Vrije Universiteit Amsterdam

Master Thesis

Design, implementation, and evaluation of data intensive applications in cluster and serverless environments

---

Author: Bogdan Gabriel Ene (2553671)

1st reader:	Prof. dr. ir. Alexandru Iosup
daily supervisor:	Ir. Sacheendra Talluri
2nd reader:	Dr. ir. Animesh Trivedi

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 20, 2022

Abstract

Cloud computing is a growing technology that gives both private and public organizations the opportunity to operate and meet their goals more efficiently. A major benefit of the cloud environments comes from the fact that it offers the necessary resources for its customers on-demand without the need of maintaining a private infrastructure [16]. Moreover, the cloud providers implemented a new way of charging for the services they offer, namely pay-per-use. In this sense, each customer only pays for the resources they utilize and they do that only for the time they use those resources. One of the emerging cloud computing paradigms that makes use of the pay-per-use schema is the serverless computing. This paradigm enables its users to deploy small software applications into the cloud, without the need of handling the operational logic. One of the most popular serverless service is Function-as-a-Service (FaaS) where small code functions are run in the cloud, while the necessary resources and execution environments are handled by the cloud provider.

Serverless computing becomes more and more popular, as resources are made available easily for its users. However, for this to be possible, cloud providers need to constantly maintain and expand their infrastructure and soon it can reach a level where it becomes unaffordable. In this sense, it is important that applications are developed to execute efficiently. Moreover, as many decisions in every work field of today's society are based on information extracted from data, it is necessary that data processing applications continue to be up-to-date, meeting the constantly changing needs and the continuously increasing volumes of data.

In this thesis we design and implement several data-intensive applications using different serverless architectures as well as different workload parallelization techniques. We use resources provided by different cloud environments, DAS-6 cluster from Vrije Universiteit Amsterdam and AWS cloud, and we create different workloads for testing our applications. In the end, we analyse the performance of the applications in relation with the given workload and compare their execution.

Contents

1 Introduction

Cloud computing is the revolutionary technology that enables organizations to grow at a rate that was never imagined before, as it removes the necessity of acquiring physical hardware resources. In this sense, cloud environments bring computing as a utility, giving the freedom for software solutions to access the necessary resources on demand [16]. By doing so, software solutions are capable of scaling further at any given point in time.

Serverless computing is a cloud computing paradigm that enables users to deploy small software applications into the cloud, without the need of handling the operational logic [3]. As a result, developers can focus only on the application development, whereas the cloud environment is responsible for resource provisioning and good performance of the application. This is usually a challenging aspect of serverless computing, as owners of the applications must rely on cloud provider's design decisions, as well as their quality of service monitoring, scaling, and fault tolerance properties [3]. However, serverless computing enables a more fine grained cost control compared to traditional cloud computing. Although in serverless computing there is no control over the resource provision, as it is all managed by the cloud environment, the user only pays for the computation time required for the application to complete.

Function-as-a-Service (FaaS) is one of the most popular serverless services available in cloud environments, where users run small code functions in the cloud while resources, lifecycle and event driven execution is managed by the cloud provider [3]. This provides opportunities to run microservices on demand while paying only on the computation time as well as new possibilities for parallel processing. A straightforward example of how cloud functions operate can be given by a microservice responsible for image resizing. In this sense, whenever an event containing image data is submitted, a new cloud function is instantiated and processes the image, creating different resolutions of it. The owner of the microservice is then billed for the exact time that the instance ran. Moreover, it enables limitless and automatic scalability, together with fast processing of enormous payload. This is due to the fact that for each submitted event, a new cloud function is instantiated and cloud environments permit for limitless concurrent instances.

Technology is continuously evolving and software systems become more enhanced in our every day lives. Because of this, an enormous volume of data is created daily in every domain of activity. As an example, 12 terabytes of data is created everyday only from tweets alone [9]. By analysing this data, improvements to many fields can be brought in, including business, the scientific research, public administration, and so on [6]. However, as the volume of data increases, current solutions get old and become inefficient at capturing, curating, analysing and visualizing it. As also the cloud services grow in popularity, in this thesis we plan on analysing the performance of two data-intensive applications, a sorting application and a TPC-DS application, on serverless architectures. Moreover, we redesign the sorting application using two different workload-parallelization techniques in order to make resource usage more efficient, while also increasing the throughput. In the end, we experiment with them and we analyse their execution.

1.1 Context

Cloud computing changed the way resources and services are being billed and it did this by introducing a new payment scheme, namely pay-per-use [16]. This ensures that the clients only pay for the resources that they use, enabling software solutions for limitless scalability together with removing the risk of under- or over-provisioning. Because cloud computing is able to reduce the costs, by removing the need for organizations to continuously update and maintain their in-house infrastructure, but also to increase the performance of their systems, many companies have gained interest in running their applications in cloud. Samsung, BMW, Siemens and Netflix are just a few of the biggest companies that make use of the Amazon Web Services [1]. Moreover, a big part of them already spent millions of dollars on those services [24]. Although cloud computing provides many services, lately serverless computing has seen a rapid increase in popularity. AWS Lambda was first released in 2014 and it is expected to be worth around 15$ billion by 2023 [20].

Technology is part of our everyday life and such, software applications gather enormous amounts of data. In this sense, decisions based on the results of data analysis are being taken in every field of activity [9]. We can observe the presence and potential of Big Data in many fields like manufacturing and transportation, through usage of IoT sensors, e-commerce, retail, banking and so on. Moreover, Big Data's potential can also be observed in scientific fields like medicine, biology, astronomy. This clearly shows that Big Data is our opportunity to continue revolutionizing the world. To demonstrate the rate at which data is growing, estimates show that the volume of business data worldwide doubles every 1.2 years [6]. Moreover, there is a lot of interest for Big Data also in the public administration field. In 2012, Obama administration announced the Big Data research and development initiative in order to investigate the problem that the government is facing [6].

As Big Data is a growing field, traditional technologies become old and inefficient to serve nowadays needs. However, cloud technologies are continuously expanding, especially serverless services. In this sense, it is interesting to observe how data-intensive applications behave when implementing different workload-parallelization techniques in order to make processing more efficient and implementing serverless architectures. This can be an opportunity for further scaling of data capturing and analysis tools.

1.2 Background

Although many definitions for cloud computing have emerged in the past years, the National Institute of Standards and Technology (NIST) defines it as *a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction* [16]. The resources and infrastructure are managed by the cloud provider and are available to the customer on request. In this case, the customer does not require to self-maintain the infrastructure while having the option to scale according to their needs. Moreover, the cloud provider must ensure the availability and good functioning of the provided services [4]. However, at its beginnings cloud providers offered Infrastructure-as-a-Service, which implies that access to resources is made through the usage of virtual machines [4] [14].

This leaves the problem of scaling on the developers shoulders, that most of the time use over-provisioning to handle sudden changes in their systems' load. Lately, serverless computing has emerged. This gives the opportunity to better scale and exactly match the needs of the system at any point in time, without the need of starting and stopping servers. Moreover, it does not require extensive expertise from the developers to analyse in-depth the resources that are necessary such that the system runs in good condition [5]. The most common way to use serverless computing is through the usage of Function-as-a-Service. In this case, software developers implement microservices in the form of code functions that are made available on the cloud platforms. The microservices are then executed by cloud functions during a limited period of time, with a minimal requirements specified. Whenever requests or events come in, new cloud functions are started in order to resolve them. At the end, the owner is billed for the exact amount of time the cloud function ran, in relation with the resources that it used [10].

In the same time, data gets created at a rate that was never imagined before. Statistics show that 90% of the data that exists in the world today, was created in the last two years [9]. Because of this, conventional systems are surpassed, become outdated, and new solutions must be found. In this sense, serverless technologies are a promising field that can offer solutions for further expansion of the data-intensive applications.

1.3 Problem Statement

From the moment cloud computing made its first appearance, we have seen a rapid and continuous evolution. Not only did it become more powerful in terms of resources that it provides, but also in terms of technologies it offers in order to consume its resources. Lately, serverless computing gained a lot of interest from both the academia and the industry. This is due to the fact that it enables the user to have more fine-grained control over their spendings on the necessary resources. Moreover, it enables applications to scale much faster by deploying new cloud function instances whenever it is needed.

Data is an important aspect in our everyday life. We create data continuously and based on analysis decisions and improvements are being made. However, the volume of data grows at an unimaginable rates and traditional techniques of capturing and analysing it easily become outdated. Serverless technologies opened up possibilities for designing new solutions that can scale further, handling more data in a more efficient manner, while keeping the costs manageable. Because of this, we are eager to explore new possibilities in processing the data efficiently by designing and analysing the performance of data-intensive applications following different serverless architectures.

Traditionally, applications perform tasks sequentially. However, in most of the cases, these tasks do not require to use the entire computing power available and also do not make use of the entire I/O capabilities. Although cloud environments offer resources on-demand, we believe that this fact is going to reach an end. As the data centers grow, cloud providers must create new ways of managing the continuously growing complexity of the infrastructure, while being able to provide the agreed services. This will make it harder and harder to expand the infrastructure available for the clients and can make it soon not affordable [11]. For this reason, and in concordance with the vision of Iosup et. al. [11], we believe that new ways of designing systems must be developed.In this work

we design, implement and evaluate data intensive applications that try to process data in a more efficient manner by implementing different ways of workload parallelization.

1.4 Research Questions

In order to respond to the problem stated above, we plan to explore different serverless architectures and different workload-parallelization techniques for data-intensive applications. In this sense, we implement a sorting application and a TPC-DS application.
To easily track down the progress of this work and to ensure that the goal is achieved, we define the following research question:

RQ1. What is the performance impact of sorting big data in cluster environments?

To answer this question we design and implement a sorting algorithm and we select multiple parallelization techniques, such as job parallelization, where series of tasks are being processed in parallel, as well as task parallelization, where each individual task is scheduled for parallel processing. We implement each technique within a different sorting application. Because we use multiple applications for answering this question, we further split the question into three sub-questions.

RQ 1.1. What is the performance of data-intensive sorting application with job parallelization in cluster environments?

This is the first implementation that intends to make resource usage more efficient, while also increasing the throughput. To answer this question, we design, implement and analyse the execution of this application.

RQ 1.2. What is the performance of data-intensive sorting application with task parallelization in cluster environments?

This is the second implementation that intends to use resources more efficient, while increasing the throughput. Different than the first implementation, in this case the application is able to parallelize each individual task. In order to answer this question, we analyse the execution times of each task using multiple experiment configurations.

RQ 1.3. What workload-parallelization technique shows better performance for the data-intensive sorting application in cluster environments?

To answer this question, we compare the execution times of the jobs and tasks resulted from the experiments ran on the two implementations of the data-intensive sorting application. Moreover, we compare what application is faster in completing the given workload.

RQ2. What is the performance impact of sorting big data in serverless environments?

To answer this question we design two serverless data-intensive applications, a sorting application and a TPC-DS application, and analyse the execution times of each cloud function involved in their executions. The applications apply the job parallelization technique that was used in the previous applications where a series of sequential tasks are grouped into a job and each job is processed by a different cloud function.

RQ3. What are the performance differences when comparing the execution of the data-intensive serverless sorting application and the execution of the data-intensive sorting applications implementing different workload parallelization techniques in cluster environments?

Finally, we look at the executions of the serverless implementation of the sorting application with the executions of the data-intensive sorting applications that use different workload parallelization techniques. We compare the execution of individual tasks as well as the execution of the applications and we draw conclusions.

1.5 Approach

To answer our research questions, we require to develop data-intensive applications that implement serverless architectures, as well as different workload-parallelization techniques. In this sense, in the first phase of the research we choose to implement a sorting algorithm, using the MapReduce paradigm [7]. We choose the radix sorting algorithm, as it requires a decent amount of computational power which is enough to measure the performance of the application. Moreover, we choose MapReduce because it is a programming model designed for distributed systems in order to process massive amounts of data. Because of this, it enables the design of scalable algorithms [13].

Step 1. ***(Answering Research Question 1)***

We select techniques to sort big data in cluster environments. We select two techniques, job and task parallelization. In the first case, the sorting application performs job-parallelization, schedules the workload in the form of jobs and each job consists of several tasks executing sequentially. In the second case, the application does task-parallelization and splits the jobs into several tasks. This enables performing task parallelization, by assigning each task to a different process on the server.We independently.

Step 1.1. ***(Answering Research Question 1.1)***

This step consists of running multiple experiments, following different configurations, using the worker sorting application to schedule jobs and execute them in parallel using multiple processes. Because a job consists of multiple tasks that are executed sequentially, we analyse the execution times of each task of a job, as well as the execution times of the jobs themselves.

Step 1.2. ***(Answering Research Question 1.2)***

Similarly, the second step consists of running multiple experiments, following different configurations. However, this time, we use the sorting application implementing task parallelization and we analyse the execution times of each task, as well as the execution of the entire application.

Step 1.3. ***(Answering Research Question 1.3)***

To finally answer Question 1, we compare the execution times of the jobs and tasks resulted from the experiments ran on the two implementations of the

worker sorting application. Moreover, we compare what application is more efficient in completing the given workload.

The next phase of the research consists of implementing two serverless applications that can be configured to run in the most popular cloud environments, using different workloads. The first application implements the same sorting algorithm that follows the MapReduce model as the worker sorting applications mentioned above. The workload is divided into equal jobs, each job being assigned to a different could function. The second application implements TPC-DS [17] - a decision support benchmark for performance measurements. The TPC-DS application implements different complex SQL queries on a given database. The application splits the SQL query into several smaller queries from which it generates workload in the form of jobs, each job being assigned to a different cloud function.

- **Step 2** ***(Answering Research Question 2)***

  To answer this question, we run and compare multiple experiments with different configurations on the two applications mentioned above. For each application, we then analyse the execution times of each cloud function that is used. As each cloud function executes one job and each job is composed of several tasks that are sequentially executed, we analyse the execution times of each task. Moreover, we analyse whether there is any variation in the execution times of the tasks and cloud functions over multiple experiment runs.

Finally, we compare the execution of the worker sorting applications using IaaS with the execution of the serverless sorting application using FaaS.

- **Step 3** ***(Answering Research Question 3)***

  Finally, we look at the executions of the serverless implementation of the sorting application with the executions of the worker sorting applications. We compare the execution of individual tasks as well as the execution of the applications and we draw conclusions. The purpose of this phase is to highlight the differences between executions of the same sorting algorithm developed in two forms, a serverless application and a worker application.

1.6 Main Contribution

This research provides **Technical** and **Experimental** contributions. Next we map them to the research question presented in Section 1.4 as follows:

- (**Technical, Research Question 1** and **Research Question 2**) Design and implement data-intensive applications implementing diverse workload-parallelization techniques in cluster environments.
- (**Experimental, Research Question 1** and **Research Question 1**) Analyse performance of each data intensive application implementing diverse workload-parallelization techniques in cluster environments.

- (**Technical, Research Question 1**) Compare the results of the executions of the data-intensive applications implementing diverse workload-parallelization techniques in cluster environments.
- (**Technical, Research Question 2**) Design and implement data-intensive serverless applications that make use of cloud functions. Compare results of the execution of the applications.
- (**Experimental, Research Question 2**) Analyse the execution of the data-intensive serverless applications that make use of cloud functions.
- (**Experimental, Research Question 3**) Compare the execution of the data-intensive applications implementing diverse workload-parallelization techniques with the execution of the serverless applications using FaaS.

1.7 Anti-Fraud Measures

All the applications used in this research were developed by the main author of the thesis with the assistance of the daily supervisor. All the applications are open-source and publicly available. The worker sorting applications are available in the owner's GitHub repository[1]. Similarly, the serverless sorting application is available in the owner's GitHub repository[2], as well as the TPC-DS application[3]. The worker sorting applications and the serverless sorting application were developed from scratch, while the TPC-DS application represents an updated version of the implementation available in the public GitHub repository[4].

1.8 Related work

Since cloud technologies have emerged, work has been conducted in order to benchmark the serverless cloud performance, especially the FaaS services. Similar to our work, previous work focused on benchmarking different aspects of the cloud platforms, such as hardware resources, startup latency, concurrency and elasticity and event trigger latency. However, as serverless computing evolves results of benchmarks become outdated.
Iosup et. al. [20] present their vision regarding a comprehensive benchmark for the serverless computing. They motivate that a very important aspect of benchmarking is reproducibility, since the cloud evolves and benchmark results get outdated. Their benchmark, similar to our study, identifies areas of the serverless environments that must be measured, such as function runtime, event propagation and software flow. However, more than this, they identify that the cost of running software in the serverless environments must be also measured, since it is an important aspect when deciding on the necessary resources and performance required for each software running in the cloud. To demonstrate the cost benefits of serverless environments, Villamizar et. al. [22] creates a performance and cost comparison between a software with a monolithic architecture, a software with a microservice architecture that is operated by the cloud customer and a a software with a

[1] https://github.com/bgdbgd1/sorting-with-threads
[2] https://github.com/bgdbgd1/lithops-radix-sort
[3] https://github.com/bgdbgd1/tpcds-lithops-scripts
[4] https://github.com/ooq/tpcds-pywren-scripts

microservice architecture operated by the cloud provider. They present that microservice architectures can help reduce the cost of the infrastructure. Moreover, they demonstrate that using services specifically designed to be used in relation with microservices reduce infrastructure costs by up to 70%.
Other related work compare the performance of the serverless environments offered by different cloud providers. Figiela et. al. [8] test several hypotheses on multiple closed-source clouds like AWS, Google, Azure and IBM. Similar to us, they observe that application server instances are reused between calls, therefore only the first cloud functions infer cold starts. However, different than us, they test cloud performance such as CPU performance and network throughput, in relation with the function size. Wang et. al. [23] also focus on CPU and network performance in relation with function size, but also with the number of concurrent functions used, as they perform the largest measurement study to date, using 50 000 cloud functions. Moreover, they observe cold starts of cloud functions in different cloud environments. An in-depth analysis on the scalability of the serverless environments is presented by McGrapth et. al. [15]. They provide measurements on cold and warm starts and analyse both container allocation and container removal. Lee et. al. [12] analyse performance of serverless environments from multiple providers in relation with the number of concurrent functions. In this sense, they use different workloads that are either CPU, disk or network intensive. Similar to us, they compare the execution of the workloads in serverless environments with the execution of the same workloads using virtual machines. In the end, they conclude that serverless execution is usually cost effective against sequential execution of workload on virtual machines. A literature study performed by Scheuner et. al. [19] documents 112 studies, both academic and grey literature, that evaluate serverless environments, specifically Function-as-a-Service. They document the trends of the publications in performance evaluation, what are the most common platforms benchmarked, what kind of experiments are commonly used and what services are used in the experiments.

1.9 Reading Guidelines

The remainder of this thesis is structured as follows: Chapter 2 presents the requirements of the applications used in research together with their design and implementation. Chapter 3 presents the experimental goals, as well as the design and configurations of the experiments. Chapter 4 presents the analysis performed on the execution of the two worker sorting applications using different workload parallelization techniques. Moreover, we create a comparison between the performance of the two worker sorting applications. Chapter 5 presents the analysis on the execution of the serverless applications using FaaS. In the end, we compare the execution of the worker sorting applications with the execution of the serverless applications. Chapter 6 presents the discussion, conclusion and future work.

2 Big Data Application Design and Implementation for Cluster and Serverless Execution

This chapter describes the design and implementation of all data-intensive applications that were used in the research and contributes and contributes to answering RQ 1 and RQ 2. In order to set the goals of these applications, we first present the functional and non-functional requirements in section 2.1. Following the requirements, two different applications were implemented, a sorting application and a TPC-DS application. However, the sorting application was developed following 3 different approaches and resulted in 3 different applications, a serverless application and 2 different worker applications. Section 2.2 presents the design of the serverless sorting application and worker sorting applications and Section 2.3 presents the design of the TPC-DS application.

2.1 Requirements

2.1.1 Functional Requirements

FR 1. Perform I/O operations and report on their execution time

This is a basic requirement for analysing the performance of the data-intensive applications. The application must be able to read and write data from different storage locations.

FR 2. Perform computational tasks and report on their execution time

We require to measure the performance of the computational tasks performed by the data-intensive applications. The application must perform different computational tasks.

FR 3. Report on the execution time of the application

The application must be able to report on the execution time of the entire application and not only of individual tasks.

FR 4. Be deployable on the most popular cloud platforms

The application must be able to run on the most popular cloud platforms such as AWS, Microsoft Azure, Google Cloud, IBM. This opens up the possibility to create comparisons between the performance of different cloud platforms.

FR 5. Generate experiment workloads

The application must be able to generate the workload that must be processed. The workload must be configurable.

FR 6. Generate and process experiment logs and create meaningful graphs

In order to understand the performance of applications, they must be able to generate and process logs, creating meaningful graphs regarding the execution of different tasks and of the application.

2.1.2 Non-functional requirements

NFR 1. Usability, Simplicity: Support of experiment configurations and specify them in 1 configuration file

The applications must be able to use multiple experiment configurations. In this way, we are able to test the performance of the cloud platforms under different workloads. Moreover, all experiment configurations must be present in 1 configuration file.

NFR 2. Usability, Simplicity: Support of deployment configurations and specify them in 1 configuration file

The application must be able to support different deployment configurations such as usage of different cloud platforms, different storage systems and locations and different number of resources allocated. Moreover, all deployment configurations must be present within 1 file for better management.

NFR 3. Extensibility: Easy to expand the experiment and deployment configurations

In case new configurations are required, it must be easy for one to expand on the experiment and deployment configurations.

NFR 4. Scalability: Easy to scale

The applications must be able to scale in order to process workload of different sizes.

2.2 Sorting Application Following Multiple Workload Parallelization Techniques for Cluster and Serverless Environments: Design and Implementation

The first application is developed in python3 by the main author of the thesis and with the help of the daily supervisor. The application implements the sorting algorithm following the MapReduce framework. We chose the radix sorting algorithm, as it requires a decent amount of computational power which is enough to measure the performance of the application, while it requires to process all the data. Moreover, we chose MapReduce because is a widely use framework for distributed data-intensive applications [7]. The reason for choosing MapReduce is that it was also proven suitable for benchmarking performance of cloud platforms [21]. In this way, the application is able to process large amounts of data in a parallel, in a distributed fashion. The application interacts with a storage system in order to read the raw data, as well as to write the processed data. The data is stored in files and these files are generated using the publicly available generator on the *Ordinal* platform[5]. The application implements the generator, so the data can be made easily available (FR 5). To ensure a high level of complexity and randomness, the data that is present in each file consists of a series of characters, each character taking one of the 256 values that can be found in the extended ASCII table [2]. Important to mention is that the data is generated uniformly, meaning that each of the 256 characters is present at least once on each position in the group. The data is formatted in groups of 100 characters, each group being further referred to as a *data entity*. Moreover, each data entity is defined as a key-value pair. The key is represented by the first 10 characters and the value by the

[5]http://www.ordinal.com/gensort.html

remaining 90. The reason why the data entities are sectioned in this manner is because the key is used for sorting and the value is needed to add weight on the I/O operations.

The application follows the MapReduce model, therefore, it executes in two stages - a *map stage* and a *reduce stage*. To be more precise on the scope of each stage, the *map stage* was named as being the **Determine Categories Stage** and the *reduce stage* was named as being the **Sort Categories Stage**. Each stage consists of a number of jobs that are scheduled in the application, each job containing 3 tasks used for reading, processing and respectively, writing the data. Additionally, the application runs following configuration parameters. The configuration defines the number of files containing data to be sorted, as well as how the application must categorize the data. Each stage is further detailed below.

Stage 1. Determine Categories Stage

The first step in this stage is to schedule a job for each file that needs to be processed by the application. The first task of the job then reads the data from one of the files. After the data is read, the second task sorts all data entities by the first two characters of their keys. Following this action, each data entity is assigned to a certain category, following a rule defined by the total number of categories specified in the configuration. An important aspect is that this number should always be a multiple of 256. The reason for this is that a category can be defined by all data entities that start with a certain character, or by a portion of them.

> *example: In the case of 256 categories, each one contains all data entities starting with a certain character, while in the case of 512 categories, each contains half of all data entities starting with a certain character.*

The last task writes the sorted data to the storage system. In the end, the job returns the positions of first and last data entities within each category. A collection of data entities that belong to a certain category is referred to as a *partition*.

The entire process described above is visually described in Figure 1.

Stage 2. Sort Categories Stage

For this stage of the application, the number of jobs that are scheduled equals the number of categories defined in the configuration. The first task of each job reads all partitions that belong to a certain category, from all the files resulted from the previous stage. The second task assembles in the designated category all data entities from the partitions read by the first task. The data entities are then sorted by their keys. In the end, the third task writes the sorted data back to the storage system.

In the same manner, Figure 2 presents an overview of the processes of this stage.

Three different design approaches that were chosen for the sorting application. The first one follows a serverless architecture, where each job is handled by a cloud function. The other two approaches design the sorting algorithm as worker applications. The worker

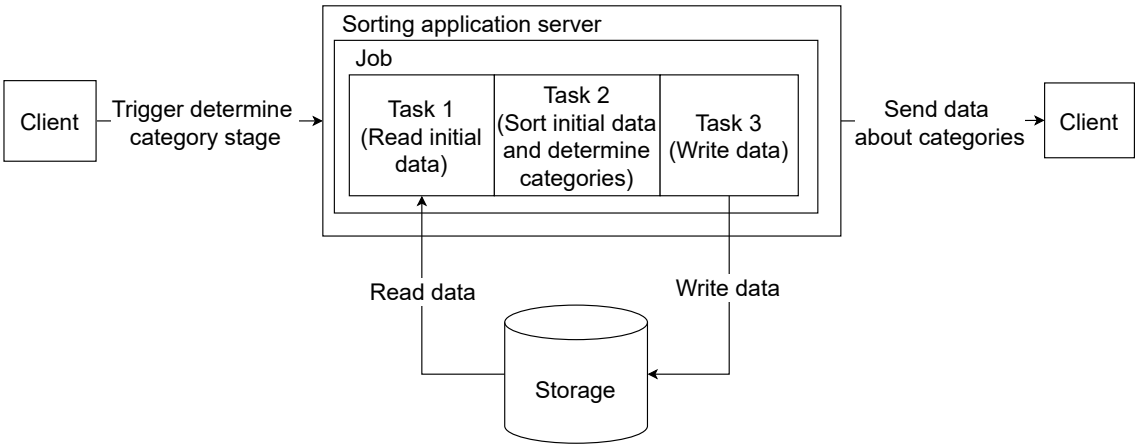

Figure 1: Overview of the Determine Categories Stage (stage 1).

