Improving Response Time and Fairness with Efficient Scheduling and Partitioning

in Multi-User Apache Spark Environments

Dāvis Kažemaks

TU Delft

ASML

Improving Response Time and Fairness with Efficient Scheduling and Partitioning

in Multi-User Apache Spark Environments

by

Dāvis Kažemaks

to obtain the degree of Master of Science
at Delft University of Technology,
to be defended publicly on Tuesday July 8, 2025 at 13:30

Student number:	5300606	
Master Program:	Computer and Embedded Systems Engineering	
Project Duration:	November 2024 – July 2025	
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science	
Thesis committee:	Dr. Jérémie Decouchant	TU Delft, supervisor
	Dr. Burcu Kulahcioglu Ozkan	TU Delft, supervisor
	Dr. Christoph Lofi	TU Delft, member
	Laurens Versluis	ASML, supervisor

Cover:	ASML Cleanroom Assembly April 2019 by Bart van Overbeeke (©ASML)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Acknowledgements

I would like to express my deepest gratitude to all my supervisors Jérémie Decouchant, Burcu Kulahcioglu Ozkan, and Laurens Versluis for guiding me through the project. I want to thank Jérémie for trusting me with the opportunity to participate in this project, helping me develop my ideas and critically assessing my progress. This helped me grow as a person and acquire new knowledge in a field I was previously unfamiliar with. I would like to thank Burcu for providing another set of critical eyes and giving feedback from a different perspective which helped me avoid common mistakes and push me to reevaluate my ideas. And lastly, I would like to thank Laurens for helping me navigate through the vast ASML infrastructure landscape, always pointing me in the right direction to find the resources or people I am looking for, and giving in-depth feedback on my proposals from an industry standpoint. You made my experience in ASML worthwhile by providing enough help to not get stuck on difficult tasks, while still ensuring I have plenty of room to learn and explore myself.

I also want to thank ASML for providing this opportunity and all the office workers that aided me with progressing with my project. Thanks to Jeroen Veenendaal for supervising me and helping me officially integrate into ASML. I am also grateful to Frans van der Meeren for helping me during the first part of my internship to set up and integrate. Special thanks go to Tom Hoeken, who assisted me when I was stuck with a problem that no one knew how to solve. Additionally, I would want to thank Delano Flipse for working alongside this project and sharing both the ups and downs that come with it.

Major thanks to TU Delft University for providing me the opportunity to study here and performing my thesis with ASML. I would also like to thank all the teachers and assistants who helped me reach this point. Additionally, I would like to thank Christoph Lofi for finding time to join the thesis committee and reviewing my work.

Lastly, thanks to all my friends and family who kept me going through not just this period, but my entire stay in the Netherlands. Without you all, I wouldn't be here in the first place.

Abstract

Apache Spark is a popular batch processing framework that is integrated into the ASML data analytics platform. However, having multiple users share the same resources creates fairness and efficiency problems in the scheduler, where some users may receive more resources than others, or the scheduler slows down all jobs equally. Apache Spark provides a built-in fair scheduler that attempts to mitigate these issues, but cannot account for changing user environments and has been shown to slow down overall job execution. Previous literature has already proposed improvements to the Spark fair scheduler, however, it does not account for user contexts to ensure fair resource distribution among users, and does not account for task skews that are present in the Spark ecosystem.

In this work, we present User Weighted Fair Queuing (UWFQ) scheduler that can minimize the response time of jobs in Apache Spark, while ensuring that users are allocated a fair share of resources that they equally distribute among their jobs. This is done by simulating a virtual fair scheduler, and assigning the highest priority to jobs that complete the earliest in the virtual scheduler. The algorithm is an extension to Cluster Fair Queuing designed by C. Chen et al., with an added fairness layer to ensure that each user is entitled to a fair share of resources. To address the problem of task skew present in Spark, we introduce runtime partitioning that can more effectively avoid task skew by splitting them into more partitions. We implement our scheduler in the Apache Spark framework and show that UWFQ with runtime partitioning can decrease the average response time of small jobs by up to 70% in certain workloads, while still providing fair resource allocation in critical scenarios.

Contents

1 Introduction

1.1. Industry context

ASML is one of the world's leading manufacturers of chip-making equipment, with 60 service points located across 16 countries. While ASML itself does not design these chips, it provides important tools that are necessary to create chips and analyze the process of chip creation.

The machines produced by ASML work with nanometer precision and perform thousands of actions every second. Many of these components are tracked by sensors sampled at a similar frequency, generating up to 2 terabytes of data every week. ASML leverages Apache Spark to process data in its data analytics platform, allowing for query execution and data retrieval.

Spark allows for fast, distributed, and in-memory batch processing of large amounts of data. This framework fits perfectly together with the ASML cluster infrastructure and provides state-of-the-art Big Data processing speeds.

1.2. Problem definition

With the amount of industrial data produced increasing exponentially every year [48], there is a large demand for frameworks and systems that can process this data. Apache Spark [68] is regarded as the state-of-the-art [55] batch processing framework and is supported by major cloud service providers such as Databricks and Amazon Web Services.

Apache Spark is a continuation of the MapReduce [17] model that was first introduced by Google in 2003. Unlike MapReduce-based frameworks, Apache Spark computations rely on in-memory data, reducing intensive disk access and providing much quicker read and write operations. On top of that, Spark integrates many other business logic tools such as an SQL engine and a machine learning library that can utilize the in-memory data for fast job processing [3].

Spark is a very flexible framework and can be used in many industry fields, most notably, in analytics platforms. Analytics platforms tend to provide services to multiple users simultaneously, where each user has a limited amount of resources, but an unlimited number of jobs they can submit for execution. To ensure that each user receives the expected amount of resources, a fair scheduler needs to ensure that jobs are allocated with respect to which user they belong to, while still ensuring acceptable response time to user requests.

In most systems, this fair scheduler is implemented outside of the Spark engine, controlling the job flow before it reaches the internal Spark scheduler. Typically, this is facilitated by each user launching a separate Spark application and the cluster manager of the system (e.g., YARN, Apache Mesos) assigning a fair amount of executors to this application. Such solutions have already been explored by W. Chen et al. [13], where the cluster manager attempts to equalize the resources spread among Spark applications at all times, and G. Wang et al. [62], where resources are allocated to applications to ensure that jobs finish within their deadlines.

However, in environments where many small jobs have to be executed, launching an application for each workload induces a large relative delay, e.g., due to the time needed to start the driver program, allocate workers or to launch executors [15]. To avoid this overhead, a single long-running Spark application can be launched, which would continuously listen for arriving user jobs and execute them. This ensures that the startup overhead is incurred only once, rather than for every single job that is submitted to the system.

While this solution is more effective for avoiding application setup delays, it forwards all scheduling responsibility to the built-in Spark task scheduler. Apache Spark currently provides only 3 scheduler settings that can be configured: first-in-first-out (FIFO), fair, and fairness pools. However, all of these schedulers fail to provide consistent quality of service requirements for most users at all times.

The FIFO scheduler is not suitable for ensuring fairness, since it only schedules jobs in the order of arrival, which could starve users from accessing the system if another user schedules too many long-running jobs. The fair algorithm only considers job-level fairness rather than user-level fairness, creating conditions where users with more jobs gain more resources than users with a lower number of active jobs. Lastly, fairness pools provide the highest degree of configuration to create fairness among users, but the user pools are configured only at the start of the application, meaning that for a long-running application with a dynamic user base, this option is also insufficient.

Previous work has already recognized the lack of fairness and inefficient response times provided by built-in Spark schedulers. L. Chen. et al. [12] implement a max-min fair scheduling algorithm for Apache Spark between multiple jobs across geo-distributed datacenters. The scheduler can minimize the response time of concurrent jobs while maintaining max-min fairness. This is achieved by modeling the scheduling problem as a linear programming problem, where the worst-case task running time is optimized while trying to minimize the response times of all jobs. Although it significantly outperforms the default Spark fair scheduler, a substantial amount of time is required to solve the linear programming problems. For environments where jobs can run within hundreds of milliseconds, this presents a significant overhead for scheduling.

A more real-time scheduling solution in Spark is proposed by C. Chen et al. [10]. C. Chen et al. design a Cluster Fair Queuing (CFQ) scheduling algorithm that calculates the completion time of jobs in a fair scheduling system, and then schedules jobs in ascending order of these finish times. By running all active jobs in order rather than interleaving between them, CFQ has been shown to significantly reduce the average response time of jobs. Even with the constraint of ensuring fairness across jobs, the performance of CFQ is comparable to Shortest Remaining Processing Time First scheduling, which is highly optimized for lowering average response times [46, 10].

While CFQ presents a great solution for ensuring close to optimal response times, the fairness notation that is used for scheduling is only fair among jobs, rather than among users. We find that most available recent literature in fair scheduling only considers job fairness [33, 10, 19, 28, 30, 44]. By centering fairness around jobs, users who schedule more jobs will receive more resources in the system, which we see as unfair and unfit for multi-user, multi-job batch processing environments.

However, this is not the only problem we observe in batch processing scheduling algorithms that exist. In Apache Spark, jobs are split into multiple partitions, where each partition can be executed in parallel, allowing jobs to finish much quicker but reserving the execution unit until its fully complete. Because partitions are not always balanced in workload, this can create 2 new problems: priority inversion and skew runtime domination.

Priority inversion happens when long-running partitions block higher priority job partitions from acquiring them, creating scenarios where lower priority jobs get to execute before higher priority jobs. This can create unfairness in the scheduler, if many jobs are inappropriately partitioned. This is additionally pointed out by C. Chen et al. in CFQ implementation [10], however, no solution for this is proposed.

Skew runtime domination occurs when the longest running partition accounts for most of the time spent running the job. This reduces the effects of parallelization, since the longest running partition can only utilize at most a single core. While not directly impacting the scheduling fairness, this does negatively affect the response times of jobs by reducing the amount of parallel resources a job can utilize.

1.3. Research questions

Current built-in Apache Spark schedulers do not ensure a strong notion of fairness, do not consider user context, and do not effectively deal with long-running tasks, which results in suboptimal job response times and unfair scheduling. We have also found that the literature lacks a clear fairness definition and a fair scheduler for multi-user and multi-job batch processing environments. In this thesis, we set out to propose improvements for Apache Spark scheduling and partitioning to simultaneously decrease the average response time of jobs and ensure a consistent notion of fairness. We condense our aims into a single research question:

What scheduling and partitioning algorithms can improve the average response time of jobs while still ensuring a notion of fairness for multi-user and multi-job environments?

To address this main question, we decompose it into more specific sub-questions:

RQ1 How can fairness be defined in batch processing environments where multiple users can schedule one or multiple jobs simultaneously?

RQ2 What scheduling approaches decrease the average response time of jobs while ensuring fairness?

RQ3 How does the partitioning of jobs in batch processing frameworks affect the average response time and fairness of a schedule?

1.4. Contribution

In this work, we propose the User Weighted Fair Queuing (UWFQ) scheduler: a multi-user and multi-job scheduling algorithm that ensures user fairness while reducing the average job response time. UWFQ extends previous work done by C. Chen et al. [10], namely CFQ, by introducing another layer of fairness to account for user context while scheduling. To keep response times minimal, the same procedure of scheduling is followed by running jobs in the order of their fair scheduler completion times.

To make fairness and response times more consistent, we implement dynamic job partitioning, which mitigates the priority inversion and skew runtime domination. This is achieved by splitting jobs into finer partitions that allow for greater parallelizability and avoid long execution unit reservations.

As a summary, this work makes the following contributions:

- We define user-job fairness, an extension to fair share scheduling that can be applied to environments where multiple users schedule one or multiple simultaneous jobs, and resources must be distributed fairly at both the level of users and user jobs.
- We design and implement the UWFQ scheduling algorithm that ensures bounded user-job fairness while reducing the average response time of jobs. This is done 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 uses the estimated runtime of jobs to derive the appropriate number of partitions, avoiding priority inversion caused by limited preemption in Apache Spark, and preventing job response time extension due to task runtime skews. This algorithm provides more flexibility to UWFQ and further improves the performance of the system.
- We showcase the importance of job context inclusion in Spark scheduling, and its impact on response times.
- We implement UWFQ with dynamic partitioning in the Apache Spark framework, and evaluate its effects on fairness and job average response times. The evaluation is performed on micro-benchmarks that highlight certain scenarios that most schedulers handle inefficiently, and macro-benchmarks that are constructed from Google traces [23] to see scheduling performance under real user behavior.
- We show that UWFQ with runtime partitioning can reduce the response times of small and medium-sized jobs by up to 78% in homogeneous workloads, and 58% in heterogeneous workloads, in comparison to a plain user-job fairness implementation. We additionally showcase that UWFQ more performant than CFQ and Spark's built-in fair scheduler in multiple micro-benchmarks.

The rest of this thesis is structured as follows. Chapter 2 describes the necessary background about Apache Spark and well-known scheduling methods and concepts that will be a part of UWFQ's design. Chapter 3 addresses the target system we are building our scheduler for, and classifies the scheduling problem using more objective scheduling notation. Chapter 4 analyzes existing scheduling methods of the literature, and presents the research methodology we used to identify them. Chapter 5 details the design of UWFQ and dynamic partitioning. Chapter 6 provides additional implementation details of how UWFQ is integrated into the Apache Spark framework and external system. Chapter 7 evaluates UWFQ's performance and compares it with other related scheduling algorithms. We analyze the performance of specific components of UWFQ to pinpoint the reasons behind this improvement. Our results indicate that UWFQ can achieve similar average response times compared to CFQ while ensuring a more consistent performance in critical scenarios. Finally, the thesis is concluded by answering the research questions and discussing future work in Chapter 8.

2

Background

In this chapter, we cover the necessary background knowledge needed to understand the context of the problem and related concepts used in UWFQ's design. We first introduce the Apache Spark framework and its most relevant components. Then we discuss well-known fairness notations and respective scheduling algorithms that attempt to implement fairness under real hardware constraints. Finally, we go over existing ways that fairness has been measured in literature.

2.1. Apache Spark

Apache Spark is a multi-language analytics engine for large-scale data processing. It consists of many components that work together to provide fast and efficient service for analytics jobs. In this section, we clarify key terminology used throughout this document, and introduce the components that explain how user-provided queries are executed in the Spark engine.

2.1.1. Core terminology

Apache Spark has its own internal infrastructure that breaks down user submitted jobs into smaller and more operable units. To make it easier to address and distinguish these units, most of them are designated using a specific term. We highlight the following definitions:

- **Resilient distributed dataset** (**RDD**) - an immutable dataset partitioned across multiple nodes and stored in their main memory. This is the main component that allows Spark to achieve fast computation times, by utilizing both quick main-memory access and performing operations in parallel for each separate partition.
- **Spark context** - an entry API for applying operations on RDDs. This is the core interface through which user queries are transformed into RDDs for execution.
- **Driver program** - the process executing the main function and initializing the Spark context. While usually driver programs run a single user query or a workflow, it can also be a long-running Spark application that facilitates multiple users and continuous scheduling of jobs.
- **Transformation** - an operation applied to an RDD and produces a new RDD. These operations are performed lazily, allowing for chaining of multiple transformations, and only performing the execution once a result is requested by issuing an **action**.
- **Action** - executes a function on the supplied RDD and returns the value to the driver program. Actions trigger the execution of all the lazy operations that were applied to the RDD, which internally is represented as a **job**.
- **Jobs** - highest level work unit that can be represented internally in Spark. However, multiple Spark jobs may be generated for a single user query. Once a job is received in the Spark context, it is forwarded to the **DAG scheduler**.
- **Directed Acyclic Graph (DAG) scheduler** - scheduling component that picks up jobs and breaks them down into a DAG of **stages**. Stages that have no dependencies are then forwarded to the

Task Scheduler.

- **Stages** - sets of **tasks** that group together multiple operations to be executed without requiring a shuffle. A shuffle is an operation where data is exchanged across multiple nodes to continue execution. A shuffle boundary represents a border of all operations that can be executed without a shuffle. For example, functions map and $filter$ can be computed together within a stage without a shuffle, since no data is required to be transferred across the cluster to complete them. Stages are constructed from a single job, where operations are grouped together until reaching a shuffle boundary.
- **Tasks** - task is the smallest unit of computation within the scheduling hierarchy. All tasks of the respective stage execute the same functions on different partitions of the same RDD. The total task amount is equal to the number of output partitions of the corresponding stage.
- **Task Scheduler (TaskSchedulerImpl)** - component responsible for deciding the priority of tasks and scheduling tasks onto executors. Task Scheduler uses stage metadata to determine its priority, and assigns executor slots to the tasks associated with the highest priority stage. It additionally accounts for task locality while scheduling, trying to minimize the communication costs.
- **Spark Listener** - interface that allows for the monitoring of events happening within the Spark engine, and triggering callback functions for handling these events. The granularity of these events ranges from job-level to task-level events.

2.1.2. Job execution

Figure 2.1 illustrates a flowchart visualizing job execution. Users construct their jobs by applying transformations to RDDs, and finalizing results by issuing an action, which internally triggers a Spark job (1). Once a 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 is done by first creating a result stage from the associated action, and recursively adding dependent map stages until no dependencies are found. Stages are then partitioned into tasks (2), and submitted to Task Scheduler if all of their dependencies are satisfied (3). Task Scheduler keeps track of all schedulable units using Root Pool. Root Pool contains both individual stages and other layers of pools used for establishing priority hierarchies. In this example, we don't consider additional pools for simplicity reasons.

Within Task Scheduler, stages are represented as Task Managers that additionally keep track of their task state. Once the cluster manager offers resources to Task Scheduler (4), it sorts Root Pool (5) using the defined scheduling policy, and submits each task one by one to be run on the executors (6). Once all tasks of a Task Manager have been executed, the stage dependency graph is updated (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 (8).

2.1.3. Spark SQL interface

When working with relational data, Spark provides built-in SQL support for executing SQL queries on the same Spark engine that is used for RDD computation. This is mainly done by interacting with the Dataset interface, where data is organized similarly to a table in a relational database. Datasets have the same properties as RDDs, with the added benefit of being able to use SQL optimization to make querying faster. Another API that can be used is the DataFrame API, which provides similar benefits as Datasets, but mainly works with untyped data. Since the majority of operations performed within the analytics platform are on relational data, the target system mostly interact with the Spark SQL interface.

Query optimization is performed by a component named Catalyst, whose procedure is shown in Figure 2.2. Given SQL queries or DataFrame operations, Catalyst constructs an unresolved logical plan, which is represented by a tree of operations to apply to the target data. Catalyst then resolves the attributes by matching them with data sources, producing a resolved logical plan. This plan is then optimized through a set of rule-based transformations to produce an optimized logical plan. This plan is then transformed into one or multiple physical plans, where the best one is selected for Whole Stage Code Generation (WSCG). WSCG is a fusion of multiple SQL operators, such as filter or map, into a single Java function to improve the performance of query execution.

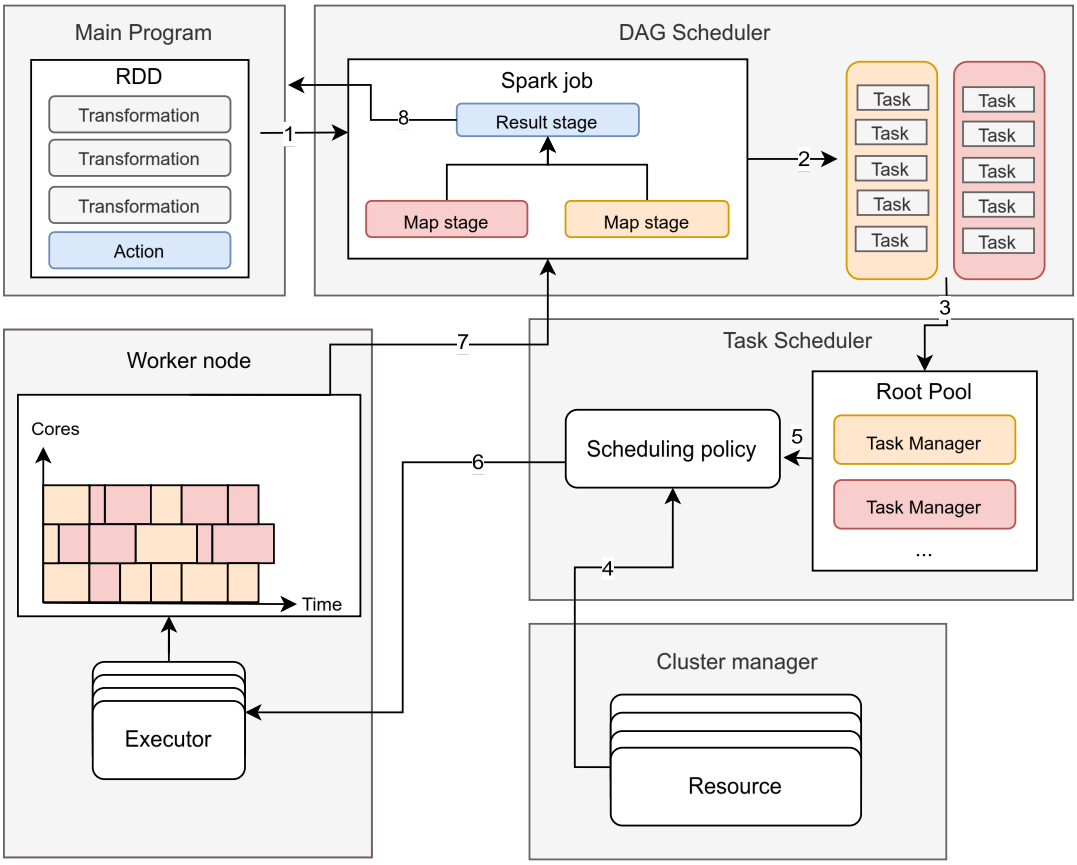


Figure 2.1: Flowchart that illustrates the execution of user-constructed RDDs on cluster resources.

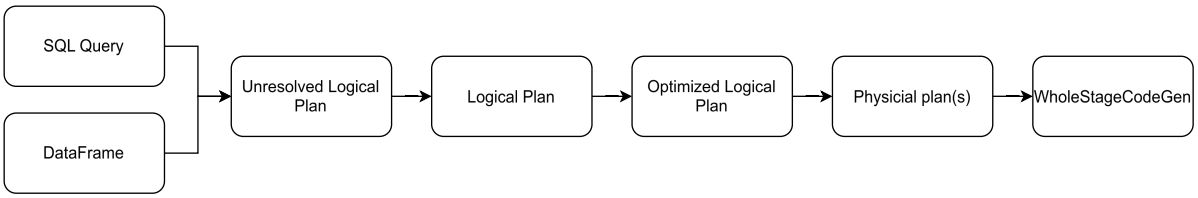


Figure 2.2: General workflow of converting SQL queries to executable code. Note that Spark only creates one physical plan if the cost-based optimizations are disabled.

2.1.4. Partitioning

Stages work on data that 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 partition.

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 amount. When data is first retrieved, the partition amount is determined by dividing equal amounts of data per available core 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 amount based on the advised size of partitions, or again, trying to maximize parallelism.

2.1.5. Adaptive Query Execution (AQE)

AQE is a Spark component that uses the runtime statistics of a Spark plan to optimize it. To not interfere with actively running tasks and not disrupt the flow of execution, AQE only activates between stages, so that it can gather statistics on all previously executed partitions and decide on changes to apply for the

upcoming stages. It follows multiple rules to enable performance gains, where the most notable ones are coalescing shuffle partitions, switching between join strategies, and balancing data skews.

Coalescing shuffle partitions adapt the number of partitions that are being transferred during the shuffle stage. By default, shuffle stages do not take into account any metadata about intermediate result sizes, hence producing partition sizes that are too small and cause scheduling and reading overhead.

Changing join strategies can decrease the amount of operations that need to be applied to perform the join. The switching is mainly done between sort-merge and broadcast hash join. If one of the join tables is small, broadcast joins are much quicker since sorting can be skipped, and the entire hash table can be stored in memory. This is not the case when both sides of the join are large, because sort-merge is then much more performant.

Skews tend to arise from the data being unevenly distributed among partitions. Using shuffle file statistics, these partitions can be equalized by taking larger tasks and dividing the data into more partitions, reducing the data size differences between tasks within a stage.

Internally, AQE encapsulates the entire Spark plan and performs optimizations whenever a leaf stage materializes. Optimizations are done by traversing the current plan bottom-up. As it is being traversed, statistics from completed child stages are collected and used to reoptimize the current Spark plan. The optimized plan is compared to the original using a simple cost estimator to assess whether it would offer improvements. This process continues incrementally as more stages are completed, until no further optimizations can be applied.

2.1.6. Built-in Schedulers

Spark comes with two already implemented priority algorithms: First-in-first-out (FIFO) and fair scheduling.

The FIFO algorithm schedules task sets based on the job ID they are associated with. Job IDs are assigned in increasing order of arrival, and smaller job IDs have greater priority.

The fair scheduling algorithm attempts to maintain fairness among all running jobs by giving precedence to jobs that have the least active running tasks on the executors. Its core idea is to divide the available resources among jobs equally. In case there is a tie among two jobs, the algorithm defaults to FIFO-like scheduling.

To introduce more fairness levels, Spark also provides fairness pools that allow for defining a fairness hierarchy among jobs. Pools have minimum share and weight properties, which respectively indicate the minimum number of tasks each pool will always have priority to run and the priority of each pool in comparison to competing pools. During task scheduling, first, the pool with the highest priority is determined based on its minimum share and weight, then the task within the pool with the highest priority is selected, which depends on the predefined pool scheduling policy. The two pool scheduling policies that are currently offered are FIFO and Fair.

Once the priority of a task is determined, the task is allocated to the resource that has the highest task locality. For example, if there are 2 nodes, where one node already contains the input data needed for computing a task, it will be preferred over a node that would need to perform a shuffle read operation.

While providing different scheduling options, Spark still lacks the necessary components to perform fair user-level scheduling in the context of long-running applications. First, the FIFO scheduler only schedules tasks based on their arrival times. Second, the fair algorithm only considers job-level fairness rather than user-level fairness, creating conditions where users with more jobs gain more resources than users with a lower number of active jobs. Last, fairness pools provide the highest degree of configuration to create fairness among users, but the user pools are configured only at the start of the application, meaning that for a long-running application with a dynamic user base, this option is also insufficient.

2.1.7. Observed problems in Spark

While working with Spark, we observed some behaviors that hurt execution performance if not accounted for manually. We are unaware if this is intended Spark behavior, but address this due to its impact on design decisions relating to job context and partitioning.

AQE splitting stages into new jobs

Idiomatically, stages are executed within the context of a particular job and are only accessible within the context of the respective job. But this does not seem to hold if AQE is enabled. Due to AQE constantly changing the Spark plan, stages seem to be separated into multiple jobs, which then converge into the original job. Stages that were planned for the original job are then skipped, and their results are replaced with the outputs of the separated jobs. An example of this occurring can be seen in Figure 2.3.

While this does not seem to impact the performance negatively, it does make job execution more obfuscated and interferes with some scheduling algorithms. For example, FIFO scheduling gives preference to jobs that arrived first to the DAG scheduler. But since AQE spawns new separate jobs, their priority is now lowered since they came into the system later and are not directly associated with the original job.

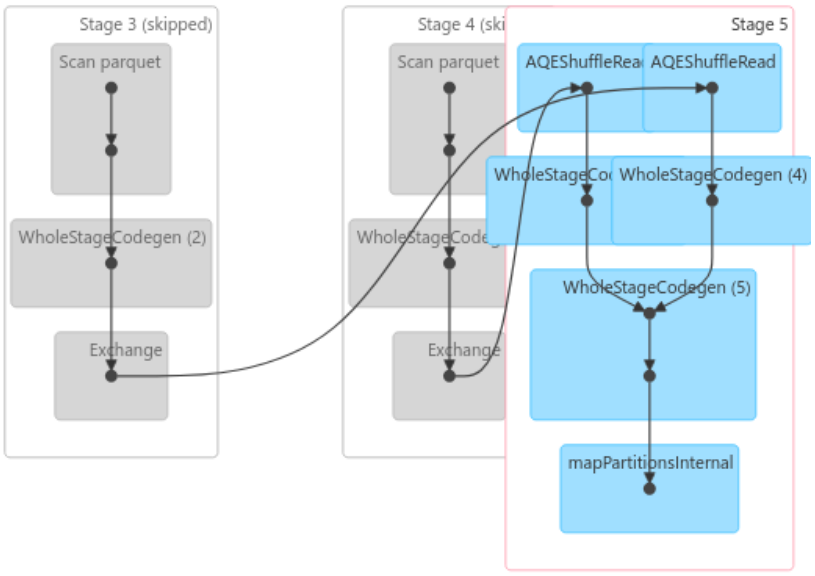


Figure 2.3: AQE skipping stages within a job, due to stages being executed on separate jobs.

Partitioning of many small files

Spark uses multiple user-defined parameters to partition files. The most notable one that can cause performance degradation is $openCostInBytes$. This option adds a constant amount of bytes to every file that is read, allowing for smaller files to still be spread across multiple partitions for parallel execution. However, this benefit diminishes as the number of small files grows. This is most noticeable for partitioned Parquet files.

Apache Parquet is a column-oriented relational data storage format, which is targeted for making aggregation and other analytical queries more efficient. While Parquet data can be stored in a single file, it can also be partitioned into multiple separate small files based on column values. This allows query engines to quickly skip over irrelevant rows that are not considered by the query.

Partitioning Parquet files is good practice for environments where most queries only work with a subset of values. However, in Apache Spark this can cause significant slowdowns if the Spark configurations are not set properly. Max partition size is controlled by the parameter $maxPartitionBytes$, which doesn't allow partitions to grow larger than the specified amount. By having a proportionally high $openCostInBytes$, partitions fill up with only a couple of small files, which creates many partitions that perform very few calculations. In the context of Spark, we define this as the **small file problem**.

We ran local experiments with the same workload under the same conditions, while only changing the value of $openCostInBytes$ setting to control the amount of partitions (task). As Figure 2.4 shows, the stage runtime linearly increases with the number of tasks. This suggests that some scheduling delays arise when the number of task becomes sufficiently large.

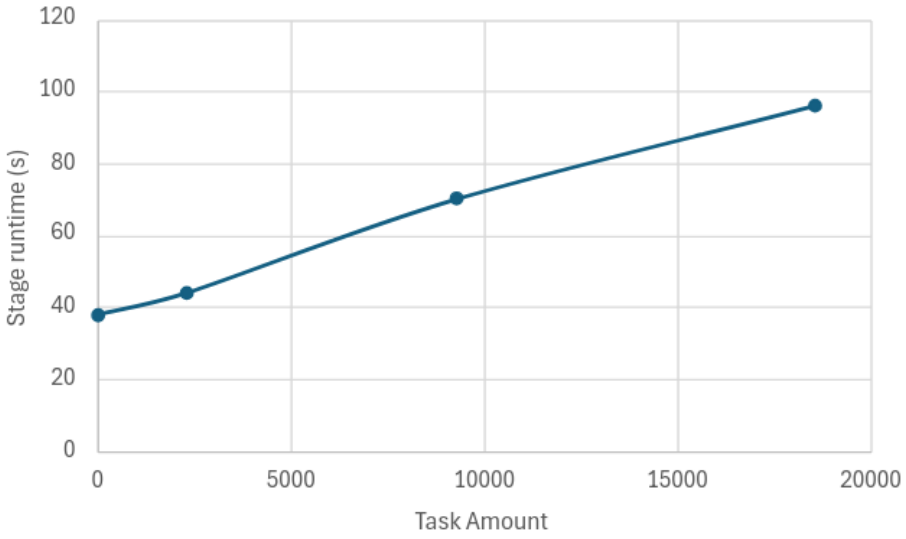


Figure 2.4: Relationship between task amount and stage runtime. The sampled task amounts are 11, 2324, 9292, and 18583

Having many small files additionally introduces other problems outside of partitioning. First, reading many small jobs is observed to cause a large constant delay compared to reading just a single whole Parquet file. We believe this is due to interfacing with the OS and needing to access multiple distributed files rather than a single entry. Secondly, Parquet the smallest splittable unit is a row group, which cannot be split into smaller portions. Because of this, if the partition amount is larger than the row group amount, some partitions will be empty. Empty partitions do not perform any computation, however still require to be scheduled, inducing a small scheduling delay that can add up.

2.2. Fair scheduling

Fairness in scheduling has been a widely addressed and discussed topic, but it still does not have a clear and encapsulating definition. In this section, we examine existing notions of fairness in job scheduling. First, we look at theoretical schedulers and commonly used fairness properties. After that we present notable fair scheduling implementations that try to approximate fairness under realistic settings. Since many of the proposed fairness definitions are framed in the context of networking, we recontextualize them for multi-user and multi-job environments.

Table 2.1: Notation definitions

Notation	Explanation
$_i$	Job i
$_k$	User k
R	Resource share
w	Weight
P	Priority
H	Historic resource usage
N_u	Amount of users
N_j	Amount of jobs

2.2.1. Theoretical schedulers and scheduling properties

In this section, we look at theoretical schedulers that are generally used in the literature as baselines. Additionally, fairness properties are discussed that define attributes a scheduler should have to ensure a certain notion of fairness.

Generalized processor sharing

Generalized processor sharing (GPS) is an idealized scheduling model that achieves perfect fairness [35] among jobs. GPS is said to be achieved if and only if it is work-conserving and always allocates resources R_i exactly proportionally to jobs' weight w_i as seen in Equation 2.1.

$$R_i = w_i * R \tag{2.1}$$

GPS has no feasible implementations since it requires jobs to be divisible into infinitesimally small quanta, and all resources to run scheduled job quanta simultaneously. However, it provides a useful theoretical baseline for comparing other scheduling algorithms.

Fair-share scheduling

Fair-share scheduling distributes the resources equally among all users in the system [31]. 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 defined in Equation 2.2.

$$\begin{aligned} R_k &= \frac{R}{N_u} \\ R_i &= w_i * R_k \end{aligned} \tag{2.2}$$

This is a very straightforward fairness strategy that provides each user with 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

Max-min fairness is a scheduling property in networking where each participant can only gain more resources if and only if it does not result in a decrease of resources for a participant with a lower or equal share [56]. 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 achieve maximum throughput or account for job priority.

Proportional-fair scheduling

To have finer control between fairness and throughput/priority, proportional-fair scheduling can be used. Proportional-fair scheduling tries 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 [1]. For the multi-user and multi-job environment, we define it in Equation 2.3

$$P_k = \frac{P_i}{H_i^\alpha} \tag{2.3}$$

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. α 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 α, a good compromise between job priority and fairness can be achieved.

Dominant resource fairness

In some systems, jobs may share more than a single type of resource, for example, memory, CPU time, disk space, etc. To establish fairness in a multidimensional resource system, dominant resource fairness (DRF) can be used. DRF is a generalization of max-min fairness for multiple resource types [21]. First, it defines a dominant share per user, which indicates the largest fraction of their requested resource type. For example, in a system of 9 CPUs, 18 GB RAM, a user requesting $\langle 1\ CPU, 3\ GB \rangle$ for each of their jobs, will proportionally be requesting 1/9 of the total CPUs and 1/6 of the total RAM for each job. This makes RAM the user's dominant resource with the fraction 1/6.

Once users' dominant resource is determined, users can be serviced in the order of the smallest possessed dominant resource. For example, all users initially start with 0 resources reserved and receive resources needed for scheduling 1 job (if there are enough resources in the system). Once each user has at least 1 job running, the user with the smallest dominant resource possessed is picked, and another of its jobs is allocated. This allocation method continues until there are no more jobs or no more available resources in the system.

DRF provides similar guarantees to max-min fairness in the context of a multidimensional resource system, and is proven to be Pareto efficient, meaning that there does not exist an allocation that would increase a user's allocation without decreasing any other user's allocation.

2.2.2. Fair scheduler implementations

In this section, we present fair schedulers that are implemented while considering the limitations of real hardware. While there are many existing fair schedulers, we only examine the ones that will be relevant in answering our research questions.

Completely Fair Scheduler

Completely Fair Scheduler (CFS) is a fair scheduler implementation mainly used by Linux systems[1]. It assigns each job in the system a virtual runtime V_r and schedules them in ascending order of these virtual runtimes. Initial virtual runtime is determined by the current smallest runtime V_r^{min} in the system, allowing for newly created jobs to be quickly serviced by the system. As the current scheduled job is running, its virtual runtime V_r increases, and once another job's virtual runtime becomes the smallest, the current job is preempted in favor of the preceding job.

To account for users, jobs can be grouped into job groups, where CPU time is then divided fairly among the job groups. Each group is assigned a weight which indicates the proportion of CPU shares it receives.

This implementation is similar to Fair-share scheduling regarding how resources are distributed among users and jobs, however it can be more fair, since it is more adaptable to variable-sized jobs.

Fair queuing

A common scheduling implementation that possesses the max-min property is Fair Queuing (FQ). FQ creates a queue for each user of the system, and services their jobs in a round-robin fashion [42]. User queues contain jobs they want to run, and can only run a single job during their turn in round-robin. Users with empty queues are skipped and do not consume any system resources.

FQ achieves max-min fairness by limiting each user with many parallel jobs to a single queue, and not providing resources for idle users. However, if job runtimes are not homogeneous, the fairness can fluctuate depending on the runtime differences of these jobs.

