Bachelor Thesis

ShareBench: Performance Characterization of Distributed Resource-Sharing Mechanisms

Author: Lennart K.M. Schulz (2734873)

1st supervisor: Prof. dr. ir. Alexandru Iosup
daily supervisor: Sacheendra Talluri, MSc
2nd reader: Dr. Daniele Bonetta

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

November 5, 2024

"You are something the whole universe is doing in the same way that a wave is something that the whole ocean is doing."

- Alan Watts

Abstract

Global data production is increasing rapidly, and our modern society increasingly relies on the availability and operation of, often warehouse-sized, data centers to satisfy the various demands to process this data. However, the energy consumption and carbon footprint of these computing facilities are a prevalent global issue. To meet the escalating computational demands, increases in efficiency are crucial. One effective strategy is to share resources among multiple applications, thereby improving resource utilization. There are numerous such distributed resource-sharing mechanisms and policies, each with distinct performance characteristics. Understanding how individual mechanisms compare is essential for their deliberate and appropriate utilization and should underpin the development of new mechanisms.

This thesis aims to contribute by proposing a systematic approach to analyzing the real-world performance of distributed resource-sharing mechanisms and policies. We design and implement both a workload generator and an infrastructure framework for automated real-world performance analysis studies of such mechanisms for Spark SQL on Kubernetes, and subsequently use both components to characterize the performance of three resource-sharing mechanisms. Our experiments demonstrate that performance varies significantly with workload characteristics, emphasizing the importance of informed decisions in the choice of mechanisms and development of novel alternatives.

Acknowledgements

I would like to thank Sacheendra Talluri for taking the time to supervise this work and introducing me to the world of distributed resource-sharing mechanisms. My gratitude also goes to Alexandru Iosup for his highly insightful meetings that always left me with a strong desire to learn more, more often than not by showing me the extent of what is still unknown to me. I also want to thank my lovely friends, not only for their academic support in each of our many collaborative study sessions, but also for giving me a welcome distraction from *sitting at a desk* just when I needed it every so often. Lastly, I want to express my deepest gratitude to my partner and my family, who had to see me stressed and frustrated more often than I would have liked. You repeatedly managed to get me back on my feet, gave me new perspectives, and helped me to keep pushing.

Contents

1 Introduction

Data is everywhere. It is collected in all thinkable situations from sports [1] to education [2], finance [3], medicine [4], transportation [5], and many others; often fully transparent to the producer [6]. With the ever-growing breadth and depth of data collection, the amount of digital data generated per year has increased steadily [7] and current estimates predict an annual digital data production of approximately 291 zettabytes (ZB) by 2027 [8]. To put this number in perspective: storing 291 ZB of data on commonly used 64 GB microSD cards would require more than 4.5×10^{11} cards, enough to physically cover over $100,000$ football fields[1] when placed next to each other, fill up three-quarters of the Empire State Building in volume, or reach the moon more than 11 times when stacked. This explosive growth is not surprising, given the increasingly low cost associated with and the business value possibly derived from large amounts of data [9, 10]

Although production and storage of data by themselves can already be viable businesses, more often than not, some processing is necessary for the data to be used to its full potential [11]. This *processing* can have many forms, from simple queries for a specific selection of data to complex multistage pipelines that sanitize, filter, transform, and combine data, to name just a few. Due to various reasons, including the *end* of Moore's Law [11, 12] and the increasing complexity and scale of such data-processing applications in both scientific and non-scientific computing, a single machine is often unable to offer enough performance to make (timely) completion possible. The work is instead divided into multiple parts that are then executed concurrently by separate machines, a *cluster*. However, writing such distributed applications is no simple task and requires careful consideration. Application Frameworks (AFs) simplify the process by offering common abstractions, hiding the intricacies of (efficiently) distributing the work and subsequently

[1]Assuming a 100×68 m field size as used in, among others, the Wembley stadium and Allianz Arena.

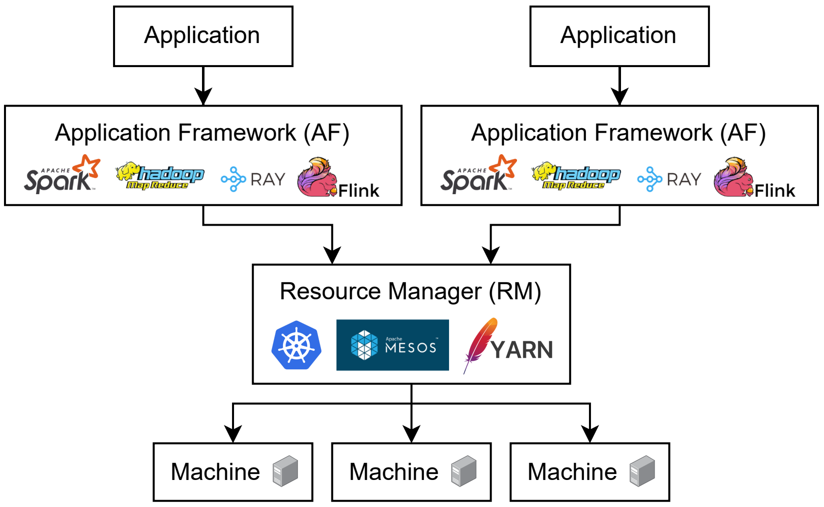


Figure 1.1: Composition of two application frameworks and a resource manager in a three-node cluster, including examples of common choices for both components.

collecting the results[1], which allow programmers to write highly distributed programs using simple building blocks. Some popular AFs include *Spark* [13, 14], *Hadoop MapReduce* [15], *Flink* [16], and *Ray* [17].

Even though such distributed applications use multiple machines to speed up the computation, they often need the (full) processing power of those machines only for brief periods. Idle machines, however, continue to consume substantial energy [18], leading to significant operational costs. This cost is further compounded by the fact that unused resources represent a lost opportunity for value as they could be used by other applications. To mitigate this, software systems known as Resource Managers (RMs) make it possible for many applications to share a set of resources by coordinating their allocation and scheduling.[2] Popular resource managers include *Kubernetes* [19], *YARN* [20], and *Mesos* [21] , among many others, with frequent advances originating from both industry and academia.

Figure 1.1 illustrates the previously introduced composition with the examcple of two applications running in a cluster of three machines. Each application uses an AF. Both are deployed through an RM to coordinate allocation and scheduling on a set of machines, effectively sharing the available resources.

[1]A rather grand simplification of the sophisticated functions of many AFs.
[2]A rather minimal summary of the intricate mechanisms of many RMs.

1.1 Problem Statement

The mechanism and policy by which resources are shared between applications[1] cannot only differ between various RMs but even for a single composition there are often many options. Although a thorough understanding is imperative for good utilization of existing mechanisms and the development of new mechanisms, **we identify a critical lack of knowledge regarding the performance characteristics of these distributed resource-sharing mechanisms and policies.** (**P1**)

Evaluating the performance characteristics of resource-sharing mechanisms, however, is not trivial and comes with many challenges. One is the complexity of the required infrastructure. While even complex algorithms of theoretical computer science, for instance, can often be evaluated on a single machine, research into massivizing computer systems commonly requires (as the name would suggest) a distributed system composed of numerous machines. Without an infrastructure that is representative of the systems used in practice evaluation of distributed resource-sharing mechanisms and policies is either not possible at all or at least will not produce meaningful results. However, **setting up a representative infrastructure is technically challenging and may pose an unfeasible overhead for some research.** (**P2**)

Having a functional infrastructure, while necessary, is not yet sufficient for the evaluation. Another critical aspect of any performance evaluation is the workload used to test the system. Whether using production workload traces or generating synthetic ones, the choice of workload can greatly influence the outcomes of performance evaluations. **Without carefully considered workloads, evaluations may fail to capture significant characteristics of resource-sharing performance.** (**P3**)

Simulation of real-world workload traces by synthetic recreation of the same load characteristics is a common option as the choice of workload in performance analysis studies [22, 23]. Another alternative is to use generators for fully synthetic workloads, typically using probabilistic distributions to model workloads [24, 25]. However, both of these options fail to offer fine-grained control over specific workload characteristics, especially considering inter-application load, which is needed for performance evaluation of distributed resource-sharing mechanisms and policies for distinct workload characteristics.

Although some of the above problems are addressed to some degree by existing research, that research does not investigate the two-level structure of AF and RM [26, 27] or uses simple *ad hoc* solutions [23, 28, 29]. Based on these observations, **we identify a lack of**

[1]Hereinafter simply referred to as *resource-sharing mechanism* or simply *mechanism*.

a systematic and generally applicable process for evaluating the performance characteristics of distributed resource-sharing mechanisms and policies. (**P4**)

1.2 Research Questions

The goal of this work is to answer the **Main Research Question (MRQ)**: *How to systematically analyze the real-world performance of distributed resource-sharing mechanisms and policies?*

Since the scope of this work is limited by time constraints[1], the objective is to propose, implement, and evaluate a system for a specific composition of AF and RM, namely Spark SQL on Kubernetes, but design it so that the findings are applicable for similar compositions with different components, thereby answering the main research question in its generality and addressing **P4**. This process is guided by the following subquestions, each aimed at addressing one of the problems illustrated in Section 1.1.

Research Question 1 (RQ1):

How to design and implement a workload generator for performance analysis studies of distributed resource-sharing mechanisms and policies? As discussed in **P3**, carefully considered workloads are essential for performance analysis studies. Especially for performance characterization of distributed resource-sharing mechanisms, it is important to have great control over the inter-application load characteristics of the workload. Existing approaches fail to provide such control and thus, while usable for "simple" performance evaluations[2], are ill-suited for answering the main research question. This question is therefore aimed at designing and implementing a new kind of workload generator that is geared toward multi-application workloads and offers extensive control over the inter-application load characteristics.

Research Question 2 (RQ2):

How to design and implement an infrastructure framework for automated real-world performance analysis studies of distributed resource-sharing mechanisms and policies? **P2** highlights a significant obstacle on the way to building a better understanding of the large variety of distributed resource-sharing mechanisms and policies.

[1]The Bachelor Thesis at VU Amsterdam is a 15 ECTS (420 hours) project, intended to be completed within 3 months.

[2]The word "simple" here refers to the evaluations that try to capture the overall performance but not necessarily the performance characteristics.

To address that problem and facilitate subsequent experiments with different mechanisms, AFs, and RMs this research question does not simply tackle the implementation of a one-off infrastructure, but is more concerned with designing a complete infrastructure framework that is highly automated and reusable for other research by various researchers.

Research Question 3 (RQ3):

What are the performance characteristics of the resource-sharing mechanisms of Spark on Kubernetes? The final research question aims to demonstrate the capabilities of the proposed solutions for **RQ2** and **RQ1** and take a step toward addressing **P1**, by evaluating three resource-sharing mechanisms of Spark SQL on Kubernetes.

1.3 Research Methodology

System Design and Implementation:

For both **RQ1** and **RQ2**, we follow the *AtLarge Design Process* [30] in conjunction with recognized and structured software architecture and software engineering practices as described by Sommerville [31] and Bass *et al.* [32], respectively. By that, we repeatedly iterate through: (i) problem analysis, (ii) architecture design, (iii) prototype implementation, and (iv) testing and validation, until the design gives a satisfactory answer to the associated research question.

The AtLarge Design Process has repeatedly shown to be successful in over a decade of use, leading to numerous publications, some of which highly cited [33–37].

Experimental Research:

For **RQ3**, we use experimental research methods, following the best practices in the field for the design, conduction, and analysis of the experiments [38–40]. To perform the experiments, we: (i) define performance metrics, (ii) use the design of **RQ1** to generate workloads, (iii) use the design of **RQ2** to set up the experiment infrastructure, including the collection of metrics, (iv) perform the experiments, and finally (v) collect and analyze the resulting data. Steps (ii) and (iii) are where the research questions interleave as advances in the designs allow more experiments and subsequent experiences uncover new requirements or missing features of the designs.

Open-Science:

All research of this work follows the principles of open science [41, 42], making the code and all data publicly available on GitHub[1], strengthening reproducibility and promoting further research.

1.4 Thesis Contributions

By answering the research questions, we produce several contributions.

Conceptual

(i) We propose the design of a workload generator for performance analysis studies of distributed resource-sharing mechanisms and policies. The design stands out through its simplicity and versatility, being applicable to any type of workload that can be represented as a collection of discrete work units with defined and foreseeable durations.

(ii) We propose the design of an infrastructure framework for automated real-world performance analysis studies of distributed resource-sharing mechanisms and policies. The conceptual design of the framework proposes a set of (abstract) components and processes to automate various types of experiments.

(iii) We provide a detailed analysis of the performance characteristics exhibited by three resource-sharing mechanisms of Spark on Kubernetes. Based on our various findings, we propose actionable insights for the use of existing mechanisms and the development of new alternatives.

Technical

(i) We provide an implementation of the workload generator for Online Analytical Processing (OLAP) workloads based on the TPC-DS data set and queries. With extensive control over the generation process, the generator can produce various workloads with specific characteristics and should be usable for various research projects.

(ii) We provide an implementation of the infrastructure framework for Spark SQL on Kubernetes. Through the use of commonly known components and a modular structure, the proposed implementation is customizable and extensible for future research projects.

[1]https://github.com/atlarge-research/ShareBench

1.5 Societal Relevance

Iosup *et al.* highlight the dependence of our modern society and economy on the various computer systems functioning of which has become a substantial requirement for many jobs and a large share of the GDP in the Netherlands [43]. In their manifesto, the authors further state four *grand societal challenges*.

The research questions of this work are, in their grand scheme, concerned with building a better understanding of distributed resource-sharing mechanisms and policies, which are highly relevant in this era of (*hyperscale*) cloud data centers, subsequently enabling better use of existing mechanisms and facilitating the development of more advanced alternatives. With that, the findings of this work can help to address three of the four challenges.

Better understanding of existing mechanisms can help to give more accurate performance predictions and avoid failures to improve availability. (**Challenge 2: Responsibility**) With better new mechanisms and more appropriate use of existing mechanisms, resources can be shared more efficiently, allowing more work to be performed by the same (existing) resources. This can not only reduce the energy footprint, as less resources are needed (**Challenge 3: Sustainability**), but may also lower the operational cost for the same reason, improving the general accessibility to computer systems (**Challenge 4: Usability**).

1.6 Plagiarism Declaration

I hereby confirm that the contents of this thesis are a product of my own independent work and writing. The work does not contain material copied from any other source (person, internet, or LLM) unless otherwise indicated, and has not been submitted for assessment elsewhere.

1.7 Thesis Structure

In Chapter 2 we present relevant background information and briefly discuss related work. To best understand the work and its position in the larger perspective, it is recommended to start traversing the thesis there. Chapters 3, 4, and 5 address **RQ1**, **RQ2**, and **RQ3**, respectively. They largely function independently, with previously introduced concepts briefly reexplained, and can thus be traversed selectively and in any order if preferred.

For those with little time to spare, each chapter begins and ends with a summary of the most important findings, which should be sufficient to get a high-level overview. For those with even less time to spare, Chapter 6 summarizes the complete work in a few pages. We do, however, recommend to read the work in its entirety.

2

Background and Related Work on Distributed Computing and Resource-Sharing

The context of this work, positioned in distributed systems, encompasses many concepts for which a brief introduction may be useful to aid in the understanding of the subsequent chapters. In this chapter, we aim to provide such introductions for the most important concepts. We furthermore present an overview of some related work in the field.

2.1 Introduction to Data Centers: Relevance and Emerging Issues

Processing of data at large scales is commonly done in data centers. Considered the “central nervous system of the 21st century” [44] they house not only servers but also the networking and storage equipment needed to support the various types of demands for computation, transportation, and storage of data. Data centers can range from a single server rack in an office back room to warehouse-sized facilities housing thousands to tens of thousands of servers [45]. However, the information and communications technology (ICT) landscape is seeing a continuous shift away from traditional (on-premise) data centers in favor of large-scale *cloud* data centers [46]. Not only are such “hyperscale” data centers able to improve the overall efficiency of the equipment with better resource utilization [47] and more advanced cooling techniques [48], but they also allow customers to get compute resources on demand, at any time, and with pay-per-use pricing models (as opposed to the upfront investment and continuous operating costs of self-deployed servers).

Functioning of our society increasingly depends on data centers. Large parts of the commercial sector, medical infrastructure, governments, education, and scientific research, to name just some examples, rely heavily on the availability of servers for their daily operational processing needs [10, 49–52]. In an effort to satisfy this demand, there are thousands of data centers worldwide. In the US, data centers consume electricity in the hundreds of TWh per year, accounting for 4% of the total electricity demand in 2022, with higher fractions expected in the coming years [53]. The energy consumption of Google alone was responsible for close to a million metric tons of CO_2 in 2023 [54], and Big Tech companies take significant investments to reduce the carbon footprint of their data centers [55].

Although advances in data center hardware technology continue to improve their energy efficiency, improvements in other areas will also be needed to satisfy our growing demands for computing over the coming years [11]. Better resource utilization through more advanced resource-sharing techniques could help address this challenge of our modern society.

2.2 Introduction to OLAP and Interactive Workloads

Big Data computing was classically associated with batch jobs that were characterized by long execution times and little to no restrictions on maximum latency. For several decades now, the landscape has gradually shifted towards more interactive and time-sensitive jobs [23, 56–58]. One type of data processing consisting largely of such jobs is Online Analytical Processing (OLAP) which describes complex data querying and analysis frequently used in business intelligence to gain data insights and increase business value [59].

2.2.1 Interactive Workloads

In the case of OLAP applications, a workload can be thought of as a timeline of queries submitted to the system. When visualizing such a timeline, the expected duration of the queries is used to plot the outstanding work over time. Figure 2.1 shows an example workload graph for a single application where the x-axis denotes time, and the y-axis denotes the number of active queries (that is, queries that have been submitted for processing but are not yet finished). The figure is additionally annotated by markers on the x-axis indicating query submission (green star) and completion (red diamond) events to clarify the example.

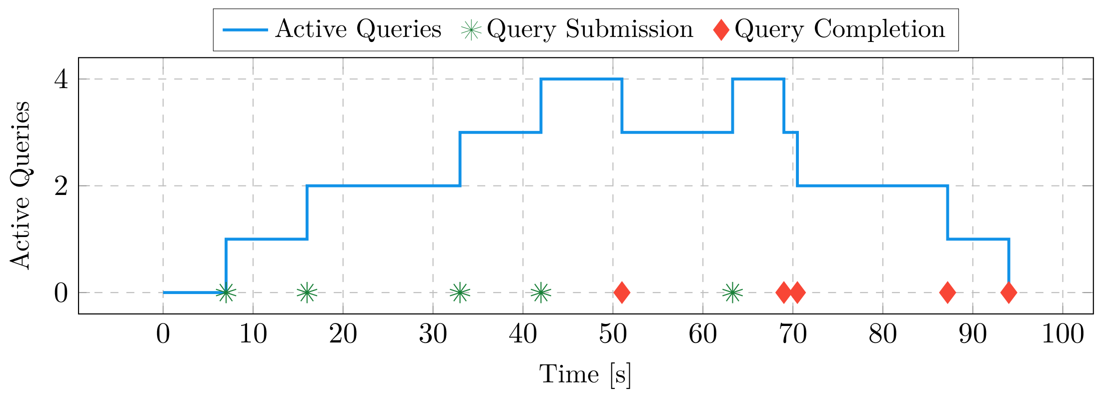


Figure 2.1: Example of a workload graph for a single application. Indicators for query submission and query completion events are added for clarification.

2.2.2 OLAP Systems and Spark SQL

Distributed processing systems are needed to handle the scale of modern data analytics and OLAP applications [56]. Although there are specialized systems like Google BigQuery [60], Microsoft SQL Server Analysis Services [61], and Oracle OLAP [62], SQL engines built on top of big data platforms that have emerged recently have gained popularity as alternatives [63]. A common advantage of the latter is the integration with open-source systems like *Hadoop* [64], which have become the "de facto processing platform for big data" and are often significantly cheaper than traditional databases for data storage [10].