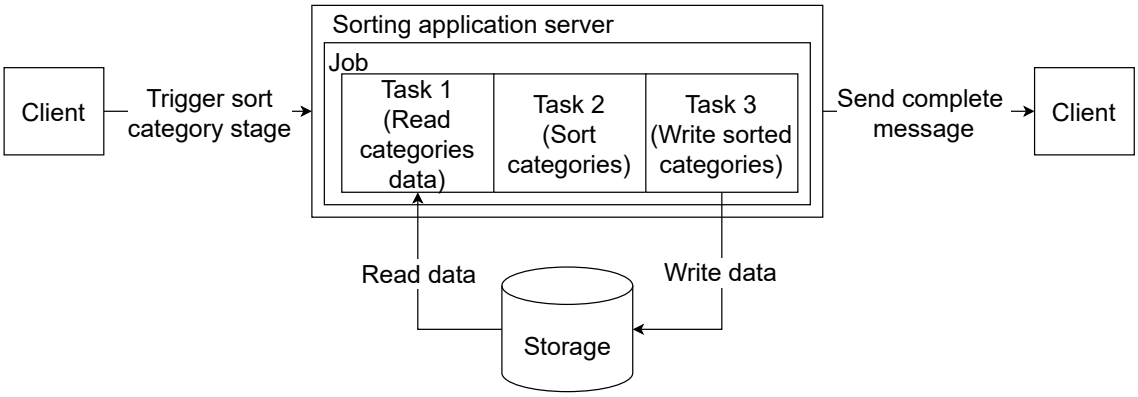

Figure 2: Overview of the Sort Categories Stage (stage 2).

applications are hosted on multiple processing nodes. In the first case, the worker application uses parallel processing in order to process multiple jobs simultaneously, while in the second case, the worker application first divides the jobs into multiple tasks and then schedules them for parallel processing.

2.2.1 Workload Parallelization Techniques for Serverless Environments - Serverless Sorting Application

The serverless sorting application is developed in python3 by the author of the thesis and with the help of the daily supervisor. It makes use of AWS Lambda cloud functions in order to process the workload. Moreover, the application also makes use of the Lithops python library [18] in order to easily manage the AWS Lambda function instances. Lithops library also enables for easy deployment of the application on other cloud platforms such as Microsoft Azure, Google Cloud and IBM Cloud (FR 4). Lithops makes the application cloud-agnostic, as it only requires a configuration file that specifies the necessary cloud endpoints where the application should be deployed, without any cloud platform specific settings. The cloud function instances are orchestrated by a client handler that assigns

the work. Figure 3 represents the design architecture of this application. The code-base of the application is stored in the GitHub repository[6].

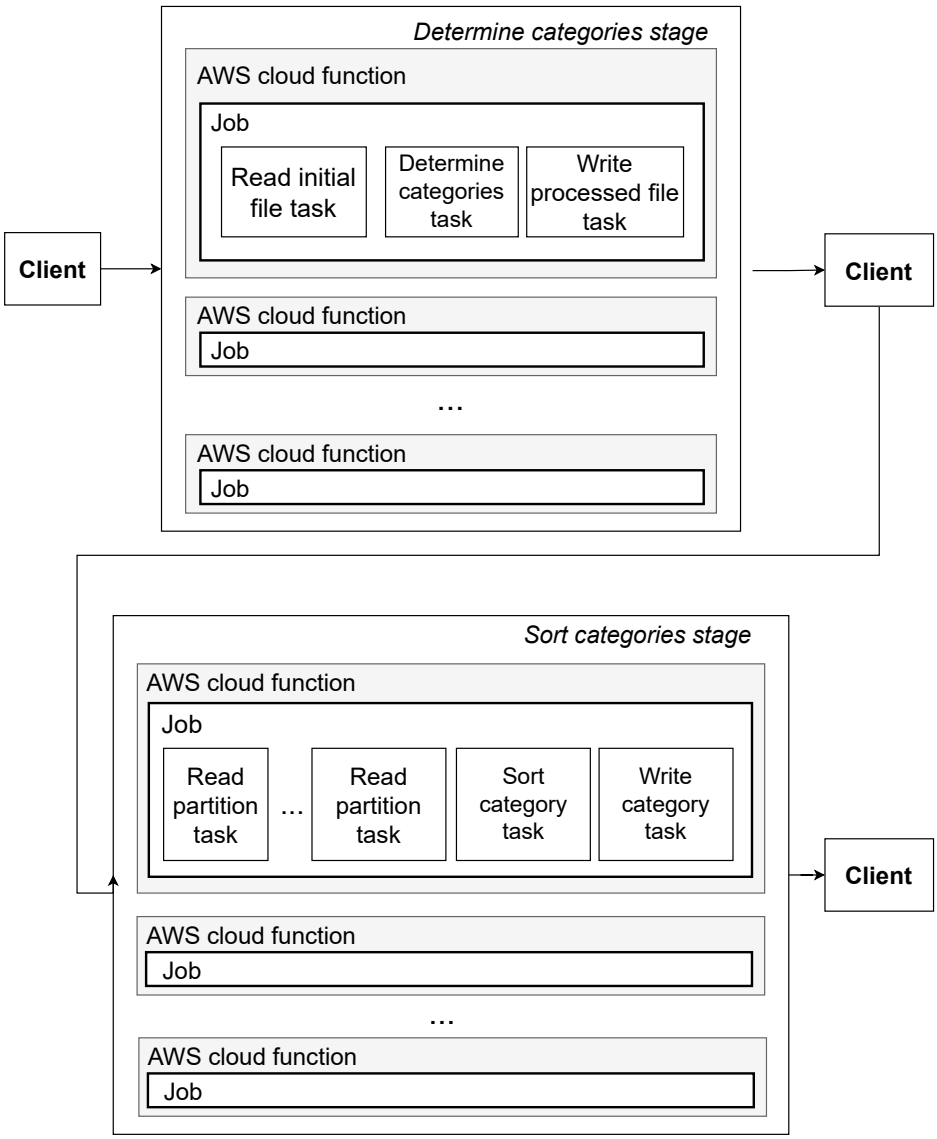

Figure 3: Architecture design of the sorting application using AWS Lambda functions.

During the *Determine Categories Stage*, to have a uniform workload, the client handler defines as many jobs as the number of initial files that contain the data to be sorted. Consequently, the client handler starts up as many cloud function instances as the jobs previously defined. Each function then reads one of the files from the storage system. Next, it sorts the data entities by the first two characters, determines the categories based on the application configuration and places the entities in the categories they belong to. Finally, the cloud function writes the sorted data entities back to the storage system and returns the first and last positions of each category to the client handler of the application.

[6]https://github.com/bgdbgd1/lithops-radix-sort

The client handler then gathers the data from all AWS Lambda function instances that were requested for the first stage and starts the *Sort Categories Stage*.

For the *Sort Categories Stage*, again for a uniform workload, the client handler triggers another batch of functions, each being responsible for sorting a single category. Each function is then required to retrieve all partitions of the given category from each file generated during the *Determine Categories Stage*, sorts the data and saves it to a file in the storage system. When completed, each cloud function notifies the client handler about the finished job without returning any data, as the algorithm is ended and no further processing is required.

2.2.2 Workload Parallelization Techniques for Cluster Environments - Worker Sorting Application with Job Parallelization

The worker sorting application with job parallelization follows a similar architecture as the serverless sorting application. The aim of this approach is to implement the sorting algorithm within a worker application that can be deployed on any computing node without requiring a serverless infrastructure. As the worker application is deployed on a computing node that has more resources than a cloud function does, we decided to enable the worker application to process the workload in parallel, instead of doing it sequentially. This decision was also taken because of the limited amount of computing nodes available. The worker application is developed in python3 by the author of the thesis and with the help of the daily supervisor. The worker sorting application uses the Flask library to implement REST APIs in order to trigger the stages of the sorting algorithm. We use Flask because it is a lightweight library that fits the purpose of this application. Multiple instances of the application are deployed on multiple computing nodes on the DAS-6 cluster of the Vrije Universiteit. The code-base of these applications is stored in a GitHub repository[7].

During the *Determine Categories Stage*, in order to create uniform workload, the client handler defines a number of jobs equal to the number of data files. The jobs are then scheduled on the worker application, which is able to process them in parallel. This is done by assigning each job to a separate process on the computing node where the worker application is deployed. If there are more jobs than the worker is able to parallelize, they are placed in a waiting queue and are picked up one by one, once a running job finishes. At this stage each job consists of three tasks that are executed sequentially. The first task reads the data entities from the file. The second task sorts the data entities by the first two characters of their keys and determines the categories to which each data entity belongs to based on the application configuration. The third task then writes the sorted data into the storage system. In the end, each job returns the first and last positions of each category it determined. When the queue is empty and all the jobs are completed, the worker application reports information about the categories determined by all the jobs that it processed back to the client handler.

During the *Sort Categories Stage*, in order to create uniform workload, the client handler defines a number of jobs equal to the number of categories that were defined in the experiment configuration. The jobs are again scheduled on the worker application, which,

[7] https://github.com/bgdbgd1/sorting-with-threads

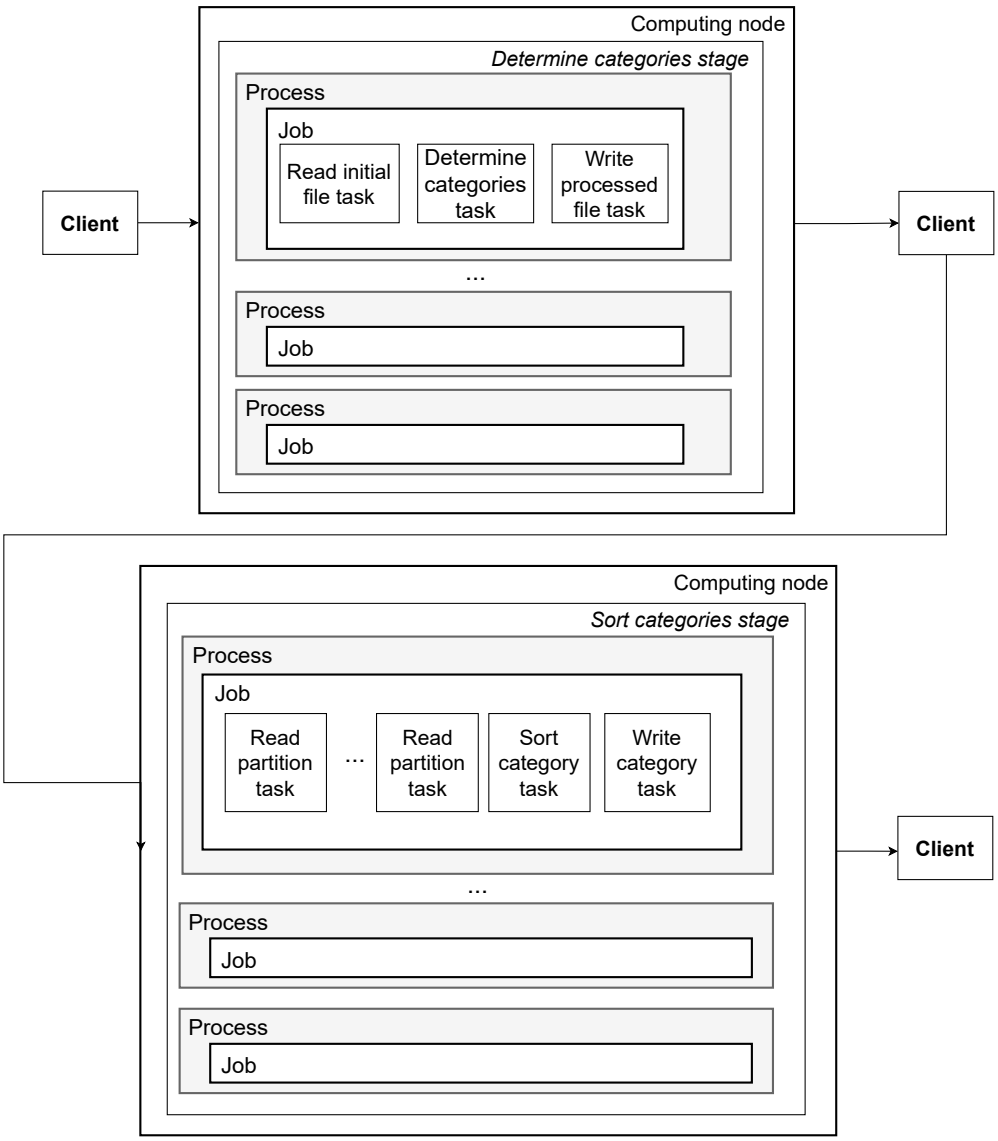

Figure 4: Overview of the worker sorting application with job pallelization.

in turn, process them in parallel. Similar to the previous stage, each job is assigned to a separate process on the computing node on which the worker application is hosted. If there are more jobs than the worker can parallelize, they are also placed in a waiting queue and are started one by one, whenever a running job completes. The jobs that are processed at this stage also consist of three tasks. First task must read all data partitions belonging to the category given to the job. The second task reassembles the category from the partitions that were read by the previous task and sorts it. The third task writes the sorted data of the category on the storage system. When the scheduling queue is empty and all jobs are completed, the worker application notifies the client handler about it.

2.2.3 Workload Parallelization Techniques for Cluster Environments - Worker Sorting Application with Task Parallelization

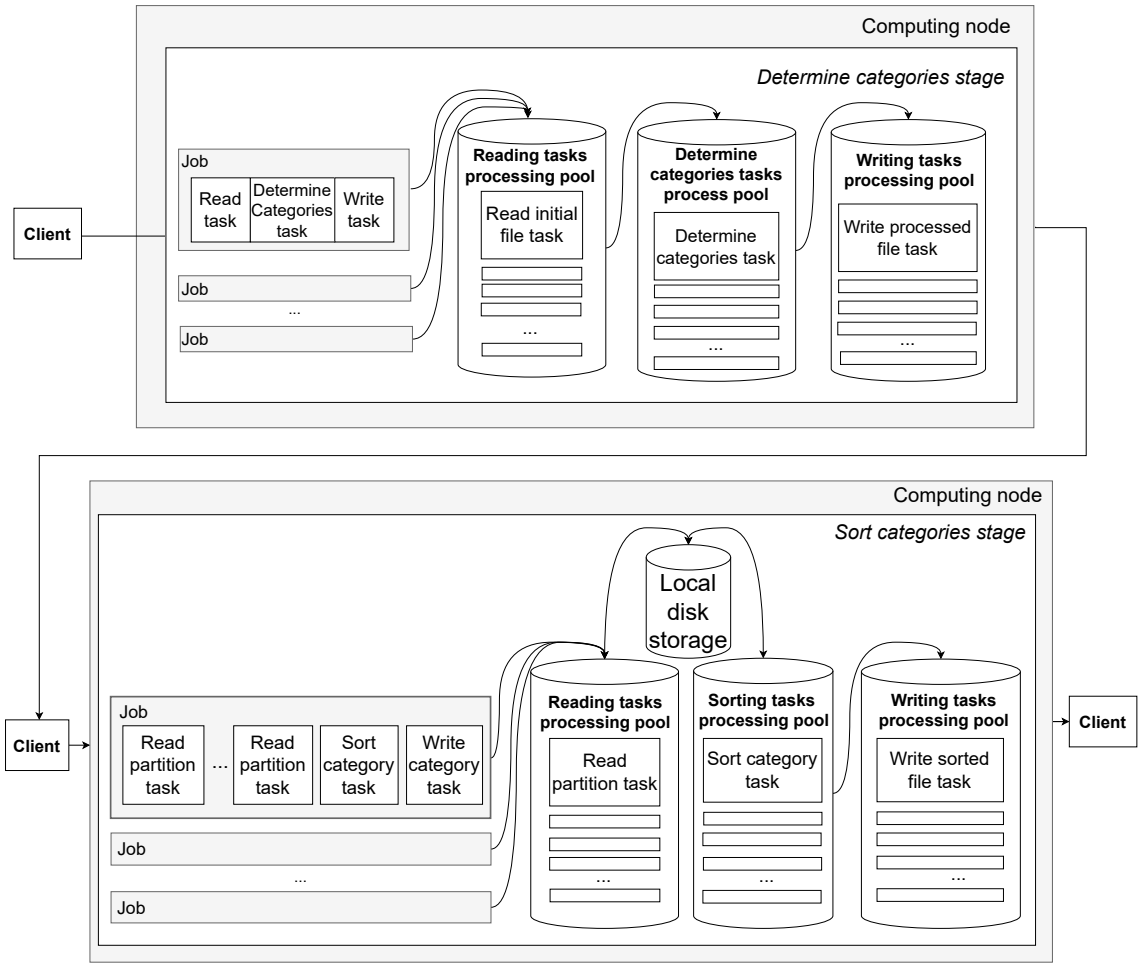

Figure 5: Overview of the worker sorting application with task parallelization.

The worker sorting application with task parallelization represents an updated version of the worker sorting application with job parallelization. This approach aims at improving the processing time of each stage of the sorting application by using the resources of the computing node more efficiently. In this sense, this approach proposes that instead of executing multiple jobs simultaneously, they must be first divided into tasks and only then the tasks to be processed in parallel.

During the *Determine Categories Stage*, in order to create uniform workload for each of the worker sorting application instance, the client handler defines a number of jobs equal to the number of files containing data to be sorted. The jobs are then sent to the worker application. Each job consists of three different tasks, a *reading task* that reads the data entities from the file, a *sort and determine categories task* that sorts the data entities based on the first two characters of their keys and determines the categories to which each

data entity belongs to, and a *writing task* that uploads the sorted data entities back to the storage system. To achieve task parallelization, the worker application defines three dedicated pools of processes, one for each type of task. At first, the worker application only schedules the *reading tasks* on the reading pool of processes. As these tasks begin to complete, they generate *sort and determine categories tasks* that are scheduled on their dedicated pool. Consequently, whenever one of these tasks finishes, it generates *writing tasks*. In the end, after all tasks are executed, the worker application returns information about the categories determined during this stage.

During the *Sort Categories Stage*, in order to create uniform workload for each of the worker sorting application instance, the client handler defines a number of jobs equal to the number of categories defined in the application configuration and they are sent to the worker application. Each job consists of a number of *read partition task* equal to the number of partitions that each category has and two more tasks, a *sort category task* and a *write category task*. Each *read partition task* reads its partition from the files on the storage system and saves the data in a temporary storage on disk. This decision was taken in order to avoid the usage of shared memory and locking mechanisms such that the data is safely accessible by the tasks requiring it. For each of the three types of tasks the worker application defines dedicated pools of processes. At start, the worker application only schedules *read partition tasks*. The main process of the worker application watches the temporary disk storage, creates a sort category task whenever all partitions of a category are present and schedules it on the dedicated pool. Consequently, as a *sort category task* is finished, a *write category task* is scheduled. When all tasks are complete, the stage finishes. The worker application returns to the client handler, informing it about the end of the stage.

2.3 TPC-DS Query Application: Design and Implementation

TPC-DS is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance. The benchmark provides a representative evaluation of performance as a general purpose decision support system [17].

The application consists of a number of python scripts that implement different TPC-DS queries. All scripts implement the Lithops library and use AWS Lambda cloud functions in order to resolve the queries. The queries are ran on data generated by the TPC-DS data generator. The application implements the generator, so the data can be easily available (FR 5). The data represents a fictive database composed of multiple tables. The data of each table is saved in CSV format on the cloud storage. Because some tables can be very large, sometimes they are split in multiple partitions. The code-base of the application is available in a GitHub repository [8].

The scripts were not developed from scratch, but an older implementation is available in the GitHub repository[9] was reused. However, a number of improvements were added:

- Migrated from *Pywren* library to *Lithops* library. Pywren is a python library very similar to Lithops that facilitates the development of serverless applications and

[8] https://github.com/bgdbgd1/tpcds-lithops-scripts
[9] https://github.com/ooq/tpcds-pywren-scripts

deployment of AWS Lambda cloud functions. However, Lithops is platform agnostic which makes it easy to deploy the same application on a different cloud platforms such as Microsoft Azure, Google Cloud and IBM Cloud (FR 4).

- Updated outdated code and libraries.
- Created the table schema. The table schema is missing from the original repository, therefore it had to be recreated using the current TPC-DS documentation.

The application follows a serverless architecture in order to resolve queries provided by the TPC-DS benchmark. As the queries are complex and they involve a big number of tables, the application runs in several stages, each stage representing a part of the original query (e.g. a subquery or a join between multiple tables). In-between each stage, a client handler orchestrates the workload and starts up Lambda cloud functions. Depending on the number of partitions of tables that are used during a certain stage, a various number of Lambda cloud functions are deployed. All four TPC-DS queries can be found in the Appendix A together with an extensive description about every stage that the application implements in order to resolve the queries.

3 Experiment Design and Configuration

This chapter presents the experiment design and configuration for the data-intensive applications used in this research and contributes to answering RQ 1 and RQ 2. In order to test the performance of these applications we design several experiments that impose different workloads. The applications are deployed on different set of resources. The worker sorting applications are deployed on the DAS-6 cluster, while the serverless applications implementing cloud functions are deployed on the AWS cloud. Moreover, the worker sorting applications implement different workload-parallelization techniques and they are deployed on DAS-6. In-depth details about the resources used by our applications in both AWS cloud and DAS-6 are presented in Section 3.3.
In order to have meaningful experiments and results, we first draw the experimental goals in Section 3.1. Following up, we present the application configurations used in running our experiments. The configurations are divided on an application basis in Section 3.2. The results of our analysis based on the execution of experiments on the data-intensive applications are presented in Chapter 4 and Chapter 5.

3.1 Experimental Goals

The main goal of this research is to **analyse the performance of data-intensive application on serverless architectures**. To reach this goal, we implement several data-intensive applications following the *Application Design and Implementation* presented in Chapter 2. The experiments plan to test different parts of the applications and they have the following goals:

1. **Goal 1: Analyse the performance of I/O operations in relation with the size of the data that must be processed as well as with the number of concurrent I/O operations for all applications.**

   For this goal, we are interested to observe whether the number of concurrent I/O operations and the size of the data to be processed impact their performance.

2. **Goal 2: Analyse the performance of processing tasks in relation with size of the data that is being processed as well as with the number of concurrent processes.**

   For this goal, we want to observe whether the size of the data that is being processed and the number of concurrent processes affect the performance of these tasks.

3. **Goal 3: Analyse the cold start times of the serverless applications using cloud functions.**

   We are interested in analysing the cold start times required by the cloud functions that get started while running our experiments. For this, we execute multiple successive runs of the same experiment and we analyse when cold starts happen and how much time they require.

4. **Goal 4: Compare the execution of the worker sorting applications with the execution of the serverless sorting applications.**

For this goal we compare the execution of the worker sorting applications with the execution of the serverless sorting applications. For a good comparison, we execute experiments following the same configurations.

3.2 Application Configuration

In this section we detail on the processes that we monitor for each data intensive application used in this research. Moreover, we present the configurations used throughout our experiments.

3.2.1 Serverless Sorting Application

The serverless sorting application is deployed on the Amazon Web Services cloud and makes use of the AWS Lambda cloud functions to process the workload. The data that is used by the application is stored in Amazon Simple Storage System (S3). The use of the cloud functions imply an extra step in each of the two stages of the application, as these functions require to be deployed before they can start the execution. For a good overview of the measurements that we perform, we detail on all the processes involved in each stage of the serverless sorting application. The results of the analysis based on the application configuration described below are presented in Chapter 5.

3.2.1.1 Determine Categories Stage

- **Init stage**

  This is the stage where the execution environment for the cloud functions gets created. The cloud environment provisions the application with all the necessary resources, extensions and code that the function must execute. The time required for this stage to complete is automatically reported by Lambda in the CloudWatch logs. Measuring the time required for this stage will help us determine the cold start of the cloud functions.

- **Invoke stage**

  This stage measures the time it takes for each AWS Lambda function to run to completion. As defined in the previous chapter, during the *Determine Categories Stage* each cloud function is assigned one job. In turn, each job consists of three tasks. In order to get a good overview of the execution times of these cloud functions, we measure both the completion time of every job, as well as every task.

  - **Read Initial Data Task**

    This task is the first one that runs in a job and it is required to read the data entities from the files stored in the S3 cloud storage.

  - **Sort and Determine Categories Task**

    This task is responsible for sorting the data entities by the first two characters of their keys and determines the categories to which each data entity belongs to.

– **Writing task**

After the sorting is done, this task writes the data entities back to the AWS S3 storage system, a location accessible to the cloud functions responsible to execute the *Sort Categories Stage*.

3.2.1.2 Sort Categories Stage

- **Init stage**

  Similarly to the Init stage from the *Determine Categories Stage*, this is where the execution environment for the AWS Lambda cloud functions gets created.

- **Invoke stage**

  This is the stage at which the actual function execution takes place. As defined in the previous chapter, during the *Sort Categories Stage* each cloud function must execute one job, with the difference that this time each job consists of a number of *Read Partitions Tasks*, a *Sort Category Task* and a *Write Category Task*.

  – **Read Partition Task**

  Each *read partition task* is responsible for reading a partition belonging to a certain category, assigned per job, from one file generated in the previous stage. At this stage, we analyse the time required to read each partition, as well as the total time required for reading all partitions of a category.