Weighted fair queuing

Weighted fair queuing (WFQ) (sometimes referred as packet-by-packet service discipline (PGPS)) executes jobs in the order of their GPS schedule expected finish times rather than round-robin fashion compared to regular FQ. Additionally, it uses job weights to prioritize some jobs over others.

To implement WFQ, first virtual job completion time V_f is calculated for every active job in the system as seen in Equation 2.4

[1] CFS: https://docs.kernel.org/scheduler/sched-design-CFS.html

$$V_f = V_a^i + \frac{L_i}{w_i} \tag{2.4}$$

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 [43]. This is also expressed in Equation 2.5

$$f_i - \hat{f}_i \leq \frac{L_{max}}{R} \tag{2.5}$$

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).

This scheduling algorithm closely approximates the perfect fairness of GPS and regulates the priority of jobs by adjusting their weights. However, the implementation depends on the concept of virtual time.

Virtual time

Virtual time is used to simulate the marginal rate of service that each job receives under GPS system. While it is not necessary to enable WFQ scheduling, it decreases the time complexity associated with maintaining the finish times of jobs.

Under a GPS scheduling, whenever a new job arrives, each existing job would have 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. This would require an $O(N)$ time to recalculate the finish times, where N is the number of currently active jobs.

However, by using virtual time, we can instead 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 Equation 2.6

$$\begin{aligned} V(0) &= 0 \\ V(t) &= \int_0^t \frac{R}{N_j^t}\, dt \end{aligned} \tag{2.6}$$

where N_j^t are the currently active jobs in the GPS schedule at time t.

When a job arrives in the system, virtual time is used to calculate its virtual deadline, which reflects when 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 using virtual time implementation instead of real time, we can reduce the time complexity $O(N)$ to $O(\log_2(N))$, since virtual deadlines are set only once, and the incoming jobs have to be put in an ordered list, which takes at most $O(\log_2(N))$ time. However, virtual time has to be maintained, which may cause additional delays if done synchronously.

Earliest Eligible Virtual Deadline First

WFQ has inspired several derivatives, the most notable of which is the Earliest Eligible Virtual Deadline First (EEVDF) algorithm [52]. While it still utilizes virtual time and schedules jobs in the order of their virtual deadlines, it introduces a new schedulable unit, viz., clients, and modifies virtual time to account for actual rather than predicted completion times.

Clients receive service proportionally to their resource share, similarly to how jobs receive their share in WFQ. However, clients internally define another layer of actions called requests that issue the amount of

time needed to perform its service. Requests are bounded by a fixed time quantum they can receive, which forces clients to divide their total needed computation time across multiple requests. Requests are issued consecutively, and once all of them are complete, the client leaves the system. Since requests are the units that represent execution, deadline calculation happens in the context of requests rather than clients. For client i, the eligible time V_a^k and finish time V_d^k of request k can be seen in Equation 2.7

$$\begin{aligned} V_a^1 &= V_a^i \\ V_d^k &= V_a^k + \frac{L_k}{w_i} \\ V_a^{(k+1)} &= V_d^k \end{aligned} \tag{2.7}$$

where V_a^i is the arrival time of the client, L_k is the assigned execution time to request k, and w_i is the weight of client i.

Due to requests sometimes executing quicker than the defined time quantum, the system may enter a state where virtual time is not fully representative of the real system, since a client may leave the system faster than anticipated. To account for this, the leaving times of users in the real system are tracked, and the period of time that the client didn't use is correctly added to the virtual time.

Finally, requests are scheduled in the order of their virtual deadlines V_d^k, but only the requests that have their eligible time V_a^k reached are considered. This is done to allow other clients to more gradually catch up, which can be favorable for some systems such as Linux[2], however, it introduces interleaving that can extend client finish times.

2.3. Fairness Metrics

While some schedulers and scheduling properties guarantee fairness among users, this may not always be the case in practice. In this section, we identify ways of measuring fairness that have been used in the literature.

2.3.1. Proportional slowdown

Each job has an ideal execution time that would be met on an idle system. This execution time can be proportionally compared to the execution time under a schedule in a shared resource environment to measure slowdown [29, 44].This is formalized in Equation 2.8

$$S_i = \frac{T_i}{\hat{T}_i} \tag{2.8}$$

where S_i is the slowdown of job i, T_i is the wall-clock time it took to execute the job within the schedule (including waiting time), and $\hat{T}_i$ is the wall-clock time it took to execute the job in isolation. A closely related metric, called the progress rate, captures the same proportion but inverts the numerator and denominator [8]. Additionally, these slowdowns can be lower bounded to 1, if speedups are to be discarded from the analysis [20]. To measure the unfairness of the entire schedule, we can calculate the average slowdown S of the job using Equation 2.9

$$S = \frac{\sum_i^n S_i}{n} \tag{2.9}$$

where n is the total number of jobs executed within the schedule.

Additionally, slowdowns can be grouped per job length to examine how the scheduler affects jobs of different sizes [13, 10].

[2]Linux kernel scheduler: https://docs.kernel.org/scheduler/sched-eevdf.html

2.3.2. Deadline violations

If jobs have defined deadlines, deadline violations or the amount of time a deadline has exceeded can be calculated to measure how unfair a scheduler is [6]. We formulate a violation V_i and the unfairness U of a schedule in Equation 2.10

$$V_i = \begin{cases} E_i - D_i & \text{if } E_i > D_i \\ 0 & \text{otherwise} \end{cases}$$
$$U = \sum_{i}^{n} V_i \tag{2.10}$$

where E_i is the end time of job i, D_i is the deadline of the job i, and n is the amount of jobs in the schedule.

While not directly applicable to our target system, we can interpret deadlines as the time the jobs would finish under a fair scheduler, either idealistic like the GPS, or practical like the built-in Spark fair scheduler.

2.3.3. Equalizing cost

Jobs can have soft and hard deadlines, which may increase the cost for the user depending on when they are executed. In this case, fairness can be expressed by attempting to equalize the "suffering" among all users [47]. We express unfairness U in Equation 2.11

$$U = \frac{\sum_{i}^{n} |C_i - \bar{C}|}{n} \tag{2.11}$$

where C_i is the normalized cost of job i, and $\bar{C}$ is the normalized average cost of all jobs.

3

System model

In this chapter, we model the target system and our scheduling approach using existing work in the literature and concepts established in Chapter 2. This will clarify the workload structure and desired fairness in the target platform, and classify the scheduling problem and solution under objective terms used in existing scheduling systems.

3.1. Workload environment and fairness objective

Our target system uses Apache Spark as its batch processing engine, and runs it as a long-running Spark application. Each user of the system schedules everything through the same Spark application, forwarding most of the scheduling decisions to the Spark built-in scheduler. While the Spark scheduler has fine control over how jobs are scheduled, this is not utilized to its fullest potential to reduce the response time of jobs, or guarantee an acceptable level of fairness.

In this section, we describe the type of work the Spark scheduler has to account for, and define the fairness objective we aim to achieve.

3.1.1. Query and workflow scheduling

The target system supports both query and workflow execution. While queries in Spark are similar to any other SQL engine, it additionally allows for executing user-defined functions on relational data. While not being directly related to scheduling itself, having user-defined functions creates more dynamic runtimes of queries, making it harder to estimate accurate query runtimes reliably.

A workflow can be represented as a DAG where each vertex represents a Spark job or a query, and each directed edge represents a data dependency between jobs. A job may only execute once all its incoming data dependencies are satisfied. A workflow starts at the DAG node that has no incoming edges (entry node) and ends at the node with no outgoing edges (exit node). To combine both queries and workflows under the same system, we model a query as a workflow with a single node, representing both the entry and the exit node. A workflow is considered complete only when the exit node or nodes of the DAG have finished executing. This is a similar property applicable to our target system, since only the final output of the job is meaningful.

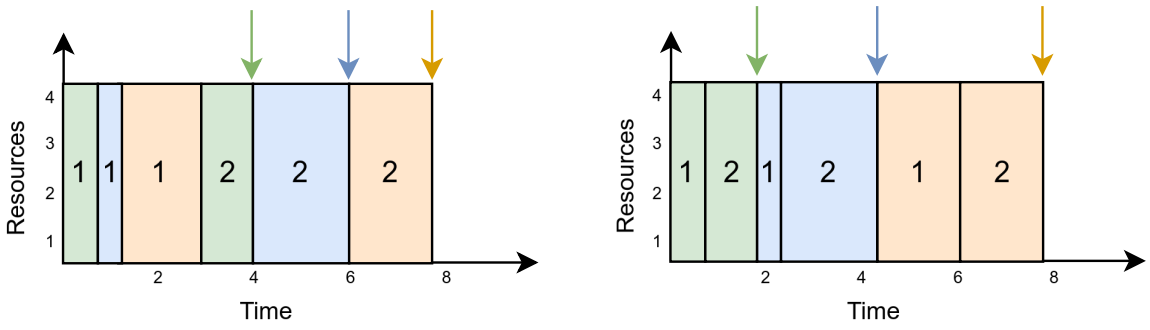
While Spark jobs themselves resemble a workflow, the layer we are discussing is outside of Spark internals, meaning that Spark does not use any of the workflow contextual information while scheduling by default. We believe that by adjusting the Spark internal scheduling to account for workflow data, the jobs can be executed more efficiently, reducing the overall job response times.

3.1.2. Job context

In Apache Spark, the highest level of work unit abstraction is a Spark job, which is the entity that encapsulates all job priority-related metadata. However, this is insufficient for our target system, due to having higher abstraction layers such as workflows or multi Spark job queries. Because users only

receive utility from workflows or queries that have fully finished, it is more efficient to model job priorities with respect to their highest abstraction level.

To implement this, we propose embedding 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, as we show in Figure 3.1.



(a) Interleaving

(b) Job context aware

Figure 3.1: Response time impact of job context. Jobs in this case are split into two separate stages, and arrows indicate when the corresponding color job has finished. It can be seen that in Figure 3.1a interleaving causes jobs to all finish later than expected, while in Figure 3.1b allows jobs to run till completion and improve average response times.

3.1.3. Fairness properties

The target system we are defining our fairness for is an analytics platform that does not have any real-time constraints imposed on it. The system has to accommodate multiple users who can each schedule multiple simultaneous 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. For users, jobs only provide utility once they are 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. We examine the fairness policies and implementations addressed in Section 2.2 on how they fit in this multi-user and multi-job environment.

Fairness policies

The most important requirement for a fair scheduling policy in this environment is to ensure fairness across users. This already eliminates GPS scheduling on the basis that it only ensures fairness across jobs, rather than users.

Fair share scheduling seems to be the most applicable for this environment, since it defines equal resource sharing across users. However, the priority assignment of jobs is left to the system requirements to define, and is not contained within the definition.

Max-min fairness is very similar to fair share scheduling, however, it allows for more needy users to take a larger portion of the available resources. While this can be a good model for systems where some users do not need the entire share of their resources, this is not the case for batch processing systems, where having more resources allows for faster execution.

Proportional fair scheduling defines a middle ground between throughput and fairness, however has a very loose definition of what would be considered unfair. In the target system, we would like to define fairness as a strict policy.

Finally, DRF fairness can be useful for systems that need to achieve fairness across multiple resource dimensions, however, this is not the case for the target system, where only CPU time is the limited resource.

Implementations
While there already exist fair schedulers that provide a certain notion of fairness, they either are not optimized for improving job response times or model the fairness inappropriately.

Both CFS and EEVDF are designed for a system where many continuous processes need consistent resource allocation to provide a certain service. For example, when a user wants to listen to music while performing other jobs on the system, the music requires consistent and continuous allocation of resources to provide the necessary service, while still allowing the user to perform their other jobs without significant interference. However, in batch processing frameworks, users only gain utility once a job is fully finished, hence not benefiting from the interleaving that is present in CFS and EEVDF. While they ensure fair scheduling in respect to users, they are not built for optimizing for response times.

However, in WFQ scheduling, interleaving is removed by scheduling jobs by their expected finish times in GPS scheduling. This avoids the interleaving of jobs, optimizing response times, and ensuring bounded GPS fairness. However, as discussed, GPS fairness is not fit for our fairness model due to the omission of user context while assigning weights to jobs.

3.1.4. User-job fairness

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, viz., fair share scheduling, to encapsulate the necessary fairness we would like to achieve.

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 in Equation 3.1

$$\begin{aligned} R_k &= \frac{R}{N_u} \\ R_i &= \frac{R_k}{N_j^k} \end{aligned} \tag{3.1}$$

where R_k^i are user k's fair share, N_u is the amount 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 the jobs' share.

The major difference between regular GPS and user-job fairness 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.

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 and while still ensuring gradual progression without starvation. We propose this as the answer for the first research question ***RQ1***.

3.2. Scheduling problem definition

To define our problem with more objective terms, we use the taxonomy provided by Versluis and Iosup [59] and Lopes et al. [37]. This can help to more objectively assess the scheduling problem, and help identify its use in other systems with a similar scheduling environment. We summarize the key features of the scheduling problem in Table 3.1 and the solution in Table 3.2.

Workload
The workload consists of multiple users scheduling multiple jobs at any point in time. Jobs are heterogeneous and consist of multiple tasks that have to be completed with respect to job dependency. Jobs are moldable, meaning that the amount of resources a job can execute on can be decided by the scheduler. Workload composition mainly consists of MapReduce jobs, with a diverse structure

Class	**Feature**	**Target**
Workload	Job source	Multi-user, multi-job
	Job structure	Heterogeneous dependant jobs
	Job flexibility	Moldable
	Arrival process	Open
	Workload composition	Same model/diverse (MapReduce)
	Quality of service	Best effort
	Real time	No real time
Resources	Heterogenety	Homogeneous
	Scaling	Static
	Sharing	Dedicated containers
	Geographical coverage	Local
	Federation	Single domain
Requirements	Scheduling goal	Fairness; Response time; Throughput
	Level	Job and task-level
	Data locality	Core affinity
	Failure model	Failure-aware
	Adaptability	Adaptable

Table 3.1: Formal description of the scheduling problem

Feature	**Target**
Optimality	Sub-optimal, heuristic
Operation	Online
Topology distribution	Non distributed
Flexibility	Flexible, Non-migration-aware, non-preemptive

Table 3.2: Formal description of the proposed UWFQ scheduling algorithm

consisting of different mixes of operations. The workloads are handled best-effort, trying to maximize system performance, while ensuring a notion of fairness.

Resources
Resources are generally homogeneous, consisting of many CPU cores and distributed RAM. The cluster is located locally and does not scale with the increase of demand and has a limited capacity. Each executor runs in a dedicated container that is federated by a single domain.

Requirements
The scheduler aims at minimizing response time, while maximizing throughput and ensuring a notion of fairness. Since the scheduler is implemented in Spark, scheduling happens at both the task and job levels, and can account for data locality as close as core affinity. Additionally, the scheduler adapts to new incoming users in the system and can tolerate failures of tasks or jobs running in the system.

Solution
The proposed UWFQ scheduler uses a heuristic approach that produces a good but suboptimal schedule. Scheduling is centralized, takes place online, and is flexible to changes based on job priority. Once tasks are scheduled, they cannot be migrated or preempted.

4

Related work

In this chapter, we describe the methodology used to find relevant papers on fair scheduling and related topics, and discuss the relevant papers in detail. Additionally, an ad-hoc exploration of performance prediction is done to analyse relevant work that can be used for Spark job runtime estimation.

4.1. Methodology

To obtain literature within this research topic, a rigorous search methodology is employed following the guidelines of Zhang et al. [69]. Using a systematic approach for finding relevant literature in a field of science has been shown to increase the amount of relevant papers retrieved and reduce reliance on researchers' background for constructing effective querying strategies [69].

Estimating the relevancy of retrieved literature can be done by comparing it to the "gold standard". The gold standard refers to the set of all primary studies that are relevant to the current research question (total number of relevant studies). Obtaining the true gold standard is difficult before surveying the research space, so to obtain a reasonable estimate, Zhang et al. defined the "quasi-gold standard", which is a set of relevant studies taken from related publication venues within a specified timeframe [69].

Two key metrics are used to estimate the quality of a search query: sensitivity and precision. Sensitivity is defined as the portion of relevant studies retrieved for the target research question (Equation 4.1), while precision is defined as the portion of retrieved studies that are relevant (Equation 4.2). In other words, sensitivity describes how much of the retrieved literature covers the field under study, while precision describes how much of the retrieved literature is relevant to the research question. The key principle is to attempt to maximize both of these metrics (maximum being 100%), but this is rarely possible in practice.

$$Sensitivity = \frac{Number\ of\ relevant\ studies\ retrieved}{Total\ number\ of\ relevant\ studies} \tag{4.1}$$

$$Precision = \frac{Number\ of\ relevant\ studies\ retrieved}{Number\ of\ studies\ retrieved} \tag{4.2}$$

Sensitivity and precision are calculated using the gold standard, which is unknown at the start of this survey. Hence, "quasi-sensitivity" and "quasi-precision" are calculated, which estimate the original metrics by replacing the gold standard with the quasi-gold standard. Using these estimates, we can refine search queries multiple times to improve their underlying sensitivity and precision.

When performing the search, we want to find as much relevant research as possible while omitting irrelevant papers. As recommended by H. Zhang et al., this can be achieved by keeping quasi-sensitivity above 80% and keeping precision within 15-25% [69]. We set these as our threshold values. Once those thresholds are reached, we will stop improving the search query.

4.1.1. Manual search

Scheduling has been a problem that dates back to early computers, thus, the relevant research timeline can go from before the 2000s until now. To reduce this timeframe and focus on more recent research, we decided to set the cut-off point to 2014, since this is the year Apache Spark was officially released. This should be a sufficiently large period of time, and will include papers and surveys that can encapsulate the most important findings before 2014.

To find the papers for the quasi-gold standard, the TPDS journal and the OSDI and CCGRID conferences were manually scanned for the defined period. We selected a small subset of conferences since going over these conferences already yielded a significant amount of relevant papers, and some ad-hoc searching was performed initially to gather papers from a more diverse set of conferences or journals.

Ad-hoc searching was performed using online search engines, namely, Google Scholar and Web of Science. The keywords that were included in the search queries with a mix of logic operators were: "scheduling", "Apache Spark", "fairness", "makespan", "throughput", "workflow", and "online".

4.1.2. Snowballing

Snowballing is a surveying strategy where more relevant research is found by examining papers related to primary literature. Backward snowballing refers to reviewing the references cited by the primary literature, while forward snowballing involves examining papers that cite the primary literature. We utilize both snowballing strategies to complement the primary literature with papers that may have been missed by both manual and query searching methods.

4.1.3. Inclusion and exclusion criteria

It is important to define explicit inclusion and exclusion criteria to avoid bias in paper selection and have more reproducible results. We define the inclusion and exclusion criteria for our main research question in Table 4.1

Table 4.1: Inclusion and exclusion criteria of retrieved literature for the primary research question

Inclusion Criteria	Exclusion Criteria
1. The paper is within the field of workload scheduling on multiple resources. 2. The paper is freely accessible to a TU Delft University member. 3. The paper is published or a part of a scientific journal, conference, or book.	1. Scheduling is not the primary focus of this paper. 2. Secondary studies, such as literature reviews or surveys. 3. Scheduling is performed for a specific resource or workload model unrelated to the target system.

4.1.4. Collecting primary literature

To taxonomize and extract information from the available metadata in the articles, a tool created by Versluis and Iosup named AIP is used [60]. AIP allows unification and filtering of scientific article metadata from multiple databases and creation of queries that highlight and quantify trends and relevant keywords.

Using the method proposed by Zhang in unison with AIP, we can perform multiple search query iterations to obtain the most qualitative set of relevant literature. The final query can be found in Appendix A. An outline of this method can be seen in Figure 4.1. The detailed steps are as follows:

1. Gathering a manual set of relevant papers by scanning through relevant conferences and online search engines. This will be used to construct the quasi-gold standard.
2. Perform keyword extraction from the set of retrieved papers and use these keywords to construct a search query to gather more relevant papers for this research topic.
3. Calculate the quasi-sensitivity and quasi-precision. If it is within the defined boundaries, continue to the next step; otherwise, repeat the above step.

4. Perform snowballing on all retrieved papers to find any literature that may have been missed with the manual and automated searching.
5. Finally, apply inclusion/exclusion criteria to the set of retrieved papers to obtain primary studies.

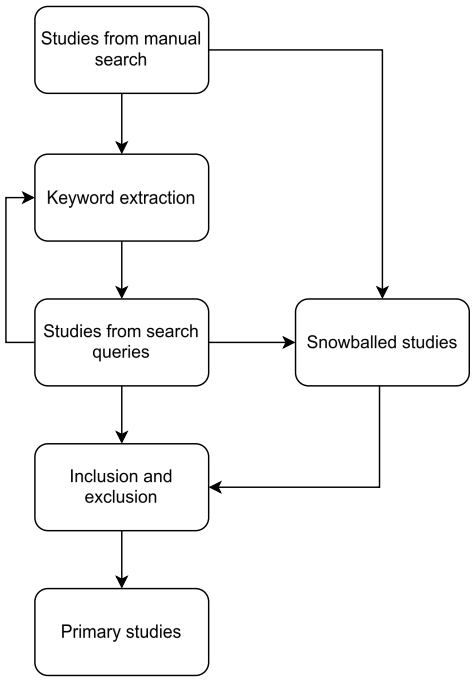


Figure 4.1: The flow diagram of obtaining primary literature using quasi-gold standard method [69].

4.2. Scheduling Algorithms

In this section, we analyze the primary papers found using the methodology described in Section 4.1. We divide surveyed scheduling algorithms based on their target workloads. This includes batch processing, workflow, and heterogeneous jobs.

4.2.1. Summary of findings

We summarize key attribute of the reviewed literature in Table 4.2.

We mainly examine papers in 3 key aspects: fairness model, response time improvement, and real-time constraint. The fairness model indicates what type of fairness constraint this scheduling algorithm aims to maintain or attempts to achieve. Response time improvement lists the core method the paper proposes to improve the response times of jobs with. Finally, we examine whether the scheduling algorithm can maintain real-time constraints within the target system to avoid scheduling delays.

We briefly describe the values used to characterize each aspect:

Fairness model

- **WFQ, Max-min, Fair share, Proportional fair** - These fairness models were discussed in Section 2.2.
- **Performance** - The fairness model tries to keep the expected performance of jobs intact.
- **Priority** - The fairness model tries to assign resources to jobs according to their priority.
- **Workflow fair** - The fairness model tries to keep the progression of workflows even, without favoring a particular workflow.
- **Deadline** - The model attempts to minimize deadline violations that occur in the schedule.
- **FIFO, Shorted job first (SJF)** - Similar to **Priority**, but job priority is determined by job arrival time (FIFO) or runtime (SJF).

Response time improvement

Workload	Paper	Fairness Model	Response time improvement	Real-time constraint
Batch processing	Pastorelli et al. [44]	WFQ	Order	Yes
	C. Chen et al. [10]	WFQ	Order	Yes
	L. Chen. et al. [12]	Max-min	Order; Locality	No
	C. Chen et al.[8]	Performance	Throughput	Yes
	D. Cheng et al. [16]	Performance	Throughput	Yes
	C. Chen et al. [9]	Priority	Maintain	Yes
	W. Chen et al. [13]	Fair share	Maintain	Partial
	G. Wang et al. [62]	Fair share	Maintain	Yes
Workflow	Topcuoglu et. al. [57]	None	Throughput	Yes
	Djigal et al. [18]	None	Throughput	Partial
	Ilyushkin et al.[29]	Workflow fair	Throughput	Yes
	Ilyushkin and Epema[28]	Workflow fair	Throughput	Yes
	Ferreira da Silva et al. [19]	Workflow fair	Throughput	Yes
	Rezaeian et al. [47]	Workflow fair	Order	Yes
	Cadorel et al. [6]	Deadline	Order	Yes
	H. Chen et al. [11]	Deadline	Order	Yes
Heterogeneous	Lifka et al. [36]	FIFO	Order	Yes
	Talby et al. [54]	FIFO	Order	Yes
	Gomez-Martin et al. [22]	FIFO	Order	Yes
	Carastan-Santos et al. [7]	SJF	Order	Yes
	Gaussier et al. [20]	Priority	Order	Yes
	Murad et al. [41]	Priority	Order; Throughput	Yes
	Grandl et al.[24]	Proportional fair	Order	Yes
	Hu et al. [27]	SJF	Order	Yes
	Zhao [70]	Fair share	Order; Throughput	No
	Shafiee and Ghaderi [50]	Max-min	Order	No
	Barika et al. [5]	None	Order; Throughput	Partial
	Chen et al. [14]	None	Throughput	No

Table 4.2: Summary of all reviewed related work on fair and response time aware scheduling.

- **Order** - Response times are improved by reordering jobs in a more effective manner.
- **Locality** - Response times are improved by considering job locality on executor machines.
- **Throughput** - Runtimes are improved by increasing overall throughput, consequently lowering response times.
- **Maintain** - Response times are not improved, but maintained to acceptable level.

Real-time constraint

- **Yes/No** - The algorithm does/does not meet the necessary time constraints of the scheduling environment.
- **Partial** - The algorithm can meet the constraints under certain conditions.

4.2.2. Batch processing schedulers

Some existing work already targets batch processing framework scheduling. Batch processing frameworks tend to have multiple layers of job granularity that the scheduler has to account for: jobs, stages, and tasks. While tasks are the only schedulable units, the higher-level abstractions determine the priority and dependency amongst the tasks.

Pastorelli et al. [44] implement a fair scheduler in Hadoop that is analogous to WFQ (see Subsection 2.2.2). This is achieved by creating a concept of virtual size, which is based on existing concepts of virtual time and aging present in WFQ. The scheduler then uses virtual size to schedule the smallest jobs first, and as the time progresses, shrinks all the unscheduled jobs. This avoids starvation while ensuring that the smallest jobs get scheduled faster. This implementation was later extended by C. Chen et al. [10] for Spark. It expresses job deadlines in virtual time, and progresses virtual time based on the number of concurrently running jobs in the system. These virtual deadlines represent when the jobs would finish under max-min fair scheduling. Jobs are then scheduled in ascending order of these deadlines. Both of these implementations show great results in improving the response time of jobs, while still ensuring fairness across the active jobs in the system.

L. Chen. et al. [12] implement a max-min fair scheduling algorithm for Apache Spark between multiple

jobs across geo-distributed datacenters. The scheduler is able to minimize the response time of concurrent jobs while maintaining max-min fairness. This is achieved by modeling the scheduling problem as a linear programming problem, where the worst-case task running time is optimized while trying to minimize response times of all jobs. The scheduler then achieves the defined objectives by first batching incoming jobs, and then iterating over them, assigning the computed worst running job's tasks to the datacenters, while accounting for efficient assignment of remaining tasks to reduce response time. Although it can significantly outperform the default Spark fair scheduler, the need to batch jobs together and the substantial time required to solve the linear programming problems can induce a delay that overshadows performance gains, especially for short jobs.

Some existing implementations attempt to increase the performance of schedulers by optimizing the level of parallelism. C. Chen et al.[8] propose a performance-aware fair scheduler, that uses past data to determine the effective amount of slots each job requires. This allows more parallelism, demanding jobs to acquire more resources than those jobs that do not benefit from parallelism that much. Similar work was done by D. Cheng et al. [16], where Spark default schedulers were replaced by a custom scheduler that accounts for dynamic parallelism and batching of streaming data to optimize throughput and minimize latency.

In another work by C. Chen et al. [9] more Apache Spark scheduling improvements are proposed. Since Spark's scheduler is work-conserving, it may schedule low priority tasks to available resources, even if higher priority tasks may be arriving at the scheduler soon. To avoid this priority inversion, computation slots can be reserved in advance if a higher priority job is expected to arrive shortly. This can be achieved by examining the job dependency tree to determine if high-priority tasks will follow after a job's completion. This, however, can introduce some idle periods, wasting some amount of resources. Additionally, C. Chen et al. proposes that tasks that have a heavy tail in latency distribution can be launched on different nodes that have already finished, which may avoid stragglers slowing down workflows.

For Apache Spark, there is also application-level scheduling, where the amount of resources given to each Spark application is distributed based on a cluster manager scheduling policy. A fair scheduling solution for application-level was developed by W. Chen et al. [13] and tries to equalize the amount of shares amongst all running Spark apps. To reduce the issue of long running jobs hogging the available resources of the cluster, W. Chen et al. develop a resource management system for Big Data frameworks that enables job preemption. In another work by G. Wang et al. [62], job completion times are improved by consider deadlines of Spark applications. However, these algorithms are not directly applicable to our target system, since it is a single long-running Spark application.

4.2.3. Workflow schedulers

Workflows represent jobs as DAGs, where each node corresponds to a task to execute and each edge denotes a dependency between tasks. Workflow scheduling can occur either locally, where priority is determined only among nodes within a single workflow, or globally, where priority is determined among all eligible workflow nodes across the system. A workflow is considered complete only when the exit node or nodes of the DAG have finished executing. This is a similar property applicable to our target system, since only the final output of the job is meaningful.

The most popular baseline in workflow scheduling is the Heterogeneous Earliest Finish Time (HEFT) algorithm proposed by Topcuoglu et. al. [57]. In essence, the algorithm calculates the expected distance of every task to the end of the workflow's execution and prioritizes scheduling tasks that are the furthest. The length of this distance is called the "upwards rank". Djigal et al. [18] propose an improvement to HEFT by utilizing a lookahead mechanism that additionally considers subsequent task scheduling to optimize task assignments. While these algorithms have demonstrated strong performance in minimizing response time, they do not explicitly optimize for other system metrics.

Ilyushkin et al.[29] compare two types of scheduling policies: work-conserving and plan-based. Work-conserving strategies schedule tasks if there is any available resource present, while plan-based strategies may hold tasks to perhaps create a more effective schedule. Results show that work-conserving strategies are better for resource utilization, but plan-based policies are better for optimizing different metrics, such as fairness or response times. In another work, Ilyushkin and Epema[28] use multiple scheduling

algorithms and analyze the impact of inaccurate runtime estimates on the performance of these algorithms. The most notable algorithms are Online Workflow Management (OWM) and Fair Workflow Prioritization (FWP). OWM uses the upwards rank to calculate each workflows eligible tasks, and schedules the task with highest largest critical path on the fastest processor. If that processor is busy, the task either waits for it to become unbusy, or schedules on a regular processor if execution time is quicker than waiting and execution on the fast processors. FWP calculates target slowdowns of each task and uses that to determine it priority. Workflows that are before their slowdown target receive less priority, while tasks that are closer to or beyond their slowdown receive more priority. Results show that plan-based schedulers such as OWM suffer severely in throughput from inaccurate runtime estimates, while dynamic policies like FWP only suffer by increasing average response times or slowdowns.

Ferreira da Silva et al. [19] created an algorithm that groups together small tasks to minimize transfer and scheduling costs, and optimizes the number of groups to fully utilize the system's parallelism. To make it fairer, a fairness metric was also introduced that prefers scheduling tasks from a workflow in such a way that equalizes the number of tasks each workflow has completed. Fairness is enforced by prioritizing tasks that would reduce unfairness once a certain threshold is reached. Results show that there is a tradeoff between minimizing average slowdown and having the optimal number of task groups.

Rezaeian et al. [47] calculate sub-deadlines and sub-budgets for tasks, and then use them to effectively schedule the tasks to meet these goals. Multiple selection policies are defined, with the most notable ones being weighted round robin, which selects tasks with the best resource sharing, and direct heuristic algorithm, which maximizes the fairness metric. Results show that there is a tradeoff between maximizing fairness and minimizing the makespan.

Cadorel et al. [6] propose an online scheduling algorithm, that prioritizes fairness and energy efficiency. Fairness in this context is defined as equalizing deadline violations across workflows. The main algorithm attempts to schedule tasks as near to the deadline as possible. If scheduling tasks near the deadline is not feasible, it defaults to best-effort scheduling, where preassigned tasks are removed from the schedule, and tasks are scheduled using a modified HEFT algorithm.

H. Chen et al. [11] use soft deadlines to calculate the latest starting time for each task in incoming workflows. The latest starting time indicates the latest time a task can be started to still make it to its deadline. These starting times are then sorted in ascending order and scheduled such that tasks that are nearing their deadline have the highest priority. Results show that this method can increase both resource utilization, and fairness of resource utilization, which tries to equalize resource usage amongst providing services.

4.2.4. Heterogeneous jobs

Most of the currently available resources on scheduling assume systems that deal with heterogeneous jobs. While the research space is vast, we focus on schedulers that target fairness and makespan optimization, and have a similar resource environment to our target system.

Backfilling is a popular industry strategy for heterogeneous job scheduling. It assigns the necessary amount of resources to the highest priority task, and tries to fill in the idle resources for any task that fits. There are 2 policies that are usually adopted for backfilling: conservative and aggressive. The conservative strategy ensures that no other job is delayed, while the aggressive strategy only ensures that the job at the head of the queue is not delayed. Conservative backfilling gives good estimates on when a task is scheduled and when it will be finished, however, it relies on runtime estimates being accurate. Aggressive backfilling on the other hand has shown even higher levels of utilization at the cost of variance in task wait time in the queue [51]. Backfilling is adapted for systems where static partitioning of tasks cause resource fragmentation [54, 40], where the system is underutilized since some tasks may not be allocatable due to insufficient cores.

EASY backfilling [36] is a well known implementation of aggressive backfilling, but its effectiveness has been challenged by more sophisticated implementations. Talby et al. [54] propose backfilling with slack, where each job can be backfilled if it does not compromise other job slack times, further reducing average wait times. A similar approach is presented by Gomez-Martin et al. [22], where the backfilled jobs can slow down the highest priority job by at most the average waiting time. Carastan-Santos et

al. [7] compare the default First-Come-First-Serve (FCFS) backfilling policy with other implementations. Policies yielding the greatest results are Shortest Job First (SJF) and Smallest Area First, where area is expressed as the product of job time and requested resource amount. Results show that these policies decrease wait times and relative slowdowns compared to EASY-FCFS. Yuan et al. [67] handle the problem that is a part of EASY backfilling, where backfilled tasks may interfere with higher priority tasks. This is handled by preempting the backfilled tasks, such that strict fairness defined in the paper is followed. To mitigate the slowdown introduced by this policy, venture backfilling is introduced that tries to backfill tasks even if they would exceed the estimated timeslot. Gaussier et al. [20] decrease the slowdown of EASY by using runtime prediction techniques and corrective mechanisms, which replace the naive job runtime estimates provided by users. In more recent work by Murad et al. [41] that targets the cloud environment, backfilling is reframed as gap-filling of idle resources, due to the resource structure present in the cloud. The proposed solution consists of a combination of multiple scheduling policies and a shortest gap-filling approach, which has been shown to decrease makespans and tardiness of jobs.

Other heuristic approaches are proposed to optimize different performance measurements of scheduling environments. Grandl et al.[24] use a fairness knob that indicates the portion of jobs that will be scheduled based on fairness principles, and rest being scheduled for optimizing target metrics such as makespan. Hu et al. [27] use the least attained service strategy with multilevel queues to enable shortest job first scheduling without knowing the runtimes of jobs. This is done by servicing jobs for a defined period of time, and moving them down to a lower priority queue if they fail to finish within the time bounds. This happens for multiple levels, where each subsequent level receives fewer resources to complete its jobs.

Besides heuristic approaches, multiple papers propose meta-heuristic algorithms to find a fair schedule based on the provided objective functions and constraints. Zhao [70] uses two-level scheduling, where the first level is used to distribute shares amongst users fairly, while the second level is used to have task fairness and efficiency of resource utilization using a bottleneck queue optimization model, which is expressed in Linear Programming. Shafiee and Ghaderi [50] define max-min fairness scheduling as the maximization of the minimum value of utility functions of each job, which is solved with Integer Programming. Barika et al. [5] use a two-phase scheduler, that uses a genetic algorithm to attempt to find a global optimum, and once a velocity change is detected in the system, switches to a greedy scheduling algorithm that can quickly scale the resources to avoid performance loss. Chen et al. [14] use a bipartite graph to represent tasks and resources in the system and perform bipartite graph matching to find an appropriate allocation.

4.2.5. Takeaways

The current implementations of fair scheduler alternatives for batch processing frameworks have shown great results, but they only address fairness across jobs rather than amongst users. Our proposed scheduler's main objective is to ensure user-job fairness, which currently is not described by any paper we surveyed. The only scheduler that can be simply converted for a multi-user, multi-job environment is proposed by W. Chen et al. [13], where each user could be isolated per application, allowing fair distribution of resources across users. However, this creates limitations on user activity, where all of their jobs must be contained within a single Spark application.

Another core issue of available batch processing schedulers is the lack of good solutions for avoiding long-running tasks from stealing resources from high-priority tasks. Pastorelli et al. address the issue by introducing preemption scheduling. However, not all systems support this functionality, and it has already been shown to reduce the throughput of the system due to wasted resource time [44]. This is partially addressed by W. Chen et al. by improving preemption in Big Data frameworks, but the solution still incurs a significant overhead [13].

Some works increase the response time of jobs by increasing job throughput [16, 8] or optimizing task allocation for task locality [12]. While showing great overall system performance, these solutions lack the enforcement of a strict fairness policy. However, we believe that these solutions are complementary and can be integrated with our proposed scheduler to further improve the system performance.