Spark SQL [65] is such an SQL engine built on top of the popular Application Framework (AF) Spark, which we selected for this research due to its performance, wide-range of applications, open-source nature, and popularity [66–69]. A Spark deployment consists of a *driver* and one to many *executors*. The driver is responsible for dividing and distributing the work among the executors, collecting the results and handling executor failures.

2.3 Resource Managers for Spark

Spark offers a *Standalone* mode [70], where executors are manually started and connected to the driver. More commonly however, Spark is deployed through a Resource Manager (RM), such as YARN, Mesos, or Kubernetes, although support for Mesos has been deprecated since Spark 3.2.0 [71].

While the Standalone mode has a significant advantage over the others for short-running jobs due to its comparatively low launch overheads [72, 73], it most notably falls short in terms of resource utilization due to its static allocation of resources. RMs, on the other hand, enable the deployment of multiple heterogeneous applications concurrently on the same cluster, dynamically allocating resources as needed.

YARN has been the *de facto* standard as the RM for Spark deployments, due to its deep integration into the Hadoop stack. However, setup, configuration, and maintenance of YARN clusters can be difficult, in part due to the need to set up the entire Hadoop stack [72], and Castro *et al.* further mention "limited reproducibility and portability across infrastructures" as a limitation of YARN for scientific computing [52].

Kubernetes has established itself as a popular solution for automated deployment of containerized applications, with all major cloud providers offering Kubernetes services as part of their Platform-as-a-Service (Paas) suite [74–76]. The rather novel (Spark 2.4.0 in late 2018 [77]) addition of Kubernetes to the supported RMs of Spark and the wide availability of Kubernetes compute services facilitate cloud native Spark deployments without the need for experience in setting up and managing clusters and thus bring high-performance big data computing to a wide range of potential users. Due to the popularity of Kubernetes and the novelty of the composition with Spark, this work focuses on Kubernetes as the RM.

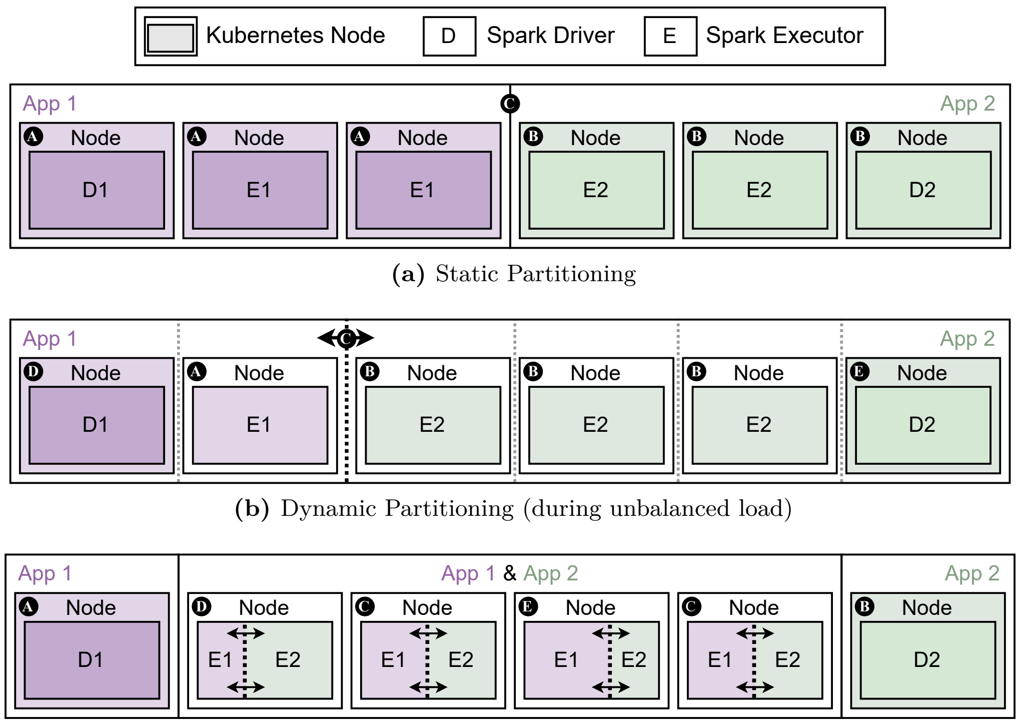
A Kubernetes cluster consists of a *control plane* and one to many worker *nodes*. Applications are deployed in the form of *pods* and each node can accommodate multiple pods, depending on the available resources and requirements of the pods. When Spark is deployed on Kubernetes, the Spark driver and the executors are individual pods.

2.4 Resource-Sharing Mechanisms of Spark on Kubernetes

When running multiple Spark applications on Kubernetes, the configurations of both the AF and the RM can be modified to achieve different resource-sharing mechanisms and policies. Figure 2.2 illustrates the three distinct mechanisms identified for this work.

While the figure uses the example of two Spark applications being deployed on a Kubernetes cluster with six nodes (not including the control plane), the mechanisms are analogously applicable to larger clusters and more applications.

It should be noted that this list of mechanisms is not exhaustive. Not only can the more fine-grained policies of each mechanism be further configured, but combinations of mechanisms are also possible. Other AFs and RMs may moreover offer different mechanisms and policies [21, 78]. These selected mechanisms, however, should represent a wide range of characteristics and thereby allow a meaningful performance characterization and comparison.



(a) Static Partitioning

(b) Dynamic Partitioning (during unbalanced load)

(c) Node-Level Sharing

Figure 2.2: Visualizations of the three resource-sharing mechanisms of Spark on Kubernetes identified for this work.

Static Partitioning Each of the Kubernetes nodes is assigned to a specific Spark application, effectively separating the cluster into multiple smaller parts. The executors (and the driver) of each application are then scheduled on the dedicated nodes, with each node exclusively being used by a single executor (or the driver).

Figure 2.2a shows the aforementioned cluster statically partitioned for two Spark applications. The nodes marked with Ⓐ are exclusively available for the first application, while the nodes marked with Ⓑ are similarly only available for the second.

With complete separation of nodes, highlighted by the boundary Ⓒ, this approach minimizes interference between applications; however, it is likely to under-utilize the available resources of the cluster when the load is unbalanced.[1]

Dynamic Partitioning Each executor is again using a node exclusively (similar to Static Partitioning); however, the allocation of nodes to applications is no longer static. The Spark applications are configured to terminate executors after a certain inactivity

[1]For example, in the case where some application does not have enough tasks to saturate its allocated resources while another application has outstanding tasks available but cannot use the idle resources

threshold [79], thus effectively freeing a node in the cluster which can then be used to accommodate a new executor of another application.

Figure 2.2b shows the same cluster as before, this time with Dynamic Partitioning. The first application currently uses only one executor (🅐), while the second has taken advantage of the available resources by deploying a total of three executors (🅑). Conceptually, the applications are still fully separated; however, this time the separation boundary 🅒 can dynamically adjust in both directions as the load between the applications changes. The drivers of each application (🅓 and 🅔) behave the same as before, as they can never be terminated as long as the application is still running.

As shown by Kaufmann *et al.*, Dynamic Partitioning, while improving executor utilization, is detrimental to the overall runtime for a workload predominantly composed of short-running tasks due to the overhead of starting new executors [28]. However, when the timescales are larger, Dynamic Partitioning could improve the overall performance due to better resource utilization.

In the context of schedulers and, or more specifically, autoscalers, Dynamic Partitioning is similar to *horizontal autoscaling* which describes increasing or decreasing the number of allocated resources according to demand [80, 81].

Node-Level Sharing Nodes are no longer exclusively occupied by a single executor, eliminating partitioning altogether. By oversubscribing the available CPU resources of a node, multiple executors (from various applications) are co-scheduled on the same node instead.

Figure 2.2c again shows the same cluster as before, but this time with Node-Level Sharing. While the deployment of the drivers (🅐 and 🅑) remains unchanged, the remaining nodes (🅒) are no longer exclusive to one application but rather accommodate multiple executors at the same time. If one executor does not need its full share of the available resources, the other one will be able to use the additional resources for itself. Nodes 🅒 show a balanced load, while nodes 🅓 and 🅔 show unbalanced distributions with executors of App 1 and App 2 respectively not using the full share of their allocated resources, allowing the other executor to use more than their fair share.

Although full parallelism has the potential to increase the overall performance by better utilizing the available resources if the CPU is not the bottleneck resource [68], interference between executors may impact performance predictability, and contention for the available CPU cycles may also negatively affect the overall performance if the load is high.

This mechanism of oversubscription is commonly used by cloud providers to utilize unused (but allocated) resources [81, 82].

2.5 Workloads in Related Work

To understand the options of workloads for performance analysis studies of distributed computing frameworks such as Spark, we surveyed existing work in the domain by exploring related papers and identified four common types of workload used that can be categorized into two categories.

Single job & sets of jobs The first category is that of workloads that consist of a single job or sets of jobs, but do not include any timely structuring, but are rather based on running individual jobs one after another.

The first and arguably simplest type of workload is to use a single application such as Word Count, Grep, or Page Rank. Although simple to apply, such a workload may suffer from bias, as only a specific part of the system is evaluated. A common option is to evaluate the system at the hand of multiple such applications to counteract this bias [83, 84].

A more extensive evaluation with varying performance demands is given by benchmark suites that offer workloads composed of many parts, each of varying characteristics, which altogether should represent the full, or at least a wide range of possible performance profiles. These benchmark suites can be further categorized into platform-independent and platform-specific, with the latter referring to benchmarks suited only for a single platform or system (e.g., Spark). Platform-independent benchmarks used in related work include *TPC-DS* [85], *BigDataBench* [86], and *BigBench* [87], while platform-specific benchmarks include *SparkBench* [68], *SparkBench*[1] [88], and *HiBench* [89].

Structured workloads The second category are *structured* workloads that not only include the work to be performed but also specify when each job should be submitted.

The first option here is to use (simulations of) real-world workload traces, collected from the live operation of some production system [22, 23]. The other option is to generate synthetic workloads, often based on probabilistic distributions and assumptions about the characteristics of the workload [24, 25, 90, 91].

All of these options found in related work are either only applicable to a single application or offer no direct control over the inter-application load characteristics and thus are, as

[1]This is indeed not a typo, but rather two individual benchmarks by different authors but with the same name.

mentioned in Chapter 1, not usable for the experiments considered in this work. However, a study of these existing approaches, especially workload generators, is important for addressing **RQ1** (Chapter 3).

2.6 Related Work on Performance Characterization of Distributed Computing Systems

There are various studies of the performance characteristics in distributed computing frameworks, some of which are similar to those considered in this work. Li *et al.* characterize various Spark workloads based on their performance profiles and resource demands [68]. Marcu *et al.* and Ahmed *et al.* compare the performance characteristics of Spark to those of Flink and Hadoop, respectively [83, 84]. Lastly, Lattuada *et al.* build a system for characterizing Spark applications to predict execution times and estimate the minimal required resources [92].

2.7 Related Work on Automated Experiments

Faciliatation of experiments through automation frameworks is not a novel area of research. In his 2003 paper, Pawlikowski describes a tool for automated control of simulation experiments in the context of improving the credibility of simulation results [93]. Perrone *et al.* follow similar motives and propose an automation framework for experiments in network simulation studies [94]. Closest to the framework proposed in this work, however, is the work of Silva *et al.* on *CloudBench* [95], a tool for automated experiments to compare the performance of various cloud providers for a given application.

3

ShareBench-Gen: Design and Implementation of the Workload Generator

Main Contribution 3.1 (MC3.1):
Analysis and elicitation of requirements for a workload generator.

Main Contribution 3.2 (MC3.2):
A simple, yet highly versatile conceptual design for a workload generator, applicable for any type of workload represented as discrete work units of predictable duration.

Main Contribution 3.3 (MC3.3):
An implementation of the workload generator design for OLAP workloads based on the TPC-DS data set and queries.

There are a multitude of benchmarks for Online Analytical Processing (OLAP) systems, such as those considered in this work, already available [68, 85–88], so the simplest choice for a workload would be to run such a benchmark. However, these benchmarks lack two critical features that are needed.

Firstly, the benchmarks are testing a single application, so do not include the idea of multiple such systems running concurrently on the same resources. Secondly, even if two instances of a benchmark were combined, one per application, this workload would still fail to capture the range of possible inter-application load characteristics, such as, for example, imbalanced or alternating loads between the applications.

To mitigate this and enable a wide range of experiments that can capture the perfor-

Table 3.1: Summary of requirements for the workload generator including priority, validation method, and status in the final design and implementation.

ID	Name	Priority	Method	Status
RE3.1	*Core Functionality*	★	❖	●
RE3.2	*Visualization*	★	❖	●
RE3.3	*Variability*	★	❖	●
RE3.4	*Reproducibility*	★	❖	●
RE3.5	*Documentation*	★	✎	●
RE3.6	*Data Independence*	☆	✎	●
RE3.7	*Scalability*	☆	✎	◑
RE3.8	*Query Diversity*	☆	❖	✻

Priority: ★ mandatory | ☆ desirable
Method: ✎ by design | ❖ by design, implementation, and real-world evaluation
Status: ● fully | ◑ partially | ○ not met | ✻ depends on data

mance of resource-sharing mechanisms in various load characteristics, we formulated **RQ1**, asking *how to design and implement a workload generator for performance analysis studies of distributed resource-sharing mechanisms and policies.* In this chapter, we address this question by following the *AtLarge Design Process* [30] to propose and evaluate the design and implementation of *ShareBench-Gen*, a workload generator based on the TPC-DS data set and queries.

The design and implementation are guided by a set of requirements, elicited in Section 3.1 and summarized in Table 3.1. The table shows the priority, the validation method, and whether a requirement is met in the final design and implementation.

3.1 Requirements Analysis

We started to elicit the core requirements for the workload generator by using the utility tree technique [32]. Subsequently, we built a simple prototype on the basis of these requirements and used it extensively. From this use, we collected shortcomings of the prototype and used those findings to formulate further requirements. The resulting list of requirements is given below. Mandatory requirements are specified using "shall", desirable requirements using "should" [31].

RE3.1 *Core Functionality* The generator shall generate workloads with various characteristics, controllable through the given parameters.

For performance characterization studies aimed at investigating the effect of various workload characteristics on the object of study, it clearly is crucial that different workloads with varying characteristics can be generated. To enable controlled research with specific workload characteristics, it is furthermore essential that the characteristics in generated workloads can be controlled.

RE3.2 *Visualization* The generator shall be able to visualize the workload to better understand the result.
It requires significant effort to understand the characteristics of a workload from a technical description, like a list of query submissions. A graphical representation is much more suitable to a human reader [96] and therefore a necessary component of the generator.

RE3.3 *Variability* The generator shall be able to generate various workloads with the same characteristics, varying only in minor details (e.g., choice of queries).
This functionality is needed to support studies in which workloads with certain characteristics are to be evaluated repeatedly but including slight variations to produce more generally applicable results.

RE3.4 *Reproducibility* The generator shall always give the same result when given identical parameters and query data.
Calibration and fine-tuning of the workload to fit the system at hand are essential processes for many experiments. A generator that produces varying results even for the same parameters would greatly increase the difficulty of this process or make it fully impossible.

RE3.5 *Documentation* The generator shall be documented properly.
Even though the design should be intuitively usable, documentation can help better utilize the full functionality and avoid issues from a lack of understanding about its function.

RE3.6 *Data Independence* The generator should function independent of the type of underlying query data.
The design is supposed to address the issue of how to generate workloads not only for this work but further research to come. For that, it is important to keep the design detached from the underlying data, such that the same design is usable for experiments with different data.

RE3.7 *Scalability* The generator should be scalable to any (reasonable) number of applications, intensity, and total duration.
As mentioned in **RE3.6**, the design should not only apply for this work. Therefore,

to enable experiments on various scales, it is important that there are no arbitrary limitations on the scalability of the workloads.

RE3.8 *Query Diversity* The generator should compose workloads of as many different queries as possible.
Composing workloads of only a small number of distinct queries would likely lead to a bias, as the specific characteristics and performance requirements of those queries dominate the workload. To get more general representation of the performance of a system, workloads should include a diverse set of queries.

The following requirements were elicited but are not considered for this work due to the limited time available. They are instead intended to guide future work of extending the workload generator design.

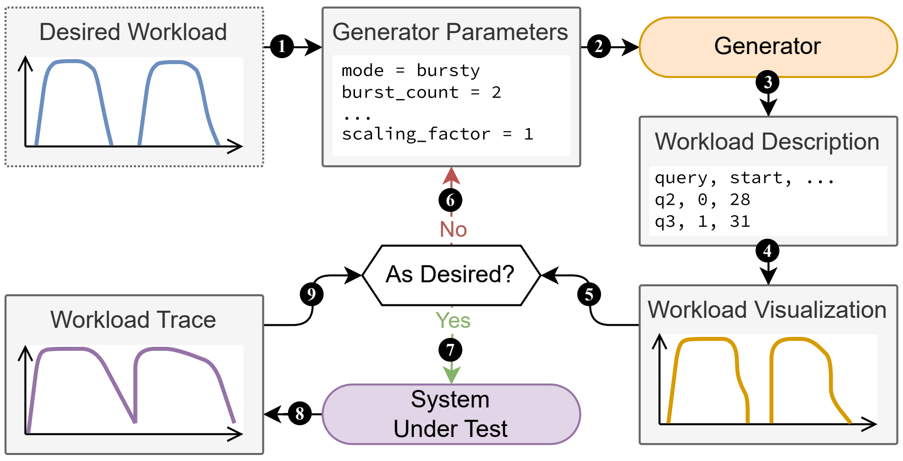
RE3.9 *Consideration of Query Characteristics* The generator should consider performance characteristics of queries in its process.
Queries can have highly varying performance profiles [56, 97]. Some queries may, for instance, read large amounts of data but only do light processing, while others may read less data but perform much more complex computations. By considering those characteristics, the generator could (i) provide more detailed information about the generated workload, and (ii) allow for more control over workload characteristics.

RE3.10 *Modeling of Workload Traces* The generator should be able to generate workloads that re-create or model existing traces.
Researchers may want to use real-world workloads for their experiments. There are numerous workload traces available freely; however, re-running them is often not easily possible, as the traces are typically limited to information about the start and end times of queries (e.g., in the *Snowflake Dataset* [40]).

3.2 Conceptual Design

Figure 3.1 illustrates an example of use of the workload generator, including testing and fine-tuning of the generated workload in the form of a flowchart.

The example starts with some desired workload, which is expressed as parameters (❶) and passed to the generator (❷). The generator uses the parameters to generate a workload, as a workload description in the form of a table (❸). To understand the generated workload, it is subsequently visualized (❹). The user can then compare this visualization



```
mode = bursty
burst_count = 2
...
scaling_factor = 1
```

```
query, start, ...
q2, 0, 28
q3, 1, 31
```

Figure 3.1: Example of use of the workload generator, including stages of testing and fine-tuning.

to their (mental) image of the desired workload (❺), and adapt the parameters to repeat the process and fine-tune the workload if needed (❻).

If the workload matches the expectation, it can be submitted to the system under test (❼), producing a trace of the actual workload run (❽). The performance of the system under test may vary, so the expected query durations used to generate the workload might not match reality, and thus need to be calibrated based on a scaling factor. For this, the trace is again compared to the desired and generated workload (❾) and the parameters (typically just the *scaling factor*) adapted accordingly (❻), repeating the subsequent steps until the actual workload exhibits the desired characteristics.

3.2.1 Generation Process

The design of the generation process itself follows a simple pattern, outlined based on an example in Figure 3.2.

The example here shows the generation of a workload consisting of two bursts (🅐); other types of workload are generated with the same steps, but possibly in different patterns of repetition. The set of all available queries 🅑 is filtered for queries for which the execution duration is in the required range. This subset of all queries 🅒 is then sampled. Depending on the number of queries needed and the number of available queries, this sampling step may produce a smaller or larger set (by sampling with replacement) of queries 🅓. These queries are then "placed" in the timeline[1] (🅔).

