UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

EXPERIMENTAL METHODS FOR THE EVALUATION OF BIG DATA SYSTEMS

THÈSE

présentée et soutenue publiquement le 17 janvier 2020

pour l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE LORRAINE
(MENTION INFORMATIQUE)

par

ABDULQAWI SAIF

Composition du jury :

Rapporteurs

Jalil Boukhobza, Maître de conférences, Université de Brest, France

Christophe Cérin, Professeur, Université de Paris-XIII, France

Examinateurs

Marine Minier, Professeur, Université de Lorraine, France

Christine Morin, Directrice de recherche, Inria, France

Directeurs de thèse

Lucas Nussbaum, Maître de conférences, Université de Lorraine, France

Ye-Qiong Song, Professeur, Université de Lorraine, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications – UMR 7503

To my beloved children, Waleed and Ilyas
Your smiles were and will always be my source of inspiration

To my parents, brothers, and sisters
For their faith in me and their encouragement

To my lovely wife, Wafa
For her support, love, and patience

To everyone who believes that peace, coexistence, and love are our powerful arms to battle for a better world

ABSTRACT (ENGLISH)

In the era of big data, many systems and applications are created to collect, to store, and to analyze massive data in multiple domains. Although those – big data systems – are subjected to multiple evaluations during their development life-cycle, academia and industry encourage further experimentation to ensure their quality of service and to understand their performance under various contexts and configurations. However, the experimental challenges of big data systems are not trivial. While many pieces of research still employ legacy experimental methods to face such challenges, we argue that experimentation activity can be improved by proposing flexible experimental methods.

In this thesis, we address particular challenges to improve experimental context and observability for big data experiments. We firstly enable experiments to customize the performance of their environmental resources, encouraging researchers to perform scalable experiments over heterogeneous configurations. We then introduce two experimental tools: *IOscope* and *MonEx* to improve observability. *IOscope* allows performing low-level observations on the I/O stack to detect potential performance issues in target systems, convincing that the high-level evaluation techniques should be accompanied by such complementary tools to understand systems' performance. In contrast, *MonEx* framework works on higher levels to facilitate experimental data collection. *MonEx* opens directions to practice experiment-based monitoring independently from the underlying experimental environments. We finally apply statistics to improve experimental designs, reducing the number of experimental scenarios and obtaining a refined set of experimental factors as fast as possible.

At last, all contributions complement each other to facilitate the experimentation activity by working almost on all phases of big data experiments' life-cycle.

Keywords: experimentation, big data, experimental methods, observability

ABSTRACT (FRENCH)

À l'ère du big data, de nombreux systèmes et applications sont créés pour collecter, stocker et analyser des données volumineuses dans des domaines divers. Bien que les systèmes big data fassent l'objet de multiples évaluations au cours de leur cycle de développement, les secteurs de recherches public et privé encouragent les chercheurs à faire des expérimentations supplémentaires afin d'assurer la qualité de leurs services et comprendre leur performance dans des contextes et des configurations variés. Cependant, les défis expérimentaux des systèmes big data ne sont pas triviaux. Alors que de nombreux travaux de recherche utilisent encore de vieilles méthodes expérimentales pour faire face à de tels défis, nous pensons que l'activité d'expérimentation peut être améliorée en proposant des méthodes expérimentales flexibles et à jour.

Dans cette thèse, nous abordons des défis particuliers pour améliorer le contexte expérimental et l'observabilité des expériences big data. Premièrement, nous permettons la personnalisation de la performance de ressources environnementales où les expériences s'exécutent, en encourageant les chercheurs à effectuer des expériences à l'échelle sur des configurations hétérogènes. Nous contribuons ensuite aux outils expérimentaux *IOscope* et *MonEx* pour améliorer l'observabilité. *IOscope* permet d'effectuer des observations de bas niveau sur la pile d'entrée/sortie afin de détecter d'éventuels problèmes de performance sur l'environnement d'exécution. *IOscope* est développé pour convaincre que les techniques d'évaluation de haut niveau doivent être accompagnées par ces outils complémentaires afin de comprendre la performance. En revanche, le framework *MonEx* fonctionne aux niveaux supérieurs pour faciliter la collecte de données expérimentales. *MonEx* est le premier outil qui fait du monitoring autour des expériences indépendamment des environnements expérimentaux sous-jacents. Nous appliquons enfin des statistiques pour améliorer les conceptions expérimentales, en réduisant le nombre de scénarios expérimentaux et en obtenant un ensemble raffiné de facteurs expérimentaux aussi rapidement que possible.

Enfin, toutes les contributions se complètent pour faciliter l'activité d'expérimentation en travaillant sur presque toutes les phases du cycle de vie des expériences big data.

Mots-clés : expérimentation, big data, méthodes expérimentales, observabilité

PUBLICATIONS

The manuscript is based on scientific articles that were already published during the working period of this thesis. Most of the thesis' ideas, figures, and tables have appeared previously in those scientific publications:

- The topic discussed in Chapter 3 has been published as an international workshop article [118].
- The ideas of Chapter 4 has appeared in two publications. The main idea has been published as an international workshop article [114] while the second idea is published as a research report [115].
- The use case study of Chapter 4 has been firstly stated in an invited talk [113] and a conference poster [112].
- The main topic of Chapter 5 has been previously published in an international workshop article [116].
- The experimental studies of Chapter 5 have also been stated in an invited talk [117].
- The main idea of Chapter 6 has been published in a conference article [111].

To summarize, the ideas presented in this manuscript are the result of five scientific publications [111, 114–116, 118], two invited talks that meet the peer review principles [113, 117], and a conference poster [112]. Beside these publications, the thesis contributed two experimental tools and extended an existing one via adding an experimental service on top of it.

None is so great that he needs no help, and none is so small that he cannot give it

— King Solomon

ACKNOWLEDGMENTS

Tackling this Ph.D. has been a life-changing experience and a cornerstone in my career, and it would not have been possible to do without help, support, and guidance received from many people.

First, it is a genuine pleasure to express my deep sense of thanks and gratitude to my supervisors *Lucas Nussbaum* and *Ye-Qiong Song* for all the support and encouragement they gave me during four years of working together, for allowing me to grow as a researcher scientist via their patience, surveillance, and trust. Without their guidance and constant feedback, this Ph.D. would not have been achievable.

Besides, I would like to thank all members of *Madynes* and *RESIST* research teams in Loria laboratory with an extended thanks and gratitude to *Teddy Valette, Clement Parisot, Florent Didier, Ahmad Abboud, Abir Laraba, Nicolas Schnepf, Daishi Kondo, Elian Aubry, Jérémie Gaidamour,* and *Arthur Garnier*. I would also send a special thanks to *Alexandre Merlin* with who I collaborated and overcome some technical challenges during parts of this thesis. With all of you, colleagues, I have shared discussions, ideas, concerns, scientific missions, travels, and meals over many years during my Ph.D.

I gratefully acknowledge the funding received towards my Ph.D. from the *Xilopix* enterprise which becomes a part of the *Qwant Research* enterprise. I would like to thank all colleagues in those two enterprises with a special thanks to *Cyril March, Sébastien Demange, Luc Sarzyniec, Marion Guthmuller,* and *Jérémie Bourseau*. Thank you for the professional and personal moments that we have shared together in the enterprise's office.

I would also send my thanks and gratitude to the staff of the *Grid'5000* platform who have been hardworking to make the platform always available for researchers. Indeed, all experimental studies presented in this thesis have been performed over that platform.

It is my privilege to thank my friends *Wazen Shbair, Safwan Alwan, Mohsen Hassan,* and *Younes Abid* for sharing their life experiences as former Ph.D. studients, and for their periodic doses of moral support that I recieved throughout my Ph.D. journey.

A very big thank you to my closest friends and colleagues *Cristian Ruiz* and *Emmanuel Jeanvoine* for their constructive criticism, motivations, and sense of humor. Without you, guys, my daily life in the laboratory may have been limited to do greetings from a distance and making yellow smiles.

Once again, thank you all. I will remain indebted to you forever.

CONTENTS

LIST OF TABLES

LISTINGS

ACRONYMS

API Application Programming Interface

BCC BPF Compiler Collection

CIFS Common Internet File System

Distem DISTributed systems EMulator

DoE Design of Experiments

eBPF extended Berkeley Packet Filter

EMF Experiment Monitoring Framework

Fio Flexible I/O Tester

HDD Hard Disk Drive

HDFS Hadoop Distributed File System

IoT Internet of Things

JBOD just a bunch of disks

mmap Memory Map

MonEx Monitoring Experiments Framework

NCQ Native Command Queuing

NFS Network File System

NoSQL Not Only SQL

OLAP Online Analytical Processing

OML ORBIT Measurement Library

OS Operating System

pnode Physical Node

RAM Random Access Memory

REST Representational State Transfer

RPC Remote Procedure Call

SNMP Simple Network Management Protocol

SQL Structured Query Language

SSD Solid State Drive

SSH Secure Shell

SSTable Sorted Strings Table

SUT System Under Test

syscall Linux System Call

VFS Virtual File System

vnode Vrtual Node

WAN Wide Area Network

WSN Wireless Sensor Network

YCSB Yahoo! Cloud Serving Benchmark

Part I

PREAMBLE

The greatest challenge to any thinker
is stating the problem in a way
that will allow a solution

— Bertrand Russell

1 INTRODUCTION

1.1 GENERAL CONTEXT

Computer science is evolving rapidly. Several decades ago, systems were designed to run on single computers, achieving their basic tasks in isolation. The non-stop breakthroughs in software and hardware sides pave the way towards having complex architectures for computer systems. Together with the fact that the curve of producing high-speed CPUs cannot be pushed even further, the advances in distributed algorithms and networking lead to considering going faster and further through distribution [74, 140]. Today, distributed systems coordinate a considerable number of independent computers that perform their tasks and share their resources in a transparent way perceived by the end users. As can be seen, multiple sectors from the economy to the entertainment are getting inspired to build scalable infrastructure, providing global services, and targeting people worldwide. Without a doubt, people are getting quickly into the digital world producing and consuming an enormous quantity of data. According to the *international data corporation – IDC*, individuals are responsible for generating around 75% of the information in the digital world [46].

Recently, big data became one of the hottest topics in computer science research. Many institutions including governments, enterprises, and media show large interests in big data and its prospective [14, 28, 80]. Although the term "big data" does not give precise boundaries of what we are dealing with, Wikipedia defines it as "*datasets that are too large or complex for traditional data processing application software*". Admitting that the words "large" and "complex" add nothing to the word "big" in terms of precision, the definition criticizes the abilities of current systems once used to deal with big data. For sure, big data challenges are significant. These challenges are not only related to the volume of data which is increasing every day but also concerning several aspects known as *big data V's*. Aside from the *volume*, the challenges touch also the *velocity* of generating and processing data, the *variety* of data, e.g., textual, images, videos, logs, and phone calls gathered from multiple sources, and the *value* to be extracted from that data [60, 66, 67, 73]. That explains a bit why traditional systems are limited to address such kind of challenges. Provided that, having new systems to overcome a partial set or all challenges related

to the storage, analysis, and processing of big data is crucial to maintaining control over that emerging data. We talk about big data systems.

Big data system development is a challenging process. Such systems should overcome two general classes of challenges. First, they face challenges associated with existing systems such as I/O systems and analytic systems as they are part of the workflow of big data systems. Second, they are mainly concerned to satisfy new requirements related to the characteristics of big data [26, 27]. On one hand, a large number of big data systems are often built over recent ideas which were not exploited before, so making little usage of the research experiences constructed over time. For instance, big data storage systems deal with challenges of holding heterogeneous data together, coming up with novel data models which are largely different from the relational model used in *SQL* databases. On the other hand, they must manage new requirements such as maintaining their on-top services always available and controlling the data consistency. This comes, however, at the price of sophisticated architectures and designs which may be error-prone and hide many kinds of issues (e.g., functional and performance issues). Overall, the points discussed here show a need for more investigation and research to create robust big data systems.

Similarly to the majority of computer systems, ensuring the quality of services of big data systems can be handled via evaluation. In theory, such evaluations can be performed with the help of formal methods. This latter represents systems' specifications as mathematical models to be validated via techniques such as model checking. These techniques are however used in domains such as formal specification [94] and verification [119], but not for systems with complex implementation. In practice, the intricate design, and implementation of systems like big data systems may represent a stumbling block against such usage. Capturing all expected behaviors by an abstracted model is not trivial. Moreover, environments which play a role in systems' life-cycle should also be considered in such evaluations. However, it is not the case with formal methods.

Experimental validation is the logical alternative to formal validation and it is primarily used in computer systems [125, 146, 158]. One reason among others to do experimental evaluation is the infeasibility to produce comprehensive models for formal validation purpose. The more the complexity of the *system under test – SUT*, the more the orientation towards validation methods that are based on observation. To this end, experiments directly face real complexities which are either related to the SUT, the experimental environment, or both of them, so bringing conclusions from real use cases. Meanwhile, there are no standards to identify good experiments or better experimental environments. Also, it is not odd to have different methodologies and environments to experiment on a given domain of computer systems.

For clarification and to define the scope of this thesis, experimental research on big data systems is the main topic here. We give particular consideration to the core component of that research: experiments. We argue that the life-cycle of big data experiments is similar to the legacy life-cycle of computer

experiments and the life-cycle of large-scale experiments mentioned in [89], having at least three main phases – *design*[1], *execution*, and *analysis*. In contrast, we claim that the particularities of big data systems (*see Section a.1.1*) incur new difficulties that affect the experimental flow along the experiments life-cycle. Continuing to perform evaluations without considering them may lead to unsatisfactory results. For instance, practicing in-depth observability is essential to understand the performance of big data systems. However, current practices in both – *academia and industry* – point to general tools [51] like *Yahoo! Cloud Serving Benchmark (YCSB)* [33], YCSB++ [102], and *BigDataBench* [154] to evaluate big data storage systems. Of course, such tools are designed for producing measurements rather than providing a meaningful understanding of performance; so they make a trade-off between experimentation flexibility and getting in-depth results. They perform evaluations over higher layers to cover many systems as they could be targeted at this level, but at the price of ignoring systems' peculiarities and internal architectures.

Facing big data challenges during experimentation implies to deal in first place with experimental methods. The state-of-the-art experimental methods in computer science are diverse. Picking out an appropriate one for big data experiments depends on several aspects including the environment and the characteristics of the target system. For instance, we often simulate the activities of a given system if experimenting on a real one incurs high costs or is just impossible. Based on the type of target systems (real or model) and the nature of environments (real or model), the experimental methods can be classified into four categories [50, 62]. *In-situ* experimentation (real system on real environment), *benchmarking* (model of a system on a real environment), *emulation* (real system on an environment model), and *simulation* (model of a system on an environment model). These approaches are usable for the evaluation of big data systems. However, derivating new methods that take precise challenges of big data systems into account is strongly required. Indeed, this is the main direction of this thesis.

1.1.1 *Characteristics of Big Data Systems*

The concept of *Big Data* is volatile. Researchers still have ambiguities regarding its precise definition, domains, technologies, and features [155]. Many pieces of research such as [36, 37, 44] appear over time to define *Big Data* regarding its characteristics found in the literature. Those works beside others encourage to describe *Big Data* by its features, commonly called *Vs*. The list of *Vs* which has been limited in an early time to *Volume* (data size), *Velocity* (data generation speed), and *Variety* (data types) is extended over time to include *Value* (the information being extracted) and *Veracity* (data quality). However, the meaning of *Vs* is changeable and can be interpreted differently regarding the underlying context. For example, the *Velocity* means by default the *pace of data generation*, but it refers to *updating speed* for social media systems, and to

1 Referred to as *composition* in [89] to show the value of design reusability

processing speed for streaming systems [52]. This high-level controversy might be reflected in the implementation of big data systems.

