External Scheduler Lab report

David Mol - 2825075 - d.r.n.mol@student.vu.nl
Omid Afroozeh - 2802676 - o.afroozeh@student.vu.nl
Jaime Perez y Perez - 2698463 - j.l.perezyperez@student.vu.nl
Supervisor: Sacheendra Talluri - s.talluri@vu.nl
External Scheduler Service Group 2

1 ABSTRACT

Distributed systems use schedulers to manage task execution and resource allocation. However, due to their complexity and tightly coupled nature, modifying them is risky and rarely done. In this report, we present our design for an external scheduler which allows for seamless integration and testing of various new policies for different systems used in production. We have implemented a basic proof of concept for our External Scheduler and have modified Dask to make use of it. Finally, we discuss our experimentation setup and evaluate the system.

2 INTRODUCTION

Modern distributed systems rely on built-in scheduling subsystems to manage task execution and resource allocation. Systems like Spark, Ray, and Dask are designed with these internal schedulers, which are efficient for general workloads. However, since they are focussing on general tasks, they have often limited performance, efficiency, and adaptability compared to newer scheduling techniques.

Modifying these internal schedulers is complex and risky, which makes users hesitant to adopt new approaches. Moreover, differences in scheduling implementations across systems create fragmentation, requiring developers to re-implement schedulers for each framework, thus reducing scalability and adoption.

This project addresses these challenges by designing an external scheduler as a service that decouples scheduling logic from the distributed system. It provides a unified, flexible, and portable scheduling layer, allowing for easier experimentation and adoption of advanced techniques.

2.1 Context

Distributed systems are widely used for large-scale data processing and computations by distributing tasks across a cluster of workers. Each system includes a built-in scheduler that handles resource management and task allocation. These schedulers are designed for general-purpose workloads and tend to follow a "one-size-fits-all" approach, which limits their adaptability to specific or emerging use cases.

Custom scheduling techniques have the potential to improve performance and efficiency, but integrating such techniques into existing systems is complex. Each distributed system maintains its own scheduler, and cross-system compatibility remains a challenge. While this project explores the development of an external scheduler for Dask, it does not yet offer cross-compatibility with other distributed systems like Spark or Ray.

The goal of this project is to design an external scheduler that integrates with a single system, Dask, and to assess its performance and flexibility. Although the implementation is not generalized for other systems, it provides a foundation for future work towards making external scheduling more adaptable.

2.2 Problems

The following problems mainly motivate our project:

- Limited Customization: Developers and researchers are unable to easily implement or test custom scheduling strategies, as doing so requires significant modifications to the internal scheduler code. These systems are tightly coupled, making any changes complex and error-prone.
- Fragmentation Across Systems: Each distributed system has its own scheduler implementation, which leads to fragmentation. If a researcher or developer creates a new scheduling technique, they must reimplement it separately for each system, reducing scalability and discouraging widespread adoption.
- Risk of Modifying Internal Schedulers: Internal schedulers are critical components of distributed systems, and any change or modification to them introduces the risk of system instability. As a result, developers and organizations are often hesitant to adopt new scheduling policies, even if they offer significant performance or efficiency gains.

2.3 Challenges

In this section we list and elaborate on the challenges faced. Several challenges were encountered during the project:

External Scheduler Overhead: Since our scheduler operates externally, there was communication overhead between the distributed system and the scheduler. While we identified this as a performance bottleneck, time constraints prevented us from exploring optimizations.

Validating Correctness: We were uncertain whether the scheduler was built correctly. Setting up validation was difficult as we had to configure a proper testing environment and use tools like `dask.distributed`'s `performance_report`. This added complexity to ensuring that the scheduler performed as intended.

Testing: Simulating realistic workloads was challenging, as Dask is used internally by companies to compute on their proprietary datasets. Therefore, our implementation lacks a realistic testing environment, this limits the scope of what we could evaluate. Setting up the necessary infrastructure for testing took considerable effort.

Tightly Coupled System Components: Integrating the external scheduler with Dask was difficult due to tightly coupled internal components, which made changes complex and risk-prone.

Unknown Environment: Working in an unfamiliar environment added another layer of difficulty. With no root privileges and a queuing system for accessing compute nodes, the setup and execution process required significantly more effort and time.

Evaluation: Performance evaluation was difficult, as fine-tuning parameters across different distributed systems and handling system-specific configurations took substantial time.

