dynamic partitioning algorithm provides more flexibility to UWFQ and further improves the system's performance.

We implement UWFQ with dynamic partitioning in the Spark framework, and evaluate its effects on fairness and job average response times. We consider synthetic micro-user workloads that highlight certain scenarios that most schedulers handle inefficiently, and real Google cluster traces [16]. Our results indicate that UWFQ with runtime partitioning reduces the response times of small and medium-sized jobs by up to 74% in homogeneous workloads compared with a plain user-job fairness implementation. We additionally showcase that UWFQ always does equally good or better than CFQ, and almost always outperforms Spark's built-in fair scheduler.

## 2 System Model and Objectives

This section provides necessary background on Spark and specifies our objectives.

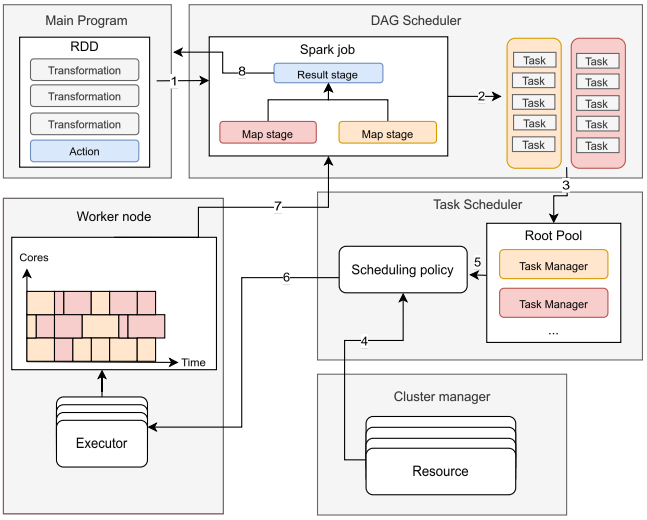



**Figure 1: Execution of user-constructed RDDs on cluster resources.**

### 2.1 Apache Spark

Apache Spark is an analytics engine for large-scale data processing across multiple programming languages. It includes a rich set of modules that support diverse workloads, including SQL querying, real-time stream processing, machine learning and graph analytics. Most of these high-level operations are compiled down to low-level API calls on Resilient Distributed Datasets (RDD), Spark's core abstraction for distributed data processing.

*2.1.1 Job execution.* Figure 1 illustrates a simplified job execution. Users construct their jobs by applying transformations on RDDs and finalizing results by issuing an action, which internally triggers a Spark job (Step 1). Once a Spark job is submitted, it is picked up by the DAG Scheduler, which breaks down the job into stages and constructs a DAG dependency graph between them. This graph is created by first creating a result stage from the associated action and recursively adding dependent map stages until no dependencies are found. Stage input is then partitioned into tasks (Step 2) and submitted to the Task Scheduler if all of their dependencies are satisfied (Step 3). The Task Scheduler keeps track of all schedulable units using a Root Pool that contains both individual stages and other layers of pools used for establishing priority hierarchies.

Within the Task Scheduler, stages are represented as Task Managers that additionally keep track of their task state. Once the cluster manager offers resources to the Task Scheduler (Step 4), it sorts the Root Pool (Step 5) according to the defined scheduling policy and submits each task one by one to be run on the executors (Step 6). Once all tasks of a Task Manager have been executed, the stage dependency graph is updated (Step 7), and any stage without pending dependencies is submitted to the Task Scheduler. Once the result stage is finished, the result is returned to the Main Program (Step 8). While scheduling decisions are mainly controlled by a scheduling policy, the scheduler always adheres to the dependency structure imposed by other layers.

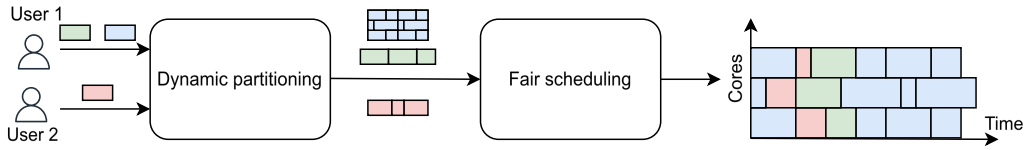



**Figure 2: System outline. Here, user 1 schedules a long job (blue), followed by a short job (red) from user 2 and another smaller job by user 1 (green). Users and jobs benefit from a low response time thanks to dynamic partitioning and fair scheduling.**

*2.1.2 Partitioning.* Stage input data is split into multiple partitions that can be operated on in parallel. For each partition in a stage, a task is created, which performs the stage-defined operations on its associated data partition. This is the main concept that allows Spark to parallelize job execution. Partitioning of stage data can be performed in two distinct phases: once when the data is first loaded into a stage, and during Adaptive Query Execution (AQE) optimizations, where partitions can be coalesced to reduce their number. When data is first retrieved, the number of partitions is determined by dividing the data equally among the available cores on the allocated executors. This allows Spark to maximize the parallelization of stage execution on executors. AQE, on the other hand, starts with 200 partitions and then reduces them to a more appropriate number based on the recommended partition size, or again, trying to maximize parallelism.

*2.1.3 Built-in Schedulers.* Spark has two built-in schedulers: first-in first-out (FIFO) and fair. The FIFO policy schedules Spark jobs in the order of their arrival, while the fair scheduler tries to equalize the number of running tasks across all submitted stages. To introduce more fairness levels, Spark also provides fairness pools that allow for defining a fairness hierarchy among jobs. During task scheduling, first, the pool with the highest priority is selected using the root scheduling policy. Then, within this pool, the task with the highest priority is selected according to a pool scheduling policy. The two pool scheduling policies that are currently offered are FIFO and Fair. Once the priority of a task is determined, it is allocated to the resource with the highest task locality.

While Spark provides several scheduling options, it does not support fair user-level scheduling for long-running applications. Its fair scheduling algorithm only considers job-level fairness, rather than user-level fairness, creating conditions where users with more active stages receive more resources. While fairness pools provide a high degree of configurability to create fairness among users, the user pools are configured only at the start of the application. This means that for a long-running application with a dynamic user base, this option is also insufficient.

## 2.2 Objective: User-Job Fairness

Our target analytics platform accommodates multiple users who may schedule several concurrent jobs. Each user in the system is entitled to an equal share of the system resources and would like their jobs to execute as quickly as possible. Jobs only provide utility once they are fully finished; hence, having a steady progression of jobs is not necessary. However, jobs should not be starved and finish within a reasonable amount of time relative to their runtime. Jobs in the system have no explicit priority, hence, job priority should only be derived from their submission time and runtime.

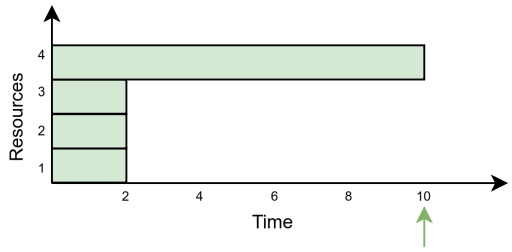
Since we do not find any existing concrete definition of fairness that would perfectly fit the target system's needs, we extend an existing definition, namely, fair share scheduling, to encapsulate our fairness objective. We define **User-Job Fairness (UJF)** as an extension of fair share scheduling, where weights for jobs would be obtained by distributing resource shares evenly among active users, and then distributing user shares among jobs belonging to that user. We formulate this using $R_k = \frac{R}{N_u}$ and $R_i = \frac{R_k}{N_j^k}$, where $R$ is the total amount of resources, $R_k$ is user $k$'s fair share, $N_u$ is the number of active users, $R_i$ is the share for job $i$, and $N_j^k$ are active jobs of user $k$. User-job fairness is achieved when resources are distributed exactly proportionally to all jobs' shares. This definition provides multi-level fairness, where each user is always entitled to an equal amount of resources, and jobs associated with the same user do not steal resources from other users, while still ensuring gradual progression without starvation. We also define bounded UJF, where a scheduling algorithm is considered bounded by UJF if every job's finish time $f_i$ is within a constant time $C$ of the time it would finish within a UJF schedule $\hat{f}_i$, i.e., $f_i - \hat{f}_i \leq C$.