We define big data systems as all computer systems that collect, store, or analyze data that holds a partial or full set of *Big Data*'s *Vs*. Those systems may either use existing techniques and technologies to operate, such as leveraging distributed systems to permit horizontal scalability and using statistical principles to analyze diverse data. On the other hand, big data systems can also have their own technologies. A suitable example here is storage systems. Instead of using traditional storage systems such as SQL databases, which do not allow to store homogenous data nor to scale, big data storage systems rely on storage technologies that fit big data features. They can be classified into four categories [141], which are 1) *Distributed File Systems* such as *Hadoop Distributed File System (HDFS)* which provides storage basis for *Hadoop* ecosystem, 2) *Not Only SQL (NoSQL) databases* that introduce new storage models such as document-based and columnar models, 3) *NewSQL databases* which try to improve traditional databases to scale while maintaining their relational data model, and 4) *Querying platforms* that provide frontend storage queries for distributed file systems and NoSQL databases.

Big data systems also have characteristics that would affect the way we evaluate them. On the one hand, they have contrasted architectures despite belonging to the same family (e.g., storage systems). Hence, systems' interfaces may be implemented differently, so the possibilities to build broad experimental tools that target multiple categories of systems are decreased. On the other hand, one would think that performing evaluations on big data system interfaces is sufficient. However, those systems have a lot of complex behaviors, such as having heavy interactions in lower layers of *Operating Systems (OSs)*, which should be targeted by experiments too. All in all, although big data systems are distinct and match various subsets of *Big Data V's*, performing experiments on those systems incur common challenges. We describe them in the next section.

1.1.2 *Big Data Experimentation Challenges*

As highlighted above, the challenges that face big data experiments are enormous. The majority of them are related to the main experimentation questions which are *how, where, and on what data experiments can be done?* Although many decades and researches are devoted to answering these questions concerning other disciplines, the same questions are revived again to be reviewed from a big data angle. Therefore, we classify the common challenges of big data experiments into three intersected categories. These categories are 1) *workload/data* as the workload selection and data characteristics are one of the main factors for experimentation, 2) *design* to encompass all challenges related to experiments themselves, and 3) *environment* which plays a significant role in experiments' flow. Figure 1.1 shows a list of the major challenges of big data experimentation. Those challenges are briefly described below.

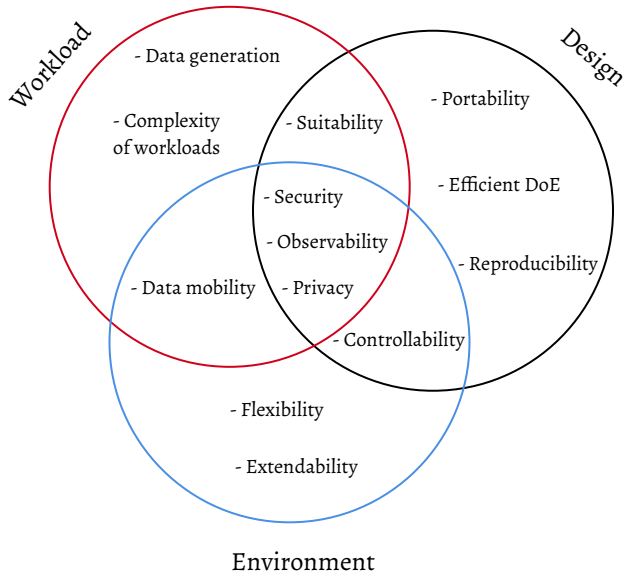


Figure 1.1: A list of the major challenges of big data experiments

- CONTROLLABILITY. Many experimental environments, including clouds and testbeds, offer resources that may not match the requirements of researchers' experiments by default. For example, an experiment needs to be performed on a heterogenous infrastructure in terms of memory performance while the available infrastructure is homogenous. Admitting that not all experiments are similar, many may require specific configurations that are, in some cases, not attainable with regard to the way the experimental resources are offered. Giving experiments further control over the environment is challenging as it implies resources and performance customization.

- OBSERVABILITY. The observability of big data experiments is not limited to see results, and resources performance, but also can be extended to observe low-level behaviors of big data systems. Using traditional observability methods (e.g., traditional monitoring systems) which are not mainly created for big data systems may have several technical and design obstacles (e.g., scalability limitation and incompatibility with specific big data systems). Hence, observability techniques should be improved, facilitating the ways of collecting experiments data and coming up with methods that fit big data experimental scenarios.

- COMPLEXITY OF WORKLOADS. Workloads for big data experiments have two sources [7]. They are either generated using data generators of known benchmarks (e.g., TPC-family and YCSB benchmarks) such as the Parallel Data Generation Framework [105], or using customized generators that fit specific experiments. For both types, the workloads are simple and do not reflect the complex characteristics of real workloads. Hence, the simplicity of workloads may be considered as a bias

that threat the results of experiments with regard to real-world use cases.

- Efficient *Design of Experiments*. Big data experiments tend to have rough designs in their early stages, i.e., collecting experimental factors by intuition. These rough designs do affect not only the experimental results but also increase experimental time and resources consumption. Proposing efficient *Design of Experiments* (*DoE*) may shorten the experimental time, reduce the experimental scenarios by focusing only on significant experimental factors which indirectly minimizes resource usage.

- Data mobility. Big data experiments often require running on multiple independent environments (e.g., Federation of networked and IoT testbeds) that do not necessarily have the same infrastructure. Maintaining efficient data movement between the experimental nodes despite the diverse technology, resources, and data transfer protocols is challenging, especially in case of having voluminous data. Even moving data for feeding the experimental nodes or finally for analysis, ensuring data consistency and data integrity should pass through eliminating multiple sources of I/O failures and coordinating many local and distributed protocols.

- Data generation. Generating realistic datasets for big data experiments is essential to enhance the activity of evaluating and comparing big data systems. This phase still has many challenges, such as generating data from a complex reference model and having hybrid data generators for structural and non-structural datasets.

- Environment extendability. Experimental environments are often static regarding their resources composition. Pushing them to embrace new scientific disciplines associated with big data systems often requires to add new resources, change their connectivity, and to build new communication layers. Doing so incur many immediate challenges as environments are always resistant to change.

- Workload suitability for experiments. Every big data system is unique in its design. Hence, experimenting on various systems using a similar workload does not guarantee to stress similar behaviors on those systems.

- Environment flexibility. Big data experiments have different needs in terms of resources and accompanying services (e.g., intermediate storage). Environments should be flexible regarding changes. They should deliver resources on-demand to experiments, supporting federation for experiments with massive and scalable resources usage, and providing secondary services such as analysis framework and storage facilities.

- REPRODUCIBILITY. Reproducing experiments is a key requirement for validating scientific results. However, it is one of the biggest challenges not only for big data experiments but for experiments in all scientific disciplines.
- EXPERIMENTS PORTABILITY. Performing an experiment on many environments is not possible without considering the portability in its design. Pushing towards making experiments 100% portable is challenging. This requires to eliminate the impact of experimental resources from the design.
- PRIVACY. Although some experimental environments such as testbeds are shared between several users over time, ensuring the privacy of experiments' logs and data may represent a real challenge.
- SECURITY. Many security challenges are present such as ensuring data integrity and encryption.

1.2 PROBLEM STATEMENT

Dealing with all challenges described in Section a.1.2 is beyond the scope of one thesis. Hence, this thesis considers a subset of those challenges, namely the *controllability*, and *observability*. Its key concern is to facilitate the experimentation activity throughout addressing this subset of challenges overall different phases of experiments' life-cycle. In particular, this thesis brings answers to the major questions listed below.

- *How to eliminate the problem of incompatibility between the offered performance of experimental environments and the sophisticated requirements of big data experiments? In particular, how to satisfy the needs of scalable experiments that require heterogeneous storage performance on clusters with homogeneous resources?*

  This question considers altering the environments to meet the needs of experiments rather than applying changes to the experiments' design or considering other experimental environments.

- *How to improve the observability for the experiments that have fine-grained measurements, e.g., thousands if not millions of operations in a small unit of time in order to have a better understanding of performance?*

- *How to improve experiments' observability on environments like networked testbeds, by overcoming challenges such as i) determining experiments' context, i.e., working at experiments' scale whatever the underlying infrastructure and ii) having methodological procedures for collecting experiments' data?*

  This research question emerges the following sub-derived questions:

  - *What are the requirements to build Experiment Monitoring Frameworks (EMFs)?*

  - *How to design and implement such a framework in order to satisfy the defined requirements?*

- *How to improve the designs of big data experiments to reduce their vast combinations that consume more resources than usual?*

  Many considerations are to be taken into account when designing such experiments. At an early stage, experiments have a large number of factors that may or may not affect the experiments' outcome. Also, constraints on resources usage may be given.

1.3 CONTRIBUTIONS AND OUTLINE

The thesis makes four major contributions to address the specific research interrogations described in the previous section. Figure a.2 shows the contributions together on the abstracted life-cycle of big data experiments. To describe the contributions regarding their placement over that figure, the opening two contributions aim at improving the experimental design and context by working on two distinct but relative points – *design* & *execution*. Firstly, given the rough state[2] of initial designs of experiments, we use statistics to obtain as fast as possible a design that allows for reduced sets of combinations which indirectly leads to consuming fewer experimental resources. We then push towards better executions by giving experiments further controllability over resources to form easy-to-configure environments. These two contributions lead to provoke questions about observing and analyzing experiments' results, especially for experiments having massive low-level behaviors and scalable infrastructure. Therefore, the last two contributions are devoted to improving the observability of experiments regardless of their experimental designs and environments. All thesis' contributions are described below with an order that matches their appearance in thesis' chapters.

HETEROGENEOUS STORAGE PERFORMANCE EMULATION

We introduce a service to customize storage experimental environments, creating realistic configurations through emulation. The service enables to have heterogeneous and controllable storage performance for complex and scalable experiments, reflecting the characteristics of end-environments to strengthen experimental results. It is implemented in the *DISTributed systems EMulator* (*Distem*), which is an open source solution known in the community of wired testbeds such as *Grid'5000*, *Chameleon*, and *CloudLab*, so reaching more people on various communities.

MICRO-OBSERVABILITY WITH IOSCOPE

We come up with a filtering-based profiling method for tracing low-level I/O operations, encouraging to observe tiny behaviors during big data experimentation. That method is implemented in a toolkit, namely *IOscope*,

2 An initial phase of most big data experiments where most experimental factors are collected by intuition to be studied

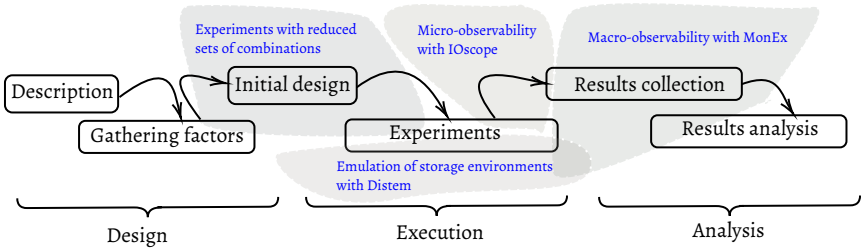


Figure 1.2: Thesis contributions presented on big data experiments' life-cycle

to characterize the I/O patterns of the dominant storage workloads. Using *IOscope* along with high-level evaluation tools is efficient for investigating workloads with performance questions as well as for understanding the main reasons behind potential performance issues.

MACRO-OBSERVABILITY WITH MONEX

We propose an experimental framework, namely *Monitoring Experiments Framework* (*MonEx*), to facilitate observing experiments from experimenters' viewpoint. MonEx eliminates the common ad hoc steps during experiments' data collection phase and is partially in charge of results' preparation, such as producing simple and ready-to-publish figures. It is built on top of off-the-shelf components to ensure running on multiple experimental environments.

REDUCTION OF EXPERIMENTAL COMBINATIONS

We confirm the applicability of statistics to reduce the number of experiments' runs, especially for experiments that do not have a clear scope at an earlier phase of design. This is done by utilizing a model that eliminates the nonrelevant factors of an experiment as well as the factors' interactions that do not influence experimental results. This does not only push towards having a stable experimental design as fast as possible, but it also saves a lot of experimental resources and effort compared with the common method that runs experiments with a full combination of experimental factors.

The thesis is structured in seven chapters, including this introductory part and the conclusion. Chapter 2 presents the state-of-the-art of the research presented in this thesis. It describes the experimental methodologies that can be used with big data systems before showing an overview of observability techniques including tracing and monitoring. It ends by presenting a short survey that studies in critical mode some selected experimental papers to characterize the ways used by big data community to perform experiments. Chapters 3,4,5, and 6 describe the main contributions of the thesis. The contribution chapters can be read separately, but with a preceding read of the state-of-the-art.

I keep six honest serving men. They taught me
all I knew. Their names are What and
Why and When and How and Where and Who

— Rudyard Kipling

2 STATE OF THE ART

This chapter describes the state of the art of this work by covering three main points that justify the placement of our contributions in the literature. First, Section 2.1 brings out background about big data evaluation and the experimental methods that can be used to evaluate big data systems. The same section also describes the continuous improvements of the benchmarking process to be adapted for big data evaluation. Second, we describe in Section 2.2 experimental observability with its two major techniques: *monitoring* & *tracing*. Indeed, those techniques represent the observation basis not only for big data, scalable, and distributed systems but rather for various fields of computer science. We then accomplish a question-based literature review in Section 2.3 to determine how experimental articles in academia and industry evaluate big data systems. Section 2.4 summarizes by describing the knowledge gap that is addressed in this thesis.

The following sections can be read separately without particular order. Regarding their contents, they contain high-level information that allows drawing the big picture of big data systems' evaluation. Hence, the particular context of each upcoming chapter complements the information stated here by dedicating an internal section to state the related work of the corresponding contribution.

2.1 BIG DATA EVALUATION

As stated in [73] that *"there is no clear consensus on what is Big Data"*, it is hard to have a stable list of features to identify big data systems. All systems that store and analyze a massive quantity of data can be considered as big data systems. Provided that, we start to see systems with non-common designs and architecture. For instance, systems that integrate different families of storage systems [79] and hybrid systems [21, 138]. In parallel, we continuously discover many challenges surrounding big data and the way we evaluate big data systems. *Bertino et al.* [13] discussed in detail diverse challenges such as the data heterogeneity, architecture-related challenges, and the importance to address them in order to leverage the full potential of big data. The authors affirmed that most of big data challenges are technical and can vary from one domain to another. Hence, applying suitable experimental methods are critically important. *Gudipati et al.* [49] confirmed that experimental approaches

Table 2.1: Experimental methodologies of large-scale systems as classified in [50]. They are usable – by variable ratios – in big data systems' evaluation

		System under test	
		Real	Model
Experimental environment	Real	*In-Situ*	*Benchmarking*
	Model	*Emulation*	*Simulation*

should not only be selected regarding the functional requirements of big data systems, but also their non-functional ones such as scalability, consistency, and fail-over. The same authors also confirmed in [47] that the challenges of big data systems are hard to be solved via legacy evaluation techniques and methods. According to them, all phases of big data testing should be improved via techniques, tools, and methods that reconsider the characteristics of big data systems (see Section a.1.1).