  – **Sort Category Task**

  The purpose of this task is to reassemble the category using the partitions read by the previous reading tasks. Moreover, the task sorts the data entities from the category, based on their entire keys.

  – **Write Category Task**

  At this point, the data entities are fully sorted and the sole purpose of this task is to write them to files on the AWS S3 storage.

To be able to analyse the performance of the serverless sorting application in the AWS cloud environment, multiple application configurations are used. An overview of the configurations can be observed in Table 1. For each application configuration 1 experiment with 10 experiment runs are being executed. The size of each input file will not go higher than 1GB, as it increases the reading, processing and writing times and requires that more memory is assigned to each AWS Lambda cloud function. Consequently, higher costs for the experiment runs are to be expected.

3.2.2 Worker Sorting Applications with Diverse Workload Parallelization Techniques

The worker sorting applications are deployed on the computing nodes of the DAS-6 cluster owned by Vrije Universiteit. Multiple instances of the worker applications are deployed

Total data size	Number of data files	Size of each data file (MB)	Number of categories defined
100	1000	100	256
100	1000	100	512
100	500	200	256
100	500	200	512
100	100	1000	256
100	100	1000	512
500	1000	500	256
500	1000	500	512
500	500	1000	256
500	500	1000	512
1000	1000	1000	256
1000	1000	1000	512

Table 1: Application configuration of the Serverless Sorting Application.

on the cluster, each instance being deployed on a separate computing node. The worker sorting applications are able to parallelize their work, assigning jobs or tasks to separate processes on the node. Moreover, the data used by the worker applications is stored on another node present on the DAS-6 cloud. The data is made accessible through MinIO, a High Performance Object Storage compatible with Amazon S3 storage.

The worker sorting applications implement the sorting algorithm in two stages, the *Determine Categories Stage* and the *Sort Categories Stage*. To analyse the performance of the worker applications we look at the execution times of the jobs running during each stage, as well as the execution times of each individual task contained in a job. The results of the analysis based on the application configuration described below are presented in Chapter 4.

3.2.2.1 Determine Categories Stage

- **Read initial data task**

  This is the first task that runs during this stage. The task reads the data entities from the data files saved on the storage system.

- **Sort and determine categories task**

  This task must sort the data that was read by the previous task.

- **Writing task**

  After the data entities are sorted, this tasks writes the data entities back to the storage.

3.2.2.2 Sort Categories Stage

- **Read partition task**

  Each read partition task is responsible for reading a partition belonging to a certain category, assigned per job, from one file generated in the previous stage.

- **Sort category task**

  The purpose of this task is to reassemble the category using the partitions read by the previous reading tasks. Moreover, the task sorts the data entities from the category, based on their entire keys.

- **Write category task**

  This task is responsible for writing the sorted category back to the storage.

To be able to analyse the performance of the worker sorting applications with diverse workload parallelization techniques, multiple application configurations are used. Several parameters of the application configuration were varied: the number of initial data files, the size of each initial data file, the number of categories that must be defined and the number computing nodes used. In all configurations, the total size of the data that must be sorted is of 100 GB. In some configurations we aim for a full parallelization of the workload, but because there is only a limited number of nodes that are available, with some of the configurations, full parallelization is not possible. An overview of the application configurations that we used can be observed in Table 1.

3.2.3 TPC-DS Query Application

This application is composed of four python3 scripts that are executed in the Amazon Web Services cloud environment. Each script implements a different TPC-DS query. Each query is divided in several stages and makes use of AWS Lambda cloud functions to execute them.
Each script implements a query through a number of stages. For each stage, a number of cloud functions are activated. For all AWS Lambda cloud functions, both the **Init stage** and the **Invoke stage** are analysed. The **Init stage** is the stage at which the execution environment is created, meaning that the resources are gathered and the code base of the function is downloaded and compiled. The **Invoke stage** is the stage at which the function actually executes. In this sense, depending on which stage of a query a cloud function is assigned to, each cloud function executes different code. However, all scripts are designed in such way that at each stage, each cloud function is required to read some data from the S3 cloud storage, process it and store the processed data. As all runs of an application configuration use the same data set, all cloud functions deployed for the same stage have to do the same amount of work. For each cloud function, no matter what

Total data size	Number of data files	Size of each data file (MB)	Number of categories	Number of computing nodes
100	1000	100	256	11
100	1000	100	512	11
100	1000	100	512	22
100	100	1000	256	11
100	100	1000	512	11
100	100	1000	512	22

Table 2: Application configuration of the Worker Sorting Applications with Diverse Workload Parallelization.

query stage it executes, there are 3 tasks to be monitored: **Read data task**, **Process data task** and **Write data task**.
All four scripts are part of the experiments and they follow two different experiment configurations: a database total size of **10 GB** and a database total size of **100 GB**. Moreover, 10 runs of the same application configuration is being executed. As the database size grows, the size of each table grows as well. In order to keep the functions within affordable execution prices, each table is split within multiple CSV files. In this sense, each cloud function is designated one CSV file to process. Moreover, at each stage, a different number of files are generated as a result. Consequently, as the database size grows, more traffic is generated, therefore more reads and writes to/from the S3 cloud storage are required. The results of the analysis based on the application configuration described below are presented in Chapter 5.

3.3 Experiment Design Across Diverse Resources

3.4 Serverless Sorting Application

The serverless sorting application is deployed in the AWS cloud environment and makes use of the AWS Lambda cloud functions. Every Lambda function that runs during the experiments is configured to use 2048 MB RAM memory. Although this can mean over-provisioning in some cases, the CPU power that each cloud function can access depends on the memory size, therefore, it was decided to keep the resource usage uniform across all experiments, in order to create meaningful comparisons between runs of different configurations. In this sense, 2048 MB of RAM memory satisfies all experiment configurations that were selected.

No. processes reading tasks pool	No. processes processing tasks pool	No. processes writing tasks pool
8	8	8
6	10	4
4	10	10

Table 3: Distribution of number of processes per pool for the *Determine Categories Stage* of the Worker Sorting Application with Task Parallelization.

No. processes reading tasks pool	No. processes processing tasks pool	No. processes writing tasks pool
18	4	2
14	6	4
10	8	6

Table 4: Distribution of number of processes per pool for the *Sort Categories Stage* of the Worker Sorting Application with Task Parallelization.

3.5 Worker Sorting Applications

The worker sorting applications are deployed on the computing nodes of the DAS-6 cluster owned by Vrije Universiteit. This cloud contains up to 34 nodes, each having 24 CPU cores. However, the experiments do not require the usage of all nodes. Multiple instances of the worker applications are deployed on the cluster, each instance being deployed on a separate computing node. The worker applications are able to parallelize their work, assigning jobs or tasks to separate processes on the node. Moreover, the data used by the worker applications is stored on another node present on the DAS-6 cloud. The data is made accessible through MinIO, a High Performance Object Storage compatible with Amazon S3 storage.

As each computing node has 24 CPU cores, we use a maximum of 24 parallel processes. Moreover, for our experiments we use use two variations of the numbers of computing nodes: 11 and 22. This means that we can have a total of 264 processes available or 528 processes available. We chose those numbers in order to reach full parallelization when having 256/512 categories to sort during the *Sort Categories Stage*.

For the worker sorting application with job parallelization each instance is able to process 24 jobs in parallel. For the worker sorting application with task parallelization this means that each application instance is able to process 24 tasks in parallel. However, for the latter one we define three pools of processes: 1 pool dedicated to the reading tasks, 1 pool dedicated to the processing tasks (sorting and determine categories tasks) and 1 pool dedicated to the writing tasks. In this case, we decided to run experiments with different number of processes per pool in order to observe how the performance of the application changes and to find the most efficient configuration. Table 3 presents the variations in

number of processes assigned to each pool for the *Determine Categories Stage* and Table 4 presents the number of processes assigned to each pool for the *Sort Categories Stage*.

3.6 TPC-DS Query Application

This application is composed of three python3 scripts that are executed in the Amazon Web Services cloud environment. Each script implements a different TPC-DS query. Each query is divided in several stages and the application makes use of AWS Lambda cloud functions to execute them.
Similar to the Serverless Sorting Application, we chose to use the same resource configurations, although it can mean over-provisioning in some cases.

4 Performance Analysis and Comparison of the Worker-Sorting Application Running on DAS-6 Resources with Diverse Workload Parallelization Techniques

Question 1: What is the performance impact of sorting big data in cluster environments?

This chapter presents the performance analysis of the worker sorting application running on DAS-6 cluster. The worker sorting application implements the sorting algorithm as an application that can be deployed on a virtual machine. In our case, we deployed the worker sorting application on virtual machines hosted on the computing nodes of the DAS-6 Vrije Universiteit cluster. The worker sorting application does not process workload sequentially, but it does it rather in parallel. The reason for this is that we want to maximize the resource usage of the computing nodes where the worker sorting application is hosted. Moreover, we implemented the worker sorting application following two different parallelization techniques, namely job parallelization and task parallelization.

4.1 Performance Analysis of the Worker-Sorting Application with Job Parallelization

Question 1.1: What is the performance of data-intensive sorting application with job parallelization in cluster environments?

For the *Worker Sorting Application with Job Parallelization*, the workload is delivered in the form of jobs and the application is able to parallelize them by assigning each job to a separate process on the computing node. In turn, each job consists of multiple tasks that are generally referred as reading tasks, processing tasks and writing tasks. In this section the we present our observations based of the experiments ran with this application following multiple experiment configurations. The section is structured based on observations at the task level, as well as observations on the entire application execution.

4.1.1 Large I/O Operations

The sorting application relies on I/O operations for both retrieving the data that must be sorted as well as submitting the sorted data (Figures 1 and 2) (**FR 1**). Figure 6 indicates the times necessary for reading task of the first stage of the sorting application to download the initial data. There are 100 jobs reading tasks per experiment run, each of them being required to download 1GB of data from storage. Moreover, the figure presents the data collected during 10 experiment runs. Likewise, Figure 7 indicates the upload times of the data processed by the writing tasks of the first stage of the sorting application, for the same duration of 10 experiment runs. Each task saves the data in one file of 1GB for every experiment. Similarly, Figures 8 and 9 represent the execution times required by 1000 reading tasks and 1000 writing tasks per experiment run for data of 100MB per task. The data is collected during 10 experiment runs.

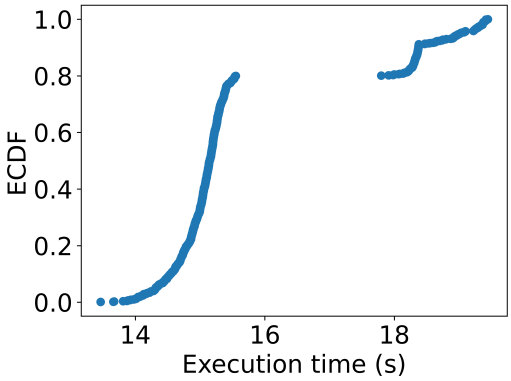

Figure 6: Execution time of the reading tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial files of 1GB each.

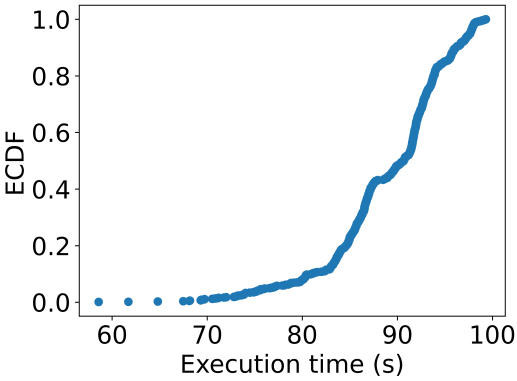

Figure 7: Execution time of the writing tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial files of 1GB each.

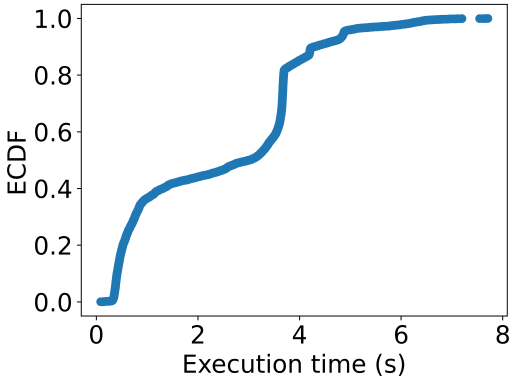

Figure 8: Execution time of the reading tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial files of 100MB each.

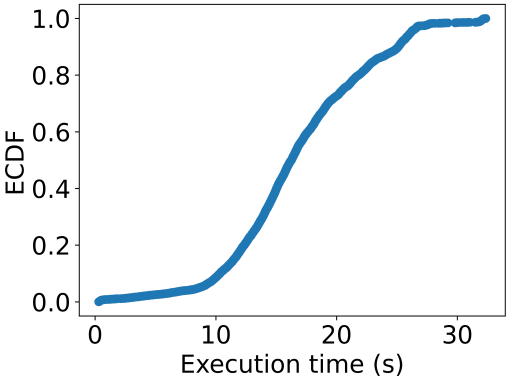

Figure 9: Execution time of the writing tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial files of 100MB each.

The reading of the data varies between 0.10 seconds and up to 8 seconds in the case of 100MB files. However, 80% of the tasks complete in less than 4 seconds, whereas only 20% of the tasks take between 4 and 8 seconds to complete. The same phenomenon can be noticed in the case of 1GB files. Each task requires between 13.5 seconds and 19.5 seconds for reading a data file of 1GB, with 80% of the reading tasks finishing in maximum 15.5 seconds and only 20% of the tasks requiring between 18 and 19.5 seconds.

It can be observed that the writing tasks take longer than the reading tasks. Here, in the case of 100MB files, the fastest execution time is of 0.10 seconds and the slowest is of 33 seconds, while for the 1GB files, the execution time varies between 57 seconds and 100 seconds. Another aspect worth mentioning is that, unlike the case of the reading tasks,

the writing tasks have a more linear variability.

The variability of the reading and writing tasks is a consequence of the shared bandwidth. Each computing node has 100Gbit of available bandwidth. However, the bandwidth is shared between the number of concurrent connections. During the experiment runs, it was ensured that no other connections were active. However, the storage system for the data files is also hosted on a computing node with a 100Gbit/s internet connection. Consequently, it means that each task must share a total of 100Gbit bandwidth with all the other reading and writing tasks that are running simultaneously. This is a consequence of the fact that the reading tasks are the first tasks in the determine categories stage jobs. In this sense, the reading tasks start and finish in approximately the same time. On the contrary, the writing tasks are the last tasks to run within the jobs of the determine category stage. As it can be observed in Figures 14 and 15, the second task within the determine categories stage job, which amounts to sorting initial data and determining categories, also shows some variability in its execution time. Consequently, the writing tasks do not start all in the same time, but rather at different points in time. As a result, the execution time of the writing tasks is more linear, with faster execution for the first tasks and longer execution for the later ones.

This analysis contributes to the observation **O-1** and towards achieving **Experimental Goal 1**.

4.1.2 Small I/O Operations

During the second stage the sorting application executes jobs where each of them is required to read a category determined during the first stage, sort the category and write it back into the storage (Figure **Stage 2.**) (**FR 2**). Each category is composed of multiple partitions. Each partition represents a small part of a file that was uploaded during the first stage. Depending on the configuration, a category can be composed of either 100 partitions or 1000 partitions. There are 100 partitions per category when the configuration contains 100 initial data files of 1GB each. Consequently, if the configuration requires to define 256 categories, each partition is approximately 3.9 MB and there is a total of 25 600 partitions. Moreover, if the configuration requires to define 512 categories, each partition is approximately 1.95 MB with a total of 51 200 partitions. On the other hand, there are 1000 partitions per category when the configuration contains 1000 initial files of 100MB each. In this sense, when the configuration defines 256 categories, each partition is approximately 0.39 MB with a total of 256 000 partitions. Similarly, when the configuration defines 512 categories, each partition is approximately 0.195 MB with a total of 512 000 partitions.

Figures 10 and 11 represent the execution times of the tasks responsible for reading each data partition from 100 files of 1 GB each, during all 10 experiment runs, while having defined 256, respectively 512 categories. Similarly, Figures 12 and 13 represent the execution times of the tasks responsible for reading each data partition from 1000 files of 100 MB each, during all 10 experiment runs, while having defined 256, respectively 512 categories. As it can be observed, all four cases have very similar results. In all cases, almost 100% of the I/O operations finish within 0.25 seconds. Moreover, outliers can be observed, as a very small percentage of the reading tasks take longer - in some cases up to 4 seconds.

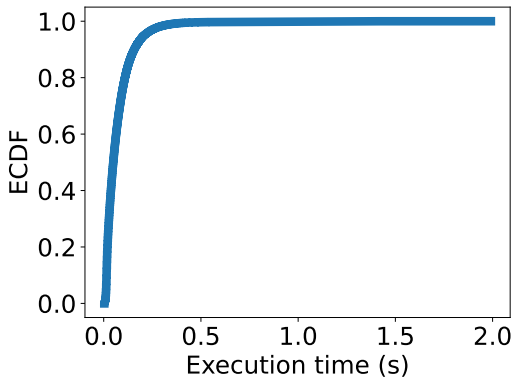

Figure 10: Execution time of the tasks reading each partition during the *Sort Categories Stage* during 10 runs of an experiment with a configuration 100 data files of 1GB each and 256 categories.

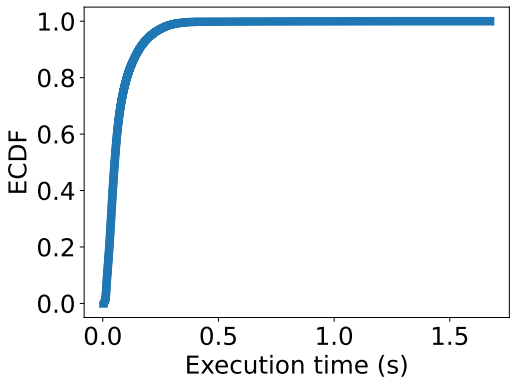

Figure 11: Execution time of the tasks reading each partition during the *Sort Categories Stage* during 10 runs of an experiment with a configuration 100 data files of 1GB each and 512 categories.

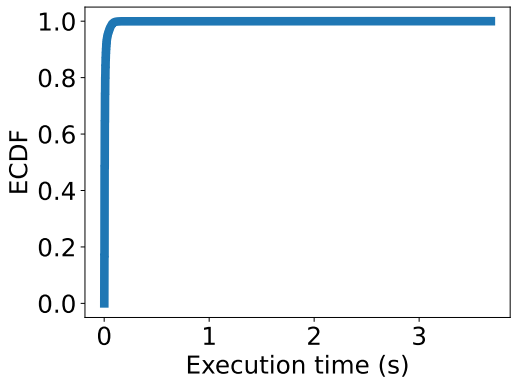

Figure 12: Execution time of the tasks reading each partition during the *Sort Categories Stage* during 10 runs of an experiment with a configuration 1000 data files of 100MB each and 256 categories.

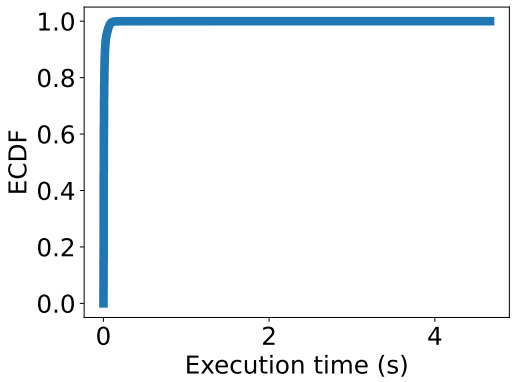

Figure 13: Execution time of the tasks reading each partition during the *Sort Categories Stage* during 10 runs of an experiment with a configuration 1000 data files of 100MB each and 512 categories.

Compared to the large I/O operations discussed in the previous section, consistency in the execution time can be observed. This is due to the small sizes of the partitions that must be downloaded by each job.

This analysis contributes to the observation **O-1** and towards achieving **Experimental Goal 1**.

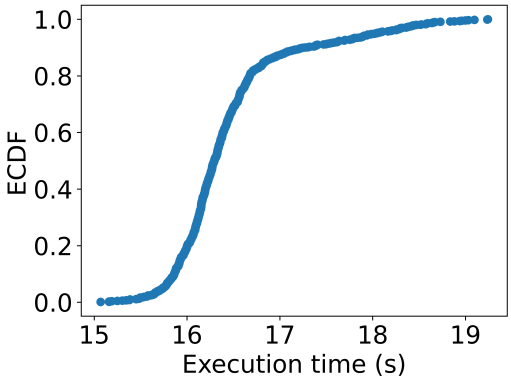

Figure 14: Processing time for sorting each of the 256 categories by the first 2 bytes within 100 files of 1GB each per experiment run during the *Determine Category Stage* within 10 experiment runs.

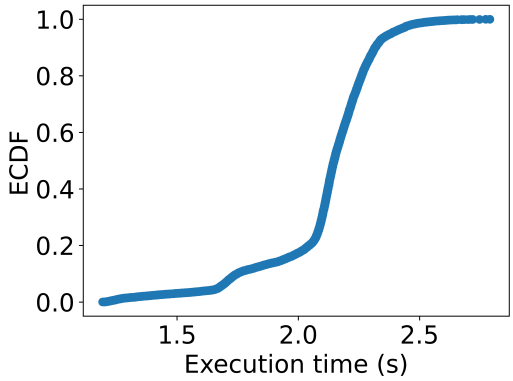

Figure 15: Processing time for sorting each of the 256 categories by the first 2 bytes within 1000 files of 100MB each per experiment run during the *Determine Category Stage* within 10 experiment runs.

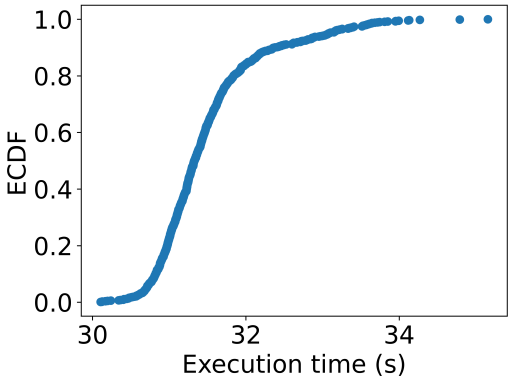

Figure 16: Processing time for sorting and determining each of the 256 categories by the first byte within 100 files of 1GB each per experiment run during the *Determine Category Stage* within 10 experiment runs.

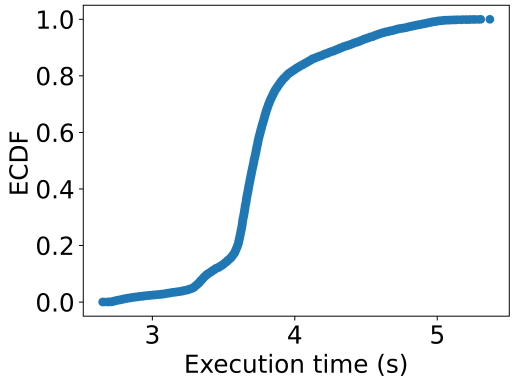

Figure 17: Processing time for sorting and determining each of the 256 categories by the first byte within 1000 files of 100MB each per experiment run during the *Determine Category Stage* within 10 experiment runs.

4.1.3 Processing Tasks

Besides the I/O operations related tasks, the jobs of the sorting application also contain tasks that process the data. During the first stage of the application each job is required to sort the data that was previously read and to determine the all categories required by the experiment configuration (**Stage 1.**) (**FR 2**). Moreover, during the second stage, after downloading all partitions of a category, each job is required to sort the category.

Figures 14 and 15 represent the processing time for sorting 1GB of data and 100MB of

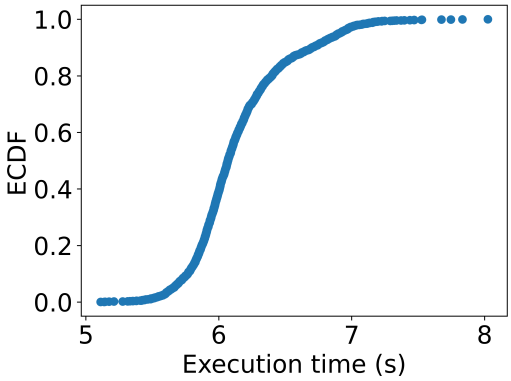

Figure 18: Processing time for sorting each of the 256 categories within 100 files of 1GB each per experiment run during the *Sort Category Stage* within 10 experiment runs.

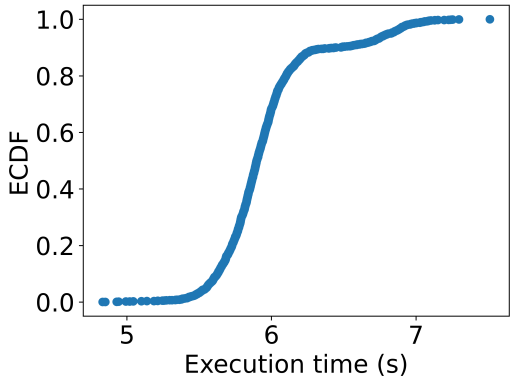

Figure 19: Processing time for sorting each of the 256 categories within 1000 files of 100MB each per experiment run during the *Sort Category Stage* within 10 experiment runs.

data respectively, during the first stage of the application within 10 different experiment runs. It can be observed that for the majority of the tasks it takes between 16 and 17 seconds to process 1 GB of data. On the other hand, it takes between 2 and 2.3 seconds for the majority of tasks to process 100 MB of data.

Figures 16 and 17 represent the processing time for sorting and determining the categories within 1GB of data and 100MB of data respectively, during 10 experiment runs. As it can be observed the two graphs have a similar curve, with the majority of tasks finishing between 30 and 35 seconds for 1GB of data and between 2.5 and 5.5 seconds for 100MB of data.