2.4 Your Approach

To address these issues, we propose an external scheduler that decouples scheduling logic from the distributed system itself. This would enable developers and researchers to experiment with and adopt advanced scheduling techniques without modifying the core of the distributed system. By providing an external unified, flexible, and portable scheduling service, our solution allows for easier customization, scalability, and experimentation across multiple systems.

2.5 Contributions

In the following, we present our design for an external scheduler. Furthermore, based on a simplified version of it, we showcase our implementation by modifying the source code of Dask to take use of our external scheduler. Finally, we evaluate the external scheduler and look to future improvements.

3 SYSTEM DESIGN

In the this section, we present our design for an ideal external scheduler, including high-level design and low-level design. Furthermore, we discuss the design process and various alternatives taken into consideration. Many ideas and methodologies were inspired by the AtLarge-research design framework [7] taught at the Distributed Systems course at the VU.

3.1 Users & Stakeholders

The main users affected by our system are the following:

- **Users of the original Distributed System**: If the users of Dask would want to benefit from various scheduling policies and mechanisms not offered by the internal scheduler, they can do so by using our external scheduler.
- **Scientists working on new scheduling methodologies**: People conducting research in academia and R&D departments in big tech companies can benefit from our system. They can easily verify and benchmark new policies by implementing them only once in our external scheduler without needing to deal with the internals of tightly coupled systems.
- **Distributed system maintainers**: As we know, systems that are currently in production are quite hesitant to implement new policies or to adopt new approaches. Our system can be leveraged to convince the original system designers and maintainers to consider the latest state-of-the-art techniques in the field.

3.2 Requirements

3.2.1 Functional Requirements. Our external scheduler must fulfill the following functional requirements:

FR1. **Task Submission and Scheduling:** The system must support receiving tasks from the distributed system and determining their scheduling order based on a configurable scheduling policy.
FR2. **Policy Implementation Interface:** It must allow developers to define and implement custom scheduling policies.
FR3. **Communication with Distributed Systems:** The external scheduler must integrate seamlessly with Dask or any other desirable distributed system through well-defined APIs or communication protocols (e.g., HTTP or RPC).
FR4. **Feedback Mechanism:** It should collect feedback and status updates from workers, enabling monitoring and adaptation to dynamic conditions.
FR5. **Fault Tolerance:** It must handle task failures by rescheduling or retrying tasks as needed.
FR6. **Monitoring and Logging:** The system should log events, errors, and metrics for debugging, evaluation, and analysis.

3.3 Non-Functional Requirements (NFR)

Our external scheduler must meet the following non-functional requirements:

NFR1. **Performance:** The scheduler should introduce minimal overhead while ensuring efficient scheduling and task execution.
NFR2. **Scalability:** It must scale to handle large numbers of tasks and workers without performance degradation.
NFR3. **Reliability:** The scheduler should operate reliably under high loads and recover gracefully from failures.
NFR4. **Usability:** It should be easy for developers and researchers to configure, deploy, and extend the system.
NFR5. **Portability:** The scheduler must be deployable on various infrastructures, including local setups, clusters, and cloud environments.
NFR6. **Maintainability:** The codebase should be modular, well-documented, and easy to maintain.

3.4 Design Process

In this section, we briefly describe the design process used for our work.

3.4.1 AtLarge-research design framework: Following the Distributed Systems course, we learned and used the Atlarge design process. Thus, being quite impressed by its simplistic and straightforward method, we adopted it. In the following, we will discuss each step in detail.

- **Requirement analysis:** After realizing the full picture of the project and its goals in the first week, we started our design by analyzing various functional and non-functional requirements. For this, we took a look into the traditional scheduler requirements and literature. [2, 6, 9] Furthermore, we modified the requirements to also include the specific needs of the project.
- **Understanding alternatives:** Having understood the scheduler landscape, we started considering various alternatives for each level of the project. From the high-level design standpoint, we examined distributed or monolithic

schedulers. Furthermore, we analyzed various ideas for fault-tolerance, communication, etc.