# 3 UWFQ: User Weighted Fair Queuing

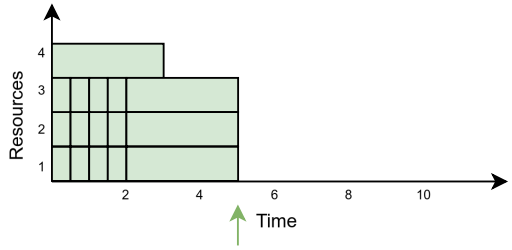
This section presents UWFQ, a Spark scheduler that implements response time-efficient user-job fair scheduling, and a dynamic partitioning algorithm that uses job runtimes to improve parallelizability of jobs, as illustrated in Figure 2. The extended version of this paper [17] contains a proof of UWFQ. Intuitively, when a user schedules a job, its input data is first partitioned into smaller slices of data using a dynamic partitioning algorithm to reduce the skews and long executor reservation times caused by large tasks, which can negatively impact performance. Once the job's input is partitioned, it is forwarded to the fair scheduler to determine its priority. UWFQ determines the priority of each job by estimating its finish time in a user-job fair scheduling system and assigning a priority to jobs in the order of their completion time. Finally, tasks of each job are scheduled onto the executors in the order of their priority.

## 3.1 Job Context Awareness

In Spark, the highest level of work unit abstraction is a Spark job, which encapsulates all job priority-related metadata. However,




**(a) Stage data partitioned using static partitioning**




**(b) Stage data partitioned using UWFQ's partitioning**

**Figure 3: Impact of task skew on job runtime. The default task partitioning leads to high completion time, in (a), while UWFQ's dynamic partitioning, in (b), reduces it. The green arrows indicate the finish time of the job.**

this is insufficient for an analytics system, since a single analytics job may represent multiple Spark jobs. Because users only receive utility from analytics jobs that have fully finished, it is more efficient to model job priorities with respect to their highest abstraction level. We therefore embed this job context into every Spark job that is created, such that it will be scheduled with respect to its corresponding workflow or query. This allows jobs to be scheduled in a much more effective sequence to optimize the response times, rather than having to interleave with every simultaneous job.

## 3.2 Dynamic Partitioning

Spark uses the estimated size of input data to partition stages across multiple executors. This partitioning method ensures that each stage can run on all cores in the system (if available), maximizing parallelism and throughput of the underlying job. However, this partitioning does not account for the runtime of these partitions and can introduce skews and priority inversion in certain cases.

For example, one of the partitions in Figure 3 (a) runs 5x longer than the others. Because of this skew, the system does not fully utilize all its available resources, and the response time of the job is delayed. Priority inversion can also be observed, e.g., when a long job completes before a higher priority job can receive any resources to execute on, as illustrated in Figure 4 (a). Since Spark tasks are not preemptable, if a longer job arrives just before a higher priority job, it cannot be interrupted and will delay the higher priority job.

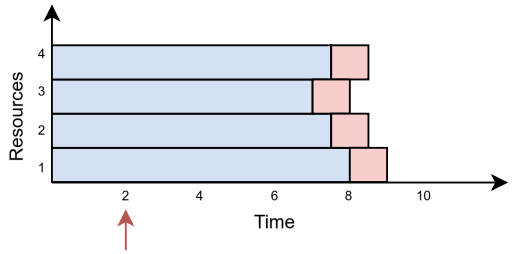
We introduce runtime partitioning to mitigate skews and priority inversions caused by default Spark partitioning of the stage input data. Runtime partitioning attempts to split the stage input into partitions that run in constant time, allowing tasks to be distributed more evenly and reducing the maximum amount of time a task can reserve an executor. The number of partitions is computed as $\frac{Stage\ runtime}{ATR}$ while their size is computed as $\frac{Total\ input\ size}{Partition\ amount}$, where Advisory Task Runtime ($ATR$) is the user or system-defined desired task runtime that the partitions are expected to run for. First, we need to collect or estimate accurate *Stage runtime*s, which are necessary to perform this partitioning method. Once the stage arrives at the dynamic partitioning component, *Partition amount* can be calculated to determine the number of partitions the data will be split into, which is then converted into *Partition size* that will be practically used to distribute the input data across partitions. After splitting the data, a task will be created for each partition, which will then be eventually scheduled on the executor according to its job's priority.

By adjusting the $ATR$ value, users or the system can control the granularity of all tasks running on the executors. By setting a relatively low advisory task runtime, we can mitigate both skews and priority inversions that were present in the original partitioning method. Skews are avoided by applying more aggressive partitioning, where the stage data can be split into significantly more partitions than there are cores in the system. Because of this, the skewed partitions are additionally split, which allows for spreading the runtime more evenly across executors, as seen in Figure 3 (b). Priority inversions are also mitigated by this, as tasks release executors much faster, allowing higher-priority tasks to be assigned much quicker than they would under regular partitioning. However, some overhead is introduced by having many active tasks in the system, hence, the $ATR$ value should not be set too low.

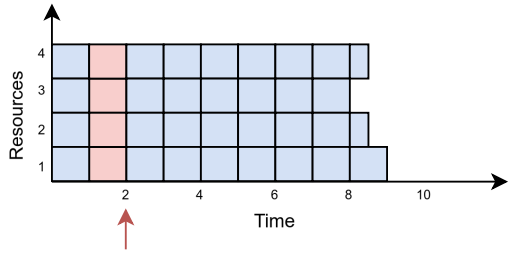
## 3.3 Fair Scheduling

UWFQ reuses principles originally proposed by Parekh et al. [18] to design WFQ. Specifically, it uses virtual time to efficiently compute job finish times under Generalized processor sharing (GPS) and leverages these virtual finish times to schedule jobs that can complete earlier than they would under GPS, while remaining bounded by it. However, these principles must be adapted to work for a user-job fair environment. We use virtual time in 2 layers: user virtual time and global virtual time. User virtual time is assigned to each user and functions similarly to regular virtual time, serving as the basis for obtaining job completion times under GPS for that specific user. Global virtual time is shared across all users and is used to determine global virtual deadlines of jobs, which are used to establish priority among all active jobs in the system.

For example, in a scenario where users compete for the same static resources, each having highly parallelizable workloads. By equally allocating resources to each user, the runtimes of jobs are extended due to not achieving optimal parallelism. However, if we were to run these jobs sequentially, each job would be able to utilize all available resources. By assigning a deadline to each job based on its expected finish time, we obtain a global order that allows quicker jobs to complete faster, while still ensuring that large jobs eventually complete once their deadline approaches. To calculate these deadlines, each user first orders its jobs using user virtual time, and then assigns a deadline to each job with respect to global virtual time, such that jobs of multiple users can be compared.




**(a) Priority inversion using static partitioning**




**(b) Priority inversion using UWFQ's partitioning**

**Figure 4: Priority inversion of red (high priority) and blue (low priority) jobs. In (a) the red job has a smaller runtime and higher priority, but only executes once the blue job's tasks have finished. But in (b), thanks to UWFQ's dynamic partitioning, the red job is able to execute earlier. The red arrow indicates the expected finish time of the red job.**

*3.3.1 Algorithm.* We show the assignment of job deadlines in Algorithm 1. To create a schedule, we order the jobs based on their assigned deadlines. The algorithm centers around the user entity $U_k$ that keeps track of its associated jobs and virtual time $V^k_{user}$. User virtual time $V^k_{user}$ is used to calculate user deadlines $D_{user}$ for incoming jobs $J_i$. Job order depends on both the user virtual time $V^k_{user}$ upon arrival and the job's slot-time $L_i$. We define job slot-time $L_i$ as the time needed to execute all of job's tasks on a single core sequentially. To allow for dynamic priorities, we also add a scalar $U_w$ for each user. This is set to 1 in the case where users have equal priorities, but can be adjusted to favor certain users in the system, Once the job is ordered in the user job set $S^k_{jobs}$, its global deadline $D^i_{global}$ can be derived. This is done by utilizing job's slot-time $L_i$ and user virtual arrival time $V^k_{arrival}$, which is measured from the current virtual global time $V_{global}$ once the user schedules their first job. The deadline is then calculated by sequentially adding each job's scaled $L_i$ to $V^k_{arrival}$ to obtain the expected virtual finish times of jobs across all users.

*3.3.2 Updating virtual time.* In 2-level virtual time, we have to update both the global time and virtual time for each user. Algorithm 2 updates the global virtual time and Algorithm 3 updates a user's virtual time. We split these algorithms in functions that we explain in the following.