[1]This formulation is to better visualize the process. What technically happens is that queries are assigned a start time and added to a table of all query submissions that constitute the workload.

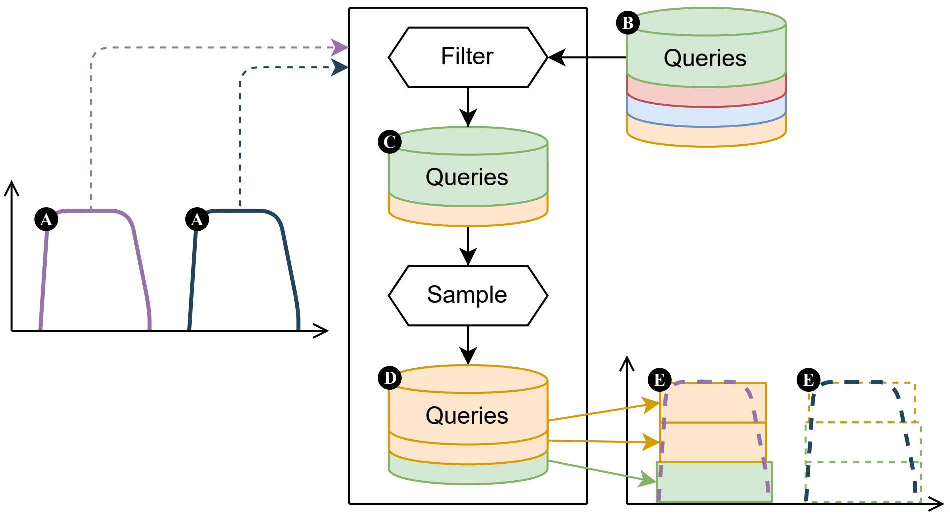


Figure 3.2: Conceptual design of the generation process.

3.2.2 Generator Functions

The described generator pattern is realized in various generator functions for workloads with different characteristics. The proposed design includes three main types of workloads and corresponding generator functions which are described below.

Random consists of queries that get submitted at fully random times within a given range. Intensity (i.e., average number of query submissions per time minute), range of query durations, and overall workload duration can all be configured.

Constant consists of queries arriving in constant intervals. Intensity, query duration, query duration variability, and query interval variability can all be configured.

Bursty consists of a series of bursts, where a number of queries are submitted concurrently (i.e., a burst) with no activity in between. The burst intensity, burst count, burst interval, query duration, inter-burst query start offset, offset per application, and random variation of many parameters can all be configured. Bursty characteristics are commonly found cloud computing [24].

It is to note here that not only can a single function be used to create a range of different workloads based on the parameter values but also composite types are possible through combining individually generated workloads. We selected this set of workload types because it can be used to represent many core characteristics found in production workload traces as found in the *Snowflake Dataset* [98].

It is vital that all parts of the generator that use randomness allow for explicitly setting a seed for the random numbers such that the exact same workload can be generated repeatedly (e.g., in the case of trying to fine-tune the scaling factor). With these generators, their versatile configurations, and many possible combinations, a wide range of workloads with numerous characteristics can be created and tuned at ease.

3.3 Implementation with TPC-DS Queries

We implement the proposed design as a generator for OLAP workloads, based on the TPC-DS data set and queries. The implementation can be categorized into three parts, each of which is explained below.

3.3.1 Data and Queries

TPC-DS is an industry standard benchmark for OLAP applications, offering both a data set generator and a set of queries with various characteristics. The benchmark models a data warehouse, a common type of OLAP system which can be described as "a copy of transaction data specifically structured for query and analysis." [99] As stated at the start of the chapter, the benchmark alone is not sufficient as a workload as it simply involves running the set of queries in succession and recording the completion time for each. Yet, the data set and queries are usable components for the workload generator.

The data set models the sales process of a multi-channel sales organization, structuring the data with multiple snowflake schemas which are widely used in practice [100], and can, as a whole, be scaled to various sizes to accommodate the evaluation of a range of differently scaled systems [56]. The queries of the benchmark focus on representing the diversity of operations and system requirements apparent in information analysis applications [85, 101, 102].

3.3.2 Modification of Queries

The number of queries that are included in the benchmark, however, is rather limited, having 99 queries in total [102]. Their distribution in terms of execution time is furthermore uneven. Many queries have similar execution times, resulting in certain runtime ranges being densely populated, while others are scarcely covered or not covered at all [97, 103]. Especially when trying to create workloads consisting of queries with a specific duration, this set would quickly lead to a low variety of queries.

Some simple studies of query behavior revealed that modifying the range of data included in the query positively correlates with the execution duration for many queries. Figure 3.3 shows the process that we use on some of the existing queries to mitigate the above-stated

```
SELECT sr_customer_sk
FROM store_returns , date_dim
WHERE d_year = 2000 AND ...
```

(a) Original query

```
SELECT sr_customer_sk
FROM store_returns , date_dim
WHERE ($DATERANGE$) AND ...
```

(b) Query with placeholder

```
WHERE (d_date = cast('2000-04-30' as date)) AND ...
```

```
WHERE (d_year = 2000 and d_moy = 1) AND ...
```

```
WHERE (d_year = 2000 or d_year = 2001) AND ...
```

(c) Query with changed date ranges

Figure 3.3: Process of modifying a single query into various new queries of different scales.

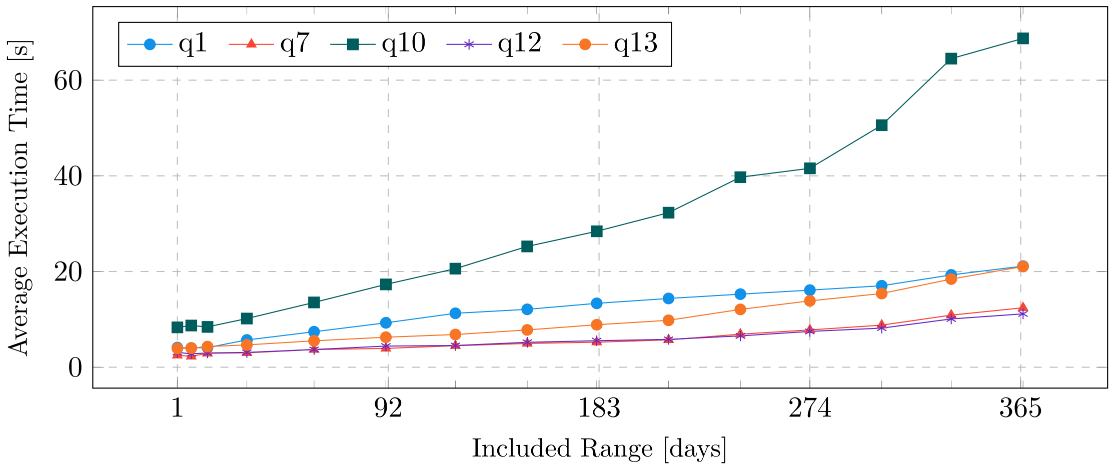


Figure 3.4: Average query execution times over 10 iterations for various ranges of included data.

issue and extend the set of queries to include more queries with various execution times. Figure 3.3a represents (part of) the original query, where a filter based on the date of entries can be seen in line 3. We replace this filter for a specific date range by a placeholder, as shown in Figure 3.3b. This placeholder can then, in an automated process, be replaced with any range (within the available data) to create a new query with a different scale. Example results of this replacement process are shown in Figure 3.3c.

To create a sufficiently sized set of available queries, ten queries were modified with the aforementioned procedure. Of these queries, 50% showed to be scalable in runtime by adjusting the range of included data. Figure 3.4 shows the average execution times of these queries plotted against the number of days included in the filtering step. As can be seen in the figure, the execution times increase with larger ranges of included data. The subsequent steps use these five queries with date ranges from one day up to two years, mostly in intervals of one month.

3.3.3 Query Analysis

Before these queries can be used by the generator, information about their expected execution times must be collected. For this, every query is run 10 times, and the execution time for each run is recorded. The results are then examined for consistency to check whether query execution times are largely consistent over multiple runs. Finally, the average times of the queries are saved to a file for further use in the workload generator.

3.3.4 Generator Functions

The three generator functions, as described in Section 3.2 are implemented as Python functions, using *pandas* [104] for the table data structures and sampling of queries.

For all operations that involve randomness (e.g., sampling of queries from the available options or random variation in start times), the seed for the random generator is set based on the provided seed parameter (**RE3.4**). If no seed is given, the current UNIX timestamp is used instead. This timestamp is also shown to the user to allow re-creating the same workload.

Many of the available parameters have default values and can thereby be omitted in an effort to keep the number of required parameters low and increase the ease of use. The return value is a pandas data frame with columns for the app index, start time, query name, date range, and expected duration.

The generator suite additionally includes a function to visualize the workload based on the resulting data frame (**RE3.2**), and all functions are complemented by *docstrings* explaining the available parameters and the proper use of the function (**RE3.5**).

3.4 Evaluation

We have taken multiple steps to evaluate the functionality of ShareBench-Gen and to determine whether it meets the requirements stated in Section 3.1. Each of the generator functions is tested individually at the hand of various parameter configurations. For each configuration:

- The workload is visualized (**RE3.2**) and examined to confirm whether it exhibits the expected characteristics for the given parameters (**RE3.1**). This process is further aided by creating a boxplot of query durations.

- The set of queries used in the workload is analyzed for the number of unique queries used (**RE3.8**).
- The same configuration is used twice with the same seed, the results are compared for equality (**RE3.4**).
- The same configuration is used twice with different seeds, the results are compared for deviations (**RE3.3**).

Lastly, the set of all generated workloads is analyzed for diversity of characteristics (**RE3.1**).

While initial designs of the workload generator were insufficient for some of the requirements, the implementation of the final design fully meets all mandatory requirements and at least partially meets all desirable requirements. For some requirements, the underlying query data can influence whether they are met or not, but the generator design itself is not limiting. Table 3.1 lists this information for all requirements along with the type of evaluation used for each requirement.

3.5 Limitations

We have identified three issues with the proposed design and implementation of ShareBench-Gen, which could possibly limit its functionality and usability.

Small set of queries Due to the limited time available for this work, only ten queries have been adapted to include a modifiable date range, and of those ten only five proved to be useful for the workload generator. With that, the generated workloads only have a small variety of query types, possibly limiting their scope in testing all aspects of an OLAP system.

No consideration of performance interference The number of concurrent queries for a single application likely affects the performance of all those queries and increases their execution times. The current model does not include such a consideration, possibly affecting workload characteristics and the accuracy of the visualization.

Inefficient implementation The implementation of the generator functions was not aimed at efficiency. Likely, much more efficient solutions are available. However, the current implementation is reasonably fast, even for larger workloads, on most common systems.

3.6 Future Work

While considering the current design and implementation, we have identified multiple possible extensions to ShareBench-Gen that could be addressed in future work.

Consideration of query characteristics In the current form, all queries are considered equal, only differing in their execution duration. However, as stated in **RE3.9**, queries can have highly varying performance profiles. To provide more detailed information on the generated workloads and extend the level of control over the generation process, the generator should be extended to support the consideration of query characteristics. For this extension, a deeper, nontrivial analysis of query performance would be necessary.

Interpolation of queries Figure 3.4 suggests a functional correlation between the number of days included in a query and its execution time. While currently only ranges for which the query has explicitly been tested for are used in the workload generator, an extension of the work could try to model this relation to give more fine-grained control over the desired query duration without requiring more query runtime evaluations by interpolating queries between the tested ranges. This extension is not elicited as a requirement as it is specific to the set of queries used in this implementation of the generator and not generally applicable to the design.

Modeling of workload traces Additionally to the current design, where a workload is described by a range of parameters and then generated from scratch, the generator could be extended to also include the option to re-create a workload from a trace (**RE3.10**). Extending the functionality of the workload generator to enable re-creating a workload of similar shape from a trace would greatly extend its use cases and provide a middle ground between synthetic and real-life workloads. As some workload traces further include meta information, like data read and data written by the query, this extension could additionally be combined with the consideration of query characteristics (**RE3.9**).

3.7 Summary

In this chapter, we addressed **RQ1** through the design (**MC3.2**) and implementation (**MC3.3**) of *ShareBench-Gen*, a workload generator for performance analysis studies of distributed resource-sharing mechanisms and policies. We guided the process by a list of requirements that we identified as necessary for a workload generator. (**MC3.1**) The implementation uses the TPC-DS data set and queries to generate OLAP workloads, but the conceptual design of the generator is applicable to any type of workload that can be represented as work units with limited and predictable durations.

ShareBench-Gen is designed with performance characterization experiments in mind. It is centered around three generator functions that together enable generation of workloads with a wide range of characteristics. Its features should support various experiment workflows, with the options to visualize the generated workloads and directly control the randomization of the process for reproducibility and fine-tuning of workloads.

We successfully used ShareBench-Gen to generate numerous workloads that support the experiments discussed in Chapter 5 and used the gained experience to iteratively improve the design and implementation of the generator.

4

ShareBench-Base: Design and Implementation of the Infrastructure Framework

Main Contribution 4.1 (MC4.1):
Analysis and elicitation of requirements for an infrastructure framework for automated real-world performance analysis studies.

Main Contribution 4.2 (MC4.2):
A generalized, process-based conceptual design for an infrastructure framework for automated real-world performance analysis studies.

Main Contribution 4.3 (MC4.3):
A structural topology of required components for performance analysis studies of Spark SQL on Kubernetes.

Main Contribution 4.4 (MC4.4):
An implementation of the infrastructure framework design for Spark SQL on Kubernetes.

Initially, this work was purely concerned with evaluating distributed resource-sharing mechanisms and policies. However, it quickly became apparent that the infrastructure needed for such experiments is complex, requires numerous heterogeneous parts to work together, and involves many steps even for simple experiments.

The idea of automating frameworks that streamline experimentation processes has been considered by other research in similar areas [93–95]. It soon became clear that an automated infrastructure framework could also here greatly facilitate the process and help future research (and researchers) to focus on the experiments themselves by reducing the

Table 4.1: Summary of requirements for the infrastructure framework including priority, validation method, and status in the final design and implementation.

ID	Name	Priority	Method	Status
RE4.1	*Core Functionality*	★	❖	●
RE4.2	*Configuration*	★	❖	◑
RE4.3	*Documentation*	★	❖	●
RE4.4	*Portability*	★	❖	●
RE4.5	*Automation*	☆	❖	◑
RE4.6	*Extensibility*	☆	✎	◑
RE4.7	*Use of Common Components*	☆	✎	●

Priority: ★ mandatory | ☆ desirable
Method: ✎ by design | ❖ by design, implementation, and real-world evaluation
Status: ● fully | ◑ partially | ○ not met

time and effort needed to install, set up, and coordinate the infrastructure, initiate the experiments, and analyze the results. We therefore formulated **RQ2**, asking *How to design and implement an infrastructure framework for automated real-world performance analysis studies of distributed resource-sharing mechanisms and policies.*

In this chapter, we address **RQ2** by following the *AtLarge Design Process* [30] to propose the design and implementation of an automated infrastructure framework, *ShareBench-Base*. Although ShareBench-Base is geared towards Spark SQL on Kubernetes, it's core architectural design should be equally applicable to various other compositions. The requirements addressed by the design and implementation are summarized in Table 4.1, which includes information about the priority, the validation method, and whether a requirement is met in the final design and implementation.

The framework assumes access to a Kubernetes cluster as a starting point. If such a cluster is not available, the *Continuum* framework by Jansen *et al.* [105] can be used to automatically deploy an emulated Kubernetes cluster on a single or multiple machines.

4.1 Requirements Analysis

We follow the same process for eliciting requirements as described in Section 3.1, starting with the utility tree technique before building a prototype and formulating additional requirements through the experience gained from its use. The resulting list of requirements is given below.

RE4.1 *Core Functionality* The framework shall set up and configure the infrastructure needed for the proposed experiments.

The installation and configuration of the required infrastructure for experiments such as those considered in this work is not trivial. The core objective of this framework is to relieve researchers of the associated effort and time needed.

RE4.2 *Configuration* The automation shall not reduce the options for configuration

A possible drawback of automation and abstraction is the loss of control over the abstracted components.[1] However, especially in a research setting it cannot be assumed that the configuration envisioned in the design will be applicable to all users.

RE4.3 *Documentation* The framework shall be documented properly.

The components and processes of the framework, while ideally less technical than those it abstracts, are still highly technical. To be used to its full extent by researchers that did not follow the design of the framework documentation is essential. Good documentation furthermore facilitates extensions and modifications to the framework.

RE4.4 *Portability* The framework shall be portable (i.e., movable) between different hosts.

Experiments may be performed from various host machines, and the framework should be able to accommodate this.

RE4.5 *Automation* The framework should offer automation and abstraction for all common processes that go beyond simple commands.

Many experiments require extensive processes to initiate, monitor, and analyze. When repeated often enough, time spent on these processes quickly accumulates to nonnegligible amounts, reducing the time that can be spent on subject of research. Automation of such processes is therefore vital for the objective of the framework.

RE4.6 *Extensibility* The framework should be easily extensible for more or different components.

Although the framework is primarily designed to support the type of experiments performed for this work, its benefit to other researchers is greatly increased through support for extensions, as those may facilitate numerous other types of experiments.

RE4.7 *Use of Common Components* The framework should use commonly known components where possible.

By using commonly known components, the framework can be understood easier by

[1]Abstractions imposing fixed assumptions for lower-level considerations has already been identified as an issue in 1971 [106].

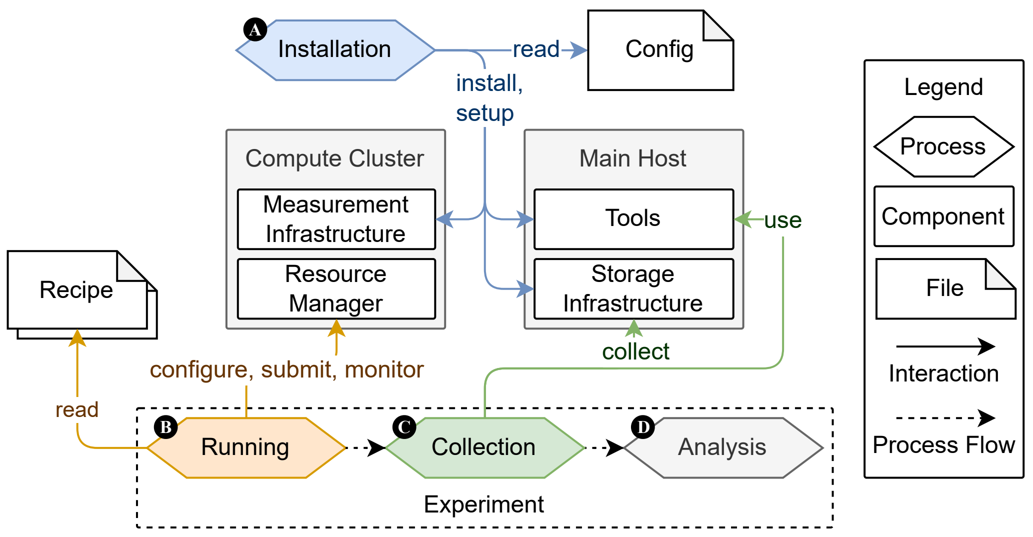


Figure 4.1: Processes of the framework and their interaction with the infrastructure components.

someone not involved in the development. With that, the framework is likely easier to use and easier to modify or extend.

4.2 Conceptual Design

The design of ShareBench-Base can be separated into two parts; a generalized process structure defining the high-level functioning of its main features (Section 4.2.1) and an infrastructure topology highlighting the needed components and their interactions (Section 4.2.2).

4.2.1 Process Structure

The design of the framework is structured into processes, each responsible for some part its functionality. A high-level overview of the processes that constitute the framework is given by Figure 4.1 and elaborated on in the following.

The *installation* process Ⓐ is responsible for setting up the tools, storage infrastructure, and measurement infrastructure required for the operation of the framework. For this, the configuration file(s) provide the necessary system information, such that the processes themselves can function independently of the specific configuration, and changes in the configuration do not require changes in the framework (**RE4.4**).