- **Bootstrapping the design process:** The Bootstrapping involved sessions of brainstorming within the team. With the exception of certain Operating System making use of external schedulers [11], we did not find much literature for Distributed System external schedulers. Therefore, we mostly relied on traditional designs for distributed systems' schedulers combined with the project requirements.
- **High-level design:** We finalized the high-level design and the interaction between various subsystems to achieve FRs and NFRs.
- **Detailed design:** Considering available options for our system, we made decisions on specific technologies and protocols.
- **Iterative analysis:** In three iterations, we analyzed the system to find shortcomings in regard to satisfying the requirements.
- **Experiments:** Finally, we designed various experiments to evaluate our system. We discuss this more in the following sections.

3.5 High-Level Design

In this section, we discuss our design from a high-level standpoint. We consider the different subsystems and the roles they play in the system. Furthermore, the interactions between the modules are analyzed to see how they can satisfy the requirements. In this section, we highlight various key decisions made.

3.5.1 *Chosen Design.* Our final design was a centralized scheduler that would completely replace the original scheduler. This design would be very powerful in terms of possible policy development and deployment as it allows complete control to start work on any of the workers.

The main modules in our design are as follows:

C1. **Central scheduling component:** This subsystem is responsible for choosing and fine-tuning the scheduling policy. New policies are implemented here, and it serves as the main interface for scientist and scheduling policy designers.
C2. **Monitoring component:** Handles the monitoring of the system. It reports its findings to the central scheduling unit.
C3. **Fault-tolerance component:** It keeps track of workers' status, and if need be, it restarts work on available workers.
C4. **Execution unit component:** Handles all logic related to worker-task submission and communications between the scheduler and workers.

3.5.2 *Alternatives.* During the design process, we briefly considered distributed external schedulers. However, due to the structure of systems intended for our project (Dask, Ray, or Spark) which all use centralized schedulers, we decided to abandon this idea.

Moreover, we considered a sidekick external scheduler. In this design, various tasks would be left to the original scheduler. In an ideal world, this design would be limiting, but for our proof-of-concept system we mainly opted for this as it is much simpler to implement.

3.5.3 *Qualitative Evaluation.* We believe this design should be able to satisfy the functional requirements as discussed in the following:

- FR1, FR2, and FR4 are satisfied by C1
- FR3 and FR7 are satisfied by C4.
- FR5 is satisfied by both C1 and C2
- FR6 is satisfied by C3

3.6 Low-Level Design

There are many finer, low-level decisions needed to be made to assure the requirements are satisfied.

3.6.1 *Chosen Design.* With regards to the communication protocol between the original system and the external scheduler, we decided on HTTP. Furthermore, we rely on heartbeat messages to keep track of workers status. The workers are expected to piggyback their current status, such as task completion progress, resource usage, etc, on the heartbeat message to minimize the communication overhead.

3.6.2 *Alternatives.* The alternatives for the low-level design were as follows:

- Standalone library with standard interface: In this design, instead of an HTTP server to handle the external scheduling, the external scheduling would be handled by an imported library. This design had the benefit of absolute minimum overhead as there would be no need for communication. However, dependency management, larger binary file size, and reduced portability are reasons why we abandoned this design
- Remote Procedure Call (RPC): Another alternative would be to use RPC to handle the scheduling. We decided to opt-out of this design as the RPC implementations often require specific client and server libraries. This tight coupling can make integration with diverse systems harder compared to HTTP, which is universally supported.

3.6.3 *Qualitative Evaluation.* We believe our low-level design should provide a best of both worlds in terms of performance and portability.

4 EXTERNAL SCHEDULER IMPLEMENTATION

In this section, we describe the implemented external scheduler in detail.

4.1 Simplified external scheduler

We have implemented a proof of concept external scheduler according to the requirements discussed above. Due to time considerations and the extreme complexity involved in disaggregating distributed systems, we opted for a much simpler implementation compared to the ideal design we discussed before.

For our implementation, we decided to use the python library Flask to implement an HTTP server as the external scheduler. There are various endpoints to submit jobs, register workers, remove worker, check the status of tasks running, etc. Internally, the server manages the state of each worker and the available tasks in the system. However, the current implementation completely relies on the internal scheduler, as it has no direct communication with the workers.

We have implemented various, specific policies based on the available statistics within the Dask internal scheduler for each workers. The policies are as follows: least memory utilization, least CPU load, shortest bandwidth latency, highest resource availability, and composite score. The composite score is the combination of the other policies.

The external scheduler simply responds to the submitted jobs by the internal scheduler by choosing the best worker based on the chosen policy.