**updateVirtualTime**. To update virtual time, we must iterate over all existing users in the order of their completion time. This is necessary to advance the global virtual time correctly, as it progresses at a rate proportional to the number of active users. We

**Algorithm 1** Job deadline assignment under UWFQ

**Globals:** A set $S_{users}$ of tuples of users $U$ and their virtual arrival times $V_{arrival}$ ; Global virtual time $V_{global}$

**Input:** Current time $T_{current}$; User $k$ $U_k$; Job $i$ arrival time $T^i_{arrival}$; Job $i$ $J_i$ duration $L_i$;

*// Phase 1: update system*

1: updateVirtualTime($T_{current}$) ▷ See Algorithm 2
2: **if** $(U_k, \_) \notin S_{users}$ **then**
3: $V^k_{arrival} \leftarrow V_{global}$
4: $S_{users} \leftarrow S_{users} \cup (U_k, V^k_{arrival})$
5: **end if**

*// Phase 2: calculate user deadline and insert job $J_j$ into set of user jobs*

6: $V^k_{user} \leftarrow$ getUserVirtualTime($U_k$)
7: $D^i_{user} \leftarrow V^k_{user} + L_i * U_w$
8: $S^k_{jobs} \leftarrow$ getUserJobs($U_k$) ▷ $S_{jobs}$ is a sorted set of tuples on virtual user deadlines $D_{user}$
9: $S^k_{jobs} \leftarrow S^k_{jobs} \cup (J_i, L_i, D^i_{user})$

*// Phase 3: update user job global virtual deadlines*

10: $(J_{first}, L_{first}) \leftarrow$ getEarliestUserDeadline($S^k_{jobs}$)
11: $V^k_{arrival} \leftarrow$ getUserVirtualArrivalTime($U_i$)
12: $D^{first}_{global} \leftarrow V^k_{arrival} + L_{first} * U_w$
13: setJobDeadline($J_{first}, D^{first}_{global}$)
14: $D^{previous}_{global} \leftarrow D^{first}_{global}$
15: **for** each $J_a$ in $S^k_{jobs}$ sorted by $D^a_{user}$ and $J_a \neq J_{first}$ **do**
16: $D^a_{global} \leftarrow D^{previous}_{global} + L_a * U_w$
17: setJobDeadline($J_a, D^a_{global}$)
18: $D^{previous}_{global} \leftarrow D^a_{global}$
19: **end for**

first determine the user's share, $R_{user}$, and use it to compute the user's actual finish time, $T^k_{finish}$. If $T^k_{finish}$ is after the current time $T_{current}$, we can conclude that the current and following users have not finished their jobs yet. If $T^k_{finish}$ is before current time $T_{current}$, then each remaining user should update until $T^k_{finish}$ with the current user share $R_{user}$, to have their virtual times up to date before the user leaves the system and distributes their share among active users. Once we have handled all leaving users, we can progress virtual time till the current time $T_{current}$. Since no user will leave the system during this period, the user share, $R_{user}$, will remain consistent and progress all users equally.

**getUserFinishTime**. We obtain the user's finish time $T_{finish}$ as the time its last job finishes. Since all user jobs are ordered based on their user virtual deadlines, we can trivially get the latest finished job by taking the last element from the user job set. To convert from global virtual deadline $D^{latest}_{global}$ to real time, we can calculate the difference between current global virtual time $V_{global}$ and the deadline $D^{latest}_{global}$, and divide it by the user share $R_{user}$. The difference represents the global virtual time that will progress between now and the time the job ends, and dividing by the user share $R_{user}$

**Algorithm 2** Virtual time updating

**Globals:** A set $S_{users}$ of tuples of users $U$ and their virtual arrival times $V_{arrival}$ ; Global virtual time $V_{global}$; Previous update time $T_{previous}$; Total resources of the system $R$

**function** updateVirtualTime($T_{current}$)
  **for** each $(U_k, \_)$ in $S_{users}$ sorted by $D_{global}$ **do**
    $R_{user} \leftarrow \frac{R}{|S_{users}|}$
    $T^k_{finish} \leftarrow$ getUserFinishTime($U_k, R_{user}$)
    **if** $T^k_{finish} > T_{current}$ **then**
      **break**
    **end if**
    $S_{users} \leftarrow S_{users} \setminus (U_k, \_)$
    progressVirtualTime($T^k_{finish}, R_{user}$)
  **end for**
  $R_{user} \leftarrow \frac{R}{|S_{users}|}$
  progressVirtualTime($T_{current}, R_{user}$)
**end function**
**function** getUserFinishTime($U, R_{user}$)
  $D^{latest}_{global} \leftarrow$ getLatestDeadline($U$)
  $T_{spent} \leftarrow (D^{latest}_{global} - V_{global})/R_{user}$
  $T_{finish} \leftarrow T_{previous} + T_{spent}$
  **return** $T_{finish}$
**end function**
**function** progressVirtualTime($T, R_{user}$)
  $T_{passed} \leftarrow T - T_{previous}$
  $V_{global} \leftarrow V_{global} + T_{passed} * R_{user}$
  **for** each $(U_k, \_)$ in $S_{users}$ **do**
    updateUserVirtualTime($U_k, R_{user}, T$)
  **end for**
  $T_{previous} \leftarrow T$
**end function**

allows us to convert from virtual to real time units. Finally, we obtain the finish time $T_{finish}$ by adding the real time spent $T_{spent}$ to the previous update time $T_{previous}$. Previous update time represents the period when virtual time was last updated, or the previous current time $T_{current}$ that was used to update virtual time.

**progressVirtualTime**. To progress virtual time, both global and user virtual times must be updated to the current time $T$. We first progress the global virtual time. This can be done by calculating the amount of real time that has passed $T_{passed}$ since the previous update $T_{previous}$, and then using it to calculate the virtual time that has passed by multiplying it with the user share amount $R_{user}$. Virtual time in this context represents the marginal rate of progress each user experiences. Then, to update the user virtual time, we iterate over all active users and progress them till the current time $T$. We cover the details of this in the following subsection. Lastly, we update the previous update time $T_{previous}$ to reflect the time $T$ it was updated to.

**Algorithm 3** Virtual time updating for users

**Globals:** Previous update time $T_{previous}$

**function** updateUserVirtualTime($U_k, R_{user}, T_{current}$)
  $S^k_{jobs} \leftarrow$ getUserJobs($U_k$)
  $T^{user}_{previous} \leftarrow T_{previous}$
  $V^k_{user} \leftarrow$ getUserVirtualTime($U_k$)
  **for** each $(J_i, L_i, D^i_{user})$ in $S^k_{jobs}$ sorted by $D_{user}$ **do**
    $R_{job} \leftarrow \frac{R_{user}}{|S^k_{jobs}|}$
    $T_{passed} \leftarrow T_{current} - T^{user}_{previous}$
    $V^i_{user} \leftarrow V^k_{user} + (T_{passed} * R_{job})$
    **if** $D^i_{user} > V^i_{user}$ **then**
      **break**
    **end if**
    $V_{spent} \leftarrow D^i_{user} - V^k_{user}$
    $T_{spent} \leftarrow \frac{V_{spent}}{R_{job}}$
    $V^k_{user} \leftarrow V^k_{user} + V_{spent}$
    $T^{user}_{previous} \leftarrow T^{user}_{previous} + T_{spent}$
    $V^k_{arrival} \leftarrow$ getUserVirtualArrivalTime($U_k$)
    $V^k_{arrival} \leftarrow V^k_{arrival} + L_i$
    $S^k_{jobs} \leftarrow S^k_{jobs} \setminus (J_i, L_i, D^i_{user})$
  **end for**
  **if** $|S^k_{jobs}| > 0$ **then**
    $R_{job} \leftarrow \frac{R_{user}}{|S^k_{jobs}|}$
    $T_{spent} \leftarrow T_{current} - T^{user}_{previous}$
    $V^k_{user} \leftarrow V^k_{user} + (T_{spent} * R_{job})$
  **end if**
**end function**

*3.3.3 updateUserVirtualTime.* Updating user virtual time requires iterating over all currently active user jobs in the order of their virtual deadlines $D^j_{user}$. This is because of the same principle as for global virtual time, to ensure that jobs progress at the correct marginal rate. We first need to set the user's previous update time, $T^{user}_{previous}$, to the previous update time, $T_{previous}$. This time will be used to track the periods between virtual time updates, ensuring the correct progression of virtual time forward. Then we iterate over all user jobs in the order of their user virtual deadlines $D_{user}$. We first calculate the current job share, $R_{job}$, and the time that has passed since the previous update, $T_{passed}$. These arguments can then be used to calculate the assumed user virtual time $V^i_{user}$ that does not account for jobs leaving the system, hence does not equate to the actual user virtual time $V^k_{user}$. However, we can use this virtual time to test whether the job with the earliest deadline, $D^i_{user}$, would have finished. If the assumed user virtual time $V^i_{user}$ is after the earliest deadline $D^i_{user}$, we can determine that no job will finish before the current time, and break the loop. However, if the job has finished, we then calculate the virtual time $V_{spent}$ spent on this job by taking the difference between its deadline $D^i_{user}$ and the current user's virtual time $V^k_{user}$. This virtual time can then be converted into real time, $T_{spent}$, by dividing it by the job shares, $R_{job}$.