Once the installation is completed and the system is configured, experiments can be run through a series of processes. The first process (Ⓑ) is responsible for reading an

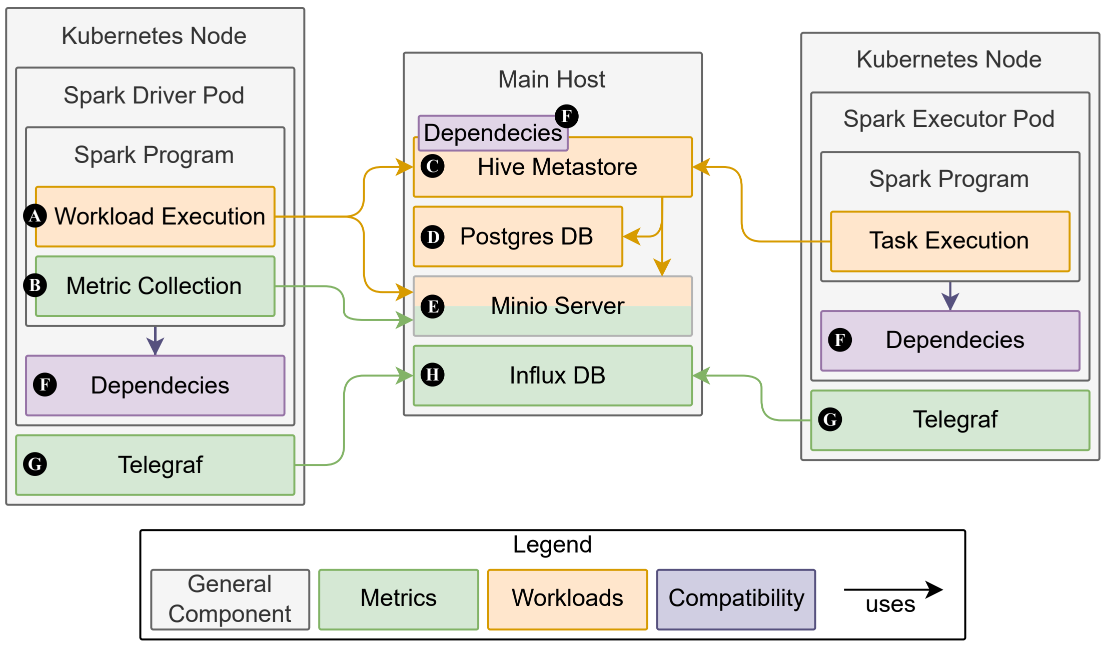


Figure 4.2: Component deployment including the Main Host and Kubernetes Nodes with various operational roles.

experiment recipe (or multiple recipes), configuring the cluster according to the recipe, submitting the work to the cluster, and monitoring the progress.

Once the work is completed, the results are collected using the installed tools (C). Finally, the *analysis* process D can be used to examine and analyze the results.

4.2.2 Infrastructure Topology

Figure 4.2 shows the topology of the infrastructure components required to support the processes outlined in Section 4.2.1 and perform the proposed experiments with SparkSQL on Kubernetes. The figure gives a snapshot of a running experiment, namely with Spark pods active in the Kubernetes nodes. All arrows in the figure indicate that a component uses the other and imply that both components are set up to function together.

The Spark program running in the driver nodes is composed of two parts; the first is responsible for executing a workload (A). SparkSQL requires external storage for both meta- and regular data. For this, the main host runs a *Hive* [107] server (C), which in turn requires a database for its operation (D), and an external storage server (E).

External storage For external storage, a cloud storage provider like *Google Cloud Storage* [108] or *Amazon S3* [109] would be valid options that require relatively little effort to set up. However, these options can not only get pricey for storing large amounts of data but also, more importantly, may reduce the reproducibility of experiments as there is little control over where and how the data is stored.

To avoid the associated cost and have more control over the external storage, a local storage service deployed as a Docker container is used instead. Here, we selected *MiniO* [110] due to its simplicity and similarity to typical cloud storage providers (specifically Amazon S3). There are alternatives to MiniO [111, 112], however, they do not offer any apparent advantages.

Access to the external storage server, both by the program and by the Hive server, is not possible out-of-the-box but requires additional Java dependencies (Ⓕ). In addition, these dependencies must be present in a specific version compatible with the remaining components.[1]

Metric collection While the infrastructure as explained so far is already sufficient to run workloads, the experiments further require the collection and storage of metrics. This is done in part by the second part of the program code (Ⓑ) which collects information such as query execution times and stores this information as a workload trace on the external storage server Ⓔ.

The first concept for collecting system performance metrics, like CPU and memory usage, was to add periodic statements in the Scala program code that, using a library, get the current system metrics and save this data to the external storage. This approach avoids having to install any additional tools directly in the machines of the cluster. However, collecting system metrics from within Scala required more effort than initially assumed, and adding extensions for more metrics would require knowledge of the program code and structure. We therefore instead decided to use *Telegraf* [113, 114], an open source tool specifically designed to collect system metrics which offers many *plug-and-play* extensions for more metrics.

Telegraf is deployed on all nodes of the cluster (Ⓖ) as a service. These services report their data to an *Influx DB* server, which is a time-series data base, running alongside the remaining storage infrastructure on the main host (Ⓗ).

[1]Yet, this version is usually not specifically declared anywhere as it is the dependency of a dependency of a dependency of a [...]. Fortunately, implementing the infrastructure gave plenty of practice for this process, as many other dependencies needed for the workload-running code were accompanied with the same issues.

4.3 Implementation for Spark SQL on Kubernetes

We built an initial prototype based on *bash* scripts for all automation processes. Bash scripts do not require any compilers or interpreters to be installed, and thereby seemed appropriate for a simple automation framework. Although mostly functional, the scripts quickly became very complex in an effort to support increasingly more features. Furthermore, discussions with colleagues showed that, while still widely popular [115], bash is poorly understood and is disliked by many. Taking this into consideration, we formulated **RE4.7** (*Use of Common Components*) and reworked the design in a new iteration. The final design consists of the following.

1. **Scala program code** that supports generation of the data set, execution of individual queries, and execution of workloads with subsequent metrics collection.

2. **Python scripts** for all automation processes.

3. A **YAML configuration file** for all configuration options needed for the installation and operation of the framework.

4. A **recipe book format** for defining experiments as YAML files.

5. Various **other components and files** needed for the operation.

4.3.1 Spark Program

While Spark itself is written in Scala, Spark applications can be written in both Scala and Python. Although Python may be easier to read and write, ShareBench-Base uses Scala, as (i) many Spark concepts and functions are closely related to Scala equivalents and are less natural to use in Python, (ii) Performance of Spark programs written in Python is typically worse [116] and could thus affect the accuracy of measurements, and (iii) support for new Spark features typically comes to Scala first before being translated to Python.

Additionally, as Scala is a compiled language, errors in the code can be identified earlier, namely, at compile time. For an environment where the code is not run locally but rather deployed remotely in, for example, a Kubernetes cluster, this can lead to significant time savings in the development process.

4.3.2 Script Structure

All scripts are written in Python and use existing Python libraries for their functionality wherever possible (**RE4.7**). The scripts are designed with a modular structure such that common operations are clearly abstracted into functions, and automation processes are compositions of these functions. Many of those functions can be used either as part of an automated process or called directly. An example of this composition is the script for running an entire experiment composed of subsequent program runs with various workloads or mechanisms. This script internally uses a function to run a single workload, but the same function can also be called directly by the user from the command line.

This modular structure, which embraces variability in the use cases of the features, requires careful consideration and additional planning in the development. It is usually easier to write many specific single-use functions than to think about reusability of code. However, with this structure, new components can be easily added (**RE4.6**), as only the function that uses the components needs to be updated, but more complex processes are still automated (**RE4.5**). Furthermore, by using abstractions with clear purposes, code readability and maintainability are typically better than for other alternatives.

Performance can also be a factor for not choosing a modular design; fewer optimizations due to more universally usable functions and additional overheads due to more function calls may have a negative impact on program performance. For ShareBench-Base, however, this is not an important factor, as script performance is not critical and the program runtime is likely to be dominated by the Spark applications.

4.3.3 Configuration

As illustrated in Section 4.2.2, many of the components need to be configured not only to work together but also to work properly on the specific system used. One example of this is the IP address of the storage servers, which depends on the IP address of the host machine, but needs to be known to all components that access the storage.

To avoid hard-coded options, which are easier to write but would not allow running the framework on various hosts without changes in many locations, all such parameters are defined in a YAML configuration file that is used by most components of the framework. YAML is favorable over other serialization languages, such as JSON or XML, for use in ShareBench-Base because it is the serialization language used by Kubernetes and therefore likely already familiar to many users.

Not all components support dynamic lookup of configuration parameters from a generalized file but instead only take configurations from a component-specific file. To use the aforementioned configuration file also to set the options for these components, their configurations are instead defined in so-called *template* files. These template files contain all the settings needed, but use placeholders for the parameters defined in the general configuration file. Once those parameters are set properly, a script takes care of filling out the template files and moving them to appropriate locations for the changes to take effect even for those components. With this approach, the configuration options of the individual components do not get limited (**RE4.2**), since the options can be easily customized, added, or removed in the template files, yet the options needed for the functionality of the infrastructure still get applied automatically (**RE4.1**, **RE4.5**).

4.3.4 Experiments

A simple approach to define automated experiments would be to write experiments directly in code, such as a program that calls the appropriate functions to run workloads and collect results. Although this method was used in an initial version of ShareBench-Base due to its simplicity, it has some disadvantages. Most notably, it hurts the readability of experiment definitions, requires a lot of redundant writing of function calls and helper code, and offers no easy way to change the process with which experiments are conducted without changing all experiment programs individually.

Instead, experiments can be specified as so-called *recipes* that define parameters such as workload, mechanism, and number of applications. Multiple recipes are then grouped into a single YAML file, called a *recipe book*. A whole recipe book can be run automatically through a script, without any user interaction beyond the starting command required, executing all experiments defined in the recipes and collecting the results. Not only is this in accordance with **RE4.5**, it also greatly facilitates running larger sets of experiments where the total duration can easily exceed multiple hours and allows easy customization and extension of the experiment process (**RE4.6**).

The format of recipe books and recipes themselves is highly versatile and allows for many different experiment types, as all parameters can be specified in a list with multiple options to easily test all possible combinations. Some example experiments include characterizing the performance of a single mechanism based on various workloads, comparing multiple mechanisms for a single workload, or exploring the performance characteristics of a mechanism for varying numbers of applications.

```
default:
  mechanisms:
    - static
    - dynamic
    - shared
  num_apps:
    - 2
    - 3
```

```
recipes:
  - workloads:
    - workload-1
    mechanisms:
    - shared
    - static
  - workloads:
    - workload-2a
    - workload-2b
    - workload-2c
    mechanisms:
    - static
    num_apps:
    - 2
  - workloads:
    - workload-3
```

Figure 4.3: Example of a recipe book with default values (left) and three recipes (right).

Table 4.2: Summary of metrics collected by the infrastructure framework.

Metric	Collected By	Usecases
Workload Trace	Spark Program	✦/☨
CPU Usage (Per Node/Global)	Telegraf	✱
Memory Usage (Per Node/Global)	Telegraf	✱
Allocation of Executors	Spark Program	☨

Usecases: ✦ overall performance | ✱ system utilization | ☨ dynamic allocation

Figure 4.3 shows an example recipe book containing three recipes. The `default` keyword can be used to define default values for all (or some) properties. If a recipe does not specify some properties, these default values are used. The `recipes` keyword defines a list of one or more recipes. Each recipe is defined as lists of values for the properties. Executing a recipe encompasses executing all possible combinations of parameters. The first recipe in the list would therefore execute `workload-1` with mechanisms `shared` and `static` each time with 2 and 3 apps (as defined in the default values).

4.3.5 Metrics

Table 4.2 gives a summary of all metrics that can currently be collected as part of the framework, including information about how they are collected and their potential usecases. Each entry is briefly explained in the following. It should be noted that this list can be easily extended with any metric that can be collected using Telegraf, since the Telegraf infrastructure is already present.

Workload traces are collected by the Spark program, logging start and completion times for every query submission. This data is sufficient to calculate the number of active queries at any time, create visualizations of the workload, and calculate metrics such as the distribution of query execution times.

CPU usage is collected by reading the `/proc/stat` file in 10-second intervals [117]. Each measurement then represents how the CPU was used since the last measurement. The usage is broken down into many fields, including idle usage, which can be used to calculate the overall utilization, but also usage by the system and usage by the user. For multicore systems, the capacity of all cores is combined. So, a situation where a single core of a quad-core system is fully used and the rest are idle would show a CPU usage of 25%.

Memory usage is collected in a similar fashion, recording not only the total memory used but also many other more detailed metrics [118]. The raw measurements consist of absolute values, but can be converted to fractions of the total memory if needed.

For collecting the allocation of executors, the Spark program is configured to log events like changes in the number of requested executors, registrations of new executors, and terminations of unused executors. This log can then be parsed automatically to plot the requested and registered executors for each application.

4.4 Evaluation

We have tested ShareBench-Base after each design iteration to identify shortcomings and areas for improvement by extensively using it to set up, conduct, and analyze the experiments discussed in Chapter 5. The framework was furthermore used for a research project by Sudnicina [119] through which we were able to uncover and subsequently address various minor issues. Table 4.1 summarizes the information about whether or not each requirement is addressed by the proposed design and implementation. The requirements that are not or only partially addressed are briefly discussed in Section 4.5.

4.5 Limitations and Future Work

We identified multiple limitations of the proposed design and implementation of ShareBench-Base, many of which could be addressed in future work.

Limited automation Most of the complex, or simply cumbersome, processes are successfully automated. However, there are still processes remaining that have to performed manually. Especially monitoring of a running workload or experiment is rudimentary and limited to whether or not the run has finished. Future work could attempt to fully address **RE4.5** by extending the automation processes and providing support for better monitoring.

Complex configuration Extensive parts of the configurable options for the framework are grouped in the generalized configuration file. The configurations for the mechanisms, however, can only be modified in one of the Python scripts, limiting the options for anyone not familiar with the code base. Future work should devise a better method of specifying and configuring resource sharing mechanisms to fully address **RE4.2**.

Missing interface for extensions Although the framework is designed with extensibility in mind, the development of extensions requires a thorough understanding of the existing implementation, as the extensions would need to be built directly into the existing code base. In future work, the extensibility aspect could be further improved by developing an interface abstraction for additional components to fully address **RE4.6**. In its current form, the infrastructure is more a proof-of-concept than a generalized and easily extensible framework. Yet, many features are either already present in a simplified form or could be added with relatively little effort.

Specialized implementation While the design of the framework is kept mostly generalized and should be applicable to compositions with other components, the implementation, naturally, is specific to Spark on Kubernetes. The Spark program is furthermore written specifically for the type of experiments considered in this work, namely experiments with Online Analytical Processing (OLAP) workloads. Further work could address these issues and try to extend the implementation to also support different Application Frameworks (AFs) and Resource Managers (RMs) for current use case or even add support for other use cases in general.

Metrics The current list of collectable metrics already enables a range of analyses. However, more metrics may be helpful or necessary for experiments that go beyond those considered in this work. One example are metrics concerned with the operation of Kubernetes and its scheduling decisions. Future work could encompass a systematic survey of existing work to determine other performance metrics used in experiments and subsequently extend the framework for those metrics.

Compatibility The implementation has been tested by multiple users; however, all tests were performed on very similar systems, giving only limited information about the compatibility with other environments.

Scalability Although the design and implementation of ShareBench-Base do not have any apparent bottlenecks to its scalability, the framework has not been tested for clusters with more than 15 nodes. With that, it is unclear whether the design and implementation would scale to larger systems.

4.6 Summary

In this chapter, we addressed **RQ2** by proposing the design and implementation of *ShareBench-Base*, an infrastructure framework for automated real-world performance analysis studies of distributed resource-sharing mechanisms and policies. We identify a topology of required components (**MC4.3**) and implement ShareBench-Base for Spark SQL on Kubernetes (**MC4.4**) but propose a generalized list of requirements (**MC4.1**) which we turn into a conceptual design (**MC4.2**) that should be applicable to other compositions.

The versatile design of ShareBench-Base allows for many uses cases, even going beyond those considered in the design process. In cases where the design or implementation is insufficient, the modular nature and use of commonly known components (such as Python) make it possible to adapt the design accordingly. Experiments can be defined in a user-friendly way as recipe books. The format of these books is intentionally designed to facilitate various types of experiments. The framework is highly automated, and consequently greatly facilitates the process of conducting experiments, allowing researchers to shift their focus more on the experiments themselves. With this automation, the framework could even be promising for use in a CI/CD pipeline.

For us, ShareBench-Base successfully enabled and facilitated conducting a wide range of experiments. The experiments and our findings are discussed in Chapter 5.

5

Performance Characterization of Resource Sharing Mechanisms of Spark on Kubernetes

Main Finding 5.1 (MF5.1):
Dynamic Partitioning leads to significant overheads from allocating and registering resources.

Main Finding 5.2 (MF5.2):
Dynamic Partitioning does not support preemption or other fairness mechanisms, leading to drastic performance differences between applications, even with near identical workloads.

Main Finding 5.3 (MF5.3):
When the concurrent load in the cluster is unbalanced, Node-Level Sharing can significantly improve performance due to better resource utilization.

Main Finding 5.4 (MF5.4):
If the concurrent load in the cluster is high, Node-Level Sharing experiences saturation leading to a degradation of performance.

Main Finding 5.5 (MF5.5):
Among the studied mechanisms, Static Partitioning has the best performance predictability, as interference between applications is minimal.

In this chapter we demonstrate the capabilities of ShareBench-Gen (Chapter 3) and ShareBench-Base (Chapter 4) by using both components to perform a series of experiments to address **RQ3**, asking *what are the performance characteristics of the resource-sharing mechanisms of Spark on Kubernetes.* The resource-sharing mechanisms and workload type

Table 5.1: Technical specifications of the experiment infrastructure.

(a) Specifications of the physical hosts.

Host	CPU	Cores	RAM	Storage
1	2x Intel Xeon Silver 4210R	20	256 GB	4TB SATA SSD
2	2x Intel Xeon Silver 4210R	20	256 GB	4TB SATA SSD, 2TB NVMe SSD

(b) Specifications of the virtual machines.

Provider	Cores	RAM	Storage
qemu	4	32 GB	48 GB

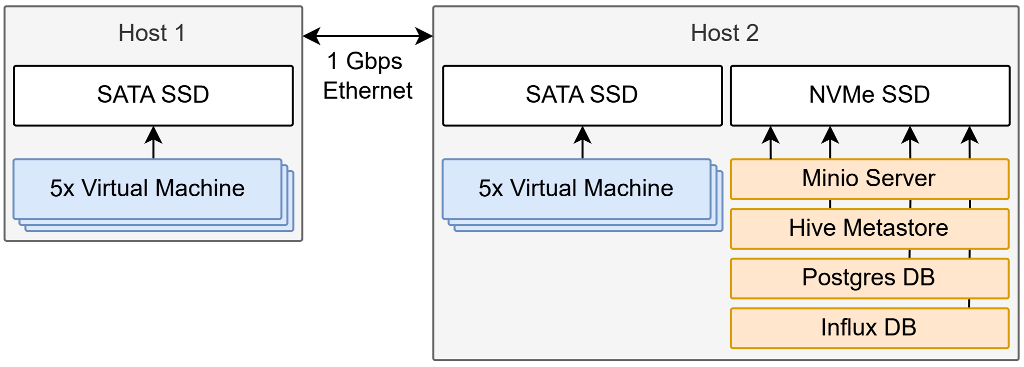


Figure 5.1: Infrastructure topology used for experiments. For each software component, the arrow indicates the used storage device.

are described in Chapter 2 (Sections 2.4 and 2.2, respectively).