4.2 Dask Modification

In order to verify our ideas, we decided to modify Dask, which is a flexible open-source Python library for parallel computing. Internally, it structures tasks as a Directed Acyclic Graph (DAG), where nodes represent computations and edges define dependencies. The scheduler orchestrates task execution. Dask's distributed scheduler uses a centralized architecture, with a scheduler managing global state and workers executing tasks. Data is partitioned and stored in memory or on disk. It integrates with NumPy, pandas, and more, allowing seamless scaling of computations.

Regarding Dask's internal, it is important to state the workers logic and scheduler logic are extremely tightly coupled. This made it incredibly hard to make meaningful changes. Nonetheless, we decoupled three major functionality of Dask's internal scheduler.

- *decide_worker* function: this function is the main decision point in the original scheduler. It is supposed to find the most suitable worker based on the input task and the available, valid workers. In order to modify this logic, we monkey-patched [5] it with our own *custom_decide_worker*. It mostly follows the same logic as before with the exception that instead of deciding the best worker on its own, it sends a HTTP request to the external scheduler. This request contains the ID of each worker alongside important statistics. It then expects an answer back with the chosen worker ID.
- *remove_worker* and *add_worker* functions: Handles the addition or removal of workers from the scheduler global state. We modified it the same way as the previous function to send a HTTP request to the external scheduler to maintain a consistent state between the two cooperating schedulers.

5 EXPERIMENT DESIGN

5.1 Experiments Overview

Using our simplified implementation described in the previous section, we designed and conducted several experiments. In this section, our experiment setup and methodology are discussed in detail.

5.2 Experiment setup

5.2.1 Devices and infrastructure. For this experiment we will use our own personal laptops to interact with the DAS6 machine located at the Vrije Universiteit. These laptops vary in specifications, ranging from a Macbook to a Windows device, with various specifications. However, since the workload of the experiment itself will be executed on the DAS6 machine and its nodes, the specifications of our personal laptops will not have a significant impact on the experiment's performance. The primary purpose of our laptops is to connect to the DAS machine and initate and monitor the experiment remotely.

The DAS6 machine itself, which is part of the VU's computing infrastructure, is a powerful system equipped with multiple CPU, GPU, and storage devices. For our node, we will use a 2.8Ghz CPU, 128GB of memory, 1.8TB of NVMe storage, and an interconnect speed of 100G ETH. This ensures that the computational tasks during our experiment can be properly handled. This infrastructure supports distributed computing, which is necessary for the experiment. This is available through the Dask package, which contains the dask-worker which is available on the nodes. The main nodes connect to the subnodes for our program and properly distributes the task over the available resources. A comprehensive overview of the DAS6 machine's capabilities and specifications can be found in the VU's DAS-6 cluster documentation[10].

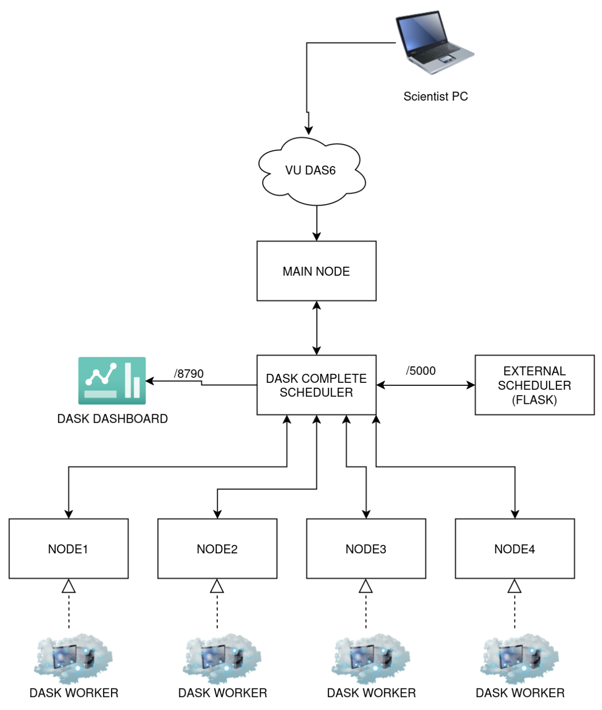

Figure 1: Overview of the Design

5.2.2 Experiment variables and settings. The experiment code is designed with flexibility in mind, which is why we made the configuration options declared dynamically. This allows us to quickly and easily modify the experiment's settings to explore different combinations of factors. This approach allows for rapid experimentation and testing of scenarios to determine the impact on the results, which is very important in the limited timeframe and computation time.