We update the user virtual time $V^k_{user}$ by adding the virtual time that has been spent $V_{spent}$, and progress the previous update time $T^{user}_{previous}$ by the real time spent $T_{spent}$. Besides the user's virtual time, we also must update the virtual arrival time $V^k_{arrival}$. This is done by progressing the virtual arrival time $V^k_{arrival}$ by the runtime of the job $L_i$. The virtual arrival time must be progressed, so future jobs that are assigned the global deadlines account for jobs that have finished in the past, and keep global order consistent. Finally, once all finishing jobs are accounted for, in the case there are still any jobs left in $S^k_{jobs}$, we must progress the user virtual time $V^k_{user}$ forward until it is caught up to the current time $T_{current}$. We can do this by calculating the remaining time $T_{spent}$ and multiplying it by the job shares $R_{job}$ to obtain the virtual time that has passed, and then adding it to the user's virtual time $V^k_{user}$.

# 4 Implementation Details

This section discusses how UWFQ is integrated into Spark and how it accounts for the dynamic runtime environment.

## 4.1 Apache Spark Integration

The integration within Spark requires addressing three parts: scheduler, partitioner, and external system integration.

*4.1.1 Scheduling.* Spark already provides an extendable interface for adding new schedulers. However, this interface is only accessible within the confinement of the source code. To make the framework more flexible, we extend the Spark source code with class loading for custom schedulers. This allows the Spark driver to load any precompiled Java Spark scheduler that is passed as an argument. We have released our modified Spark 3.5.5 version[1]. In current Spark releases, the scheduler only operates at the level of stages, where the stage with the highest priority is determined, and its tasks are scheduled onto the executors whenever resources are available. UWFQ is implemented in this layer, where incoming stages have their priority assigned by UWFQ. However, instead of isolating the priority for stages, each stage is first mapped to its corresponding analytics job from which the priority is inherited. Based on when the job first arrived in the system and its total runtime across all stages, the virtual deadline is determined, which will be assigned to every stage belonging to this analytics job. This ensures that even if a job is spread across multiple stages, it will still be executed within the guaranteed user-job fair time bounds and avoid unnecessary interleaving. In the Spark ecosystem, the highest priority is assigned to the stage with the lowest priority value $P_s$. This aligns perfectly with global virtual deadlines, as the job with the lowest deadline value $D$ should be scheduled the earliest. We express the priority assignment as $P_s = D^i_{global}$.

*4.1.2 Partitioning.* Spark performs partitioning in two phases of job execution: during the initial input reading and when coalescing between shuffle stages. When a Spark job is submitted, it must perform a file scan to load the necessary data into the first stage. During the file scan, partition sizes are calculated only using the input data size, which is then used to divide the input files between the tasks. To introduce runtime partitioning, we override this function with custom partitioning, which then attempts to partition the stage input using estimated runtime. After finishing the leaf stages of the DAG, there may be multiple shuffle stages that funnel the output of previous stages as input for the upcoming stage. By default, all shuffle stage outputs are written into 200 partitions. However, the Adaptive Query Execution (AQE) module coalesces these partitions into more appropriately sized partitions using the intermediate output sizes, without considering the upcoming stage runtime. Because the minimum partition amount is set to 1 by default in this phase, it can create long-running tasks if the following stage runtimes are not considered. To improve AQE coalescing, we replace the default minimum partition amount with our dynamically calculated number from the estimated stage runtime. This ensures that the partitions never coalesce down to an amount that would introduce long-running tasks, and also minimally interferes with AQE's own methods of reducing runtime skews. To add more flexibility to the framework, in both cases, the partitioning algorithm is class-loaded into the framework on startup, similarly to the scheduler.

*4.1.3 External system.* The external layer that uses Spark's engine for batch processing only has to interact with Spark Context to ensure compatibility with UWFQ. This is because all metadata that is embedded into the Spark Context is attached to the submitted job that will propagate the metadata to subsequent layers, allowing UWFQ to make proper scheduling decisions. UWFQ needs two information when scheduling a user job: user context and job context. User context allows UWFQ to map each of the stages to their corresponding user, to enable user-job fairness in the schedule. Job context is used to group together stages that were submitted under the same analytics job, allowing stages to be scheduled with respect to the job they belong to instead of being independent. Since users are only concerned with the final outcome of their analytics job, the intermediate results of individual stages do not matter. UWFQ needs runtime predictions to perform effective scheduling, which can be provided by the external system or a performance estimation module within Spark's engine. We achieve this by adding a class-loaded performance estimator to the framework, which can then be accessed across all Spark components to inquire performance metrics of a stage or other units. The benefit of this component is decoupling the performance prediction from the external system, and providing a much more specialized interface for other components to acquire runtime estimates of work units.

## 4.2 Accounting for Dynamic Environments

UWFQ stops considering users as soon as all of their jobs have finished. This is necessary to accurately distribute resources among users who still have pending jobs running. However, due to delays caused by inaccuracies in runtime estimation, there are cases where users would exit the system before all the stages associated with their job have finished. This creates a scenario where a discrepancy exists between the real system and virtual time, where the real job is still executing even though it has finished in the virtual scheduler. If the user's job still has stages that have not been scheduled, these stages will not be attributed to the correct start

[1] https://github.com/kazemaksOG/spark-3.5.5-custom

time. To correct this, we introduce a grace period, in which the following stages can "revive" a user who has exited the system with their original arrival time.

While this provides the solution for these scenarios, it presents some caveats. By setting the grace period too small, it will not be triggered in the highlighted scenarios, rendering it useless. However, setting it too high could allow inaccurately estimated jobs to gain high priority, since they will be assumed to be delayed by the system rather than their real runtime. For our environment, we dynamically adjust the grace period to last 2 resource seconds, as the resource may slightly fluctuate over time. The user will be revived if the inequality $V_{global} < V^k_{global,\,end} + T_{grace} * R$ is satisfied, where $V^k_{global,\,end}$ is the global virtual end time of user $k$, $R$ is the total system resource amount, and $T_{grace}$ is the grace period in resource seconds. The amount of 2 resource seconds has been shown to work for our environment, however, it can differ per system and may need to be more dynamic for appropriate behavior.

## 5 Evaluation

We evaluate UWFQ on micro and macro-benchmarks and compare it with the default Spark built-in fair scheduler and representative schedulers. Additionally, we quantify the impact of runtime partitioning on response time and fairness.

### 5.1 Setup

Experiments are conducted on a the DAS-5 cluster [19]. We reserve 5 computing nodes equipped with dual 8-core processors and 60 GB of RAM. Communication between nodes is facilitated by Gigabit Ethernet (GbE) and InfiniBand (IB). We use 4 of the 5 reserved computing nodes to spawn executors, where each node would have at most 2 executors, each reserving 4 cores and 4 GB of RAM, with a total of 32 cores and 32 GB across the cluster. The last node was reserved to run the driver program, with 8 cores and 60 GB of memory. We run our experiments on our modified Spark 3.5.5 version built with Scala 2.12. The modifications applied to Spark enable custom scheduler and partitioner support and do not interfere with the performance of built-in schedulers. The driver is compiled with Java 17 and runs with the Spark standalone cluster manager. We leave most Spark configuration parameters at default values with a few exceptions. We enable event logging to collect execution traces after the application has finished, and set high job retention thresholds to avoid omitting earlier jobs. For our partitioned dataset, Spark's default settings cause the dataset to be partitioned too extensively, creating task scheduling overhead. We solve this by increasing the *maxPartitionBytes* to an empirically measured amount to avoid this. We have released the experiment setup source code[2]. Designing an accurate runtime predictor is orthogonal to our work. We therefore assume a perfect runtime prediction for our experiments and discuss this assumption in Sec. 6.4.

*5.1.1 Metrics.* With our experiments, we aim to demonstrate that the UWFQ scheduler can improve response time while still ensuring bounded user-job fairness. To measure this, we collect job response times, slowdowns, and deadline violations and slack ratios.