5.1 Experiment Setup

We conducted all experiments with a 10-node Kubernetes cluster, emulated on two physical machines using Continuum [105]. Table 5.1a provides the hardware specifications of the physical machines, Table 5.1b the specification of the virtual machines, and Figure 5.1 the topology of the physical and virtual machines and additional services. Experiments were also performed with a larger cluster of three hosts, however, due to technical issues with the available hardware, the experiment size was reduced to that of this setup.

The exact Spark configurations used for each of the mechanisms are given in Appendix A.

5.2 Workloads

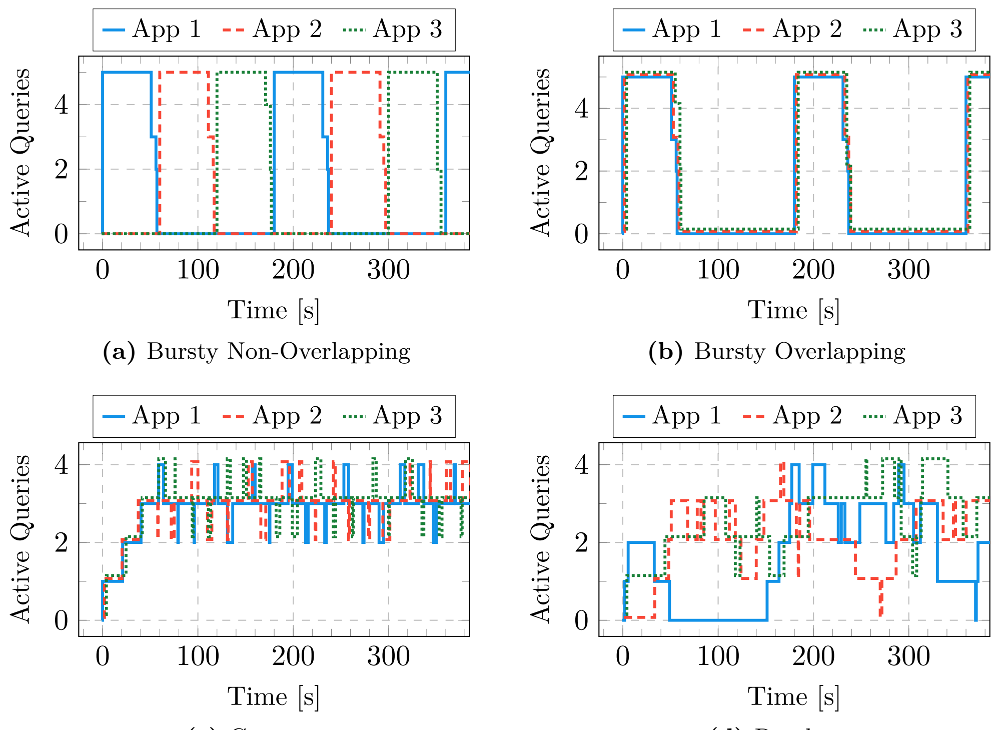
To identify common characteristics of workloads and get an understanding of relevant ranges of query duration, we analyzed workload traces from the *Snowflake Dataset* [98] based on their core characteristics. We then augmented the set of findings with more workloads to include additional characteristics that are likely to exploit the strengths and weaknesses of individual mechanisms and facilitate meaningful findings on their performance characteristics.

5.2.1 Workload Types

We selected four types of workload that exhibit considerably different characteristics. The workloads are shown in Figure 5.2 and explained below, along with hypotheses for the performance of the mechanisms for each type of workload. All workload plots follow the style introduced in Figure 2.1, with time on the x-axis and the number of active queries on the y-axis. Note that individual applications are slightly offset in some of the figures to aid in the visualization of coinciding lines. The figures furthermore only show a short excerpt of the workloads while for the experiments workloads of more than 30 minutes were used.

Bursty load In *bursty* workloads, which are commonly found in cloud computing environments [24], many queries are submitted concurrently in a so-called *burst*, with no activity in between. Figure 5.2a shows a workload with three applications, each having a burst of 5 queries every 180 seconds. Notably in this workload: the individual apps are offset so that only a single app is active at any given time. Figure 5.2b changes this by shifting the individual applications so that their bursts overlap perfectly.

Non-overlapping bursts should especially suit Node-Level Sharing, and, if there is a long enough period of inactivity between bursts, also Dynamic Partitioning. Static Partitioning will likely exhibit poor resource utilization as only one application is active at any given time and the remaining resources remain unused. Overlapping bursts should inverse the picture; Even with Static Partitioning, all resources of the cluster are utilized during the bursts of activity. Dynamic Partitioning could perform similarly to the former; however, overheads for executor allocation could negatively impact the performance. Node-Level Sharing, on the other hand, could suffer from high contention.



(a) Bursty Non-Overlapping **(b)** Bursty Overlapping

(c) Constant **(d)** Random

Figure 5.2: Types of workloads used in the experiments.

Constant load In *constant* workloads, all applications have a constant load of active queries (with a given intensity). Figure 5.2c shows an example with three apps and an average intensity of 3.

Constant workloads are likely best suited for Static Partitioning, as the load in the cluster is constantly even between applications. Dynamic Partitioning should again perform similar, as resources are likely going to be acquired at the start and never released as the load remains constant. Node-Level Sharing could again suffer from contention, depending on the overall load of the cluster.

Random load In *random* workloads, queries are distributed in a fully random fashion, resulting in a workload with highly variable characteristics. Also here, the intensity can be chosen. Figure 5.2d shows such a workload for three applications with an intensity of 2.5.[1].

If the intensity of the random workload is high enough for apps to constantly hold on to their resources even in Dynamic Partitioning, both of the partitioning mechanisms will likely perform similarly. For lower intensities, however, Dynamic Partitioning could

[1]The intensity is here, unlike with constant workloads, not directly visible from the plot. Yet, it is not an arbitrary measure, but rather the average number of queries submitted per minute and application

Table 5.2: Summary of experiment instances. The generator type is indicated in the instance name. The parameter values most significant for the experiment goal are highlighted in bold. In the specification of burst intervals, n stands for the number of apps.

Title	**Gen. Parameters**				**Goal(s)**
	AO	**I**	**QD**	**BI**	
Bursty Non-Overlapping	**60**	4	**[45, 65]**	**60**n	Compare mechanism performance at a fully unbalanced load.
Bursty Non-Overlapping with Delay	**90**	4	**[45, 65]**	**90**n	Evaluate performance of Dynamic Partitioning in ideal conditions (30 seconds of no activity between bursts).
Bursty Overlapping	**0**	4	**[45, 65]**	**60**n	(i) Compare mechanism performance at a fully balanced load, and (ii) investigate impact of interference by comparison to *Bursty Non-Overlapping*.
Constant	**0**	4	[50, 70]		Establish a baseline for evaluation of fairness with Dynamic Partitioning.
Constant with Offset	**1**	4	[50, 70]		Determine effect of offset app start times on fairness with Dynamic Partitioning.
Random Low	0	**1.5**	[30, 120]		Compare mechanism performance at low overall load.
Random Medium	0	**4**	[30, 120]		Compare mechanism performance at medium overall load.
Random High	0	**6**	[30, 120]		Compare mechanism performance at high overall load.

AO: app offset | **I**: intensity | **QD**: query duration | **BI**: burst interval

provide a performance benefit due to better resource utilization during periods of uneven load. Node-Level Sharing could achieve good performance when the load is unbalanced, but this benefit could be counteracted by performance penalties from contention when all apps are active with a high load, making random workloads an interesting point of study.

5.2.2 Experiment Instances and Goals

Table 5.2 gives a summary of the workload instances used in the experiments. The table includes the relevant generator parameters and a brief description of the experiment goal(s) for each instance. The title of each workload instance indicates the type of generator used. Generator parameters that are not relevant for the key workload characteristics are omitted for better readability. Some parameters do not apply to all generator types and are left empty for those instances.

5.2.3 Workload Calibration

For each workload, we performed a calibration of the query scale factor to match the actual and desired workload performance based on the performance with Static Partitioning, which we selected as the baseline mechanism due to its simplicity and predictable performance. This calibration is necessary due to the differences in system performance between measuring queries individually (as described in Section 3.3.3) and running multiple queries concurrently. We then use the exact same (calibrated) workload for all mechanisms under evaluation.

5.3 Findings

This section discusses the most relevant results and findings of the experiments. All boxplots are *modified* boxplots [120], points with values higher than $Q_3 + (1.5 \times IQR)$ or lower than $Q_1 - (1.5 \times IQR)$, where Q_1 and Q_3 are the first and third quartiles, respectively, and IQR is the interquartile range calculated as $Q_3 - Q_1$, are plotted as outliers such that the whiskers extend only to the minimum and maximum data values not considered outliers.

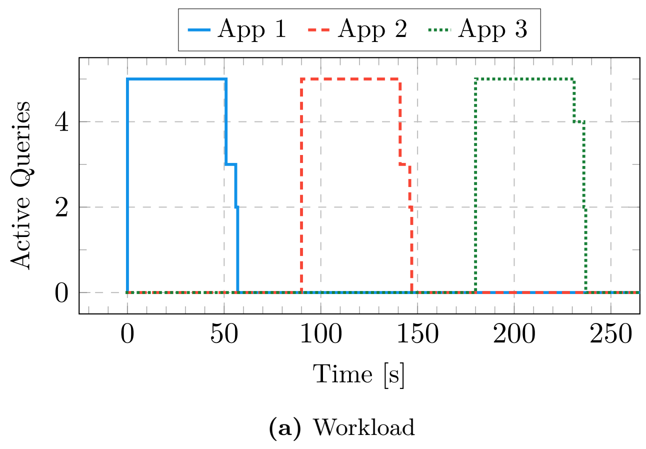
All experiments have been performed with a workload duration of at least 30 minutes and average query durations of around 60 seconds. The results exclude a warm-up period of 3 to 5 minutes, after which the performance typically stabilized for all mechanisms.[1] All values mentioned in the discussion are rounded to full seconds.

5.3.1 Dynamic Partitioning Performance Suffers from Overheads

Figure 5.3a shows a modification of the aforementioned non-overlapping bursty workload where a 30-second delay between the bursts of applications has been added. With Dynamic Partitioning configured such that applications release resources after 15 seconds of inactivity, this workload represents highly favorable conditions for the mechanism; applications have enough time to release their resources after a burst, and only one application is active at the same time such that it can acquire all available resources.

Figure 5.3b shows a comparison of the performance of all three mechanisms for a 30-minute period of this workload. Each mechanism is represented by a boxplot, which

[1]Experiments showed that performance of all mechanisms is typically much worse in the first few minutes than after that. To focus on the most meaningful results, this *warm-up* period is excluded.



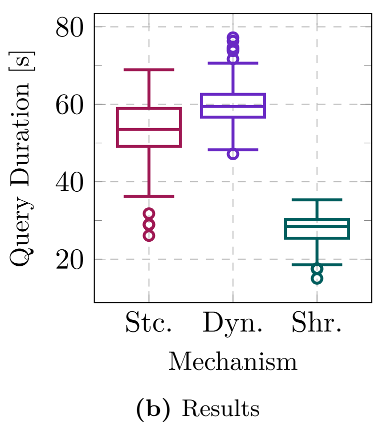


Figure 5.3: Visualization and results for a modified non-overlapping bursty workload.

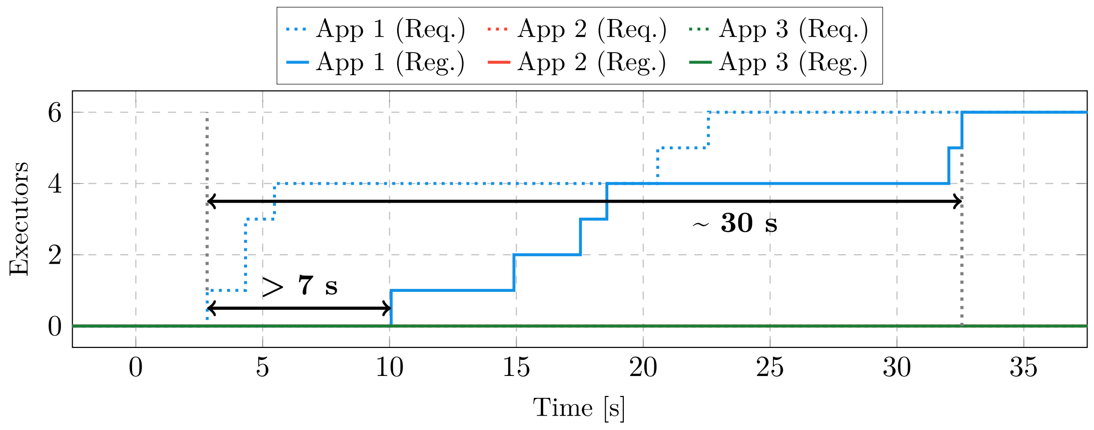
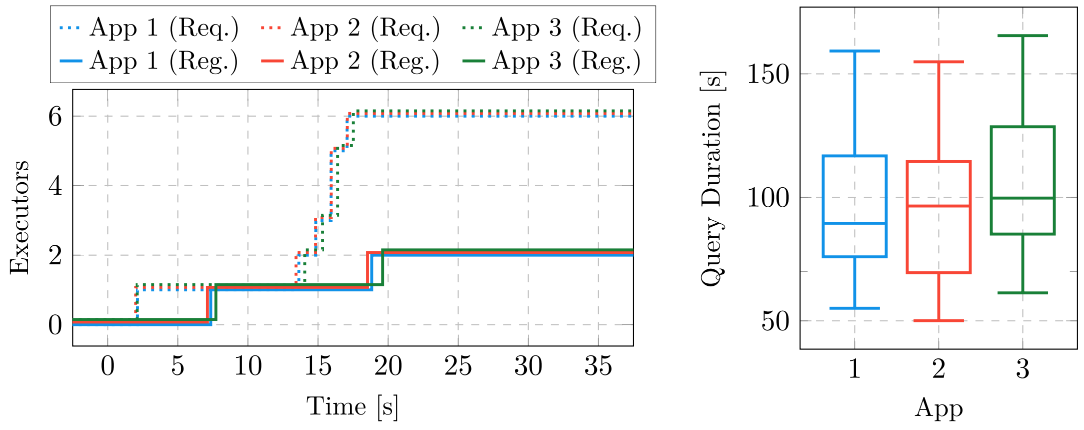


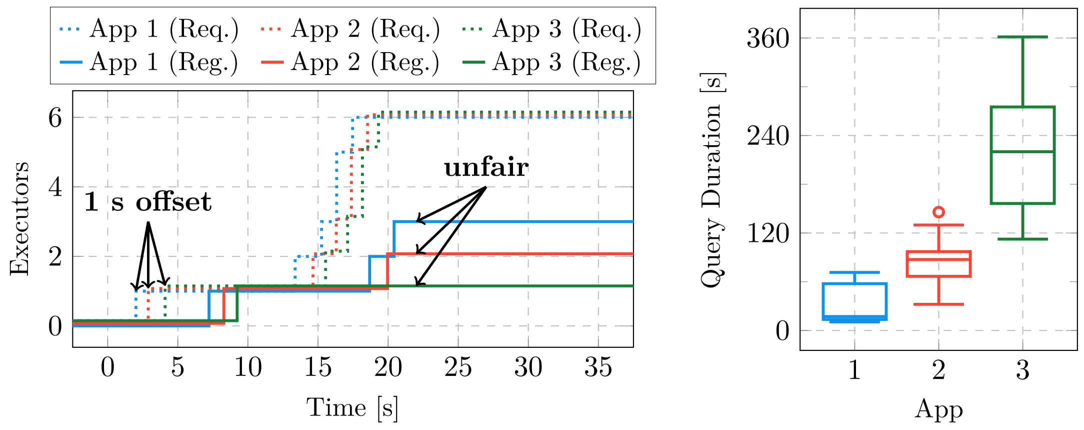
Figure 5.4: Executor requests and registrations during a burst of App 1.

shows the distribution of query durations. Surprisingly, Dynamic Partitioning performs the worst with a median query duration of 59 seconds, 6 and 31 seconds slower than Static Partitioning and Node-Level Sharing, respectively. To better understand the cause of this discrepancy between the hypothesis and result regarding the mechanism performance, it is vital to investigate the operation of the mechanism in more detail.

Figure 5.4 shows the requests and registrations of executors over time for a single application at the start of a burst. As can be seen in the figure, it takes more than 7 seconds between the request and registration of the first executor. It furthermore takes over 30 seconds before all of the six available executors are registered to the application. With that, the advantage of the mechanism that should allow a single application to acquire all available resources is overshadowed by large overheads associated with allocating and registering executors at time scales like those in question. (**MF5.1**)



(a) All apps are started at the same time.



(b) Each app is started 1 second after the previous.

Figure 5.5: Executor requests and registrations (left) and per-application results (right) for two, almost identical runs of a constant workload.

5.3.2 Dynamic Partitioning Cannot Establish Fairness

The aforementioned overheads are not the only issue with Dynamic Partitioning. In its current form, the mechanism does not support preemption or other fairness mechanisms. So once an application has acquired more resources than its theoretical fair share, it can hold onto those resources as long as it still has work, even when other applications have equal or larger amounts of outstanding work.

An exemplary manifestation of the subsequent problems is shown in Figure 5.5. We ran the same workload with a constant and equally distributed load twice, once with all applications starting concurrently and once where the second and third application were delayed by 1 and 2 seconds, respectively. As can be seen in Figure 5.5a, where the plot on the left again shows the executor requests and registrations over time, all applications acquire the same number of executors in the former scenario and thus exhibit similar performances.

However, when the applications are started with increasing offsets, the resources are unevenly distributed in favor of the first application, as shown in Figure 5.5b. Because there is no preemption, this application can hold onto its unfair share of the available resources for the duration of the workload, resulting in significant differences in performance between applications. (**MF5.2**)

The shown result is after a relatively short 5 minute workload; longer workloads further increase the differences in performance if the load is high enough, as the applications with insufficient resources keep getting increasingly overwhelmed by incoming queries, leading to long delays.

These results highlight that the current form of Dynamic Partitioning is not usable in any practical setting similar to those evaluated in this work. All subsequent results and discussions will omit Dynamic Partitioning to allow a greater focus on the remaining mechanisms.

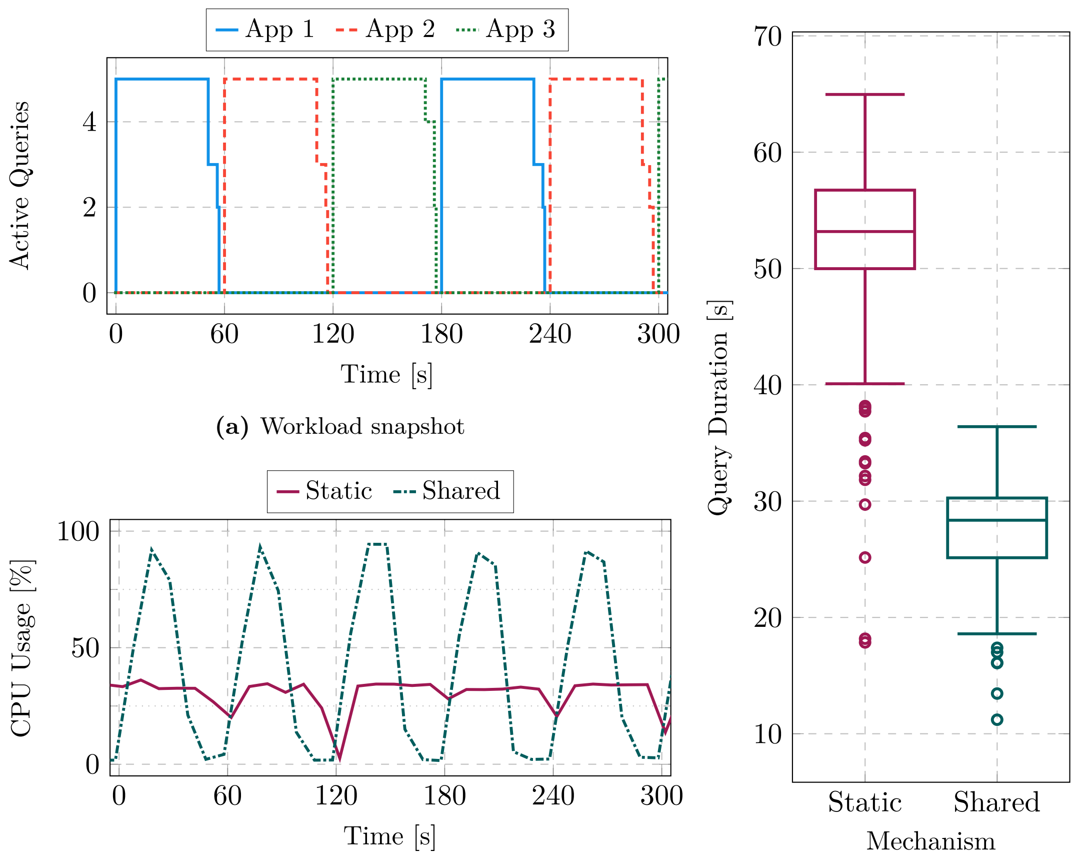
5.3.3 Node-Level Sharing can Increase Performance with Better Utilization

Theoretically, Node-Level Sharing should allow for better resource utilization when the cluster load is unbalanced as the applications under high load can make use of the resources not needed by the others. A prime example of such an unbalanced load is the non-overlapping bursty workload introduced earlier (Figure 5.2a), shown in Figure 5.6a.