2.1.1 *Legacy Experimental Methods of Evaluation*

Big data systems are overall a part of computer systems, so all experimental methodologies available several decades ago are theoretically suitable for big data systems' evaluation. However, the functional and non-functional requirements in big data context play a role in the selection of a suitable experimental methodology, given the target experiments' goals and circumstances. For instance, big data experiments that focus on consistency and data integrity may require a different experimental method than experiments that evaluate computer networks. Indeed, consistency experiments should be tested on real and complex infrastructure, which plays a role in the determination of experiments' outcomes while most network experiments supply reliable results using simulation. Although the absence of a survey that assembles experimental methodologies for big data systems, we can rely on the survey that classified experimental methods for large-scale systems as a basis. Indeed, *Gustedt et al.* [50] have classified the experimental methodologies for large-scale systems into four categories: *In-Situ*, *benchmarking*, *emulation*, and *simulation* (see Table 2.1). This classification is made based on two key questions on the System Under Test (SUT) and the experimental environment: "*is the SUT real or model? is the experimental environment real or model?*". These questions are not literally meant for big data systems as the simulation is not highly used in the presence of massive clouds and data centers. However, some methodologies such as in-situ experimentation and benchmarking are still mostly usable in big data experimental research. We briefly describe below those experimental methodologies.

2.1.1.1 *In-Situ Experimentation*

In-Situ experimental method targets experiments with a real application or a SUT on a real environment. It is the most straightforward methodology to be

used as it requires no more preparation than having the *SUT* and the target environment. However, it is not as simple as this since all experimental configurations should be covered in order to attain the experimental goals behind the experiments. Obviously, it is tough to combine together all experimental scenarios due to several reasons such as the high experimental costs and the unreality to perform exhaustive plans of testing. Hence, real constraints and obstacles that encounter *in-situ* experiments make the experimental results of those experiments more natural to be generalized.

Experimental infrastructure at the disposal of big data researchers is diverse. The competitivity to offer best experimental conditions and services to experimenters over many experimental disciplines contributes to promoting *in-situ* experimentation. Hence, experimental testbeds can be found on the top of the experimental infrastructure, which also includes clouds, data centers, and private experimental resources.

2.1.1.2 *Benchmarking*

Benchmarking is the experimental method to use when having a model of the SUT to be evaluated over a real environment. According to that definition, we should have an application model, but in practice, this point is not literary respected when it comes to benchmark big data systems. We instead use real *SUT* throughout the evaluation process but with synthetic data. For instance, the *Yahoo! Cloud Serving Benchmark (YCSB)* tool can be used to evaluate different big data systems; it generates synthetic data to be used with real deployments of *SUT*.

In general, benchmarking is regularly used in computer science. *Huppler* discussed in [57] the grew usage of benchmarking in computer science experiments and then give five characteristics that design valuable benchmarks. These characteristics are *relevance*, *fair*, *economical*, *repeatable*, and *verifiable*. Even if several types of benchmarking like functional, micro, and macro benchmarks have differences in their scope and target behaviors inside the *SUT*, we argue that these characteristics should be present in all types of benchmarking.

2.1.1.3 *Emulation*

Emulation requires a real *SUT* to be evaluated over a model of experimental environment. This experimental method is extremely used when it is technically or costly impossible to have a real environment. In the evaluation of big data systems, the emulation is not only limited to environments but also includes performance and resources emulation. Eventually, it is not mainly used as in-situ experimentation and benchmarking, especially when it comes to evaluating non-functional requirements as scalability. However, outside the evaluation purpose, system deployments on emulated environments like clouds are frequent.

2.1.1.4 *Simulation*

Experimental simulation is about evaluating models of *SUT*s over models of experimental environments. Although this experimental method is known as the first evaluation step in other scientific domains like operational research, computer networks, and applied statistics, it is less used in practice in big data evaluation. Indeed, the realistic configurations are complex and may not be fully respected by simulation models.

2.1.2 *Benchmarking for Big Data Evaluations*

Benchmarking is mostly used to evaluate big data systems thanks to their simple mechanisms of data generation and connectivity with *SUT* through high-level interfaces. However, several studies focus on improving the benchmarking process to overcome the challenges of big data systems, which includes data heterogeneity and scalability. Additionally, several particular meetings, such as the *workshop on big data benchmarks (WBDB)* that target improving the big data benchmarking process are held regularly. The outcome of such meetings brought out several important tools such as *BGDC* [91], and *HcBench* [120] as well as valuable experimental articles.

Big data systems should be accompanied by a robust benchmarking process to cover its different aspects. *Baru et al.* stated in [11, 12] many suggestions to better benchmark big data systems. The authors stressed that big data benchmarks should support diverse workloads to cope with the flexibility of big data systems in processing various pipelines. The same authors confirmed in [10] that the process of building a good benchmark for big data systems should consider the workloads' type – *application-level and component-level* – and the target *SUT*s. They confirmed that the industry and academia should go more in-depth to uncover today and future's benchmarking problems. More precisely, *Fox et al.* listed in [43] all characteristics of big data systems under four categories: 1) *data source and style*, 2) *execution model*, 3) *processing view*, and 4) *problem architecture view*. The authors affirmed that several big data benchmarks already covered certain features while other features such as dataflow and veracity should be covered in the future.

Researchers dedicate a lot of time and efforts to find a suitable benchmark, to configure it, and to deploy the required software/hardware for the benchmarking process. *Ceesay et al.* [23] considered this as an issue and took the initiative to simplify the experimentation activity. They took the simple way to react by leveraging the advantages of containers' isolation. Their tool can package any benchmark inside a container in order to mitigate the effort of configuration. Besides, an equation to calculate the cost of any running workload on the cloud is given.

Surveying and evaluating the existing benchmarks can be seen as a good way to identify perspectives. *Ivanov et al.* published a survey [58] on different existing big data benchmarks to understand which benchmark is recommended for a given big data experiment. Although many benchmarks can be used on big data analytics, clouds, and grids, fewer benchmarks are targeting

distributed storage systems including NoSQL solutions. *Qin and Zhou* surveyed [104] the existing benchmarking that can be used with NoSQL databases and *Online Analytical Processing (OLAP)* transactions. They affirm that big data benchmarks should also focus on non-performance metrics as some of them, such as scalability and energy saving can influence the systems' behaviors. *Han et al.* [52] performed an evaluation of big data benchmarks over two aspects: generating complex data and covering different application-specific workloads. They affirmed that there are too much to be done in order to build representative and comprehensive benchmarks for big data systems. One of the future directions of their work is to encourage workloads characterizations and micro observability.

Eventually, several mistakes may be hidden behind the complexity of benchmarks and the way a given *SUT* should be benchmarked. *Chapira and Chen* [132] have identified five common mistakes that may be occurred throughout benchmarking big data systems. These pitfalls are 1) *comparing apples to oranges*, 2) *not testing at scale*, 3) *believing in miracles in results*, 4) *using unrealistic benchmarks*, and 5) *misleading result communications*. These mistakes often happen in experimental papers. For instance, a debate has been originated over how to benchmark Cassandra's NoSQL solution[1] after considering a recent benchmarking study that compares several NoSQL solutions including Cassandra as an *apple-to-orange* comparison.

2.2 EXPERIMENTAL OBSERVABILITY

Observability is one of the baselines of experimental science. Its overall goal is to allow understanding systems' behaviors in different situations, including either when things go wrong or not. Moreover, the repositories of observability can also be considered as a source of future analysis and maintenance [70]. Although the long history of experimental research in computer science, optimal observability cannot be totally achieved as there are always directions of improvements. Indeed, observability techniques are domain-dependent and should be flexible to include the change in designs and technologies of *SUT*s. For instance, the observability techniques for scalable systems should take into account the synchronization of systems' components in case of observing shared-component behaviors such as data partitioning in real-time.

The key-idea of observability is to understand systems' behaviors through analyzing data related to those systems, no matter if that data is generated by the target systems or by the underlying environments. The main data sources of observability, which are shown in Figure 2.1 are detailed below.

- METRICS. The simplest form of data that have several representations. Metrics can be either *counters* that represent a data feature in a given moment (e.g., number of connected clients), *gauges* which represents numerical values that increase or decrease over time (e.g., number of

1 *Datastax*'s blog on benchmarking *Cassandra*: https://www.datastax.com/dev/blog/how-not-to-benchmark-cassandra-a-case-study

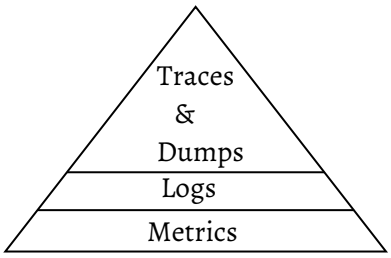


Figure 2.1: Main data sources of observability in big data experiments

concurrent requests), and histograms which summarize data over a given feature (e.g., doing sum on a stream of data). Although having several forms of metrics, they share one significant feature which is the ability to apply mathematical formulas.

Metrics are helpful to explore systems' data in an easy way. Constraints can be defined on specific metrics to alert users or to execute scripts in order to solve a raised issue. Also, they are the data which is mostly used for observation dashboards such as *Redash*[2], *Freeboard*[3], and *Grafana*[4].

- Logs. Logs are the second type of data sources that are used in experimental observations. They can be collected either from the *SUT*s or from the environment. Usually, they seem like textual data with inlined metadata that need to be parsed, analyzed, and even summarized to form refined data. Logs do not only allow to establish the order of events but also to obtain detailed information about precise events.

- Traces & Dumps. Traces and dumps are placed at the top of data sources pyramid as they are not usually used in observation if the metrics and logs are sufficient. Otherwise, traces construct the path of execution of systems' code generated either by the *SUT* itself or from the interaction of that system in lower layers of execution. Indeed, it comes with a cost during execution, i.e., adding a significant overhead in the worst case. On the other hand, dumps are thrown when a given system exits abnormally. Hence, ideally, it is better to avoid them, but they can assist in understanding certain behaviors once thrown. Both traces and dumps are useful to investigate why a system's component does behave in a certain way.

Many techniques are used to achieve the goals of observability by using the data sources mentioned above. Monitoring and tracing are the most used in practice when it comes to observing big data systems. Each technique complements the other where both can be used together to collect necessary information from different layers of execution. These two techniques are described in the following sections.

2 Redash website: https://redash.io/
3 Freeboard website: https://freeboard.io/
4 Grafana website: https://grafana.com/

2.2.1 *Monitoring*

Monitoring is mostly used to know more about the infrastructure state and the experiments' execution in run time. Almost all monitoring techniques are based on metrics collection from the experimental environment. The collected metrics may directly be part of monitoring results, or they pass in several mathematical calculations to form interpretable results.

Generally, monitoring tools and services are diverse. However, almost all of them are built around the infrastructure and how to provide live feedback regarding the resources' state. Hence, all of them are generic and can be used in multiple scenarios of usage, of course, including the big data experimentation scenarios. In contrast, we can find several tools that are developed around big data usage. For instance, *John et al.* [72] have built an observability framework for big data storage systems. Their framework automates the data collection phase for several NoSQL solutions. However, and globally, the idea of covering big data experiments as a separate track of monitoring is not yet exploited in the literature. Provided that, building monitoring frameworks around big data experiments is a main direction of this work (see Chapter 5).

Although the diversity that can be found in monitoring software and services, almost all of them are built around a monitoring approach that is compatible with their use cases. In the next section, an abstraction of all the monitoring approaches is shown.

2.2.1.1 *Monitoring approaches*

Two families of monitoring approaches can be crossed over to obtain a suitable monitoring method for any given use case. The first family is about agents while the second one corresponds to the way the data is acquired from the target system. Each family has two types. These two families are described below.

Agent- and Agentless-based Monitoring

- Agent-based monitoring. This approach consists to place a monitoring agent in every component/machine of target system regardless of their scalability. The agents communicate with their surrounding environment to look for the target metrics before transmitting those metrics' values to a central monitoring node.

- Agentless monitoring. This monitoring approach does not require any agent installation in the target environments. In fact, this approach mainly relies on some network communication protocols such as *Secure Shell (SSH)* and *Simple Network Management Protocol (SNMP)*. Using such a monitoring approach is impossible without the support of some operating systems such as Linux which prepares some statistics about internal behaviors and infrastructure, and facilitates collecting data remotely. Generally speaking, this monitoring approach depends on natively-built components in underlying environments.

Push- and Pull-based Monitoring

- Push-based monitoring. In this monitoring approach, the monitoring agents or components are responsible for transferring data to the central monitoring node regarding their own rhythms of data acquisition. This approach is useful either for systems that collect their data by a higher rate than that supported by monitoring systems (e.g., high-frequency measurements) or either for non-periodic events.
- Pull-based monitoring. Using this approach means that the monitoring system is responsible for collecting monitoring data from all monitored nodes. The data collection is periodic, so when the time lab is fired, the monitoring node sends a request to the involved machines in order to claim pre-defined metrics or logs.

2.2.2 *Tracing*

Tracing can be used to collect data from the *SUT* or the *OS*. It is mostly used to investigate certain behaviors of *SUT* in order to judge their quality of service or generally to understand those behaviors. Collecting traces is a complex activity which should be done without delaying or impacting systems' execution, i.e., without adding an *observer effect*. Moreover, although some traces are simple and ready to be parsed to construct the high-level image about the investigated behaviors, certain traces are like row data and may be subject to more work to extract a meaningful information.

Two techniques of tracing can be used with big data systems: *static and dynamic tracing*. *Static tracing* requires doing the instrumentation during or before the compile phase. Hence, using such a tracing approach may have several challenges to trace systems' behavior if there is no access to systems' code. In contrast, *dynamic tracing* gives experimenters the possibility to trace inside *OS-cores* by establishing connections with target instrumentation points during runtime.

Focusing on *dynamic tracing*, various tracing methods like *DTrace* [90], *LTTng* tools [40], *SystemTap* [59], and *extended Berkeley Packet Filter (eBPF)* are used for tracing inside and outside the Linux kernel. With the exception of eBPF, those tools are all limited to connecting to predefined data sources (e.g. *kprobes, uprobes, tracepoints, and kernel events*). Furthermore, *DTrace*, *LTTng*, and *SystemTap* create modules representing the tracing tasks to be dynamically loaded into the kernel. Although loading modules does not require modifying the running kernel, this action is not allowed in case of having signed kernels. Using such tracing tools also has several secondary impacts. Such usage implies doing posterior efforts for analyzing massive quantity of collected traces and it requires to have a set of compilation tools on the target environment, which could not be feasible on a production usage. Regarding *eBPF*, it resides in the Linux kernel starting from *Linux v3.19* [78] in order to extend Linux kernel functionalities. It requires no configuration to run,

Table 2.2: Comparison between *eBPF* and other tracing tools

	eBPF	DTrace [90]	SystemTap [59]	LTTng [40]
Scripting	byte-code, c	D language	stap language	Only commands
How it works	Standalone VM in kernel	Loads dynamic modules into the kernel	Dynamic modules, *eBPF backend*	Dynamic modules
Restricted to tracing, debugging, and profiling?	No, also filtering, networking	Yes	Yes	Yes
Officially in Linux?	Yes, starting from v3.18	No	Yes	No
Frontends	Python & Lua by BCC project, go	No	Yes	No

and it incurs negligible overhead [133, 139] which makes it suitable to run in production. Given those features, some tracing methods such as *SystemTap* starts to support *eBPF* as a backend to facilitate their usage. Table 2.2 shows an abstracted comparison between some selected tracing tools while more information about *eBPF* can be found in Chapter 4 (see Section 4.2.1).