- The **response time** (RT) of an analytics job is the time that elapses since its first stage is submitted until the last stage is completed, i.e., $RT_i = \text{MAX}(T^i_{s,end}) - \text{MIN}(T^i_{s,start})$.
- **Slowdown** (SL) is calculated by dividing the response time of the job running in the schedule with the response time when the same job is run in the idle system, i.e., as $SL_i = \frac{RT^i_{shared}}{RT^i_{idle}}$. Slowdowns show relative gains and losses in performance instead of absolute units.
- The **deadline violations ratio** (DVR) is computed by first calculating the ratio $r_i$ for each job by taking the end time difference between the target scheduler and UJF and normalizing it by the UJF runtime of the job, as seen in Equation 1. Then to calculate the DVR value, the average of the of incurred proportional violations is taken, as seen in Equation 2. Since we do not have a "true" UJF running in the system, we implement a practical UJF scheduler in Spark, and use its execution trace as a substitute. This metric essentially gives us an overview of how much the target scheduler slows down the jobs in comparison to a UJF scheduler.
- The **deadline slack ratio** (DSR) calculates the average of proportional slack time gained, as seen in Equation 3. This showcases the speed-up that the target scheduler brings in comparison to UJF.

$$r_i = \frac{\text{MAX}(T^i_{s,end,\text{target}}) - \text{MAX}(T^i_{s,end,\text{UJF}})}{RT^i_{\text{UJF}}} \tag{1}$$

$$DVR = \frac{\sum_i \max(0, r_i)}{\sum_i 1_{\{r_i > 1\}}} \tag{2}$$

$$DSR = \frac{\sum_i \max(0, -r_i)}{\sum_i 1_{\{r_i \leq 1\}}} \tag{3}$$

*5.1.2 Baseline Schedulers.* Our micro-benchmarks compare UWFQ to representative schedulers. We show the general priority formulas for each of these schedulers. Note that in Spark, a stage with the lowest priority value $P_s$ has the highest scheduling priority.
- **Fair scheduling.** Spark comes in with a built-in Fair scheduler that assigns the highest priority to stage $s$ with the least amount of running tasks $N^s$, using $P_s = N^s_{active\ task\ amount}$. While this is the scheduler we aim to replace, it does not implement UJF, hence it does not necessarily give a good indication of fairness.
- **UJF scheduler.** We implement a practical UJF scheduler in Spark, which we use as the baseline for fairness. This is done by dynamically creating pools for each user as they arrive, assigning the highest priority to the user $k$ with the least amount of tasks $N^k$, with $P_k = N^k_{active\ task\ amount}$. Internally, pools use Fair scheduling to distribute resources among their active stages. Note that this does not implement perfect fairness, since we are working with real hardware, where perfect parallelism is not achievable.
- **Cluster Fair Queuing.** We implement Cluster Fair Queuing (CFQ) [8] for our benchmark by updating the public source code published on GitHub [3]. While CFQ implements GPS bounded fairness, it is still valuable to compare UWFQ, to see if our proposals bring practical benefit to the system. CFQ priority is decided similarly to UWFQ, but only considers fairness across stages by assigning a deadline $D$ based on traditional virtual time [18], using $P_s = D_s$, omitting the wider context of users and jobs.

[2]Experiment source code: https://github.com/kazemaksOG/spark-benchmark-tool

[3]CFQ implementation: https://github.com/chenc10/spark-CFQ-INFOCOM17

**Table 1: Comparison of scheduler performance metrics for scenarios 1 and 2.**

| | Scheduler | Response time (s) ↓ \| Slowdown ↓ | | | | Fairness | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Avg. | Worst 10% | Freq. \| Infreq. | First \| Last | DVR ↓ | Violation # ↓ | DSR ↑ | Slack # ↑ |
| Scen. 1 | Fair | 44.0 \| 19.9 | 86.1 \| 39.0 | 44.5 \| 40.8 | - | 3.25 | 141 | 0.27 | 227 |
| | UJF | 43.1 \| 19.5 | 74.7 \| 33.9 | 48.8 \| 5.13 | - | - | - | - | - |
| | CFQ | 32.5 \| 14.7 | **49.8** \| **22.5** | **32.3** \| 34.2 | - | 3.69 | 91 | **0.38** | 277 |
| | UWFQ (this work) | **29.4** \| **13.3** | 50.3 \| 22.8 | 33.1 \| **4.50** | - | **0.23** | **58** | 0.37 | **310** |
| Scen. 2 | Fair | 28.1 \| 32.9 | 41.6 \| 48.9 | - | 21.4 \| 33.7 | 0.78 | 95 | 0.31 | 145 |
| | UJF | 29.1 \| 34.1 | **41.3** \| **48.5** | - | 25.4 \| 33.3 | - | - | - | - |
| | CFQ | 43.2 \| 50.7 | 49.5 \| 58.1 | - | 36.7 \| 47.3 | 0.80 | 211 | **0.51** | 29 |
| | UWFQ (this work) | **25.5** \| **29.9** | 46.5 \| 54.6 | - | **15.8** \| **31.3** | **0.50** | **94** | 0.43 | **146** |

• **Using runtime partitioning.** All schedulers are initially evaluated using the default partitioning of Spark. However, to highlight the impact of using runtime partitioning, each scheduler is also evaluated while employing our partitioning algorithm. To distinguish them from their original version, we add a -P to the end of the schedulers utilizing runtime partitioning, e.g., UWFQ-P. Note that when calculating DVR and DSR values, we compare finish times to UJF with the same partitioning implementation.

## 5.2 Micro-benchmarks

Our microbenchmarks use For-Hire Vehicle High Volume (FHVHV) Trip Records from the NYC Taxi and Limousine Commission (TLC)[4] datasets of August 2024. The data is stored in a Parquet file format, which we further partition on *PULocationID* to create more row groups, so the file can be split into more partitions by Spark. The total size of the partitioned parquet file is 752 MB, with 19.1M rows.

We simulate analytics jobs by applying a varying number of operations per row of the dataset. A single analytics job consists of 3 phases: loading the dataset, applying the computation, and retrieving results. Each of these phases has its own stages, creating a linear stage dependency tree. We load the same TLC dataset for all jobs, however, we load them separately, so each job has its own dataset loaded without reusing others. The main time spent in each job is applying computations, which can range from sub-second to 10 seconds in wall-clock time. Finally, the results are collected, which takes only a couple of milliseconds.

We define tiny and short jobs, where each type of job always performs the same operations. We use their runtimes in the idle system for slowdown calculations. In such settings, short, and tiny jobs respectively require 2.25, and 0.90 s to run. While this may not accurately simulate dynamic runtimes seen in real settings, it is sufficient to highlight the interaction between jobs based on their arrival times and system congestion. In one of the scenarios, we mimic infrequent user behavior by using a Poisson distribution. A Poisson distribution is commonly used in scheduling to simulate user action frequency [12, 14, 20], and it gives fine control over user workload without hard-coding arrival times.

*5.2.1 Scenarios.* In collaboration with industry specialists, we consider two scenarios that have been observed in production.

• **Scenario 1: infrequent and frequent users**. We first examine how schedulers handle the case where some users have many concurrent jobs while some other users only infrequently come to the system to run their workload. This is the main scenario that UWFQ is trying to improve by introducing user context in scheduling, where more emphasis is put on resource distribution among users rather than the runtime of jobs. The main issues with this scenario arise when infrequent users are left without any resources, or all jobs are computed simultaneously, causing a significant increase in response times. We construct this scenario by introducing 2 infrequent users and 2 frequent users in the system. Infrequent users follow a Poisson distribution when scheduling their jobs. A frequent user schedules a burst of short jobs every 30 seconds that fully congests the system, introducing delay for all jobs.

• **Scenario 2: multiple frequent users**. We study how the system handles a burst of jobs from multiple users and how it recovers while maintaining fairness. Key challenges include unequal user prioritization and excessive parallel job execution, which can raise response times. We simulate 4 users submitting many small jobs simultaneously, each with a fixed start delay to preserve consistent arrival order across runs, though job completion order may vary.

*5.2.2 Results.* Table 1 summarizes the micro-benchmark results. UWFQ achieves the best average response time in both scenarios, lowering the mean response time by up to 32% compared to UJF. We attribute these improvements to both job-context and user-context that are present in UWFQ. When utilizing job-context, jobs are favored to run till completion, instead of interleaving between all active jobs. This is best visualized in scenario 2 where all jobs are completed gradually, rather than in batches, as seen in Figure 6. We can also see that for this scenario, CFQ performs the worst out of all schedulers by a significant amount. We attribute this to the fact that CFQ does not utilize job context, hence executes each job one stage at a time, and finishes them all only at the very end.

The most substantial improvement can be seen by introducing user context, where infrequent users have a much better response time in UWFQ and UJF, lowering the average response time in UWFQ by 89% compared to Fair, which is also highlighted in Figure 5. We believe that in scenarios where users differ in the amount of scheduled jobs, user context allows for fair resource distribution across users, and algorithms that perform fairness only with respect to jobs will disproportionally allocate resources to users with more jobs. This is one of the main drawbacks observed in CFQ, which in this scenario increases the response times of infrequent user jobs by more than 7 times compared to UWFQ.