While the scheduling happens in the context of Spark, the overall context of how jobs fit within a

workflow can be used to better schedule tasks to minimize the makespan of the underlying workflow. We believe workflow scheduling can be added on top of the Spark scheduler to improve the response times or maintain fairness more effectively. Solutions similar to HEFT [57, 18] can be used to determine priority amongst jobs within the same workflow. Additionally, the broader context of workflow interaction can also be involved, by either determining sub-deadlines of each job [28, 47], or ensuring the slowdowns do not exceed the expected amount [11].

Backfilling approaches for job scheduling are useful for keeping high system utilization and allowing smaller jobs to run while larger jobs are still executing. By examining current Apache Spark scheduling, it can be observed that backfilling is already implicitly implemented in the scheduler. Spark schedulers are work-conserving and will always try to assign tasks to executors if resources are available. Since every task in Apache Spark requires only a single executor to run on, if the highest priority job has already scheduled all its tasks, the next job with the highest priority will be allowed to schedule its tasks. This suggests that to enable effective backfilling in Spark, we do not need to create a separate policy for backfilling, and only modify the general scheduling policy.

When examining fairness, many important findings have been highlighted in previous work. Quang et al. [46] compare the max-min fairness scheduling strategy to SJF. Results show that for multi-user environments, SJF can have much lower average response time for users scheduling small jobs while only slightly increasing the response times for users running long jobs. This is attributed to the fact that small jobs can finish sooner when utilizing all resources, rather than the fair share, and long-running jobs gain more resources to execute on, since it does not need to be shared with already completed short jobs. Two separate works by Rezaeian et al. [47] and Grandl et al. [24] highlight that fairness and job completion times are competing metrics, where both cannot be maximized simultaneously.

4.3. Performance prediction

Since our selected scheduling algorithm requires runtime estimates to produce a good schedule, we perform an ad-hoc exploration of available literature in performance prediction. Besides only examining literature related to batch processing job runtime estimation, we additionally look at some methods used for estimating SQL query prediction, since Spark operations are exact or similar to SQL query operations. Ad-hoc searching was performed using online search engines, namely, Google Scholar and Web of Science. The keywords that were included in the search queries with a mix of logic operators were: "performance", "runtime", "prediction", "estimation", "Apache Spark", and "SQL".

In the literature, 3 main approaches have been utilized to estimate job runtimes: analytics, machine learning (ML), and simulation.

4.3.1. Analytical methods

Analytical solutions rely on examining the metadata or historic runs of respective queries or jobs to derive the runtime estimates. While some papers have shown that standalone analytical methods can have competitive prediction accuracies [66], these techniques tend to be combined with either simulation [26] or ML [25, 49] methods.

Wu et al. [66] express a runtime model similar to the PostgreSQL cost model. It tries to find the cost units of operations by performing calibration queries that collect runtimes, and then use a sophisticated analytical model to extract the number of executed operations from active queries in the system. To account for interference, the analytical model accounts for concurrency by progressively calculating finish times and their effects on concurrent queries. While the predictions are accurate, due to the analytical models used in the system, the non-linear calculations can consume a significant amount of time to find a solution.

Gulino et al. [25] model their jobs within their Spark-based application into a DAG of tasks. Each of the tasks in the DAG has a pre-trained model that estimates the runtime of the individual task based on the input profiles and arguments. These profiles and arguments can be obtained by analytically examining the initial input size and operations performed in the submitted job. Finally, the job execution time is estimated by summing the runtimes of individual tasks.

Al-Sayeh and Sattler [49] use metadata to derive RDD sizes for stages, and historic runtimes of tasks to

create runtime models to estimate the runtime. They show that their predictions reach 94% accuracy compared to the real runtimes. However, the approach presented does not work on live data, and trains these models offline.

Herodotou et al. [26] propose a big data analytics system for self-tuning of queries. Within this system, job profiles are collected using instrumentation, and the following job profiles are estimated using these profiles and simulating the execution on the expected resources. However, increasing the prediction accuracy consequently increases the profiling overhead.

4.3.2. ML methods

ML approaches use different machine learning algorithms to train the models offline and then infer the execution time on live data. ML solutions have shown high accuracies [49, 38, 53, 34] and fast inference times [53, 34], but require substantial training time for more complex models [38, 53]. Each ML algorithm uses different techniques to perform regression, which can create an advantage for certain scenarios and queries [39], requiring domain experience or experimentation to determine the most suitable model.

Marcus and Papaemmanouli [38] construct a Deep Neural Network (DNN) using SQL operator-level neural units, which help to more accurately propagate features to subsequent neural nodes. This implementation has shown higher accuracy than a similar analytical implementation made by Wu et al. [66], but requires much larger training and inference time. Sun and Li [53] propose a similar DNN solution, but distribute their estimation model into 3 layers. To properly propagate information between sub-plans of the query, the representation layer recursively applies a shared neural model across the query plan tree, which creates a network of neural networks. Li et al. [34] have a similar structure to previous DNN implementations, but instead use regression trees to estimate runtime profiles of operators, and use scaling functions to fit the resource usage curves with fixed functional forms. Using regression trees has a significant benefit of providing fast training times and extremely quick inference times.

4.3.3. Simulation-based methods

Simulation-based methods run a job with a smaller load or in a simpler environment to estimate the runtime. This adds a constant or relative delay for each of the estimated jobs [63, 64, 45, 58]. A significant number of papers do not address the computational overhead induced by their algorithms [49, 63, 64], making it hard to estimate their viability for online scheduling.

Wang et al. [63] run a Spark job with only a fraction of the real data and resources to collect runtime data, which then is used to estimate the runtime of the full job. In a later work [64], Wang et al. account for interference amongst jobs by performing the same idea, but simulating the active jobs running concurrently to obtain the interference coefficients. These solutions present a substantial overhead in an online setting.

Popescu et al. [45] propose a methodology for estimating runtimes of iterative algorithms. It uses both historic runs and sample runs to capture the number of iterations and important features, which is then extrapolated to obtain the runtime of the incoming computation. Similarly, Venkataraman et al. [58] create a system that estimates the runtime by running smaller samples of the original job. Using the features and outputs from these runs, it fits a mathematical model that can be extrapolated to predict the total runtime on larger inputs.

4.3.4. Takeaways

Analytical approaches provide a good way of composing runtime estimation into smaller units, such as operations [66], stages [49], or tasks [25], and then estimating the runtime of these individual units to calculate the total runtime. However, tend to use more sophisticated methods such as machine learning or simulation to model the estimation of the units. The most applicable approach to our scenario is proposed by Wu et al. [66], since the majority of the actions performed in Spark can be mapped to SQL operators, allowing us to work with low-level information.

ML estimation methods show high accuracies [49, 38, 53, 34], but may not be suitable for real-time estimation due to long training times and significant estimation time. Additionally, machine learning

requires much more domain knowledge to implement effectively [39]. We find the most fitting approach to be presented by Li et al. [34] by using regression trees for estimating query runtime, since it has shown minimal training times and negligible inference times, which can work for a real-time scheduling system.

Lastly, we examine the simulation approaches, which seem to be the least applicable approach for our scenario. Due to its necessity of job simulation, it always incurs a significant runtime inference overhead [63, 64, 45, 58], which cannot be avoided. However, ideas presented by Venkataraman et al. [58] can be used with historic data instead of simulated runs, to extrapolate or infer the runtime.

Unfortunately, there can be a significant runtime skew for jobs even if they are recurring due to factors caused by resource sharing [27] or straggler tasks [49], which significantly impacts the variance of estimates. We currently do not see any solution that mitigates this in the available literature.

5

UWFQ Design

In this chapter, we present a scheduling system that implements both dynamic partitioning and response time-efficient user-job fair scheduling. We first give an overview of the proposed scheduling system. Then we propose a new scheduling algorithm, the User Weighted Fair Queuing (UWFQ) scheduler, which aims to minimize job response times while ensuring bounded user-job fairness (UJF) among multiple users. Finally, we explain the necessity of considering job runtimes while partitioning and the implementation of the dynamic partitioning component.

5.1. Overview

A simple overview of job scheduling can be seen in Figure 5.1. When a user schedules a job, it is first partitioned into smaller slices of data using a dynamic partitioning algorithm. This is done to reduce the skews and long executor reservation times caused by large tasks, which can negatively impacts the performance metrics.

Once the job is partitioned, it is forwarded to the fair scheduler, to determine its priority. The priority of each job is determined by UWFQ, which estimates the finish time of each job in a user-job fair scheduling system and assigns 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.

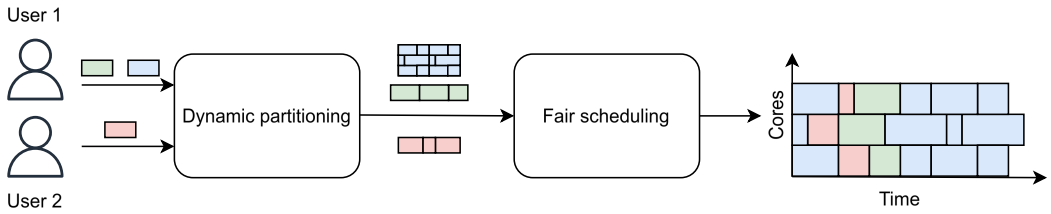


Figure 5.1: Overall system outline. In this scenario, user 1 first schedules a long-running job, which is followed by user 2's short job, and then another smaller job by user 1 again. It can be observed that user 2 and the last job do not endure a long delay as a result of effective partitioning.

5.2. Scheduling

In this section, we present the UWFQ scheduling algorithm, which uses the concepts of virtual time to simulate a virtual user-job fair scheduler more efficiently, and uses the order of completion times under this virtual scheduler to assign priorities to jobs.

5.2.1. User-job fairness scheduling with virtual time

UWFQ is heavily inspired by Cluster Fair Queueing (CFQ) [10] and WFQ [43]. Both of these algorithms use virtual time to calculate job finish times in the virtual GPS scheduler, which are then used as

deadlines for scheduling. Virtual GPS scheduler does not run real jobs, but uses job runtime estimates for advancing the virtual scheduler, which is then used for deadline calculations. While virtual time can accurately represent a virtual GPS scheduler, the fairness we are trying to achieve is user-job fairness (UJF), which distributes resources differently than GPS. To implement virtual UJF using virtual time, modifications have to be applied to keep the schedules consistent.

The main benefit of virtual time is that each job's finish time has to be computed only once. This allows virtual time implementations in WFQ and CFQ to achieve $O(log_2(n))$ worst case runtime when adding a job to a deadline order list. This deadline ordering reflects the finish times in a GPS system shown in Figure 5.2 and remains consistent as the virtual time progresses. When a new job arrives, it does not alter the finish times of other jobs, and can correctly infer its finish time by using the current virtual time and its estimated runtime. We visualize how virtual time progresses and adapts to jobs leaving and entering the system in Figure 5.3. Note that we are looking at idealistic cases, where we assume resources are fluid and can be of any valid fraction.

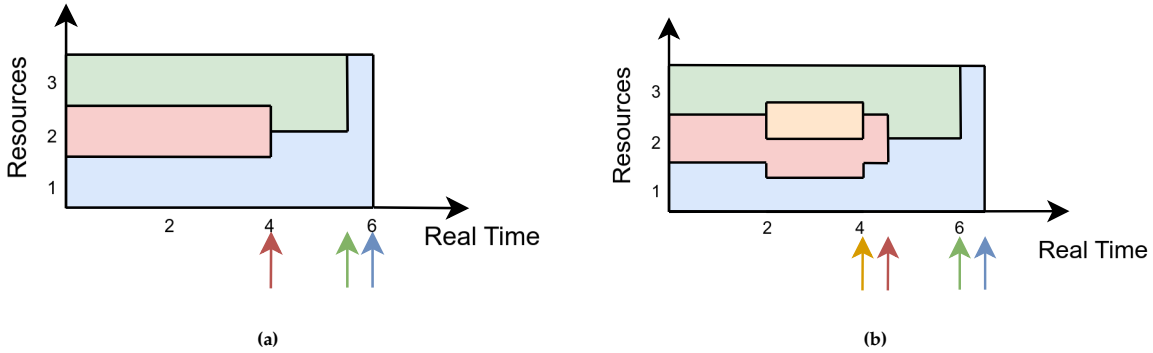


Figure 5.2: Scheduling under GPS. In (a), there are 3 different-sized jobs, with their finish times indicated by arrows in their respective color. In (b), a new small job enters the system at time 2 and receives equal resources from existing jobs, shifting the finish times of the other active jobs slightly forward.

Virtual time is used to express priorities, but does not necessarily represent the time the job would finish under WFQ or CFQ. These algorithms give the entire resource pool to the highest priority task, which may therefore finish faster than it would under GPS. While this may allow lower priority jobs to complete quicker than higher priority jobs that arrive slightly later to the system, it still preserves the deadlines that are enforced by GPS. We show this in Figure 5.4, where the orange job completes after the red job, even with a closer deadline, however, all deadlines are still kept, ensuring GPS bounded response times.

Virtual time produces an order equivalent to GPS if jobs proportionally distribute their resources among each other. However, under user-job fairness, the shares are not proportionally distributed among all jobs, but only among a subset of jobs that are under the same user. This can change the job completion time order as new jobs arrive, which can be seen in Figure 5.5, forcing jobs to adjust their deadlines due to disproportionate resource loss or gain. This is not supported by the original virtual time implementation, hence it cannot be directly used to simulate a virtual UJF scheduler. While a virtual UJF scheduler could be implemented using real time instead of virtual time, we would still want to keep some of the beneficial properties of virtual time.

We solve this by introducing multi-layered virtual time, specifically a 2-layer virtual time, which first produces the correct order of jobs for each user using local virtual time, and then uses global virtual time to assign scheduling deadlines across all users. For each user, local virtual time ensures that their jobs are ordered by their completion times, while global virtual time enables fair comparison of jobs across users, preserving the order that was established for jobs within each user.

When a new job arrives for a user, their jobs would have to shift their deadlines based on how early the arriving job has to be scheduled, showcased in Figure 5.6. While this does increase the worst runtime complexity to $O(n)$, the complexity is isolated per user, and job deadlines do not need to be updated as jobs leave the system, reducing the bookkeeping time.

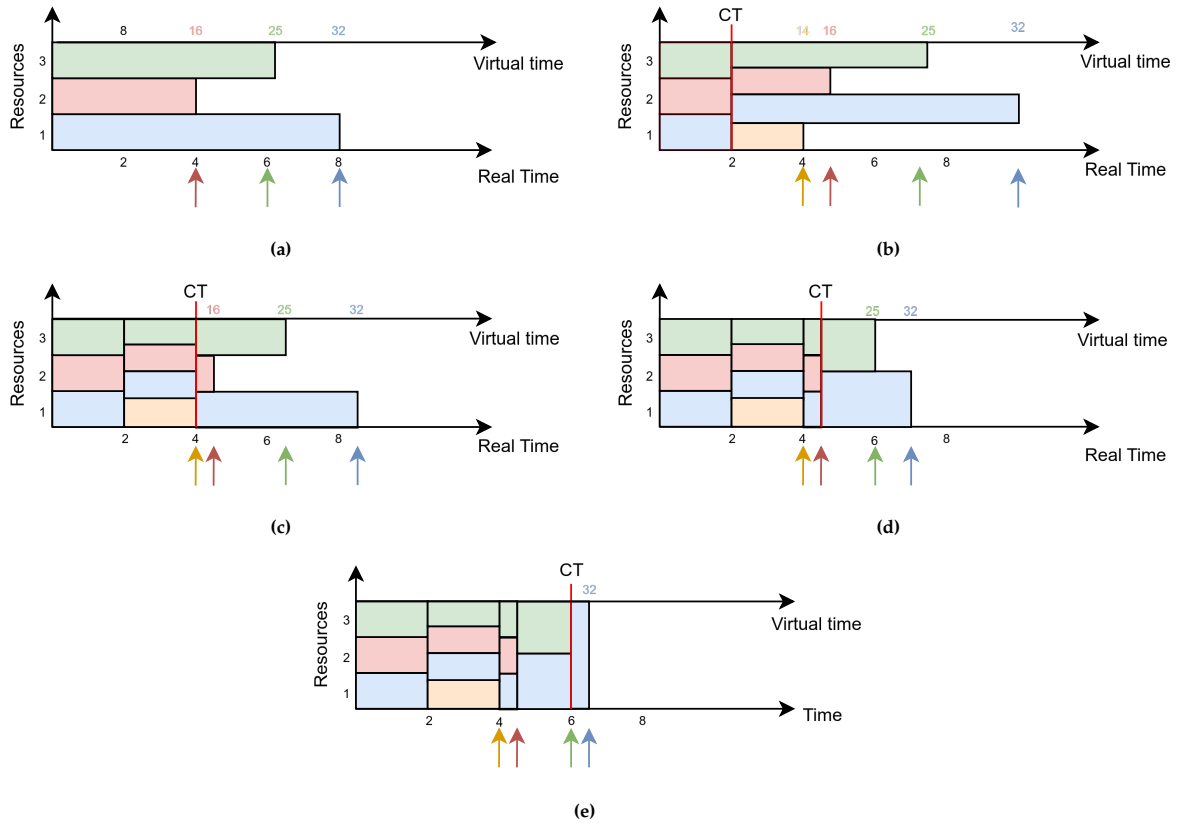


Figure 5.3: Scheduling from the virtual time computation perspective. As the current time (CT) progresses, we can see that virtual time expands as a job enters the system in (b) or contracts as jobs leave the system in (c)-(e). Note that the completion order always reflects the order we see in Figure 5.2.

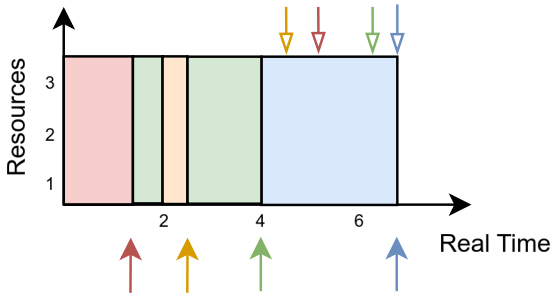


Figure 5.4: Scheduling of jobs in the real system, where all resources are given to the job that finishes the soonest in GPS. The orange job enters the system at time 2, hence it is serviced later than the red job. The arrows indicate the finish time of jobs from Figure 5.2 (b).

5.2.2. The User Weighted Fair Queuing principles

UWFQ reuses principles originally proposed by Parekh et al. [43] to design WFQ. Specifically, it uses virtual time to efficiently compute job finish times under 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 defined in this project.

We use virtual time in 2 layers: user (local) virtual time and global virtual time. User virtual time is assigned to each user and functions as regular virtual time, and is used to obtain 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 that are used to establish priority among all active jobs in the system.

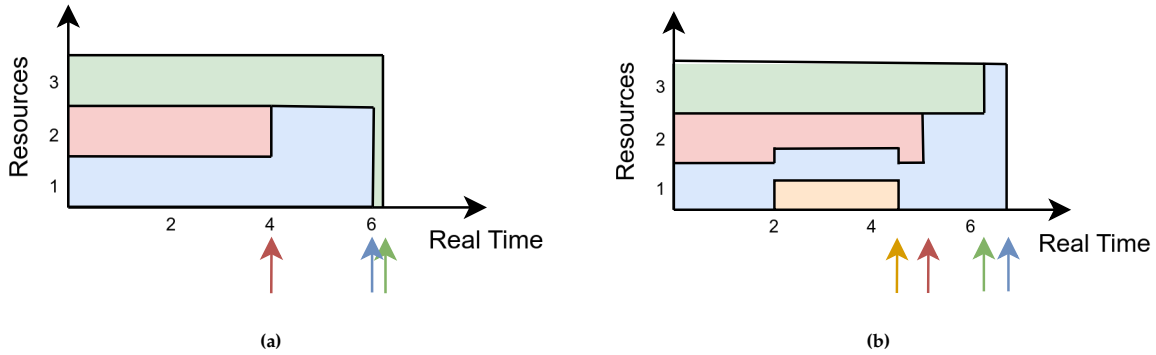


Figure 5.5: Job finish time order changing as a new job enters a system. In this example, one user is only running a green job with 1/3 of the resources, while another user is running the rest of the jobs with 2/3 of the resources. In (a), the blue job finishes earlier than green, but in (b), the blue job is delayed, since a new job arrived that only affects one of the users.

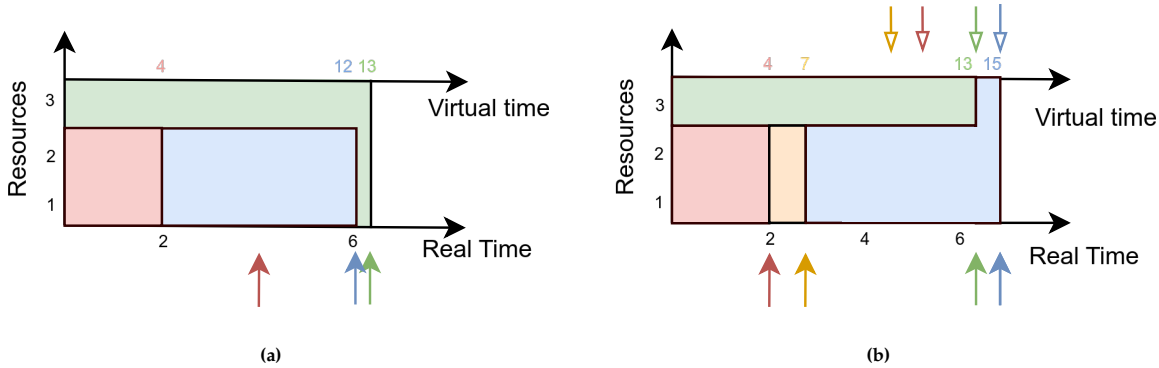


Figure 5.6: Multi-layered virtual time updating as a new job arrives. In (a) jobs are already ordered for each user, and as a job arrives in (b), it has to be just correctly inserted in this schedule for its corresponding user. The arrows indicate the finish time of jobs from Figure 5.5 respectively, and it can be seen that the jobs are correctly bounded by these deadlines.

Once a user enters a system, they receive a global virtual start time, which indicates the virtual time this user started receiving their fair share of resources. User can only join a system if they have an active job, and leave the system as soon as all their jobs have finished in global virtual time. Once a job passes its global virtual deadline, the corresponding user removes it from their active job list and progresses the global virtual start time forward to this virtual deadline. Global virtual time represents the amount of resource time each user has received so far, so progressing each user's global start time forward is necessary to track each user's resource usage over time.

Incoming jobs for a user are ordered based on their user virtual deadlines that are calculated by adding the user's current local virtual time to the job's expected runtime. Once the order of jobs within a user is established, the first global virtual deadline is assigned to the earliest finishing user job, which is calculated by adding its runtime and the user's global start time. This global virtual deadline effectively represents the completion time of a job, which in turn serves as the start time for the next job in sequence. The global deadline for the subsequent job can then be calculated in the same way by taking the previous job's deadline as its starting point. This process is repeated in a chain until all user jobs have received a global virtual deadline.

By assigning global virtual deadlines to all jobs, we now have a global order that can be used for scheduling. This allows us to schedule jobs such that they may complete much quicker than they would when scheduled under user-job fairness, while still being bounded by user-job fairness.

We show this conversion between the user-job fairness (UJF), 2-level virtual time (2-LV), and UWFQ schedules in Figure 5.7, where it can be seen that job allocation order can be significantly changed by 2-LV and UWFQ to decrease the response times, while still maintaining the deadlines that are established by UJF in Figure 5.7(a).

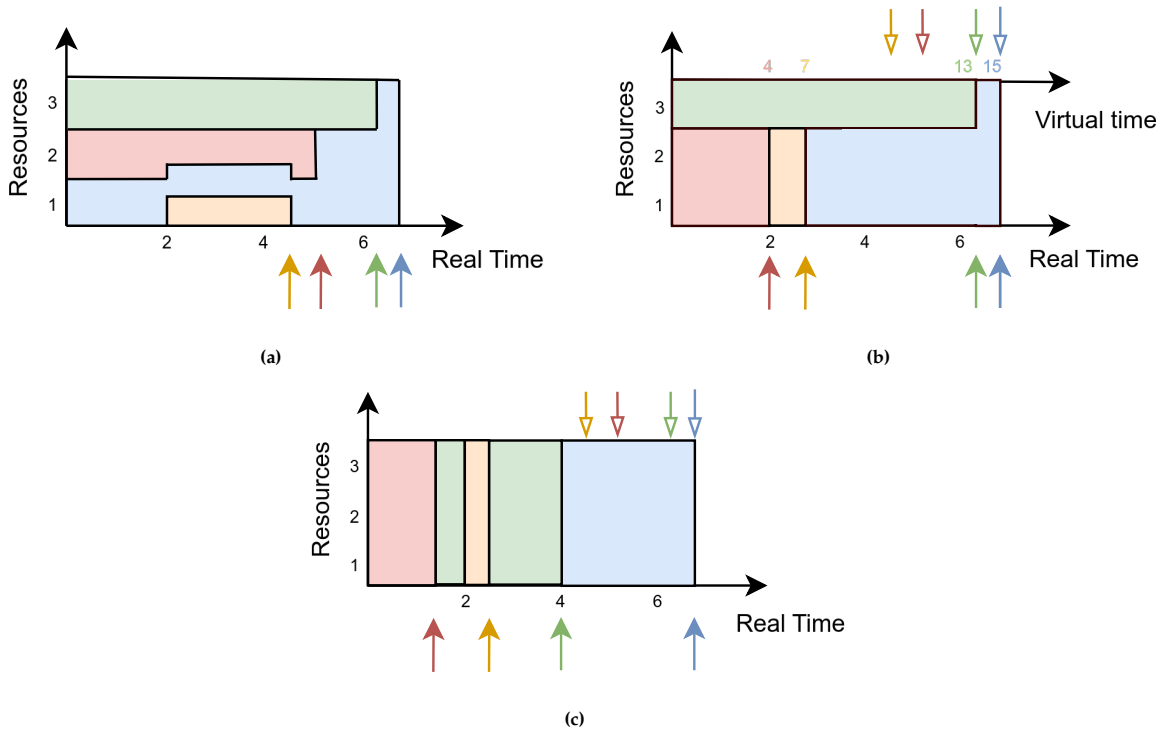


Figure 5.7: Differences between (a) user-fair schedule (b) 2-level virtual time and (c) UWFQ. The deadline arrows are established by (a), and it can be seen that they are abided by both (b) and (c)

5.2.3. User Weighted Fair Queuing algorithm

We show deadline assignment of jobs in Algorithm 1. To create a schedule, we order the jobs based on their assigned deadlines. To present the algorithm more clearly, we first define the necessary notations used in the algorithm, and divide it into phases, providing a detailed explanation of each phase.

Definitions

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.

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 runtime L_i and user virtual arrival time $V^k_{arrival}$, which is measured from current virtual global time V_{global} on user's arrival.

Finally, both global virtual time V_{global} and user virtual time V_{user} must be up kept. This is achieved by tracking active resource shares amongst users R_k and jobs within a user R^k_{job}, and using these shares to advance virtual times respectively.

Phase 1

In phase 1, we update the system that is keeping track of users and virtual time. We explain how to update virtual time in Subsection 5.2.4. If the user U_k submitting a job is already a part of the system, no further update is needed. If the user is new to the system or resumes activity, we insert it in the user set S_{users} along with its global virtual arrival time $V^k_{arrival}$. This arrival time is necessary for assigning the global virtual deadlines of user jobs.

Phase 2

In phase 2, we calculate the user deadline D^i_{user} for the submitted job i and insert it in its correct position within the sorted user job set. To calculate the job's user deadline, we add together the user's current

Table 5.1: Notation definitions

Notation	Explanation
$_i$	Job i
$_k$	User k
U	User entity
J	Job entity
S	Set of items
T	Absolute time
V	Virtual time
D	Virtual deadline
L	Job slot-time (sum of task runtimes)
R	System resources

virtual time V^k_{user} with the duration of the job L_i. This deadline represents the virtual time at which the job would finish for this user and can be compared with other active user jobs in the user job set S^i_{jobs}. We use this property to insert the job J_i in the correct position in the set S^k_{jobs}.

Phase 3

In the final phase, we update the global virtual deadlines of all users' jobs, which are then used by the system to create the schedule.

First, we must set the correct global virtual deadline for the earliest finishing user job J_{first}. We can find the earliest finishing job for the user U_k by identifying the job with the lowest user deadline D_{user}. We then calculate the global deadline D^{first}_{global} by adding the duration of the job L_{first} to the user's arrival time $V^k_{arrival}$. This deadline represents the global virtual time the job would finish at in the system and can be compared with all the other active jobs. The user's arrival time is used for the first job, since it represents the time the user started acquiring resources from the system. Assuming the user is work-conserving, this is the same time the user would schedule their first job, which corresponds to J_{first}.

Second, we use the global virtual deadline D^{first}_{global} of the first job to calculate and update the rest of the user's job deadlines. Since each user schedules only one job at a time, the job global virtual deadlines will be chained one after the other. We achieve this by looping over the rest of the remaining jobs in the order of their user deadline D_{user} and assign the global deadline D_{global} by adding the previous deadline $D^{previous}_{global}$ and the duration of the job L. In this context, the global deadline can be interpreted to represent the global virtual finish time of each job, and by assuming work-conserving scheduling, this finish time would correspond to the following job's start time. We note that this procedure is very similar to EEVDF's [52] deadline assignment to requests, where the request deadline is the sum of the previous request deadline and its weighted runtime.

Minor performance improvements

In Algorithm 1, we update all deadlines of user jobs, without considering if they are affected by the incoming job. Suppose the user deadline for the incoming job is after all the current running jobs; only the deadline of the incoming job needs to be updated, since all the currently active jobs are already correctly sorted. While this would empirically reduce the computation time of the algorithm, it does not impact the runtime complexity, hence it is left out from the pseudocode for simplicity reasons.

Algorithm 1 Job deadline assignment under User Weighted Fair Queuing

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;

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

7: // Phase 2: calculate user deadline and insert job J_j into set of user jobs
8: $V^k_{user} \leftarrow$ getUserVirtualTime(U_k)
9: $D^i_{user} \leftarrow V^k_{user} + L_i$
10: $S^k_{jobs} \leftarrow$ getUserJobs(U_k) ▷ S_{jobs} is a sorted set of tuples on virtual user deadlines D_{user}
11: $S^k_{jobs} \leftarrow S^k_{jobs} \cup (J_i, L_i, D^i_{user})$

12: // Phase 3: update user job global virtual deadlines
13: $(J_{first}, L_{first}) \leftarrow$ getEarliestUserDeadline(S^k_{jobs})
14: $V^k_{arrival} \leftarrow$ getUserVirtualArrivalTime(U_i)
15: $D^{first}_{global} \leftarrow V^k_{arrival} + L_{first}$
16: setJobDeadline($J_{first}, D^{first}_{global}$)

17: $D^{previous}_{global} \leftarrow D^{first}_{global}$
18: **for** each J_a in S^k_{jobs} sorted by D^a_{user} and $J_a \neq J_{first}$ **do**
19: $D^a_{global} \leftarrow D^{previous}_{global} + L_a$
20: setJobDeadline(J_a, D^a_{global})
21: $D^{previous}_{global} \leftarrow D^a_{global}$
22: **end for**

5.2.4. Updating virtual time

In 2-level virtual time, we have to update both the global time and virtual time for each user. We propose an algorithm for updating the global virtual time in Algorithm 2 and the user virtual time in Algorithm 3. We split the updating into multiple functions and explain each function.

updateVirtualTime

To update virtual time, we must iterate over all existing users in the order of their completion time. This is necessary to correctly advance the global virtual time, since it progresses with a rate that is proportional to the number of active users. We first get the user share that each user receives R_{user}, and use it to compute user's real 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 stay consistent and progress all users equally.

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

1: **function** UPDATEVIRTUALTIME($T_{current}$)
2: **for** each $(U_k, \_)$ in S_{users} sorted by latest job deadline D_{global} **do**
3: $R_{user} \leftarrow \frac{R}{|S_{users}|}$
4: $T^k_{finish} \leftarrow$ GETUSERFINISHTIME(U_k, R_{user})
5: **if** $T^k_{finish} > T_{current}$ **then**
6: **break**
7: **end if**
8: $S_{users} \leftarrow S_{users} \setminus (U_k, \_)$
9: PROGRESSVIRTUALTIME(T^k_{finish}, R_{user})
10: **end for**
11: $R_{user} \leftarrow \frac{R}{|S_{users}|}$
12: PROGRESSVIRTUALTIME($T_{current}, R_{user}$)
13: **end function**
14: **function** GETUSERFINISHTIME(U, R_{user})
15: $D^{latest}_{global} \leftarrow$ GETLATESTDEADLINE(U)
16: $T_{spent} \leftarrow (D^{latest}_{global} - V_{global})/R_{user}$
17: $T_{finish} \leftarrow T_{previous} + T_{spent}$
18: **return** T_{finish}
19: **end function**
20: **function** PROGRESSVIRTUALTIME(T, R_{user})
21: $T_{passed} \leftarrow T - T_{previous}$
22: $V_{global} \leftarrow V_{global} + T_{passed} * R_{user}$
23: **for** each $(U_k, \_)$ in S_{users} **do**
24: UPDATEUSERVIRTUALTIME(U_k, R_{user}, T)
25: **end for**
26: $T_{previous} \leftarrow T$
27: **end function**

getUserFinishTime

We can obtain the user's finish time T_{finish} by calculating the time the last job will finish. Since all user jobs are ordered based on their user virtual deadlines, we can trivially get the latest finish 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 will end, and dividing by user share R_{user} 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 have to be updated till the current time T.

Algorithm 3 Virtual time updating for users

Globals: Previous update time $T_{previous}$

1: **function** UPDATEUSERVIRTUALTIME($U_k, R_{user}, T_{current}$)
2: $S^k_{jobs} \leftarrow$ GETUSERJOBS(U_k)
3: $T^{user}_{previous} \leftarrow T_{previous}$
4: $V^k_{user} \leftarrow$ GETUSERVIRTUALTIME(U_k)
5: **for** each (J_i, L_i, D^i_{user}) in S^k_{jobs} sorted by D_{user} **do**
6: $R_{job} \leftarrow \frac{R_{user}}{|S^k_{jobs}|}$
7: $T_{passed} \leftarrow T_{current} - T^{user}_{previous}$
8: $V^i_{user} \leftarrow V^k_{user} + (T_{passed} * R_{job})$
9: **if** $D^i_{user} > V^i_{user}$ **then**
10: **break**
11: **end if**
12: $V_{spent} \leftarrow D^i_{user} - V^k_{user}$
13: $T_{spent} \leftarrow \frac{V_{spent}}{R_{job}}$
14: $V^k_{user} \leftarrow V^k_{user} + V_{spent}$
15: $T^{user}_{previous} \leftarrow T^{user}_{previous} + T_{spent}$
16: $V^k_{arrival} \leftarrow$ GETUSERVIRTUALARRIVALTIME(U_k)
17: $V^k_{arrival} \leftarrow V^k_{arrival} + L_i$
18: $S^k_{jobs} \leftarrow S^k_{jobs} \setminus (J_i, L_i, D^i_{user})$
19: **end for**
20: **if** $|S^k_{jobs}| > 0$ **then**
21: $R_{job} \leftarrow \frac{R_{user}}{|S^k_{jobs}|}$
22: $T_{spent} \leftarrow T_{current} - T^{user}_{previous}$
23: $V^k_{user} \leftarrow V^k_{user} + (T_{spent} * R_{job})$
24: **end if**
25: **end function**

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.

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 are progressing at the correct marginal rate.

We first must 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, to correctly progress 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 if 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. But if the job has finished, we then calculate the virtual time V_{spent} that was 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 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 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. 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 job shares R_{job} to obtain the virtual time that has passed, and adding it to the user virtual time V^k_{user}.

2-level virtual time runtime complexity

Original virtual time was able to ensure $O(log(n))$ job insertion time into the schedule, where n is the amount of active jobs in the system. However, virtual time itself has to be updated before performing deadline calculations to accurately represent job deadlines, which can take at most $O(n)$ time to iterate overall all jobs. The proposed 2-level virtual time (2-LV) increases both of these computation times due to added complexity by each layer. To insert a job in 2-LV, it takes at worst $O(n)$ time due to having to account for other user jobs to update their deadlines, since they can be shifted by the incoming job. However, only the jobs belonging to the corresponding user have to be updated, which means that in systems where users have few concurrent jobs, the practical runtime is much smaller.

The biggest runtime bottleneck comes from updating 2-LV, which can at worst be $O(n^2)$. This is caused by updateVirtualTime($T_{current}$), which iterates over all users u in the order of their completion time, and if a user has finished, progresses virtual time forward for the remaining users. Progressing virtual time for every user can at worst take $O(n)$ time, since it can at most iterate over all jobs in the system, resulting in $O(n * u)$ worst case time complexity. Assuming the worst case conditions where each user has a single job, user amount u equals the job active amount n, simplifying the worst case runtime to $O(n^2)$.

Note that all lists in the system are ordered during insertion, which means that every time an element is inserted into an order list, it takes $O(log_2(n))$ worst-case time complexity. Since the job insertion time is already bounded by $O(n)$, the runtime complexity is unaffected by upkeeping sorted lists. And due to all lists being sorted, there is no sorting penalty incurred when iterating over them during the virtual time updating phase.