2.3 A REVIEW OF BIG DATA EXPERIMENTS

This section explores the methods and techniques that are practically used in experimental articles to evaluate big data systems. The goal is to describe common issues in big data experimental papers. Hence, the literature has been studied around the following questions:

- Does the *DoE* play a major role in big data experimental articles?
- How deep is the observation process that is used for collecting results in experimental articles?
- What are the aspects and requirements to be considered in order to perform fair testing on big data systems?
- What are the aspects of data, resources, and services that are common in experimental articles?

Many peer-reviewed articles were selected to prepare this review. The diversity of application domains of big data systems as well as the lack of unique proceedings for big data experiments pushes towards having mix-collection of papers. Hence, this section is not a whole contribution and cannot be considered as a standalone review or survey. Instead, it delivers convincing arguments that support the contributions of this thesis. All questions stated above are separately discussed in the upcoming subsections.

2.3.1 *Rare Use of DoE*

The main usage of experimental designs in big data experiments is to identify the cause and effect relationships between experimental factors [75]. Although researchers collect and identify several experimental factors in initial designs by intuition, *Design of Experiments (DoE)* principles take several aspects such as experimental objectives, randomization, and homogeneity of factors to clarify the experimentation activity and its potential scenarios.

Screening techniques are the famous type of experimental designs used in computer science. They help to identify the impact of experimental factors on experiments' outcome in order to build coherent experimental scenarios and conclusions. It can be employed to obtain meaningful results when the number of independent factors is huge and the known information of their causality is scarce. Although it is the most used technique, the number of experimental articles in diverse fields of big data which employs it is not high due to several reasons including the simplicity of natural alternatives such as

testing all possible combinations of experimental factors and the tendency to bring faster results despite designs' effectiveness. To name a few articles, studies such as [68, 151] have employed screening *DoE* models to identify the importance of the employed experimental factors in noise filtering and grid computing fields, respectively.

Employing the principles of experimental design can deal with functional and non-functional requirements. Several directions and challenges are still to be explored. For instance, explaining the usefulness of using *DoE* techniques to deal with non-functional requirements of experimentation such as minimizing the utilization of experimental resources or coming up with new models and techniques that can deal with bias challenge [103] in big data experimental context.

2.3.2 *Lack of Deeper and Scalable Observations*

The presence of low-level experimental studies is required to evaluate and to understand the behaviors of big data systems. However, the majority of experimental articles in diverse domains of big data applications focus on high-level evaluations and comparisons. This tendency is supported by the increasing number of big data systems that should be compared together (e.g., big data storage) and the reputation of benchmarking tools to handle this type of evaluation.

Exploring the experimental articles of big data storage systems confirms the lack of detailed evaluations. *Dede et al.* [38] evaluated *Cassandra* storage system that allows for fast queries in case of being used in big data analytics. Although the number of metrics that can be used to support the authors' idea is high, the authors have been satisfied to rely only on the processing time metrics in that study. The same authors have made the same study with *MongoDB* instead of Cassandra [39], always focusing on the processing time of queries as the main output of their experimental study. Similarly, *Nelubin and Engber* [93] performed a performance evaluation on several NoSQL databases using *YCSB*. Their experiments have only focused on latency as a primary experimental factor as it is the main output of *YCSB* benchmark.

Similarly, non-functional aspects like scalability are not well treated in the majority of experimental articles. For instance, *Gandini et al.* [45] have evaluated many NoSQL databases for clouds usage, focusing on the relationship between the number of cores and the performance. However, their experiments were restricted to run on sixteen virtual nodes only, which is not highly scalable. Indeed, such a study did not reflect the reality as hundreds if not thousands of cores can be used in production usage. The same scene is repeated in several articles. *Klein et al.* [71] tried to determine how to evaluate NoSQL databases for specific and scalable scenarios of usage. However, their experiments have been run in a cluster with nine nodes at maximum.

2.3.3 *Fair Testing*

Performing fair experimentation or fair testing is a hot topic not only in big data systems but overall in computer science. Indeed, the heterogeneous architectures of big data systems and the enormous number of experimental articles in big data contribute to wondering how to carry fair testing.

Experimental studies that evaluate several systems together may not seriously take the architecture into account during evaluations. Hence, results may have bias and cannot be easily generalized. *Harizopoulos et al.* [53] explained that performing more in-depth evaluation can satisfy the design aspects in evaluation. The authors have put the hybrid systems under the microscope and have discussed the architecture change and its impacts on the way we evaluate systems. Similarly, *Cattell Rick* [22] has performed an architecture study on SQL and NoSQL storage systems to understand their scalability differences and how their scalability feature should be evaluated. The author has confirmed that having different architectures in today's storage architecture prevent to generalize evaluation results and in particular which category scales better than the other. He also concluded this study with a call for building scalability benchmarks that cover different categories of systems.

Coming up with new fine-grained testing metrics can be considered as a potential way that allows for fair testing. Although generic metrics are significant, especially for testing several systems together, they do not take into account the varieties of *SUTs* and what happens in lower layers of executions. *Shi et al.* [136] confirmed that the architecture differences between systems present obstacles for using unified and high-level evaluation metrics. Hence, the authors have divided the evaluation activity into several phases that can be identified in *SUTs* and then correlate between the phases' execution time and resources utilization. This allows to compare different systems under the same category such as *Hadoop*, *Spark*, and *Flink* with giving insights for understanding their performance results. Their correlation metric, which is used for the first time in an experimental article on big data systems, has inspired others to do similar evaluations. For instance, *Marcu et al.* [86] have resued that metric to spot out the differences in performance between *Spark* and *Flink*.

2.3.4 *Data, Resources, and facilities*

Many improvements are proposed to deal with data-related challenges during experimentation. *Li et al.* [77] discussed the idea of creating a full testing framework for big data analytics. A framework that generates a representative set of data to be used in the testing process instead of working on massive data. The main idea of the authors was to compare the datasets before and after each phase of the pipeline stack of big data applications (e.g., extract, transform, and load). Although that proposition is designed for big data analytics experiments, no insights have been given about how such a

framework could be generalized for other categories of big data systems. The second challenge is about data scalability. The direct experimental way to evaluate scalable systems is to rely on scalable data deployed on extended experimental architecture. However, other techniques can allow performing experiments on scalable data without having it and without orchestrating several resources. *Haitsa et al.* [147, 148] contributed a new experimental approach (*CODD*) to evaluate storage systems at scale by allowing to manipulate their metadata. Instead of having one gigabyte of real data, *CODD* allows considering ten gigabytes or even 100 gigabytes via metadata manipulation. This way was a potential direction to target the scalability challenge of big data systems. However, it was limited only to deal with structured data such as *SQL* tables.

The experimentation process often takes a lot of time and effort to adjust the target architecture and resources usage of *SUT*. *Paraiso et al.* [101] contributed a model-driven tool to verify the architecture of containerized *SUT* at the design time. This approach can prevent additional time to fix applications' architecture at run time on the cloud; instead, once the architecture is validated, the tool allows to deploy the application on several cloud providers regarding the experimenters' needs.

Resources are an essential facet to big data experiments. Improving that feature for big data ecosystems is one of literature's direction. *Ma et al.* [84] claimed that proposing resources in-demands to applications is a limitation. Although the actual frameworks of cluster resources management such as *YARN* [149], spark standalone, and *Mesos* [54] follow that strategy, performance overhead raises proportionally if many resources propositions can be made before the acceptance of applications. As an interaction, the authors propose Custody, which postpones the resources allocation for being done after collecting dynamic data about applications' data input in order to enhance the resource allocation process. Before that, *Wang et al.* [153] discussed the issue behind the possible inability of *Hadoop* to scale in the future and to be well-suited for processing fine-grained workloads. One of these issues is the centric-nature of Hadoop's YARN resource manager. The authors proposed using a distributed task execution framework (*MATRIX*) to address this scalability issue. The main feature of the *MATRIX* framework is the delegation of resources managers to manage local resources. Based on this property, the authors did a campaign of experiments to compare between *YARN* and *MATRIX* for distributing Hadoop data. Their experimental results showed that *MATRIX* outperforms *YARN* but by a small speedup factor. Hence, *MATRIX* can be leveraged in big data experiments that include using *Hadoop* ecosystem.

2.4 SUMMARY

Big data systems have been spread to cover almost all fields of numerical applications. Accompanying those systems with improved experimental methods is worthy not only to allow for better deployment but also to cover specific

experimental challenges. Throughout this chapter, we have firstly seen in Section 2.1 that the legacy experimental methods of computer science are still usable with big data systems. However, as big data systems have their particularities and experimental challenges, works are in progress to improve those experimental methods for big data evaluation as it is the case with the benchmarking process (see Section 2.1.2).

We have then described in Section 2.2 the importance of experimental observability and their two primary techniques: monitoring & tracing. Although the high usage of those techniques in experimentation, there are still many challenges to adapt them to big data experiments. For instance, proposing monitoring frameworks that detect, identify, and facilitate the monitoring of experiments rather than mainly working on whole experimental infrastructure is needed. Similarly, works should be done to pave the way towards having precise-objective tracing tools that do not only add no costs in terms of overhead but also being safe and usable for production usage.

At last, we have achieved a literature review in Section 2.3 about the experimental methods in published articles on big data systems. One of the outcomes of that review is the fact that it is scarce to see experimental designs that deal with non-functional requirements, e.g., experimental designs to optimize the usage of experimental resources. Meanwhile, the community still fights to allow having tools that accomplish fair evaluations and coming up with specific metrics and methods to enrich the experimental evaluation of big data systems.

Part II

IMPROVING EXPERIMENTAL CONTEXT

Indeed, current experimental conditions are already less than an order of magnitude away from required conditions

— JEAN DALIBARD

3

HETEROGENEOUS STORAGE PERFORMANCE EMULATION

3.1 INTRODUCTION

Experimental resources are one of the principal actors in the experimental context. Performing experiments on traditional systems is often done by using experimental resources as offered without any modifications since their experimental requirements were simple and straightforward. However, most big data experiments need to run on resources with different performance than actually offered by their experimental environments. Hence, giving big data experiments more autonomy to customize experimental resources in regard to their requirements can be seen as an improvement of experimental context. Therefore, resources customization challenge is discussed in detail here, giving particular attention to storage resources and big data experiments that run on experimental environments like testbeds.

Testbeds are a key asset for experimental research in computer science. They generally provide access to a large number of resources, trying to be representative of the hardware that is or was (or sometimes will be) available. However, despite this variety in resources characteristics (and their performance), experimenters remain limited to what is made available by testbeds, which might not match their needs. For instance, experimenting on a testbed offering high-speed networking and modern storage is not convenient if the target system is supposed to run on commodity hardware with moderate networking performance and rotational storage devices. Provided that, the emulation of resources' performance can come to rescue in order to provide suitable environments for experiments.

Although emulation is widely applied for some resources such as CPUs, networks, and memory during experimentation, emulating I/O performance is not common in the testbeds' community. One could wonder about the behavior of big data systems on different scenarios where the I/O performance varies. However, almost no experimental tool can be used to achieve such experiments. Indeed, the existing I/O emulation tools focus on the proportional sharing of I/O throughput [142] [56] [2], targeting balanced or fair resources allocation rather than the experimentation use cases.

The increasing importance of data makes it crucial to build experimental frameworks that enable emulation of I/O performance, in order to be able to

investigate the I/O behavior of Big Data systems. Firstly, I/O operations are centric for this kind of systems, so testing them at scale under heterogeneous I/O performance can help with revealing potential performance issues, or getting a better understanding of their I/O behaviors. Secondly, emulating the I/O performance creates more testing configurations for experiments and avoids a bias about storage resources. For example, one could tune the performance of an *SSD* to act like an *HDD* in order to determine how the target system reacts accordingly.

In this chapter, we provide an I/O emulation service through extending the *DISTributed systems EMulator (Distem)* [124], which is available on different testbeds [110] such as *Grid'5000*, *CloudLab*, and *Chameleon*. That service allows either statically imposing limitations on the I/O performance on one or several nodes at the start of experiments or changing the I/O performance of involved nodes dynamically over time. In Section 3.2 we briefly describe the *Distem* emulator to state the basic information about the software that will adopt our I/O emulation service. In Section 3.3 we describe how the I/O emulation service is created. That section also includes an experimental investigation on the underlying technology used to build our emulation service (Linux's *control groups* – a.k.a *cgroups*). Indeed, *cgroups* is intensively explored in the literature to achieve process isolation and containerization [99, 100] on the scale of a single node, but not yet examined to emulate the I/O performance for testbed experiments. Section 3.4 describes a series of experiments done over *Hadoop* to highlight the advantage of emulating the I/O performance at scale, configuring nodes with heterogeneous I/O performance. Section 3.5 states the related work on controllability of I/O performance while Section 3.6 summarizes.

3.2 THE DISTEM EMULATOR

Distem[1] is an open source framework that enables experimenters to build customized virtual experimental environments over physical infrastructure. It leverages the *Linux Containers* technology which isolates the namespaces of many types of resources such as memories, CPUs and network interfaces to deliver its services. *Distem* allows the creation of *Vrtual Nodes (vnodes)* over *Physical Nodes (pnodes)* regarding the needs of experiments. Even if experimenting on an environment with a limited number of nodes, *Distem* allows to increase the number of virtual nodes[2] compared with the underlying infrastructure. That virtual scalability is useful for experiments that do not have constraints on the available resources in each *vnode*. For example, testing the scalability in terms of the number of independent processes/*daemons* of the system under test.

Distem controls the experimental environment by applying a server/slave communication model as it selects a physical node to be a coordinator. This

1 http://distem.gforge.inria.fr

2 The number of *vnodes* over a physical node is limited by the number of CPUs of that physical node

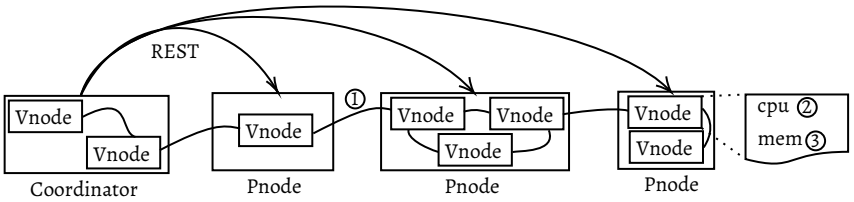


Figure 3.1: An experimental environment created by the *Distem* emulator. It consists of eight virtual nodes (*vnodes*) on top of four physical nodes connected using a *REST API*. Experimenters can emulate the network performance, the CPU and the memory performance for any virtual node or all of them

node communicates with the rest of the physical nodes via a *Representational State Transfer* (*REST*) *API* since every physical node hosts a *Distem* daemon. That coordinator node can be considered as a proxy as it receives the experimenter's requests and then forwards them to the corresponding *physical nodes*. The physical nodes, in turn, communicate with their internal virtual nodes to execute the experimenters' commands (second layer of communication in *Distem*). Figure 3.1 shows an experimental environment created by the *Distem* emulator.