In scenario 1, UWFQ achieves the lowest number of deadline violations and the smallest DVR value. However, in scenario 2,

[4]TLC trip dataset: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

**Table 2: Comparison of scheduler performance metrics with the macro-benchmark.**

| Scheduler | Runtime ↓ | Response time (s) ↓ | | | | Fairness | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Avg.% | 0–80% | 80–95% | 95–100% | DVR ↓ | Violation # ↓ | DSR ↑ | Slack # ↓ |
| Fair | 563.1 | 46.12 | 28.29 | 100.5 | **164.4** | 1.00 | 83 | 0.42 | 91 |
| UJF | 565.7 | 54.12 | 27.43 | 140.9 | 215.5 | - | - | - | - |
| CFQ | 621.0 | 37.05 | 11.05 | 85.67 | 298.1 | 0.55 | 30 | 0.52 | 144 |
| UWFQ (this work) | 624.2 | 41.42 | 12.33 | 92.36 | 343.5 | **0.44** | 38 | 0.51 | 136 |
| Fair-P | 563.8 | 49.66 | 27.94 | 112.7 | 202.7 | 1.30 | 102 | 0.28 | 72 |
| UJF-P | **562.8** | 50.44 | 23.02 | 140.2 | 214.4 | - | - | - | - |
| CFQ-P | 592.5 | **28.64** | **5.51** | **63.78** | 284.2 | 0.69 | 28 | **0.61** | 146 |
| UWFQ-P (this work) | 591.8 | 31.19 | 6.05 | 67.44 | 314.6 | 0.61 | 27 | 0.56 | 147 |

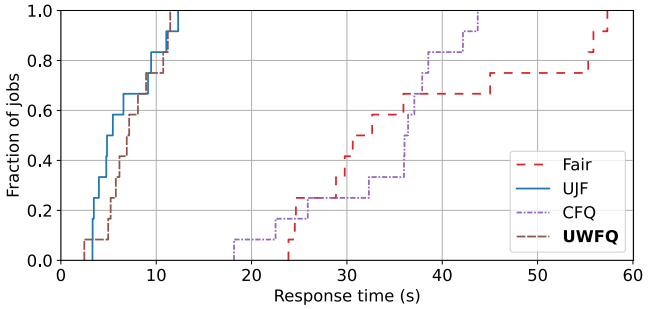



**Figure 5: Empirical CDFs for infrequent users in scenario 1.**

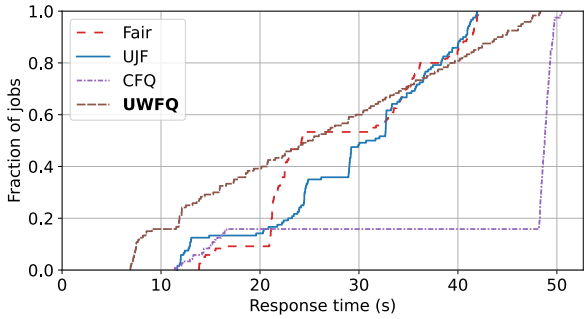



**Figure 6: Empirical CDFs in scenario 2.**

the fairness metrics are not as representative. This is due to direct completion time comparisons between jobs in the target and UJF schedules, and because the arrival times of all jobs are within a very small interval of time, the order of job completion differs between all schedulers arbitrarily, since there is no global priority between these jobs across schedulers. However, we can analyze the response times between different users to see how resources are distributed among them.

UWFQ achieves the best performance for its first arriving user and its last arriving user. We do note that the first arriving user has a much lower average response time than the last arriving user with UWFQ, however, the same pattern is observed in UJF, hence it could happen due to slightly quicker arrival time of the first user, rather than scheduling unfairness.

## 5.3 Macro-benchmarks

To the best of our knowledge, there is no accepted workload to benchmark batch processing schedulers in multi-user and multi-job environment. Popular benchmark suites such as TPC-H[5] or HiBench[6] do not contain the user context information that is necessary to represent such an environment. We therefore convert existing system traces that contain multi-user and multi-job metadata into compatible Spark job execution traces, as done previously in some works [4, 8, 21]. To compose our macro-benchmark, we use Google traces (2014) [16] originally taken from Google cluster usage traces [22] but standardized into Workflow Trace Archive (WTA) format [23]. We select a period of 500 seconds within the trace, and scale the tasks within it to reach a certain utilization threshold. We slightly refine the traces to create workloads that we target to solve, while keeping realistic user arrival times and behavior. We select the jobs that occur between 1'473'800'000 ms and 1'474'300'000 ms. We filter out any jobs whose runtime is 10x longer than the initial median and scale the rest to achieve around 100% or above theoretical resource utilization. Resource utilization is high to ensure that there is always competition between resources, allowing us to analyze how the system can fairly recover from a burst of jobs. The final dataset contains 25 users, where the majority only schedule infrequent small jobs, and 5 large users whose jobs represent more than 90% of the total workload.

*5.3.1 Results.* Table 2 reports job response times. We group the jobs into the first 80th percentile, the next 15th percentile (medium-sized jobs), and the final 5th percentile. UJF and Fair execute within the same margin of time, while there is a significant slowdown for both CFQ and UWFQ, both increasing the benchmark time by 10% in comparison to UJF. We attribute this slowdown to long-running tasks, which may increase the runtime due to ineffective parallelization. However, the slowdown goes down to 5% when introducing runtime partitioning, which then is most likely caused by JVM warmup [21], where Fair and UJF algorithms tend to better utilize warm executors.

CFQ and UWFQ achieve the best overall response time, which is attributed to the significant speedups seen for small and medium length jobs. They respectively improve the average response time by 32% and 24% compared to UJF. For jobs in the 80th percentile, the response time is decreased by 60% and 55% compared to UJF,

[5]TPC-H: https://www.tpc.org/tpch/
[6]HiBench: https://github.com/Intel-bigdata/HiBench

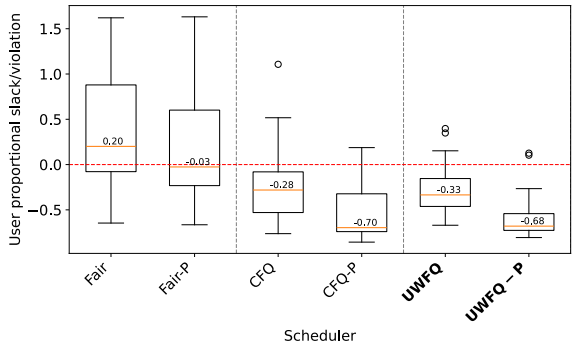



**Figure 7: Proportional deadline violations of users, where slacks are negative values and violations are positive.**

and for medium jobs it is lowered by 40% and 34% respectively. However, there is a major gain for longer job response times in CFQ and UWFQ, with 38% and 60% increase compared to UJF.

UWFQ has a slightly lower DVR value than CFQ, but CFQ has slightly higher DSR value. However, for this benchmark we further analyze the average response times of users to determine fairness differences. We visualize this in Figure 7 by calculating the DVR and DSR values between user average response times rather than job runtimes. We observe that UWFQ is able to more tightly contain the response time improvements with respect to users compared to CFQ. However, the improvement is much less significant than what we have seen in micro-benchmarks.

Runtime partitioning for a homogeneous workload can significantly improve performance metrics. CFQ-P and UWFQ-P can further reduce average response times by 43% and 38% compared to UJF-P, respectively. This massive improvement is caused by lowering the average response times of the 80th percentile by 76% and 74% compared to UJF-P, and lowering the next 15th percentile by 55% and 52%. While CFQ and UWFQ still increase the runtime of long jobs, partitioning slightly decreases them to 32% and 46% compared to UJF-P. Lastly, partitioning additionally makes the algorithms slightly fairer, by lowering the amount of violations and DVR values compared to regular partitioning for both CFQ and UWFQ, and improving fairness across users, as seen in Figure 7.

# 6 Related Work

## 6.1 Scheduling Algorithms

**Generalized processor sharing (GPS)** is an idealized scheduling model that achieves perfect job-level fairness [24]. GPS allocates resources $R_i$ proportionally to a job's weight $w_i$ following $R_i = w_i \times R$. In practice, GPS cannot be implemented exactly since it requires jobs to be divisible into infinitesimally small quanta, and all resources to run scheduled job quanta simultaneously. However, it is a theoretical baseline that can be used to evaluate other scheduling algorithms. The major difference between regular GPS and our user-job fairness definition is that job weights are tied to two separate entities. In GPS, once a job enters or leaves the system, the job shares are taken from or distributed among all active jobs on the system. However, under user-job fairness, only the jobs associated with the same user will have to share their resources.