During preliminary testing, we encountered multiple issues related to the default Dask ports, which we assume to be caused by other conflicting services. As a results, we have defined the ports for the Dask scheduler ourselves. Specifically, during the experiment we will use port 8790 for the dashboard of Dask and port 8788 for the job workers, where the computation tasks will be distributed and executed. The external scheduler, will run on port 5000 through a Flask application.

5.2.3 Reservation of Nodes. To manage the resources on the DAS6 machine, we have implemented a reservation system within our program. Since the DAS6 machine is shared among multiple users, the reservation system ensures that we have the necessary nodes available for our experiment at runtime. Without such a system in place, the experiment could potentially run without the intended number of nodes, or even without any nodes at all, leading to unreliable results.

The reservation system operates by requesting the desired number of nodes and patiently waiting for them to be allocated. It checks the availability of the requested nodes at regular intervals, and if nodes were previously requested but not yet allocated, it will attempt to reuse them. The system includes several environment variables to configure its behavior. Specifically, the claim time, which is defined in the format HH:MM:SS, specifies the maximum duration of our reservation. Additionally, the node request timeout, defined in seconds, dictates the time interval between each check for node availability.

For our experiment, the claim time has been set to fifteen minutes. This should be sufficient for the experiment to complete within the desired time frame, while also providing enough time for the nodes to be allocated. The timeout mechanism is set to fifteen seconds, which ensures that the program doesn't hang indefinitely and provides a way to handle cases where node allocation takes longer than expected.

5.2.4 Tools. Since we have to connect to the DAS machine, and are not always on the VU, we needed some way to connect. For this, we use a VPN to the VU, using the application or the extension remote-ssh for Visual Studio Code. Once we have a connection to the VU, we connect to the DAS6 machine. On the DAS6 machine, we open our project and run the experiment. The project is retrieved on the machine by pulling it using git. Our project is available through github[1] which allows for proper version control and collaboration.

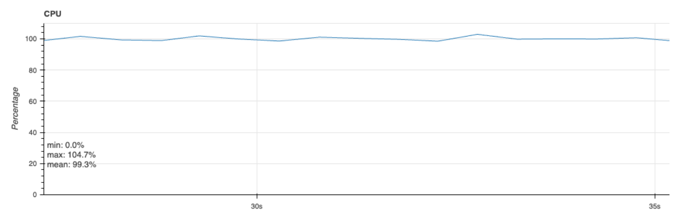

Figure 2: Example CPU utilization of the entire system provided by Dask's performance report

5.3 Experiment 1: Monte Carlo simulation

5.3.1 Motivation. This experiment was designed to push the system to its limits in terms of the number of decisions that need to be made. Therefore, since Monte Carlo simulation is an embarrassingly parallel application, it leads to a very large number of tasks that have no interdependency.

5.3.2 Variable and Constant Parameters. We have conducted this experiment in two phases. In the first phase, the main variables will be the increasing workload and the external scheduler policy, while the amount of resources used is kept constant. In the second phase, we use the same workload but will increase the amount of nodes available.

Finally, we make sure the resources are properly utilized. For this, we used Dask's dashboard to keep track of the entire system' utilization. CPU utilization for the majority of this experiment was near 100% as depicted in Fig 2.

5.4 Experiment 2: SQL query execution

5.4.1 Motivation. Because the Monte Carlo simulation is a random workload, we wanted to include an experiment that is based on real data and features a realistic workload. Dask supports executing SQL queries in distributed fashion through its Dask-SQL library. Therefore, we decided to implement simple Dask-SQL queries on the NYC yellow taxi dataset of 2014, this is a dataset of 165 million entries.[3]

5.4.2 Variable and Constant Parameters. In this experiment we send 10,20,30,40,and 50 simultaneous queries to the cluster. We limited these values, we wanted to test up to 100s of simultaneous queries, because we are reading in the full dataset each time. We tried to prevent this by persisting the data as in the Dask documentation[4]. This would have kept the data in memory to avoid having to re-read data. Additionally, we found that the system does become relatively unstable with many queries. For example we ran 100 queries on the Dask scheduler which took more than 9 minutes due to many failed tasks which got moved to other workers. However, running 150 queries on the same cluster afterwards resulted in a more reasonable execution time of approximately 56 seconds. Finally, when we go even further to 168 simultaneous queries we crash the cluster. We are unsure why this happens but it affects this experiment significantly.