Alternatives like *Mininet* [145] allow to create an experimental environment in a similar way as *Distem*. However, *Mininet* is limited to single-machine deployments (even if prototypes for multi-node Mininet exist), and thus does not scale to large experiments like Distem does. Also, it mainly focuses on networking experiments. Since scalability is a typical goal of testbed I/O experiments, *Distem* is more suitable to be extended for emulating I/O performance, thanks to its native distributed architecture.

3.2.1 *Emulation Services in Distem*

Creating a suitable experimental environment is not the only service offered by *Distem*. The emulator also gives experimenters the possibility to customize the environment by emulating network CPU, and memory performance. The experimental gain of controlling these resources is not trivial. It is useful to increase the number of experimental configurations that can be obtained over the same infrastructure. Concerning the network, *Distem* uses *Virtual Ethernet Device – veth* to emulate the performance. *veth* does not only allow to exchange the network packets between the *vnode*s and *pnode*s, it also allows to specify the target network interface when the *vnode* has several ones. Moreover, the virtual nodes on a particular physical node are connected, which means that those *vnode*s can still do their work when the communication link between *Distem* and that particular physical node is down.

In parallel, the emulator provides two methods for controlling the performance of CPUs assigned to a given *vnode*. It either modifies the frequencies of the target CPUs to work as required, or it stuffs some real processes to consume a precise amount of CPU time, so the *vnode* consumes the com-

plementary as configured. Moreover, *Distem* deals with the size of memory instead of performance when it comes to emulating the memory.

3.3 CONTROLLABILITY OF I/O PERFORMANCE

Every operating system has its own mechanism to access storage devices. Limiting the discourse to the I/O operations on Linux, this is done through dividing the I/O operations to target block of data via sequences of I/O requests. Linux kernel hosts all the components in charge of treating I/O operations. For example, the memory management system, and several layers of the I/O stack such as the filesystem, schedulers/elevators, and block I/Os. As a result, the Linux kernel is the suitable place for any I/O emulation approach to be implemented.

Controlling the I/O performance implies having a mechanism that alters the I/O throughput for a single or several I/O processes accessing data on storage devices, either for the purpose of writing or retrieving data. This mechanism must be generic, i.e., it must be feasible for every single I/O operation regardless of whatever the path on the I/O stack it takes. Therefore, this section begins by describing the abstracted mechanisms of accessing data on Linux. It describes our experimental investigation over *cgroups* technology as serve as a basis for our emulation service (see Section 3.3.2). After that, Section 3.3.3 describes how the *Distem* emulator adopts our I/O emulation service.

3.3.1 *Mechanisms of Data Access*

The memory is involved in the management of I/O operations. Data could be written to the *page cache*[3] instead of being committed to the storage devices immediately (e.g., using asynchronous I/Os). Similarly, data that was already read previously is kept in the *page cache*, resulting in much faster access in subsequent reads. Given that, when emulating I/O performance, it is essential to control the memory size of target machines, thus controlling the size of the *page cache*. The memory limitation, in turn, covers two main cases that possibly happen during the I/O emulation:

1. The interactions between the target applications and the memory during the execution of their I/O operations
2. The kernel's *write-back I/Os* used for writing the data permanently into disks (more details in the following paragraphs)

Any I/O operation has at least two main actors: the I/O process which initiates that operation and the Linux kernel which treats them. I/O processes select a suitable *system call* to deliver their I/O requests to the Linux kernel. The most-known *system calls* are *read* & *write* with many variations such as

3 Defined in Wikipedia as a transparent cache for the pages originating from a secondary storage device such as a hard disk drive (HDD) or a solid-state drive (SSD)

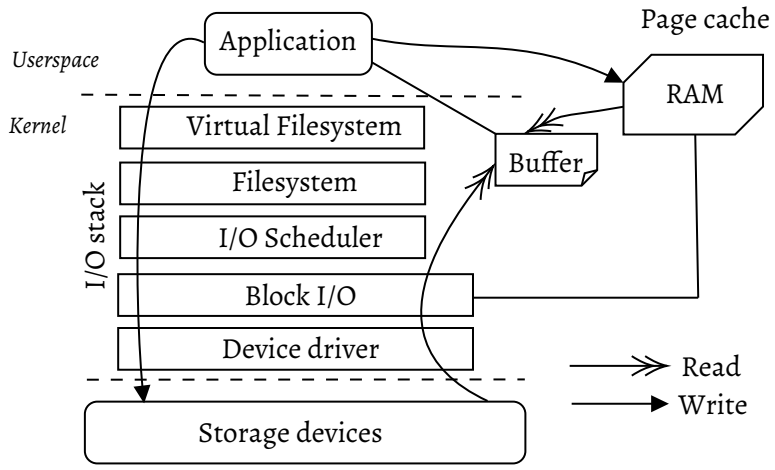


Figure 3.2: Data communication approaches over the I/O stack of Linux

pread/pwrite, *readv/writev*, etc. These *system calls* determines the I/O method that the application follows (e.g., sync I/O & vectored I/O), and the way that the kernel treats them. However, using any of them leads to one of four data communication cases. These cases are described as follows, regarding the I/O emulation context.

- READ-FROM-DISK. A straightforward operation where the *userspace* process initiates its request and communicates directly with the I/O device. The request typically passes through the layers of the I/O stack (i.e., *virtual filesystem – VFS*, *filesystem*, *I/O Scheduler*, *block I/O*, and *device driver*) to finally reach the target storage device. Of course, that happens after checking out that the target data is not yet loaded into the *page cache*. The operation starts at the time of issuing the request, and it ends when the corresponding process is notified about the termination of data retrieval. This operation is repeated as many times as the target process requires to read disk's data. From the viewpoint of emulation, it is easy to emulate the I/O operations that follow this way of communication. Since the memory does not have any impact on this operation, it is sufficient to inject a desired I/O throughput value at the level of the block I/O layer to emulate the I/O performance.

- READ-FROM-CACHE. When the I/O requests arrive at the *virtual file system* layer on the I/O stack, the *VFS* checks if the data is already available in memory. In such a case, the *VFS* avoids further communication with lower layers of the I/O stack and retrieves the data from memory. In this case, injecting an I/O throughput value at the level of the block I/O layer has no sense as the I/O requests are served before reaching that layer. Hence, applying limitations on the size of the *page cache* of the target process is crucial in such a situation. Indeed, having an appropriate *page cache* size in regard to the desired I/O performance should allow to altering the I/O throughput. However, the I/O operation should pro-

ceed at the normal memory speed i.e., without any slowdown caused by emulation.

- WRITE-TO-DISK-SYNCHRONOUSLY. In this case, I/O requests pass through the I/O stack layers until reaching the appropriate storage device. The process that initiates the I/O requests should be notified about the termination of the current I/O request before initiating the next one. indeed, I/O requests are supposed to bring feedback to the corresponding process when the data is written permanently into the storage device. Like the *read-from-disk* case, this is a direct operation where the block I/O layer is the suitable place to adjust the I/O throughput value.
- WRITE-TO-CACHE. Writing data asynchronously can lead to activate this option, no matter which *system call* is used. Although this can lead to lost data if the machine is shut down, it accelerates the I/O activities of the *userspace* process. The I/O requests write the data to the *page cache* and return immediately. Then, the kernel notifies the corresponding process in *userspace* that the I/O operation is terminated while the data is still in memory and not yet written into the storage device. Asynchronously, the kernel will then proceed with the *write-back* operations in order to transfer the data from the *page cache* into the storage device. This step represents a challenge in case of measuring the I/O performance since the kernel writes the data into the storage support behind the scenes.

The read and write throughputs should be distinguished when controlling the I/O performance. This granularity provides the basis to separate the study of reading and writing behaviors for target applications, as well as evaluating the influence of read-optimized or write-optimized storage devices. Users should be able to limit the read throughput, the write throughput or both according to the needs of experiments.

In the next section, we study the feasibility of *cgroups* to cover all the beforementioned data communication cases, while being used to emulate the I/O performance for testbed experiments.

3.3.2 *I/O Performance Emulation with cgroups*

Linux *cgroups* is a technology used at the scale of a single machine to control that machine's resources. The key idea of *cgroups* is to isolate target processes into one or several groups and then provides mechanisms to limit and measure their resources usage. Therefore, emulation in this context does not refer to emulating a real resource inside the isolated groups of processes, but rather a manipulation of the usage domain and ratio of a given resource. To achieve its goals, *cgroups* provides many controllers (subsystems) to control specific types of resources, i.e., each resource interface is accessed via a separate controller. To name a few: *ns – the namespace controller, cpu – to generate general reports on cpu usage, cpuset – to do cpu assignements,* and others. In this

section, we only discuss the *blkio – block I/O controller* and *memory – the memory controller* as these controllers[4] are the highly involved in the context of storage performance emulation.

3.3.2.1 *cgroupV1*

cgroupV1, which appeared in Linux v2.6.24, is hierarchical where every resource has its own hierarchy. Cgroups can then be added inside the proper hierarchy. For instance, Limiting the memory size of a given process can be tackled by the following procedures. First, a *cgroup* with whatever name should be placed in the hierarchy of the memory resource. Then, the target process should be added via its *process identifier* to that group, before using the appropriate interface command to alter the memory size of that group (limitation and accounting are performed per cgroup rather than per process). To summarize, no more than one controller can be activated on a given group.

During the I/O performance emulation, we need to activate both controllers (*memcg* and *blkio*) to cover all the data communication cases described previously. However, we expect that emulating the I/O performance is not feasible in *cgroupV1* due to its separated hierarchies of resources. Although adding the same target process to different resources hierarchies (here, *memcg* and *blkio*) is possible, this does not indicate any coordination between the controllers when the process access disk's data. Indeed, the limitations set by the *memcg* controller for the *page cache* management may be ignored: instead, the whole memory of the host system might be used for the *page cache*, resulting in performance that is higher than expected. In our expectations, this leads to wrong measurements (higher I/O throughput) for memory-dependent I/O workloads. In contrast, using *cgroupv1* may still work perfectly in case of direct I/O operations as they bypass the *page cache*.

3.3.2.2 *cgroupV2*

cgroupV2, included in Linux v4.5, addresses the drawback of its predecessor by having a more flexible structure. Groups are not any more enrolled under several hierarchies. Indeed, several controllers can be assigned over the same group. Figure 3.3 shows the difference between both versions through an example.

The advantage of using *cgroupv2* in the I/O emulation context is that the memory limitations become active as the memory controller works together with the block I/O controller on the same group, adjusting the *page cache* size of the target group. On one side, the block I/O controller assigns an *inode* number to the group that possesses the process which sends I/O requests. On the other side, the memory controller applies its limitation while processing the *dirty pages* of that group. This also covers the *write-back* operations performed by the kernel as the responsibility of *write-backs* is charged to the group in which the target I/O process is attached. Hence, *cgroupV2* may lead to obtaining correct measurements of I/O throughputs for all data access mechanisms.

4 We refer to the memory controller as *memcg*, and I/O controller as *blkio* in the rest of sections

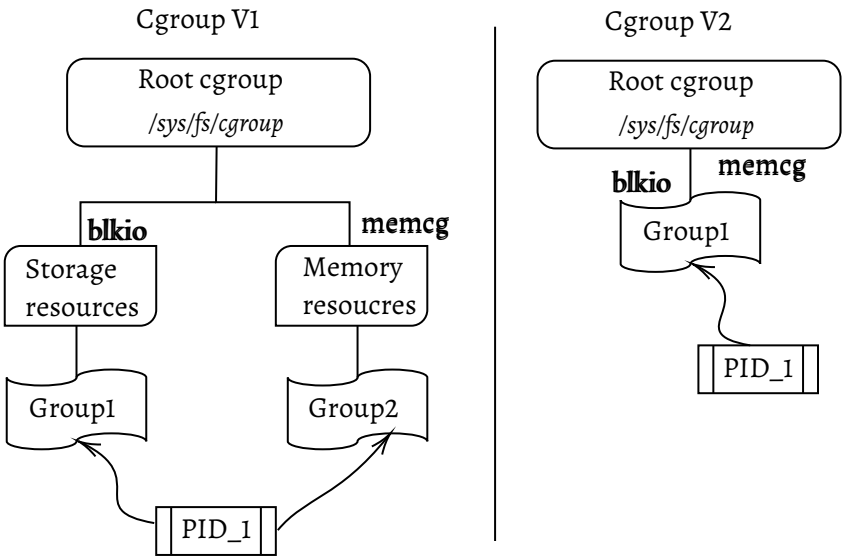


Figure 3.3: An illustrative schema about adding an I/O process with PID_1 to both *cgroup* versions, to emulate its I/O performance

Technically, the support of *cgroup*'s *write-back* is limited to certain *filesystem* such as *ext2*, *ext3*, *ext4*, and *btrfs*. On other *filesystems*, *write-back* I/Os are still assigned to the root *cgroup*. However, this is not a problem in practice as other filesystems are not interesting in our use case.

In the section, we evaluate both versions of *cgroups* to determine if they are usable for storage performance emulation, by applying various workloads and limitations and examining the resulting behaviors.

3.3.2.3 *Experimental Validation*

The *Flexible I/O Tester* (*Fio*) – (*Fio* v3.12) is used to cover all data communication cases described in Section 3.3.1. It provides several *IOengines*, each stressing different set of I/O *system calls* (e.g., *sync IOengine* uses *read &write Linux System Calls* (*syscalls*) while *Posixaio* covers *aio_read* and *aio_write system calls*). Our experiments apply a set of I/O limitations following a full-factorial design of five experimental factors:

- I/O methods with the following *Fio's IOengines* as values: *Sync, Psync, Pvsync, Pvsync2, Posixaio,* and *Mmap*
- Type of workload including *read, write, randread, randwrite, readwrite,* and *randreadwrite*
- I/O accessibility with direct and buffered I/Os as values
- Storage device (*HDD* and *SSD*)
- A set of files with different sizes, varying from 10 MB to 2 GB

We apply two memory limitations (1 GB for *SSD* experiments and 128 MB *HDD* experiments). We expect that these values are compatible with file sizes

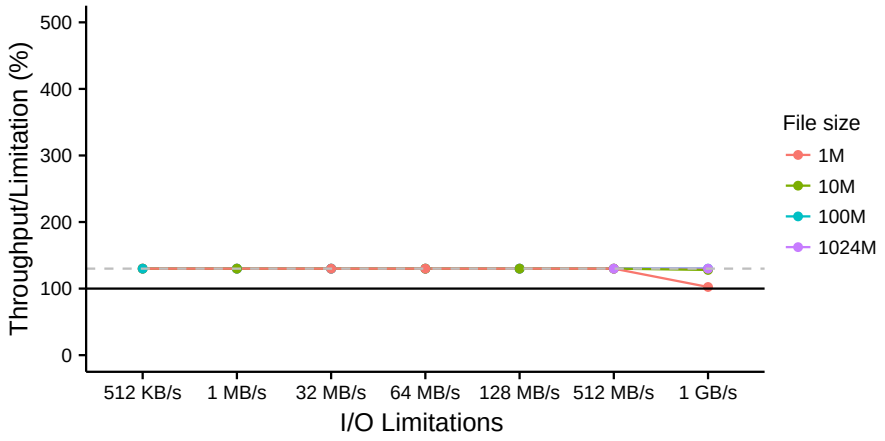


Figure 3.4: Ratio of measured vs defined I/O throughput for a buffered write workload over an HDD, using *Fio*'s *sync IOengine*

as we use different sizes for each storage device. Regarding the storage devices that are used in these experiments, their reference performance is as follows: 197 MB/s & 710 kB/s as direct read and write I/O throughputs of *HDD*, while the corresponding *SSD* performance is 119 MB/s & 108 MB/s (measurements obtained using *Fio*). The write bandwidth on HDD might seem low but is expected since *Fio* performs random writes. The *SSD* performance seems not very high compared with the *HDD* performance, but this is due to the low-default block size applied by *Fio* during tests (4 kB) and concerns only direct I/Os. At last, the experiments are run on a machine hosts *Debian 9* with *Linux 4.9.0*.