Figures 18 and 19 represent the execution time of the tasks sorting each category during 10 different experiment runs during the second stage of the application. As each category resembles to the same size, despite the number and size of the initial data files, it can be observed that the graphs are very similar, with most of the tasks finishing between 5.5 and 6.5 seconds. However, there are small differences, depending on the complexity of the shuffle of the data to be sorted.

This analysis is summarized in observation **O-2** and participates towards achieving **Experimental Goal 2**.

4.1.4 Application Execution Time

The sorting algorithm is designed to execute in two stages, *Determine Categories Stage* and *Sort Categories Stage*, and they are implemented in the worker sorting application (**FR 4**. For each stage the worker application must process a number of jobs, depending on the application configuration. Moreover, the worker application cannot proceed to the second stage until all the jobs related to the first stage are completed. In turn, the application execution is finished only when all the jobs related to the second stage finish. Each job consists of multiple tasks that are referred to as reading tasks, processing tasks

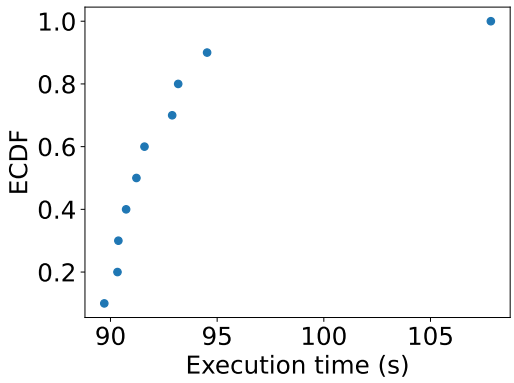

Figure 20: Execution times for the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial data files of 100MB each and 11 computing nodes.

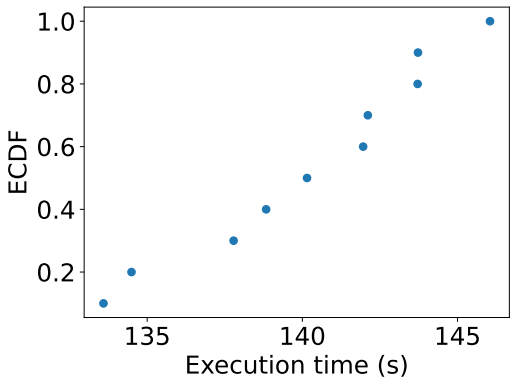

Figure 21: Execution times for the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial data files of 1GB each and 11 computing nodes.

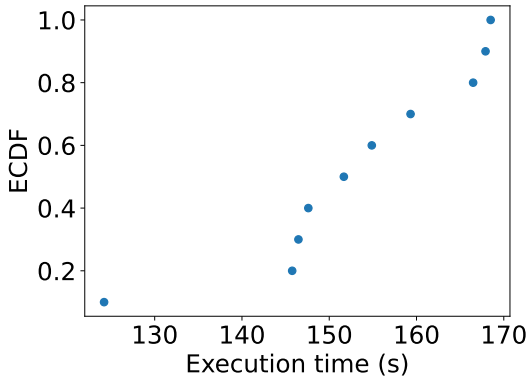

Figure 22: Execution times for the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial data files of 100MB each and 22 computing nodes.

and writing tasks. In this sense, the execution time of a job is directly affected by the execution of its tasks.

Figures 20, 21 and 22 present the execution times of the *Determine Categories Stage* during 10 runs of experiments using 3 different configurations. For the experiment used in Figure 20 the configuration specifies that 11 computing nodes are used to process 1000 data files of 100MB each, in Figure 21 the configuration specifies that 11 computing nodes are used to process 100 data files of 1GB each and in Figure 22 the configuration specifies that 22 computing nodes are used to process 1000 data files of 100MB each. It can be observed that in all of the cases the execution time of the *Determine Categories Stage* is

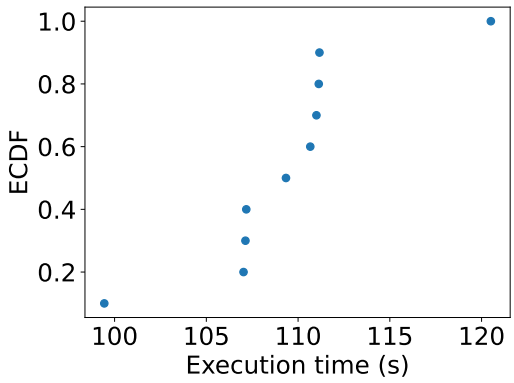

Figure 23: Execution times for the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial data files of 100MB each, 256 categories to be sorted and 11 computing nodes used.

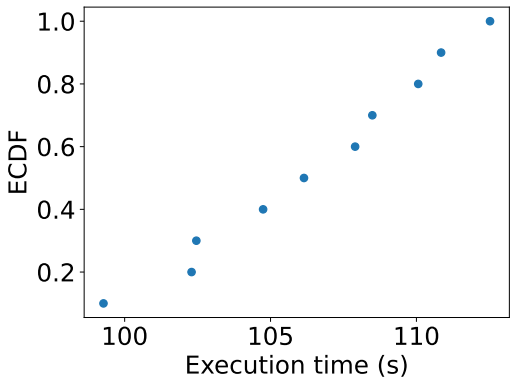

Figure 24: Execution times for the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial data files of 1GB each, 256 categories to be determined and 11 computing nodes.

not constant, but it rather varies. In Figure 20 it can be noticed that in 9 out of 10 the stage completion time had a variation of 5 seconds, with the lowest being 90 seconds and the highest being 95 seconds. However, there was 1 experiment run that required 107.5 seconds to complete the stage. For an experiment configuration of 100 files of 1 GB each (Figure 21) the stage completion times has higher variation with the lowest being of 133.8 seconds and the highest being of 146 seconds. In the last case, where the experiment configuration requires that 22 computing nodes should process a total of 1000 data files of 100MB each, in 9 out of 10 runs the stage execution time has a variation of 25 seconds, with the lowest being 145 seconds and the highest being 170 seconds, and in 1 run it only required 125 seconds. While figures 20 and 21 present the execution times of experiments running configurations with the same total size of data to be processed, 100GB, and the same amount of computing nodes used, 11, with the difference that in one case the data is split among 1000 files of 100 MB each and in the second case the data is split among 100 files of 1GB each, it can be noticed that it was faster to process 1000 files of 100 MB each rather than 100 files of 1 GB each. Moreover, comparing Figures 22 and 20 where the experiment configurations required that 1000 data files of 100 MB each must be sorted, with the difference that in the first case 22 computing nodes were involved and in the second case there were only 11 computing nodes used, it can be observed that while more servers are used, the stage execution time increases. This is the cause because each node, including the storage node, has a limited bandwidth which must be shared with all concurrent connections.

Figures 23, 24, 25, 26, 27 and 28 present the execution times of the *Sort Categories Stage* in different experiment configurations such as different number of initial data files used, different sizes of the initial data files, different categories to be sorted and different number of computing nodes. It can be observed that in all of the cases there the stage execution time is variable. It is interesting to note that in the experiments running a configuration

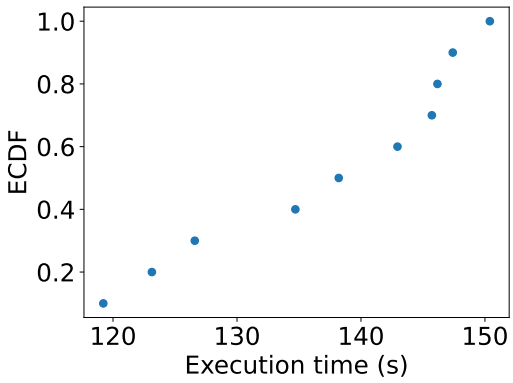

Figure 25: Execution times for the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial data files of 100MB each, 512 categories to be sorted and 11 computing nodes.

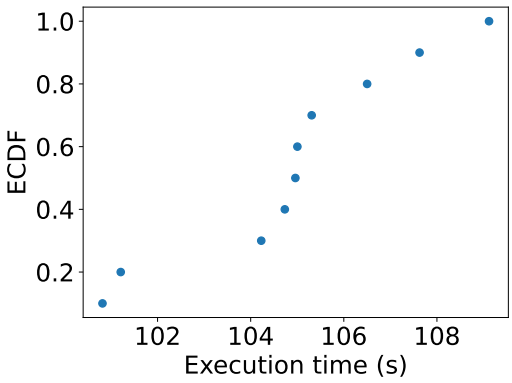

Figure 26: Execution times for the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial data files of 1GB each, 512 categories to be sorted and 11 computing nodes.

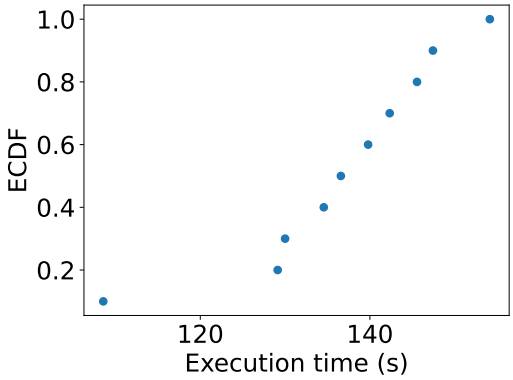

Figure 27: Execution times for the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial data files of 100MB each, 512 categories to be sorted and 22 computing nodes.

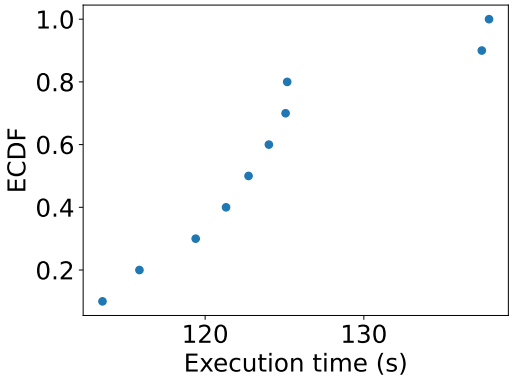

Figure 28: Execution times for the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial data files of 1GB each, 512 categories to be sorted and 22 computing nodes.

that requires 256 data categories to be sorted, no matter whether the data is split among 1000 files of 100MB each or 100 files of 1GB each, the stage execution times are very similar with a variation between 102 and 112 seconds. However, in the cases at which the experiment configuration requires 512 categories to be sorted, it can be noticed that the stage execution time is lower when the data is split among 100 files of 1GB each, rather than when the data is split among 1000 files of 100 MB each. Moreover, it requires less time for the *Sort Categories Stage* to complete when there are only 11 computing nodes executing (Figure 26), than when there are 22 computing nodes executing (Figure 28).

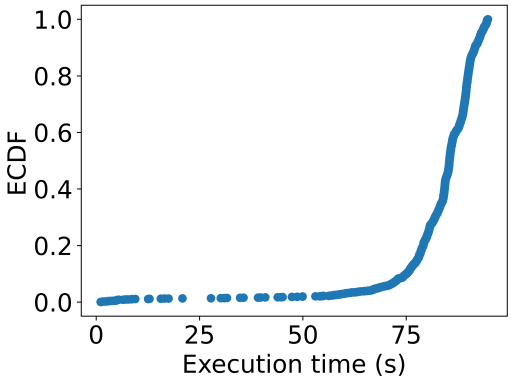

Figure 29: Execution times for the writing tasks of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial data files of 1GB each, 512 categories to be sorted and 11 computing nodes.

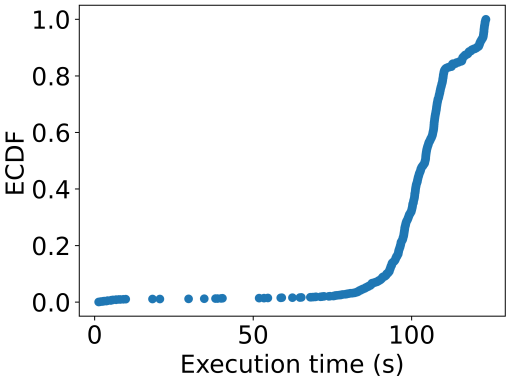

Figure 30: Execution times for the writing tasks of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial data files of 1GB each, 512 categories to be sorted and 22 computing nodes.

The difference in the execution times of the two cases is mainly generated by the slower executions of the writing tasks. As it can be observed in Figure 29, when using only 11 computing nodes, it takes 90 seconds for the slowest task to complete. However, when using 22 computing nodes, it requires up to 120 seconds for the slowest writing task to complete (Figure 30). This is again a caused by the numerous writing tasks executing concurrently, thus sharing the bandwidth.

This analysis is summarized in observation **O-3** and participates towards achieving **Experimental Goal 4**.

4.1.5 Observations

In this section we analysed the execution of the worker application using job parallelization. Multiple instances of the worker application were deployed on the DAS-6 cluster, each instance being hosted on one computing node. Moreover, the data storage was hosted on a different node and made accessible through MinIO, a High Performance Object Storage compatible with Amazon S3 storage. A list of the main observations on the execution of different tasks during our experiments on the worker sorting application with job parallelization is presented below:

- **O-1. I/O Operations Tasks**

  In section 4.1.1 we analysed the execution of the large I/O operations, in Section 4.1.2 we analysed the execution of small I/O operations, while in Section 4.1.3 we analysed the execution of the processing tasks. We define large I/O operations by referring to reading and writing data objects of sizes varying from 100 MB and up to 1GB from/to storage. Similarly, we refer to small I/O operations by referring to reads and writes of data objects with sizes varying from 0.195 MB and up to 3.9

MB. Moreover, we refer to the sorting and category determination tasks as processing tasks.

On one hand, we observed that for the large I/O operations a major drawback is made by the available bandwidth. As more jobs are executed simultaneously, the available bandwidth for each connection gets smaller, thus requiring more time for each job to finish their I/O operations. In this context, we observed a high variability in the execution of the reading and writing tasks of the jobs, more precisely of up to 8 seconds for the reading tasks and up to 40 seconds for the writing tasks. On the other hand, for the small I/O operations the execution time was, in general, constant, due to the fact that those tasks did not require a lot of bandwidth. More than 99% of the small I/O operations completed instantly. However, we could observe a small number of outliers where few tasks required up to 4 seconds to complete.

- **O-2. Processing Tasks**

  We observed variability in the execution of the the processing tasks as well. However, the variability in the execution time was only of a couple of seconds and not of tens of seconds as in the case of the large I/O operations tasks.

- **O-3. Application Execution Time**

  The worker sorting application with job parallelization executes in two stages. In order to complete, both stages must be finished. Moreover, for each stage to complete, all jobs generated for that stage must be successfully executed. In turn, each job is composed of I/O operations tasks and processing tasks. As we previously observed, each type of task has a certain variability in its execution time. The execution time of each application stage and of the entire application is highly influenced by the variability of the tasks executed. As a result, in Section 4.1.4 we could observe a variability of up to 40 seconds in the completion time of a stage.

4.2 Performance Analysis of the Worker-Sorting Application with Task Parallelization

Question 1.2: What is the performance of data-intensive sorting application with task parallelization in cluster environments?

For the *Worker Sorting Application with Task Parallelization*, the workload is delivered in the form of jobs, just as in the previous case. Different than the previous implementation, this application first divides all the jobs that it must process into multiple tasks, namely reading tasks, processing tasks and writing tasks. Moreover, the application creates a pool of processes for each type of task and it submits each task to its dedicated pool.

For both the *Determine Categories Stage* and *Sort Categories Stage* the application creates 3 pools of processes, each dedicated to one type of task. The number of processes that a pool can have is configurable, but the total number of processes used by the pools is always 24. This is the case because each computing node has 24 CPU cores, therefore it can handle at most 24 parallel tasks. It was observed that based on the number of processes assigned to each pool, the application yields different execution times. To have a good analysis we

first experimented with different numbers of processes each pool can have. The results are presented in the next section. Furthermore, this section presents our observations on the execution of the tasks, as well as on the execution of the entire application. These results are based on experiments that were ran using the best configuration in terms of number of processes per pool.

4.2.1 Process pools

The application implements both the *Determine Categories Stage* and *Sort Categories Stage*. At each stage, the application creates three different pools of processes, each being assigned to handle tasks of one type, reading tasks, processing tasks and writing tasks. Moreover, the application is able to configure the number of processes available for each pool based on the stage that it must execute. In this sense, it can have two different configurations, one for each stage.

To understand what is the best configuration in terms of the number of processes assigned to each type of pool we analyse the execution times of both the *Determine Categories Stage* and the *Sort Categories Stage*. We determine the best configuration as being the one that executes faster on the given stage.

Determine Categories Stage

Figure 31 presents the stage execution times during 10 runs of an experiment configured to use 8 processes for each type of task. Figure 32 presents the stage execution times during 10 runs of an experiment configured to use 4 processes for the reading tasks, 10 processes for the processing tasks and 10 processes for the writing tasks. Moreover, Figure 33 presents the stage execution times during 10 runs of an experiment configured to use 6 processes for the reading tasks, 10 processes for the processing tasks and 8 processes for the writing tasks. As it can be observed, in all of the cases, the stage finishes within a similar amount of time. However, it can be noticed that when using 6 processes for the reading tasks, 10 processes for the processing tasks and 8 processes for the writing tasks (Figure 33), the execution of the stage takes less time, with the majority of runs requiring between 75 and 78 seconds. In the other two cases, all the stage execution times take longer than 78 seconds, but less than 82.5 seconds.

To understand the cause of the different execution times further analysis is performed on the execution times of each task based on their type. The results revealed that for each type of task, the execution times are similar, regardless of the number of processes that were used for each pool. The reading tasks have the same variation in all the cases presented above, with the fastest execution time being of 0.5 seconds and the slowest execution time being of 4.5 seconds. Moreover, the curve of the graph is the same, the majority of tasks requiring around 2 seconds to complete (Figure 34). A similar behavior was observed for the processing tasks for which, regardless of the configuration, the tasks required between 2.8 and 7.5 seconds, with 60% of the tasks finishing within 3 seconds. Also, for all the cases the ECDF graphs present a similar curve for the execution times of these tasks (35. In the case of the writing tasks, they all have a similar execution time variation, regardless of the number of processes used for their pool (36). The writing tasks execution times varies from 0.3 seconds up to 18 seconds. However, 80% of the tasks finish

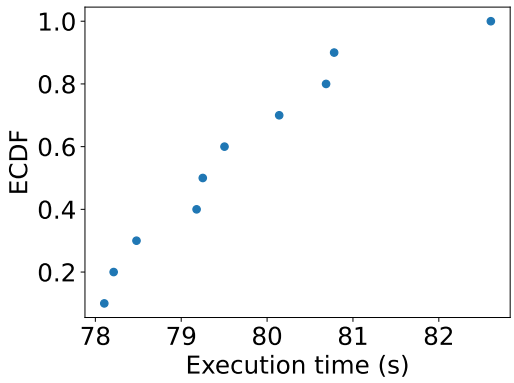

Figure 31: Execution times of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 8 processes for the reading tasks, 8 processes for the processing tasks and 8 processes for the writing tasks.

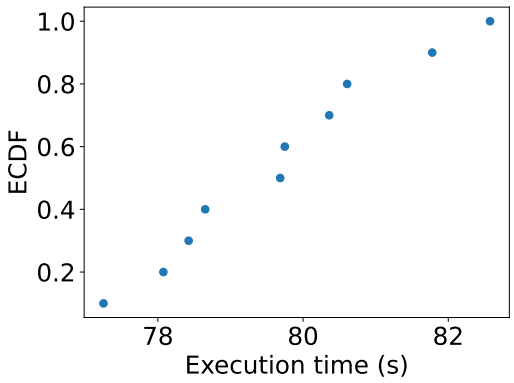

Figure 32: Execution times of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 4 processes for the reading tasks, 10 processes for the processing tasks and 10 processes for the writing tasks.

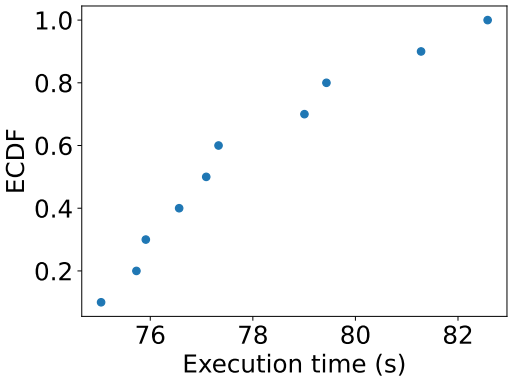

Figure 33: Execution times of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 6 processes for the reading tasks, 10 processes for the processing tasks and 8 processes for the writing tasks.

within 5 seconds. As the tasks of each type have similar execution times, regardless of the configuration, it can be concluded that in the case of 6 processes for the reading tasks pool, 10 processes for the processing tasks pool and 8 processes for the writing tasks pool, the processes of each pool are less likely to starve, meaning that better parallelization was achieved.

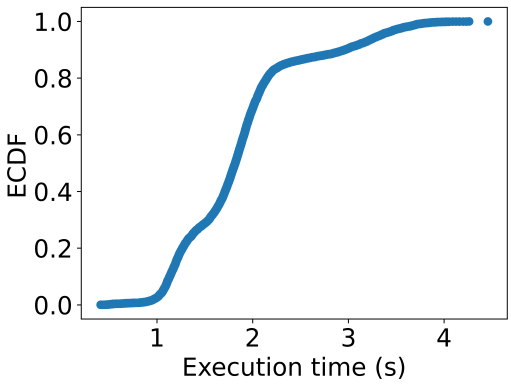

Figure 34: Execution times of the reading tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 6 processes for the reading tasks.

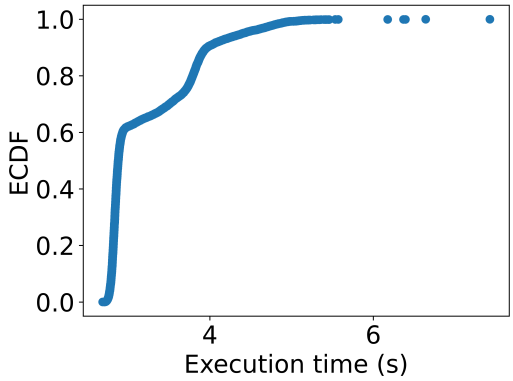

Figure 35: Execution times of the processing tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 6 processes for the processing tasks.

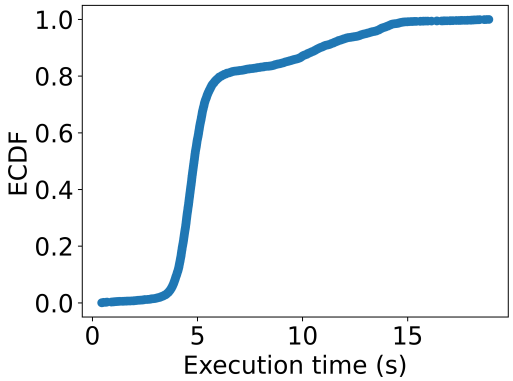

Figure 36: Execution times of the writing tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 6 processes for the processing tasks.

Sort Categories Stage

Figures 37, 38 and 39 present the execution time of the *Sort Categories Stage* during 10 runs of experiments that use different configurations in terms of the number of processes assigned for each of the three pools. Figure 37 presents the results of an experiment where there were 18 processes assigned to handle the reading tasks, 4 processes for handling the processing tasks and only 2 processes for handling the writing tasks. We chose a large number of processes for handling the reading processes because in the *Sort Categories Stage* the application is required to execute a large number of reading tasks, due to the multiple partitions that must be read in order to reassemble the categories to be sorted.

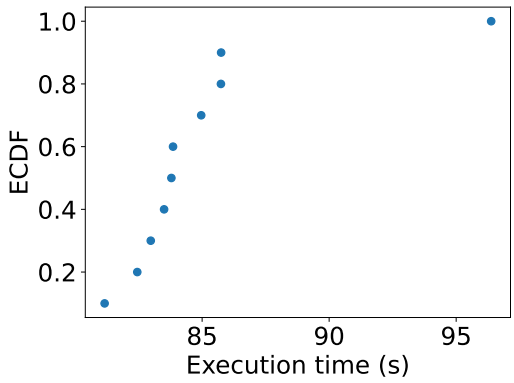

Figure 37: Execution times of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 18 processes for the reading tasks, 4 processes for the processing tasks and 2 processes for the writing tasks.

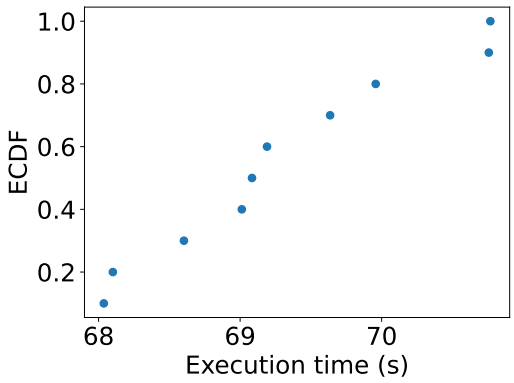

Figure 38: Execution times of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 14 processes for the reading tasks, 6 processes for the processing tasks and 4 processes for the writing tasks.

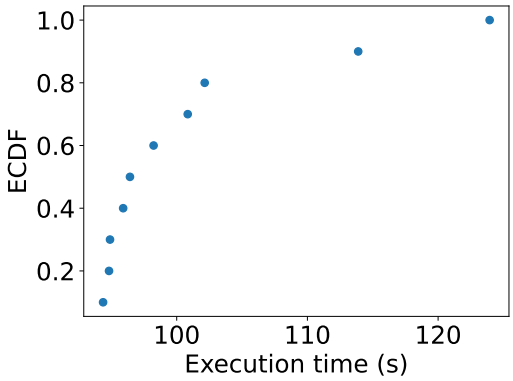

Figure 39: Execution times of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 10 processes for the reading tasks, 8 processes for the processing tasks and 6 processes for the writing tasks.