5.5 Experiment 3: Pipeline execution experiment

5.5.1 Motivation. So far we have only experimented on BOTs-like(Bag of Tasks) workloads. Therefore, we decided to also evaluate our scheduler on a workload of workflows. We adapted a data preparation pipeline we created for the Data Mining Techniques course of the VU for the Expedia recommendation datasets to be able to run on the Dask server with some parallelization datasets[8].

5.5.2 Variable and Constant Parameters. Here we wanted to run a similar experiment as the SQL experiment (i.e. increase the workload in steps), however after some informal experimentation we found that the parallelization in the pipeline was so limited that the schedulers basically just scheduled as many pipelines as the system could support. And then all pipelines would finish at the same time and the next pipelines would be scheduled etc. until all pipelines were scheduled. This lack of parallelization is due to us not using the Dask Dataframe as we ran into errors with it and we did not have enough time to debug it as we tried to include this evaluation quite late in the project. Therefore, the schedulers did not have a significant impact on the execution of the pipelines. Thus, we decided to not run this experiment however we like to include this here for completion sake and to explain the omission of pipeline workloads in the evaluation.

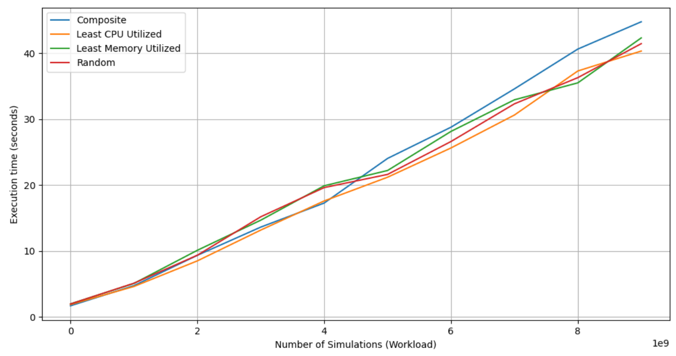

Figure 3: Comparison between different scheduling policies. Composite score, least loaded CPU, least memory utilization, and random policies are compared to each other. X-axis is the increasing workload, while the Y-axis represents performance in execution time.

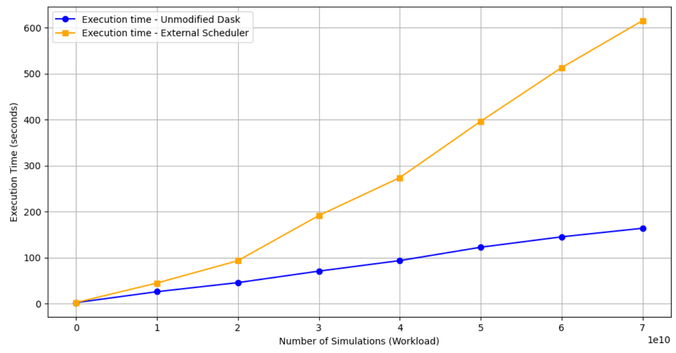

Figure 4: Original Dask compared to External Scheduler. Measuring the performance loss when our External Scheduler is used compared to the unmodified Dask implementation. Note, the random policy was used for this experiment as the baseline.

5.6 Evaluation

After conducting the experiments discussed previously, we attempted to evaluate our design using the implemented External Scheduler. Dask use cases are rather varied; Thus it was very important to evaluate the system with regards to different workloads.

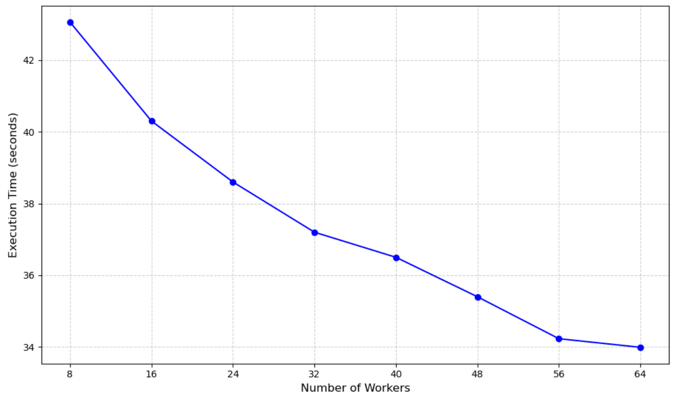

Figure 5: Communication overhead experiment. Same workload is run on increasing number of workers to measure the communication overhead. Ideally, the workload should be done much faster due to the increased resource, but the main bottle neck seems to be communication between the schedulers.