The results for this workload, shown in Figure 5.6c, confirm this hypothesis; Node-Level Sharing clearly outperforms Static Partitioning with a mean query time of 27 seconds for the former, 25 seconds faster than for the latter with 52 seconds.

Figure 5.6b shows a snapshot of the CPU utilization in all executor nodes in the cluster during the workload. This data further supports the hypothesis; Node-Level Sharing shows a significantly higher peak utilization of around 90% during the bursts. With Static Partitioning, as expected, two thirds of the cluster are idle and with that the overall CPU utilization is only around 30%. (**MF5.3**)

The same is true in workloads that are not as specifically crafted to suit the mechanism. Figure 5.7a shows the results for a random workload (as shown in Figure 5.2d) of low intensity with an average of 1.5 queries per minute per application, where it can be seen that Node-Level Sharing similarly outperforms Static Partitioning with mean query times of 22 and 43 seconds, respectively.



(a) Workload snapshot

(b) Executor CPU usage snapshot

(c) Results

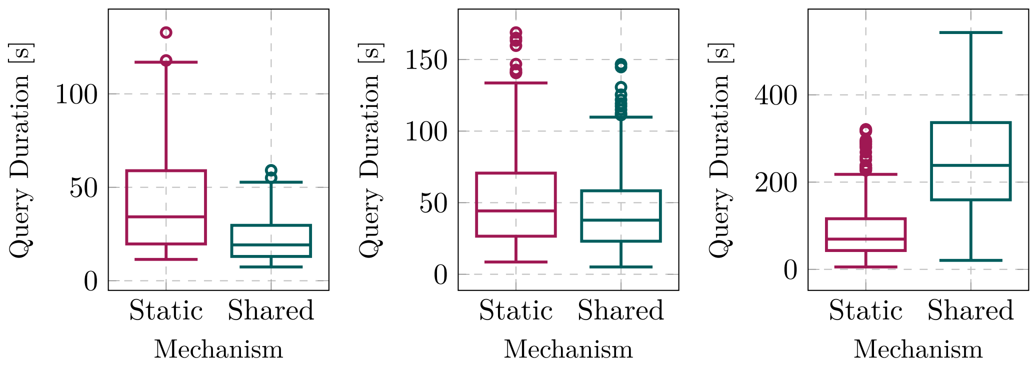
Figure 5.6: Results and executor CPU usage for the non-overlapping bursty workload.

5.3.4 Node-Level Sharing Performance can Suffer from Saturation

What is a clear advantage for Node-Level Sharing (i.e., the direct sharing of all resources) with workloads where the overall cluster load is low quickly becomes its disadvantage once the overall concurrent load in the cluster increases, as is the case for the overlapping bursty workload, shown in Figure 5.8a,

Figure 5.8c shows the result for the overlapping bursty workload, with a snapshot of the CPU utilization again shown in Figure 5.8b. Now, as all applications are active and utilizing their static partition of the resources at the same time, Static Partitioning achieves a high CPU utilization across all executor nodes. Node-Level Sharing on the other hand now experiences CPU saturation, due to all applications trying to use all nodes at the same time. With such an overly high system load, performance degrades due to contention overheads. (**MF5.4**)

This behavior continues to show for random workloads as well. When increasing the in-



(a) Low-intensity ($i = 1.5$) **(b)** Medium-intensity ($i = 4$) **(c)** High-intensity ($i = 6$)

Figure 5.7: Results for random workloads with varying intensity.

tensity from 1.5, as previously tested, to 4, the performance of both mechanisms degrades, but much more significantly for Node-Level Sharing, as can be seen in Figure 5.7b. While Static Partitioning shows an increase in outliers, likely due to periods of high load for some application, its mean query duration only degrades by 23%. For Node-Level Sharing, while still slightly better, the mean query duration is almost doubled (98% slower).[1]

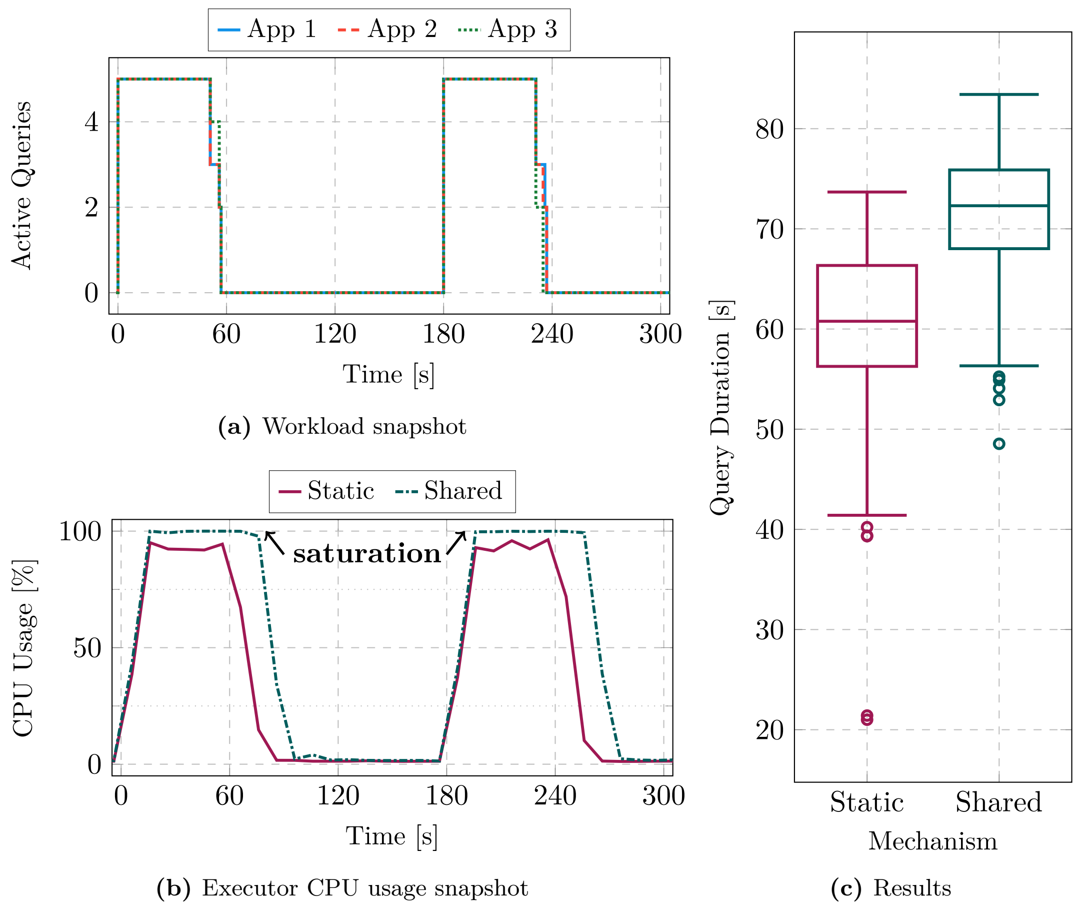
Further increasing the intensity only magnifies the same trend. Figure 5.7c shows the results of both mechanisms when increasing the intensity to 6; the overall load in the cluster is so high that the contention for the available resources greatly degrades the performance of Node-Level Sharing, now resulting in a mean query time duration of 254 seconds, almost 3 times that of Static Partitioning with 88 seconds.

5.3.5 Static Partitioning Offers the Best Performance Predictability

The workloads shown in Figures 5.6a and 5.8a are largely identical from the point of view of a single application. For both mechanisms under test, the performance is worse in the workload with overlapping bursts. This shows that both mechanisms are affected by performance unpredictability (for a single application) due to interference between applications. The cause for this interference in the case of Static Partitioning is not clear, as theoretically the applications are fully separated in the cluster. Some possible causes for the interference could be the colocation of virtual machines on the same host, use of the same storage server, or overheads from the Kubernetes control.

While mean query duration under Static Partitioning is degraded by 16%, Node-Level Sharing shows a performance penalty of over 160%. If a user would have no information

[1]For those interested: We also tested Dynamic Partitioning with the same workload, which was able to achieve a mean query duration of nothing short of respectable 330 seconds!



(a) Workload snapshot

(b) Executor CPU usage snapshot

(c) Results

Figure 5.8: Results and executor CPU usage for the overlapping bursty workload.

about the remaining users of the cluster, they could make very few assumptions about the expected performance when Node-Level Sharing is used. As shown in Section 5.3.2, the performance of a single application under Dynamic Partitioning is also highly dependent on the remaining load in the cluster. Static Partitioning, while also experiencing some interference, therefore gives the best performance stability and predictability of the three mechanisms. (**MF5.5**)

5.4 Limitations

We identified multiple limitations in the setup of the experiments that could reduce the accuracy and impact the relevance of the findings.

Technical setup All experiments were performed on a 10-node cluster, emulated on two physical machines, resulting in multiple limitations for the precision and relevance of the findings.

Firstly, the relatively small cluster size limits not only the number of concurrent applications but also the number of executors available for each. Kaufmann *et al.*, for instance, use a physical cluster of more than 5 times the size (224 cores in total) to evaluate a single Spark application [28]. However, as mentioned in Section 4.5, the scalability of ShareBench-Base has not been tested. It is thus unclear whether experiments with (much) larger clusters would be possible.

Running multiple virtual nodes on the same physical machine may have implications on performance, especially with regard to performance variability due to *noisy neighbors* [121]. Additionally, all applications used the same storage server, which again might introduce a noisy neighbor effect between multiple applications accessing the storage concurrently.

Lastly, due to technical issues, the two physical machines were only connected with a 1-Gbps Ethernet link, which showed to negatively affect the performance of executors running on the host that did not also host the storage servers, due to slow storage access.

Limited exploration of parameters The complex technical setup, numerous options for mechanism configurations, and versatile workload generator give the experiments a high-dimensional parameter space with many possible explorations. Yet, due to the relatively little time available for this work, many parameters were not explored to a great extent, but rather kept fixed at some initial value. Some examples include executor timeouts in Dynamic Partitioning, oversubscription ratios in Node-Level Sharing, or average query durations used in the workloads.

5.5 Summary

To demonstrate their practical application and refine the designs and implementations, we used ShareBench-Gen (Chapter 3) and ShareBench-Base (Chapter 4) together to evaluate three selected resource sharing mechanisms of Spark on Kubernetes at the hand of Online Analytical Processing (OLAP) workloads as can be found in data warehouse systems.

5.5.1 Performance Characteristics of the Resource Sharing Mechanisms

The experiments showed significant differences in the performance characteristics of the mechanisms; the main findings are summarized below and supplemented with actionable insights.

Dynamic Partitioning, while theoretically promising, not only leads to high overheads from allocating and registering resources (**MF5.1**), but also has no mechanism or policy for preemption of executors (**MF5.2**). Performance can consequently drastically differ between applications in the same cluster in many scenarios. **In its current form, Dynamic Partitioning is ill suited as a resource sharing mechanism and should not be considered in most cases.**

Static Partitioning and Node-Level Sharing are both viable options, but which one to favor highly depends on the expected workload characteristics. **If the overall load in the cluster is low and unbalanced, Node-Level Sharing can greatly increase the resource utilization and thereby improve the overall performance.** (**MF5.3**) However, once the load increases, the performance of Node-Level Sharing can quickly suffer due to CPU saturation and subsequent contention overheads. (**MF5.4**) **For workloads where the load is high throughout the cluster, Static Partitioning performs the best.** It is furthermore the only mechanism that offers good performance predictability without information about all applications in the cluster, as each application operates mostly isolated. (**MF5.5**)

The choice between Static Partitioning and Dynamic Partitioning therefore depends on knowledge of the expected workloads. If such knowledge does not exist, Static Partitioning is the safer choice but could compromise performance for some workloads.

5.5.2 Implications for Choice and Development of Resource Sharing Mechanisms and Policies

These results show that performance of distributed resource-sharing mechanisms and policies can be highly dependent on the characteristics of the workload. When choosing between existing mechanisms or even developing new ones, **the requirements and expected workloads should be carefully evaluated to meet performance objectives and use the available resources efficiently.**

5.5.3 Potential of the Workload Generator and Infrastructure Framework

The experiments showed that the proposed designs and implementations for the workload generator and infrastructure framework are able to facilitate a wide range of experiments and can offer a high degree of automation. A large variety of workloads can be generated, those workloads can be run individually with as little as a single command, and whole experiments with many workloads and mechanisms are equally simple to initiate and analyze.

6

Conclusion

When starting this thesis, our aim was to address the lack of knowledge about the performance characteristics of distributed resource-sharing mechanisms and policies in the composition of application framework and resource manager. We quickly realized that the apparent information deficit was due to multiple factors, which we narrowed down into problems problems **P1**–**P4**. Based on these problems, we formulated one main and three sub-research questions, the answers to which would help address the problems.

6.1 Summary of the Work

With the research questions in mind we structured the work into three main chapters, each of which contributes to answering the main research question.

6.1.1 Chapter 3 - *ShareBench-Gen*

We addressed **RQ1** (*How to design and implement a workload generator for performance analysis studies of distributed resource-sharing mechanisms and policies?*) by proposing the design and implementation of a workload generator for performance analysis studies of distributed resource-sharing mechanisms and policies.

ShareBench-Gen, is based on the data set and queries of the TPC-DS benchmark, and aimed at generating workloads for evaluating Online Analytical Processing systems, like those commonly found in many Business Intelligence environments. The design of the generator is versatile, not restricted to the specific type of data and queries, and can be applied for any workload consisting of a series of discrete work units with defined and foreseeable durations.

Although we identified multiple limitations and possible future extensions of the proposed design and implementation, we were able to successfully use ShareBench-Gen to support various experiments to characterize the performance of resource-sharing mechanisms of Spark on Kubernetes (Chapter 5).

With our design and implementation, we provide multiple contributions, summarized in the following.

MC3.1: Analysis and elicitation of requirements for a workload generator.

MC3.2: A simple, yet highly versatile conceptual design for a workload generator, applicable for any type of workload represented as discrete work units of predictable duration.

MC3.3: An implementation of the workload generator design for OLAP workloads based on the TPC-DS data set and queries.

6.1.2 Chapter 4 - *ShareBench-Base*

To address **RQ2** (*How to design and implement an infrastructure framework for automated real-world performance analysis studies of distributed resource-sharing mechanisms and policies?*) we followed a similar approach by proposing the design and implementation of *ShareBench-Base*, an infrastructure framework for automated real-world performance analysis studies of distributed resource-sharing mechanisms and policies. The proposed conceptual design describes the general architecture of the framework, likely applicable to experiments with various system compositions; the implementation is a realization of that architecture for Spark SQL on Kubernetes.

ShareBench-Base supports a wide range of automated experiments through a versatile experiment recipe format and collection options for numerous metrics. Its modular nature and the use of common components like Python and YAML facilitate extension of the implementation for needs going beyond the current capabilities.

However, the extensibility could be greatly improved by designing an interface abstraction for *plug-in* extensions, and also the automation processes could be greatly extended. Other limitations include the complex nature of some configuration options and limited information about compatibility of the implementation with host systems different to those tested.

We developed ShareBench-Base alongside the experiments conducted to answer **RQ3**. With this process, we were not only able to iteratively improve the design and implementation based on our first-hand experience but also got a direct impression of the time savings enabled through the automation of the framework. In its final version, the infrastructure framework reduced the time we spent performing experiments from hours to tens of minutes per experiment.

Our design and implementation provide multiple contributions, summarized in the following.

MC4.1: Analysis and elicitation of requirements for an infrastructure framework for automated real-world performance analysis studies.

MC4.2: A generalized, process-based conceptual design for an infrastructure framework for automated real-world performance analysis studies.

MC4.3: A structural topology of required components for performance analysis studies of Spark SQL on Kubernetes.

MC4.4: An implementation of the infrastructure framework design for Spark SQL on Kubernetes.

6.1.3 Chapter 5 - Performance Characterization

Finally, we addressed **RQ3** (*What are the performance characteristics of the resource-sharing mechanisms of Spark on Kubernetes?*) by using ShareBench-Gen and ShareBench-Base together to perform a series of performance analysis experiments aimed at characterizing research-sharing mechanisms of Spark on Kubernetes. We were able to utilize both the infrastructure and the workload generator for the experiments and use the experience to improve their design and implementation through many iterations.

The experiments enabled us to thoroughly evaluate the resource-sharing mechanisms based on various workload characteristics. Through the analysis, we came to numerous conclusions. The main findings are summarized below; a more detailed explanation and actionable insights for the choice of existing or development of new mechanisms can be found in Chapter 5. Due to limitations of the available hardware and the relatively little time available for this work, the findings may be limited in accuracy and relevance for larger systems.

MF5.1: Dynamic Partitioning leads to significant overheads from allocating and registering resources.

MF5.2: Dynamic Partitioning does not support preemption or other fairness mechanisms, leading to drastic performance differences between applications, even with near identical workloads.

MF5.3: When the concurrent load in the cluster is unbalanced, Node-Level Sharing can significantly improve performance due to better resource utilization.

MF5.4: If the concurrent load in the cluster is high, Node-Level Sharing experiences saturation leading to a degradation of performance.

MF5.5: Among the studied mechanisms, Static Partitioning has the best performance predictability, as interference between applications is minimal.

6.2 Summary and Future Work

With the explosive increases in global data production [7, 8] and the need for extensive analysis to derive value from this data [9, 10], data centers are highly relevant for the functioning of many areas of modern society [1–5, 10, 43, 49–52]. Yet, energy consumption and carbon footprint of the, often warehouse-sized, computing facilities is a prevalent issue [53–55]. Increases in efficiency are needed to support the growing demand for computation, especially considering the limits of available energy, which are unlikely to improve significantly in the near future [11].

With the continuous shift towards (*hyperscale*) cloud data centers [47], resource-sharing mechanisms can have a significant impact as they can improve the utilization of available resources and thus increase efficiency. Our experiments showed that the performance of various mechanisms can differ significantly. However, performance highly depends on the characteristics of the workload, implying that informed decisions for the choice configuration, or development of resource-sharing mechanisms and policies could significantly improve overall performance if the assumptions about the expected workloads are accurate.

Our work aims to facilitate performance analysis studies of distributed resource-sharing mechanisms and policies by proposing a systematic approach based on a workload generator and an infrastructure framework. We further characterize three resource-sharing mechanisms of Spark on Kubernetes. For future research, we identify three main directions in which to extend this work. (1) Further experiments using the existing ShareBench-Gen and ShareBench-Base implementations, with more extensive exploration of the possible parameters for new findings, or a larger experiment cluster to validate or correct the findings of this work for larger systems. (2) Extensions to ShareBench-Gen to improve existing workloads, generate new types of workloads, or refine usability with more features and better abstractions. (3) Extensions to ShareBench-Base that expand the set of possible experiments and introduce more metrics to further extend the capabilities of the framework, add support for other Application Frameworks or Resource Mangers, or extend the automation processes to further improve usability and time savings.