**Fair-share scheduling** equally distributes resources among users [25]. Jobs are associated with a user $k$ and are serviced proportionally to the user resource share $R_k$ and the job scheduling policy that assigns a weight $w_i$ for the job. In a system with $N_u$ users, the resource $R_i$ each job receives is computed as $R_k = \frac{R}{N_u}$ and $R_i = w_i \times R_k$. This ensures that each user receives their fair share of resources, guaranteeing progression for their work. While providing a fairness definition for user-level, it leaves job-level scheduling to be defined in the implementation.

**Max-min fairness** ensures that each user gains more resources if and only if it does not result in a decrease of resources for another user with a lower or equal share [26]. In other words, it maximizes the minimum amount of resources $R_k$ for each user $k$, ensuring fairness, while still distributing resources proportionally by demand. This property ensures that each user is guaranteed their fair share of resources, while allowing more demanding users to gain a larger share if permitted. While max-min fairness improves the throughput of the scheduler, it does not account for job priority.

**Proportional-fair scheduling** can be used to have finer control between fairness and throughput/priority. Proportional-fair scheduling aims to maximize a performance metric of the system while still ensuring at least minimal level of service to all users. A common implementation assigns a scheduling interval to the user with the highest priority metric [27]. For the multi-user and multi-job environment, it is defined by $P_k = \frac{P_i}{H_i^\alpha}$, where $P_k$ is priority of user $k$, $P_i$ is the highest priority job $i$ of the user $k$, and $H_i$ is the historical resource usage time of this user. $\alpha$ is used to tweak the fairness of the algorithm. If set to 0, it will always service the user with the highest job priority. By setting it proportionally high, it gives service to the least serviced user requesting resources, achieving max-min fairness. By setting a reasonable value for $\alpha$, a good compromise between job priority and fairness can be achieved.

**Weighted fair queuing (WFQ)** [18] executes jobs in the order of their GPS expected finish times and uses job weights to prioritize some jobs. WFQ first computes the virtual job completion time $V_f$ for every active job in the system as $V_f = V_a^i + \frac{L_i}{w_i}$, where $w_i$ is the weight of job $i$, $V_a^i$ is the virtual arrival time of job $i$ and $L_i$ is the total execution time of job $i$. Afterward, WFQ schedules the job with the earliest virtual finish time $V_f$. WFQ approximates GPS, ensuring that every job is completed no later than it would be under GPS, with maximum delay bounded by the size of the largest unsplittable job in the system. This is also expressed by $f_i - \hat{f}_i \leq \frac{L_{max}}{R}$, where $f_i$ is the completion time of the job $i$ under WFQ, $\hat{f}_i$ is the completion time of the job $i$ under GPS, $L_{max}$ is the maximum job runtime, and $R$ is the amount of resources that can execute a job in parallel (typically the amount of cores). WFQ closely approximates the perfect fairness of GPS and regulates the priority of jobs by adjusting their weights. However, its implementation depends on the concept of virtual time.

**Virtual time** [8, 18] simulates the marginal service rate that each job receives under GPS. It decreases the time complexity associated with maintaining the finish times of jobs to ensure WFQ. Under a GPS scheduling, whenever a new job arrives, each existing

job has its finish time extended by the proportion of the resources they have to forfeit to the arriving job. The same happens whenever a job leaves the system, spreading its share across all active jobs. Both events imply an $O(N)$ complexity to recalculate the finish times of $N$ active jobs. However, using virtual time, it is possible to warp the time itself based on the amount of resources that are being shared across jobs. Since all jobs equally forfeit or gain resources as jobs enter or leave the system, we can instead advance or slow down virtual time based on the number of active jobs in the system. This is expressed in the time domain by $V(t) = \int_0^t \frac{R}{N_j^t}\, dt$, where $N_j^t$ are the currently active jobs in the GPS schedule at time $t$ and $V(0) = 0$. When a job arrives in the system, its virtual deadline is computed as the time at which it would finish under GPS. While a virtual deadline does not directly map to real time, sorting by virtual deadlines would yield the same order of finish times under GPS scheduling. By manipulating virtual time instead of real time, the time complexity incurred by new or ending jobs is reduced from $O(N)$ to $O(\log_2 N)$, since virtual deadlines are set only once, and the incoming jobs have to be put in an ordered list and ending jobs have to be removed from the list, which takes at most $O(\log_2 N)$ time.

## 6.2 Scheduling in Batch Processing Systems

Pastorelli et al. [14] described a scheduler that is analogous to Weighted Fair Queuing for Hadoop [18], and that was later adapted to Spark [8]. These implementations address fairness across jobs rather than among users, as we do in this work. L. Chen. et al. [7] implement a max-min fair scheduling algorithm for Spark jobs across geo-distributed datacenters. A fair scheduling solution for application-level scheduling that enables job preemption was developed by W. Chen et al. [4]. Wang et al. [5] improve job completion times by considering deadlines of Spark applications. However, application level scheduling algorithms are not directly applicable to our target industrial system, which is mostly built around a single long-running Spark application. Some implementations attempt to increase the performance of schedulers by optimizing the level of parallelism [21, 28]. In another work, C. Chen et al. [15] use speculative execution and resource reservation to improve scheduling performance. These solutions do not enforce a strict fairness policy. However, we believe that these solutions could be integrated with our proposed scheduler to further improve its performance. While some works mention the performance slowdown caused by long-running tasks [4, 8, 14], there currently is no effective solution proposed that is able to reduce the impact of long-running tasks while ensuring minimal performance degradation. We solve this issue by introducing runtime partitioning.

## 6.3 Existing Fairness Definitions

Classic scheduling algorithms ensure fairness by distributing resources equally among their users [25, 26, 29, 30] or jobs [4, 7, 24]. These definitions compromise other performance metrics such as throughput or response times, hence it can be beneficial to find a compromise between performance and fairness, which is achieved by proportional fairness [27, 31]. With job deadlines, fairness requires jobs to be serviced before their deadlines [5, 12, 32, 33]. More recent works equalize some form of detriment caused by sharing resources, e.g., by equalizing the slowdowns each job or user may experience [9, 13]. Bochenina et al. [34] define unfairness as the maximum difference between any two workflow fines, while Rezaeian et al. [35] try to equalize the savings each workflow achieves. Ferreira da Silva et al. [11] define fairness as the difference between the workflow with the most tasks finished and the workflow with the fewest tasks finished. Fairness is also sometimes defined by some constraint that has to be satisfied. C. Chen et al. [8] propose that fairness is a constraint where the difference between the finish time of a task and the finish time of the same task under max-min scheduling never goes beyond a certain constant value, similarly to Pastorelli et al. [14]. In another paper, fairness is used to impose a maximum performance degradation a job can incur by sharing its resources [21]. In the context of backfilling, fairness also refers to the property that lower priority jobs do not delay higher priority jobs [36, 37]. We find C. Chen et al. [8]'s to be the most appropriate fairness definition in batch processing environments.

## 6.4 Job runtime prediction

Accurate job runtime prediction is crucial for efficient scheduling and partitioning. However, runtime prediction is orthogonal to our focus, and we assumed perfect runtime prediction for our experiments. Several existing works demonstrate efficient runtime prediction methods that could be integrated with UWFQ. First, by decomposing jobs into smaller units such as operations [38], stages [39], or tasks [40], and estimating their runtime based on these individual units. Second, by training a machine learning model using historic data, for example, using regression trees [41]. Finally, job runtime can be estimated through job simulations [42–45]. Even with the high accuracy of these methods, it is worth noting that virtual time-based scheduling, which our work builds upon, has been shown to be robust to inaccurate runtime predictions [8].

# 7 Conclusion

We presented the User Weighted Fair Queuing (UWFQ) scheduling algorithm, which aims to minimize the average response time of jobs while ensuring that users and their jobs receive a proportional resource share. We additionally introduced a runtime partitioning algorithm to reduce the impact of long-running tasks on fairness and job response times by partitioning the jobs into finer amounts. We implemented UWFQ with runtime partitioning in Spark to measure its performance and show its viability. Comparing UWFQ to a simple fair scheduler that evenly spreads resources across users, UWFQ with runtime partitioning reduces the average response times of small jobs by up to 74% in homogeneous workloads, while still ensuring that most jobs are completed within the fairness boundaries. We additionally showed that UWFQ is more robust for handling multi-user, multi-job critical micro-benchmark scenarios, being able to maintain a 6x lower fairness violation ratio compared to other scheduling algorithms.