While 2-LV introduces a higher worst-case runtime complexity, it still significantly outperforms the alternative of using real time for calculating job finish times in UJF. There are two general implementations of simulating UJF in real time: a live real-time UJF scheduler, or a lookahead real-time UJF scheduler.

A live real-time scheduler would keep track of all jobs constantly and update the weights as jobs enter or leave the system. While it would only take O(n) time to update all job runtimes, the system would have to constantly keep track of jobs leaving the system, which creates a practical bottleneck and can

cause significant delays if many jobs leave simultaneously. This is completely avoided by using 2-LV, since it only has to track job arrival times.

A lookahead real-time scheduler would avoid the necessity of tracking leaving jobs by already accounting for them while calculating real job finish times. This can be done by first finding a job that has the earliest finish time in the current schedule, assigning its deadline as that finish time, progressing the virtual system forward until the finish time, and recalculating the completion times of all other jobs. This process is repeated until every single job has its deadline assigned. Since recalculating the deadlines may change the order of jobs, the entire job list must be iterated to find the next earliest job, creating the expected runtime of this algorithm to be $O(n^2)$.

While the worst-case runtime complexities of 2-LV and lookahead real-time UJF scheduler are the same, we argue that the expected runtime is better in 2-LV. In a system where there are significantly more active jobs than users, the expected runtime of 2-LV goes down to $O(n)$. However, in a system where the user amount is more proportional to job amount, a lookahead real-time scheduler can yield the same performance, and make it less complex to implement.

Asynchronous virtual time

The proposed implementation of 2-LV is synchronous, meaning that it is updated only once a job arrives, and does not asynchronously keep track of leaving jobs and inactive users. However, this does come at a computational cost, since the 2-LV update worst case time complexity is $O(n^2)$, where n is the number of active jobs in the system before the update. While the synchronous approach is appropriate for our target system, this may cause a significant scheduling delay in the case where there is a burst of incoming jobs with a very immediate silence period. The first job that will be scheduled after the silence period will have to update the virtual time for all the previously finished jobs, which may cause a significant delay if the number of jobs is large. For these cases, handling virtual time asynchronously would be more appropriate.

To integrate asynchronous virtual time in the proposed scheduler, the updateVirtualTime($T_{current}$) function has to be periodically called to alleviate the computational costs of calculating virtual time. To avoid race conditions, the updating of virtual time within Algorithm 1 must be ensured not to happen concurrently with periodic updating. Alternatively, updating in Algorithm 1 can be removed in the case where jobs are scheduled in time slices proportional to virtual time periodic update.

5.2.5. Soundness of UWFQ

In this section, we prove that our UWFQ algorithm correctly implements bounded user-job fairness. We borrow many structural components of the proof from CFQ [10] and WFQ [43], since our algorithm extends on some aspects of these schedulers.

Assumptions

For these proofs, it is necessary to assume an idealized system for user-job fair and 2-level virtual time schedulers, where each job can be divided into infinitesimally small and equal tasks, and all resources can execute them simultaneously and preempt them without scheduling delay. These are equivalent assumptions that are used for hypothetically implementing the GPS scheduler. We define R to represent the available resource amount, which can be infinitely divisible into shares and does not encounter any scheduling delay.

However, for UWFQ, we assume a limit in job division, which can create a skew in task runtimes, the longest task in the system being l_{max}. Additionally, tasks cannot be preempted and have to run till completion once scheduled, creating even more boundaries.

Theorems and proofs

First, we prove that jobs in 2-level virtual (2-LV) time do not finish later than they would in the user-job fair (UJF) scheduler. For this purpose, we first introduce lemmas.

Lemma 1. *Let U_k be an arbitrary user in both 2-level virtual time and user-job fairness. The share R_k this user possesses is equivalent in both schedules at any period of time.*

Proof. Take user U_k. Assume there are N_u users that finish before U_k in UJF. Since both 2-LV and UJF distribute shares equally among users and both are work-conserving, users in both schedules progress

Table 5.2: Notation definitions

Notation	Explanation
i	Job i
k	User k
U	User entity
N	Amount of items
S	Set of items
L	Job slot-time (sum of task runtimes)
R	System resources
f	Finish time in 2-level virtual time schedule
$\hat{f}$	Finish time in user-job fairness schedule
F	Finish time in UWFQ schedule
a	Arrival time
e	Full resource access time
l	Task runtime

at the same rate, meaning that all N_u users that finish before U_k in UJF will finish in the same order in 2-LV, and redistribute the same amount of resources to user shares R_k overtime. □

Lemma 2. *Let U_k be an arbitrary user in both 2-level virtual time and user-job fairness. Let f_i be the finish time of job i in 2-level virtual time, and let $\hat{f}_i$ be the finish time of job i in the user-job fair scheduler. For any arbitrary job i in both schedules, user U_k share increase or decrease does not impact the order of completion time f_i and $\hat{f}_i$.*

Proof. Take user U_k. Assume user U_k resources R_k are changing due to users joining and leaving the system. Following Lemma 1, we know that both schedules will have equivalent shares R_k across this period, hence will contribute equal resources to job execution in both schedules. We can express the finish time f_i as

$$f_i = \frac{\sum_{n=1}^{i} L_n}{\hat{R}_k} \tag{5.1}$$

where n till i are all jobs that finish before and including i, and $\hat{R}_k$ is the volatile resources assigned to user k. Additionally, we can derive the finish time for $\hat{f}_i$

$$
\begin{aligned}
\hat{f}_1 &= \frac{N_s^k}{\hat{R}_k} * L_1 \\
\hat{f}_2 &= \hat{f}_1 + \frac{N_s^k - 1}{\hat{R}_k} * (L_2 - L_1) \\
\hat{f}_3 &= \hat{f}_2 + \frac{N_s^k - 2}{\hat{R}_k} * (L_3 - L_2) \\
&\dots \\
\hat{f}_i &= \hat{f_{i-1}} + \frac{N_s^k - i + 1}{\hat{R}_k} * (L_i - L_{i-1}) \\
\hat{f}_i &= \sum_{n=1}^{i} (\frac{N_s^k - n + 1}{\hat{R}_k} * (L_n - L_{n-1})) \text{ , where } L_0 = 0
\end{aligned}
\tag{5.2}
$$

where N_s^k is the number of jobs the user had at the start of the execution. Let's assume $f_i \leq \hat{f}_i$. By substituting it with Equation 5.1 and Equation 5.2

$$
\begin{aligned}
\frac{\sum_{n=1}^{i} L_n}{\hat{R}_k} &\leq \sum_{n=1}^{i} (\frac{N_s^k - n + 1}{\hat{R}_k} * (L_n - L_{n-1})) \\
\frac{1}{\hat{R}_k} * \sum_{n=1}^{i} L_n &\leq \frac{1}{\hat{R}_k} * \sum_{n=1}^{i} ((N_s^k - n + 1) * (L_n - L_{n-1})) \\
\sum_{n=1}^{i} L_n &\leq \sum_{n=1}^{i} ((N_s^k - n + 1) * (L_n - L_{n-1}))
\end{aligned}
\tag{5.3}
$$

We get that the order of completion f_i and $\hat{f}_i$ is independent of user resources. □

Theorem 1. *Let f_i be the finish time of job i in 2-level virtual time, and let $\hat{f}_i$ be the finish time of job i in the user-job fair scheduler. For every job i in the system, we show that*

$$
f_i \leq \hat{f}_i \tag{5.4}
$$

Proof. We consider two cases that cover all jobs in the system.

Case 1: take arbitrary job i that is already present in the schedule. Since it is present in the schedule, the job is tied to the corresponding user and is ordered within their job set. We show that the job's finish time f_i will not exceed $\hat{f}_i$ under any circumstance.

First, let's consider that no new job arrives in the system. Following Lemma 2, we know that user share $\hat{R}_k$ changes over time do not affect the order of finish times. We can then substitute Equation 5.4 with Equation 5.1 and Equation 5.2

$$
\begin{aligned}
\frac{\sum_{n=1}^{i} L_n}{\hat{R}_k} &\leq \frac{N_s^k}{\hat{R}_k} * L_1 + \frac{N_s^k - 1}{\hat{R}_k} * (L_2 - L_1) + ... + \frac{N_s^k - i + 1}{\hat{R}_k} * (L_i - L_{i-1}) \\
\sum_{n=1}^{i} L_n &\leq N_s^k * L_1 + (N_s^k - 1) * (L_2 - L_1) + ... + (N_s^k - i + 1) * (L_i - L_{i-1}) \\
L_1 + L_2 + ... + L_i &\leq N_s^k * L_1 + (N_s^k - 1) * (L_2 - L_1) + ... + (N_s^k - i + 1) * (L_i - L_{i-1}) \\
L_2 + ... + L_i &\leq N_s^k * L_1 - L_1 + (N_s^k - 1) * (L_2 - L_1) + ... + (N_s^k - i + 1) * (L_i - L_{i-1}) \\
L_2 + ... + L_i &\leq (N_s^k - 1) * L_1 + (N_s^k - 1) * (L_2 - L_1) + ... + (N_s^k - i + 1) * (L_i - L_{i-1}) \\
L_2 + ... + L_i &\leq (N_s^k - 1) * L_2 + ... + (N_s^k - i + 1) * (L_i - L_{i-1}) \\
&... \\
L_i &\leq (N_s^k - i + 1) * L_i \\
0 &\leq (N_s^k - i) * L_i
\end{aligned}
\tag{5.5}
$$

This inequality holds true for all values $N_s^k \geq i$. Now let's consider the case where a job c arrives for the corresponding user. The job c will either finish before job i or after. If job c finishes after i, it is obvious that $f_i \leq \hat{f}_i$, since f_i will not encounter any delay. If job c finishes before i:

$$
\begin{aligned}
&L_1 + L_2 + ... + L_c + ... + L_i \leq \\
&\leq N_s^k * L_1 + (N_s^k - 1) * (L_2 - L_1) + ... + (N_s^k - c + 1) * (L_c - L_{c-1}) + ... + (N_s^k - i + 1) * (L_i - L_{i-1}) \\
&... \\
&L_c + ... + L_i \leq (N_s^k - c + 1) * L_c + ... + (N_s^k - i + 1) * (L_i - L_{i-1}) \\
&L_i \leq (N_s^k - i + 1) * L_i \\
&0 \leq (N_s^k - i) * L_i
\end{aligned}
\tag{5.6}
$$

Case 2: take arbitrary job i that is arriving in the schedule. If the job belongs to a user that does not exist, the job will create a user and assign itself to it. In that case, since both 2-LV and UJF are work-conserving, the job will finish at the same time for both schedules. If the user already exists, the job has to find its order in the existing job set of the corresponding user. Once the job is ordered, *case 1* can be applied to show that $f_i \leq \hat{f}_i$.

□

Second, we prove that UWFQ finishes jobs in bounded 2-level virtual time.

Theorem 2. *Let f_i be the finish time of job i in 2-level virtual time, and let F_i be the finish time of job i in the UWFQ scheduler. For every job i in the system, we show that*

$$
F_i - f_i \leq \frac{L_{max}}{R} + 2 * l_{max} \tag{5.7}
$$

Proof. Let a_i be the arrival time of job i, and job i belonging to user job set S_{jobs}^k. In UWFQ, since there may be other jobs running at a_i, job i tasks may be scheduled later due to priority or task runtime skews. Let e_i represent the time when job i has full access to resources R until completion of all its tasks. Since no other job can interfere with job i during period e_i, its runtime is bounded by the longest running task, i.e.

$$
F_i \leq e_i + l_{max}^i \tag{5.8}
$$

By analyzing the period e_i in all cases, we can determine the worst-case response time F_i for job i in UWFQ. We split this into two major cases:

Case 1: The job i immediately receives all resources R, hence $a_i = e_i$. In 2-LV, the finish time of job i would be

$$f_i \leq a_i + \frac{L_i}{R_k} \tag{5.9}$$

By subtracting Equation 5.9 from Equation 5.8

$$\begin{aligned} F_i - f_i &\leq e_i + l^i_{max} - a_i - \frac{L_i}{R_k} \\ F_i - f_i &\leq a_i + l^i_{max} - a_i - \frac{L_i}{R_k} \\ F_i - f_i &\leq l^i_{max} - \frac{L_i}{R_k} \leq l_{max} \end{aligned} \tag{5.10}$$

Case 2: The job i is delayed to receive all resources R, hence $a_i < e_i$. There are two types of jobs that delay e_i: jobs that have higher priority than job i, and jobs that have lower priority but started before job i. Let job index indicate the order of execution in UWFQ, and let set S_{high} be the set of all jobs that execute before i with higher priority, i.e. $S_{high} = \{\, d \mid d < i \text{ and } f_d < f_i \,\}$, and let set S_{low} be the set of jobs that execute before job i with lower priority, i.e., $S_{low} = \{\, g \mid g < j \text{ and } f_g > f_i \,\}$.

Case 2-1: First, let's examine the case where there are only higher priority jobs from set S_{high} running before job i. In that case, 2-LV would finish

$$f_i \leq a_i + \frac{L_i}{R_k} + \sum_{n \in S^k_{jobs} \cap S_{low}} \frac{L_n}{R_k} \tag{5.11}$$

For UWFQ, the finish time of job i also depends on other user jobs that may run before it. We assume a worst-case scenario, where for every user there is a job that finishes just before f_i. This means that all other users will be fully utilizing their share of the resource before job i can be scheduled in UWFQ. Since both UWFQ and 2-LV are work-conserving, user U_k will have the same amount of resources time to execute on in both UWFQ and 2-LV, so the job i will be able to receive all resources at the same time it would in 2-LV, i.e.

$$e_i \leq a_i + \sum_{n \in S^k_{jobs} \cap S_{low}} \frac{L_n}{R_k} + C \tag{5.12}$$

where C is a delay caused by skews. To account for skews, we can assume a worst-case scenario, where previously running jobs have accumulated a long-running task on all resource instances, creating a worst-case delay of $C = l_{max}$. We can now substitute Equation 5.8 with Equation 5.12 and substract Equation 5.11

$$\begin{aligned} F_i - f_i &\leq a_i + \sum_{n \in S^k_{jobs} \cap S_{low}} \frac{L_n}{R_k} + C + l^i_{max} - \\ &-(a_i + \frac{L_i}{R_k} + \sum_{n \in S^k_{jobs} \cap S_{low}} \frac{L_n}{R_k}) \\ F_i - f_i &\leq C + l^i_{max} - \frac{L_i}{R_k} \\ F_i - f_i &\leq l_{max} + l^i_{max} - \frac{L_i}{R_k} \leq 2 * l_{max} \end{aligned} \tag{5.13}$$

Case 2-2: Now let's examine the case where the set of both high priority jobs S_{high} and low priority jobs S_{low} runs before job i.

For 2-LV, lower priority jobs cause no delay. The only case a job from S_{low} can run is in the period where all jobs in S_{high} have finished and job i has not arrived yet. As soon as job i arrives, it will preempt the lower priority job, resulting in the same finish time as in Equation 5.9.

For UWFQ, a job from S_{low} can also only run in the period where all jobs in S_{high} have finished and job i has not arrived yet. However, since there is no preemption, in the worst case scenario where job i arrives just before a job from S_{low} is scheduled, it will be delayed by the entire runtime of the low priority job, i.e.

$$e_i \leq a_i + \frac{L_{max}}{R} + l_{max} \tag{5.14}$$

Substituting Equation 5.8 with Equation 5.14 and substracting Equation 5.9

$$\begin{aligned} F_i - f_i &\leq a_i + \frac{L_{max}}{R} + l_{max} + l^i_{max} - (a_i + \frac{L_i}{R_k}) \\ F_i - f_i &\leq \frac{L_{max}}{R} + l_{max} + l^i_{max} - \frac{L_i}{R_k} \leq \frac{L_{max}}{R} + 2 * l_{max} \end{aligned} \tag{5.15}$$

□

Lastly, we establish a corollary that UWFQ is bounded by user-job fairness.
Corollary 1. *Since jobs in 2-level virtual time are bounded by user-job fairness, and UWFQ is bounded by 2-level virtual time, we can express that UWFQ is bounded by user-job fairness.*

5.3. Dynamic partitioning

Partitioning is a parallelization method that splits input data across multiple execution units, enabling faster job execution by utilizing more resources. Apache Spark already has a built-in partitioning strategy that tries to evenly distribute input data across all available cores. However, we argue that partitioning based solely on resource amount and data size can degrade system performance in two key ways: long-running tasks delaying execution of more deserving jobs, and creating large skews that prolong finish times of jobs.

5.3.1. Spark partitioning

Spark uses the estimated size of input data to partition it across multiple executors. The formula used for determining partition amount and partition size can be seen in Equation 5.16 and Equation 5.17 respectively. Additionally, users can configure the maximum and minimum sizes of partitions, but they do not adapt or change throughout the lifetime of the application.

$$Partition\ amount = Total\ amount\ of\ cores \tag{5.16}$$

$$Partition\ size = \frac{Total\ input\ size}{Partition\ amount} \tag{5.17}$$

This partitioning method ensures that each job can run its job on all cores in the system (if available), maximizing parallelism and throughput of the 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, examine the situation in Figure 5.8 (a) where one of the partitions runs 5x longer than the others. Because of this skew, the system is not fully utilizing all of its available resources, and the response time of the entire job is delayed.

In Figure 5.9 (a), priority inversion is also observed, where the long job completes before the higher priority job can receive any resources to execute on. Since 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.

While the Spark AQE tries to balance partitions to avoid skews, it comes with two main drawbacks. First, the skew mitigation is only performed after recording initial runtime data of leaf stages, which may be the main execution units of the job. Secondly, the rebalancing only considers data distribution, rather than the runtimes of stages. While distributing data evenly can achieve a more balanced runtime, this is not always applicable with non-linearly scaling operations.

5.3.2. Runtime partitioning

To mitigate skews and priority inversions caused by default Spark partitioning of jobs, we introduce runtime partitioning. Runtime partitioning attempts to split jobs into partitions that run in constant time, allowing for tasks to be distributed more evenly and reducing the maximum amount of time a task can reserve an executor. The formulas for calculating the number of partitions and their size can be seen in Equation 5.18 and Equation 5.19

$$Partition\ amount = \frac{Job\ runtime}{ATR} \tag{5.18}$$

$$Partition\ size = \frac{Total\ input\ size}{Partition\ amount} \tag{5.19}$$

where ATR value is the user or system-defined Advisory Task Runtime (ATR) that the partitions are expected to execute in. First, we need to collect or estimate accurate $Job\ runtime$, which is necessary to perform this partitioning method. Once the job 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, for each of the partitions, a task will be made, 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 jobs can be split into significantly more partitions than there are cores in the system. Since tasks are split into more partitions than cores, the skewed partitions are additionally split, which allows for spreading the runtime more evenly across executors, as seen in Figure 5.8 (b). Priority inversions are also mitigated by this, since now tasks release executors much faster, allowing for higher priority tasks to get assigned much quicker than they would under regular partitioning, as exemplified in Figure 5.9 (b).

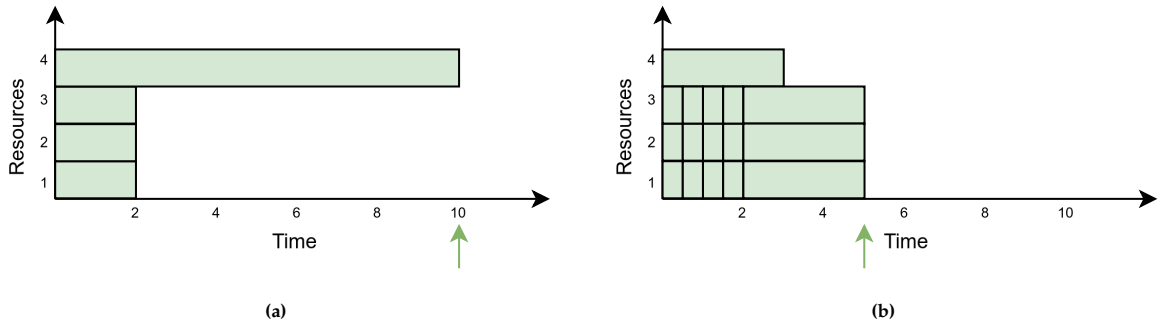
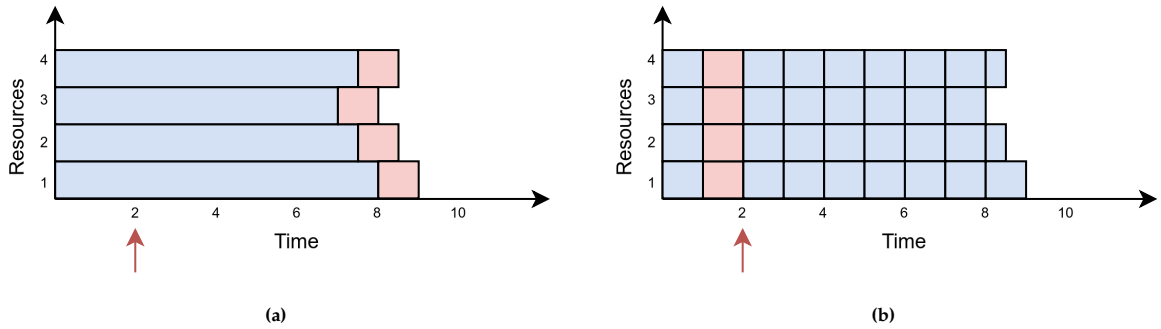


Figure 5.8: Skew impact on job runtime performance. The green arrow indicates the finish time of the job. In (a) jobs are partitioned using static partitioning, while in (b) using runtime partitioning.

Examining this from a more theoretical standpoint, we know that the bound $F_i - f_i \leq \frac{L_{max}}{R} + 2 * l_{max}$ is mainly impacted by runtime skews that account for the term $2 * l_{max}$ and priority inversion that makes up the term $\frac{L_{max}}{R}$. By introducing runtime partitioning, we are reducing l_{max} to $ATR * C$, where ATR



(a) (b)

Figure 5.9: Priority inversion of red and blue jobs. The red arrow indicates the expected finish time of the red job. In (a) red job has a smaller runtime and higher priority, but only executes once the blue job's tasks have finished. But in (b) red job can claim resources before the blue job has fully finished.

represents the median task runtime of the job, and C is the maximum factor that the runtime of a task can skew from the median. Additionally, runtime partitioning limits the amount of time each task can reserve an executor for, meaning that long-running tasks now run in multiple chunks, where an opportunity to schedule other jobs' tasks is present once a chunk finishes. This allows higher priority jobs to gain resources faster, which in theory reduces the term $\frac{L_{max}}{R}$ to ATR. With effective runtime partitioning, we can approach the theoretical bound $F_i - f_i \leq ATR + ATR * C = \hat{C} * ATR$, which gives users better control of maximum job response time delays.

6

Implementation details

In this chapter, we discuss implementation details of our scheduling model that is specific to our target system. The majority of the implementation is integrated into Apache Spark, allowing this model to be portable to similar systems that use the Spark engine for batch processing. First, we give an overview of the entire model and its scheduling workflow, then we discuss the details for integrating UWFQ and runtime partitioning with Apache Spark scheduling hierarchy, and finally, we address performance prediction as it currently exists in the target system.

6.1. Overview

General overview of the entire scheduling model can be seen in Figure 6.1. It consists of 3 main components: the UWFQ scheduler, the custom partitioner, and the performance estimator. The UWFQ scheduler is a Java implementation of the scheduler presented in the previous chapter, which extends Spark's `SchedulingAlgorithm` interface to ease integration. The custom partitioner splits jobs based on their expected runtime into appropriately sized partitions to reduce the impact of task skews and avoid priority inversions caused by long-running tasks. Finally, the performance estimator keeps track of all the jobs that are running in the system and tries to estimate their runtime by using historical data. To integrate it into Spark, we use the Spark Listener interface that allows the performance estimator to easily track job events that occur asynchronously in the system.

The workflow of jobs is as follows. First, users submit their jobs to the Spark driver (1), where it is first picked up by the Spark listener (2) and forwarded to the performance estimator (3). The performance estimator uses metadata and historic runs to assign a runtime estimate to the submitted job, which is then used by the custom partitioner (4) to derive the appropriate partition size. Then, the UWFQ scheduler uses the partitioned jobs (5) and their corresponding runtime estimates (6) to determine their priority and schedule them onto the executors (7, 8). Once jobs complete, their runtime metrics are collected by the Spark listener (9) and forwarded to the performance estimator (10) to update the estimates with real-time measurements of runs.

6.2. Scheduling details

The UWFQ algorithm aims to replace the default fair scheduling algorithm that is provided by Apache Spark. Because of this, it has to be integrated into the infrastructure to account for Spark internal scheduling units, and the flow of scheduling. Additionally, we discuss some external components that need to be accounted for while scheduling, and how to deal dynamicity present in the system environment.

6.2.1. Scheduling of Spark internals

The Spark computation hierarchy defines jobs, which consist of a DAG of stages, where each stage consists of one or more independently executable tasks. We describe how our algorithm fits within the existing Spark scheduling infrastructure, and define a new layer, viz., user jobs. The computing

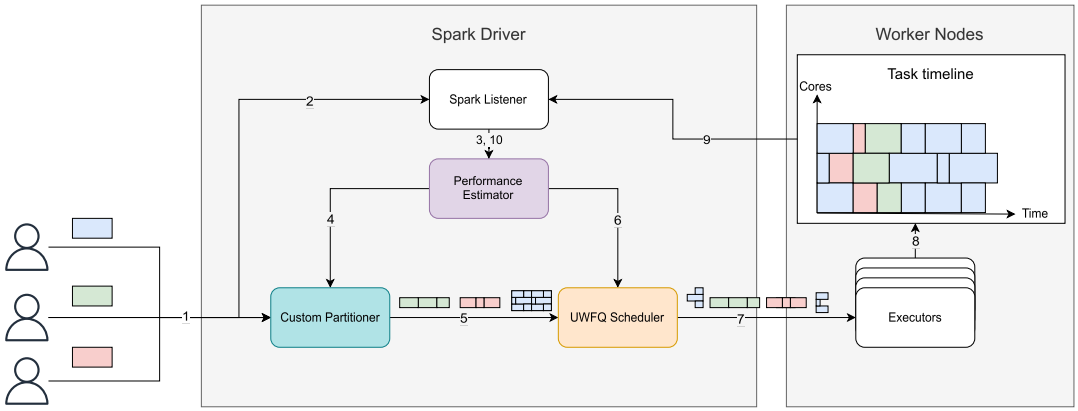


Figure 6.1: Overall scheduling system outline. In this scenario, there are 3 users asynchronously submitting jobs with varying runtimes. Each job is then partitioned according to its total runtime and scheduled onto the executors. Since we assume that the largest job arrives first, its task will be initially scheduled first, but tasks associated with smaller jobs will take priority once the resources are free.

hierarchy can be seen in Figure 6.2

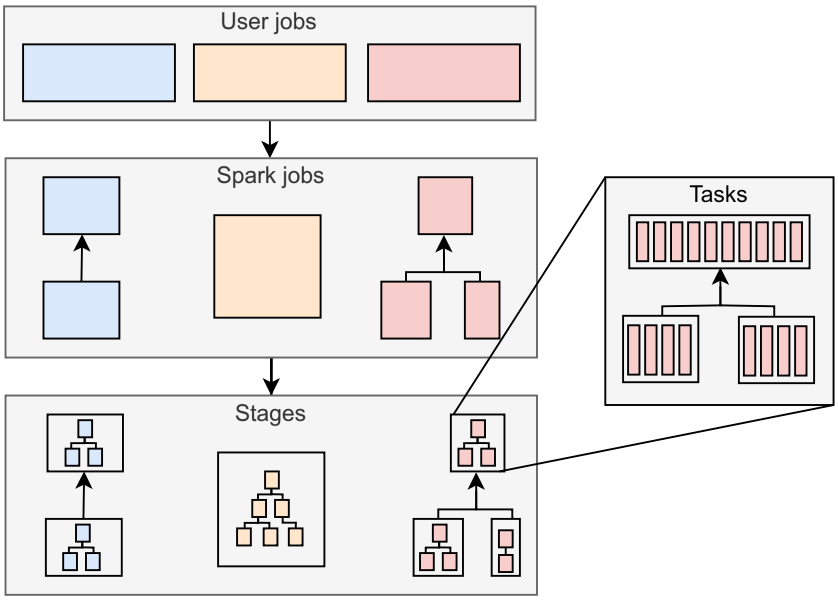


Figure 6.2: The computing hierarchy we use to represent computing units in Spark infrastructure. The user job layer is defined by us, while the following layers are already present in the Spark infrastructure.

Spark job and user job

The highest level of Spark's computation unit hierarchy is a Spark job. A Spark job correlates to a single Spark action submitted by the end user to perform a computation. In the current Spark implementation, there is no scheduling that happens on the job layer, and the scheduling decisions are forwarded to the lower-layer units. However, some metadata is embedded during the conversion from Spark jobs to a DAG of stages, which influences later scheduling decisions, for example, priority of arrival, which is used by FIFO scheduling.

Ideally, Spark represents each user query as a single job. However, this is not necessarily the case when using AQE in Spark, which may spawn multiple jobs for the same query. To account for this, we define a **user job** to be a DAG of internal Spark jobs, which in the end produce a result that is useful for the user. User jobs additionally hold the necessary metadata needed for UWFQ scheduling, such as user association and job identifying information, which will be preserved as the user job is decomposed by subsequent layer units.

Spark stages and job context
Stages are the next level in the computing hierarchy, and are the main units responsible for defining task priority in the Apache Spark infrastructure. Stages have the embedded information of the Spark job that created it, and the user job metadata. For this reason, scheduling in Spark happens on the level of stages, where the stage with the highest priority is determined, and its tasks are scheduled onto the executors whenever the resource is 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 user job from which the priority is inherited. Based on when the user's 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 user's job. This ensures that even if a job is scattered around many stages, it will still be executed within the guaranteed user-job fair time bounds. We define this stage to the user job association as **job context**, which we will later show can produce more efficient schedules.

In the Spark ecosystem, the highest priority is assigned to the stage with the lowest priority value P_s. This fits perfectly with global virtual deadlines, since the job with the lowest deadline value D should be scheduled the earliest. We express the priority assignment in Equation 6.1.

$$P_s = D^i_{global} \tag{6.1}$$

Spark tasks
Finally, the lowest level of computing units is tasks, which are the concrete units that are scheduled onto the executors. Tasks are fully independent of one another and can run on any executor, but are only scheduled once the higher-level dependencies are resolved, i.e, stage and job dependencies.

Each task operates on a slice of the stage input data, which is assigned during stage creation. By default, each stage will partition its data at most by the number of total cores in the system. However, using custom partitioning, we go beyond this amount and attempt to partition tasks into finer units.

Scheduling of tasks happens one at a time, by first determining the highest priority stage and taking one of its tasks to be scheduled onto the executor. Since some executors may have the input data already present, tasks prefer being scheduled on the executor that has the highest locality of input data.

To enable UWFQ, we did not have to make any changes to the Spark task layer, thereby preserving its original behavior.

6.2.2. Integration of external layers
The external layer that uses the Spark engine for batch processing only has to interact with the user job scheduling layer to properly integrate with UWFQ. This is because all metadata that is embedded into the user job will be propagated to subsequent layers, allowing UWFQ to make proper scheduling decisions.

UWFQ only needs 2 essential properties to be provided 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 scheduled by the same user job, which allows stages to be scheduled as a singular entity, instead of being independent. Since users are only concerned with the final outcome of their user job, the intermediate results of individual stages do not matter.

The target system we are working with introduces another layer of abstraction, namely, workflows. Workflows in this scenario are at the highest level in the hierarchy, where they define the dependencies amongst multiple Spark jobs. To accommodate the workflow layer in our scheduling, we simply define them as user jobs. This is because the intermediate results of each job within the workflow do not matter, and only the final result of the workflow is essential. This is equivalent reasoning that is applied for the definition of user jobs. To estimate the runtime of a workflow, we sum each of the Spark jobs that are a part of the DAG, which is then the estimate we would use to perform the same scheduling approach we do with regular user jobs.

While modelling workflows as user jobs allows for simple integration with UWFQ, this does not necessarily produce the most efficient schedule. Since we only assign a single priority for the entire workflow, each job under the corresponding workflow inherits the same priority, which may cause inefficient scheduling amongst workflow jobs that simultaneously have no pending dependencies. To avoid this pitfall, we plan to incorporate another layer of priority assignment within a workflow, which would function similarly to HEFT [57]. This would allow prioritizing the critical path of the workflow while scheduling, minimizing the makespan and consequently the response time of the entire workflow. However, due to time limitations, we leave it as future work, as there is not enough time in this project to implement it on the target system and measure its impact.

6.2.3. Accounting for dynamic environments

UWFQ quality is highly dependent on the accuracy of runtime estimation. Since we are working with a live system, where estimates can be inaccurate or updated during execution, some additional components were added to deal with the real scheduling environment.

Grace period

UWFQ removes users from the system as soon as all of their jobs have finished. This is necessary to accurately distribute resources amongst 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 the user's job have finished. This creates a scenario where there is a disconnect between the real system and 2-level virtual time, where the real job is still executing even though it has finished in virtual time. If the user job still has stages that have not been scheduled, these stages will not be attributed to the correct start time, as visualized in Figure 6.3.

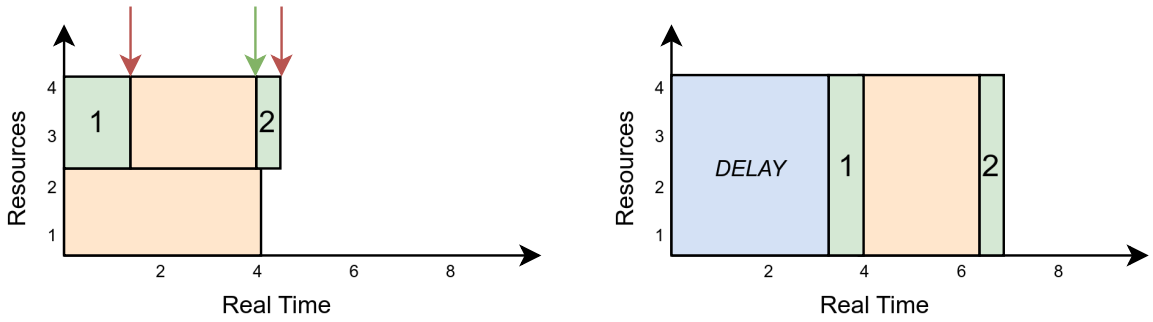


Figure 6.3: The disconnect between real and virtual representations of the schedule. In virtual schedule (a), user exits and forfeits their resources (red arrow), and comes back (green arrow) only after stage 1 has completed in the real scheduler (b). This delays the completion of the green user job.

To correct this, we introduce a grace period, in which the following stages can "revive" a user that has exited the system with their original arrival time. This allows stages to be retroactively scheduled at their appropriate start times, even if they are delayed in the real scheduler as seen in Figure 6.4.

While this provides the solution for these scenarios, it does present 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 can potentially 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 in Equation 6.2 is satisfied, where $V^k_{global,\,end}$ is the global virtual end time of user k, and T_{grace} is the grace period. 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.

$$V_{global} < V^k_{global,\,end} + T_{grace} * R \tag{6.2}$$

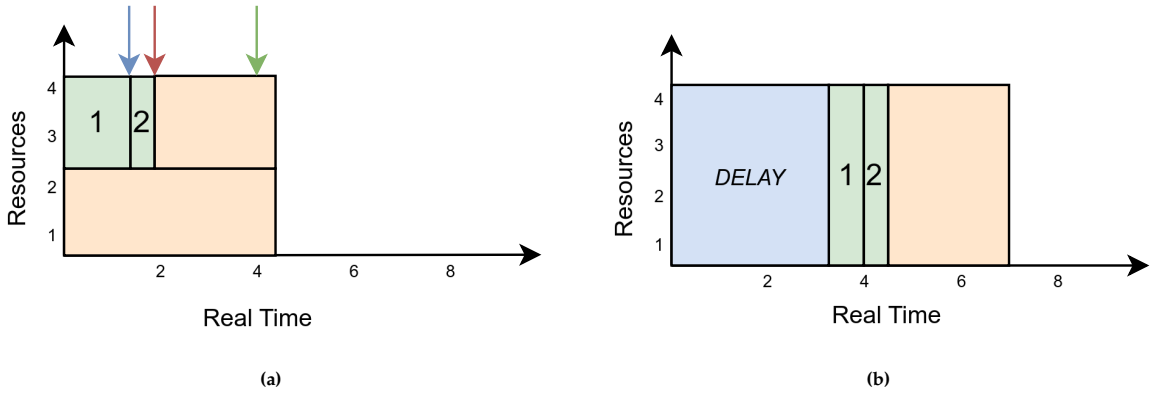


Figure 6.4: Using grace period to correctly attribute start times of jobs. In virtual schedule (a), user retroactively "revives" (blue arrow), and schedules the second stage right after the first one, allowing them to execute sequentially in the real scheduler (b).

An alternative solution that was explored is flagging stages as final if they were to finish the associated user job. This would indicate that a user must be kept alive until the last stage of a job has arrived. However, this solution is currently unfeasible, due to not having enough information to infer if a stage is final, and scheduler inflexibility to handle user job cancellation.