The total number of partitions that must be read by all the worker applications deployed on the computing nodes during the *Sort Categories Stage* for a configuration of 1000 data files and 256 categories is 256 000. However, it is important to note that only a fraction of the total number of partitions must be read by each worker application. Figure 38 presents the stage execution times during 10 runs of an experiment with a configuration of 14 processes for the reading tasks, 6 processes for the processing tasks and 4 processes for the writing tasks while Figure 39 uses a configuration of 10 processes for the reading tasks,

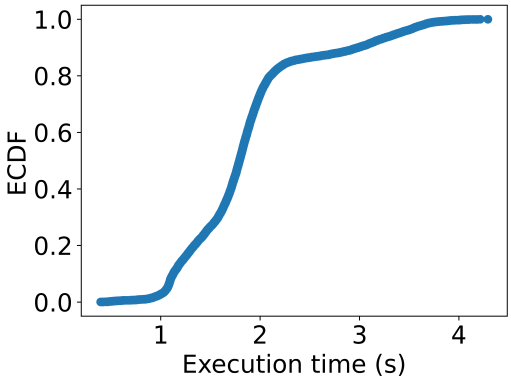

Figure 40: Execution times of reading tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each.

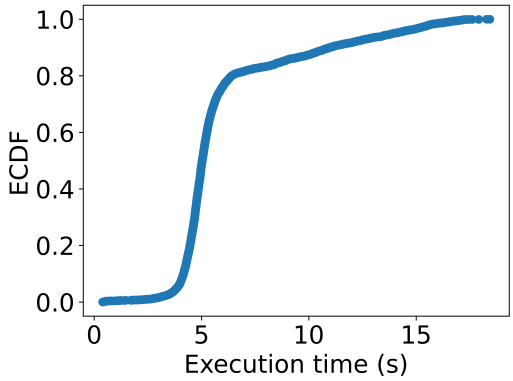

Figure 41: Execution times of writing tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each.

8 processes for the processing tasks and 6 processes for the writing tasks. As it can be observed, each configuration yields different results. However, it can be noticed that using 14 processes for the reading tasks, 6 processes for the processing tasks and 4 processes for the writing tasks achieves the fastest execution times. In this case, this configuration was chosen for further analysis.

This analysis is part of the observation **O-6** and contributes towards achieving **Experimental Goal 1**.

4.2.2 Large I/O Operations

The sorting application performs I/O operations for both retrieving the data that must be processed from the storage system, as well as writing the processed data back into the storage system (**FR 1**). During the *Determine Categories Stage* each of the reading and writing tasks are required to read and write data of size varies from 100MB to 1GB, depending on the experiment configuration.

Figures 40 and 41 present the execution times of the reading and writing tasks respectively, during 10 runs of an experiment with a configuration of 1000 data files of 100MB each. Moreover, Figures 42 and 43 present the execution times of the same tasks during 10 runs of an experiment, but with a configuration of 100 data files of 1GB each. It can be observed that in both experiments the execution time of the tasks is variable. Furthermore, the graphs depicting the execution times of the reading and writing tasks have similar curve, depending on the experiment configuration. This is a consequence of the number of concurrently running tasks sharing the available bandwidth. As for the *Determine Categories Stage* the configuration sets a number of 6 processes for handling the reading tasks and 8 processes for handling the writing tasks per worker application and there were 11 instances used in the experiments presented in the graphs, there can be at most 6 * 11 + 8 * 11 = 154 I/O operations running concurrently. However, this number is fluctuating,

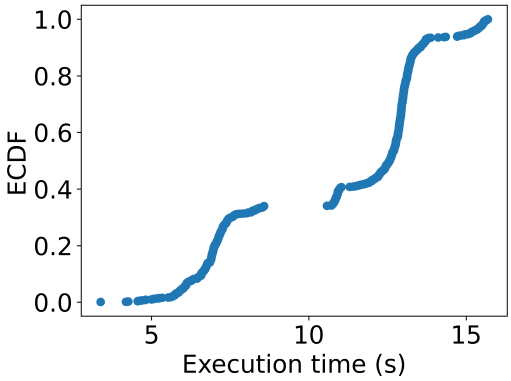

Figure 42: Execution times of reading tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 100 files of 1GB each.

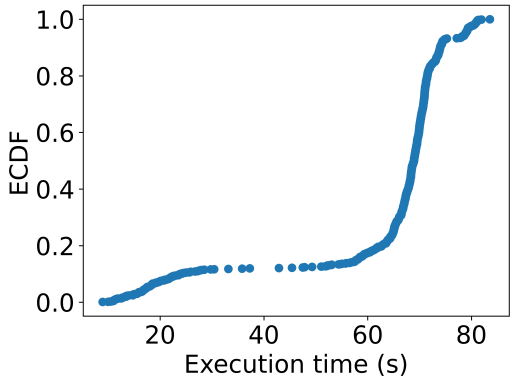

Figure 43: Execution times of writing tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 100 files of 1GB each.

depending on whether there are still data files to be read and whether there is any data that has been processed and that must be written to storage.

This analysis is part of the observation **O-4** and contributes towards achieving **Experimental Goal 1**.

4.2.3 Small I/O Operations

During the *Sort Categories Stage* the application is required to read all the partitions of all the categories defined in the *Determine Categories Stage* (**FR 1**). A partition of a category represents a small part of a data file generated at the previous stage. Moreover, each category has exactly one partition in each data file. In this sense, for an experiment configuration with 100 data files of 1GB each and 256 categories, each category has 100 partitions, each of size 1GB / 256 = 3.9 MB. However, for an experiment configuration with 1000 data files of 100MB each and 256 categories, each category is composed of 1000 partitions, each of size 100MB / 256 = 0.39 MB. Moreover, in the cases where the experiment configuration requires that 512 categories must be determined, their partitions have half the size of the ones where 256 categories must be determined.

Figures 44, 45, 46 and 47 present the execution times of the tasks required to read the partitions of 256 and 512 categories respectively. For the Figures 44, 45 the tasks were required to read the partitions from 1000 files of 100MB each, while for the Figures 46 and 47, the tasks were required to read the partitions from 100 files of 1GB each. It can be observed that in all 4 cases the majority of the tasks finish in less than 0.1 seconds. However, in the case where the data was split among 1000 files of 100MB, some outliers are present, with few tasks requiring up to 6 seconds in case of 256 categories and up to 3 seconds in case of 512 categories. In the case where the data was split among 100 files of 1GB, the slowest task required 0.45 seconds when 256 categories were defined and it required only 0.35 seconds when 512 categories were defined. It can be concluded that the

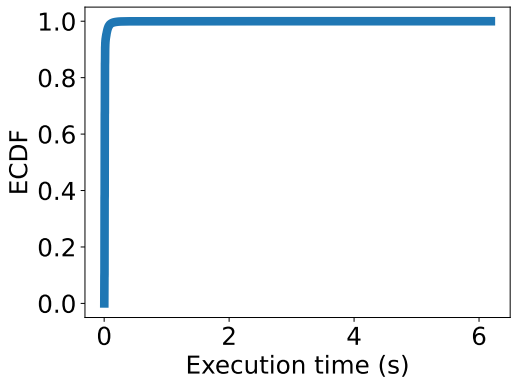

Figure 44: Execution times of reading tasks of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each and 256 categories.

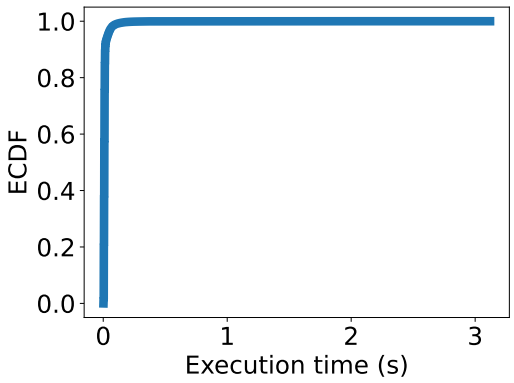

Figure 45: Execution times of reading tasks of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each and 512 categories.

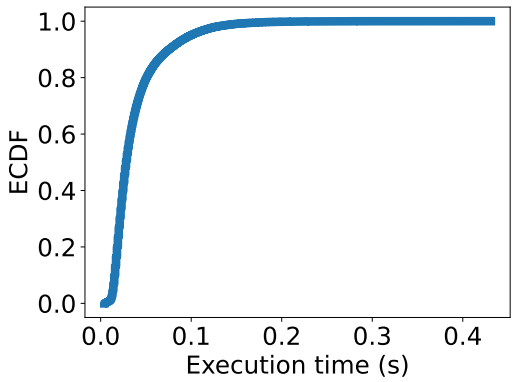

Figure 46: Execution times of reading tasks of the *Determine categories Stage* during 10 experiment runs of an experiment with a configuration with 100 files of 1GB each and 256 categories.

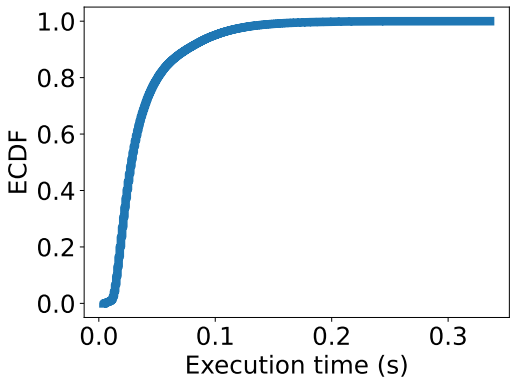

Figure 47: Execution times of writing tasks of the *Determine categories Stage* during 10 experiment runs of an experiment with a configuration with 100 files of 1GB each and 512 categories.

small I/O operations are more likely to have a constant execution. However, exceptions should be expected.

This analysis is part of the observation **O-4** and contributes towards achieving **Experimental Goal 1**.

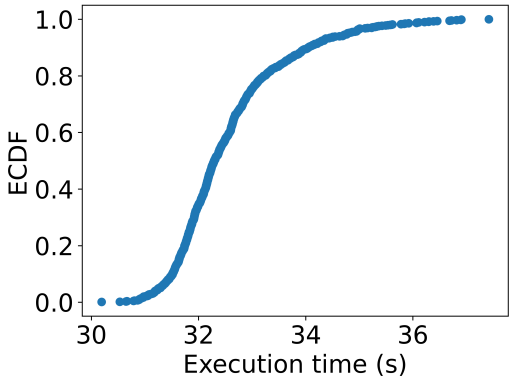

Figure 48: Execution times of sort and determine categories tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 100 files of 1GB each and 256 categories.

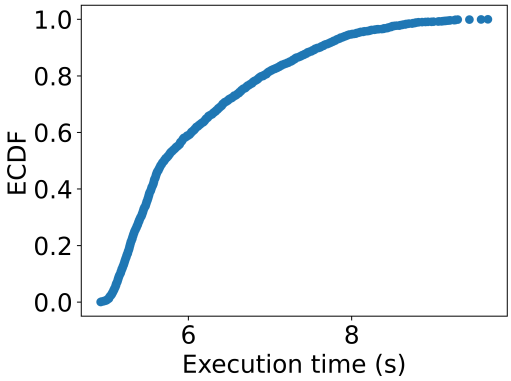

Figure 49: Execution times of sort categories tasks of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 100 files of 1GB each and 256 categories.

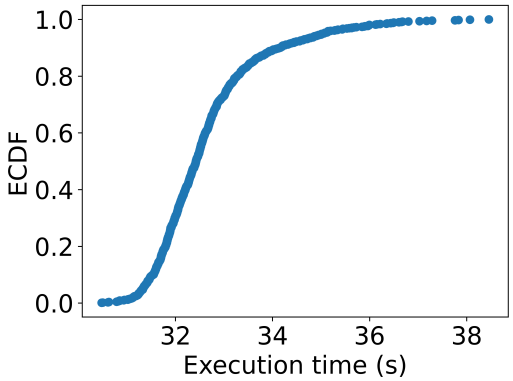

Figure 50: Execution times of sort and determine categories tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 100 files of 1GB each and 512 categories.

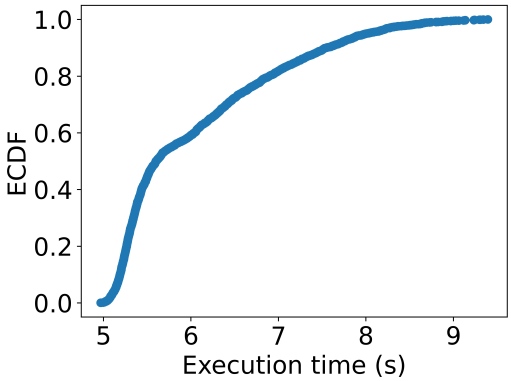

Figure 51: Execution times of sort categories tasks of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 100 files of 1GB each and 512 categories.

4.2.4 Processing Tasks

During the *Determine Categories Stage*, after the data entities are read from the files, they must be sorted by the first two characters of their keys and placed in the category they belong, before being written to the storage system. Moreover, during the *Sort Categories Stage*, the data entities belonging to each category must be sorted by their keys (**FR 2**).

Figures 48, 49, 50, 51, 52, 53, 54, 55 present the execution times of the sort and determine categories tasks and the sort categories tasks executing during experiments following different configurations. It can be observed that in all of the cases the tasks execution

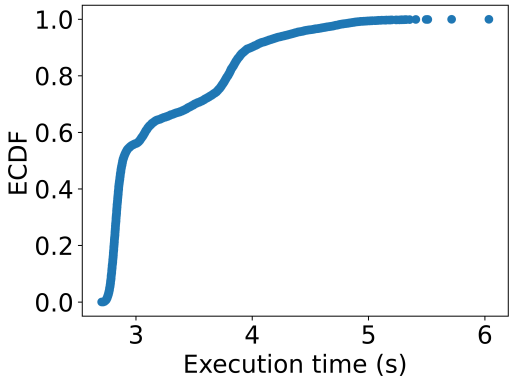

Figure 52: Execution times of sort and determine categories tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each and 256 categories.

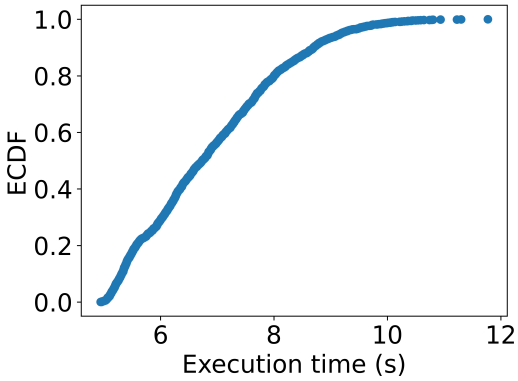

Figure 53: Execution times of sort categories tasks of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each and 256 categories.

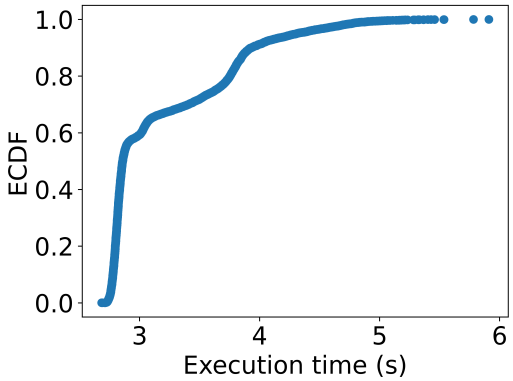

Figure 54: Execution times of sort and determine categories tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each and 512 categories.

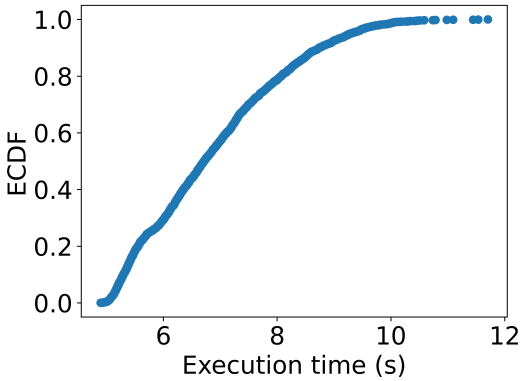

Figure 55: Execution times of sort categories tasks of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each and 512 categories.

times are varying. Moreover, the execution time of the tasks operating on the same file sizes vary between the same values, no matter what is the number of categories to be determined or sorted. Furthermore, the execution times have a normal distribution on the graphs.

This analysis is summarized in observation **O-5**.

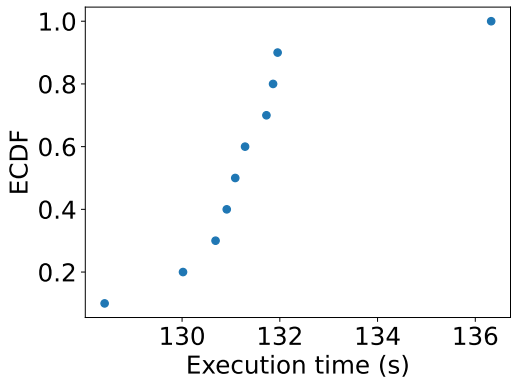

Figure 56: Execution times of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 11 computing nodes, 100 files of 1GB each and 512 categories.

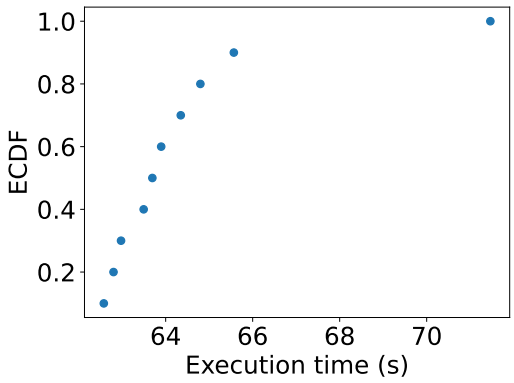

Figure 57: Execution times of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 11 computing nodes, 100 files of 1GB each and 512 categories.

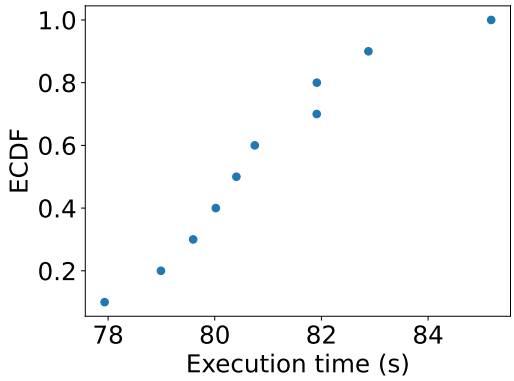

Figure 58: Execution times of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 11 computing nodes, 1000 files of 100MB each and 512 categories.

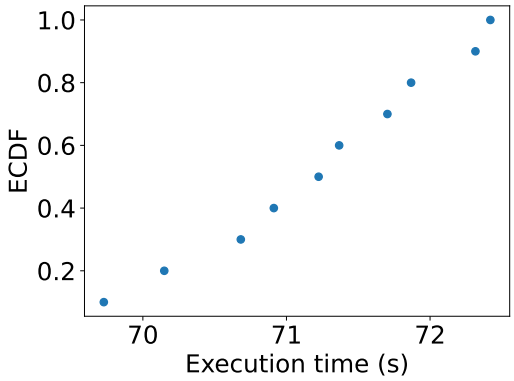

Figure 59: Execution times of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 11 computing nodes, 1000 files of 100MB each and 512 categories.

4.2.5 Application Execution Time

The application implements the sorting algorithm and executes in two stages, *Determine Categories Stage* and *Sort Categories Stage* (**FR 3**). At each stage the worker application must complete a number of jobs. For the case of this application, the jobs are split into multiple tasks. The application advances to the next stage or completes, only when all the tasks of the previous stage finished. In this sense, the execution time of the tasks of each stage directly impacts the execution time of the stage and in consequence, the execution time of the entire application.

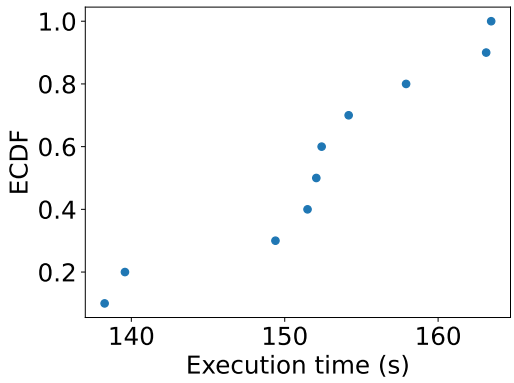

Figure 60: Execution times of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 22 computing nodes, 100 files of 1GB each and 512 categories.

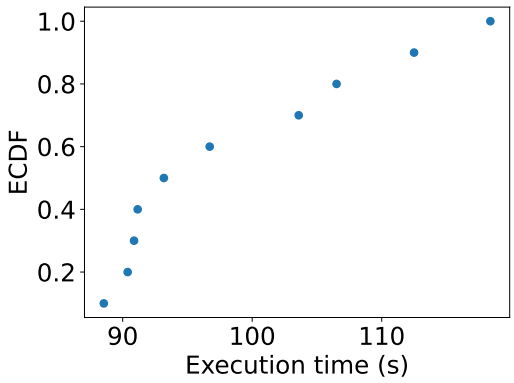

Figure 61: Execution times of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 22 computing nodes, 100 files of 1GB each and 512 categories.

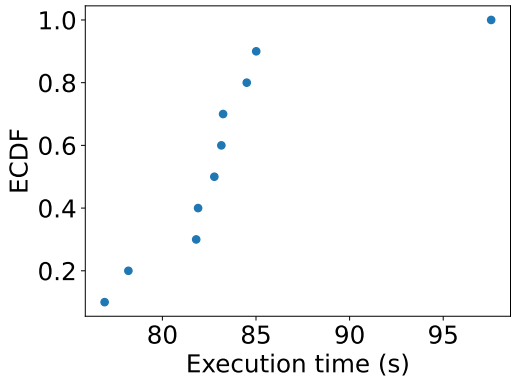

Figure 62: Execution times of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration with 22 computing nodes, 1000 files of 100MB each and 512 categories.

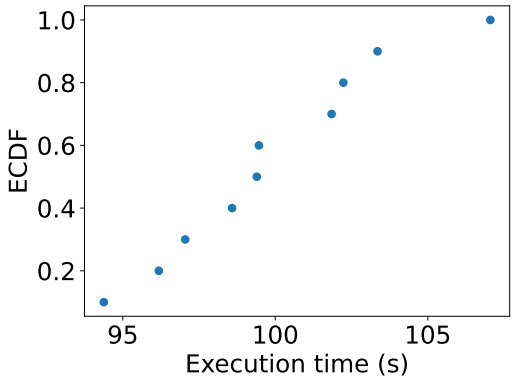

Figure 63: Execution times of the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 22 computing nodes, 1000 files of 100MB each and 512 categories.

Figures 56, 57, 58, 59 present the execution times of the two application stages when 11 computing nodes were used. It can be observed that the *Determine Categories Stage* completes faster when the data is split among 1000 files of 100MB each (Figure 58), rather than when it is split among 100 files of 1GB each (Figure 56). This is the case because 1 data file of 1GB is processed entirely by 1 process, while the same amount of data is processed by 10 processes in parallel when the data uses 100MB files. However, this has no major influence over the *Sort Categories Stage*. In both configurations the stage complete in approximately the same time, being just a little bit faster in the case where data is split among 100 files of 1GB each.

Figures 60, 61, 62, 63 present the execution times of each stage when using 22 computing nodes. It can be observed that it takes longer times for the stages to complete in this case, rather than when using only 11 processing nodes. This is an effect of the fact that more tasks run in parallel when using 22 processing nodes, thus more tasks share the bandwidth.

This analysis is summarized in observation **O-6** and contributes towards achieving **Experimental Goal 4**.

4.2.6 Observations

In this section we analysed the execution of the worker sorting application using task parallelization. Multiple instances of the worker sorting application were deployed on the DAS-6 cluster, each instance being hosted on one computing node. Moreover, the data storage was hosted on a different computing node and was made accessible through MinIO, a High Performance Object Storage compatible with Amazon S3 storage. A list of the main observations on the execution of different tasks during our experiments on the worker sorting application with task parallelization is presented below:

- **O-4. I/O Operations Tasks**

  In section 4.2.2 we analysed the execution of the large I/O operations, in Section 4.2.3 we analysed the execution of small I/O operations, while in Section 4.2.4 we analysed the execution of the processing tasks. We define large I/O operations by referring to reading and writing data objects of sizes varying from 100 MB and up to 1GB from/to storage. Similarly, we refer to small I/O operations by referring to reads and writes of data objects with sizes varying from 0.195 MB and up to 3.9 MB. Moreover, we refer to the sorting and category determination tasks as processing tasks.

  On one hand, we observed that for the large I/O operations the major drawback for the worker sorting application was made by the available bandwidth. This is because the more reading and writing tasks executing simultaneously, the less bandwidth each task had available. However, the application was designed such that we could configure the number of read and write tasks that can be executed in parallel. In this way, we ran experiments with different configurations in order to get the best results in terms of execution time for all tasks involved. Although we managed to improve the execution times of the tasks and consequently, of the application, variation was still observed.

  On the other hand, for the small I/O operations the execution time was, in general, constant. This is due to the fact that the small I/O operations do not require a lot of bandwidth. As a result almost all tasks finish instantly. However, we could observe that some tasks would require up to 6 seconds to complete.

- **O-5. Processing Tasks**

  We observed variation in the execution time of the processing tasks. In this case, the variation is around 8 seconds between the fastest and slowest processing task.

- **O-6. Application execution Time**

The worker sorting application with task parallelization executes in two stages. In order to complete, both stages must be finished. Moreover, for each stage to complete, all tasks generated for that stage must be successfully executed. As we previously observed, each type of tasks has a certain variability in its execution time. The execution time of each application stage and of the entire application is highly influenced by the variability of the tasks executed. Because we were able to configure the number of processes dedicated to each type of task, we managed to minimize the variability of the execution time of the tasks and to reduce the execution time of the stages and, in turn of the application.

4.3 Comparison Between the Execution of the Worker Sorting Applications - Job Parallelization vs. Task Parallelization

Question 1.3: What workload-parallelization technique shows better performance for the data-intensive sorting application in cluster environments?