RESULTS

The results of the experiments are exhibited in a way that indicates if the I/O limitations (desired I/O throughput) are considered or not. Hence, all figures in this section represent results in percentage the ratio of obtained I/O throughputs to the used I/O limitations. Indeed, quantifying the I/O throughput indicates to what extent the used limitations are considered. Additionally, the dashed line in all figures is set to 130% to represent the bandwidth measurements that are too high. If that line is reached, this indicates that the corresponding limitation is not applied.

As expected, the obtained results of *cgroupV1* are not coherent regarding the applied I/O limitations, especially for the buffered I/Os, for the reasons mentioned above in the previous section. For example, Figure 3.4 shows the results buffered writing using sync *IOengine*. In that figure, that desired I/O throughputs are not considered since all points are located over the dashed line except one point. That point indicates that the smallest file (1 MB) is written at a rate of 1 GB/s. Although the desired throughput is admired here, it might be an exception for small files where they can be written at once to the host memory.

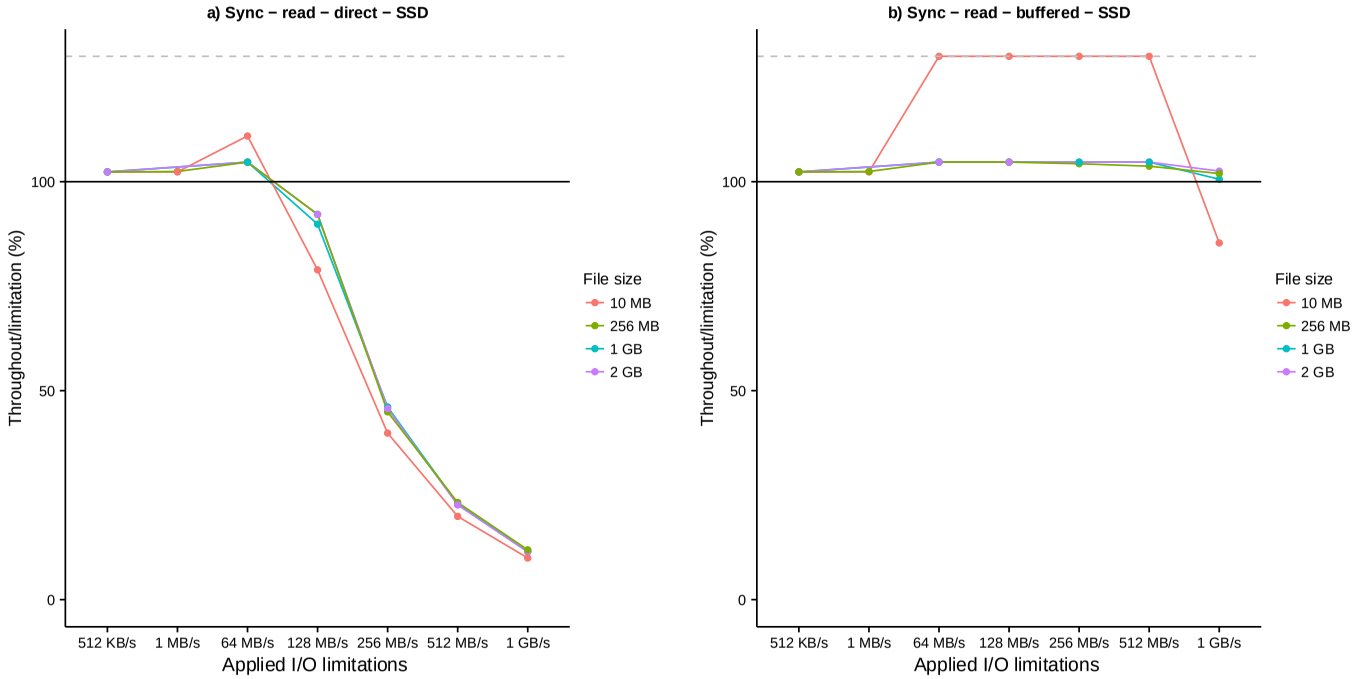


Figure 3.5: Ratio of measured vs defined I/O throughput for a) direct & b) buffered read workload over an *SSD*, using *Fio*'s *sync IOengine*

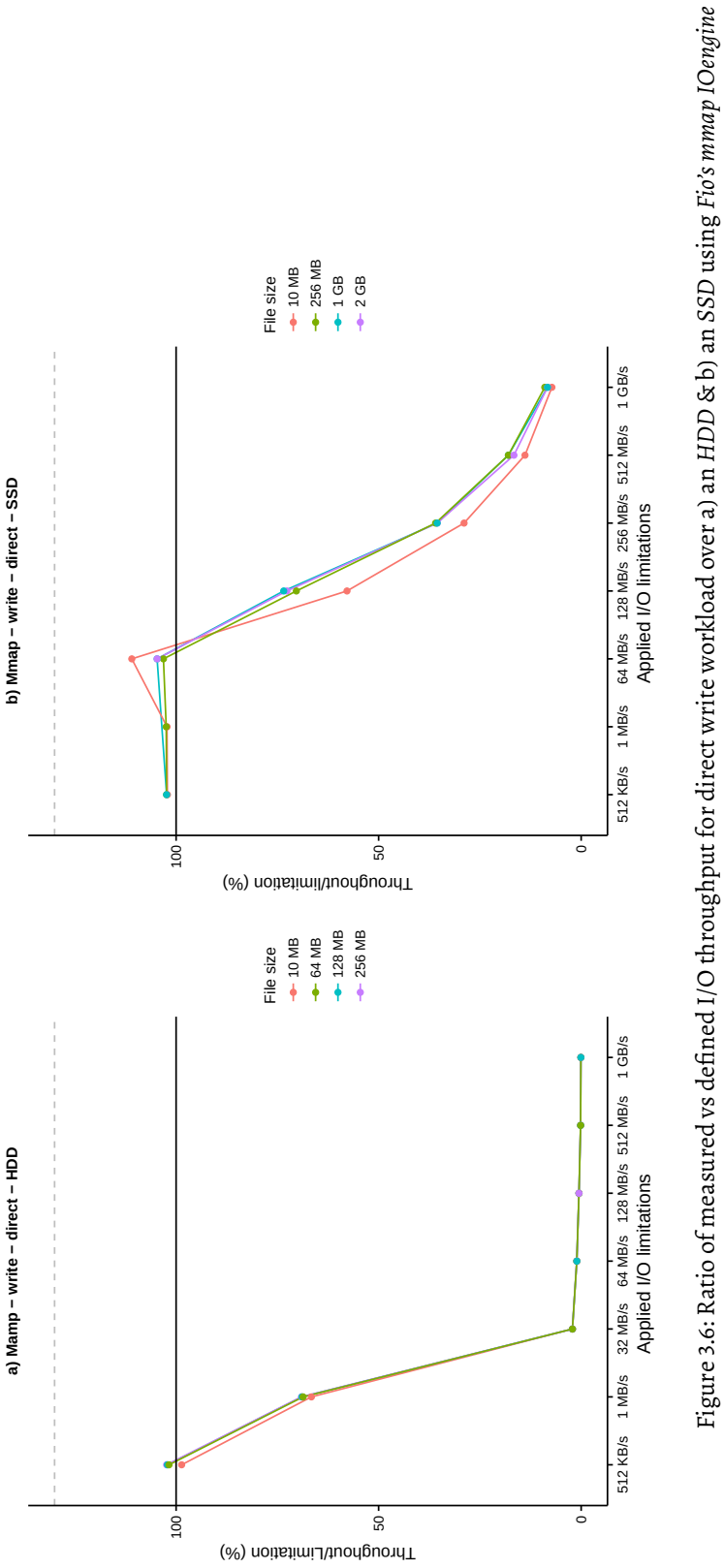


Figure 3.6: Ratio of measured vs defined I/O throughput for direct write workload over a) an *HDD* & b) an *SSD* using *Fio's mmap IOengine*

In contrast, results of *cgroupV2* are more consistent. Among a large set of results, we discuss here two experiments that cover multiple experimental factors. The conclusions is by the way applicable to the rest of the results that can be found in the thesis repository. Figure 3.5 and Figure 3.6 show the results of the selected experiments performed using cgroupV2.

Figure 3.5 compares the direct and buffered I/Os on an *SSD*. In case of direct I/Os, one can see that all I/O limitations are efficient. The emulated I/O throughputs are respected for the points of 512 kB/s, 1 MB/s, and 64 MB/s included. However, the rest of I/O limitations achieves lower I/O throughputs since they are limited by the storage device's performance (119 MB/s in this case). Figure 3.5-b shows that *cgroupV2* is capable to emulate the I/O performance for buffered I/Os. It shows that the emulated I/O performance corresponds to the measured I/O performance for almost all tested files, even for files that fit into memory. For instance, reading the 256 MB file with a rate of 1 GB/s reports a correct measurement near 1 GB/s despite of the fact that the file is smaller than the applied memory (1 GB). However, the result of the 10 MB file shows that it is read at once, ignoring the defined I/O limitations. Hence, the two files (10 MB & 256 MB) are not similarly treated, showing sophisticated caching behaviors which requires more investigation.

Figure 3.6 shows results when performing direct write I/Os using *mmap* over both storage devices. It indicates that the performance of storage device influences the applied I/O limitations. In the *HDD* experiment, the measured throughput decreases starting from the point of 1 MB/s (*HDD* direct write throughput is 710 kB/s). Similarly, the I/O throughput of *SSD* experiments decreases starting from the point of 128 MB/s which is very close to the *SSD* performance (108 MB/s).

3.3.3 *Extending Distem to Support I/O Emulation*

According to the results shown in the previous section, we extended *Distem* to leverage the potential of *cgroupV2*. Integrating this into Distem makes it possible to go beyond controlling the resources of a single machine. It allows emulating the I/O performance for dozens to thousands of experimental nodes simultaneously, creating more testing configurations for large-scale systems and emulating the end-environment performances.

Technically, *Distem* uses the parameter *disk_throttling* of the *vfilesystem* resource to control the I/O throughput. Beside of specifying the target *vnode*, this parameter requires the desired values of *read* & *write* I/O throughput or both, and on which storage device the control should be applied. Concerning the memory limitation, it can be applied using the resource *vmem* which represents the memory space of the corresponding virtual node over any physical node. hence, two modes can be used in regard to the experiments' requirements: either *soft_limit* which allows exceeding the memory limit without killing the target process, or *hard_limit* which kills the processes that exceed the specified amount of *Random Access Memory (RAM)*.

Distem provides two methods to emulate the experiments' I/O performance. These methods are shown below.

- STATIC EMULATION. Users can emulate the I/O performance of all experimental *vnodes* or a subset of them at any moment during the experiment timeline. This can be done by sending a request using the *Distem client* interface. This emulation mode is useful to study the behaviors of the target systems under many scenarios. For instance, creating heterogeneous experimental environment where each *vnode* can have different I/O throughput during the experiment.

- DYNAMIC EMULATION, BASED ON TIME EVENTS. This emulation mode allows users to achieve several changes of I/O performance during the same experiment. Users can define several time points to alter the I/O performance of the selected *vnodes* automatically. This emulation mode is useful when emulating a performance degradation over fine-grained intervals or switching the I/O performance between *vnodes*. However, it is up to users to choose the time points to stress the system during the experiment. Hence, knowing the system's behaviors in advance is required in order to decide when to alter the performance.

An example that illustrates the advantages of dynamic emulation mode is load balancing experiments. For instance, suppose that we have a system installed on two nodes (vnode1 & vnode2), each holds the same data set. If the reading throughput of a *vnode1* is degraded at *t1* and the writing throughput of the *vnode2* is increased at the same time, and then both increased to their maximum values at *t2*, this could help to investigate if the target system is dynamically balanced or not. Ideally in regard to balancing the load between the nodes, the system should send more reading and writing requests to the *vnode2* between the time points *t1* and *t2*. The following pseudo-code explains how *Distem* configures the *vnodes* for this example.

Listing 3.1: Emulating the I/O performance of two virtual nodes, following the time events approach of the *Distem* emulator

```
require 'distem'
Distem.client do |cl|
cl.vfilesystem_update('vnode1,vnode2', {'disk_throttling' => {'
    hierarchy' => 'v2'}})

cl.event_trace_add({'vnodename' => 'vnode1', 'type' => '
    read_limit', 'device' => '/dev/sda'},
'disk_throttling', { t1 => 10 MB/s, t2 => 'max'})

cl.event_trace_add({'vnodename' => 'vnode2', 'type' => '
    write_limit', 'device' => '/dev/sda'},
'disk_throttling', { t1 => 10 MB/s, t2 => 'max'})
cl.event_manager_start
```

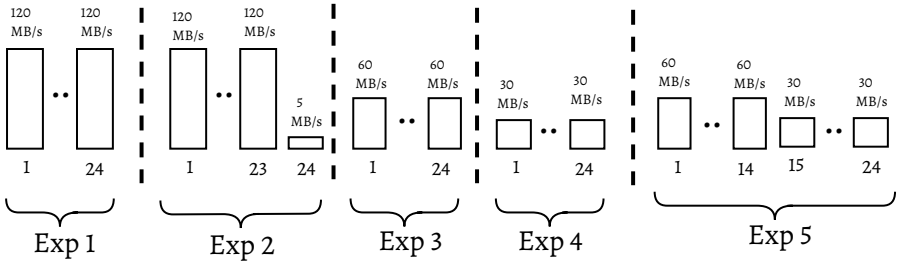


Figure 3.7: Design of five experiments that shows the distribution of I/O throughput between a cluster of twenty-four *datanodes* of Hadoop

3.4 USE CASE STUDY: HETEROGENEITY OF STORAGE PERFORMANCE IN HADOOP EXPERIMENTS

This section describes a series of experiments done on the *Hadoop* framework to highlight the usefulness of emulating the I/O performance for testbed experiments. It starts by describing the experiments' goals and setup before presenting the obtained results.

3.4.1 *Experiments' Hypothesis*

The goal of these experiments is twofold. First, they aim at determining the behaviors of *Hadoop* after several changes in the I/O performance of one or several nodes of the *Hadoop* cluster. Second, they also illustrate how to mitigate the impacts of testbeds' storage resources over results. Indeed, if the storage performance is emulated, experiments' results should be comparable despite the nature of the storage devices in use: *HDDs* or *SSDs*.

In general, big data systems such as *Hadoop* are susceptible to CPU and memory resources while scheduling their map-reduce tasks. However, we are not sure if they take into account the I/O performance of their nodes during jobs' execution. Provided that, five experiments are designed to spot *Hadoop* behaviors, in order to validate this hypothesis (see Figure 3.7). We expect that *Hadoop* will be influenced by emulating the I/O performance of its nodes, incurring a delayed execution time according to the applied I/O limitations. Our expectations are formulated in terms of execution time and I/O throughput for each experiment separately. To define our experiments, we suppose that 1) n is the number of *datanodes* in a *Hadoop* cluster, 2) the initial value of the read and write I/O throughput is 120 MB/s for each *datanode*, and 3) each *datanode* has d GB of data to be read/written regarding the workload. Table 3.1 describes the defined experiments and their expected results.