Updating user job runtime

User jobs consist of many stages, where each stage is submitted independently to the scheduler. Since there is some time in between stage submissions, the overall runtime of the associated user job may have updated in that period to be more accurate than the initial estimate. To facilitate this update, each stage interfaces with the performance estimator component to get the most up-to-date runtime of the corresponding user job, and updates the local runtime in the UWFQ. The update of virtual deadlines is already facilitated by Algorithm 1, since all global virtual deadlines are reassigned for the corresponding user.

However, the current implementation does not update the runtime of the job once it's finished and only uses the most recent estimate. This can create skew in virtual time that may cause priority inversions or deadline violations to occur. We propose that this can be solved by having another scheduling component that updates all jobs with their real runtimes and adjusts the virtual time accordingly. A similar approach is done for EEVDF [52], where clients distribute their unused execution time to other clients. But due to time limitations, we leave it as future work.

6.3. Partitioning

Partitioning has to be done at a finer granularity level than scheduling, since partitioning happens on the stage layer instead of the user job layer. This complicates runtime estimation, because instead of just estimating the runtime of user jobs, intermediate stage runtime has to be estimated as well. By naively implementing partitioning without considering the possible inaccuracies of estimation, the performance of the system can be greatly reduced.

Partitioning integration with Spark

Spark performs partitioning in two distinct phases of job execution: during the initial input reading and when coalescing between shuffle stages.

When any Spark job is submitted, it will have to perform a file scan that will load the necessary data into the first stage. During the file scan, partition sizes are calculated with Equation 5.17, which are then used to divide the given input files between the tasks. To introduce runtime partitioning, we override this function with runtime partitioning, which interfaces with the performance estimator to get a runtime estimate, and attempts to partition it according to Equation 5.19.

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, AQE coalesces these partitions into more appropriately sized partitions using

the intermediate output sizes, without considering the upcoming job 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 a dynamically calculated amount from Equation 5.18. This ensures that the partitions never coalesce down to an amount that would introduce long-running tasks, and also does not interfere with AQE's own methods of reducing runtime skews.

Partitioning explosion

An observed phenomenon that can occur when partitioning based on the historic runtimes is the "partitioning explosion". By increasing the number of partitions, a very gradual increase in job slot-time is observed. This slowdown is usually negligible and overshadowed by the skew reduction. However, by gradually increasing the slot-time of jobs, the number of partitions increase as well, which further increases the overhead time added to the slot-time. This can continue indefinitely until the overhead of having too many tasks accounts for a significant portion of the runtime, as was also pointed out in Subsection 2.1.7.

To avoid this issue, partitioning is done more gradually while keeping track of changes in slot-time and longest task time. The custom partitioner tries to approach the predefined $Advisory\ task\ runtime$ for the longest running task, while keeping within the bounds of maximum allowed slowdown. Once maximum slowdown is encountered, the partition amount will be reduced until no significant performance improvements are observed.

6.4. Performance prediction

Performance prediction is at the core of enabling efficient scheduling and partitioning in the system. However, deriving an accurate runtime estimate is difficult given so many layers of computing unit abstraction, changing input data, and variance in operations. To obtain as much of the available information as possible without interrupting execution, we utilize Spark listeners that can asynchronously gather system event information and metadata.

Data collection

Spark listeners can collect many types of events that happen in the Spark ecosystem. For our purpose, the most notable events are SQL, stage, and task events.

Most Spark jobs are internally represented as SQL queries, which are described by a Spark plan. The Spark plan initially describes the logical operations that will be applied to the input data to produce the result of the query. This plan is then optimized into a physical plan, which describes in more detail the physical operations that will be applied. Because of AQE, the physical plan is updated after every stage using the most up to date runtime statistics to further improve the plan. Spark listeners are able to track all these updates and send them to the performance estimator. However, since the plans are constantly updating, it makes it difficult to perform estimation without knowing the final plan. Additionally, SQL operations do not directly map to stages, since multiple operations can be grouped together in code generation to make the stage execution faster.

To work around these limitations, we create a wrapper data structure that crafts a stage-node tree from the given Spark plan, and updates the representation as the plan changes. Stage-node represents the grouped SQL operations that are executed within a single stage, as can be seen in Figure 6.5, and can be more directly mapped to stages and vice versa. By having this stage-node structure, we can more easily associate stage runtimes with the underlying SQL query, and perform stage-level runtime estimation by mapping query stage-nodes to already finished stage-nodes of similar structure.

Stage events allow us to keep track of real stage execution statistics, which we use for gathering real runtimes of previous stages that can be used for future estimation. The same is done with task events that allow us to examine task-level statistics, which can be useful for identifying skew and long-running tasks.

Runtime inference

Current implementation infers stage runtime by averaging the last few runtimes of the corresponding stage-node. To get the runtime estimate of an entire user job, the runtime of individual stages can be

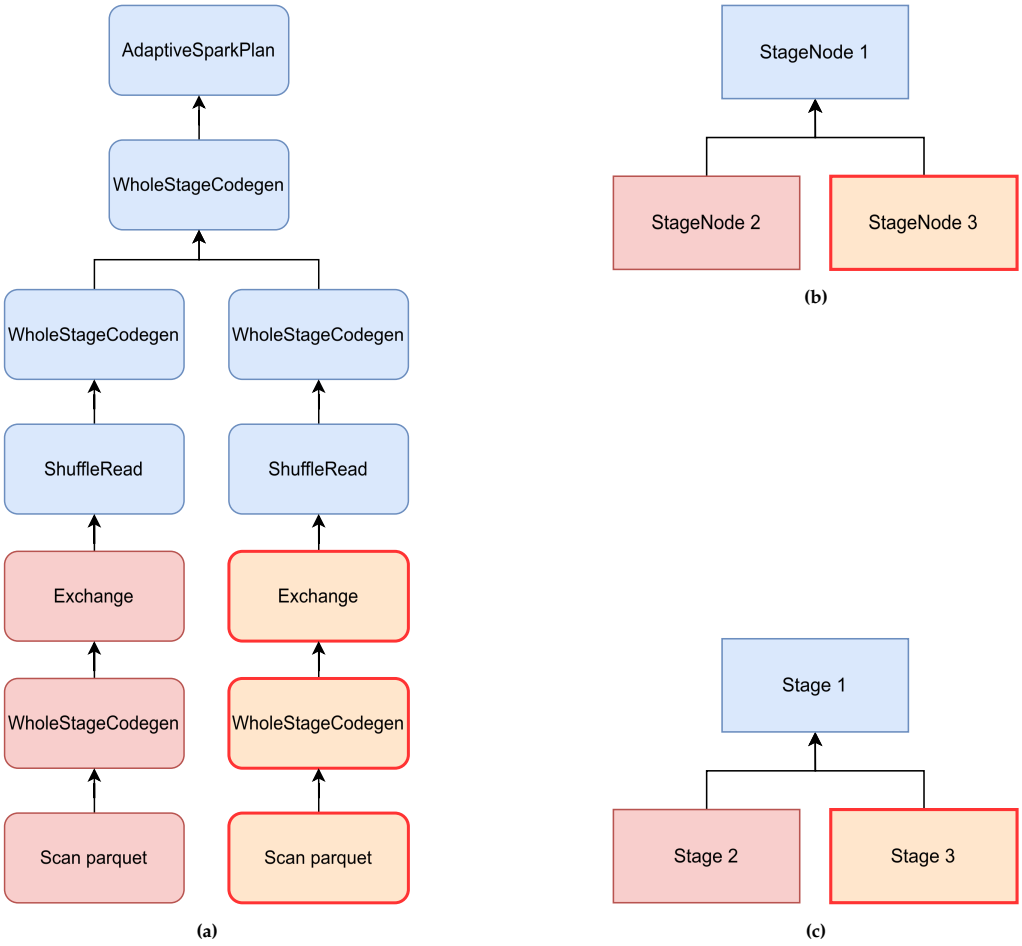


Figure 6.5: Spark plan being represented by (a) SQL operation tree, (b) stage-node tree, and (c) real stage tree. Each stage node maps to a sequence of Spark internal SQL operations. It can be observed that the stage tree and the stage-node tree are equivalent.

summed up together, or the user job can be mapped to previously finished user jobs, estimating the runtime by averaging the latest finished jobs.

In addition to leveraging historical runtimes for estimation, information about the applied SQL operations can also be used to predict runtime. This can be achieved by benchmarking individual SQL operations offline with varying input sizes, allowing us to model the relationship between input size and execution time of certain SQL operations, as was done by Wu et al. [66]. Another method that can be used is using regression trees for inferring the runtimes of SQL operators, and scaling them based on user input, as shown by Li et al. [34]. Unfortunately, due to time limitations, we were unable to make these estimators work as of the writing of this report, and leave it for future work.

7

Performance Evaluation

In this chapter, we evaluate the performance of UWFQ and report on our results. We use two main types of benchmarks: micro-benchmarks and macro-benchmarks, and address testing under an industry environment. We compare UWFQ with the default Apache Spark built-in fair scheduler to measure the absolute improvement, and other representative schedulers to better isolate the primary reason for the performance gain. Additionally, we address the impact of runtime partitioning on response time and fairness.

Table 7.1: Notation definitions

Notation	Explanation
$_i$	Job i
$_s$	Stage s
$_k$	User k
T	Absolute time
P	Priority
N	Amount of items
D	Deadline
RT	Response time
SL	Slowdown
DVR/DSR	Deadline violation/slack ratio

7.1. Performance Metrics

With our experiments, we want to primarily show that the UWFQ scheduler can improve response time while still ensuring bounded user-job fairness. To measure this, we collect user job response times, slowdowns, and deadline violation and slack ratios. We derive these metrics from previous work done to measure fairness in scheduling, as discussed in Section 2.3. While we consider all these metrics for the analysis, some may be omitted in the result sections to highlight the most important findings.

We calculate the **response time** (RT) of a user job by measuring the time that elapses since its first stage is submitted until the last stage is completed, as seen in Equation 7.1. To visualize the differences between schedules, we plot RTs as ECDFs. We additionally examine the differences of average and worst 10% RTs between all user jobs, different types of user jobs, and different users, if applicable.

$$RT_i = \text{MAX}(T^i_{s,end}) - \text{MIN}(T^i_{s,start}) \tag{7.1}$$

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 an idle system, as shown in Equation 7.2. Compared to response times, slowdowns show relative gains and losses in performance instead of absolute units. We examine the average and worst 10% slowdown between all user jobs, different types of user jobs, and different users. Since slowdown is relative to response time, we focus our analysis mostly on response times for a clearer overview of the system, and provide slowdowns to give readers a proportional reference point.

$$SL_i = \frac{RT^i_{shared}}{RT^i_{idle}} \tag{7.2}$$

Finally, we measure **deadline violations ratio** (DVR) by taking the end time difference between the target scheduler and UJF, normalizing it by the UJF runtime (equivalent to response time) of the job, and finally calculating the average of the of incurred proportional violations, as seen in Equation 7.3. 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.

Since DVR only measures violations, we contrast this with **deadline slack ratio** (DSR), which correspondingly calculates the average of proportional slack time gained, as seen in Equation 7.4. This showcases the speed-up that the target scheduler brings in comparison to UJF. We visualize the proportional violations and slack with box-plots and calculate DVR and DSR for the set of all jobs, the set of jobs for each user, and the set of jobs of each type. In box-plots, positive and negative values indicate job's proportional violation or slack, respectively.

$$DVR = \frac{\sum_i \max(0, r_i)}{\sum_i 1_{\{r_i > 1\}}} \quad \text{where } r_i = \frac{\text{MAX}(T^i_{s,end,\text{target}}) - \text{MAX}(T^i_{s,end,\text{UJF}})}{RT^i_{\text{UJF}}} \tag{7.3}$$

$$DSR = \frac{\sum_i \max(0, -r_i)}{\sum_i 1_{\{r_i \leq 1\}}} \quad \text{where } r_i = \frac{\text{MAX}(T^i_{s,end,\text{target}}) - \text{MAX}(T^i_{s,end,\text{UJF}})}{RT^i_{\text{UJF}}} \tag{7.4}$$

Data collection

The Apache Spark listener framework provides all the necessary metrics needed for measuring performance. We collect this data by enabling event logging, which collects Spark events during execution, which can be later replayed using the Spark History Server. Spark History Server recreates the SparkUI of previously run applications and allows using the SparkUI API endpoints to extract runtime data. We list the extracted data in Table 7.2.

Table 7.2: Extracted measurements from Spark UI

Unit	Collected measurements
Job	submissionTime, completionTime, stageIds, jobGroup, jobId
Stage	status, firstTaskLaunchedTime, completionTime, stageId, tasks
Task	launchTime, executorId, executorRunTime, writeTime
Executors	JVMHeapMemory, JVMOffHeapMemory, totalShuffleWrite, totalShuffleRead

For each benchmark, we run it at least 3 times, collecting its metrics and aggregating the results. The table entries reflect these aggregated metrics, however, the visuals are based on a single run. This is mainly to ease visualization, since there are no significant scheduling differences across runs.

7.1.1. Baseline Schedulers

In micro-benchmarks, we compare UWFQ to several representative schedulers. We show the general priority formulas for each of these schedulers. Note that in the Spark ecosystem, 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, as seen in Equation 7.5. While this is the scheduler we are aiming to replace, it does not implement UJF, hence it does not necessarily give a good indication of fairness.

$$P_s = N^s_{active\ task\ amount} \tag{7.5}$$

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 least amount of tasks N^k, as seen in Equation 7.6. 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 infinite parallelism is not feasible to achieve.

$$P_k = N^k_{active\ task\ amount} \tag{7.6}$$

Cluster Fair Queuing

We implement Cluster Fair Queuing (CFQ) [10] for our benchmark by updating the public source code published on GitHub [1]. 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 [43], as seen in Equation 7.7, omitting the wider context of users and jobs.

$$P_s = D_s \tag{7.7}$$

True FIFO

To showcase the isolated benefits of job context when scheduling stages, we implement True FIFO (T-Fifo), where the highest priority is assigned to stage s that is associated with the earliest arriving job time T^i, as seen in Equation 7.8. We use this scheduler to showcase the effects of considering job context in certain scenarios to improve job response. Note that UWFQ is also job-aware, hence inherits the benefits that will be highlighted by True FIFO.

$$P^i_s = T^i_{arrival} \tag{7.8}$$

Random scheduling

Random scheduling is used as a general baseline to see if performance can be significantly impacted by mindful decisions in comparison to random actions. The priority is determined by hashing the name of the stage s as seen in Equation 7.9.

$$P_s = \textsc{hash}(s_{name}) \tag{7.9}$$

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

Using runtime partitioning

All schedulers are initially measured by using the default partitioning already present in Apache Spark. However, to highlight the impact of using runtime partitioning, each scheduler is additionally measured while employing the partitioning algorithm. To separate them from their default partitioning counterparts, 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.

7.2. Micro-benchmarks

Micro-benchmarks are synthetic workloads that isolate specific characteristics to better measure scheduler impact on them. In this section, we describe the benchmark setup utilized for collecting measurements, baseline schedulers that will be compared to UWFQ, and the synthetic scenarios used for evaluation.

7.2.1. Setup

Cluster and hardware

Experiments for micro-benchmarks are conducted on the DAS-5 cluster [4]. DAS-5 is a wide-area distributed system maintained by the Advanced School for Computing and Imaging (ASCI). It provides researchers the ability to perform experiments on multiple computing nodes.

For our experiments, we reserve 5 computing nodes, each 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.

Environment

Spark's source code, while providing the necessary interfaces for replacing the scheduling algorithm, does not provide any driver configuration that can load a custom scheduler without having to recompile its source code. To tackle this, we extend the Spark source code to allow for dynamic class loading of custom schedulers, allowing us to compile the source code once and load whichever scheduler we want to use dynamically.

We run our experiments on a modified Apache Spark 3.5.5 version[2] built with Scala 2.12. The modifications applied to Spark are only to enable custom scheduler 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 configurations at default values with a few exceptions (see Appendix B). 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 default settings cause the small file problem to occur as documented in Figure 2.1.7. We solve this by increasing the $maxPartitionBytes$ to an empirically measured amount to avoid this. The experiment setup source code can be viewed on GitHub[3].

Dataset

Data used for all micro-benchmarks is NYC Taxi and Limousine Commission (TLC) For-Hire Vehicle High Volume (FHVHV) Trip Records[4] from August 2024. The records contain fields capturing pickup and drop-off times, locations, and other information about the trips. 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.

[2]Modified Apache Spark 3.5.5: https://github.com/kazemaksOG/spark-3.5.5-custom
[3]Experiment source code: https://github.com/kazemaksOG/spark-benchmark-tool
[4]TLC trip dataset: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

Workloads

We simulate user jobs by applying a varying number of operations per row on the provided dataset. A single user 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 to have 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 is measured to takes only a couple of milliseconds.

In the following scenarios, we only define tiny, short, and long jobs, where each type of job always performs the same operations. We measure their runtimes in an idle system, which are collected in Table 7.3, and use them for slowdown calculations. While this may not accurately simulate dynamic runtimes seen in real settings, it is sufficient to highlight the interaction between jobs of the same and different types based on their arrival times and system congestion. In some scenarios, we mimic infrequent user behavior by using a Poisson distribution. A Poisson distribution is commonly used in scheduling to simulate user action frequency in previous works [29, 28, 44], and it gives fine control over user workload without hard-coding arrival times.

Job type	Response time (s)
Long job	10.86
Short job	2.25
Tiny job	0.90

Table 7.3: Job runtimes when running in an idle system, with default Spark settings and Fair scheduler.

For these benchmarks, we use accurate job profiling to isolate analysis purely on the scheduler's quality. Job profiles are obtained by analyzing runtimes collected from scenario runs using the built-in Fair scheduler. While these runtimes accurately represent runtime for the Fair scheduler, this may not be the case for other schedulers, due to fluctuation induced by JVM warm-up and dynamic data locality. To reduce the impact of JVM warm-up, before each micro-benchmark, we run each of the job types twice on the same input data. This ensures that benchmarks are less influenced by previous executor activity.

7.2.2. Scenarios

In collaboration with industry specialists, we define a small subset of scenarios that have been observed in the real system. These scenarios have been exaggerated to better highlight the key differences in scheduling. To further isolate specific aspects of each scenario, we simplify certain dynamic properties of the jobs, such as varying runtimes.

In each scenario, we simulate different user classes competing for resources. We define user classes for each scenario to associate certain characteristics with the users' job scheduling patterns. In scenarios where we have 2 classes, having 4 users ensures that each class is represented by at least 2 users. This allows us to not just measure resource competition among user classes, but also among users of the same class.

Scenario 1: short and long-running users

The most common situation that is observed in the real system is when there are some users who have scheduled a long-running job (long-running users), while other users are trying to obtain results for relatively quicker jobs (short-running users). The main issues that can arise from this are long-running jobs delaying the execution of shorter jobs, or shorter jobs taking too much priority and violating the deadlines of longer jobs.

To emulate this scenario, we create 2 short-running users and 2 long-running users that compete for system resources. The short-running user class schedules their jobs following a Poisson distribution to mimic user submission behavior, while long-running users continuously schedule jobs, always demanding system resources. We visualize this scenario in Figure 7.1.

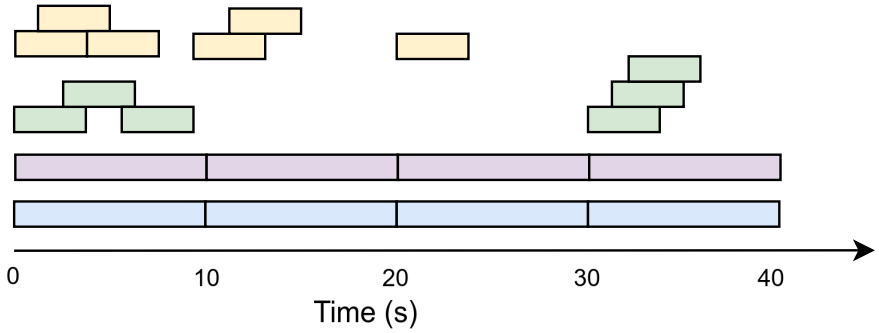


Figure 7.1: Job arrival timeline of scenario 1. The length of each job equates to expected runtime, however may significantly differ due to system congestion. Each color represents a different user's jobs.

Scenario 2: infrequent and frequent users

The previous scenario can similarly be expressed by replacing long-running jobs with more frequent short-running jobs. This essentially causes the same amount of resource congestion as the previous problem, however, having the jobs be the same runtime and arriving in bursts creates a more dynamic scheduling scenario, where more emphasis is put on resource distribution among users rather than the runtime of jobs. Main issues in the 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 similar to the previous scenario, however, they are scaled to better span the runtime of the scenario. A frequent user class schedules a burst of short jobs every 30 seconds that fully congests the system, introducing delay for all jobs. We show this scenario visually in Figure 7.2.

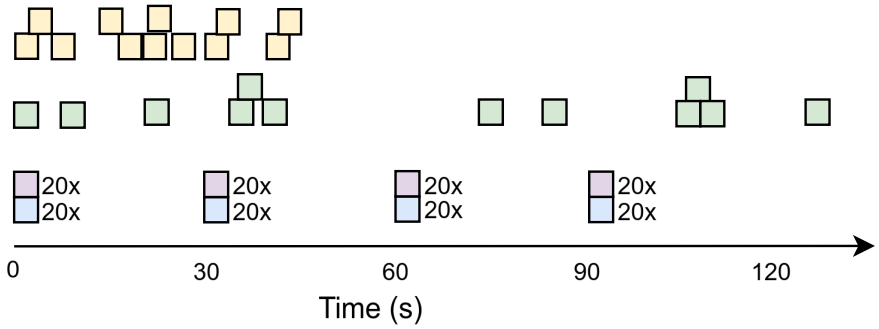


Figure 7.2: Job arrival timeline of scenario 2. The length of each job equates to expected runtime, however may significantly differ due to system congestion. Each color represents a different user's jobs.

Scenario 3: multiple long-running users

Since most short-running jobs leave the system quickly, there are scenarios where the only jobs that are left running in the system are the long-running jobs. This scenario explores how long-running job runtimes are affected by different scheduling methodologies. The main issue that has been observed is equally distributing available resources among all long-running jobs, which ensures fairness, however, it increases the response time of all long-running jobs.

For this scenario, we define 4 users where each of whom runs long-running jobs that are consecutively scheduled one after another. Each user has a predefined start delay, to ensure that the arrival order is consistent across different runs. While being less realistic, some findings can still be extracted from this scenario. We show the schedule timeline in Figure 7.3.

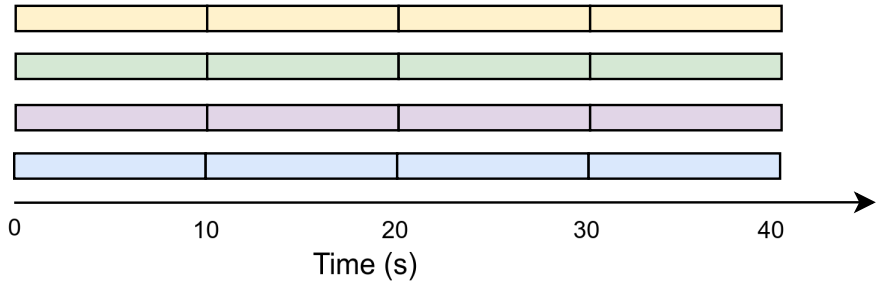


Figure 7.3: Job arrival timeline of scenario 3. The length of each job equates to expected runtime, however may significantly differ due to system congestion. Each color represents a different user's jobs.

Scenario 4: multiple frequent users

Lastly, we explore the scenario where multiple users supply a burst of jobs to the system, and how effectively the system can recover while providing fair service to all users. The main issues of this scenario are favoring some users more than others, and attempting to compute all jobs simultaneously, which can increase response times.

We implement this scenario by simply having 4 users schedule many tiny jobs simultaneously. Each user has a predefined start delay, to ensure that the arrival order of users is consistent across different runs, however, job completion order may differ. We show a general visualization of this in Figure 7.4.

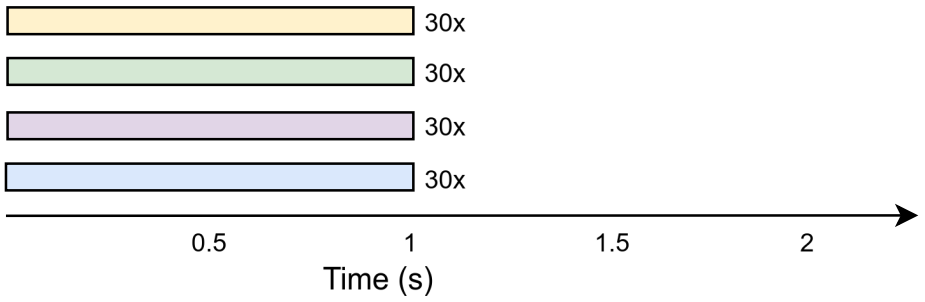


Figure 7.4: Job arrival timeline of scenario 4. The length of each job equates to expected runtime, however may significantly differ due to system congestion. Each color represents a different user's jobs.

7.2.3. Results

In this section, we analyze each scenario and identify key factors that influence the response times and fairness of these schedules, and how UWFQ and partitioning affects these results. To highlight only the most important aspects, most collected metrics are omitted from the analysis. Full tables and traces can be viewed in Appendix C.

Scenario 1: short and long-running users

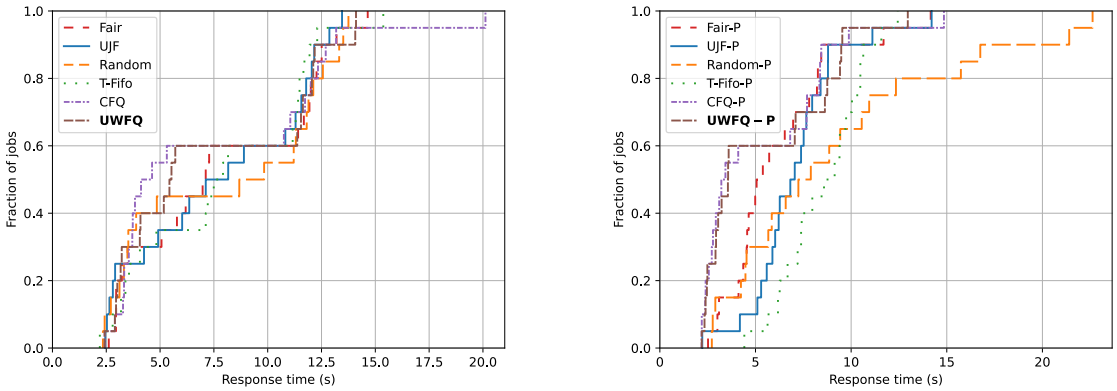
The values of key performance metrics are reported in Table 7.4, where we additionally highlight the response time and slowdown differences between short and long jobs. We showcase the ECDFs of scheduling algorithm response times in Figure 7.5, separating the figures for default and dynamic partitioning, respectively.

Schedulers that do not use dynamic partitioning tend to have similar performance as can be observed in Figure 7.5, with most schedulers only deviating by less than 5% for average response time. However, CFQ and UWFQ do reduce the average response times for short jobs by 19% and 22% respectively compared to UJF, which can also be observed in Figure 7.6 (a). But it comes with a cost that larger jobs have a longer tail response time, increasing the worst 10 percent response times by 24% and 10% respectively compared against UJF, seen in Figure 7.7 (a).

Due to inefficient partitioning, we observe that in all schedules, long-running tasks can reserve the

Scheduler	Response time (s) | Slowdown				Fairness			
	Avg.	Worst 10%	Short	Long	DVR	Count	DSR	Count
Fair	7.75 | 1.72	13.7 | 3.32	4.66 | 2.11	12.3 | 1.14	0.13	27	0.16	13
UJF	7.72 | 1.78	13.0 | 3.86	4.93 | 2.24	11.9 | 1.10	-	-	-	-
Random	7.78 | 1.75	14.2 | 4.25	4.79 | 2.17	12.2 | 1.13	0.23	23	0.19	17
T-Fifo	7.97 | 1.86	13.7 | 3.81	5.21 | 2.36	12.1 | 1.12	0.16	28	0.11	12
CFQ	7.50 | 1.56	16.2 | 2.43	4.00 | 1.81	12.7 | 1.17	0.37	22	0.28	18
UWFQ (this work)	7.25 | 1.51	14.3 | 2.57	3.88 | 1.76	12.3 | 1.13	0.16	25	0.29	15
Fair-P	6.20 | 1.50	13.0 | 2.50	4.30 | 1.95	9.04 | 0.83	0.13	11	0.23	29
UJF-P	7.07 | 1.91	12.6 | 3.66	5.80 | 2.63	8.96 | 0.82	-	-	-	-
Random-P	8.99 | 2.86	20.8 | 8.04	9.36 | 4.25	**8.44** | **0.78**	1.02	27	**0.73**	13
T-Fifo-P	8.52 | 2.64	11.6 | 5.28	8.56 | 3.88	8.48 | **0.78**	0.72	21	0.32	19
CFQ-P	**5.22** | **1.07**	12.1 | **1.56**	**2.70** | **1.22**	9.01 | 0.83	**0.06**	8	0.41	32
UWFQ-P (this work)	5.36 | 1.11	**11.2** | 1.61	2.83 | 1.28	9.15 | 0.84	0.10	**5**	0.37	**35**

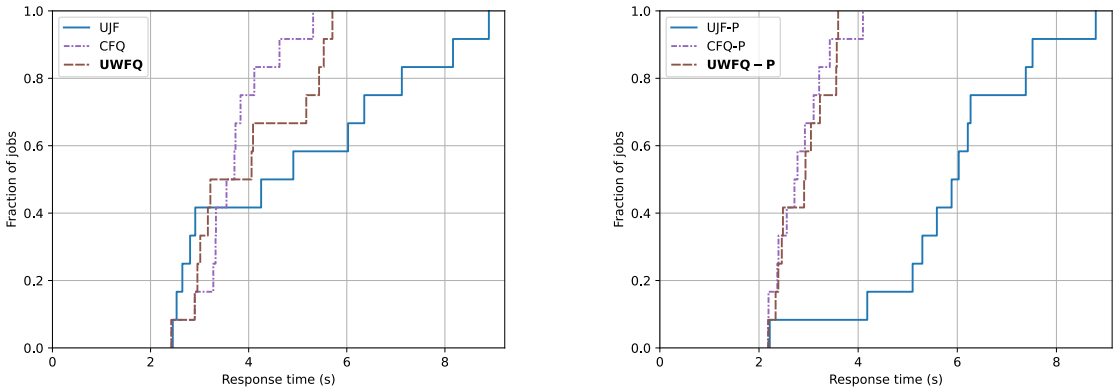
Table 7.4: Comparison of scheduler performance metrics for scenario 1.



(a) Default partitioning

(b) Runtime partitioning

Figure 7.5: Empirical CDFs of schedulers for scenario 1. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.

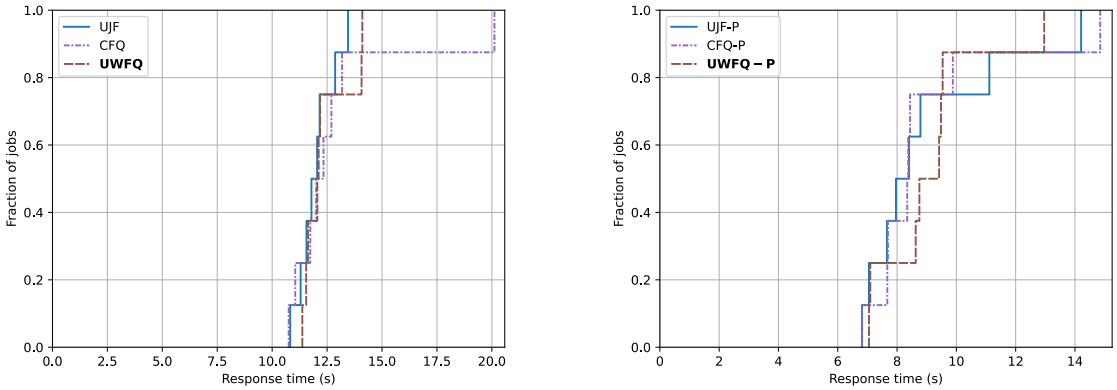


(a) Default partitioning

(b) Runtime partitioning

Figure 7.6: Empirical CDFs of small job response times for different schedulers in scenario 1. (a) compares UWFQ and CFQ with regular partitioning to UJF, while (b) compares UWFQ-P and CFQ-P with runtime partitioning to UJF-P.

executor resources for their entire run period, blocking short-running jobs from acquiring them to execute faster, highlighted in the task allocation trace in Figure 7.8 (a) and (b). Additionally, skews create scenarios where scheduling tasks based on their priority may delay long-running tasks from



(a) Default partitioning

(b) Runtime partitioning

Figure 7.7: Empirical CDFs of large job response times for different schedulers in scenario 1. (a) compares UWFQ and CFQ with regular partitioning to UJF, while (b) compares UWFQ-P and CFQ-P with runtime partitioning to UJF-P.

being allocated quicker, creating more gaps in the resource schedule of CFQ and UWFQ, which we can observe in Figure 7.8 (a), compared to UJF task allocation in Figure 7.8 (b).

But once we introduce runtime partitioning, CFQ-P and UWFQ-P can further decrease short job response times by 53% and 51% compared to UJF-P, respectively, and can be observed empirically in Figure 7.6 (b). Interestingly, it additionally improves response times of long-running jobs, removing the long tail response times, as seen in Figure 7.7 (b).

We believe that the first improvement is caused by the partitioning of long tasks, so short-running jobs are able to acquire all resources faster and finish execution quicker. Second improvement can be caused by skew mitigation instigated by dynamic partitioning, reducing the impact of long-running tasks that would elongate the runtime of the job. This can be clearly seen in Figure 7.8 (c), where dynamic partitioning causes the task allocation to be much more packed than in the regular partitioning algorithm.

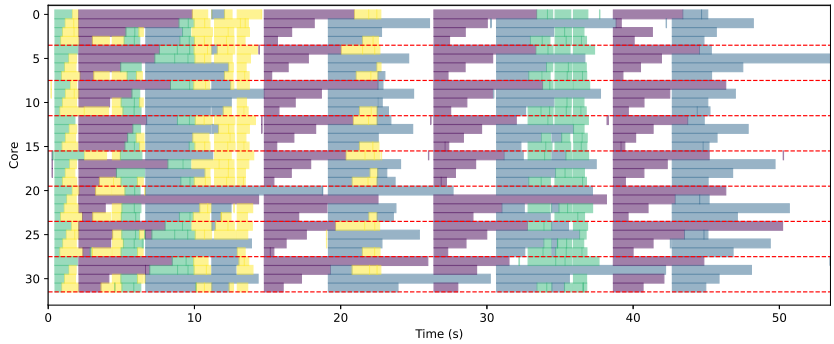
Examining the deadline violation and slack values for default partitioning approaches, it can be seen that most algorithms significantly violate the UJF deadlines, which is better visualized in Figure 7.9. However, when using runtime partitioning, most of the jobs for UWFQ-P and CFQ-P are able to have deadline slack and minimal deadline violations. While CFQ-P has slightly better response time and DVR value, UWFQ-P has less total violations. We believe that CFQ-P is able to perform well in this scenario because each user has usually at most a single concurrent job, which allows fairness across jobs to be almost equal to fairness across users, creating almost equivalent schedules to UWFQ-P.

Scenario 2 results

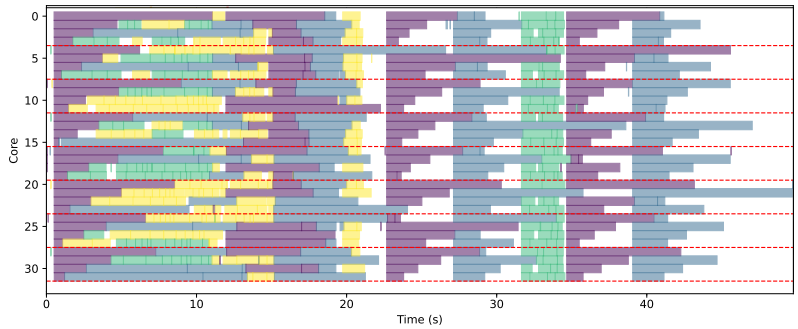
Key metrics are highlighted in Table 7.5, where we additionally showcase the response time and slowdown differences between frequent and infrequent users. Since all the jobs for this scenario are already short and well partitioned by the default partitioner, we do not include metrics from the schedulers using dynamic partitioning, since they do not show any significant changes. However, we do still showcase the ECDFs of scheduling algorithm response times for both partitioning approaches in Figure 7.10, so it can be visually observed.

T-Fifo and UWFQ achieve the best average response times and are 32% lower than UJF. We attribute this improvement to the job context that is present in both T-Fifo and UWFQ, where jobs are favored to run till completion, instead of interleaving between all active jobs. This is better visualized in Figure 7.10, where all jobs are completed gradually, rather than in batches.