Section 4.1 presents the analysis of the execution of the worker sorting application implementing job parallelization, while Section 4.2 presents the analysis of the execution of the worker sorting application implementing task parallelization. Multiple instances of the applications were deployed on the DAS-6 cluster, each instance being hosted on one computing node. Moreover, in both cases, the data was stored on a different node and was made accessible through MinIO.

Large I/O operations tasks

In both cases we observed that the large I/O operations have a high variation of around 8 seconds. We consider that this is influenced by the available bandwidth. The more reading and writing operations execute concurrently, the less bandwidth is available for each of them. However, in the case of task parallelization, because we were able to configure the number of concurrent connections, we managed to reduce the overall execution time of the tasks. As we see in Figure 6, for the worker sorting application with job parallelization, the reading tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 100 initial files of 1GB each varied between 13.5 and 20 seconds, while the same tasks in the case of the worker sorting application with task parallelization varied between 4 and 16 seconds (Figure 42). Similarly, the reading tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial files of 100MB each varied between 0 and 8 seconds (Figure 8), while the same tasks in the case of the worker sorting application with task parallelization varied between 0 and 4.5 seconds (Figure 40).
In the case of the writing tasks of the *Determine Categories Stage*, during 10 experiment runs of an experiment with a configuration of 100 initial files of 1GB each, the execution time of the tasks varied between 57 and 100 seconds (Figure 7), while the same tasks in the case of the worker sorting application with task parallelization varied between 10 and 85 seconds (Figure 43). Similarly, the writing tasks of the *Determine Categories Stage* during 10 experiment runs of an experiment with a configuration of 1000 initial files of 100MB each varied between 0 and 33 seconds (Figure 9), while the same tasks in the

case of the worker sorting application with task parallelization varied between 0 and 17.5 seconds (Figure 41).

Small I/O operations tasks

In both cases we observe that the execution of small I/O operations tasks happen almost instantly (Figures 10, 12, 44, 46). Since the data that is being read has very small sizes (between 0.195MB and 3.9 MB), the tasks do not require a lot of bandwidth to finish their work. In this sense, the execution time is not affected by the number of concurrent tasks that are running.

Processing tasks

We observe that the applications perform very similar in the case of the processing tasks. Figure 17 represents the sorting and determining categories processing time of the tasks of the worker sorting application with job parallelization, while Figure 52 represents the sorting and determining categories processing time of the tasks of the worker sorting application with task parallelization. Both graphs depict the execution times from experiments running the same configuration, namely 1000 files of 100 MB each with 256 categories to be determined. Although the graphs do not present the same distribution, the execution times in both cases vary between the same limits (2.5 - 5.5 seconds). The same observation can be made for Figures 19 and 53 where we observe the execution times of the sort categories tasks during the *Sort Categories Stage* during 10 experiment runs of an experiment with a configuration with 1000 files of 100MB each and 256 categories for both the worker sorting application with job parallelization and the worker sorting application with task parallelization.

Application execution time

Both the worker sorting application with job parallelization and the worker sorting application with task parallelization execute in two stages and they must be completed in order for the application to finish. Moreover, each stage is considered complete when all workload assigned for it is processed. In subsections 4.1.4 and 4.2.5 we observed the execution times of the stages of the worker sorting applications. As we could see, in both cases the execution time varied. However, by doing task parallelization it was possible not only to control the number of concurrent I/O operations and maximizing the throughput, but also to overlap the execution of multiple types of tasks. In this way, we observed that using task parallelization we obtained an improvement up to 40% in the application execution time.

5 Performance Analysis of the Serverless Applications Running on AWS Resources and Comparison with the Worker-Sorting Applications Running on DAS-6 Resources

This chapter presents the performance analysis of the two serverless applications, the sorting application and the TPC-DS application, running on the AWS cloud. After we perform the analysis, we compare the results with the ones from the performance analysis of the worker-sorting applications that ran on the DAS-6 cluster.

5.1 Performance Analysis of the Serverless Applications Running on AWS Resources

Question 2: What is the performance impact of sorting big data in serverless environments?

This section presents the analysis of the execution of the serverless applications described in Section 2. The applications executed following the experiment design and configurations discussed in the Section 3. During the experiment runs, the applications logged data regarding the execution times of the different processes involved. The experiment results are structured based on the type of applications developed. Furthermore, for each application type we analyse the results from the task level as well as from the application level.

5.1.1 Cold Start

The first observation identified while running the experiments on AWS cloud is the cold start. This is present in every experiment run, but it is the most noticeable during the first one. As discussed in the *Background section*, each cloud function lifecycle starts with the **Init stage**. At this point an execution environment is created, necessary resources are gathered, external dependencies are brought in and the function code-base is deployed. This is required in order to ensure good conditions for executing the cloud functions. Moreover, the execution environment is not immediately disbanded after the function completes. The cloud environment monitors the execution the AWS Lambda environments already created and disbands the ones for which inactivity is detected over a certain time period. The user does not have any control over the inactivity timeout and it has been proven that it is not static. This results in execution environments to be dissolved at different time intervals.

AWS Lambda automatically reports the initialization times of every function that runs on the AWS cloud. This data was used to create following ECDF graphs. Figure 64 and Figure 65 represent the initialization times of the cloud functions that were deployed during the first run of a selected experiment and executed the *Determine Categories Stage* of the sorting application. As it can be observed, these cloud functions required initialization time. While 65% of the cloud functions managed to get initialized within 0.6 seconds, 35% of them required longer times that go up to 1.6 seconds. Figure 66 depicts the initialization times of the cloud functions that were deployed during the other 9 runs of

the same experiment and executed the same stage of the sorting application. It can be noticed that almost none of cloud functions required any initialization times. However, there are few cloud functions that still required to run the *initialization stage*. It can be deduced that cloud function execution environments reached their timeouts in-between the experiment runs.
It was observed that the AWS Lambda cloud functions executing at the first stage of an experiment run require initialization time in order for the execution environment to get created. However, for cloud functions executing at a later application stage, as well as in the other experiment runs there is no need for initialization time. These AWS Lambda cloud functions can reuse the execution environments that were created for the functions executing at the first stage of the application during the first experiment run. It was noticed that when 1000 cloud functions are requested, sometimes a few of them require initialization time, meaning that some of the previously created execution environments got disbanded.

The cold start directly impacts the execution time of the application as some cloud functions require time to create the execution environments before being able to execute its job. Moreover, as execution environments get disbanded and having no control over the timeout required before terminating the environment, it is important to note that even if one cloud function requires a cold start, the initialization time needed for this function to run, is inflicted by the entire application. The same happens for applications where a later stage requires to deploy more functions than any previous stage. The extra functions have a cold start, thus initialization time is created.

The cold start affects the application execution and even more, its variation. Moreover, the fact that the user cannot control the timeout for terminating the execution environment affects the application execution even further. In this sense, it can be the case that even though a batch of functions just finished their execution, some of the execution environment could be instantly terminated, the new functions requiring initialization time again.

On one hand, it can be observed that cold starts mostly happen for the functions that execute the first stage of any of serverless application (Figures 64 and 65. During this time an execution environment needs to be created for each cloud function that deployed, therefore each cloud function registers some initialization time.

On the other hand, as it can be noticed in Figure 67, representing the ECDF graph of all functions deployed for the *Sort Categories Stage* of the sorting application where only 512 functions were started, the cloud functions executing this stage does not require any initialization time. It is important to mention that for the experiment about which Figure 67 presents data, there were 1000 cloud functions running the *Determine Categories Stage*. However, Figure 68 depicts the initialization times of cloud functions executing the *Sort Categories Stage* of the sorting application, in an experiment where there are 512 cloud functions executing this stage and only 500 cloud functions for the *Determine Categories Stage*. As it can be observed in Figure 68, the extra 12 cloud functions required for the *Sort Categories Stage* inferred some initialization time.

Comparing data of different experiments (Figures 64 and 65), we notice that functions initialization times are never the same throughout the experiments. This is not caused

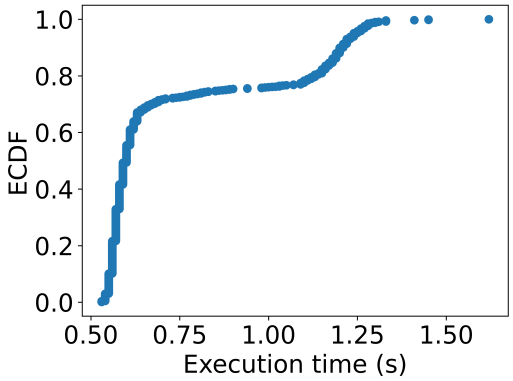

Figure 64: Initialization time for 1000 cloud functions executing the *Determine Categories stage* of the sorting application during the first experiment run of an experiment with a configuration of 1000 initial data files of 100MB each.

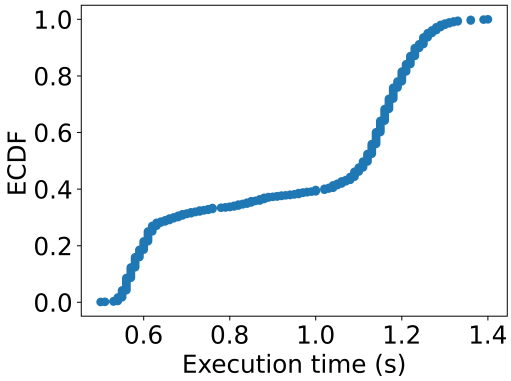

Figure 65: Initialization time for 1000 cloud functions executing the *Determine Categories stage* of the sorting application during the first experiment run of an experiment with a configuration of 1000 files of 500MB each.

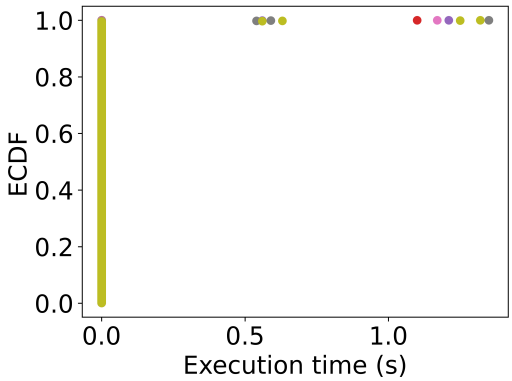

Figure 66: Initialization time for cloud functions executing the *Determine Categories stage* of the sorting application during the last 9 experiment runs of an experiment with a configuration of 1000 files of 500MB each.

by the functions code base, as all functions within one application execute the same code and require the same dependencies. In this case, the variation of the initialization time is likely to be influenced by the cloud platform.

It can be concluded that cold starts mostly affects the cloud functions executing the first stage of each of the serverless applications during the first experiment run and does not generally pose a problem for the cloud functions that execute the first stage during the other runs or that execute at later stages in the application flow during any experiment

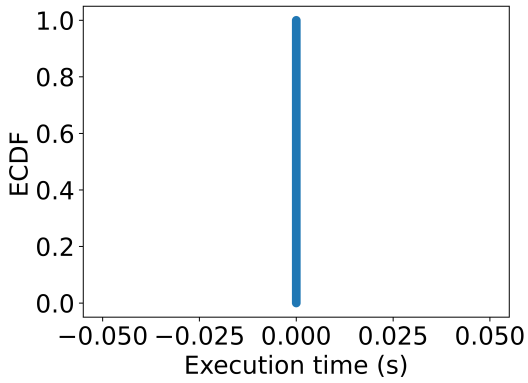

Figure 67: Initialization time for the cloud functions executing the *Sort Categories Stage* of the sorting application for all 10 runs of an experiment with a configuration where 512 categories to be sorted during each run.

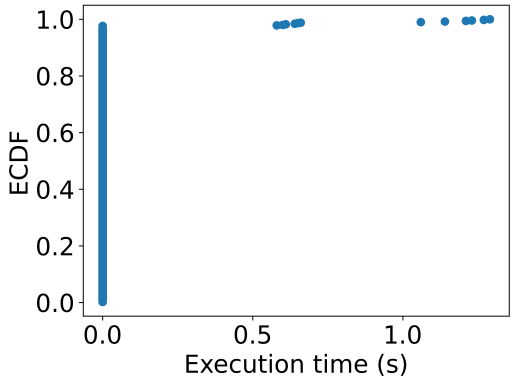

Figure 68: Initialization time for the cloud functions executing the *Sort Categories Stage* of the sorting application for the first run of an experiment with a configuration with 500 initial data files of 200MB each and 512 categories.

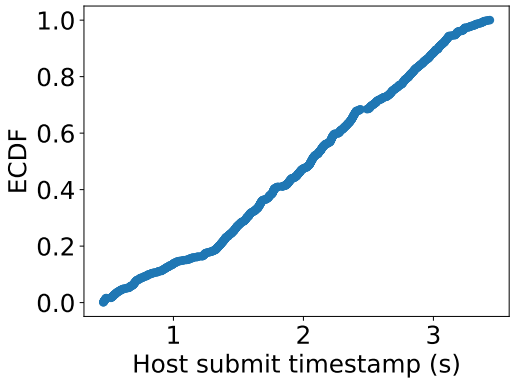

Figure 69: Host submit events 1 worker process.

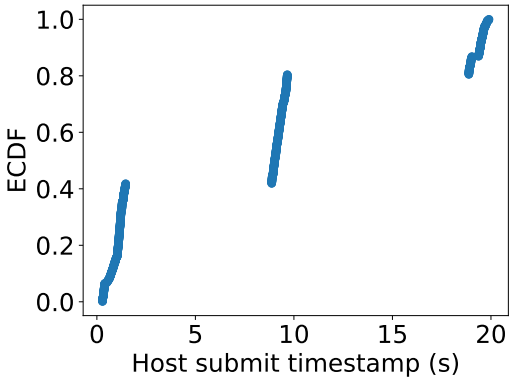

Figure 70: Host submit events 100 worker processes.

run. Other initialization times are reported for the cloud functions that execute at a later stage in the application flow when more instances are required for that stage than the highest number of functions required for any of the previous stage (Figure 68).

This analysis is summarized in the observation **O-7** and contributes towards achieving **Experimental Goal 3**.

5.1.2 Function Invocation Gap

The analysis of the data generated during experiments revealed limitations of the Lithops library. It can be observed that Lithops does not submit event data to start all required

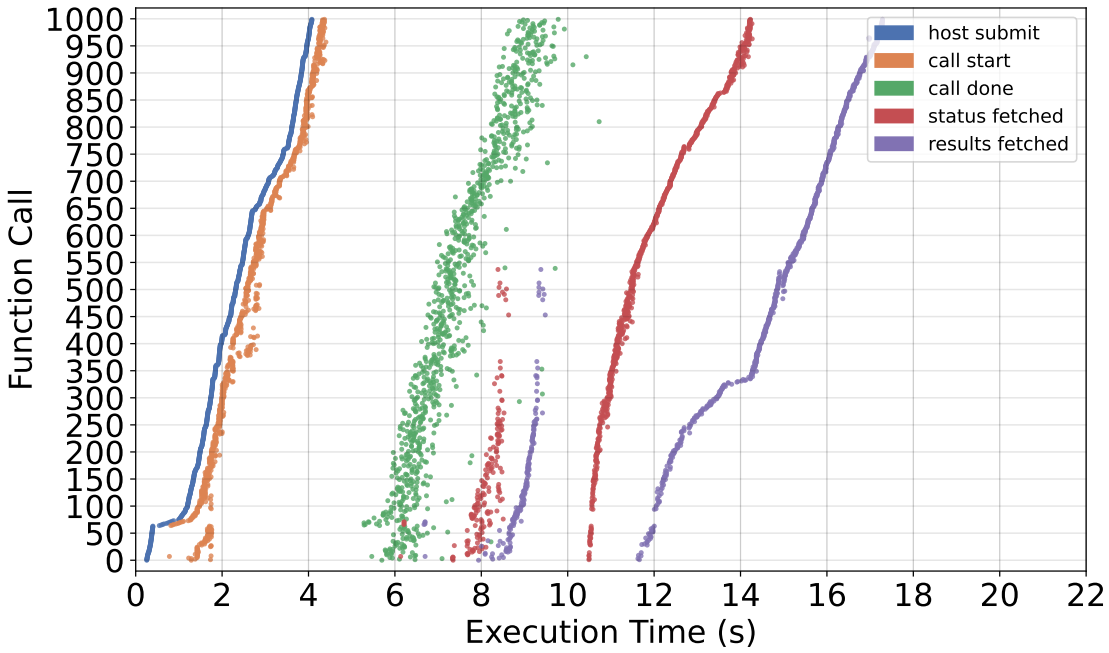

Figure 71: Timeline of functions events during one experiment with 1 process per worker.

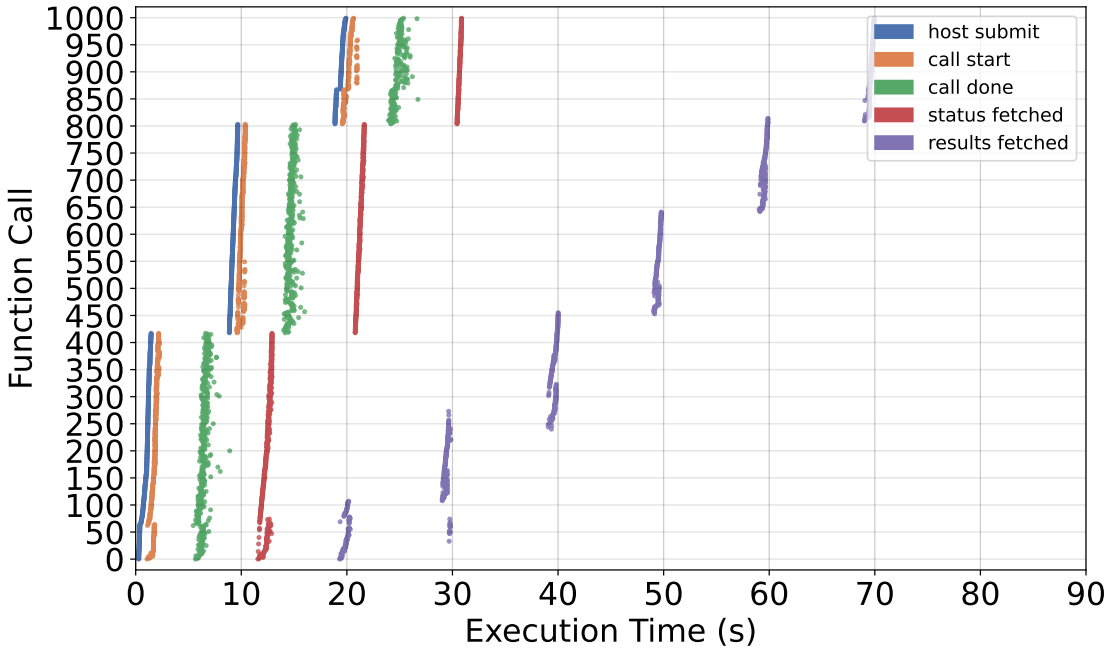

Figure 72: Timeline of functions events during one experiment with 100 processes per worker.

functions at the same time, but it does it rather sequential. This is the case when using the default Lithops configuration that sets only one process responsible for starting up Lambda cloud functions. As a result, Figure 69 presents the fact that Lithops manages to trigger 1000 functions within 3.5 seconds.

Although Lithops gives the possibility of increasing the number of Lithops processes within a worker in order to increase parallelization of function activation, it is not possible to activate very large numbers of cloud functions simultaneously. Figure 70 presents the host submit events for an experiment that requires 1000 functions while Lithops is configured to use 100 processes per worker. In this case, Lambda functions are triggered in bursts. However, we can see that each batch of cloud functions are being triggered at an approximately 10 seconds difference between each other, ending up with the last batch of functions being deployed approximately 20 seconds later than the first function of the experiment. This means that depending on the workload, the last batch of functions can be deployed even later than the first batch of functions finish their execution. This increases the overall time of the experiment runs, as the applications must wait for all cloud functions executing a certain application stage before being able to move to a next stage.

Moreover, increasing the number of Lithops processes also affects the results fetching phase. Figures 71 and 72 represent the timeline of events of the cloud functions executing the *Determine Categories stage* stage of the sorting application during all 10 runs of an experiment while using only 1 process and 100 processes per Lithops worker respectively. As it can be observed, while using only 1 process per Lithops worker, the results are fetched sequentially. However, when using 100 processes per Lithops worker the results are fetched in bursts, but it takes much longer to collect the results of all cloud functions that ran.

In conclusion, the experiment finishes a lot faster when having only 1 process per worker then when having 100 processes. We experimented with a higher number of processes as well, but it yielded the same results as when using 1 process per worker. This means that we reached the maximum number of processes that Lithops can use for a worker. Moreover, the time it takes for the Lithops library to activate all the cloud functions adds up to the total execution time of the application. Furthermore, this time is inflicted by the application at every stage because for every stage the application requires Lithops to activate a number of functions.

This analysis is part of the observation **O-10**.

5.1.3 Large I/O Operations

Each of the applications used in the experiments execute reading and writing tasks on the data files from and to the AWS S3 storage (**FR 1**). The experiments results revealed that the execution times of these tasks are generally constant. However, it was observed that in every experiment there are a number of tasks that require more time to complete.

Figures 73 and 74 represent the times necessary for the cloud functions executing during the *Determine Categories Stage* of the sorting application to read and write the data files from and to the S3 storage environment. The figures present the ECDF graphs of these execution times within an experiment of 10 runs using a configuration of 1000 initial files

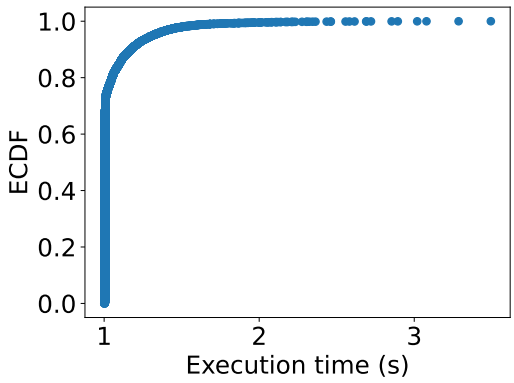

Figure 73: Execution times of reading tasks of the *Determine Categories Stage* of the sorting application during 10 experiment runs with a configuration of 1000 files of 100 MB each.

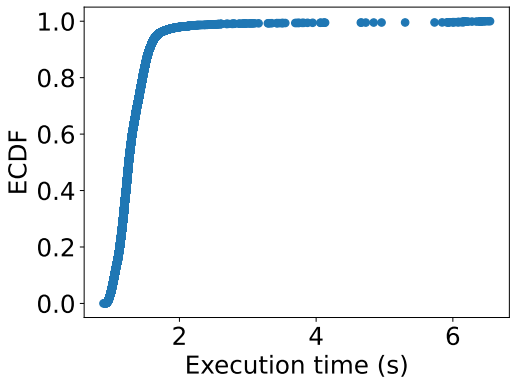

Figure 74: Execution times of writing tasks of the *Determine Categories Stage* of the sorting application during 10 experiment runs with a configuration of 1000 files of 100 MB each.

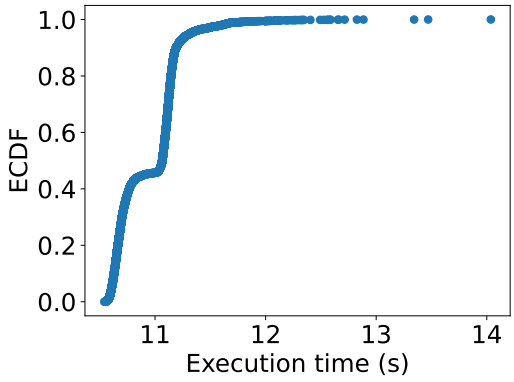

Figure 75: Execution times of reading tasks of the *Determine Categories Stage* of the sorting application during 10 experiment runs with a configuration of 1000 files of 1GB each.

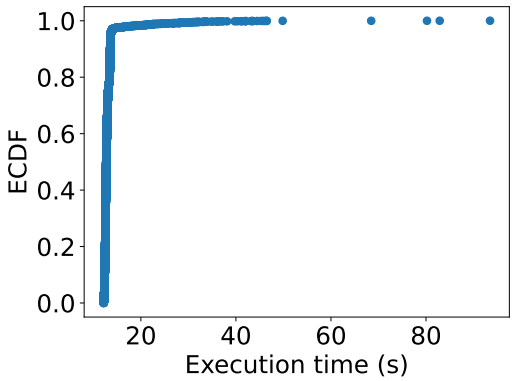

Figure 76: Execution times of writing tasks of the *Determine Categories Stage* of the sorting application during 10 experiment runs with a configuration of 1000 files of 1GB each.

of 100 MB each for each. In total, each graph presents the execution times of 10 000 tasks. For this experiment we used 1 process per Lithops worker meaning that the Lambda cloud functions were activated sequentially, in a timespan of 4 seconds. Because of this, not all reading tasks were concurrently active. On average, there were 500 active tasks at any give moment.

Looking at the execution times of the reading tasks, it can be observed that approximately 75% (73) of them had a constant completion time of 1 second. On the other hand, the rest 25% of the tasks exceeded this value with some of them requiring up to 3.5 seconds

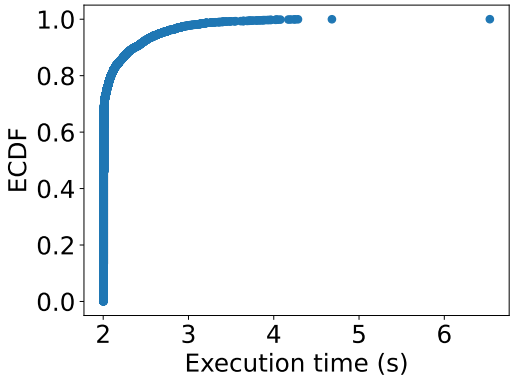

Figure 77: Execution times of reading tasks of the *Determine Categories Stage* of the sorting application during 10 experiment runs with a configuration of 500 files of 200MB each.