The source code and all experiment data produced as part of this work are publicly available on GitHub[1].

[1]https://github.com/atlarge-research/ShareBench

References

[1] Professional Engineering. “The striking engineering inside the euro 2024 ball”. (7 Oct. 2024), [Online]. Available: `https://www.imeche.org/news/news-article/the-striking-engineering-inside-the-euro-2024-ball` (visited on 18 Jul. 2024).

[2] V. J. García-Morales, A. Garrido-Moreno and R. Martín-Rojas, “The transformation of higher education after the COVID disruption: Emerging challenges in an online learning scenario”, *Frontiers in Psychology*, vol. 12, p. 616 059, 11 Feb. 2021, ISSN: 1664-1078. DOI: `10.3389/fpsyg.2021.616059`. [Online]. Available: `https://www.frontiersin.org/articles/10.3389/fpsyg.2021.616059/full`.

[3] M. M. Hasan, J. Popp and J. Oláh, “Current landscape and influence of big data on finance”, *Journal of Big Data*, vol. 7, no. 1, p. 21, Dec. 2020, ISSN: 2196-1115. DOI: `10.1186/s40537-020-00291-z`. [Online]. Available: `https://journalofbigdata.springeropen.com/articles/10.1186/s40537-020-00291-z`.

[4] S. Shilo, H. Rossman and E. Segal, “Axes of a revolution: Challenges and promises of big data in healthcare”, *Nature Medicine*, vol. 26, no. 1, pp. 29–38, Jan. 2020, ISSN: 1078-8956, 1546-170X. DOI: `10.1038/s41591-019-0727-5`. [Online]. Available: `https://www.nature.com/articles/s41591-019-0727-5`.

[5] Progressive Railroading. “On a high-tech trek: Norfolk southern notes progress in its quest to become a ’technology-enabled railroad of the future’”. (Aug. 2018), [Online]. Available: `https://www.progressiverailroading.com/norfolk_southern/article/On-a-high-tech-trek-Norfolk-Southern-notes-progress-in-its-quest-to-become-a-technology-enabled-railroad-of-the-future--55310` (visited on 18 Jul. 2024).

[6] R. F. Jørgensen, Ed., *Human rights in the age of platforms*, Information policy series, Cambridge, MA: The MIT Press, 2019, 342 pp., ISBN: 978-0-262-03905-5.

[7] Statista, “Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025”, 2021. [Online]. Available: `https://www.statista.com/statistics/871513/worldwide-data-created/` (visited on 17 Jul. 2024).

[8] J. Rydning, "Worldwide IDC global DataSphere forecast, 2023-2027: It's a distributed, diverse, and dynamic (3d) DataSphere", International Data Corporation (IDC), US50554523, Apr. 2023. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=US50554523.

[9] J. Fan, F. Han and H. Liu, "Challenges of big data analysis", *National Science Review*, vol. 1, no. 2, pp. 293–314, 1 Jun. 2014, ISSN: 2053-714X, 2095-5138. DOI: 10.1093/nsr/nwt032. [Online]. Available: https://academic.oup.com/nsr/article/1/2/293/1397586.

[10] T. H. Davenport and J. Dyché, "Big data in big companies", *International Institute for Analytics*, vol. 3, no. 1, 2013.

[11] R. S. Williams, "What's next? [the end of moore's law]", *Computing in Science & Engineering*, vol. 19, no. 2, pp. 7–13, Mar. 2017, ISSN: 1521-9615. DOI: 10.1109/MCSE.2017.31. [Online]. Available: http://ieeexplore.ieee.org/document/7878940/.

[12] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology", *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, Mar. 2017, ISSN: 1521-9615. DOI: 10.1109/MCSE.2017.29. [Online]. Available: http://ieeexplore.ieee.org/document/7878935/.

[13] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", in *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[14] The Apache Software Foundation., *Apache spark*. [Online]. Available: https://spark.apache.org/ (visited on 27 May 2024).

[15] D. Jeffrey and S. Ghemawat, "MapReduce: A flexible data processing tool", *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[16] The Apache Software Foundation., *Apache flink*. [Online]. Available: https://flink.apache.org/ (visited on 27 May 2024).

[17] Anyscale, Inc., *Ray*. [Online]. Available: https://www.ray.io/ (visited on 27 May 2024).

[18] C. Jin, X. Bai, C. Yang, W. Mao and X. Xu, "A review of power consumption models of servers in data centers", *Applied Energy*, vol. 265, p. 114 806, May 2020, ISSN: 03062619. DOI: 10.1016/j.apenergy.2020.114806. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0306261920303184.

[19] Kubernetes, *Kubernetes*. [Online]. Available: https://kubernetes.io/ (visited on 18 May 2024).

[20] V. K. Vavilapalli *et al.*, "Apache hadoop YARN: Yet another resource negotiator", in *Proceedings of the 4th annual Symposium on Cloud Computing*, Santa Clara California: ACM, Oct. 2013, pp. 1–16, ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. [Online]. Available: https://dl.acm.org/doi/10.1145/2523616.2523633.

[21] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center", in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[22] Y. Chen, A. Ganapathi, R. Griffith and R. Katz, "The case for evaluating MapReduce performance using workload suites", in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, Singapore, Singapore: IEEE, Jul. 2011, pp. 390–399, ISBN: 978-1-4577-0468-0. DOI: 10.1109/MASCOTS.2011.12. [Online]. Available: http://ieeexplore.ieee.org/document/6005383/.

[23] D. Cheng, X. Zhou, P. Lama, J. Wu and C. Jiang, "Cross-platform resource scheduling for spark and MapReduce on YARN", *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1341–1353, 1 Aug. 2017, ISSN: 0018-9340. DOI: 10.1109/TC.2017.2669964. [Online]. Available: http://ieeexplore.ieee.org/document/7857034/.

[24] J. Tai, J. Zhang, J. Li, W. Meleis and N. Mi, "ArA: Adaptive resource allocation for cloud computing environments under bursty workloads", in *30th IEEE International Performance Computing and Communications Conference*, Orlando, FL, USA: IEEE, Nov. 2011, pp. 1–8. DOI: 10.1109/PCCC.2011.6108060. [Online]. Available: http://ieeexplore.ieee.org/document/6108060/.

[25] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox and D. Patterson, "Rain: A workload generation toolkit for cloud computing applications", 2010.

[26] J. Khamse-Ashari, I. Lambadaris, G. Kesidis, B. Urgaonkar and Y. Zhao, "An efficient and fair multi-resource allocation mechanism for heterogeneous servers", *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2686–2699, 1 Dec. 2018, ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2018.2841915. [Online]. Available: https://ieeexplore.ieee.org/document/8368291/.

[27] A. Beltre, P. Saha and M. Govindaraju, "KubeSphere: An approach to multi-tenant fair scheduling for kubernetes clusters", in *2019 IEEE Cloud Summit*, Washington, DC, USA: IEEE, Aug. 2019, pp. 14–20, ISBN: 978-1-72813-101-6. DOI: 10.1109/CloudSummit47114.2019.00009. [Online]. Available: https://ieeexplore.ieee.org/document/9045748/.

[28] M. Kaufmann, K. Kourtis, A. Schuepbach and M. Zitterbart, "Mira: Sharing resources for distributed analytics at small timescales", in *2018 IEEE International Conference on Big Data (Big Data)*, Seattle, WA, USA: IEEE, Dec. 2018, pp. 231–241, ISBN: 978-1-5386-5035-6. DOI: `10.1109/BigData.2018.8622363`. [Online]. Available: `https://ieeexplore.ieee.org/document/8622363/`.

[29] C. Zhu, B. Han and Y. Zhao, "A comparative study of spark on the bare metal and kubernetes", in *2020 6th International Conference on Big Data and Information Analytics (BigDIA)*, Shenzhen, China: IEEE, Dec. 2020, pp. 117–124, ISBN: 978-1-66542-232-1. DOI: `10.1109/BigDIA51454.2020.00027`. [Online]. Available: `https://ieeexplore.ieee.org/document/9384578/`.

[30] A. Iosup *et al.*, "The AtLarge vision on the design of distributed systems and ecosystems", in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, Dallas, TX, USA: IEEE, Jul. 2019, pp. 1765–1776, ISBN: 978-1-72812-519-0. DOI: `10.1109/ICDCS.2019.00175`. [Online]. Available: `https://ieeexplore.ieee.org/document/8885212/`.

[31] I. Sommerville, *Software Engineering* (Always learning), Tenth edition, global edition. Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo: Pearson, 2016, 810 pp., ISBN: 978-1-292-09613-1.

[32] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice* (SEI Series in Software Engineering). Pearson Education, 2021, ISBN: 978-0-13-688602-0.

[33] A. Iosup *et al.*, "The grid workloads archive", *Future Generation Computer Systems*, vol. 24, no. 7, pp. 672–686, Jul. 2008, ISSN: 0167739X. DOI: `10.1016/j.future.2008.02.003`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0167739X08000125`.

[34] Y. Guo, A. L. Varbanescu, D. Epema and A. Iosup, "Design and experimental evaluation of distributed heterogeneous graph-processing systems", in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, Colombia: IEEE, May 2016, pp. 203–212, ISBN: 978-1-5090-2453-7. DOI: `10.1109/CCGrid.2016.53`. [Online]. Available: `http://ieeexplore.ieee.org/document/7515690/`.

[35] A. Manterola Lasa, S. Talluri, T. De Matteis and A. Iosup, "The cost of simplicity: Understanding datacenter scheduler programming abstractions", in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, London United Kingdom: ACM, 7 May 2024, pp. 166–177, ISBN: 9798400704444. DOI:

10.1145/3629526.3645038. [Online]. Available: https://dl.acm.org/doi/10.1145/3629526.3645038.

[36] E. Van Eyk *et al.*, "The SPEC-RG reference architecture for FaaS: From microservices and containers to serverless platforms", *IEEE Internet Computing*, vol. 23, no. 6, pp. 7–18, 1 Nov. 2019, ISSN: 1089-7801, 1941-0131. DOI: 10.1109/MIC.2019.2952061. [Online]. Available: https://ieeexplore.ieee.org/document/8894540/.

[37] G. Andreadis, L. Versluis, F. Mastenbroek and A. Iosup, "A reference architecture for datacenter scheduling: Design, validation, and experiments", in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA: IEEE, Nov. 2018, pp. 478–492, ISBN: 978-1-5386-8384-2. DOI: 10.1109/SC.2018.00040. [Online]. Available: https://ieeexplore.ieee.org/document/8665816/.

[38] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* 1991, ISBN: 978-0-471-50336-1.

[39] J. Ousterhout, "Always measure one level deeper", *Communications of the ACM*, vol. 61, no. 7, pp. 74–83, 25 Jun. 2018, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3213770. [Online]. Available: https://dl.acm.org/doi/10.1145/3213770.

[40] G. Heiser. "Systems benchmarking crimes". (), [Online]. Available: https://gernot-heiser.org/benchmarking-crimes.html.

[41] E. D. Berger, S. M. Blackburn, M. Hauswirth and M. W. Hicks. "A checklist manifesto for empirical evaluation: A preemptive strike against a replication crisis in computer science", PL Perspectives. (28 Aug. 2019), [Online]. Available: https://blog.sigplan.org/2019/08/28/a-checklist-manifesto-for-empirical-evaluation-a-preemptive-strike-against-a-replication-crisis-in-computer-science/ (visited on 4 Jul. 2024).

[42] S. Bezjak *et al.*, *Open science training handbook.* Zenodo, Apr. 2018. DOI: 10.5281/zenodo.1212496. [Online]. Available: https://doi.org/10.5281/zenodo.1212496.

[43] A. Iosup *et al.*, *Future computer systems and networking research in the netherlands: A manifesto*, 2022. arXiv: 2206.03259[cs.CY].

[44] B. Whitehead, D. Andrews, A. Shah and G. Maidment, "Assessing the environmental impact of data centres part 1: Background, energy use and metrics", *Building and Environment*, vol. 82, pp. 151–159, Dec. 2014, ISSN: 03601323. DOI: 10.1016/j.buildenv.2014.08.021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S036013231400273X.

[45] S. Moss. "In search of the world's largest data center". (18 May 2022), [Online]. Available: https://www.datacenterdynamics.com/en/analysis/in-search-of-the-worlds-largest-data-center/ (visited on 14 Jul. 2024).

[46] E. Masanet, A. Shehabi, N. Lei, S. Smith and J. Koomey, "Recalibrating global data center energy-use estimates", *Science*, vol. 367, no. 6481, pp. 984–986, 28 Feb. 2020, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aba3758. [Online]. Available: https://www.science.org/doi/10.1126/science.aba3758.

[47] A. Shehabi, S. J. Smith, E. Masanet and J. Koomey, "Data center growth in the united states: Decoupling the demand for services from electricity use", *Environmental Research Letters*, vol. 13, no. 12, p. 124 030, 18 Dec. 2018, ISSN: 1748-9326. DOI: 10.1088/1748-9326/aaec9c. [Online]. Available: https://iopscience.iop.org/article/10.1088/1748-9326/aaec9c.

[48] A. S. Fleischer, "Cooling our insatiable demand for data", *Science*, vol. 370, no. 6518, pp. 783–784, 13 Nov. 2020, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.abe5318. [Online]. Available: https://www.science.org/doi/10.1126/science.abe5318.

[49] S. V. Patankar, "Airflow and cooling in a data center", *Journal of Heat Transfer*, vol. 132, no. 7, p. 073 001, 1 Jul. 2010, ISSN: 0022-1481, 1528-8943. DOI: 10.1115/1.4000703. [Online]. Available: https://asmedigitalcollection.asme.org/heattransfer/article/doi/10.1115/1.4000703/451420/Airflow-and-Cooling-in-a-Data-Center.

[50] M. v. Steen and A. S. Tanenbaum, *Distributed Systems*, Fourth edition, version 4.01 (January 2023). Erscheinungsort nicht ermittelbar: Maarten van Steen, 2023, 669 pp., ISBN: 978-90-815406-3-6.

[51] B. G. Lindsay, "Jim gray at IBM: The transaction processing revolution", *ACM SIGMOD Record*, vol. 37, no. 2, pp. 38–40, Jun. 2008, ISSN: 0163-5808. DOI: 10.1145/1379387.1379401. [Online]. Available: https://dl.acm.org/doi/10.1145/1379387.1379401.

[52] D. Castro, P. Kothuri, P. Mrowczynski, D. Piparo and E. Tejedor, "Apache spark usage and deployment models for scientific computing", *EPJ Web of Conferences*, vol. 214, A. Forti, L. Betev, M. Litmaath, O. Smirnova and P. Hristov, Eds., p. 07 020, 2019, ISSN: 2100-014X. DOI: 10.1051/epjconf/201921407020. [Online]. Available: https://www.epj-conferences.org/10.1051/epjconf/201921407020.

[53] IEA, "Electricity 2024", IEA, Paris, 2024. [Online]. Available: https://www.iea.org/reports/electricity-2024.

[54] J. Calma. "Google's carbon footprint balloons in its gemini AI era", The Verge. (2 Jul. 2024), [Online]. Available: `https://www.theverge.com/2024/7/2/24190874/google-ai-climate-change-carbon-emissions-rise` (visited on 8 Jul. 2024).

[55] J. Calma. "This climate tech startup wants to capture carbon and help data centers cool down", The Verge. (11 Jul. 2024), [Online]. Available: `https://www.theverge.com/2024/7/11/24195989/climate-change-carbon-removal-startup-280-earth-google` (visited on 14 Jul. 2024).

[56] A. Bog, *Benchmarking Transaction and Analytical Processing Systems: The Creation of a Mixed Workload Benchmark and its Application* (In-Memory Data Management Research). Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. DOI: `10.1007/978-3-642-38070-9`. [Online]. Available: `https://link.springer.com/10.1007/978-3-642-38070-9`.

[57] S. Melnik *et al.*, "Dremel: Interactive analysis of web-scale datasets", *Communications of The Acm*, vol. 54, no. 6, pp. 114–123, 2011. DOI: `10.1145/1953122.1953148`. [Online]. Available: `https://doi.org/10.1145/1953122.1953148`.

[58] S. Melnik *et al.*, "Dremel: A decade of interactive SQL analysis at web scale", *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3461–3472, 2020. DOI: `10.14778/3415478.3415568`. [Online]. Available: `http://www.vldb.org/pvldb/vol13/p3461-melnik.pdf`.

[59] E. Codd, S. Codd and C. Salley, *Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate.* Codd & Associates, 1993.

[60] Google Cloud, *Big query.* [Online]. Available: `https://cloud.google.com/bigquery` (visited on 14 Jul. 2024).

[61] Microsoft, *SQL server analysis services overview.* [Online]. Available: `https://learn.microsoft.com/en-us/analysis-services/ssas-overview?view=asallproducts-allversions` (visited on 19 Jul. 2024).

[62] Oracle, *Getting started with oracle OLAP.* [Online]. Available: `https://docs.oracle.com/en/database/oracle/oracle-database/19/olaug/getting-started-oracle-olap.html` (visited on 19 Jul. 2024).

[63] J. Arnold, B. Glavic and I. Raicu, "A high-performance distributed relational database system for scalable OLAP processing", in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rio de Janeiro, Brazil: IEEE, May 2019, pp. 738–748, ISBN: 978-1-72811-246-6. DOI: `10.1109/IPDPS.2019.00083`. [Online]. Available: `https://ieeexplore.ieee.org/document/8820952/`.

[64] The Apache Software Foundation., *Apache hadoop.* [Online]. Available: `https://hadoop.apache.org/` (visited on 14 Jul. 2024).

[65] M. Armbrust *et al.*, “Spark SQL: Relational data processing in spark”, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne Victoria Australia: ACM, 27 May 2015, pp. 1383–1394, ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797. [Online]. Available: https://dl.acm.org/doi/10.1145/2723372.2742797.

[66] G. Cheng, S. Ying, B. Wang and Y. Li, “Efficient performance prediction for apache spark”, *Journal of Parallel and Distributed Computing*, vol. 149, pp. 40–51, Mar. 2021, ISSN: 07437315. DOI: 10.1016/j.jpdc.2020.10.010. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0743731520303993.

[67] GitHub, *apache/spark*. [Online]. Available: https://github.com/apache/spark (visited on 18 Jul. 2024).

[68] M. Li, J. Tan, Y. Wang, L. Zhang and V. Salapura, “SparkBench: A comprehensive benchmarking suite for in memory data analytic platform spark”, in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, Ischia Italy: ACM, 6 May 2015, pp. 1–8, ISBN: 978-1-4503-3358-0. DOI: 10.1145/2742854.2747283. [Online]. Available: https://dl.acm.org/doi/10.1145/2742854.2747283.

[69] D. Harris. “Survey shows huge popularity spike for apache spark”, Fortune. (2015), [Online]. Available: https://fortune.com/2015/09/25/apache-spark-survey/ (visited on 18 Jul. 2024).

[70] Apache Spark, *Spark standalone mode*. [Online]. Available: https://spark.apache.org/docs/latest/spark-standalone.html (visited on 18 May 2024).

[71] Apache Spark, *Running spark on mesos*. [Online]. Available: https://spark.apache.org/docs/latest/running-on-mesos.html (visited on 19 May 2024).

[72] P. S. Janardhanan and P. Samuel, “Launch overheads of spark applications on standalone and hadoop YARN clusters”, in *Advances in Electrical and Computer Technologies*, T. Sengodan, M. Murugappan and S. Misra, Eds., vol. 672, Series Title: Lecture Notes in Electrical Engineering, Singapore: Springer Singapore, 2020, pp. 47–54, ISBN: 978-981-15-5558-9. DOI: 10.1007/978-981-15-5558-9_5. [Online]. Available: http://link.springer.com/10.1007/978-981-15-5558-9_5.