5.6.1 *Bag of task workload evaluation.*

Motivation. Bag of tasks are one of the most common workloads executed on Dask and similar distributed systems. Since the majority of the tasks actually do not have any interdependency, the majority of the decisions are made by the scheduler. As mentioned before, each decision made by the local scheduler is first redirected to the external scheduler. Hence, it is very important to put the system under a large number of small tasks to measure the overhead of communication between the two scheduler components.

Evaluation procedure. Using the Experiment 1 discussed before, we gradually increase the workload and observe the system during execution. We collect various data, such as number of workers, execution time, CPU usage, etc. Finally, using the collected information, we analyze the system.

Evaluation results. As seen in Fig 4, there is a considerable amount of overhead incurred by our external scheduler when compared to the original Dask functionality. We believe the majority of this overhead is caused by the communication between the internal and external schedulers.

Moreover, it is noteworthy to mention that *least_CPU_load* scheduling policy seems to outperform the other implemented policies as seen in Fig 3.

Finally, it is clear from the results depicted in Fig 6 that communication overhead is the clear bottleneck in our implementation. As the experiment chosen is highly parallelizable, we expected a much better speedup. However, the outcome is far from ideal.

Result Analysis. As discussed, our implementation is highly inefficient for applications that require a large number of tasks as it leads to greater communication overhead. This becomes more of an issue for large number of resource. This is mainly because the internal scheduler may need to construct larger message to account for the greater number of suitable workers.

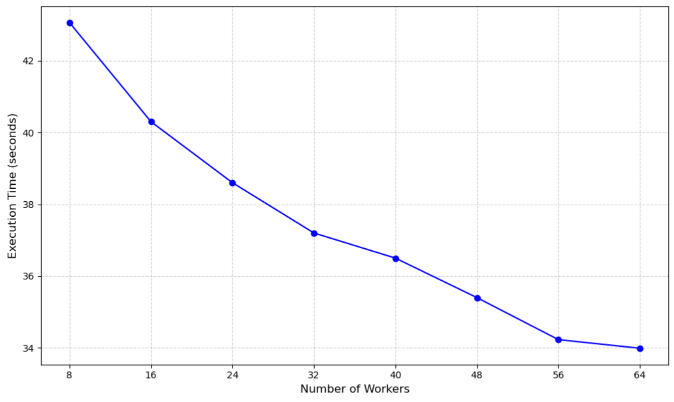

Figure 6: Communication overhead experiment. Same workload is run on increasing number of workers to measure the communication overhead. Ideally, the workload should be done much faster due to the increased resource, but the main bottle neck seems to be communication between the schedulers.

Evaluation Implication. We believe our work can be a first stepping stone to the ideal external scheduler designed in the previous sections. However, due to time and resource constraints, we had to rely on a naive implementation, but performance optimizations for this project are endless. For starters, we suggest improving the current work by decoupling the worker and scheduler code such that workers can directly communicate with the External Scheduler. This alone can reduce the overhead significantly and bring the performance on par with the internal scheduler.

5.6.2 *SQL query evaluation.*

Motivation. SQL database systems commonly used are greatly optimized for a single node. However, if the data needed to be processed is larger than the available storage in a single node, it is best suited to carry out this operation in a distributed system. There are many frameworks that speed up data wrangling pipelines using distributed systems. One such system is Dask-SQL extension.

Evaluation procedure. Using the Experiment 2 discussed before, we gradually increase the workload and observe the system during execution. We collect various data, such as number of workers, execution time, CPU usage, etc. Using the collected information, we analyze the system.

Evaluation results. As seen in Fig 4, there is a considerable amount of overhead incurred by our external scheduler when compared to the original Dask functionality. We believe the majority of this overhead is caused by the communication between the internal and external schedulers.

Moreover, it is noteworthy to mention that *least_CPU_load* scheduling policy seems to outperform the other implemented policies as seen in Fig 3.

Finally, it is clear from the results depicted in Fig 6 that communication overhead is the clear bottleneck in our implementation.