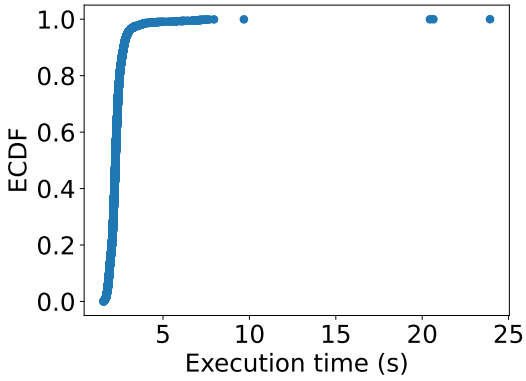

Figure 78: Execution times of writing tasks of the *Determine Categories Stage* of the sorting application during 10 experiment runs with a configuration of 500 files of 200MB each.

to finish.

Similar observations can be made about the execution times of the writing tasks. In this case, more than 90% of the functions take between 1 and 2 seconds to finish, while only a small percentage exceed this timespan. However, it can be noticed that for some tasks the necessary time for writing the data file can reach up to 6.5 seconds. Analysing the data showed that there is an average of 20 writing tasks exceeding 2 seconds to complete.

The same analysis was also performed based on experiments using other configurations. Figures 75 and 76 depict the execution times of the reading and writing tasks executed for an experiment of 10 runs using a configuration of 100 data files of 1GB each. In this case, little variation can be observed, the majority of the reading tasks executing within 10 - 11,5 seconds. Only a very small percentage of the tasks require longer execution time, but even in those cases, they do not exceed 14 seconds. On the other hand, looking at the execution times of the writing tasks, it can be observed that most of the tasks finish in 12 seconds, but for the rest it can take anywhere between 12 and 90 seconds. Figures 77 and 78 represent the execution times of reading and writing tasks during an experiment with 10 runs using a configuration of 500 data files of 200MB each. Also in this case it can be observed that the majority of the tasks require the same amount of time to finish. However, outliers are present in this case as well.

We try to understand the reason for the cloud functions to require longer times for reading and writing their files. First of all, we observe that there is never the same cloud function that executes the slower reading and writing tasks. Secondly, we identify how many functions were concurrently performing the same task when the slowest reading and writing started. Moreover, we also identify the highest number of functions performing the same task and we look at the completion times of the last function that joined the batch. However, no correlation could be done. This is because we could see that the slower task was performed while different number of functions were already running. Moreover, while

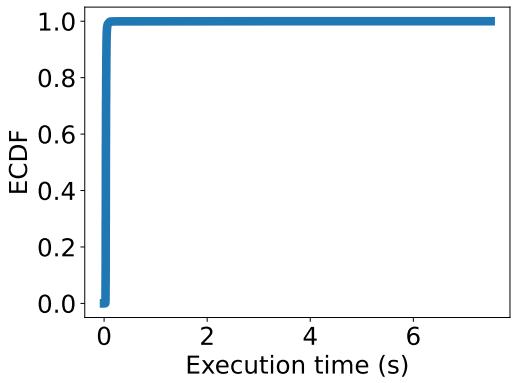

Figure 79: Execution times of the read partition tasks for each of the 512 categories, within 10 runs of an experiment

the most number of functions that were concurrently performing the same task, the last cloud functions that joined those groups did not take longer than the majority to finish.

This analysis showed that there is no configuration that can guarantee that all reading and writing tasks will complete in the same timespan considering files of the same size. Moreover, we could not find that something within our application and setup could be the cause for some cloud functions to take longer to finish their reading and writing tasks.

This analysis is part of the observation **O-8** and contributes towards achieving **Experimental Goal 1**.

5.1.4 Small I/O Operations

Our applications perform both small and large I/O operations (**FR 1**). During the *Determine Categories Stage*, the sorting application reads and writes the data from and to files of minimum 100 MB and maximum 1 GB. During the *Sort Categories Stage*, the sorting application reads multiple partitions of each category, resulting in many, small I/O operations. A partition of a category consists of a small part of each file generated during the *Determine Categories Stage*. As each cloud function is given one category to sort and each category has a partition in each file uploaded during the first application stage, depending on the configuration of the experiment, each function must read a number of partitions ranging from 100 to 1000 in order to reassemble a category. The application implements a mechanism which only allows reading the data of one partition, without reading the entire file. In this sense, for a configuration of 1000 initial files of 100 MB that are being split in 512 categories, during the *Sort Categories Stage*, we require 512 cloud functions to read 1000 partitions each, each partition being of size 195.3125 KB. In total, each cloud function will have read 195.3125 MB.

In Section 5.1.3, we analysed large I/O operations and it can be noticed that most of the cloud functions complete within the same time frame. However, there are always a few cloud functions that take considerably longer than the majority. For a better understanding, figure 79 depicts the reading time achieved by each function for each partition of

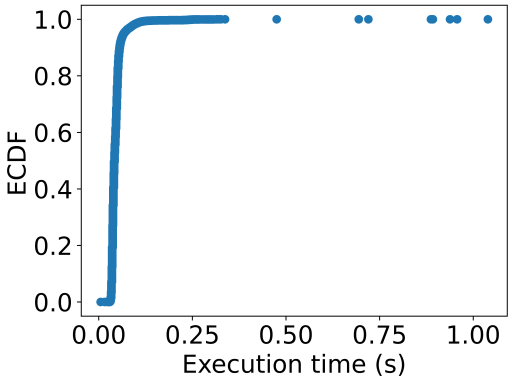

Figure 80: Execution times for every reading task of each file executed in the fourth stage of the first TPC-DS script, during 10 experiment runs.

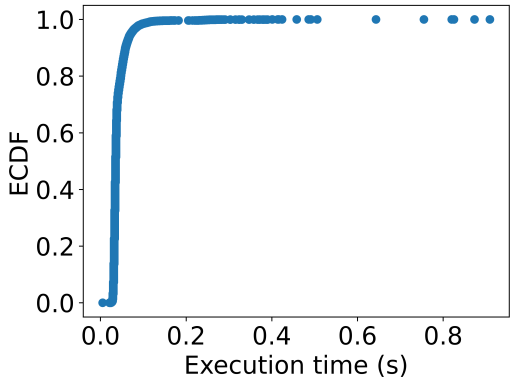

Figure 81: Execution times for every writing task of each file executed in the third stage of the first TPC-DS script, during 10 experiment runs.

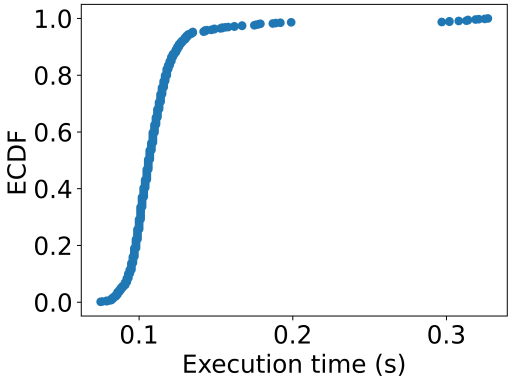

Figure 82: Execution times for every reading task responsible for the first partition of each file executed in the fourth stage of the first TPC-DS script, during 10 experiment runs.

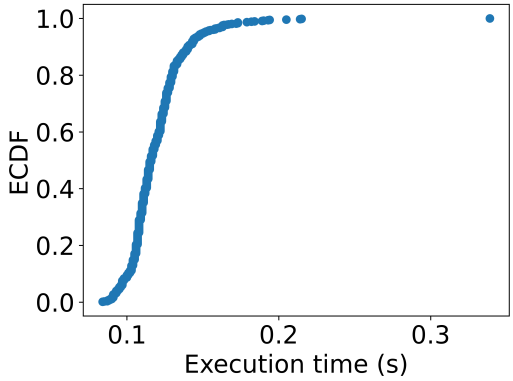

Figure 83: Execution times for every writing task responsible for the first partition of each file executed in the third stage of the first TPC-DS script, during 10 experiment runs.

195.3125 KB, during 10 runs of an experiment. In total, there were 5 120 000 reading tasks performed - 512 categories, multiplied by 1000 files and 10 runs. As it can be observed, the majority of functions have a constant reading time. However, again, some outliers are present. The data was analysed and it can be noticed that from 5 120 000 downloads, there are only 687 different download times, from which only 298 download times took longer than 0.5 seconds, 67 took longer than 1 second and 231 took between 0.5 and 1 second.

The experiments performed on the TPC-DS scripts provided the possibility of further analysing the performance of the small I/O operations. During certain stages of some of the scripts, reading and writing tasks execute on small files. For example, at the third

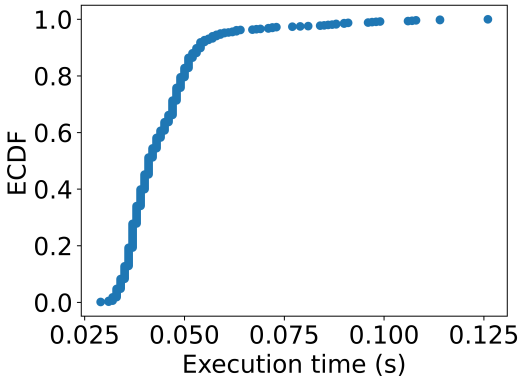

Figure 84: Execution times for every reading task responsible for the 62nd partition of each file executed in the fourth stage of the first TPC-DS script, during 10 experiment runs.

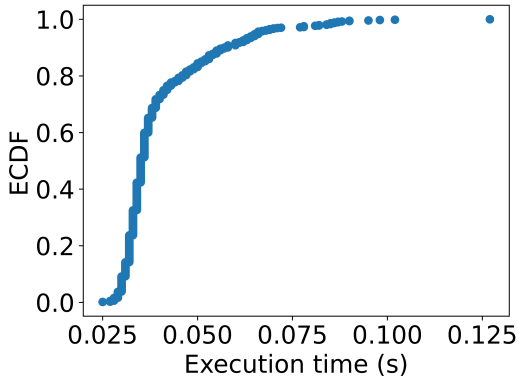

Figure 85: Execution times for every writing task responsible for the 62nd partition of each file executed in the third stage of the first TPC-DS script, during 10 experiment runs.

stage of the first **TPC-DS** script, the data resulted from the first and second stages is merged and the result is written to the storage system. Moreover, at stage 4, it is required that the data written during stage 3 is read. The script was configured such that the merged result is split into multiple, very small files. Consequently, during the third stage, 100 functions are started and each function writes 100 files. During stage 4, 100 functions are started and each function reads 100 files. Each file has a size varying from a few hundred bytes to a maximum of 1.5 KB.

Figure 81 represents the execution times of the reading tasks within all functions, responsible for all partitions, executed during 10 runs of an experiment, during stage 4. Figure 80 represents the execution times of the writing tasks within all functions, responsible for all partitions, executed during 10 runs of an experiment, during stage 3. It can be observed that the majority of the reading and writing tasks complete almost instantly. However, although the files are very small, some reading and writing tasks can take up to one second to finish.

Besides the exceptional cases where reading and writing tasks take longer to finish, the experiments data revealed that it generally takes longer for the first task to execute, rather than for the rest of them. Figures 82 and 83 depict the reading and writing times of the first partition, during an experiment. It can be observed that most of the functions take more than 0.1 seconds to finish. However, figures 84 and 85 show the reading and writing tasks execution times of the 62nd partition that was done by each function during an experiment. For this case, it can be noted that more than 80% of the functions finish their task in less than 0.06 seconds. This is the case for any other partition that is either read or write to the AWS S3 storage system.

This analysis is part of the observation **O-8** and contributes towards achieving **Experimental Goal 1**.

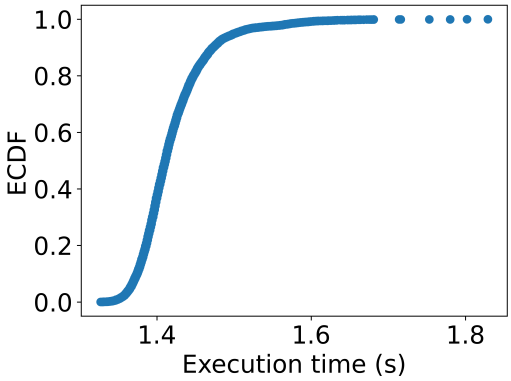

Figure 86: Execution times of the determine categories task of the *Determine Categories Stage*, during 10 runs of an experiment with a configuration of 1000 files of 100MB each.

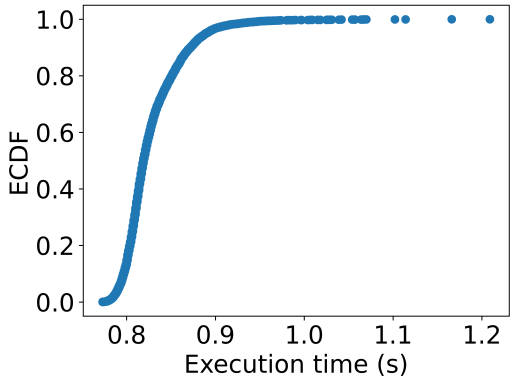

Figure 87: Execution times of the sorting categories task of the *Determine Categories Stage*, during 10 runs of an experiment with a configuration of 1000 files of 100MB each.

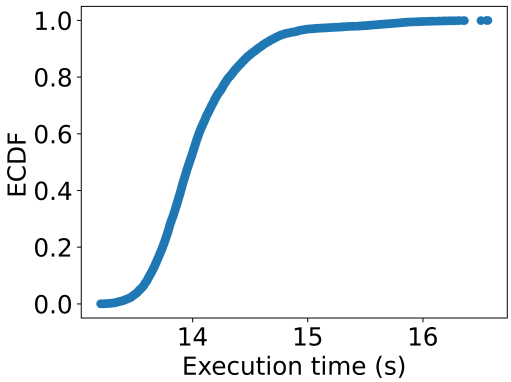

Figure 88: Execution times of the determine categories task of the *Determine Categories Stage*, during 10 runs of an experiment with a configuration of 1000 files of 1GB each.

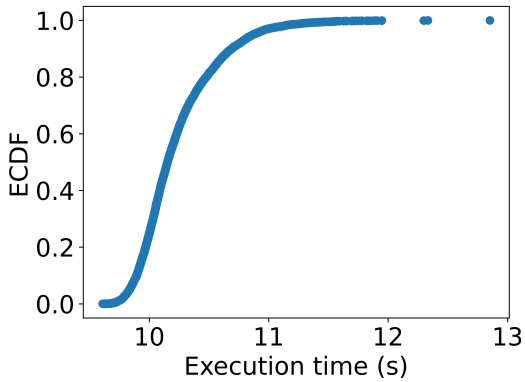

Figure 89: Execution times of the sorting categories task of the *Determine Categories Stage*, during 10 runs of an experiment with a configuration of 1000 files of 1GB each.

5.1.5 Processing Tasks

During the experiments, the execution time of the processing tasks were also monitored. In the case of the sorting application, the processing tasks are responsible for determining categories and sort the data entities, while in the case of the TPC-DS application, the processing tasks are responsible for resolving the queries (**FR 2**).

Figures 86 and 87 depict the necessary time for each task to determine the necessary categories and sort the data entities, during the *Determine Categories Stage* of the sorting application. Each graph depicts the times of 10 000 tasks, that were executed for an experiment of 10 runs, with the configuration of 1000 initial files, each file having the size

of 100 MB. Although the computational work is equal for the two types of tasks, time variability can be observed in both cases. Moreover, it is noticeable that there are no two tasks with the exact same completion time. Similar results can be observed for all processing tasks from any experiment configuration of any application. The only difference that can be seen is in variability, but that is always bound to the size of the data that needs to be processed. For example, figures 88 and 89 depict the necessary time for each type of task to sort, respectively determine the categories, during the *Determine Categories Stage* of the sorting application. In this case, each graph depicts the execution times of 10 000 functions that ran for an experiment of 10 runs with 1000 initial files, but with each file having 1 GB in size. It can be observed that the completion times follow a similar pattern as in the previously analysed case, where for both types of tasks there is a variation, in this case of approximately 4 seconds. Moreover,there are no two tasks with the same completion time, although all functions require the same computational effort.

This analysis is summarized in the observation **O-9** and contributes towards achieving **Experimental Goal 2**.

5.1.6 Application Execution Time

Both the serverless sorting application and the serverless TPC-DS application execute in multiple stages. At each stage, multiple work jobs are created, each job being executed by one AWS Lambda cloud function. Moreover, each job is composed of multiple tasks generally referred as reading tasks, processing tasks and writing tasks. All applications, in order to successfully execute a stage, require that all jobs belonging to that stage must be completed (**FR 3**).

The execution time of a stage depends on the execution times of all its jobs. In turn, the execution time of a job depends on the sum of the execution times of its tasks. It was previously observed that the majority of tasks finish within the same timespan. However, it was noticed that at each stage, there is a small percentage of tasks that take longer to complete. The execution times of these tasks have a direct impact on the execution time of the entire stage to which the tasks belong and consequently to the execution time of the application.

Previously it was observed that all tasks belonging to a stage show some variation in the execution time. Moreover, it was noticed that the major drawback for the execution time of a stage is created by the execution times of reading and writing tasks. While for processing tasks, such as sorting in the case of the sorting application, and query resolving in the case of the TPC-DS application, there is only a variation of a couple of seconds no matter what is the size of the data to be processed, for the reading and writing tasks variation of the execution time can reach tens of seconds.

The execution of the reading tasks has a variation of almost 3 seconds for reading 100 MB of data per task (figure 73). For the execution of the reading tasks required to read 200MB of data per task a variation of 4 seconds was observed(figure 77. A similar variation was also observed for the tasks required to read 1 GB of data each (figure 75).

The execution times of the writing tasks are the most varying. In the first case, the tasks required to write 100 MB of data each have an execution time variation of 6 seconds (74.

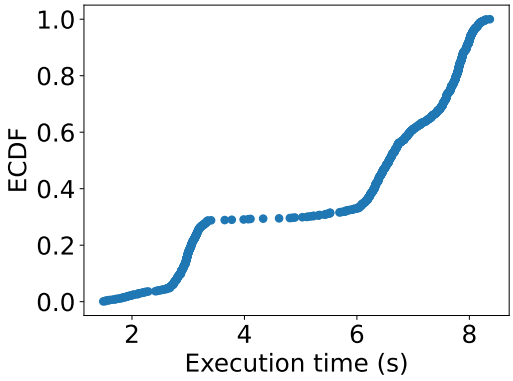

Figure 90: Fetch result time for the *Determine Categories Stage* of the serverless sorting application of an experiment with 1000 cloud functions.

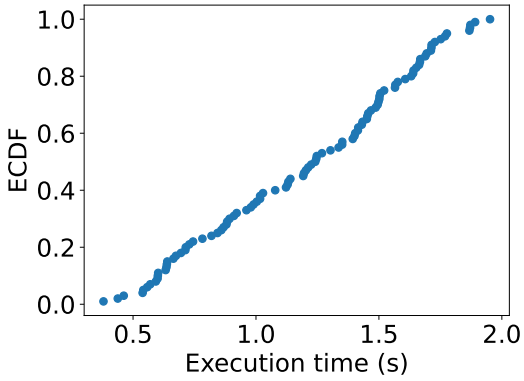

Figure 91: Fetch result time for the *Determine Categories Stage* of the serverless sorting application of an experiment with 100 cloud functions.

In the second case, writing 200 MB of data per task has a variation of 23 seconds. In the last case, the tasks required to write 1 GB of data complete anywhere between 12 and 92 seconds.

In conclusion, the analysis revealed that for each task performed by a cloud variability in the completion time always exist. Even though the number of tasks that require longer times to complete is very small compared to the total number of tasks of the same kind involved in a stage, the fact that task completion time can vary directly affects the execution time of the stage to which the tasks belong to and in consequence the execution time application.

This analysis is summarized in the observation **O-11** and contributes towards achieving **Experimental Goal 4**.

5.1.7 Fetch Results Time

At each stage of the applications data from the previous stages is required. In this sense, each stage must wait until not only the previous stage has finished, but also wait for Lithops to gather the result data of the previous stage.

Figures 90 and 91 depict the delay from the moment the cloud functions finished their execution and the moment Lithops retrieved the functions' results for the first stage of the sorting application. The graphs represent the data points gathered throughout the 10 runs of an experiment. As it can be observed some time is spent before all the results can be gathered. Only after this point the application can continue and can start a new stage. However, comparing the two figures (90 and 91), we observe that the smaller the number of functions that were running for the stage, the faster the results are gathered.

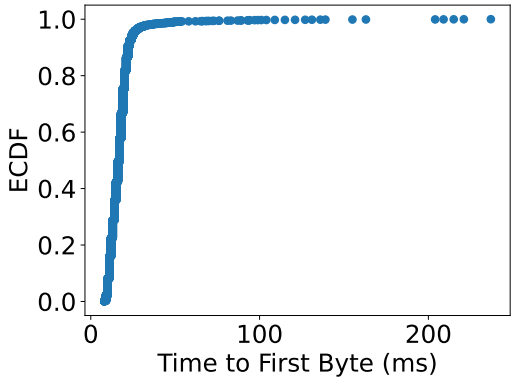

Figure 92: Time to first byte for the GET requests on the S3 objects used during experiments.

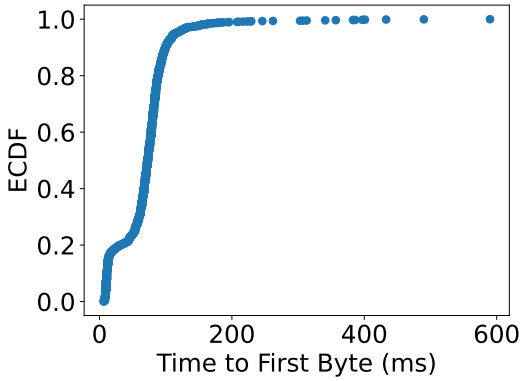

Figure 93: Time to first byte for the PUT requests on the S3 objects used during experiments.

5.1.8 Time to First Byte

Figures 92 and 93 represent the time to first byte (TTFB) measured during experiments for the GET and PUT requests of the applications to AWS S3 storage. As it can be observed, the GET requests have a lower TTFB than the PUT requests as the majority of the GET requests are answered in less than 50ms while the majority of the PUT requests have around 100ms TTFB. Moreover, the GET requests have outliers of maximum 300ms while the PUT requests reach up to 600ms.

5.1.9 Observations

In this section we presented our analysis on the execution of the serverless applications in the AWS cloud environment. In this process we analysed the execution times of different tasks performed by the cloud functions that ran during our experiments. Moreover, we observed the cold start times required by the cloud functions, as well as the limitations imposed by the Lithops library, which we used for handling the functions. A list of the main observations is available below:

- **O-7. Cold Start**

  We observed that cloud functions require a cold start in order to create the execution environment where they can run. This is especially the case for the cloud functions that execute during the first stage of an application. The cloud functions executing at a later stage usually reuse the execution environments created by the previous cloud functions. Nonetheless, if more cloud functions are required to execute in the next stage than they were required to execute in the previous stage, then the additional cloud functions have a cold start. Our measurements showed that cold starts can take up to 1.6 seconds and it directly impacts the application execution time.

- **O-8. I/O Operations Tasks**

In the subsections 5.1.3 and 5.1.4 we analysed the execution times of the tasks performing I/O operations, while in Section 5.1.5 we analysed the execution times of processing tasks. All tasks were carried out by the Lambda AWS cloud functions used in the experiments of the serverless sorting application and the serverless TPC-DS application. In Section 5.1.3 we present the analysis of the execution of large I/O operations and in Section 5.1.4 we analyse the execution of small I/O operations. We define large I/O operations by referring to reading and writing operations of data objects of 100 MB and up to 1 GB. Similarly, we define small I/O operations by referring to reads and writes of data objects of few hundred bytes and up to 200Kb. Moreover, we define as processing tasks, the operations where sorting is performed, in the case of the serverless sorting application, or queries are executed, in the case of the serverless TPC-DS application.

The analysis revealed that the cloud functions have, in general, a steady execution, with the majority of their I/O operations tasks finishing in the same timeframe. We observed this regardless of the number of concurrent active functions. However, there is always a small percentage of tasks that take longer to finish.

- **O-9. Processing Tasks**

  We observed steady execution for the processing tasks. However, a small percentage of the cloud functions require more time to finish. This translates in an increased execution time for the entire application, since the applications require that all clouyd functions executing at a certain stage must finish before proceeding to a next stage.

- **O-10. Lithops limitations**

  We noticed the application execution time was impacted by limitations of the Lithops library. In this sense, we observed that the library was not capable of starting all the necessary cloud functions in the same time, but it did it rather sequential. For 1000 cloud functions to start, it would require 3.5 seconds, in general. Similarly, we observed that the results returned by the cloud functions were not fetched immediately when they were made available, but it would require up to 8 seconds when fetching results from 1000 cloud functions.

- **O-11. Application execution time**

  All applications used in our analysis execute in multiple stages, therefore the application execution time represents the sum of the execution times of all its stages. At each stage, different numbers of cloud functions are started up in order to execute the given workload. All cloud functions executing at a certain stage are given the same amount of work to complete. To complete a stage, all cloud functions executing it must finish their execution and the results must be fetched by the client handler. As we previously observed, the completion time of a stage is influenced by the execution time of different tasks, such as I/O operations tasks and processing tasks, as well as by the functions' cold starts of the cloud functions and by the limitations imposed by Lithops. Because there is variability in the execution of every task, whether it is cold start, I/O and processing tasks or Lithops tasks, and because an application stage is complete only when all functions finished their execution, the execution time of a certain application stage is the sum of the highest execution time of each type

of task. However, this number can be slightly smaller, as most of the time it is not the same cloud function that require the maximum time for all the tasks.

5.2 Comparison Between the Execution of the Serverless Sorting Application and the Execution of the Worker-Sorting Applications

Question 3: What are the performance differences when comparing the execution of the data-intensive serverless sorting application and the execution of the data-intensive sorting applications implementing different workload parallelization techniques in cluster environments?