[73] A. Raju, R. Ramanathan and R. Hemavathy, “A comparative study of spark schedulers’ performance”, in *2019 4th International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS)*, Bengaluru, India: IEEE, Dec. 2019, pp. 1–5, ISBN: 978-1-72812-619-7. DOI: 10.1109/CSITSS47250.2019.9031028. [Online]. Available: https://ieeexplore.ieee.org/document/9031028/.

[74] Amazon AWS, *Amazon elastic kubernetes service (EKS).* [Online]. Available: https://aws.amazon.com/eks/?nc2=type_a (visited on 19 May 2024).

[75] Google Cloud, *Google kubernetes engine (GKE).* [Online]. Available: https://cloud.google.com/kubernetes-engine/?hl=en (visited on 19 May 2024).

[76] Microsoft, *Azure kubernetes service (AKS).* [Online]. Available: https://azure.microsoft.com/en-us/products/kubernetes-service/ (visited on 19 May 2024).

[77] S. Agarwal, X. Li, R. Xin and J. Damji. "Introducing apache spark 2.3". (2018), [Online]. Available: https://www.databricks.com/blog/2018/02/28/introducing-apache-spark-2-3.html (visited on 18 May 2024).

[78] P. Saha, A. Beltre and M. Govindaraju, "Exploring the fairness and resource distribution in an apache mesos environment", in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, San Francisco, CA, USA: IEEE, Jul. 2018, pp. 434–441, ISBN: 978-1-5386-7235-8. DOI: 10.1109/CLOUD.2018.00061. [Online]. Available: https://ieeexplore.ieee.org/document/8457829/.

[79] Apache Spark, *Dynamic resource allocation.* [Online]. Available: https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation (visited on 21 May 2024).

[80] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration", *Sensors*, vol. 20, no. 16, p. 4621, 17 Aug. 2020, ISSN: 1424-8220. DOI: 10.3390/s20164621. [Online]. Available: https://www.mdpi.com/1424-8220/20/16/4621 (visited on 18 Jul. 2024).

[81] K. Rzadca *et al.*, "Autopilot: Workload autoscaling at google", in *Proceedings of the Fifteenth European Conference on Computer Systems*, Heraklion Greece: ACM, 15 Apr. 2020, pp. 1–16, ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3387524. [Online]. Available: https://dl.acm.org/doi/10.1145/3342195.3387524.

[82] S. A. Baset, L. Wang and C. Tang, "Towards an understanding of oversubscription in cloud", in *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, San Jose, CA: USENIX Association, Apr. 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/baset.

[83] O.-C. Marcu, A. Costan, G. Antoniu and M. S. Perez-Hernandez, "Spark versus flink: Understanding performance in big data analytics frameworks", in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Taipei, Taiwan: IEEE, Sep. 2016, pp. 433–442, ISBN: 978-1-5090-3653-0. DOI: 10.1109/CLUSTER.2016.22. [Online]. Available: http://ieeexplore.ieee.org/document/7776539/.

[84] N. Ahmed, A. L. C. Barczak, T. Susnjak and M. A. Rashid, "A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using HiBench", *Journal of Big Data*, vol. 7, no. 1, p. 110, Dec. 2020, ISSN: 2196-1115. DOI: 10.1186/s40537-020-00388-5. [Online]. Available: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-020-00388-5.

[85] M. Pöss, B. Smith, L. Kollár and P.-Å. Larson, "TPC-DS, taking decision support benchmarking to the next level", in *Proceedings of the 2002 ACM SIGMOD international conference on management of data, madison, wisconsin, USA, june 3-6, 2002*, M. J. Franklin, B. Moon and A. Ailamaki, Eds., ACM, 2002, pp. 582–587. DOI: 10.1145/564691.564759. [Online]. Available: https://doi.org/10.1145/564691.564759.

[86] L. Wang *et al.*, "BigDataBench: A big data benchmark suite from internet services", in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Orlando, FL, USA: IEEE, Feb. 2014, pp. 488–499, ISBN: 978-1-4799-3097-5. DOI: 10.1109/HPCA.2014.6835958. [Online]. Available: http://ieeexplore.ieee.org/document/6835958/.

[87] A. Ghazal *et al.*, "BigBench: Towards an industry standard benchmark for big data analytics", in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York New York USA: ACM, 22 Jun. 2013, pp. 1197–1208, ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2463712. [Online]. Available: https://dl.acm.org/doi/10.1145/2463676.2463712.

[88] D. Agrawal *et al.*, "SparkBench – a spark performance testing suite", in *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things*, R. Nambiar and M. Poess, Eds., vol. 9508, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 26–44, ISBN: 978-3-319-31408-2 978-3-319-31409-9. DOI: 10.1007/978-3-319-31409-9_3. [Online]. Available: http://link.springer.com/10.1007/978-3-319-31409-9_3.

[89] S. Huang, J. Huang, J. Dai, T. Xie and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis", in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, Long Beach, CA, USA: IEEE, 2010, pp. 41–51, ISBN: 978-1-4244-6522-4. DOI: 10.1109/ICDEW.2010.5452747. [Online]. Available: http://ieeexplore.ieee.org/document/5452747/.

[90] S. Rizzi and E. Gallinucci, "CubeLoad: A parametric generator of realistic OLAP workloads", in *Advanced Information Systems Engineering*, M. Jarke *et al.*, Eds., red. by D. Hutchison *et al.*, vol. 8484, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 610–624, ISBN: 978-3-

319-07880-9 978-3-319-07881-6. DOI: 10.1007/978-3-319-07881-6_41. [Online]. Available: http://link.springer.com/10.1007/978-3-319-07881-6_41.

[91] Z. Liu, X. Zuo, Z. Li and R. Han, “SparkAIBench: A benchmark to generate AI workloads on spark”, in *Benchmarking, Measuring, and Optimizing*, W. Gao, J. Zhan, G. Fox, X. Lu and D. Stanzione, Eds., vol. 12093, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 215–221, ISBN: 978-3-030-49555-8 978-3-030-49556-5. DOI: 10.1007/978-3-030-49556-5_21. [Online]. Available: http://link.springer.com/10.1007/978-3-030-49556-5_21.

[92] M. Lattuada, E. Barbierato, E. Gianniti and D. Ardagna, “Optimal resource allocation of cloud-based spark applications”, *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1301–1316, 1 Apr. 2022, ISSN: 2168-7161, 2372-0018. DOI: 10.1109/TCC.2020.2985682. [Online]. Available: https://ieeexplore.ieee.org/document/9057697/.

[93] K. Pawlikowski, “Towards credible and fast quantitative stochastic simulation”, Jan. 2003.

[94] L. F. Perrone, C. S. Main and B. C. Ward, “SAFE: Simulation automation framework for experiments”, in *Proceedings Title: Proceedings of the 2012 Winter Simulation Conference (WSC)*, Berlin, Germany: IEEE, Dec. 2012, pp. 1–12, ISBN: 978-1-4673-4782-2 978-1-4673-4779-2 978-1-4673-4780-8 978-1-4673-4781-5. DOI: 10.1109/WSC.2012.6465286. [Online]. Available: http://ieeexplore.ieee.org/document/6465286/.

[95] M. Silva, M. R. Hines, D. Gallo, Qi Liu, Kyung Dong Ryu and D. Da Silva, “CloudBench: Experiment automation for cloud environments”, in *2013 IEEE International Conference on Cloud Engineering (IC2E)*, Redwood City, CA: IEEE, Mar. 2013, pp. 302–311, ISBN: 978-0-7695-4945-3 978-1-4673-6473-7. DOI: 10.1109/IC2E.2013.33. [Online]. Available: http://ieeexplore.ieee.org/document/6529297/.

[96] A. Cairo, *The functional art: An introduction to information graphics and visualization* (Voices that matter series). New Riders, 2013, tex.lccn: 2012289140, ISBN: 978-0-321-83473-7. [Online]. Available: https://books.google.de/books?id=BiT1ugAACAAJ.

[97] Y. Hong, S. Du and J. Leng, “Evaluating presto and SparkSQL with TPC-DS”, in *Database Systems for Advanced Applications. DASFAA 2022 International Workshops*, U. K. Rage, V. Goyal and P. K. Reddy, Eds., vol. 13248, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 319–329, ISBN: 978-3-031-11216-4 978-3-031-11217-1. DOI: 10.1007/978-3-

031-11217-1_23. [Online]. Available: https://link.springer.com/10.1007/978-3-031-11217-1_23.

[98] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala and T. Cruanes, "Building an elastic query engine on disaggregated storage", in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 449–462, ISBN: 978-1-939133-13-7. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/vuppalapati.

[99] R. Kimball, *The data warehouse toolkit: practical techniques for building dimensional data warehouses.* John Wiley & Sons, Inc., 1996, ISBN: 0-471-15337-0.

[100] M. Levene and G. Loizou, "Why is the snowflake schema a good data warehouse design?", *Information Systems*, vol. 28, no. 3, pp. 225–240, May 2003, ISSN: 03064379. DOI: 10.1016/S0306-4379(02)00021-2. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0306437902000212.

[101] M. Barata, J. Bernardino and P. Furtado, "An overview of decision support benchmarks: TPC-DS, TPC-h and SSB", in *New Contributions in Information Systems and Technologies*, A. Rocha, A. M. Correia, S. Costanzo and L. P. Reis, Eds., vol. 353, Series Title: Advances in Intelligent Systems and Computing, Cham: Springer International Publishing, 2015, pp. 619–628, ISBN: 978-3-319-16485-4 978-3-319-16486-1. DOI: 10.1007/978-3-319-16486-1_61. [Online]. Available: https://link.springer.com/10.1007/978-3-319-16486-1_61.

[102] Transaction Processing Performance Council (TPC), *TPC benchmark ™ DS - standard specification, version 3.2.0*, 2021. [Online]. Available: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf.

[103] M. Rodrigues, M. Y. Santos and J. Bernardino, "Experimental evaluation of big data analytical tools", in *Information Systems*, M. Themistocleous and P. Rupino Da Cunha, Eds., vol. 341, Series Title: Lecture Notes in Business Information Processing, Cham: Springer International Publishing, 2019, pp. 121–127, ISBN: 978-3-030-11394-0 978-3-030-11395-7. DOI: 10.1007/978-3-030-11395-7_12. [Online]. Available: http://link.springer.com/10.1007/978-3-030-11395-7_12.

[104] NumFOCUS, Inc., *Pandas.* [Online]. Available: https://pandas.pydata.org/ (visited on 8 Jul. 2024).

[105] M. Jansen, L. Wagner, A. Trivedi and A. Iosup, "Continuum: Automate infrastructure deployment and benchmarking in the compute continuum", in *Proceedings of the first FastContinuum workshop, in conjuncrtion with ICPE, coimbra, portugal, april, 2023*, 2023. [Online]. Available: https://atlarge-research.com/pdfs/2023-fastcontinuum-continuum.pdf.

[106] M. Hopkins, “Problems of PL/i for system programming”, *ACM SIGPLAN Notices*, vol. 6, no. 9, pp. 89–91, Oct. 1971, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/942596.807063. [Online]. Available: https://dl.acm.org/doi/10.1145/942596.807063.

[107] The Apache Software Foundation., *Apache hive*. [Online]. Available: https://hive.apache.org/ (visited on 13 Jul. 2024).

[108] Google Cloud, *Google cloud storage*. [Online]. Available: https://cloud.google.com/storage (visited on 20 Aug. 2024).

[109] Amazon AWS, *Amazon s3*. [Online]. Available: https://aws.amazon.com/s3/ (visited on 20 Aug. 2024).

[110] MiniO, Inc., *Minio*. [Online]. Available: https://min.io/ (visited on 11 Jul. 2024).

[111] OpenIO, *Quick start*. [Online]. Available: https://docs.openio.io/latest/source/sandbox-guide/quickstart.html (visited on 20 Aug. 2024).

[112] GitHub, *seaweedfs/seaweedfs*. [Online]. Available: https://github.com/seaweedfs/seaweedfs (visited on 20 Aug. 2024).

[113] InfluxData, Inc., *Telegraf*. [Online]. Available: https://www.influxdata.com/time-series-platform/telegraf/ (visited on 11 Jul. 2024).

[114] N. Chan, “A resource utilization analytics platform using grafana and telegraf for the savio supercluster”, in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, Chicago IL USA: ACM, 28 Jul. 2019, pp. 1–6, ISBN: 978-1-4503-7227-5. DOI: 10.1145/3332186.3333053. [Online]. Available: https://dl.acm.org/doi/10.1145/3332186.3333053.

[115] Y. Dong, Z. Li, Y. Tian, C. Sun, M. W. Godfrey and M. Nagappan, “Bash in the wild: Language usage, code smells, and bugs”, *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–22, 31 Jan. 2023, ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3517193. [Online]. Available: https://dl.acm.org/doi/10.1145/3517193.

[116] H. Karau and R. Warren, *High performance Spark: best practices for scaling and optimizing Apache Spark*. O’Reilly Media, Inc., 2017, ISBN: 978-1-4919-4320-5.

[117] InfluxData, *Telegraf cpu plugin readme*. [Online]. Available: https://github.com/influxdata/telegraf/blob/master/plugins/inputs/cpu/README.md (visited on 20 Aug. 2024).

[118] InfluxData, *Telegraf mem plugin readme*. [Online]. Available: https://github.com/influxdata/telegraf/blob/master/plugins/inputs/mem/README.md (visited on 20 Aug. 2024).

[119] K. Sudnicina, “Task-in-pod scheduling support for kubernetes and apache spark stack”, Bachelor Thesis, VU Amsterdam, 2024.

[120] M. Triola, *Elementary statistics: Pearson New International Edition*. Pearson Education, 2013, ISBN: 978-1-292-05578-7. [Online]. Available: `https://books.google.de/books?id=j1SpBwAAQBAJ`.

[121] J. Ericson, M. Mohammadian and F. Santana, “Analysis of performance variability in public cloud computing”, in *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, San Diego, CA: IEEE, Aug. 2017, pp. 308–314, ISBN: 978-1-5386-1562-1. DOI: `10.1109/IRI.2017.47`. [Online]. Available: `https://ieeexplore.ieee.org/document/8102951/`.

Appendix A

Spark Configurations

```
apiVersion v1
kind Pod
metadata
  name driver
spec
  containers
  - name sharebench
    image lkmschulz2/sharebenchlatest
    resources
      requests
        ephemeral-storage "30G"
      limits
        ephemeral-storage "40G"
  nodeSelector
    driver-node true
  tolerations
  - key role
    value driver
    operator Equal
    effect NoSchedule
  imagePullSecrets
  - name regcred
```

Figure A.1: Driver pod template used in the experiments. Node selector and tolerations are used for controlling where drivers are scheduled.

```
apiVersion v1
kind Pod
metadata
  name executor
  labels
    spark-app-id $(SPARK_APP_ID)
spec
  containers
  - name sharebench
    image lkmschulz2/sharebenchlatest
    resources
      requests
        ephemeral-storage "8G"
      limits
        ephemeral-storage "8G"
  affinity
    podAntiAffinity
      requiredDuringSchedulingIgnoredDuringExecution
      - labelSelector
          matchExpressions
          - key spark-app-id
            operator In
            values
            - $(SPARK_APP_ID)
        topologyKey "kubernetes.io/hostname"
  imagePullSecrets
  - name regcred
```

Figure A.2: Executor pod template used in the experiments. (`$SPARK_APP_ID`) is replaced by a unique identifier for each of the concurrently active applications such that the affinity rules don't allow multiple executors of a single application on the same node and all applications are distributed over all nodes in Node-Level Sharing.

Table A.1: Spark configuration used in the experiments.

Property	**Value**
(a) All mechanisms	
spark.driver.cores	4
spark.driver.memory	20g
spark.kubernetes.driver.request.cores	2100m
spark.kubernetes.driver.limit.cores	4000m
spark.executor.cores	8
spark.executor.memoryOverheadFactor	0.25
spark.kubernetes.executor.deleteOnTermination	false
spark.kubernetes.allocation.batch.size	3
(b) Static Partitioning	
spark.executor.instances	$\left\lfloor \frac{\#workers}{\#apps} \right\rfloor$
spark.executor.memory	24g
spark.kubernetes.executor.request.cores	2000m
spark.kubernetes.executor.limit.cores	4000m
(c) Dynamic Partitioning	
spark.executor.memory	24g
spark.kubernetes.executor.request.cores	2000m
spark.kubernetes.executor.limit.cores	4000m
spark.dynamicAllocation.enabled	true
spark.dyanmicAllocation.minExecutors	0
spark.dynamicAllocation.maxExecutors	$\#workers$
spark.dynamicAllocation.initialExecutors	0
spark.dynamicAllocation.executorIdleTimeout	15s
spark.dynamicAllocation.shuffleTracking.enabled	true
spark.dynamicAllocation.shuffleTracking.timeout	15s
spark.kubernetes.allocation.batch.size	$\left\lfloor \frac{\#workers}{\#apps} \right\rfloor$
(d) Node-Level Sharing	
spark.executor.instances	$\#workers$
spark.executor.memory	$\left\lfloor \frac{24g}{\#apps} \right\rfloor$
spark.kubernetes.executor.request.cores	$\left\lfloor \frac{3600m}{\#apps} \right\rfloor$

Appendix B

Self Reflection

Although the outcome of this work is (mostly) satisfactory to me, the journey up to this point was far from smooth with a plethora of both technical and non-technical issues. In this chapter, I will give a brief self-reflection of the work that had a commanding influence on my life over the past 3 months.

Before starting work on this thesis, I had virtually no experience with frameworks like Spark and Kubernetes, concepts like containerization, and distributed computing as a whole. With this, the majority of the first month was spent on trying to amass as good of an overview of the field as possible in the limited time, mostly through reading. At the same time, however, I had to start getting to grips with the software and tools required for the work. Even though I genuinely consider myself a fast learner, diving into a whole field of unknown software and software systems, often without or with minimal documentation, was a very steep learning curve. Further compounded by the fact that I am hesitant to ask for help when I feel that I should be able to solve an issue myself, it took over 2 weeks to get a simple Spark application running in a Kubernetes cluster.

Even after conquering the initial hurdles, the work continued to be challenging in various aspects. I cannot recall a single week where I did not have to get acquainted with at least one new concept or tool to continue my work. Although often exhausting, these continuous challenges allowed me to notice a significant improvement in my problem solving skills. Problems that were similar in complexity to ones that took me days to solve at the beginning were now often only a matter of hours. I was, and still am, nonetheless surprised by the sheer quantity of *things* that had to be thought about, understood, engineered, fixed, or improved. The thousands of lines of code and tens of pages in writing tell a part of the story, but, in my view, fail to capture the full extent of my work and efforts.

B. SELF REFLECTION

Although I expected the thesis to be more challenging than the preceding courses in the curriculum, I did not expect the divergence to be this significant. Furthermore, the thesis required a drastically different approach, as both the solution and the requirements needed to be developed at the same time, which is a contrast to the assignments of regular courses where the latter is typically given and clearly defined. Although much more demanding, I often found myself appreciating both the challenge and the freedom it allowed for my solutions. However, not having a clear goal and consequently spending a lot of time on issues for which the solutions ultimately did not contribute to the final work had a noticeable impact on my motivation numerous times.

As the thesis is an individual work, a significant part of the 3 months was spent working alone. Furthermore, since the curriculum does not include any courses on distributed systems, conversations with my friends and fellow students about this work were limited to very high levels of abstraction. I personally found myself to dislike this rather lonely way of working and much prefer collaborative projects involving the management of a diverse team and engaged conversations about the various and complex issues. However, working by and for myself did have the advantage of a great level of freedom, not usually found in collaborative work.

In conclusion, I am happy with my effort and the work I was able to produce. Looking back, there are many things that could have gone better, yet through the issues and struggles that I experienced I was able to develop and refine many of my skills, improve my processes, and greatly extend my mental toolset. I also learned what I am still struggling with, such as maintaining motivation after setbacks, which helps me to understand what skills I still need to develop for my future work. I look forward to continuing this journey in my next academic project(s) - *I'm looking at you, Master Thesis!*