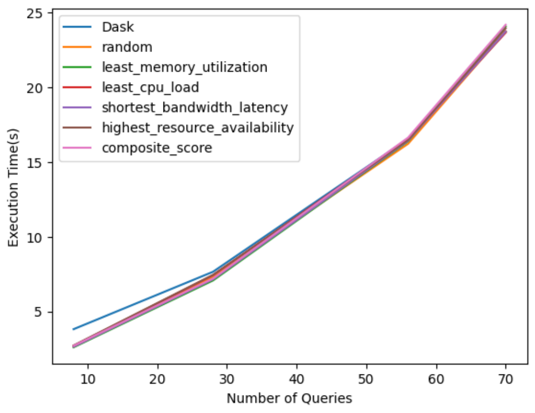

Figure 7: Our limited evaluation shows that every scheduler is competitive when it comes to SQL queries.

As the experiment chosen is highly parallelizable, we expected a much better speedup. However, the outcome is far from ideal.

Result Analysis. As discussed, our implementation is incapable of dealing with a large number of queries. Therefore, our results are not very interesting. But they do suggest that the dask-sql library does not parallelize well as our scheduler and its policies are competitive and, in some cases, even beat the Dask scheduler. Before our results can be truly meaningful, we would need to find the source behind some of the strange behaviour in our experimentation setup.

6 DISCUSSION AND CONCLUSION

In this project, we successfully implemented the external scheduler. However, its performance has not yet reached the level of the default Dask implementation. Since optimizing performance was not the primary focus, there are still plenty of opportunities for improvement. For instance, exploring an alternative to Flask for the API endpoint could lead to better speed. We could also develop our own API framework, focused on simplicity and speed. These improvements could significantly enhance the results and fully realize the potential of the external scheduler in terms of performance. However, if the current implementation would be used in system workloads that are orders of magnitude higher, the performance would exponentially get worse, which is unwanted. The performance decrease for bigger workloads is not avoidable, however the baseline, the Dask scheduler, does perform better in this regard. The base setup works as intended, with computations being handled properly, indicating that the various components interact correctly, even though they are not yet optimized. The external scheduling project has the potential to support developers and researchers into implementing their own policies and discovering new opportunities.

7 APPENDIX A: TIME SHEETS

The table below shows roughly the hours that were spend for each major part of the project. It is divided in the various time spending categories: thinking time, development time, experimenting time, analysis time, writing time, wasted time and total time.

Category	Think	Dev	XP	Analyse	Write	Waste	Tot
Design	20	40	8	0	6	3	77
Implement	10	80	15	5	5	3	118
Experiment	5	10	20	40	3	2	80
Writing	1	0	0	1	23	0	25

REFERENCES

[1] Omid Afroozeh, David Mol, and Jaime Perez y Perez. 2025. ExternalScheduler. *Github* (2025). https://github.com/OmidAfroozeh/ExternalScheduler

[2] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. 2018. A reference architecture for datacenter scheduling: design, validation, and experiments. *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (2018), 478–492.

[3] NYC Taxi Limousine Commission. 2025. TLC Trip Record Data. *Taxi Limousine Commission* (2025). https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[4] dask. 2025. dask-sql quickstart. *dask documentation* (2025). https://dask-sql.readthedocs.io/en/latest/quickstart.html

[5] GeeksforGeeks. 2024. Monkey Patching in Python (Dynamic Behavior). *GeeksforGeeks* (2024). https://www.geeksforgeeks.org/monkey-patching-in-python-dynamic-behavior/

[6] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in {Multi-Resource} clusters. *12th USENIX symposium on operating systems design and implementation (OSDI 16)* (2016), 65–80.

[7] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. 2019. The AtLarge Vision on the Design of Distributed Systems and Ecosystems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1765–1776. https://doi.org/10.1109/ICDCS.2019.00175

[8] Kaggle. 2024. Personalize Expedia Hotel Searches - ICDM 2013. *Kaggle* (2024). https://www.kaggle.com/c/expedia-personalized-sort/data

[9] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), 351–364.

[10] VU University. [n. d.]. Clusters of the DAS group. https://www.cs.vu.nl/das/clusters.shtml. Accessed: 2025-01-10.

[11] Mikael Åsberg, Thomas Nolte, Shinpei Kato, and Ragunathan Rajkumar. 2012. ExSched: An External CPU Scheduler Framework for Real-Time Systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 240–249. https://doi.org/10.1109/RTCSA.2012.9