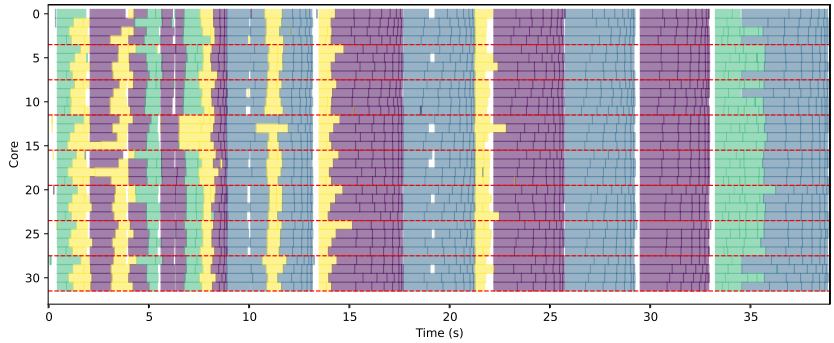
The most substantial improvement can be seen by introducing user context, where infrequent users have a much better response time in UWFQ and UJF, lowering the average response time in UWFQ by 89% compared to Fair, which is also highlighted in Figure 7.11. We believe that in scenarios where users differ in the amount of scheduled jobs, user context allows for fair resource distribution across users,



(a) UWFQ task allocation



(b) UJF task allocation



(c) UWFQ-P task allocation

Figure 7.8: Task allocation timeline on executor cores for different schedulers. Each color represents a different user's task, and gaps indicate an idle executor. Note that the time scales are different across figures.

Scheduler	Response time (s) | Slowdown				Fairness			
	Avg.	Worst 10%	Freq.	Infreq.	DVR	Count	DSR	Count
Fair	44.0 | 19.9	86.1 | 39.0	44.5 | 20.2	40.8 | 18.5	3.25	141	0.27	227
UJF	43.1 | 19.5	74.7 | 33.9	48.8 | 22.1	5.13 | 2.33	-	-	-	-
Random	37.9 | 17.2	113. | 51.2	38.2 | 17.3	35.6 | 16.1	3.48	124	**0.56**	244
T-Fifo	**29.3** | **13.3**	**47.0** | **21.3**	**29.1** | **13.1**	31.1 | 14.1	3.89	76	0.43	292
CFQ	32.5 | 14.7	49.8 | 22.5	32.3 | 14.6	34.2 | 15.5	3.69	91	0.38	277
UWFQ (this work)	29.4 | 13.3	50.3 | 22.8	33.1 | 15.0	**4.50** | **2.04**	**0.23**	**58**	0.37	**310**

Table 7.5: Comparison of scheduler performance metrics for scenario 2.

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.

By examining the DVR and DSR values, we can see that UWFQ achieves the least amount of deadline

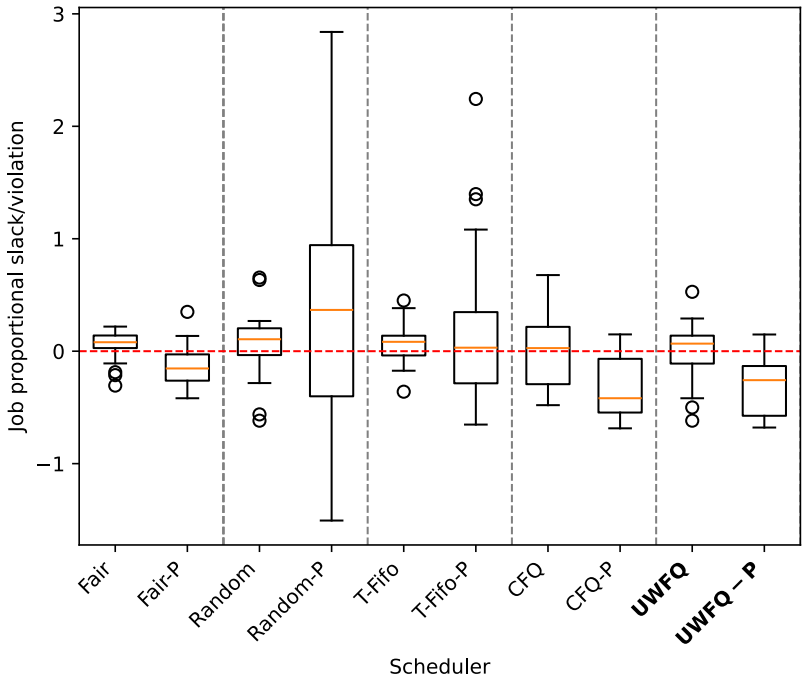
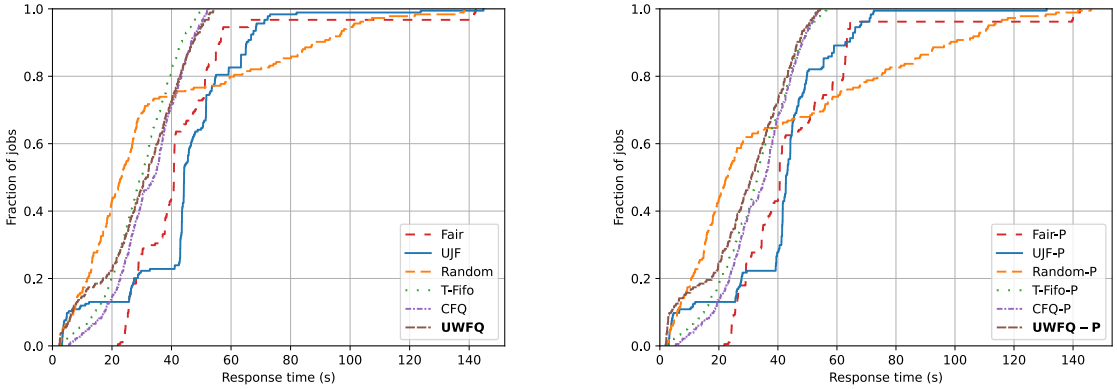


Figure 7.9: Box-plots of proportional deadline violations of jobs, where slacks are negative values and violations are positive. The value 0 indicates no improvement or degradation.



(a) Default partitioning

(b) Runtime partitioning

Figure 7.10: Empirical CDFs of schedulers for scenario 2. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.

violations and the smallest DVR value compared to the other algorithms. This can also be seen in Figure 7.12, where UWFQ has the outliers packed much closer to the deadline compared to all other scheduling algorithms.

Scenario 3 results

We showcase collected metrics in Table 7.6, where we additionally highlight the response time and slowdown differences between best and worst performing users in the benchmark. We showcase the ECDFs of scheduling algorithm response times in Figure 7.13, separating the figures for default and dynamic partitioning, respectively.

Without runtime partitioning, Fair and UJF scheduling achieve a slightly lower average response time than other algorithms, but significantly lower worst 10 percent response time, UJF being able to reduce

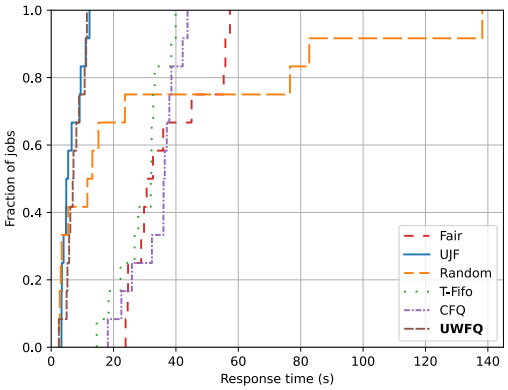


Figure 7.11: Empirical CDFs of one of the infrequent users in scenario 2.

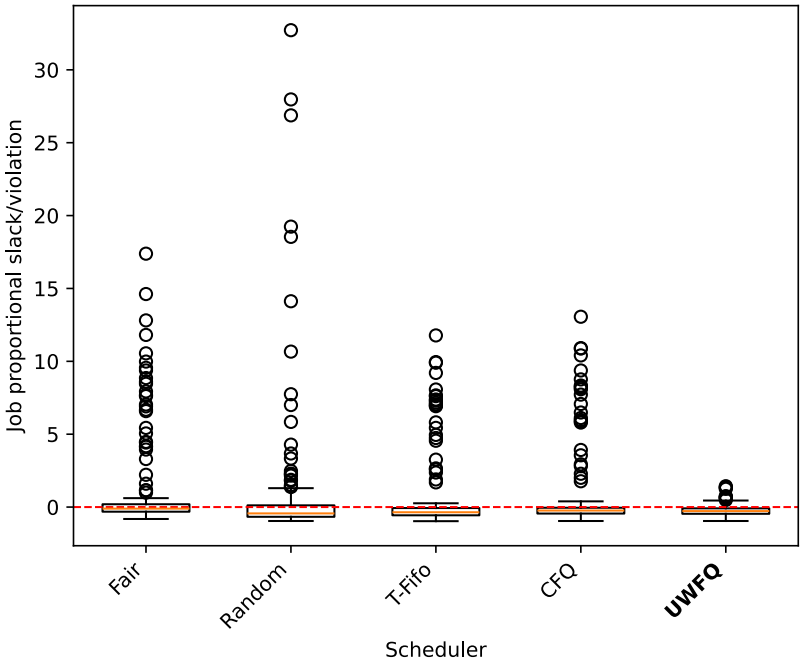


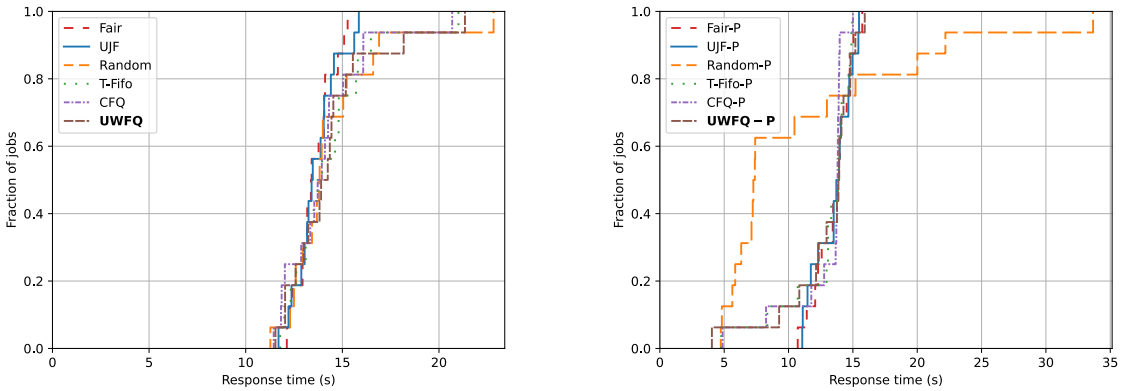
Figure 7.12: Box-plots of proportional deadline violations of jobs, where slacks are negative values and violations are positive. The value 0 indicates no improvement or degradation.

it by 24% compared to UWFQ. This again can be attributed to skews, which fair algorithms are better at dealing with in practice as seen in Figure 7.8. We additionally observe response time tails of CFQ and UWFQ in Figure 7.13 (a), which are the main contributors for the high worst 10 percent response time.

But when introducing runtime partitioning, UWFQ-P and CFQ-P can eliminate the tail response times, having almost equal performance to Fair-P and UJF-P. Additionally, we can observe in Figure 7.13 (b) that UWFQ-P and CFQ-P improve the response times of the first few jobs, but this improvement diminishes in subsequent jobs. This highlights that UWFQ-P can improve the response time of equally sized large jobs, but only if the system is not constantly congested by them. This can be explained by the fact that once a user finishes their long-running job, their next job has to wait for all the other users' long-running jobs to finish first, essentially creating the same response time every cycle. However, this response time benefit is only experienced by users who arrive earlier to the system, with late arriving users not receiving any response time benefit but also no significant delay, as visualized in Figure 7.14. Interestingly, Random-P achieves the best overall average response times, however, this is done by

Scheduler	Response time (s) | Slowdown				Fairness			
	Avg.	Worst 10%	Worst	Best	DVR	Count	DSR	Count
Fair	13.6 | 1.26	15.5 | 1.43	12.8 | 1.19	14.2 | 1.31	0.07	16	0.06	16
UJF	13.6 | 1.25	16.3 | 1.50	12.7 | 1.17	14.2 | 1.31	-	-	-	-
Random	14.3 | 1.32	22.6 | 2.08	13.5 | 1.25	15.4 | 1.42	0.24	28	0.09	4
T-Fifo	14.3 | 1.32	20.0 | 1.85	13.5 | 1.25	15.4 | 1.42	0.23	26	0.11	6
CFQ	14.1 | 1.30	20.8 | 1.92	13.0 | 1.20	15.2 | 1.40	0.17	29	0.11	3
UWFQ (this work)	14.2 | 1.32	21.3 | 1.97	13.0 | 1.20	15.4 | 1.42	0.21	29	0.11	3
Fair-P	13.5 | 1.25	15.3 | 1.42	12.9 | 1.20	13.9 | 1.28	0.06	17	0.05	15
UJF-P	13.4 | 1.24	15.3 | 1.41	12.6 | 1.16	13.8 | 1.28	-	-	-	-
Random-P	**11.6** | **1.07**	30.7 | 2.83	**9.48** | **0.87**	**13.4** | **1.24**	0.60	10	**0.58**	22
T-Fifo-P	12.8 | 1.18	15.5 | 1.43	11.5 | 1.06	13.9 | 1.29	**0.03**	2	0.33	30
CFQ-P	12.8 | 1.18	**15.1** | **1.39**	11.6 | 1.07	13.8 | 1.28	-	**1**	0.28	**31**
UWFQ-P (this work)	12.8 | 1.18	15.4 | 1.43	11.5 | 1.06	13.9 | 1.29	0.04	2	0.32	30

Table 7.6: Comparison of scheduler performance metrics for scenario 3.

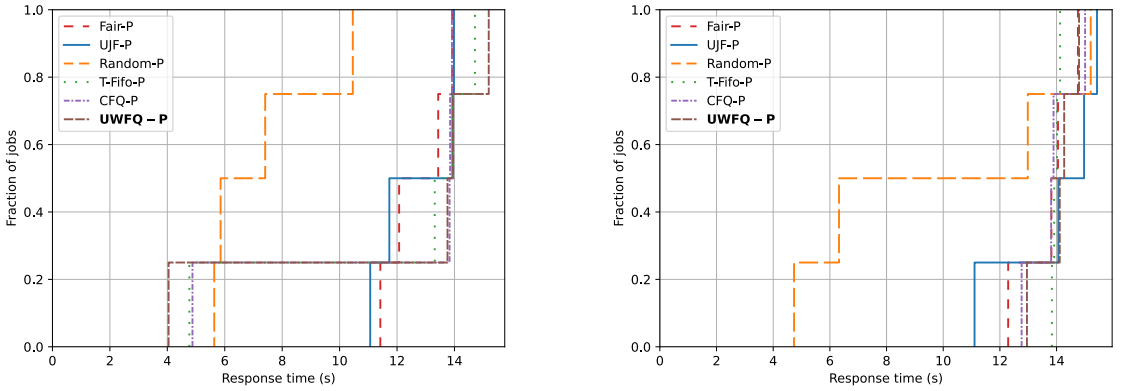


(a) Default partitioning

(b) Runtime partitioning

Figure 7.13: Empirical CDFs of schedulers for scenario 3. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.

severely violating other job deadlines.



(a) First arriving user

(b) Last arriving user

Figure 7.14: Empirical CDFs of best and worst users in scenario 3. (a) shows response time differences in the best user for UWFQ-P, while (b) shows response time differences in the worst user for UWFQ-P.

Examining the fairness metric values, we see that most algorithms that use default partitioning violate more deadlines than they bring slack, which can be more clearly observed in Figure 7.15. However,

when using dynamic partitioning, most algorithms are able to stay within the UJF deadlines, with CFQ-P, UWFQ-P and T-Fifo-P having the least amount of deadline violations, and providing much higher DSR values compared to Fair.

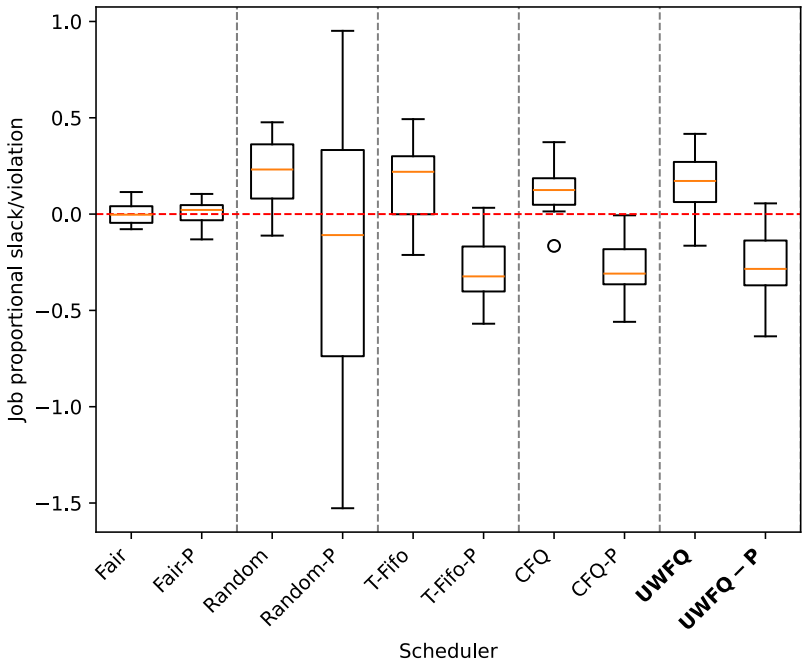


Figure 7.15: Box-plots of proportional deadline violations of jobs, where slacks are negative values and violations are positive. The value 0 indicates no improvement or degradation.

Scenario 4 results

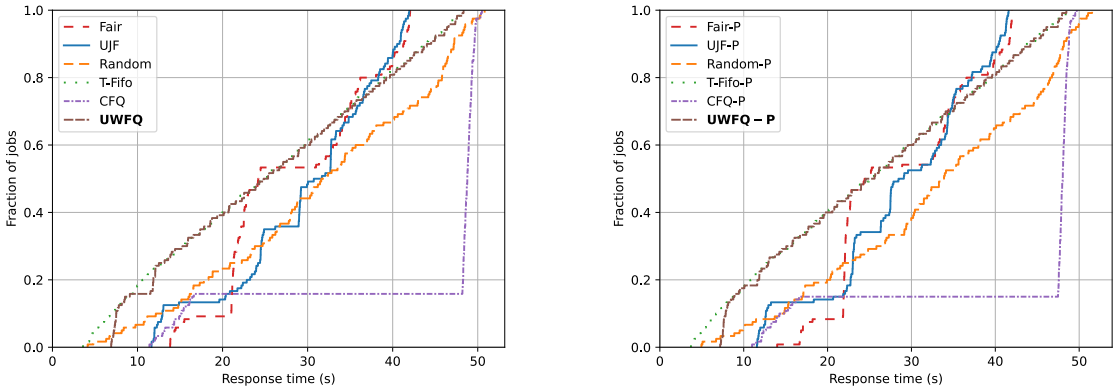
The values of key performance metrics are highlighted in Table 7.7, where we additionally address response times and slowdowns of first and last arriving users. Since all the jobs for this scenario are already small and well partitioned by the default partitioner, we do not include metrics from the schedulers using dynamic partitioning, since they do not show any significant changes. However, we do still showcase the ECDFs of scheduling algorithm response times for both partitioning approaches in Figure 7.16, so it can be visually observed.

Scheduler	Response time (s) | Slowdown				Fairness			
	Avg.	Worst 10%	First	Last	DVR	Count	DSR	Count
Fair	28.1 | 32.9	41.6 | 48.9	21.4 | 25.1	33.7 | 39.6	0.78	95	0.31	145
UJF	29.1 | 34.1	**41.3** | **48.5**	25.4 | 29.8	33.3 | 39.0	-	-	-	-
Random	32.2 | 37.8	49.1 | 57.7	29.9 | 35.0	33.7 | 39.6	0.66	141	0.35	99
T-Fifo	**25.1** | **29.5**	46.3 | 54.4	**9.36** | **10.9**	39.5 | 46.4	**0.39**	100	0.45	140
CFQ	43.2 | 50.7	49.5 | 58.1	36.7 | 43.0	47.3 | 55.5	0.80	211	**0.51**	29
UWFQ (this work)	25.5 | 29.9	46.5 | 54.6	15.8 | 18.5	**31.3** | **36.7**	0.50	94	0.43	146

Table 7.7: Comparison of scheduler performance metrics for scenario 4.

We see that the best average response time is achieved by UWFQ and T-Fifo, UWFQ decreasing the average response time by 13% compared to UJF. We attribute this improvement to improved job context information used both by UWFQ and T-Fifo, since they are able to more gradually complete the jobs, compared to other schedulers as seen in Figure 7.16. 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. However, we do note that this only happens in cases where jobs have a very tight arrival time interval, hence, this was not observed in scenario 2.

In this particular scenario, the fairness metrics are not as representative. This is due to direct completion



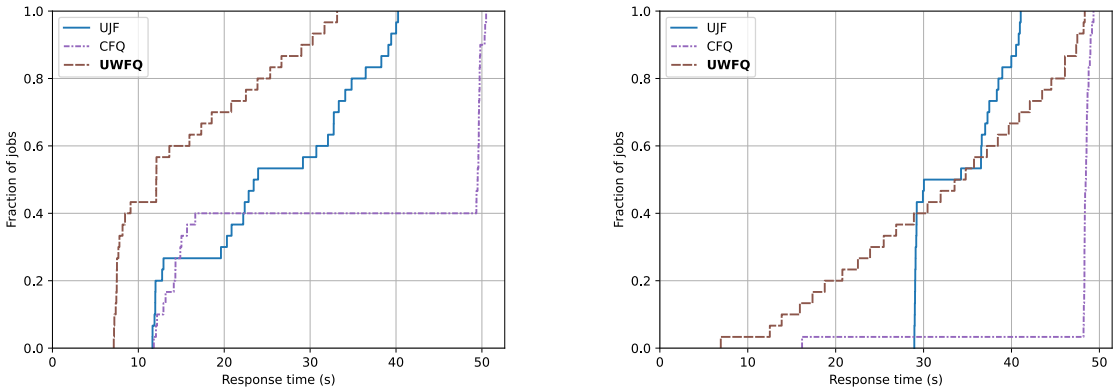
(a) Default partitioning

(b) Runtime partitioning

Figure 7.16: Empirical CDFs of schedulers for scenario 4. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.

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.

We observe that T-Fifo achieves the best performance for its first arriving user, however slows down its last arriving user more than other schedulers. We point out that UWFQ achieves the best performance for the last arriving user, having slightly faster response time than UJF, but achieves significant gains for its first arriving user, having average response time reduced by 38% compared to UJF. This is visually highlighted in Figure 7.17, where it can be seen that all jobs are more gradually completed under UWFQ compared to CFQ and UJF. We do note that the first arriving user has a much lower average response time than the last arriving user in the UWFQ scheduler, 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.



(a) First arriving user

(b) Last arriving user

Figure 7.17: Empirical CDFs of first and last arriving users in scenario 4. (a) shows response time differences in the first arriving user for UWFQ, while (b) shows response time differences in the last arriving user for UWFQ.

7.3. Macro-benchmark

Macro-benchmarks are crafted from real trace data, which gives a better representation of user interaction with a shared resource system. While there is currently no Apache Spark benchmark that is focused on

analyzing a multi-user and multi-job environment, we present a way to convert an existing system trace into a trace that can be executed within Apache Spark.

7.3.1. Setup

Macro-benchmarks were run under the same DAS-5 [4] system as micro-benchmarks, hence, the cluster, hardware, environment, and dataset details are shared between these benchmarks. However, the workloads are much different and based on existing trace data rather than synthetic scenarios.

Lack of existing traces

To accurately represent the target system, the traces need to at least contain the following meta-data:

1. **User context** - Each job can be attributed to a uniquely identifiable user, who is entitled to an equal share of the system resources in respect to other users.
2. **Job meta-data** - Jobs should have a submission time that indicates the time the job arrived rather than when it was scheduled, and the total duration it took to execute.
3. **Job types** - Jobs can be of any type that can be converted into either a job with a single task or multiple tasks.

Currently, since very little research has been done in the field of user fair scheduling, there is no benchmark trace that is widely used in existing research to measure the performance of batch processing schedulers that would fit our criteria. Popular benchmark suites such as TPC-H[5] or HiBench[6] do not contain any user context information that is necessary to represent the target scheduling environment. Most work on scheduling algorithms craft or generate their own benchmarks [12, 18, 19, 47, 11] or use simulation software [29, 28] to isolate analysis purely on the scheduler.

To tackle this problem, we convert existing system traces that contain the necessary multi-user and multi-job metadata into compatible Spark job execution traces. This method has already been used in previous work [13, 8, 10], and has been shown to be effective in emulating the workload for Apache Spark infrastructure.

Workloads

To compose our macro-benchmark, we use Google traces (2014) [23] originally taken from Google cluster usage traces [65] but standardized into Workflow Trace Archive (WTA) format by Versluis et al. [61]. WTA format allows scientists to more easily share and use existing trace data, reducing the work needed to convert traces for conducting experiments. We utilize the WTA format to extract the necessary data from Google traces and convert them to analogous Spark jobs.

To transform the traces, we iterate over the users present in the WTA format. For each user, we collect the start times of all their workflows and corresponding task metadata. Workflows consist of a DAG of tasks, where edges represent data dependencies between tasks. To calculate the total runtime of a workflow, we sum the individual task runtimes. For the following experiments, we do not consider data dependencies among tasks and treat workflows as a bundle of tasks scheduled all at the same time. Since the UWFQ and other schedulers do not account for workflow dependency structures, this should not have a severe impact on the findings.

Each task of a workflow is implemented as a Spark job that includes the necessary metadata to be properly associated with its corresponding workflow. To enable Spark jobs with elastic runtime, we dynamically generate a Spark user-defined function that will be executed by the Spark job, having a proportional execution time to the task it is imitating. To have representable runtimes, the task resource runtime is calculated by multiplying the task runtime by the resource amount requested.

Since the system under test is much different than the Google cluster traces that were collected, we scale task resource runtimes based on available cores and the target resource utilization we want to achieve. Having utilization beloew 80% gives some downtime for recovery, while aiming for 100% utilization will cause the system to always be congested with work.

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

7.3.2. Selected trace

Since the entire Google trace spans a period of one month, we sample a small subset of that period to evaluate our proposed scheduler. We do this by picking a period of 500 seconds within the trace, and scaling the tasks within that period to reach a certain utilization threshold. A larger time interval than 500 seconds was not selected due to hardware limitations when analyzing results. However, we believe that 500 seconds is a sufficient period to analyze, and the analysis should not be severely changed by extending the timeframe.

While traces are good for mimicking user behavior patterns, the trace workloads are not necessarily representative of patterns that exist in the target system. For example, the vast majority of jobs in the Google trace run within 1 resource second, while multiple outliers can run for more than 500s resource seconds. This essentially only creates scenarios where resource competition happens between at most 2 users, one running a long-running job, while another quickly joins the system to perform short-running jobs.

For this reason, we manually select and slightly refine the traces to create more appropriate workloads that we target to solve, while still keeping the realistic user arrival times and behavior. We craft two different workload types we will be analyzing and measuring: heterogeneous and homogeneous workloads.

Heterogeneous workload

This workload represents the most common scenarios occurring in the system, where jobs of different runtimes compete for shared system resources. We use the workload to highlight the importance of considering job runtime while scheduling, and its impact on response times and fairness.

To craft this, we go over multiple 500s periods and select a timeframe if within it most jobs execute within 1 resource second, and multiple jobs take more than 20 resource seconds to execute. Due to the traces containing a lot of extremely long-running jobs that go beyond 100 resource seconds, we filter them out to not scale the entire workload around a few outliers dominating the resource runtime.

To represent the workload, we select the trace that spans between 76'630'000 ms and 77'130'000ms in the original trace. We filter out extremely long jobs and scale the rest to achieve around 80% theoretical resource utilization. We represent the filtered trace timeline in Figure 7.18. We can see in Figure 7.19 that the majority of jobs finish within 1 resource second, a significant amount of jobs that finish within 20 seconds, and some outliers that run for more than 20 seconds. By examining the job and user execution theoretical overlap in Figure 7.20, we can see that for a significant period of the benchmark, the resource competition occurs between multiple users and multiple jobs.

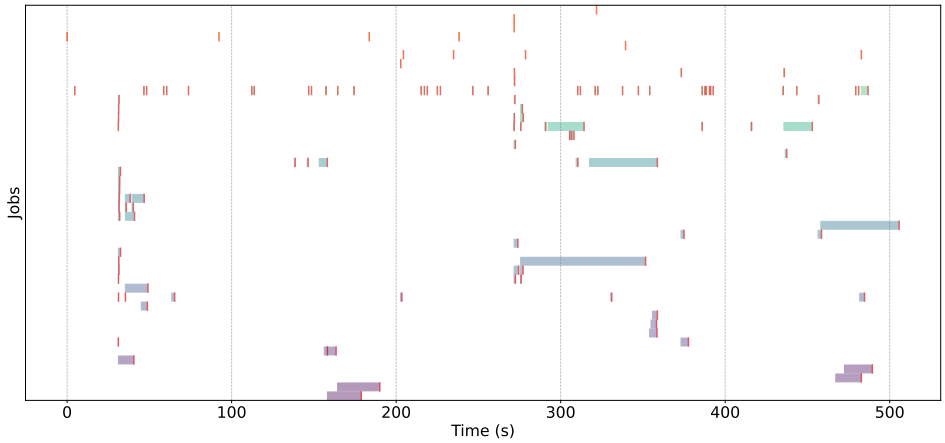


Figure 7.18: Job arrival timeline of heterogeneous macro-benchmark. The length of each job equates to expected runtime, however may significantly differ due to system congestion. For each job, a color is associated with it to indicate its corresponding user, but we do note that the small jobs are too small to notice their user association.

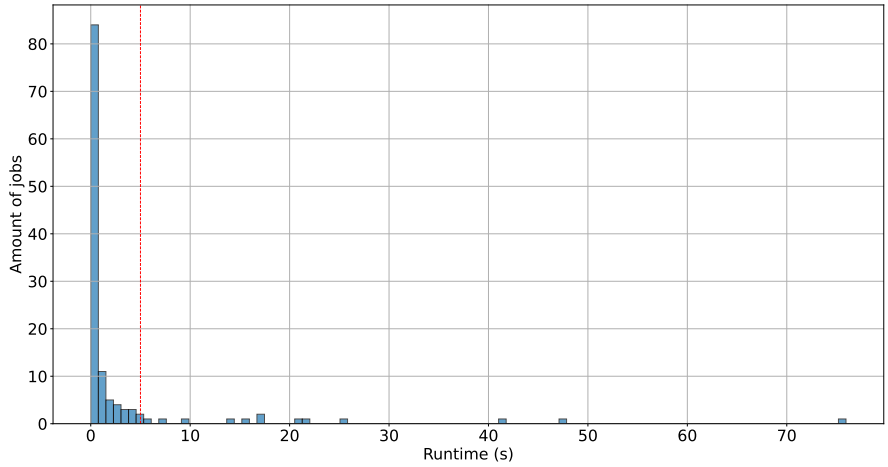


Figure 7.19: Histogram of job runtimes in heterogeneous macro-benchmark. We put a red line to separate jobs that run within 5 seconds and jobs that run longer than 5 seconds.

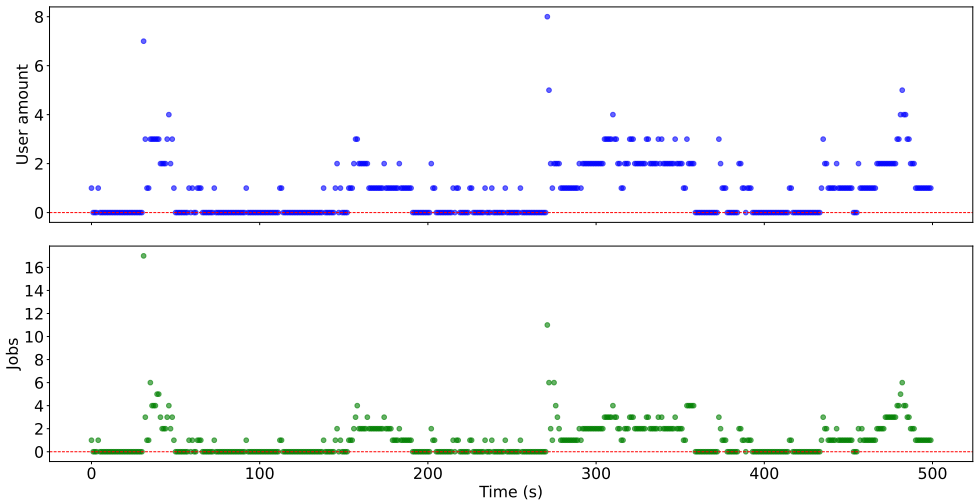


Figure 7.20: User and job theoretical congestion timelines in heterogeneous macro-benchmark. The top plot shows the congestion between users over time, while the bottom plot shows congestion between jobs over the same period.

Homogeneous workload

Homogeneous workload contains multiple users competing for shared resources with jobs of similar runtime, where the resource time is dominated by bursts of jobs, rather than long-running jobs. We craft this workload, since this is the main use case where user context is necessary for user fair scheduling. For example, if one user schedules 100 small jobs, they should not be receiving 100 times more resources than users with only a single active job. This benchmark emphasizes scenarios that may occur in the target system, where users with bursty workloads gain proportionally more resources than infrequent users, and additionally highlights the drawbacks of schedulers that do not consider user context, for example, the built-in Fair scheduler and CFQ [10].

We craft this workload by analyzing multiple 500-second periods and choosing an extremely congested period of similar runtime jobs. Due to the original trace resource time being severely dominated by long-running jobs, we remove any job that runs 10 times longer than the median before filtering. This allows us to analyze resource congestion among similarly short-running jobs, rather than the interaction between long jobs or between long and short jobs.

We select the trace that spans between 1'473'800'000ms and 1'474'300'000ms in the original trace. 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 set 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. We represent the filtered trace timeline in Figure 7.21. We can see in Figure 7.22 that the job runtimes are much more evenly spread compared to the heterogeneous benchmark. By examining the job and user execution theoretical overlap in Figure 7.23, there are many more periods where users are directly competing for resources. While there are noticeable gaps in the congestion graph, we note that this is because the theoretical graph does not account for slowdowns induced by resource sharing, which we expect to cause significant delays due to higher expected resource utilization.

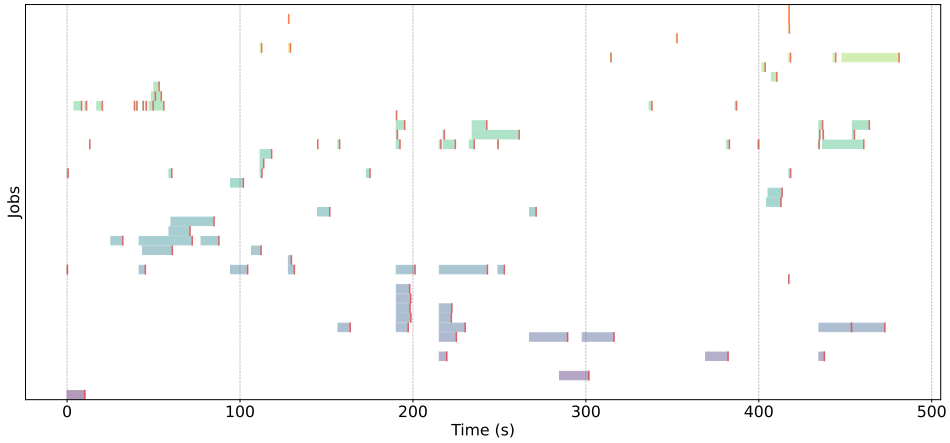


Figure 7.21: Job arrival timeline of homogeneous macro-benchmark. The length of each job equates to expected runtime, however may significantly differ due to system congestion. For each job, a color is associated with it to indicate its corresponding user, but we do note that the small jobs are too small to notice their user association.

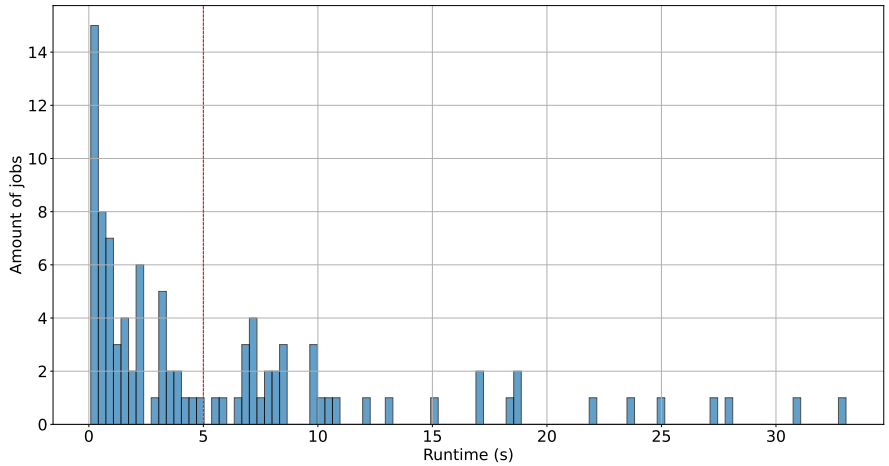


Figure 7.22: Histogram of job runtimes in homogeneous macro-benchmark. We put a red line to separate jobs that run within 5 seconds and jobs that run longer than 5 seconds.