3.4.2 *Experimental Setup*

The experiments are performed on the *Grid'5000* testbed [9]. Twenty-five machines are used as infrastructure. Each machine is equipped with two Intel Xeon E5-2630 V3 CPUs (8 cores/CPU), 128 GB of RAM, and a 10 GB/s

Table 3.1: Description of *Hadoop*'s experiments and their expected results in terms of throughput and execution time

Experiments	Description	Expected results
Exp 1	I/O read and write throughputs are set up to 120 MB/s for all datanodes. Ref. experiment	$t_{exec} = t1, th_r = d \times n/t1$
Exp 2	I/O read and write throughputs are set up to 5 MB/s for one data node	$t_{exec} = t_2, th_r = d/t2$
Exp 3	I/O read and write throughputs of all datanodes are limited to 60 MB/s	$t_{exec} = t_3, th_r = (d \times n/t3)$
Exp 4	I/O read and write throughputs of all datanodes are limited to 30 MB/s	$t_{exec} = t_4, th_r = (d \times n/t4)$
Exp 5	I/O read and write throughputs of half nodes are limited to 60 MB/s, the other half to 30 MB/s	$t_{exec} = t_5$ while $t3 < t5 < t4$

Table 3.2: Averages Results of *Hadoop*'s experiments' in terms of execution time

	Exp 1	*Exp 2*	*Exp 3*	*Exp 4*	*Exp 5*
HDD	8 min	21.10 min	24.3 min	84 min	51.3 min
SSD	4.5 min	19.6 min	20.6 min	91 min	47 min

Ethernet interface. Also, every node has a 186 GB *SSD* and a 558 GB *HDD* connected as a *JBOD*. Debian 9 is deployed on top of these nodes with *ext4* as a *filesystem* and *CFQ* as an I/O scheduler. The *Distem* emulator is then used to deploy its virtual nodes (*vnodes*) over those 25 physical nodes with one-to-one mapping, i.e., one *vnode* over a physical node. This mapping simplifies the deployment process since it does not consider a local resource sharing of I/O performance.

Hadoop v2.8.4 is installed on the *vnodes* with *YARN* as its default map-reduce framework. A cluster of one *namenode* and twenty-four *datanodes* is created on top of the *Distem* environment. All *Hadoop*'s default configurations are maintained except the number of mappers and reducers per job. All *datanodes* are configured to have at maximum one mapper per job. Indeed, we do not allow to have two mappers per job, in order to simplify the throughput calculation process, ensuring that all nodes are busy in a given time during the experiment. Additionally, only half of the nodes are allowed to have one reducer at maximum, i.e., half of the nodes are not allowed to perform reduce tasks. Otherwise, it is not possible to control where to put each reducer as this depends on the resources availability and the internal algorithm of *YARN*. For instance, if a *datanode* is configured to have a moderate I/O throughput and the last job's reducer is to be run on that *datanode*, this leads to non-reproducible results since it is not guaranteed that this reducer will always run on that *datanode*.

The *Terasort* benchmark is used as the main workload for this experiment. Before each execution, the benchmark's data generator is firstly used to generate 5 GB of data for each *datanode*, resulting to have 120 GB as a total data for the cluster. *Terasort*'s map-reduce job is then executed on the generated data, and the execution time of experiments is calculated as the time elapsed between the command execution and its termination. *Hadoop* is restarted between the executions and the cache is cleaned out too. Finally, each experiment is performed ten times to increase results reliability.

3.4.3 *Results*

Figure 3.8 and Table 3.2 provide execution time of all experiments on *HDDs* and *SSDs*. The results show that the execution time changes regarding the emulated I/O throughput in each experiment. The time is always stated as the time of the *datanode(s)* with the lowest I/O throughput. For instance, the second experiment has one *datanode* with an emulated I/O throughput of

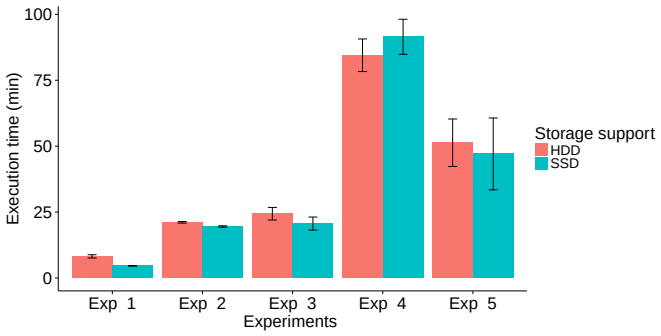


Figure 3.8: Averages Results of *Hadoop*'s experiments' in terms of execution time. Applying the same I/O limitations on both storage types produces comparable results.

5 MB/s while the I/O throughput of the rest of *datanodes* is 120 MB/s. The obtained execution time of this experiment when performed on *HDDs* is 21.10 min. This value corresponds to the execution time of the *datanode* with the moderate I/O throughput. We verify that by calculating the approximate throughput from the obtained execution time. We obtain 4 MB/*s* which is very close to the applied throughput on the limited *datanode* (5 MB/s). Similarly, this analysis is correct even in case of having several classes of emulated I/O throughputs. For example, the execution time of the fifth *SSD* experiment (12 *datanodes* with an I/O throughput of 60 MB/s & 12 *datanodes* with an I/O throughput of 30 MB/s) is influenced by the group of nodes with the lower I/O performance. The overall throughput of that experiment is 21.7 MB/*s*, which is relatively close to 30 MB/*s*.

One can see in Figure 3.8 that performance varies similarly for both storage types thanks to the defined I/O emulation. However, there are still some differences between *SSDs* and *HDDs*, especially for the first experiment. We suspected that this is either due to 1) the fact that the applied I/O throughput (120 MB/s) is close to the peak performance of those HDDs (measured at 133 MB/s); 2) the fact that I/O emulation only affects the throughput of the storage device, but not its latency, and that SSDs provide lower latency than HDDs. The same figure also shows that the variability is increased between repetitions of the fourth and fifth experiments. We suspected that this is due to different decisions being made by *YARN* during the scheduling of the last reducer job. This reducer may run early if a *datanode* becomes available or it may be executed at later stages with delayed execution time.

3.5 RELATED WORK

Several studies tried to manage the I/O performance for enhancing the *QoS* of I/O applications. They use either *cgroups* or other methods to do so. Using *cgroups*, Suk *et al.* [142] allocate the I/O dynamically based on the needs of virtual machines. Their tool provides feedback to *cgroups*, which in turn, uses that information to alter the I/O throughput accordingly. However, their tool cannot be used to apply I/O limitations, and it is not distributed to manage the

I/O throughputs over several machines. Huang *et al.* [56] contributes an I/O regulator to fairly allocate the I/O throughput for big data frameworks. They claimed that *cgroups* does that unfairly, so the I/O regulator measures the I/O load of *datanodes* to adjust the *IOPS*. However, the I/O regulator is a Hadoop-specific solution. Also, it cannot be used to apply I/O limitations. Ahn *et al.* [2] proposed a weight-based allocation for I/O throughput. Their solution uses a dynamic I/O throttling based on a prediction of future I/Os of containers. However, it is enhanced for the proportional sharing of throughput and not for applying I/O limitations.

Other studies proposed I/O schedulers to improve the *QoS* of I/O applications. Zhang *et al.* [159, 160] proposed an interposed I/O scheduling framework (*AVATAR*) to guarantee a proportional sharing of I/O throughput. *AVATAR* does not scale for several machine usage as it focuses on managing the I/Os for local and concurrent processes. Xu *et al.* [157] proposed a scheduler for *HDFS* to control the I/O throughput for map-reduce jobs internally. However, their solution is Hadoop-specific. Despite the scheduler's ability to work on several *HDFS* nodes, it manages the nodes' I/Os locally, in isolation.

To summarize, no experimental framework was proposed to emulate the I/O performance of testbed experiments. Almost all described frameworks are designed to share the I/O throughput fairly between I/O processes and are unsuited to cases where I/O limitations should be applied differently from one node to another, to perform evaluations under like-end environment circumstances.

3.6 SUMMARY

Together with the CPU and the network, storage plays an essential role in the overall performance of applications, especially in the context of big data applications, that can deal with enormous datasets. However, testbeds have scarce support for experiments involving storage performance. Experimenters can use the storage devices provided on the testbed directly, but this can be seen as an issue from experimentation viewpoint. The direct usage of storage resources might not provide suitable performance characteristics for experiments; it might also leave an uncontrolled bias over the experiments' results. Thus, forcing experiments to follow such usage may threaten the experiments' conclusions.

In this chapter, we explored the idea of customizing the I/O performance to overcome the above-described issues, giving experimenters more control over testbed resources, and opening new directions to evaluate under suitable testing configurations. We demonstrated through a series of validation experiments that Linux's *cgroups* is suitable for emulating the I/O performance for the dominant I/O workloads. That technology is then implemented in the *Distem* emulator to provide an I/O emulation service at scale. Indeed, *Distem* is selected to adopt our technology since it already proposes to emulate other resources like CPU and network, so having a full emulator usable on several testbeds was a secondary objective here. The advantage of emulating

the experiments' I/O performance is highlighted via a *Hadoop* map-reduce experiments. The experiments show that the performance of *Hadoop* varies under heterogeneous I/O throughput configurations. They also confirm the bias-free of storage devices as *HDDs* & *SSDs* are interchangeable in certain conditions (e.g. a desired emulation point is below both devices capabilities).

Part III

IMPROVING EXPERIMENTAL OBSERVABILITY

Experiment is fundamentally
only induced observation

— Claude Bernard

4 MICRO-OBSERVABILITY WITH IOSCOPE

4.1 INTRODUCTION

Storage systems become complex to keep pace with the requirements of big data. The current way of evaluating storage systems, especially the data stores, has not changed adequately. It still focuses on high-level metrics which completely ignores potential issues in lower layers of execution. For instance, studies like [1, 45, 65, 71] use *YCSB* [33] benchmark. The core evaluation metrics of such a tool are limited to workloads' throughput and execution time. One would know more why a given system achieves modest or strange results, but unfortunately, such metrics cannot explain I/O performance. They only give indications that something goes wrong. Therefore, in-depth observability such as evaluating I/O activities and interactions in lower layers are required. This leads to explain high-level measurements and to examine production workloads. Thus, our main concern here is to analyze how data files are accessed during workloads' execution, investigating if such activity hides potential I/O issues.

Henceforth, we define workloads' I/O pattern as the order of files' offsets targeted by I/O requests during accessing on-disk data. Potential I/O issues may appear due to various reasons. For instance, reordering I/O requests in lower layers of I/O stack or a content distribution issue in the applicative layers may transform sequential access into random access or vice versa. However, analyzing the I/O pattern is less practiced during systems' evaluations for two reasons. Firstly, there is a lack of specific tools to directly analyze in-production I/O workloads. Secondly, this is considered as an internal testing procedure which is often faced by a convention that all storage systems are well tested internally.

Uncovering the I/O patterns involves dealing with lower levels of applications as well as different kernel interfaces. Constructing a general tool to do so is complicated since such a tool must deal with multiple and heterogeneous I/O methods. Moreover, the complexity will be doubled when targeting in-production systems as such a tool needs to have a negligible overhead and safe behaviors on the production environment. From this angle, the main feature of an observability tool is to run at lower layers of execution such as inside the kernel for several reasons. Firstly, doing so will bring more detailed results by exposing low-level kernel data, and secondly, this leads to having a negligible overhead when compared with tools running in userspace.

Tracing is highly involved in observing storage systems [150]. However, tracing tools often collect generic I/O traces from several layers of I/O stack. This leads not only to incur high overheads regarding a large number of diversified interceptions inside and outside the Linux kernel but also to generate a large quantity of tracing files that need huge effort for being analyzed. Of course, we can find multiple tracing tools that are specific enough, performing tracing over well-defined targets and collecting precise data. However, they partially cover the dominant data access methods the storage systems use for issuing their I/O workloads.

In this chapter, we present *IOscope* toolkit. *IOscope* applies both – tracing and filtering – techniques to address the before-mentioned limitations by generating specific and ready-to-visualize traces about workloads' I/O patterns. *IOscope* relays on *extended Berkeley Packet Filter (eBPF)* which incurs negligible overhead[1] [133], and covers the dominant methods of issuing I/O workloads such as *sync I/O*, *async I/O* and *mmap I/O*. Its design and experimental validation are shown in Section 4.2. In Section 4.3, we describe original use case study over *MongoDB* and *Cassandra* databases, over a precise workload: secondary indexation. The goal behind such use case study is to reveal potential performance issues related to I/O patterns. *IOscope* was able to discover a pattern-related issue in clustered *MongoDB* which is behind the performance variability of experiments (see Section 4.3.2). Of course, that section also describes an *ad hoc* solution to overcome. In parallel, the variability of performance results was not only observed on *Hard Disk Drives (HDDs)*, but also confirmed on *Solid State Drives (SSDs)*, which are fortified against random workloads. Hence, further experiments to investigate the sensibility of *SSDs* to I/O patterns are described in Section 4.4. Section 4.5 presents the related work of *IOscope* while Section 4.6 delivers a synthetic summary.

4.2 IOSCOPE DESIGN & VALIDATION

This section firstly presents the necessary information about *eBPF*. It then shows the design of *IOscope* tools and their experimental validation.

4.2.1 *Foundation: extended Berkeley Packet Filter (eBPF)*

The basic idea of *eBPF* is to inject byte-code programs into the kernel for extending a given kernel functionality. For example, injecting a program into the kernel to perform statistics on the ratio of used RAM. The communication with the *eBPF* virtual machine which is adopted by Linux kernel[2] can be achieved through *bpf syscall*.

The virtual machine, which is the core of *eBPF* has three significant features. Firstly, it has eleven 64-bit registers. One of them is only readable for holding the frame pointer of the injected *eBPF* program. Secondly, it has an extended verifier which checks that the *eBPF* byte code is free of loops, has no side

1 https://lwn.net/Articles/598545

2 BPF manual page on Linux: http://man7.org/linux/man-pages/man2/bpf.2.html

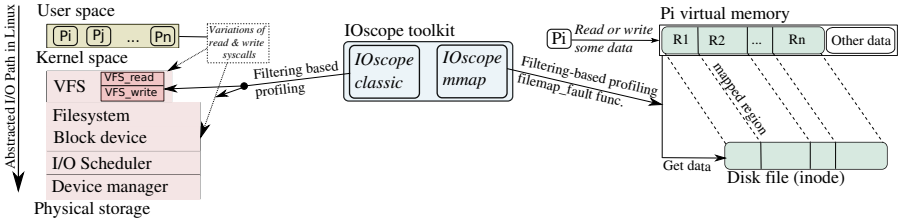


Figure 4.1: *IOscope* tools and their instrumentation points inside the Linux kernel

effect behaviors (e.g., could not lead to crashing the kernel), and terminates without problems. At third, *eBPF* also posses in-kernel data structures (*eBPF-maps*), which are accessible by userspace processes, suitable to use for data communications between *eBPF* and userspace programs [131]. Through those data structures, the *bpf syscall* bidirectionally transfers the data between the kernel/userspace pair. It carries out both injecting the *eBPF* byte code into the kernel, and communicating the target kernel data towards a userspace process.