# References

[1] A. Saxena, D. Claeys, H. Bruneel, and J. Walraevens. "Analysis of the age of data in data backup systems". In: *Computer Networks* 160 (2019), pp. 41–50.

[2] S. Tang, B. He, C. Yu, Y. Li, and K. Li. "A Survey on Spark Ecosystem: Big Data Processing Infrastructure, Machine Learning, and Applications (Extended abstract)". In: *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2023, pp. 3779–3780.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1383–1394.

[4] W. Chen, X. Zhou, and J. Rao. "Preemptive and Low Latency Datacenter Scheduling via Lightweight Containers". In: *IEEE Transactions on Parallel and Distributed Systems* 31.12 (2020), pp. 2749–2762.

[5] G. Wang, J. Xu, R. Liu, and S. Huang. "A Hard Real-time Scheduler for Spark on YARN". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2018, pp. 645–652.

[6] W. Chen, A. Pi, S. Wang, and X. Zhou. "Characterizing Scheduling Delay for Low-Latency Data Analytics Workloads". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 630–639.

[7] L. Chen, S. Liu, B. Li, and B. Li. "Scheduling jobs across geo-distributed datacenters with max-min fairness". In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9.

[8] C. Chen, W. Wang, S. Zhang, and B. Li. "Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees". In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9.

[9] F. Li, W. J. Tan, M. G. Seok, and W. Cai. "Clustering-based multi-objective optimization considering fairness for multi-workflow scheduling on clouds". In: *J. Parallel Distrib. Comput.* 194.C (Dec. 2024).

[10] B. T. Quang, J.-S. Kim, S. Rho, S. Kim, S. Kim, S. Hwang, E. Medernach, and V. Breton. "A Comparative Analysis of Scheduling Mechanisms for Virtual Screening Workflow in a Shared Resource Environment". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015, pp. 853–862.

[11] R. Ferreira da Silva, T. Glatard, and F. Desprez. "Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions". In: *Concurrency and Computation: Practice and Experience* 26.14 (2014), pp. 2347–2366.

[12] A. Ilyushkin and D. Epema. "The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2018, pp. 331–341.

[13] Z.-h. JIA, L. Pan, X. Liu, and X.-j. Li. "A novel cloud workflow scheduling algorithm based on stable matching game theory". In: *The Journal of Supercomputing* 77 (Oct. 2021), pp. 1–28.

[14] M. Pastorelli, D. Carra, M. Dell'Amico, and P. Michiardi. "HFSP: Bringing Size-Based Scheduling To Hadoop". In: *IEEE Transactions on Cloud Computing* 5.1 (2017), pp. 43–56.

[15] C. Chen, W. Wang, and B. Li. "Speculative Slot Reservation: Enforcing Service Isolation for Dependent Data-Parallel Computations". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 549–559.

[16] Google. *Workflow Trace Archive Google trace*. https://zenodo.org/record/3254540. Zenodo. June 2019.

[17] D. Kažemaks, L. Versluis, B. K. Ozkan, and J. Decouchant. *Balancing Fairness and Performance in Multi-User Spark Workloads with Dynamic Scheduling (extended version)*. 2025. arXiv: 2510.15485 [cs.DC].

[18] A. Parekh and R. Gallager. "A generalized processor sharing approach to flow control in integrated services networks: the single-node case". In: *IEEE/ACM Transactions on Networking* 1.3 (1993), pp. 344–357.

[19] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *Computer* 49.5 (2016), pp. 54–63.

[20] A. Ilyushkin, B. Ghit, and D. Epema. "Scheduling workloads of workflows with unknown task runtimes". In: *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. CCGRID '15. Shenzhen, China: IEEE Press, 2015, pp. 606–616.

[21] C. Chen, W. Wang, and B. Li. "Performance-Aware Fair Scheduling: Exploiting Demand Elasticity of Data Analytics Jobs". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 2018, pp. 504–512.

[22] J. Wilkes. *Google cluster-usage traces v3*. Technical Report. Posted at https://github.com/google/cluster-data/blob/master/ClusterData2019.md. Mountain View, CA, USA: Google Inc., Apr. 2020.

[23] L. Versluis, R. Mathá, S. Talluri, T. Hegeman, R. Prodan, E. Deelman, and A. Iosup. "The Workflow Trace Archive: Open-Access Data From Public and Private Computing Infrastructures". In: *IEEE Trans. Parallel Distributed Syst.* 31.9 (2020), pp. 2170–2184.

[24] T. Li, D. Baumberger, and S. Hahn. "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin". In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '09. Raleigh, NC, USA: Association for Computing Machinery, 2009, pp. 65–74.

[25] J. Kay and P. Lauder. "A fair share scheduler". In: *Commun. ACM* 31.1 (Jan. 1988), pp. 44–55.

[26] L. Tassiulas and S. Sarkar. "Maxmin fair scheduling in wireless networks". In: *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 2. 2002, 763–772 vol.2.

[27] S. Ahmadi. "Chapter 6 - The IEEE 802.16m Medium Access Control Common Part Sub-layer (Part I)". In: *Mobile WiMAX*. Oxford: Academic Press, 2011, pp. 169–279.

[28] D. Cheng, X. Zhou, Y. Wang, and C. Jiang. "Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming". In: *IEEE Transactions on Parallel and Distributed Systems* 29.12 (2018), pp. 2672–2685.

[29] J. Nagle. "On Packet Switches with Infinite Storage". In: *IEEE Transactions on Communications* 35.4 (1987), pp. 435–438.

[30] L. Zhao. "A Two-Level Multi-task Fair Allocation Mechanism in Cloud Computing". In: *2023 8th International Conference on Computer and Communication Systems (ICCCS)*. 2023, pp. 1158–1162.

[31] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. "Multi-resource packing for cluster schedulers". In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 455–466.

[32] H. Chen, X. Zhu, G. Liu, and W. Pedrycz. "Uncertainty-Aware Online Scheduling for Real-Time Workflows in Cloud Service Environment". In: *IEEE Transactions on Services Computing* 14.4 (2021), pp. 1167–1178.

[33] E. Cadorel, H. Coullon, and J.-M. Menaud. "Online Multi-User Workflow Scheduling Algorithm for Fairness and Energy Optimization". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020, pp. 569–578.

[34] K. Bochenina, N. Butakov, and A. Boukhanovsky. "Static scheduling of multiple workflows with soft deadlines in non-dedicated heterogeneous environments". In: *Future Gener. Comput. Syst.* 55.C (Feb. 2016), pp. 51–61.

[35] A. Rezaeian, M. Naghibzadeh, and D. Epema. "Fair multiple-workflow scheduling with different quality-of-service goals". In: *The Journal of Supercomputing* 75 (Feb. 2019).

[36] Y. Yuan, Y. Wu, W. Zheng, and K. Li. "Guarantee Strict Fairness and UtilizePrediction Better in Parallel Job Scheduling". In: *IEEE Transactions on Parallel and Distributed Systems* 25.4 (2014), pp. 971–981.

[37] C. Gómez-Martín, M. A. Vega-Rodríguez, and J.-L. González-Sánchez. "Fattened backfilling". In: *J. Parallel Distrib. Comput.* 97.C (Nov. 2016), pp. 69–77.

[38] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. "Towards predicting query execution time for concurrent and dynamic database workloads". In: *Proc. VLDB Endow.* 6.10 (Aug. 2013), pp. 925–936.

[39] H. Al-Sayeh and K.-U. Sattler. "Gray Box Modeling Methodology for Runtime Prediction of Apache Spark Jobs". In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 2019, pp. 117–124.

[40] A. Gulino, A. Canakoglu, S. Ceri, and D. Ardagna. "Performance Prediction for Data-driven Workflows on Apache Spark". In: *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2020, pp. 1–8.

[41] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. "Robust estimation of resource consumption for SQL queries using statistical techniques". In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1555–1566.

[42] K. Wang and M. M. H. Khan. "Performance Prediction for Apache Spark Platform". In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 2015, pp. 166–173.

[43] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale. "Modeling Interference for Apache Spark Jobs". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, pp. 423–431.

[44] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. "PREDIcT: towards predicting the runtime of large scale iterative analytics". In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 1678–1689.

[45] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. "Ernest: efficient performance prediction for large-scale advanced analytics". In: *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. NSDI'16. Santa Clara, CA: USENIX Association, 2016, pp. 363–378.