7.3.3. Results

In this section, we analyze both scenarios and measure the performance of UWFQ compared to other schedulers. We exclude Random and True FIFO from this analysis, since they perform poorly in real

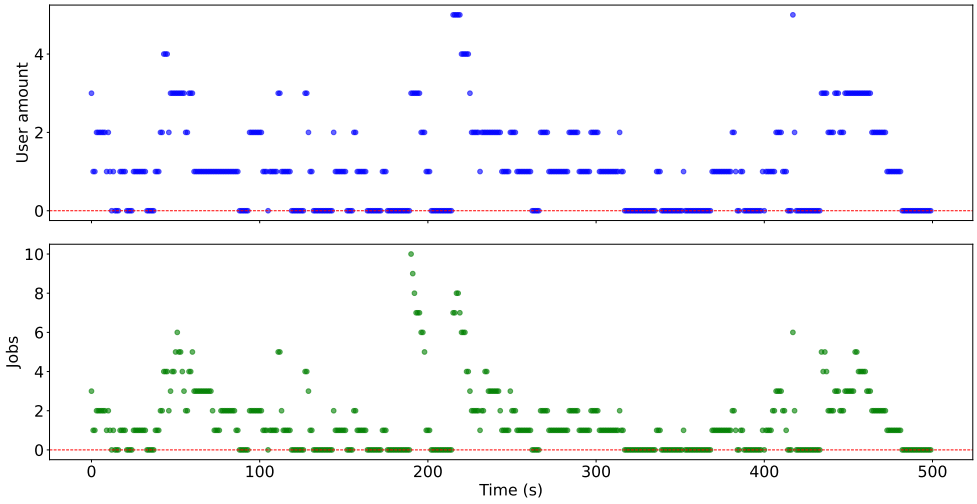


Figure 7.23: User and job theoretical congestion timelines in homogeneous macro-benchmark. The top plot shows the congestion between users over time, while the bottom plot shows congestion between jobs over the same period.

settings. To highlight only the most important aspects, most collected metrics are omitted from the analysis. Full tables and traces can be viewed in Appendix D.

Heterogeneous benchmark

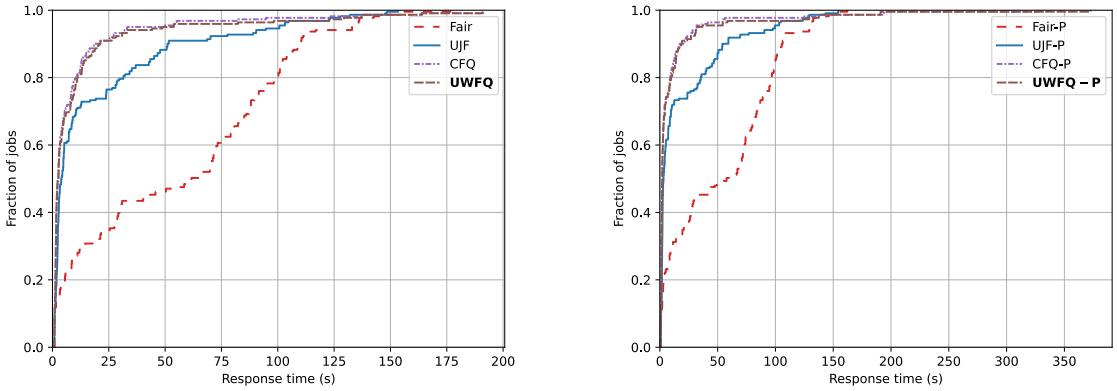
We show the collected metrics in Table 7.8, where we also highlight job response times categorized by their response time percentiles. Specifically, we group the jobs into the first 80th percentile, the next 15th percentile (medium-sized jobs), and the final 5th percentile. We showcase the ECDFs of all job response times in Figure 7.29.

Scheduler	Runtime	Response time (s)				Fairness			
		Avg.%	0–80%	80–95%	95–100%	DVR	Count	DSR	Count
Fair	642.8	55.43	40.03	106.0	142.0	10.6	192	0.28	29
UJF	620.6	18.99	5.57	53.27	121.6	-	-	-	-
CFQ	647.8	10.44	2.81	17.87	**101.9**	**0.44**	69	**0.49**	152
UWFQ (this work)	648.8	11.56	2.91	19.29	117.2	0.48	71	0.48	150
Fair-P	628.4	53.27	37.97	103.7	138.8	11.5	187	0.27	34
UJF-P	**620.2**	18.73	5.29	53.53	120.1	-	-	-	-
CFQ-P	654.6	**10.34**	**2.31**	**16.63**	110.8	0.60	**52**	0.48	**169**
UWFQ-P (this work)	657.8	11.64	2.37	17.88	130.4	0.60	59	0.48	162

Table 7.8: Comparison of scheduler performance metrics for heterogeneous benchmark.

The benchmark is set to have about 80% theoretical resource congestion, however, it may vary due to real system conditions and variance in true runtime. We can additionally observe that the longest-running jobs appear more during the later parts of the benchmark, partially explaining why most schedulers finish the benchmark at least 100 seconds after new jobs have stopped arriving. With this in mind, we still observe that UJF and Fair schedulers can complete the benchmarks faster than CFQ and UWFQ, where UJF is able to complete almost 30 seconds faster than UWFQ. We believe this overall throughput improvement stems from Fair and UJF algorithms tending to allocate multiple tasks of the same stage to the same executors. Since executors have already warmed up by executing previous tasks of the same stage, the following tasks of that stage are expected to complete much faster than running them on a cold executor [8]. However, the observed relative throughput improvement of UJF is at most 5%, which can be negligible compared to the response time improvements UWFQ or CFQ can bring.

We see that both CFQ and UWFQ are able to significantly reduce the average response time of jobs by 45% and 40% compared to UJF, as can be seen in Figure 7.24(a). By analysing the percentile average response times, we see that overall response time improvements come from allowing small and medium

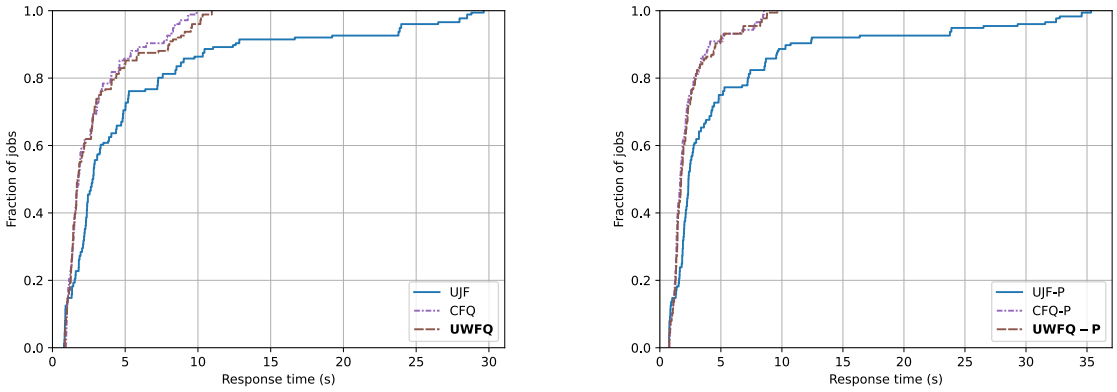


(a) Default partitioning

(b) Runtime partitioning

Figure 7.24: Empirical CDFs of schedulers for heterogeneous macro-benchmark. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.

jobs to execute faster, as seen in Figure 7.25(a) and Figure 7.26(a), while only slightly increasing the runtimes of longer jobs. For the 80th percentile, we see that CFQ and UWFQ reduce the response times by 50% and 48% respectively, compared to UJF. For jobs between the 80th and 95th percentiles, we see the most major improvements, where CFQ and UWFQ reduce the response times by 67% and 64% respectively. And lastly, we see that there is a significant long job response time reduction seen in CFQ and a minor reduction in UWFQ, with 16% and 4% reduction compared to UJF, with the outliers being observed in Figure 7.27(a).



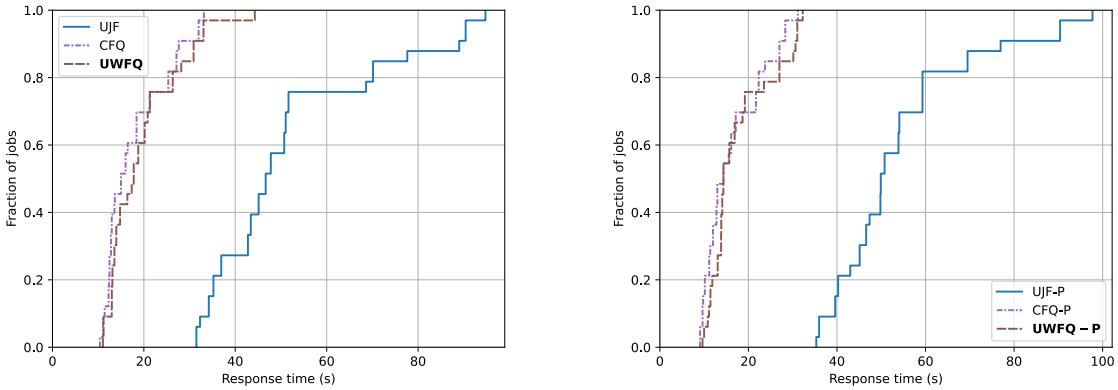
(a) Default partitioning

(b) Runtime partitioning

Figure 7.25: Empirical CDFs of schedulers for homogeneous macro-benchmark, isolating the 80th percentile of response times. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.

The small job runtime improvement is to be expected and aligns with findings of micro-benchmarks. However, the runtime of medium-sized jobs is substantially larger, which was not initially expected. We attribute this finding to the fact that most medium-sized jobs were scheduled around the same time, and for a fair scheduler that tries to run them all simultaneously, it does the interleaving of tasks, which overall reduces the performance of all concurrently running medium-sized jobs.

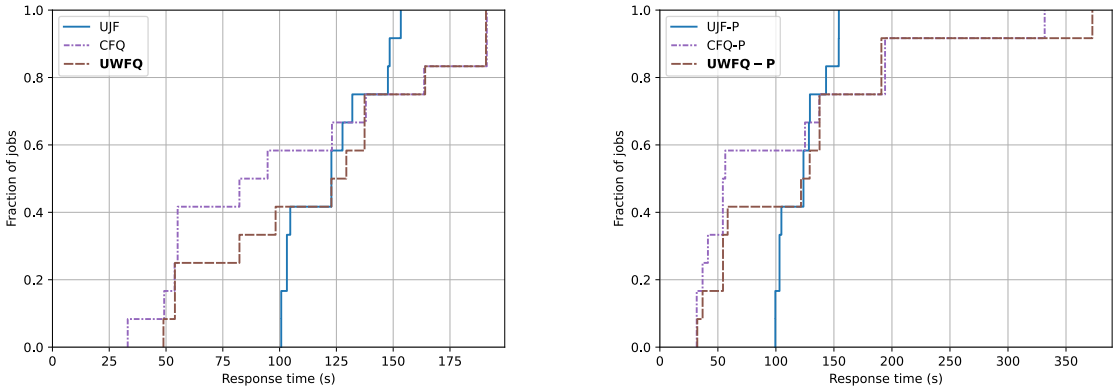
When examining the fairness of CFQ and UWFQ, their DVR and DSR values are close to equal, with CFQ having slightly better average values and a lower amount of violations than UWFQ, as can also be seen in Figure 7.28. While we expect UWFQ to be more fair than CFQ, because the jobs vairy significantly in sizes, CFQ and UWFQ tend to converge to a similar schedule, since job runtime has more impact in this scenario than user context.



(a) Default partitioning

(b) Runtime partitioning

Figure 7.26: Empirical CDFs of schedulers for homogeneous macro-benchmark, isolating the 80th to 95th percentile of response times. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.



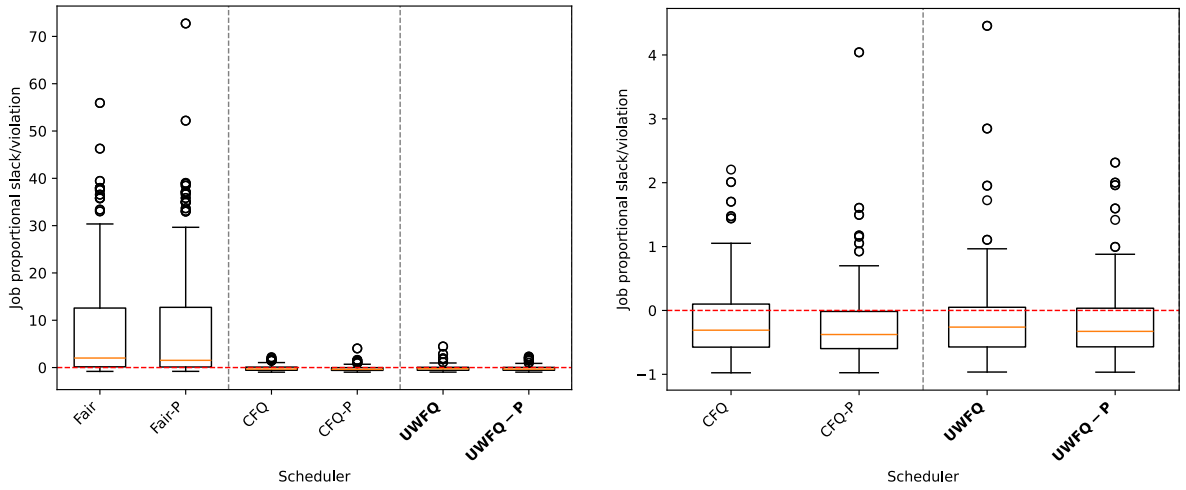
(a) Default partitioning

(b) Runtime partitioning

Figure 7.27: Empirical CDFs of schedulers for heterogeneous macro-benchmark, isolating 95th till 100th percentile of response times. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.

Lastly, we examine the effects of runtime partitioning for this algorithm. We can see that for Fair and UJF the effects of runtime partitioning are negligible. However, for CFQ-P and UWFQ-P it impacts the response time of small and medium jobs. While the average response time of CFQ-P and UWFQ-P is unaffected by partitioning, it does reduces the response time of small jobs by 18% and 19% when comparing to their default partitioning counterparts respectively. A smaller impact is observed for medium jobs, where the response time is reduced by 7% and 8% respectively. Unfortunately, this does increase the response times of larger jobs, where CFQ-P increases it by 9% and UWFQ-P by 11%.

We note that there is a significant increase in longer job runtimes in this benchmark for UWFQ when comparing to UJF, and we believe it is caused by two main factors. First is the inaccuracies in runtime estimates, which could cause skews in virtual time, and allow faster jobs to claim resources that were supposed to be allocated to long-running jobs. Second is the unfairness that is present in UJF's underlying implementation of the Fair scheduler, where it has a slight preference for running jobs that arrived first, rather than jobs that have historically had the least amount of resources allocated. Because of that, long-running jobs may gain more resources over time, allowing them to finish execution faster than they would under a theoretical UJF scheduler. This would mean that UWFQ is more fair than practical UJF, however, we do not have enough evidence to prove which is the most contributing factor that would explain the runtime differences of longer running jobs.



(a) All schedulers

(b) Isolated schedulers

Figure 7.28: Box-plots of proportional deadline violations of jobs, where slacks are negative values and violations are positive. Figure (a) shows all scheduler boxplots, while (b) shows CFQ and UWFQ isolated to better examine the difference.

Homogeneous benchmark

We show the collected metrics in Table 7.9, where we also highlight job response times categorized by their response time percentiles. Specifically, we group the jobs into the first 80th percentile, the next 15th percentile (medium-sized jobs), and the final 5th percentile. We showcase the ECDFs of all job response times in Figure 7.29.

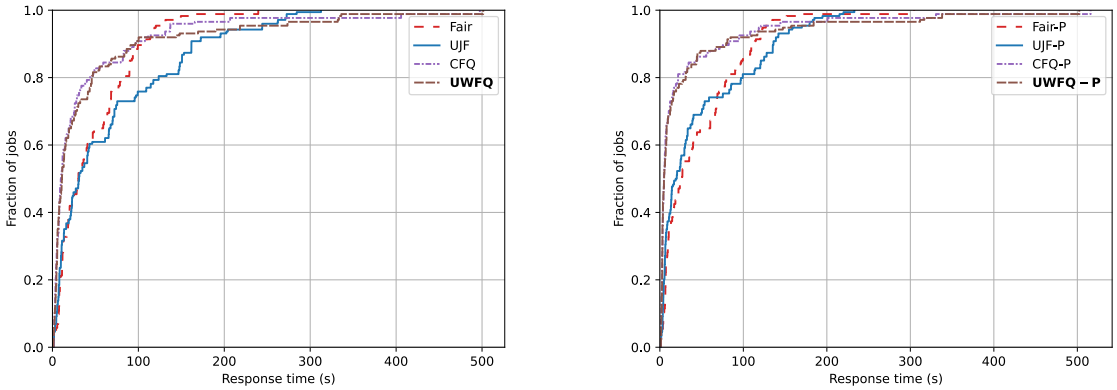
Scheduler	Runtime	Response time (s)				Fairness			
		Avg.%	0–80%	80–95%	95–100%	DVR	Count	DSR	Count
Fair	563.1	46.12	28.29	100.5	**164.4**	1.00	83	0.42	91
UJF	565.7	54.12	27.43	140.9	215.5	-	-	-	-
CFQ	621.0	37.05	11.05	85.67	298.1	0.55	30	0.52	144
UWFQ (this work)	624.2	41.42	12.33	92.36	343.5	**0.44**	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

Table 7.9: Comparison of scheduler performance metrics for homogeneous benchmark.

These benchmarks create a workload that is a bit above 100% resource utilization, which explains why it takes longer than 500 seconds to execute. UJF and Fair can 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 comparsion to UJF. We attribute this throughput slowdown to long-running tasks, which have been shown in previous experiments to slow down both CFQ and UWFQ, since running them in parallel has been shown to be more efficient. However, the slowdown goes down to 5% when introducing runtime partitioning, which is the same slowdown that was observed in heterogeneous benchmarks, and most likely caused by JVM warmup [8].

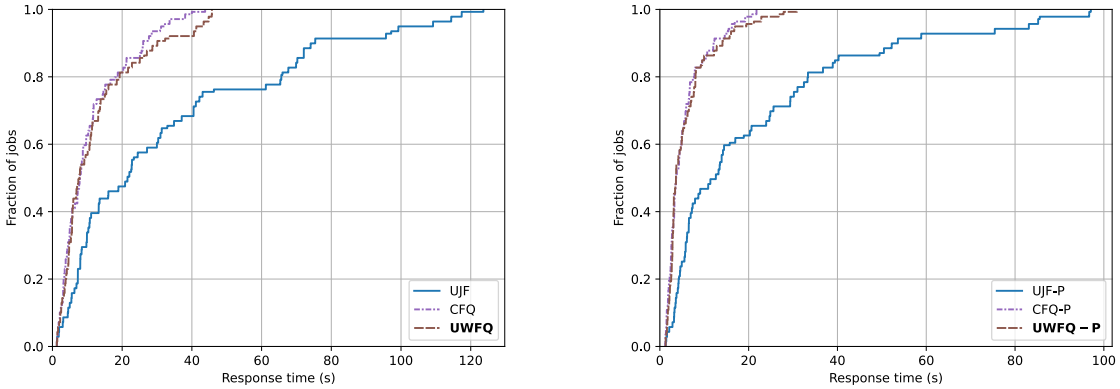
Both CFQ and UWFQ achieve the the best overall response time, which is attributed to the significant speedups seen for small and medium length jobs, as seen in Figure 7.30(a) and Figure 7.31(a). CFQ and UWFQ improve the average response time by 32% and 24% compared to UJF, respectively. For jobs in the 80th percentile, the response time is decreased by 60% and 55% compared to UJF, and for medium jobs it's 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, as is highlighted in Figure 7.32(a).

In this specific benchmark, we additionally see that Fair slightly outperforms UJF average response times by 15%, and significantly reduces medium and long running job response time by 29% and 24%



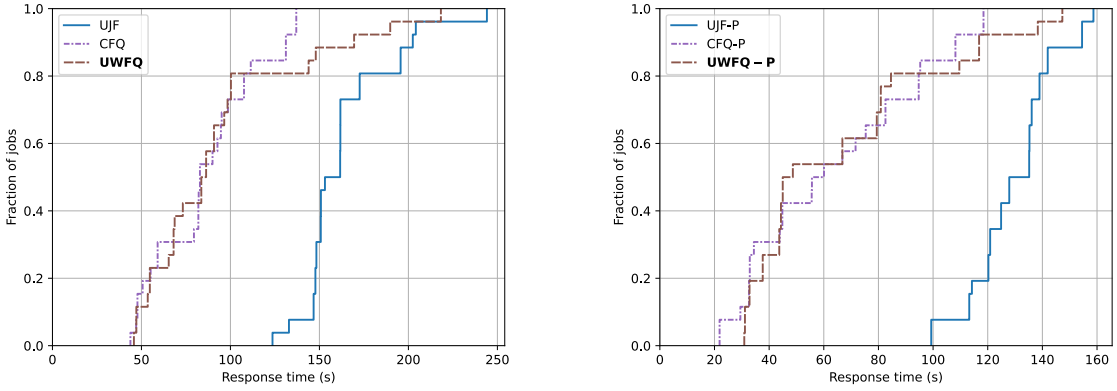
(a) Default partitioning **(b)** Runtime partitioning

Figure 7.29: Empirical CDFs of schedulers for homogeneous macro-benchmark. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.



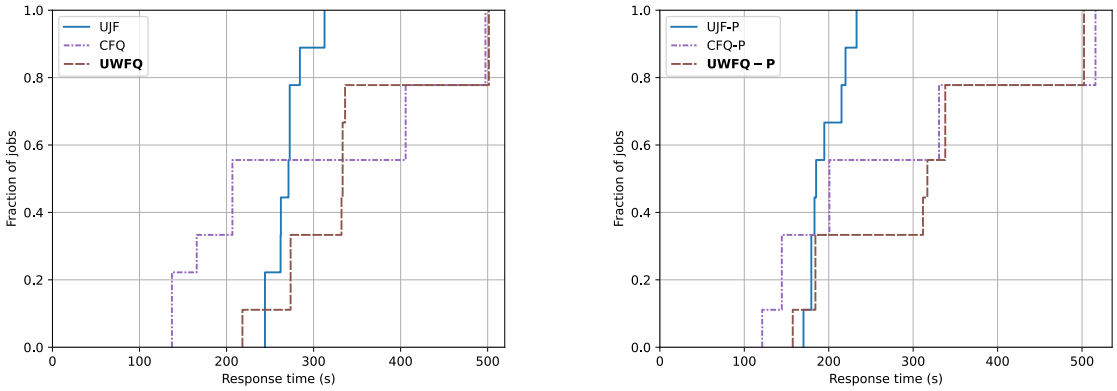
(a) Default partitioning **(b)** Runtime partitioning

Figure 7.30: Empirical CDFs of schedulers for homogeneous macro-benchmark, isolating the 80th percentile of response times. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.



(a) Default partitioning **(b)** Runtime partitioning

Figure 7.31: Empirical CDFs of schedulers for homogeneous macro-benchmark, isolating the 80th till 95th percentile of response times. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.



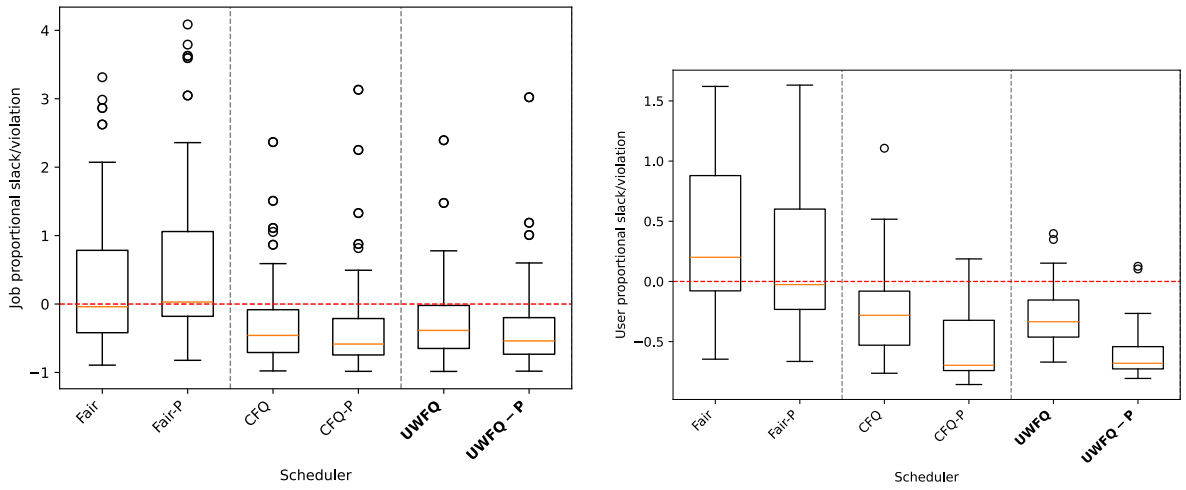
(a) Default partitioning

(b) Runtime partitioning

Figure 7.32: Empirical CDFs of schedulers for homogeneous macro-benchmark, isolating 95th till 100th percentile of response times. (a) showcases schedulers with regular partitioning, while (b) compares schedulers with runtime partitioning.

respectively. This can be attributed to the fact that the Fair algorithm is not fair regarding users, allowing some jobs to finish faster by violating fairness, which is highlighted by the high DVR value and violation count.

When examining fairness, we see that UWFQ has a slightly lower DVR value than CFQ, but CFQ has slightly higher DSR value, which can be observed in Figure 7.33(a) as UWFQ has less outliers but worse mean. However, for this benchmark we further analyse the average response times of users to determine fairness differences. We visualize this in Figure 7.33(b) 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.



(a) Boxplot of jobs

(b) Boxplot of users

Figure 7.33: Box-plots of proportional deadline violations of jobs and users, where slacks are negative values and violations are positive. Figure (a) shows boxplots of all job slacks and violations, while (b) shows slacks and violations of all users.

Runtime partitioning for 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 80th percentile by 76% and 74% compared to UJF-P, and lowering the next 15th percentile by 55% and 52%, which is highlighted in Figure 7.30(b) and Figure 7.31(b). While CFQ and UWFQ still increases the runtimes of long jobs, partitioning slightly decreases them to 32% and 46% compared to UJF-P. While CFQ achieves much

better response times for medium jobs, UWFQ compensates by decreasing the runtimes of the last 5th percentile, as observed in Figure 7.32(b). Lastly, partitioning additionally makes the algorithms slightly more fair, 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 7.33(b).

For homogeneous benchmarks, the runtime improvements are much more substantial than heterogeneous benchmarks. We attribute this to the fact that homogeneous workloads contain a lot more even spread of job runtimes, inducing collisions of many small and medium-sized jobs. Fair scheduling algorithm response times do not suffer as much from heterogeneous environments, since resources are only split between a limited amount of jobs, but in a system where there are multiple competing similar length jobs, the fair algorithms have a much worse performance while interleaving all of them, as is supported by the results. However, due to CFQ and UWFQ preference for smaller jobs, they tend to induce longer response times for long jobs. While this slowdown affects only a small portion of jobs, it can still cause noticeable delays. We believe this can be mitigated by having much more accurate runtime estimates.

7.4. Multi-user industry benchmarks

For this project, we set out to test our UWFQ scheduler under real industry workload to gather performance metrics on actual user submitted Spark jobs. However, during performance evaluation it was identified that a software bug in the existing system prevents the Spark executors from reaching the necessary levels of saturation for the scheduler to significantly effect the target metrics. This made the system almost indifferent to the underlying scheduling algorithm, where hardware variance had a larger impact than any of the tested schedulers.

This issue was made aware to the responsible developers, and has not been addressed by the time of writing this report. We hope this can be addressed in future work.

7.5. Discussions

In this section, we briefly go over the main findings that we observe across all benchmarks. We first go over the main improvements and caveats of UWFQ, then discuss the benefits that runtime partitioning can introduce, and finally, perform meta-analysis on the implementation of UWFQ.

7.5.1. User Weighted Fair Queuing

The experiments have shown that UWFQ can improve job response time while keeping within reasonable bounds of UJF. We discuss these improvements in more detail.

Response time improvements

In heterogeneous job environments, we can see that a similar performance is observed between CFQ and UWFQ. We believe the main reason for this is that both algorithms order jobs based on their virtual deadlines, instead of interleaving between all active jobs. This gives more preference to shorter jobs instead of long-running jobs, allowing them to execute and finish much quicker than running multiple jobs in parallel. This creates schedules that are extremely similar to Shortest-Job-First (SJF) scheduling, which is shown to achieve the best average response times [46]. A similar finding is documented in the paper presenting CFQ scheduling [10].

One of the main improvements of UWFQ over CFQ is adding job context to Spark scheduling, which improves job completion times by grouping stages of the same job together, rather than treating them as separate scheduling entities. This allows jobs consisting of multiple stages to run consecutively until finishing, rather than interleaving with other concurrent stages, effectively introducing the same interleaving as seen in fair schedulers. This finding is best highlighted in micro-benchmark scenario 4 in Subsection 7.2.3.

Fairness improvements

To the best of our knowledge, UWFQ is the only scheduler that implements bounded UJF while still optimizing for runtimes and ensuring real-time performance. The main drawback of CFQ is its lack of regard for user context, which creates scenarios where users who have more jobs receive proportionally

more resources. This is what UWFQ solves by introducing more layers of virtual time, to ensure that UJF bounds are kept while still providing similar response time benefits as CFQ. This can be clearly observed in micro-benchmark scenario 2 in Subsection 7.2.3.

While we have already implemented the UJF scheduler that achieves UJF fairness, we highlight the response time benefits that UWFQ brings in comparison to the UJF scheduler, which are substantial in most scenarios. We also see that the default Fair scheduler underperforms under most scenarios, hence the majority of the analysis was done to compare UWFQ with UJF or CFQ.

7.5.2. Runtime partitioning

Runtime partitioning has been shown to significantly improve scheduling performance in certain scenarios by both reducing the skew of larger jobs and allowing smaller jobs to receive deserved resources much faster. The default partitioning algorithm does not account for runtimes of jobs, and distributes input data using simple user-defined heuristics that only account for data size rather than runtime. While in many cases the runtime of jobs is relative to the size of the data, this is not always the case.

In cases where some partitions have more algorithmic work to perform within their partition, a large skew can be created that bottlenecks the entire runtime of the job. Runtime partitioning partially solves this by introducing much more aggressive partitioning based on runtime, which allows for large, skewed tasks to be split into smaller partitions, which is highlighted in micro-benchmark scenario 1 in Subsection 7.2.3.

While we only observe some improvement for small and medium job response times in heterogeneous macro-benchmark in Subsection 7.3.3, we do see significant response time reduction in homogeneous macro-benchmark for all job types. The lack of response time improvement in the heterogeneous case can be explained by many different task sizes already mixing well in the schedule, reducing the benefits brought by runtime partitioning. However, we see substantial improvements in cases where job runtimes are more homogeneous, which we believe solves the scenarios where jobs may have been delayed by longer-running jobs that were occupying the executors. Since in both cases we do not observe significant degradation of system performance, we believe that runtime partitioning can only improve scheduling performance if the system can support accurate runtime estimation before execution of tasks.

7.5.3. Meta analysis of UWFQ

UWFQ has slightly higher response times than CFQ in heterogeneous workloads. We believe there are 2 main explanations for this. First, UWFQ tries to account more explicitly for user fairness. This puts slightly more limitations on allowing quicker jobs to finish first, if the user submitting the respective job is not entitled to the resources. Second is that runtime estimation is much more granular for CFQ since it works in the context of stages rather than jobs, which is slightly more beneficial for this type of benchmark due to avoiding prediction errors that accumulate when working with the entire job context. Job context has to account for in-between-stage delays, which for user jobs of many stages can be significant.

We observe that Fair and UJF perform some benchmarks noticeably faster in macro-benchmarks. We estimate that this throughput difference can be caused by fair algorithms better handling JVM warm-up by allocating jobs on the same cores. It has been shown in previous literature [8] that JVM warm-up can significantly increase the runtime of jobs, and tasks from previously executed stages benefit from cache locality and preloaded classes, allowing cores to execute them more efficiently than tasks from unseen jobs. Due to UWFQ and CFQ maximizing parallelism, many tasks are executed on cold JVM environments, where the delay can add up and slightly reduce overall throughput.

We recognize that the runtime of scheduling for UWFQ can scale significantly faster than CFQ or other fair schedulers, however, for target workloads where the concurrent job amount does not exceed 100, the scheduling time is sufficiently low. While we do not perform specific measurements on the time needed for assigning a deadline to a job and scheduling it, we do examine the log statements and see that the scheduling decision can be made within 1 millisecond majority of times. Large skews in scheduling time seem to only occur when the scheduling thread is preempted by another process, causing the scheduling to be paused by the underlying operating system. For our current experiments, we do not

observe any significant scheduling delay caused by UWFQ.

8

Conclusion

In this project, we presented the User Weighted Fair Queuing (UWFQ) scheduling algorithm, which can minimize average response times of jobs in the system while still ensuring that users and their jobs all receive a proportionally equal amount of service. We additionally introduce a runtime partitioning algorithm, which can reduce the impact of long-running tasks on fairness and job response times by partitioning the jobs into finer amounts.

We implement 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, we are able to reduce average response times of small jobs by up to 78% in homogeneous workloads, while still ensuring that most jobs are completed within the fairness boundaries. We additionally show that UWFQ is much 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.

We additionally summarize the answers to the defined research questions.

RQ1 *How can fairness be defined in batch processing environments where multiple users can schedule one or multiple jobs simultaneously?*

We define User-Job Fairness (UJF) as an extension of fair share scheduling, where resources are distributed equally first among users, and then among jobs. This allows each user to have an isolated amount of resources, ensuring that they are receiving an equal service as any other user, and allowing each of user's job to gradually progress without starvation. We additionally define bounded UJF as any schedule in which jobs complete within a constant time bound relative to their completion time under ideal UJF scheduling.

RQ2 *What scheduling approaches decrease the average response time of jobs while ensuring fairness?*

We find that Weighted Fair Queuing (WFQ) and Cluster Fair Queuing (CFQ) are the most applicable scheduling algorithms for real-time fair scheduling, providing close to optimal response times and bounding maximum job execution time. However, these algorithms only define fairness on the level of jobs, rather than users, which does not ensure UJF fairness we are trying to achieve. To solve this, we extend the CFQ scheduling algorithm with User Weighted Fair Queuing (UWFQ), which introduces another layer of fairness to account for user contexts. We then show during evaluation that UWFQ is able to achieve a similar response time performance compared to CFQ while ensuring a more consistent performance in critical scenarios.

RQ3 *How does the partitioning of jobs in batch processing frameworks affect the average response time and fairness of a schedule?*

In this work, we define runtime partitioning that splits jobs into partitions that run in constant time, allowing for tasks to be distributed more evenly and reducing the maximum amount of time a task can reserve an execution unit. We do this by first estimating the runtime of the following job, and then splitting it into tasks such that the average task runtime does not pass a certain maximum threshold.

We compare our partitioning algorithm with the built-in Apache Spark partitioner that only uses input sizes and static user configurations to divide jobs into multiple partitions. The results show that runtime partitioning is able to reduce task skews in workloads and avoid priority inversions that are present in default Spark partitioning, further improving the response times of jobs and making fairness more consistent.

8.1. Future work

We discuss some parts of the work that were not finished due to time limitations or were too out of scope to explore within this project.

Improvements neglected due to limited time

Since the target system allows scheduling of workflows, a workflow scheduling layer can be introduced in the scheduler to speed up the completion times of workflows. We believe that even a similar implementation to HEFT [57] can yield significant performance benefits and would further improve the average response times of jobs.

Due to confidentiality restrictions imposed by ASML, we are not allowed to use user traces to analyze the efficiency of our scheduling algorithm with more representative user interactions. We hope that in the future, traces from similar systems can be made publicly available for more thorough evaluation of UWFQ.

Complementary improvements to explore

When it comes to batch processing frameworks, there are many avenues that can be improved to speed up the overall performance of the system. For systems that are limited by network capacity during burst periods, the coflow completion time minimization problem can be studied and applied to reduce the overall makespans of jobs [2].

Data locality was not explored in this work, and Apache Spark's native task locality policy was utilized. Having more sophisticated data locality aware scheduling strategies have shown to increase throughput [32], which we leave as future work to explore.

Chen et al. [8] show that jobs within the Spark framework exhibit different resource elasticity patterns. A similar observation is made by D. Cheng et al. [16] for Spark streaming. For certain jobs, allocating more resources can start diminishing the rates of speedup [8], and having a scheduler that accounts for dynamic parallelism can increase scheduling efficiency. We believe that for future work, these methods can be complementary to UWFQ, and further increase its efficiency.

References

[1] Sassan Ahmadi. "Chapter 6 - The IEEE 802.16m Medium Access Control Common Part Sub-layer (Part I)". In: *Mobile WiMAX*. Ed. by Sassan Ahmadi. Oxford: Academic Press, 2011, pp. 169–279. ISBN: 978-0-12-374964-2. DOI: https://doi.org/10.1016/B978-0-12-374964-2.10006-2. URL: https://www.sciencedirect.com/science/article/pii/B9780123749642100062.