The serverless sort application was deployed on the AWS cloud and used AWS Lambda cloud functions for its execution, while the worker sorting applications were deployed on the DAS-6 cluster and executed the workload in parallel by assigning jobs and tasks to separate processes on the computing nodes. Because the applications were deployed in different environments, an exact comparison of their execution times cannot be made.
We could observe that for the serverless sorting application the cloud functions had, in general, a steady execution, most of them finishing their work in the same time frame. However, in the case of the worker sorting applications, variability in the execution times of the jobs and tasks was observed. Moreover, the biggest variability was observed for the I/O operations tasks. The reason for this is given by the bandwidth availability. The more concurrent I/O operations tasks there are, the less bandwidth is available for each of them. However, we could observe that this was not an issue in the case of the serverless sorting application. We can conclude that the cloud functions have guaranteed bandwidth, while in the case of the worker sorting applications, the jobs and tasks running in parallel must share the available bandwidth.
In the case of the serverless application we could observe overhead inherited from the cold start of the cloud functions. Moreover, we observed limitations imposed by the Lithops library which was used for managing the execution of the cloud functions. However, in the case of the worker applications these overheads were not present. The applications were up and running at the moment of receiving the workload, therefore no cold start was required. Moreover, the worker applications do not use Lithops for managing the workload, but it is passed by the client to the workers using REST APIs. The client is able to parallelize the calls for submitting the workload, thus doing it instantly.

6 Discussion, Conclusion and Future Work

6.1 Discussion

In this work we analyse the performance of the AWS serverless cloud environment through the usage of two different applications. We developed a sorting application and a TPC-DS application that make use of AWS Lambda cloud functions and can be deployed in the AWS cloud. The applications can be configured to handle different workloads, such that we can analyse the performance of the serverless cloud environment under different conditions. Moreover, we design the serverless sorting application into a worker application that can be deployed on regular computing nodes and we implement two different approaches. The first approach only aims at transforming the serverless sorting application into a worker application, where each job that is handled by a cloud function is, instead, handled by a separate process on the computing node. The second approach goes a step further and the application is able to first divide the jobs into smaller tasks in order to process them in parallel. The worker applications were developed to offer a good overview of the difference in the execution of the same workload by using cloud functions and parallel processes on multiple computing nodes.

Cold start in serverless applications

While experimenting with the serverless applications, it was observed that the cloud functions need to perform additional actions in order to execute their code. In this sense, each cloud function has an initialization phase where the execution environment gets created. The execution environment gathers the necessary resources, installs the required packages and compiles the code. The initialization phase is also referred as cold start. The experiments revealed that only the first batch of functions require to go through the initialization phase, while the other functions can reuse the execution environments previously created. However, the execution environments get disbanded after a certain amount of time which makes that some functions, although they run at a later stage of the application, are still affected by a cold start. As the serverless applications execute in multiple stages, each stage requiring a number of cloud functions to run and all of them must complete before moving to the next stage, the cold start directly affects the execution time of the entire application.

I/O operations tasks and processing tasks in serverless applications

By analysing the execution of different tasks ran by the cloud functions, we could observe that their completion time is generally constant. This includes reading tasks, processing tasks and writing tasks. Moreover, it is important to note that the execution times were not affected by the number of cloud functions running concurrently, although all of them require to access the same AWS S3 storage location to read and write the data. However, for every type of task we could observe a small number of outliers that would require longer times to complete. This is, again, a drawback to the execution time of the stage, thus of the entire application.

Comparison between the worker sorting application with job parallelization and the worker sorting application with task parallelization

Next, we ran experiments with the worker applications. The applications were deployed on the computing nodes from the DAS-6 Vrije Universiteit cluster. For the experiments multiple computing nodes were used, each node hosting one application instance. The workload is orchestrated by a client handler that divides it equally among the available application instances and delivers it in the form of jobs. The first implementation is able to process the jobs in parallel by assigning each job to a separate process on the node, while the second implementation first divides the jobs into smaller tasks and then processing them in parallel. Each computing node has 24 CPU cores, therefore, each worker application was limited at executing a maximum of 24 processes in parallel. Moreover, while the first approach handles a fixed number of 24 jobs in parallel, for the second approach we were able to distribute the 24 processes in different proportions for each type of task. We experimented with different numbers of processes per type of tasks such that we can obtain faster stage execution time, thus faster application execution time. In this way, we wanted to eliminate as much as possible the probability that some processes become idle while waiting for work to be passed, as well as minimizing the execution time of each task. By comparing the two approaches we can observe that the task parallelization approach obtained better results for both the individual tasks, as well as for the execution times of each stage. In some cases we observe an improvement of up to 40% in the stage execution time (Figures 57 and 26, Figures 58 and 27).

Comparison between the worker sorting applications and the serverless sorting applications

Because the serverless sorting application was deployed on the AWS cloud environment and the worker applications were deployed on the DAS-6 cluster, we could not create a relevant comparisons between their executions. However, on one hand, it can be observed that the cloud functions have a much more predictive execution, with constant completion times, while the worker applications do not. The major drawback of the worker application is caused by the reading and writing data from and to the storage system. The processes performing the I/O operations must share the bandwidth with the other processes that are active at that moment. Because the number of active processes is constantly changing, their execution time is varying. On the other hand, although the cloud functions execution times are not influenced by the number of instances executing in parallel, they require cold starts, which represents an additional overhead that is added to the execution time of the application.

6.2 Conclusion

We guided our work through the means of five main research questions. The questions were answered in Chapters 4 and 5 where we present the results of our experiments using multiple applications deployed in the AWS cloud, as well as in the DAS-6 Vrije Universiteit cluster.

RQ1. What is the performance impact of sorting big data in cluster environments?

In order to answer this question we designed and implemented the radix sort algorithm within two different sorting applications. Each application uses different

workload parallelization techniques in order to efficiently process the work. One of the sorting applications performs job parallelization, where each job consists of several tasks executed sequentially, while the other sorting application performs task parallelization, where each individual task is scheduled for parallel processing.

RQ 1.1. What is the performance of data-intensive sorting application with job parallelization in cluster environments?

This application processes the workload in parallel by assigning each job to a separate process on the computing node. In order to analyse the performance of this application, we divided the job in three main tasks - reading tasks, processing tasks and writing tasks. The experiments revealed that the execution times of each type of task is not reliable, but is rather varying. Smaller variations could be observed for the processing tasks. However, for the reading and writing tasks higher variations were observed. The main reason for this is that all tasks concurrently running at a certain moment in time must share the bandwidth when reading and writing data from and to the storage system.

RQ 1.2. What is the performance of data-intensive sorting application with task parallelization in cluster environments?

For this application we observed similar behavior as in the case of the worker application that uses job parallelization. Variation was observed in the execution of the tasks, mainly due to the fact that the bandwidth must be shared among the running tasks.

RQ 1.3. What workload-parallelization technique shows better performance for the data-intensive sorting application in cluster environments?

The worker sorting application that uses task parallelization is able to better accommodate the workload. The application defines three different pools of processes, each of them dedicated to one of the three types of tasks. Although the number of the same tasks running concurrently is smaller than in the case of the other worker application, using task parallelization makes it possible to execute some tasks of each category in the same time. In this way, there are also less tasks that must share the bandwidth and as a consequence the reading and writing tasks execution time increases. Comparing the two approaches we observed that in some cases the worker application using task parallelization obtain up to 40% faster execution times.

RQ2. What is the performance impact of sorting big data in serverless environments?

The experiments revealed that the execution times of the stages is generally constant. However, we observed that for every experiment outliers were present. Because of how the application is designed, no stage is finished and the application cannot advance, as long as there are still cloud functions running the certain stage. This directly impacts the execution time of the entire stage.

RQ3. What are the performance differences when comparing the execution of the data-intensive serverless sorting application and the execution of the data-intensive sorting applications implementing different workload parallelization techniques in cluster environments?

Because the serverless sorting application were deployed on AWS cloud and the worker applications were deployed on the DAS-6 cluster, it was not possible to create an in-depth comparison of their executions. However, we could observe that experiment runs performed on the serverless application offered more reliable results with execution times being generally constant, while for the worker application this was not the case. The execution times of experiments ran on the worker application are highly influenced by the number of concurrent tasks that must share the bandwidth. However, by using task parallelization we could improve on the execution time of the worker sorting application. Moreover, the serverless sorting application inferred overhead from the cold starts of the cloud functions, as well as from the limitations imposed by the Lithops library.

6.3 Future Work

In this thesis we developed several data-intensive applications that implement diverse workload parallelization technique, as well as serverless architectures. First, we implemented the radix sorting algorithm into two different worker sorting applications following the MapReduce model, and we deployed them on the DAS-6 cluster. Secondly, we implemented two different serverless applications that make use of FaaS, a serverless sorting application and a TPC-DS application. Although we only experimented on two different cloud platforms, the DAS-6 cluster and the AWS cloud, all the applications are cloud agnostic and can be deployed on some of the most popular cloud platforms such as Amazon Cloud, Microsoft Azure, Google Cloud and IBM Cloud. Moreover, the applications are highly configurable, being very easy to set up experiments with different workloads. Also, the applications implement functionality to automatically generate the necessary workload for every experiment configuration. In this sense, further analysis on the cloud platforms can be done using the benchmark applications developed in this work, such as:

- **Other cloud platforms**

  Analyse and compare the execution of the execution of the data-intensive applications on the most popular cloud platforms (Amazon Cloud, Microsoft Azure, Google Cloud and IBM Cloud). Also, it is interesting to analyse the performance of open-source cloud platforms such as Alibaba and OpenShift. It is interesting to observe what are the trade-offs between different cloud platforms in terms of I/O operations performance, computing power and cold starts.

- **Cost analysis**

  Create a cost analysis. Analyse and compare costs of running the same experiments on different cloud platforms. Observe what are the costs of each process involved in the execution of the benchmark applications, such as I/O operations tasks, processing tasks and cold starts.

A TPC-DS queries

The TPC-DS queries are extracted from the source code available on the official TPC-DS website[10].

TPC-DS-1

```
WITH customer_total_return AS
(SELECT
 sr_customer_sk AS ctr_customer_sk,
 sr_store_sk AS ctr_store_sk,
 sum(sr_return_amt) AS ctr_total_return
  FROM store_returns, date_dim
  WHERE sr_returned_date_sk = d_date_sk AND d_year = 2000
  GROUP BY sr_customer_sk, sr_store_sk)
SELECT c_customer_id
FROM customer_total_return ctr1, store, customer
WHERE ctr1.ctr_total_return >
  (SELECT avg(ctr_total_return) * 1.2
  FROM customer_total_return ctr2
  WHERE ctr1.ctr_store_sk = ctr2.ctr_store_sk)
  AND s_store_sk = ctr1.ctr_store_sk
  AND s_state = 'TN'
  AND ctr1.ctr_customer_sk = c_customer_sk
ORDER BY c_customer_id
LIMIT 100
```

The query is implemented throughout 8 stages, each of them solving a different part of the query. The first 4 stages are dedicated to the **WITH** statement, whereas the rest 4 are dedicated to the main **SELECT** statement.

- **Stage 1**. At stage 1 the *date_dim* table is retrieved from the storage and all the rows with year 2000 are selected. The selected rows are saved to an intermediate file on the cloud storage system.

- **Stage 2**. At stage 2 the *store_returns* table is retrieved from the storage and the entire *sr_returned_date_sk* column is saved to a new intermediate file on the cloud storage system.

- **Stage 3**. At stage 3 the intermediate files created at stages 1 and 2 are downloaded from the storage system and their data is merged on the condition that *sr_returned_date_sk = d_date_sk*. The result is uploaded to an new intermediate file on the cloud storage system.

[10] https://www.tpc.org/tpc$_{d}ocuments_{c}urrent_{v}ersions/current_{s}pecifications5.asp$

- **Stage 4**. At stage 4 the intermediate files created at stage 3 is downloaded, grouped by *sr_customer_sk* and *sr_store_sk* and the *SELECT* statement is performed. The result is then uploaded to an intermediate file.

- **Stage 5**. At stage 5 the *customer* table is downloaded and the *c_customer_sk* and *c_customer_id* columns are selected. The results are saved to an intermediate file on the cloud storage.

- **Stage 6**. At stage 6 the result data of stages 4 and 5 are merged on the condition that *ctr_customer_sk* = *c_customer_sk*. The results are then uploaded to an intermediate file on the cloud storage.

- **Stage 7**. At stage 7 the *state* table is downloaded *s_state* and *s_store_sk* columns are selected. The results are saved to an intermediate file on the cloud storage.

- **Stage 8**. At stage 8 the result data of stages 6 and 7 are merged on the condition that *ctr_store_sk* = *s_store_sk*. The result is grouped by *ctr_store_sk* and the mean of the *ctr_total_return* is calculated. The mean of the *ctr_total_return* is used to construct the main **WHERE** clause. In the end the main **SELECT** statement is resolved and the result is uploaded to the cloud storage.

TPC-DS-16

```
SELECT
 Count(DISTINCT cs_order_number) AS 'order count' ,
 Sum(cs_ext_ship_cost)           AS 'total shipping cost' ,
 Sum(cs_net_profit)              AS 'total net profit'
FROM      catalog_sales cs1 ,
 date_dim ,
 customer_address ,
 call_center
WHERE     d_date BETWEEN '2002-3-01' AND       (
           Cast('2002-3-01' AS DATE) + INTERVAL '60' day)
AND       cs1.cs_ship_date_sk = d_date_sk
AND       cs1.cs_ship_addr_sk = ca_address_sk
AND       ca_state = 'GA'
AND       cs1.cs_call_center_sk = cc_call_center_sk
AND       cc_county IN ('Williamson County',
                        'Williamson County',
                        'Williamson County',
                        'Williamson County',
                        'Williamson County' )
AND       EXISTS
          (
                 SELECT *
                 FROM   catalog_sales cs2
```

```
                WHERE   cs1.cs_order_number = cs2.cs_order_number
                AND     cs1.cs_warehouse_sk <> cs2.cs_warehouse_sk)
AND     NOT EXISTS
        (
                SELECT *
                FROM   catalog_returns cr1
                WHERE  cs1.cs_order_number = cr1.cr_order_number)
ORDER BY count(DISTINCT cs_order_number)
LIMIT 100;
```

The query is split among 6 stages as follows:

- **Stage 1**. At stage 1 columns *cs_order_number, cs_ext_ship_cost, cs_net_profit, cs_ship_date_sk, cs_ship_addr_sk, cs_call_center_sk, cs_warehouse_sk* are retrieved from the *catalog_sales* table and are saved to an intermediate file.

- **Stage 2**. At stage 2 column *cr_order_number* is retrieved from the *catalog_returns* table and the result is saved to an intermediate file.

- **Stage 3**. At stage 3 the order numbers of the *catalog_sales* from the results of stage1 that are found within the order numbers from *catalog_returns* from the results of stage2 are selected. Finally the *date_dim* table is retrieved and the **WHERE** clause involving dates is resolved. The result of the two procedures are merged and saved to an intermediate file.

- **Stage 4**. The column *ca_address_sk* from *customer_address* table where the *ca_state* column value equal 'GA' is retrieved and saved to an intermediate file.

- **Stage 5**. This stage merges result of stages 3 and 4 on the condition that *cs_ship_addr_sk* (from results of stage3) = *ca_address_sk* (from results of stage4). Moreover, the *ca_address_sk* column from the *call_center* table where the county is one of the counties in the list 'Williamson County', 'Williamson County', 'Williamson County', 'Williamson County', 'Williamson County'. The result of the merge is then merged with the filtered *call_center* table.

- **Stage 6**. At this stage the sums of the *cs_ext_ship_cost* and *cs_net_profit* on results of stage 5.

TPC-DS 94

The query is split into 6 stages as follows:

- **Stage 1**. At this stage the columns needed for making the query (*ws_order_number, ws_ext_ship_cost, ws_net_profit, ws_ship_date_sk, ws_ship_addr_sk, ws_web_site_sk, ws_warehouse_sk*) from the *web_sales* table are selected. The results are saved to an intermediate file on the cloud storage.

- **Stage 2**. At this stage the *wr_order_number* column from the *web_returns* table are retrieved and saved to an intermediate file.

- **Stage 3**. First it sums up the *ws_warehouse_sk* grouping the *web_sales* table data from the results of stage 1 by *ws_order_number*. From the results it extracts the rows with *ws_warehouse_sk* greater than 1. Next the results are used to match stage 1 results against them based on the order number. Following up the results of this are merged with the rows that have the *d_date* between '1999-02-01' and '1999-04-01' in the *date_dim* table. The merged data is saved to an intermediate file.

- **Stage 4** At this stage ca_address_sk column is retrieved from table customer_address where the ca_state == IL. The results are saved to an intermediate file.

- **Stage 5** First results of stage 3 and 4 are merged. Next the *web_site_sk* column is retrieved from table *web_site* where *web_company_name = pri*. The results of the two are then merged on $ws\_web_site\_sk = web\_site\_sk$.

- **Stage 6** The last stage calculates the sum of *ws_ext_ship_cost*, *ws_net_profit* and counts *ws_order_number* based on the results of stage 5.

```
SELECT
      Count(DISTINCT ws_order_number) AS 'order count' ,
      Sum(ws_ext_ship_cost)           AS 'total shipping cost' ,
      Sum(ws_net_profit)              AS 'total net profit'
FROM
      web_sales ws1 ,
      date_dim ,
      customer_address ,
      web_site
WHERE d_date BETWEEN '1999-02-01' AND (Cast('1999-04-01' AS DATE) + INTERVA
AND   ws1.ws_ship_date_sk = d_date_sk
AND   ws1.ws_ship_addr_sk = ca_address_sk
AND   ca_state = 'MT'
AND   ws1.ws_web_site_sk = web_site_sk
AND   web_company_name = 'pri'
AND   EXISTS
      (
            SELECT *
            FROM   web_sales ws2
            WHERE  ws1.ws_order_number = ws2.ws_order_number
            AND    ws1.ws_warehouse_sk <> ws2.ws_warehouse_sk)
AND       NOT EXISTS
      (
            SELECT *
            FROM   web_returns wr1
            WHERE  ws1.ws_order_number = wr1.wr_order_number)
```

```
ORDER BY count(DISTINCT ws_order_number)
LIMIT 100;
```

The query is implemented throughout 8 stages as follows:

- **Stage 1**. During the first stage columns *ws_order_number, ws_warehouse_sk* are retrieved from the *web_sales* table and are saved to an intermediate file.

- **Stage 2**. At stage 2 an aggregation of the *ws_warehouse_sk* values is performed grouped by the *ws_order_number* on the data resulted from stage 1. In the end, all the rows with *ws_warehouse_sk* greater than 1 are retrieved and saved to an intermediate file.

- **Stage 3**. At the third stage the columns *ws_order_number, ws_ext_ship_cost, ws_net_profit, ws_ship_date_sk, ws_ship_addr_sk, ws_web_site_sk, ws_warehouse_sk* are retrieved from table *web_sales*. The result is then saved to an intermediate file.

- **Stage 4**. At this stage the *wr_order_number* column is retrieved from the *web_returns* table and the results are saved to an intermediate file.

- **Stage 5**. At this stage the application resolves the **EXISTS** subquery. Moreover, it solves the date related part of the **WHERE** clause and merges the two results.

- **Stage 6**. During the sixth stage the *ca_address_sk* column is retrieved from table *customer_address* for all the rows where *ca_state == 'IL'*.

- **Stage 7**. At stage 7 the results of stage 5 and 6 are merged. Moreover, the *web_site* table is filtered and all the rows where *web_company_name == 'pri'* are fetched. The two results are again merged and columns *ws_order_number, ws_ext_ship_cost, ws_net_profit* are saved to an intermediate file.

- **Stage 8**. At the last stage the sum for *ws_ext_ship_cost* and *ws_net_profit* are calculated as well as the order numbers being counted. The calculations are made on the results from stage 7.

TPC-DS 95

```
SELECT
   count(DISTINCT ws_order_number) AS `order count `,
   sum(ws_ext_ship_cost) AS `total shipping cost `,
   sum(ws_net_profit) AS `total net profit `
FROM
   web_sales ws1, date_dim, customer_address, web_site
WHERE
   d_date BETWEEN '1999-02-01' AND
   (CAST('1999-02-01' AS DATE) + INTERVAL 60 days)
```

```
AND ws1.ws_ship_date_sk = d_date_sk
AND ws1.ws_ship_addr_sk = ca_address_sk
AND ca_state = 'IL'
AND ws1.ws_web_site_sk = web_site_sk
AND web_company_name = 'pri'
AND EXISTS(SELECT *
           FROM web_sales ws2
           WHERE ws1.ws_order_number = ws2.ws_order_number
             AND ws1.ws_warehouse_sk <> ws2.ws_warehouse_sk)
AND NOT EXISTS(SELECT *
               FROM web_returns wr1
               WHERE ws1.ws_order_number = wr1.wr_order_number)
ORDER BY count(DISTINCT ws_order_number)
LIMIT 100
```

The query is implemented throughout 8 stages as follows:

- **Stage 1**. During the first stage columns *ws_order_number, ws_warehouse_sk* are retrieved from the *web_sales* table and are saved to an intermediate file.

- **Stage 2**. At stage 2 an aggregation of the *ws_warehouse_sk* values is performed grouped by the *ws_order_number* on the data resulted from stage 1. In the end, all the rows with *ws_warehouse_sk* greater than 1 are retrieved and saved to an intermediate file.

- **Stage 3**. At the third stage the columns *ws_order_number, ws_ext_ship_cost, ws_net_profit, ws_ship_date_sk, ws_ship_addr_sk, ws_web_site_sk, ws_warehouse_sk* are retrieved from table *web_sales*. The result is then saved to an intermediate file.

- **Stage 4**. At this stage the *wr_order_number* column is retrieved from the *web_returns* table and the results are saved to an intermediate file.

- **Stage 5**. At this stage the application resolves the **EXISTS** subquery. Moreover, it solves the date related part of the **WHERE** clause and merges the two results.

- **Stage 6**. During the sixth stage the *ca_address_sk* column is retrieved from table *customer_address* for all the rows where *ca_state == 'IL'*.

- **Stage 7**. At stage 7 the results of stage 5 and 6 are merged. Moreover, the *web_site* table is filtered and all the rows where *web_company_name == 'pri'* are fetched. The two results are again merged and columns *ws_order_number, ws_ext_ship_cost, ws_net_profit* are saved to an intermediate file.

- **Stage 8**. At the last stage the sum for *ws_ext_ship_cost* and *ws_net_profit* are calculated as well as the order numbers being counted. The calculations are made on the results from stage 7.

References

[1] *5 Biggest Companies That Use AWS*. `https://www.insidermonkey.com/blog/5-biggest-companies-that-use-aws-915759/`. Accessed: 2022-08-17.

[2] *ASCII Code*. `https://www.ascii-code.com/`.

[3] Ioana Baldini et al. "Serverless computing: Current trends and open problems". In: *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[4] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. "Cloud computing: A study of infrastructure as a service (IAAS)". In: *International Journal of engineering and information Technology* 2.1 (2010), pp. 60–63.

[5] Paul Castro et al. "The rise of serverless computing". In: *Communications of the ACM* 62.12 (2019), pp. 44–54.

[6] CL Philip Chen and Chun-Yang Zhang. "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data". In: *Information sciences* 275 (2014), pp. 314–347.

[7] Tyson Condie et al. "MapReduce online." In: *Nsdi*. Vol. 10. 4. 2010, p. 20.

[8] Kamil Figiela et al. "Performance evaluation of heterogeneous cloud functions". In: *Concurrency and Computation: Practice and Experience* 30.23 (2018), e4792.

[9] MaryAnne M Gobble. "Big data: The next big thing in innovation". In: *Research-technology management* 56.1 (2013), pp. 64–67.

[10] Joseph M Hellerstein et al. "Serverless computing: One step forward, two steps back". In: *arXiv preprint arXiv:1812.03651* (2018).

[11] Alexandru Iosup et al. "Massivizing computer systems: a vision to understand, design, and engineer computer ecosystems through and beyond modern distributed systems". In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2018, pp. 1224–1237.

[12] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. "Evaluation of production serverless computing environments". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 442–450.

[13] Jimmy Lin and Chris Dyer. "Data-intensive text processing with MapReduce". In: *Synthesis Lectures on Human Language Technologies* 3.1 (2010), pp. 1–177.

[14] Wes Lloyd et al. "Serverless computing: An investigation of factors influencing microservice performance". In: *2018 IEEE international conference on cloud engineering (IC2E)*. IEEE. 2018, pp. 159–169.

[15] Garrett McGrath and Paul R Brenner. "Serverless computing: Design, implementation, and performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2017, pp. 405–410.

[16] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing". In: (2011).

[17] Raghunath Othayoth Nambiar and Meikel Poess. "The Making of TPC-DS." In: *VLDB*. Vol. 6. Citeseer. 2006, pp. 1049–1058.

[18] Josep Sampe et al. “Outsourcing Data Processing Jobs with Lithops”. In: *IEEE Transactions on Cloud Computing* (2021).

[19] Joel Scheuner and Philipp Leitner. “Function-as-a-service performance evaluation: A multivocal literature review”. In: *Journal of Systems and Software* 170 (2020), p. 110708.

[20] Erwin Van Eyk et al. “Beyond microbenchmarks: The spec-rg vision for a comprehensive serverless benchmark”. In: *Companion of the ACM/SPEC international conference on performance engineering*. 2020, pp. 26–31.

[21] Blesson Varghese et al. “A survey on edge performance benchmarking”. In: *ACM Computing Surveys (CSUR)* 54.3 (2021), pp. 1–33.

[22] Mario Villamizar et al. “Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures”. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE. 2016, pp. 179–182.

[23] Liang Wang et al. “Peeking behind the curtains of serverless platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 133–146.

[24] *Who’s Using Amazon Web Services?* `https://www.contino.io/insights/whos-using-aws`. Accessed: 2022-08-17.