[2] Afaf Arfaoui et al. "ELITE: Near-Optimal Heuristics for Coflow Scheduling". In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2022, pp. 665–674. DOI: 10.1109/CCGrid54584.2022.00076.

[3] Michael Armbrust et al. "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. ISBN: 9781450327589. DOI: 10.1145/2723372.2742797. URL: https://doi.org/10.1145/2723372.2742797.

[4] Henri Bal et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *Computer* 49.5 (2016), pp. 54–63. DOI: 10.1109/MC.2016.127.

[5] Mutaz Barika et al. "Online Scheduling Technique To Handle Data Velocity Changes in Stream Workflows". In: *IEEE Transactions on Parallel and Distributed Systems* 32.8 (2021), pp. 2115–2130. DOI: 10.1109/TPDS.2021.3059480.

[6] Emile Cadorel, Hélène Coullon, and Jean-Marc 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. DOI: 10.1109/CCGrid49817.2020.00-36.

[7] Danilo Carastan-Santos et al. "One Can Only Gain by Replacing EASY Backfilling: A Simple Scheduling Policies Case Study". In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2019, pp. 1–10. DOI: 10.1109/CCGRID.2019.00010.

[8] Chen Chen, Wei Wang, and Bo 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. DOI: 10.1109/INFOCOM.2018.8486026.

[9] Chen Chen, Wei Wang, and Bo 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. DOI: 10.1109/ICDCS.2017.174.

[10] Chen Chen et al. "Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees". In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8056948.

[11] Huangke Chen et al. "Uncertainty-Aware Online Scheduling for Real-Time Workflows in Cloud Service Environment". In: *IEEE Transactions on Services Computing* 14.4 (2021), pp. 1167–1178. DOI: 10.1109/TSC.2018.2866421.

[12] Li Chen et al. "Scheduling jobs across geo-distributed datacenters with max-min fairness". In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 2017, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8056949.

[13] Wei Chen, Xiaobo Zhou, and Jia Rao. "Preemptive and Low Latency Datacenter Scheduling via Lightweight Containers". In: *IEEE Transactions on Parallel and Distributed Systems* 31.12 (2020), pp. 2749–2762. DOI: 10.1109/TPDS.2019.2957754.

[14] Wei Chen et al. "Adaptive multiple-workflow scheduling with task rearrangement". In: *J. Supercomput.* 71.4 (Apr. 2015), pp. 1297–1317. ISSN: 0920-8542. DOI: 10.1007/s11227-014-1361-0. URL: https://doi.org/10.1007/s11227-014-1361-0.

[15] Wei Chen et al. “Characterizing Scheduling Delay for Low-Latency Data Analytics Workloads”. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 630–639. DOI: 10.1109/IPDPS.2018.00072.

[16] Dazhao Cheng et al. “Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.12 (2018), pp. 2672–2685. DOI: 10.1109/TPDS.2018.2846234.

[17] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[18] Hamza Djigal et al. “IPPTS: An Efficient Algorithm for Scientific Workflow Scheduling in Heterogeneous Computing Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (2021), pp. 1057–1071. DOI: 10.1109/TPDS.2020.3041829.

[19] Raphaël Ferreira da Silva, Tristan Glatard, and Frédéric 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. DOI: 10.1002/cpe.3303. URL: https://hal.science/hal-01016491.

[20] Eric Gaussier et al. “Improving backfilling by using machine learning to predict running times”. In: *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–10. DOI: 10.1145/2807591.2807646.

[21] Ali Ghodsi et al. “Dominant resource fairness: fair allocation of multiple resource types”. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI’11. Boston, MA: USENIX Association, 2011, pp. 323–336.

[22] César Gómez-Martín, Miguel A. Vega-Rodríguez, and José-Luis González-Sánchez. “Fattened backfilling”. In: *J. Parallel Distrib. Comput.* 97.C (Nov. 2016), pp. 69–77. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2016.06.013. URL: https://doi.org/10.1016/j.jpdc.2016.06.013.

[23] Google. *Workflow Trace Archive Google trace*. https://zenodo.org/record/3254540. Zenodo. June 2019. DOI: 10.5281/zenodo.3254540.

[24] Robert Grandl et al. “Multi-resource packing for cluster schedulers”. In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 455–466. ISSN: 0146-4833. DOI: 10.1145/2740070.2626334. URL: https://doi.org/10.1145/2740070.2626334.

[25] Andrea Gulino et al. “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. DOI: 10.1109/MASCOTS50786.2020.9285944.

[26] Herodotos Herodotou et al. “Starfish: A Self-tuning System for Big Data Analytics”. In: Jan. 2011, pp. 261–272.

[27] Zhiming Hu et al. “Job Scheduling without Prior Information in Big Data Processing Systems”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 572–582. DOI: 10.1109/ICDCS.2017.105.

[28] Alexey Ilyushkin and Dick 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. DOI: 10.1109/CCGRID.2018.00048.

[29] Alexey Ilyushkin, Bogdan Ghit, and Dick 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. ISBN: 9781479980062. DOI: 10.1109/CCGrid.2015.27. URL: https://doi.org/10.1109/CCGrid.2015.27.

[30] Zhao-hong JIA et al. “A novel cloud workflow scheduling algorithm based on stable matching game theory”. In: *The Journal of Supercomputing* 77 (Oct. 2021), pp. 1–28. DOI: 10.1007/s11227-021-03742-3.

[31] J. Kay and P. Lauder. “A fair share scheduler”. In: *Commun. ACM* 31.1 (Jan. 1988), pp. 44–55. ISSN: 0001-0782. DOI: 10.1145/35043.35047. URL: https://doi.org/10.1145/35043.35047.

[32] Chunlin Li, Jing Zhang, and Hengliang Tang. "Replica-aware task scheduling and load balanced cache placement for delay reduction in multi-cloud environment". In: *The Journal of Supercomputing* 75 (May 2019). DOI: 10.1007/s11227-018-2695-9.

[33] Feng Li et al. "Clustering-based multi-objective optimization considering fairness for multi-workflow scheduling on clouds". In: *J. Parallel Distrib. Comput.* 194.C (Dec. 2024). ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2024.104968. URL: https://doi.org/10.1016/j.jpdc.2024.104968.

[34] Jiexing Li et al. "Robust estimation of resource consumption for SQL queries using statistical techniques". In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1555–1566. ISSN: 2150-8097. DOI: 10.14778/2350229.2350269. URL: https://doi.org/10.14778/2350229.2350269.

[35] Tong Li, Dan Baumberger, and Scott 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. ISBN: 9781605583976. DOI: 10.1145/1504176.1504188. URL: https://doi.org/10.1145/1504176.1504188.

[36] D Lifka, M Henderson, and K Rayl. *EASY-DOS. Extensible Argonne SP Scheduling sYstem*. Tech. rep. Argonne National Lab. (ANL), Argonne, IL (United States), May 1995. URL: https://www.osti.gov/biblio/253207.

[37] Raquel V. Lopes and Daniel Menascé. "A Taxonomy of Job Scheduling on Distributed Computing Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 27.12 (2016), pp. 3412–3428. DOI: 10.1109/TPDS.2016.2537821.

[38] Ryan Marcus and Olga Papaemmanouil. "Plan-structured deep neural network models for query performance prediction". In: *Proc. VLDB Endow.* 12.11 (July 2019), pp. 1733–1746. ISSN: 2150-8097. DOI: 10.14778/3342263.3342646. URL: https://doi.org/10.14778/3342263.3342646.

[39] Alexandre Maros et al. "Machine Learning for Performance Prediction of Spark Cloud Applications". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 99–106. DOI: 10.1109/CLOUD.2019.00028.

[40] A.W. Mu'alem and D.G. Feitelson. "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling". In: *IEEE Transactions on Parallel and Distributed Systems* 12.6 (2001), pp. 529–543. DOI: 10.1109/71.932708.

[41] Saydul Akbar Murad et al. "SG-PBFS: Shortest Gap-Priority Based Fair Scheduling technique for job scheduling in cloud environment". In: *Future Gener. Comput. Syst.* 150.C (Jan. 2024), pp. 232–242. ISSN: 0167-739X. DOI: 10.1016/j.future.2023.09.005. URL: https://doi.org/10.1016/j.future.2023.09.005.

[42] J. Nagle. "On Packet Switches with Infinite Storage". In: *IEEE Transactions on Communications* 35.4 (1987), pp. 435–438. DOI: 10.1109/TCOM.1987.1096782.

[43] A.K. Parekh and R.G. 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. DOI: 10.1109/90.234856.

[44] Mario Pastorelli et al. "HFSP: Bringing Size-Based Scheduling To Hadoop". In: *IEEE Transactions on Cloud Computing* 5.1 (2017), pp. 43–56. DOI: 10.1109/TCC.2015.2396056.

[45] Adrian Daniel Popescu et al. "PREDIcT: towards predicting the runtime of large scale iterative analytics". In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 1678–1689. ISSN: 2150-8097. DOI: 10.14778/2556549.2556553. URL: https://doi.org/10.14778/2556549.2556553.

[46] Bui The Quang et al. "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. DOI: 10.1109/CCGrid.2015.123.

[47] Amin Rezaeian, M. Naghibzadeh, and D. Epema. "Fair multiple-workflow scheduling with different quality-of-service goals". In: *The Journal of Supercomputing* 75 (Feb. 2019). DOI: 10.1007/s11227-018-2604-2.

[48] Apoorv Saxena et al. "Analysis of the age of data in data backup systems". In: *Computer Networks* 160 (2019), pp. 41–50. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2019.05.020. URL: https://www.sciencedirect.com/science/article/pii/S1389128618308727.

[49] Hani Al-Sayeh and Kai-Uwe 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. DOI: 10.1109/ICDEW.2019.00-23.

[50] Mehrnoosh Shafiee and Javad Ghaderi. "On Max-Min Fairness of Completion Times for Multi-Task Job Scheduling". In: *2020 IFIP Networking Conference (Networking)*. 2020, pp. 100–108.

[51] S. Srinivasan et al. "Characterization of backfilling strategies for parallel job scheduling". In: *Proceedings. International Conference on Parallel Processing Workshop*. 2002, pp. 514–519. DOI: 10.1109/ICPPW.2002.1039773.

[52] I. Stoica et al. "A proportional share resource allocation algorithm for real-time, time-shared systems". In: *17th IEEE Real-Time Systems Symposium*. 1996, pp. 288–299. DOI: 10.1109/REAL.1996.563725.

[53] Ji Sun and Guoliang Li. "An end-to-end learning-based cost estimator". In: *Proc. VLDB Endow.* 13.3 (Nov. 2019), pp. 307–319. ISSN: 2150-8097. DOI: 10.14778/3368289.3368296. URL: https://doi.org/10.14778/3368289.3368296.

[54] D. Talby and D.G. Feitelson. "Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling". In: *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*. 1999, pp. 513–517. DOI: 10.1109/IPPS.1999.760525.

[55] Shanjiang Tang et al. "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. DOI: 10.1109/ICDE55515.2023.00316.

[56] 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. DOI: 10.1109/INFCOM.2002.1019322.

[57] H. Topcuoglu, S. Hariri, and Min-You Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), pp. 260–274. DOI: 10.1109/71.993206.

[58] Shivaram Venkataraman et al. "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. ISBN: 9781931971294.

[59] Laurens Versluis and Alexandru Iosup. "A survey of domains in workflow scheduling in computing infrastructures: Community and keyword analysis, emerging trends, and taxonomies". In: *Future Generation Computer Systems* 123 (2021), pp. 156–177. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.04.009. URL: https://www.sciencedirect.com/science/article/pii/S0167739X21001308.

[60] Laurens Versluis and Alexandru Iosup. "A survey of domains in workflow scheduling in computing infrastructures: Community and keyword analysis, emerging trends, and taxonomies". In: *Future Generation Computer Systems* 123 (2021), pp. 156–177. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.04.009. URL: https://www.sciencedirect.com/science/article/pii/S0167739X21001308.

[61] Laurens Versluis et al. "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. DOI: 10.1109/TPDS.2020.2984821. URL: https://doi.org/10.1109/TPDS.2020.2984821.

[62] Guolu Wang et al. "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. DOI: 10.1109/CCGRID.2018.00096.

[63] Kewen Wang and Mohammad Maifi Hasan 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. DOI: 10.1109/HPCC-CSS-ICESS.2015.246.

[64] Kewen Wang et al. “Modeling Interference for Apache Spark Jobs”. In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, pp. 423–431. DOI: 10.1109/CLOUD.2016.0063.

[65] John 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.

[66] Wentao Wu et al. “Towards predicting query execution time for concurrent and dynamic database workloads”. In: *Proc. VLDB Endow.* 6.10 (Aug. 2013), pp. 925–936. ISSN: 2150-8097. DOI: 10.14778/2536206.2536219. URL: https://doi.org/10.14778/2536206.2536219.

[67] Yulai Yuan et al. “Guarantee Strict Fairness and UtilizePrediction Better in Parallel Job Scheduling”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.4 (2014), pp. 971–981. DOI: 10.1109/TPDS.2013.88.

[68] Matei Zaharia et al. “Apache Spark: a unified engine for big data processing”. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[69] He Zhang, Muhammad Ali Babar, and Paolo Tell. “Identifying relevant studies in software engineering”. In: *Information and Software Technology* 53.6 (2011). Special Section: Best papers from the APSEC, pp. 625–637. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2010.12.010.

[70] Lihua 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. DOI: 10.1109/ICCCS57501.2023.10150636.

A

Search queries

```
    query1 = ("""
SELECT * FROM publications
    WHERE venue IN %s
    AND year >= %s
    AND (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)

    AND ((lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s))

    AND ((lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s))

    AND NOT ((lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s)
    OR (lower(title) ILIKE %s OR lower(abstract) ILIKE %s))

 ;
""",
[tuple(venues),
start_year,
'%schedul%', '%schedul%',

'%workflow%', '%workflow%',
'%batch%', '%batch%',
'%spark%', '%spark%',
'%backfilling%', '%backfilling%',

'%fair%', '%fair%',
'%slowdown%', '%slowdown%',
'%response%', '%response%',
'%throughput%', '%throughput%',
'%shared%', '%shared%',

'%deep%', '%deep%',
'%dnn%', '%dnn%',
'%gpu%', '%gpu%',
'%cuda%', '%cuda%',
'%bandwidth%', '%bandwidth%',
'% edge%', '% edge%',
```

```
'% fog%', '% fog%',
'% tcp%', '% tcp%',

])
```

B

Spark benchmark configurations

SparkConf

```
    "spark.driver.cores": 8,
    "spark.cores.max": "32",
    "spark.executor.memory": "4g",

    "spark.eventLog.enabled":  true,
    "spark.ui.retainedJobs": "100000",
    "spark.ui.retainedStages": "100000",
    "spark.ui.retainedTasks": "10000000",

    "spark.sql.adaptive.enabled": true,
    "spark.sql.adaptive.coalescePartitions.enabled": true,
    "spark.sql.adaptive.coalescePartitions.parallelismFirst": false,
    "spark.sql.adaptive.coalescePartitions.minPartitionSize": "20mb",
    "spark.sql.adaptive.advisoryPartitionSizeInBytes": "100mb",

    "spark.sql.files.maxPartitionBytes": "800mb"
```

spark-defaults.conf

```
    spark.master                     spark://__MASTER__:7077
    spark.driver.memory              60g
    spark.executor.instances         8
    spark.executor.cores             4
    spark.executor.memory            60g # this is inital memory, which is then overriden by
        Spark Config to 4gb

    # Event enabling event logging
    spark.eventLog.enabled           true
    spark.eventLog.dir               /var/scratch/__USER__/eventlogs/
    spark.history.fs.logDirectory    /var/scratch/__USER__/eventlogs/

    spark.ui.retainedJobs       100000
    spark.ui.retainedStages    100000
    spark.ui.retainedTasks    10000000
```

C

Micro-benchmark scenario extra data

Full tables of the runs can be found on the GitHub[1] repository, we do not include them due to difficulty of condensing large tables into A4 layout.

We show all the micro-benchmark traces of all the schedulers. The figures are split into two separate graphs: user job timeline (bottom) and task allocation on executors (top). Each color indicates the corresponding user that the job or task belongs to. This is done to identify job interactions between users competing for the same resources. The top graph displays each system core and the tasks assigned to it over time. The bottom graph represents jobs as rectangles, indicating their execution periods over time.

C.1. Scenario 1

The execution traces of schedulers can be seen in Figure C.1.

C.2. Scenario 2

The execution traces of schedulers can be seen in Figure C.2.

C.3. Scenario 3

The execution traces of schedulers can be seen in Figure C.3.

C.4. Scenario 4

The execution traces of schedulers can be seen in Figure C.4.

[1] Full run tables: https://github.com/kazemaksOG/spark-benchmark-tool/tree/master/results/tables

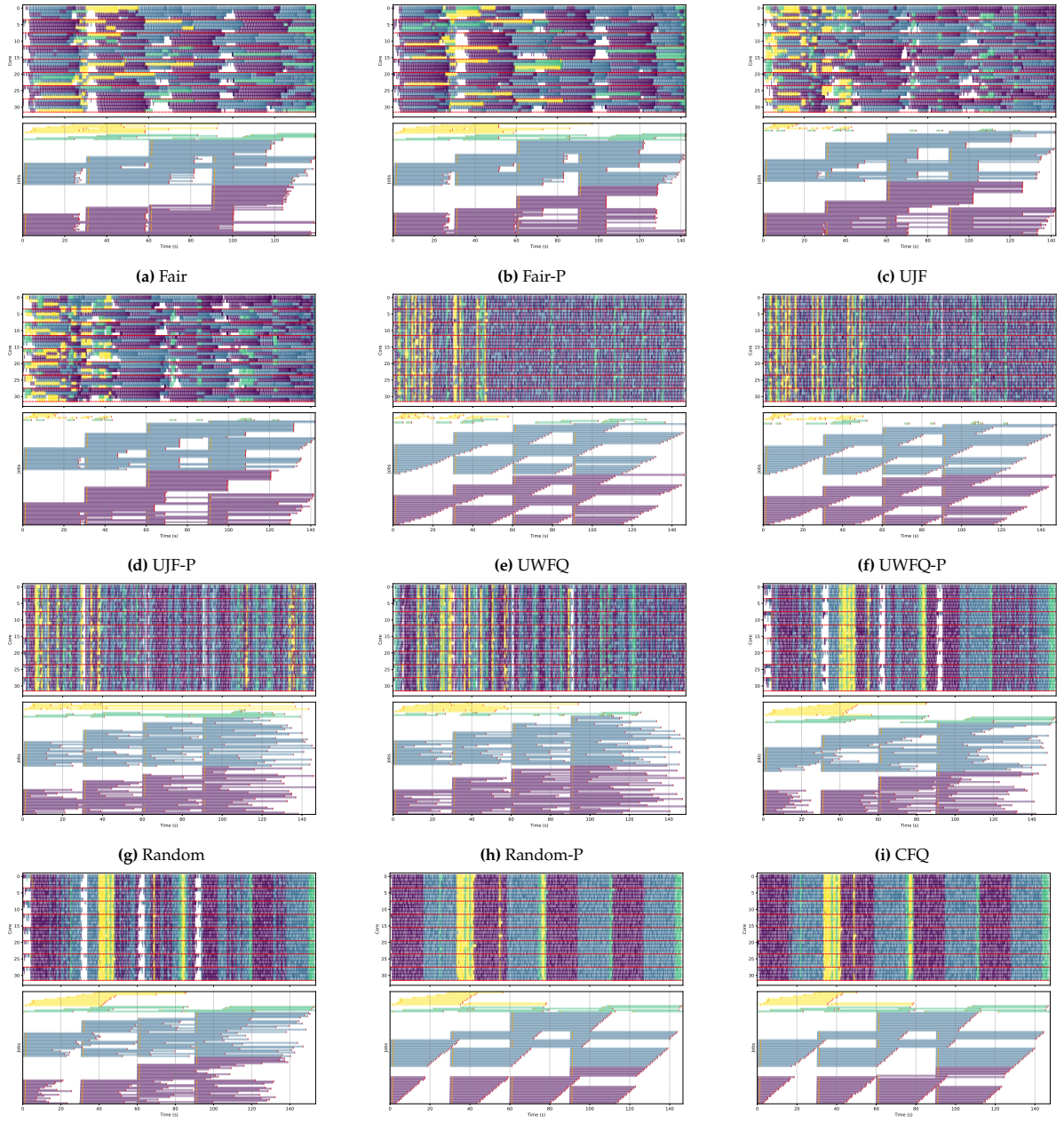
(a) Fair **(b)** Fair-P **(c)** UJF

(d) UJF-P **(e)** UWFQ **(f)** UWFQ-P

(g) Random **(h)** Random-P **(i)** CFQ

(j) CFQ-P **(k)** T-Fifo **(l)** T-Fifo-P

Figure C.1: Scheduler traces for scenario 1.



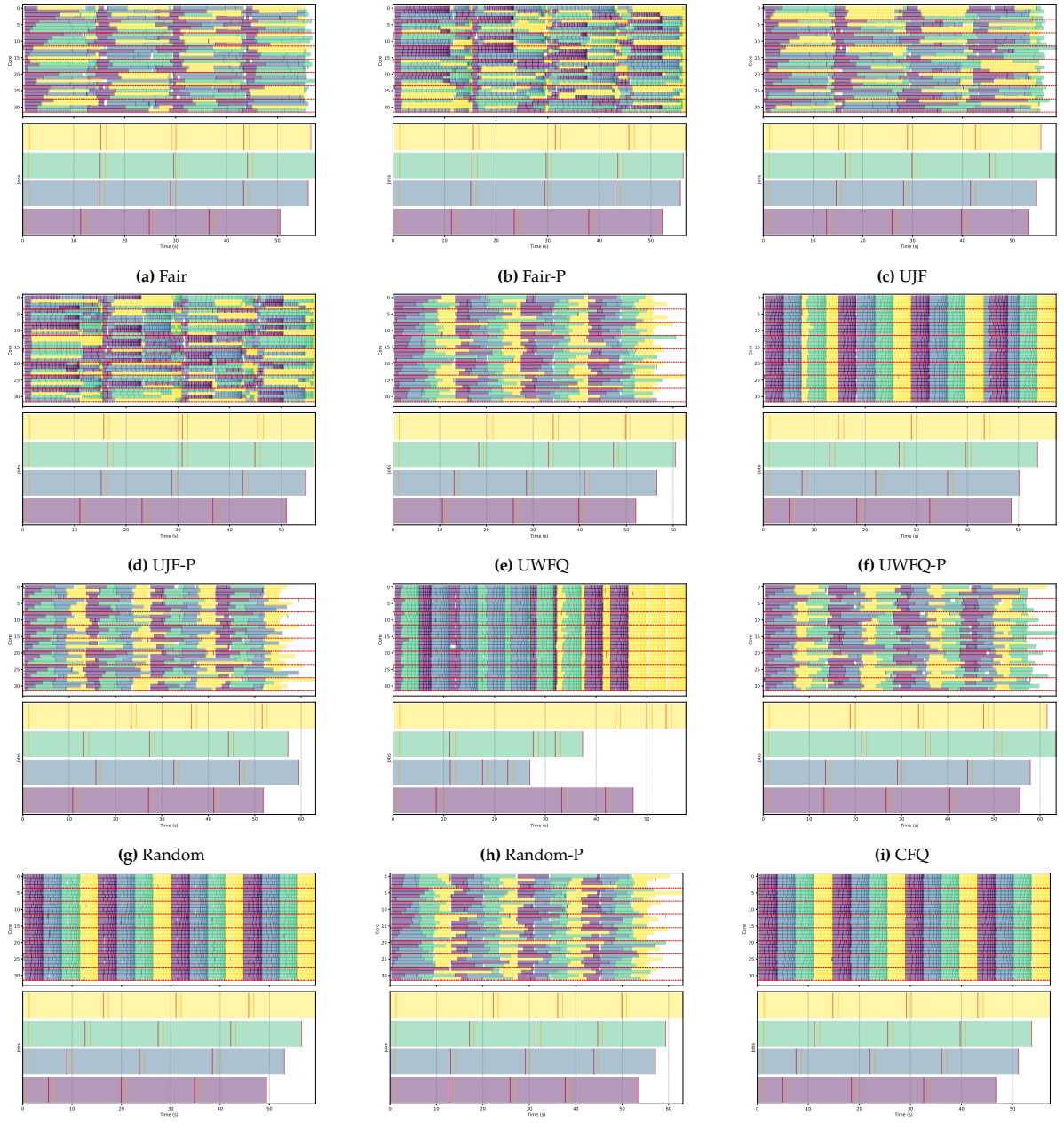
(a) Fair **(b)** Fair-P **(c)** UJF

(d) UJF-P **(e)** UWFQ **(f)** UWFQ-P

(g) Random **(h)** Random-P **(i)** CFQ

(j) CFQ-P **(k)** T-Fifo **(l)** T-Fifo-P

Figure C.2: Scheduler traces for scenario 2.



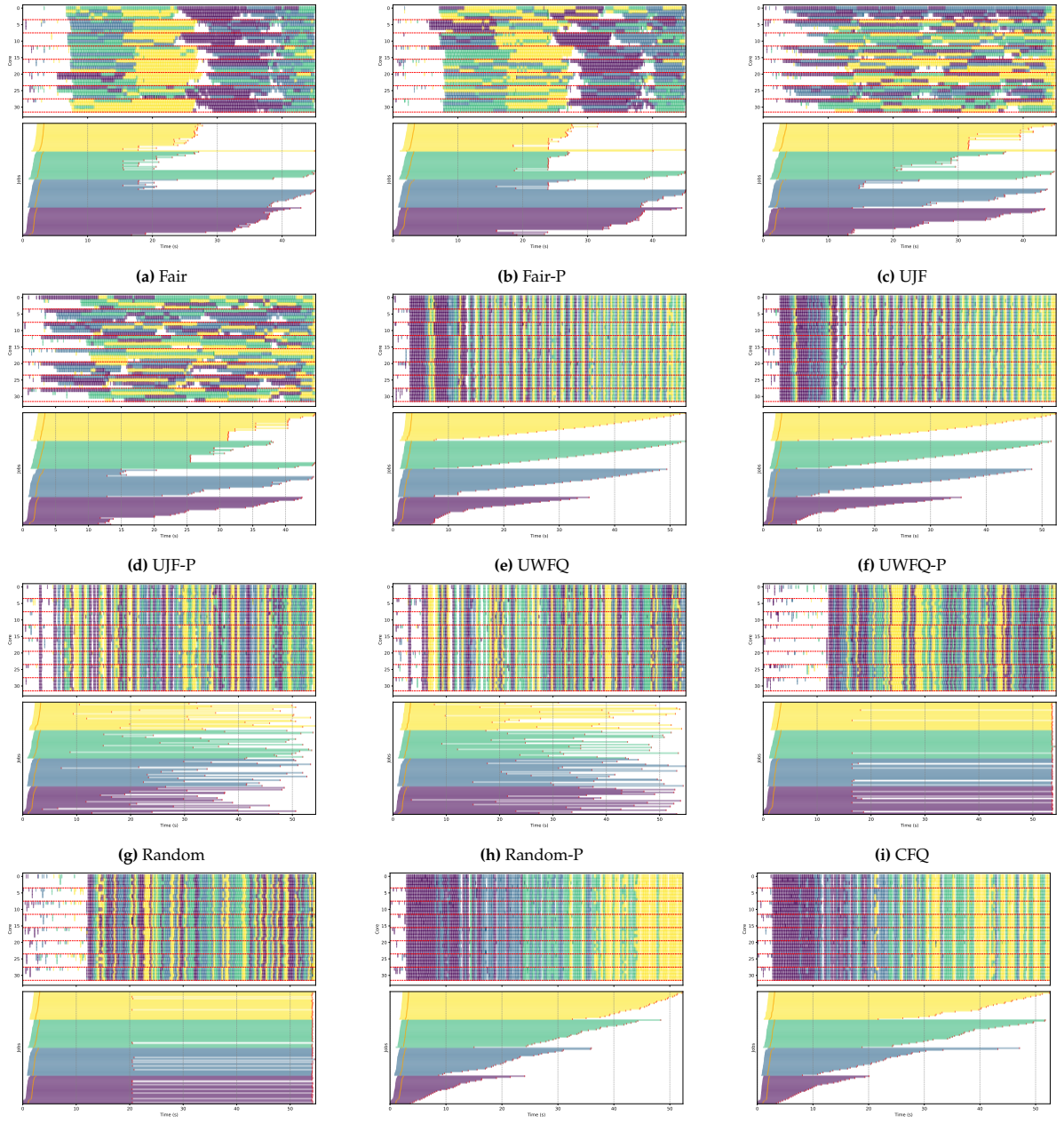
(a) Fair **(b)** Fair-P **(c)** UJF

(d) UJF-P **(e)** UWFQ **(f)** UWFQ-P

(g) Random **(h)** Random-P **(i)** CFQ

(j) CFQ-P **(k)** T-Fifo **(l)** T-Fifo-P

Figure C.3: Scheduler traces for scenario 3.



(a) Fair **(b)** Fair-P **(c)** UJF

(d) UJF-P **(e)** UWFQ **(f)** UWFQ-P

(g) Random **(h)** Random-P **(i)** CFQ

(j) CFQ-P **(k)** T-Fifo **(l)** T-Fifo-P

Figure C.4: Scheduler traces for scenario 4.

D

Macro-benchmark scenario extra data

Full tables of the runs can be found on the GitHub[1] repository, we do not include them due to difficulty of condensing large tables into A4 layout.

We show all the macro-benchmark traces of all the schedulers that were experimented with. Due to visualization limitations, we only showcase the job timeline without the task allocation to executors. Each color indicates the corresponding user that the job belongs to, which allows to examine job interactions between users competing for the same resources.

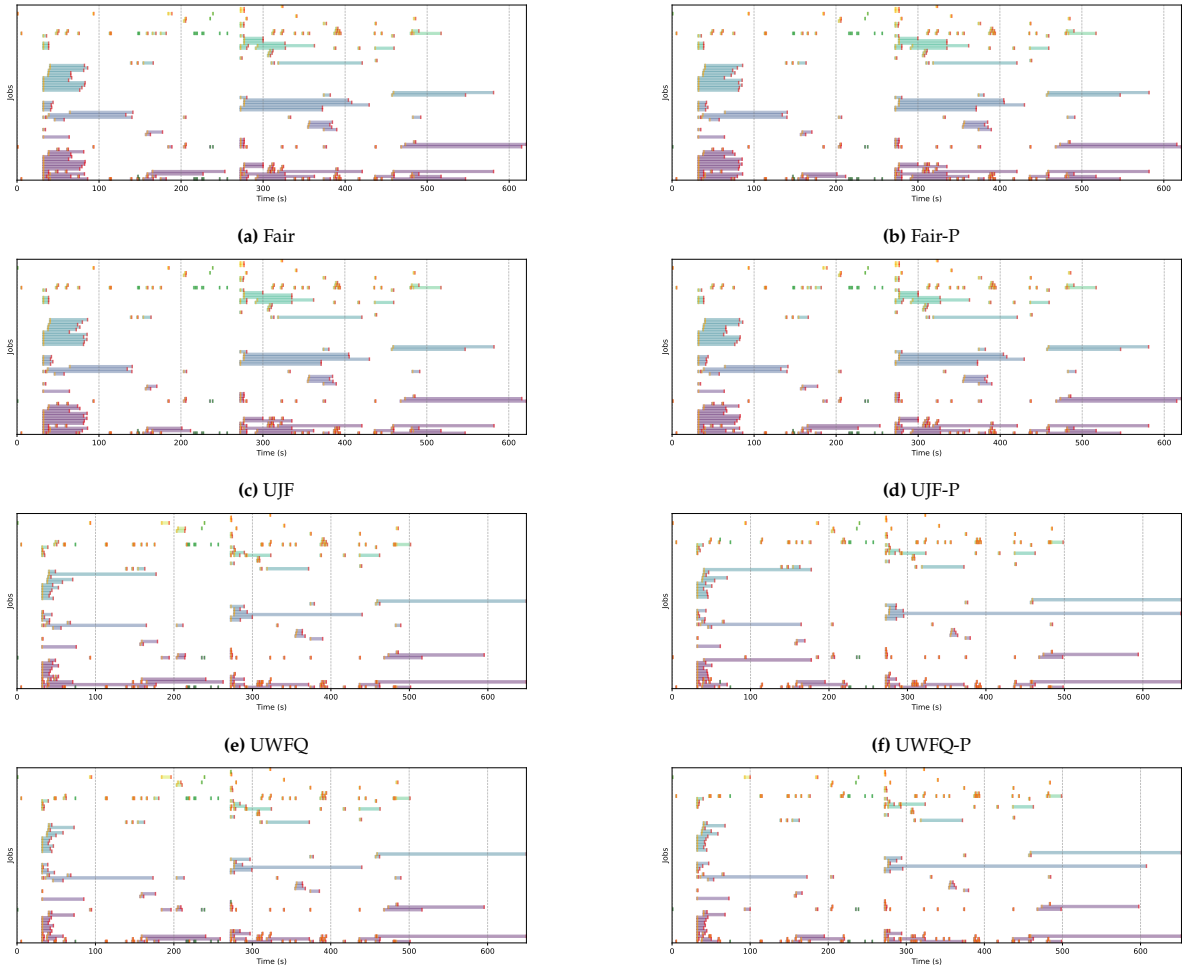
D.1. Heterogeneous macro-benchmark

The execution traces of schedulers can be seen in Figure D.1

D.2. Homogeneous macro-benchmark

The execution traces of schedulers can be seen in Figure D.2

[1] Full run tables: https://github.com/kazemaksOG/spark-benchmark-tool/tree/master/results/tables



(a) Fair

(b) Fair-P

(c) UJF

(d) UJF-P

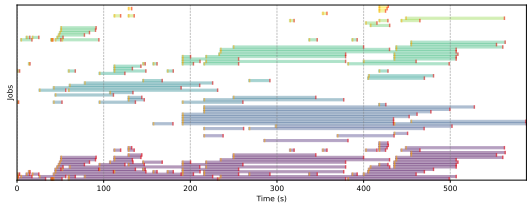
(e) UWFQ

(f) UWFQ-P

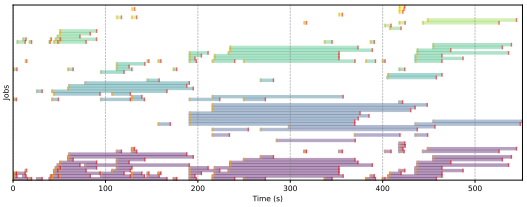
(g) CFQ

(h) CFQ-P

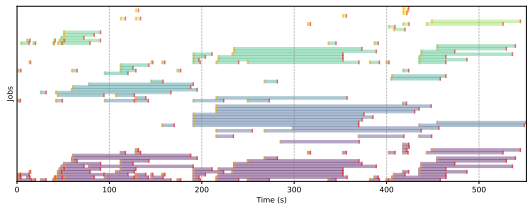
Figure D.1: Scheduler traces for Heterogeneous macro-benchmark.



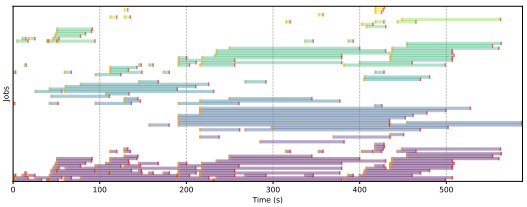
(a) Fair



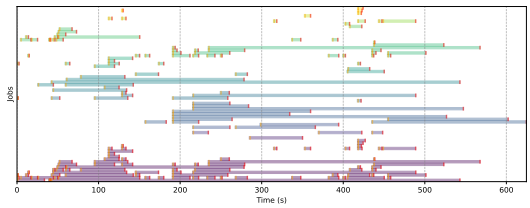
(b) Fair-P



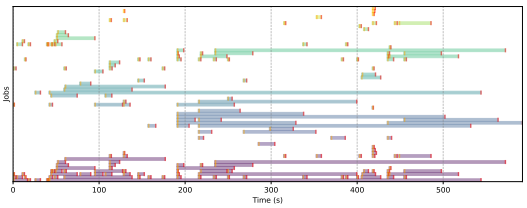
(c) UJF



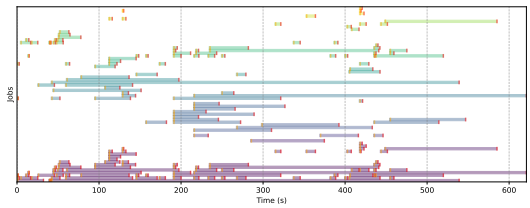
(d) UJF-P



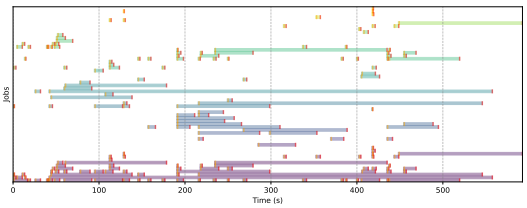
(e) UWFQ



(f) UWFQ-P



(g) CFQ



(h) CFQ-P

Figure D.2: Scheduler traces for Heterogeneous macro-benchmark.