4.2.1.1 *eBPF for Tracing*

eBPF can be used to do in-kernel tracing thanks to its ability to connect to various data sources of Linux (*kprobes*, *uprobes*, *tracepoints* and *perf_events*). Whenever an *eBPF* program is loaded into the kernel and attached to a data source (e.g., a kernel function), the *bpf syscall* introduces a breakpoint on the target function. This allows *eBPF* stack-pointer to capture the functions' context (e.g., *parameters*). The *eBPF* program is then supposed to run as configured, i.e., before and after every single event on the target function. The program nevertheless cannot alter the execution context of traced functions because a not-writable register holds the context.

eBPF can filter the events before collecting data, contributing to have real raw data for post-analysis. This is different from the *blind-profiling* method whereby the tools records various data from the system, i.e., collect the data from multiple events. With *eBPF*, a filtering test can be accomplished before each time the traced point processes an event. Although events generated by various userspace processes can activate most traced points in the kernel (e.g., kernel functions), the purpose of doing earlier tests is to check if the corresponding event is originated from the target userspace process or not. This narrow the scope of the investigation and eliminates unwanted events in case of having concurrent processes.

4.2.2 *IOscope Design*

The key idea of *IOscope* is to construct the workloads I/O patterns by tracing and filtering sequences of workloads' I/O requests. *IOscope* contains two tools: *IOscope_classic* and *IOscope_mmap* (see Figure 4.1). Both tools work with files offsets. An offset can be defined as a placement value (in byte) held by an I/O

request. It indicates where is the beginning placement in the target file to read from or to write into. The tools apply many filters in earlier steps, e.g., describing the I/O activities of a specific process or a specific data file. They aim at performing a precise-objective tracing and incurring less overhead. They are attached to the target functions using the *kprobe & kretprobe* modes which allow executing the tracing code before and after the target functions' body, respectively. Indeed, tracing based on internal kernel functions is dependent on kernel changes. However, the target functions of *IOscope* tools seem to be stable over multiple kernel releases.

IOscope tools are designed using the Python frontend of the *BPF Compiler Collection (BCC)* project[3]. That dependency implies having two processes for each tool. The first process which runs in userspace is responsible for:

- Injecting the *IOscope*-core code into the Linux kernel
- Performing posterior filtering tasks on the received data from the Linux kernel.

As for the injected program, it runs as a second process inside the kernel. It intercepts the target functions to perform the filtering-based profiling task. It regularly exposes the matched traces into a ring buffer for which the userspace process is connected as a consumer.

In the next sections, the two different tools of *IOscope* are described.

4.2.2.1 *IOscope_classic*

IOscope_classic covers different kinds of I/O methods. To name a few: synchronous I/O, asynchronous I/O, and vectored I/O which works with multiple buffers. It obtains the tracing data from two data sources:

- The *Virtual File System (VFS)* as many I/O *syscalls* terminate at this layer in the kernel I/O path. Although various applications have many choices of I/O methods to issue their I/O requests, there are unified kernel functions in the VFS that receive those I/O requests and try to put them in a similar form before directing them to lower levels.
- Different kernel functions that catch up I/O requests which bypass the *VFS*. Indeed, this can happen when the I/O requests deal with multiple buffers (e.g., *preadv* & *preadv2*).

Therefore, *IOscope _classic* covers almost all I/O methods that are based on the variations of *read, write syscalls* (e.g., *pread, pwritev, read, preadv, pwritev2*). Over one or different data files, *IOscope _classic* can either grab read and write workloads separately, or working on mixed I/O workloads. It reports a clear view of how the analyzed system is accessing data over the execution time. Indeed, the data profiling is made at the absence of any reschedule, which may be proposed by the kernel in later phases such as the *I/O scheduler* phase.

3 A project that facilitates the development of eBPF-based tools: https://github.com/iovisor/bcc

Algorithm 1 Pseudo-code of *IOscope_classic* main actions after being injected into the Linux kernel

Require:

1: D *as a struct* : $fname, offset, byteSize, ts, latency_{ms}$
2: m(D) ▷ *eBPF hash, holding data of type D*
3: p_1, p_2, p_n ▷ *Target function parameters*
4: c ▷ *eBPF context pointer*
5: p_{app} ▷ *pid of the target process to trace*
6: **procedure** TRAPBEFOREEXECUTION(c, p_1, p_2, p_n)
7: $pid \leftarrow c(pid)$ ▷ *take the pid from the context*
8: **if** $pid \neq p_{app}$ **then**
9: **return** ▷ *ignore the current I/O request*
10: **end if**
11: $temp = new(D);$
12: $temp.fname = take\ filename\ from\ file^*\ or\ fd;$
13: $temp.offset = from\ parameters;,$
14: $temp.byteSize = from\ parameter;$
15: $temp.ts = kernelTime(ns);$
16: Update(m(&pid, &temp)) ▷ *stored as a hash*
17: **end procedure**
18: **procedure** TRAPAFTEREXECUTION(c)
19: $pid \leftarrow c(pid)$ ▷ *take the pid from the context*
20: **if** $pid \neq p_{app}$ **then**
21: **return** ▷ *ignore the current I/O request*
22: **end if**
23: $temp_1$=lookup(m(&pid,temp)) ▷ *lookup process, return if the I/O request is not found*
24: $temp_1.latency \leftarrow (kernelTime(ns) - temp_1.ts)$
25: Send $temp_1$ over a *bpf_perf_output* to *usespace*
26: **end procedure**

IOscope_classic collects various informations from the parameters of its target functions. The collected information varies from simple to complicated regarding its acquisition. The simple ones are the *offsets* of I/O requests, their *data size*, their *target file name* or *file identifier*, and the *request type* (read or write). This is done before executing the target function body and the information is stored in a hash map (a type of *eBPF-maps*). As for complicated information, the tool measures the latency for every I/O request as the elapsed time to execute the body of the target function, so the time to serve the corresponding I/O request to write into or to read from a given offset on a target file. This is done by the *kretprobe* mode, which allows running a part of *IOscope_classic* code before the closing bracket of the target function. Algorithm 1 describes the main steps of *IOscope _classic* tool.

The *PID* of the target process must be provided to allow *IOscope* to check every issued request if it belongs to that process or not. Indeed, doing so

Table 4.1: Synthetic workloads originating from different system calls and *Fio IO-engines* used to validate *IOscope* tools

Fio IOengine	Target syscalls	Tested workloads: *read*, *write*, *randread*, *randwrite*, *readwrite*, and *randreadwrite*
Sync	read, write	all
Psync	pread, pwrite	all
Pvsync	preadv, pwritev	all
Pvsync2	preadv2, pwritev2	all
posixaio	aio_read, aio_write	all
Mmap	mmap, memcpy	all

prevent to profile all the I/O requests found in the kernel. Otherwise, the I/O request continues its path without being traced.

4.2.2.2 *IOscope_mmap*

IOscope_mmap tackles the I/O activities that access the *memory mapped files*. This *Memory Map* (*mmap*) method allows a given system or application to map the data files into its private address space, manipulating data like it is already located in memory. The CPU and some internal structures of the kernel bring out the data from the physical storage each time requested by userspace processes.

The main challenge for *IOscope _mmap* is to find a stable instrumentation point through which the I/O patterns can be studied. Indeed, the absence of *syscalls* that carry out the data access pushes us to activate the second option, which is to trace inside the kernel. We find that the kernel function *filemap_fault* fits our context as it is charged to treat the memory faults of mapped files. When a process attempts to read or to write some data, a generated memory fault occurs. The CPU then investigates if the data is already in memory (during previous retrievals) or not yet loaded. Each memory fault has an offset that indicates where the required data is found inside a memory page. Even if a file is mapped into several memory regions, the offsets are still useful as the address space of regions are incremental. Hence, a given offset can be determined if it belongs to this or that memory region.

The typical workflow of using this tool is to provide either the *inode* number in case of revealing the I/O patterns of one data file, or a data path with an extension of target files. *IOscope_mmap* then starts to examine only the memory faults over the given file/files thanks to its preliminary filter. Its mechanism for connecting to the target functions is similar to the *IOscope_classic* tool (*kprobe* and *kretprobe*). *IOscope_mmap* also comes out the same data as the *IOscope_classic* tool (see the previous section).

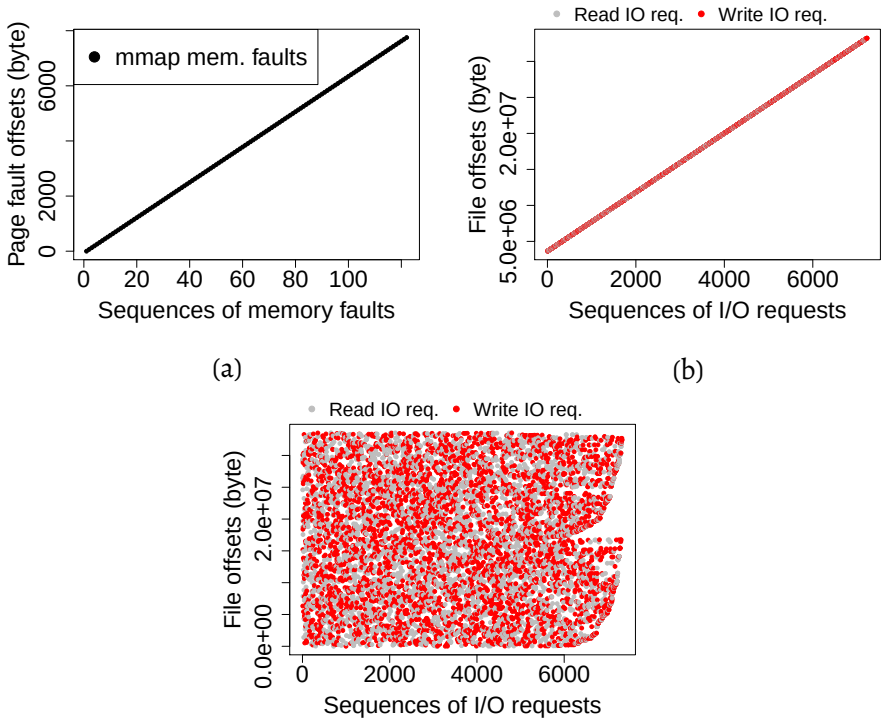


Figure 4.2: Selected results from the experimentation campaign of validating *IOscope* via *Fio* benchmark, over a 32 MB file. a) shows I/O patterns of a read workload on *Mmap IOengine*, b) Shows I/O patterns of a readwrite workload on *Posixaio IOengine*, and c) presents I/O patterns of a random readwrite workload on *Psync IOengine*

4.2.3 *IOscope Validation*

IOscope is experimentally validated over synthetic workloads. The *Flexible I/O Tester* (*Fio*) is used to generate workloads by diverse I/O methods, trying to encompass the applied workloads by real systems and applications. As mentioned in previous chapters, different *Fio IOengines* are responsible for covering the data communication types (e.g., direct access, vectored I/O, memory-mapped access). Table 4.1 lists those engines, the *syscalls* through which the I/O requests pass, and the mode of data access.

The validation experiments are executed on a single machine running *Ubuntu 14.04 LTS*, *Linux kernel 4.9.0* and *Fio-v2.1.3*. The results can be interpreted as follows to highlight the I/O pattern analysis. The flow of sequential workloads is represented as a diagonal line of file offsets over the sequences of memory faults (in case of *mmap* or the I/O requests. Indeed, the I/O request number i access the data starting from the offset s, the request $i+1$ accesses the offset $s + i_{datasize}$, and so on. As for random workloads, they are represented as scratch dots, indicating that the I/O requests target random offsets during the workload execution.

Figure 4.2 shows selected validation results. It is noticeable that the diagonal lines offsets are present in the sub-figures of sequential workloads. Similarly, the random workload is represented as scatter points. Figure 4.2-c shows that the workload is totally shapeless. Indeed, this is determined by the degree of randomness of *Fio*'s random data access which is very high.

4.2.3.1 *IOscope Overhead*

We used two aspects in measuring the overhead of *IOscope* tools: the additional time and memory usage. Provided that, the overhead in terms of additional time is measured by getting the difference between the execution time of an experiment with & without using *IOscope* over 70 GB of data. The maximum overhead obtained for an experiment was less than 0.8% of the execution time regardless of analyzing millions of I/O requests. In terms of memory overhead, the ring buffer of *IOscope* is limited up to 8 MB. Moreover, no single event is lost with this configuration even in case of having high frequency I/Os.

4.3 USE CASE STUDY: SECONDARY INDEXATION IN NOSQL DATABASES

The variety of *NoSQL* data models (e.g., key-value stores, document-based, columnar-based, and graph databases) makes them one of the first choices for big data storage [141]. Although the differences in data models between *Structured Query Language* (*SQL*) and *NoSQL*, their I/O characteristics are almost similar [127, 128] regarding the manipulation of workloads in lower layers of I/O stack. Indeed, I/O stack is stable, so optimization works are often done near to the disks' storage interface [129, 130]. However, dealing with a diversity of data (structured, semi-, and non- structured) without a unified scheme has significant influences on the way of storing data. *NoSQL* databases store data on disks using variable-length records, unlike SQL databases that deal with structured data already optimized to be stored using a unified format on disks. This on-disk dynamism increases the challenges to have an optimized performance in case of dealing with I/O-intensive workloads.

In a usage that fit big data style, a tremendous amount of data is generated and stored into *NoSQL* databases without having prior knowledge about querying them. Hence, creating secondary indexes is crucial to facilitate data retrieval. They could not only improve knowledge extraction but also enhance the performance by making faster queries. This process is always seen as a sequential workload since each data element must be traversed at least once. Since each *NoSQL* database has its algorithms for creating indexes, this may add some algorithmic and technical challenges that may be hidden for high-level testing tools. That is why our study is based on that workload.

In this section, we highlight the advantages of using *IOscope* to perform micro-observability over selected Not Only SQL (NoSQL) solutions. We firstly describe in Section 4.3.1 the I/O systems of the selected databases for this

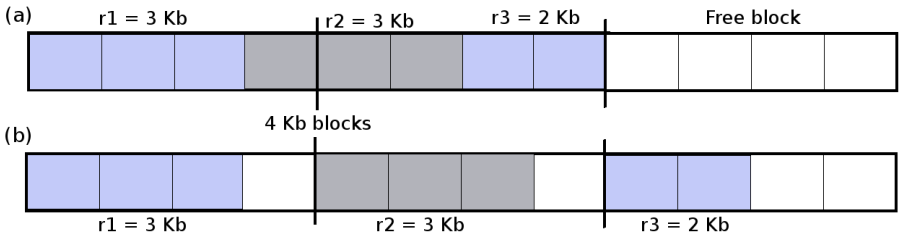


Figure 4.3: a) Spanned allocation with 3 records fit into 2 blocks, and b) Unspanned allocation

study (*MongoDB* & *Cassandra*)[4]. Then, Section 4.3.2 presents the secondary indexation experiments done on *MongoDB* & *Cassandra*.

4.3.1 *I/O Characteristics of Databases Used in the Study*

This section describes the I/O systems of *MongoDB* and *Cassandra*. It also briefly explains the indexation process in those databases.

4.3.1.1 *MongoDB*

MongoDB is a document-based *NoSQL* database. It has several storage engines, among them *WiredTiger*[5], which is the default one starting from *MongoDB v3.0*. *WiredTiger* is recommended by *MongoDB* to be used for different workloads as it is optimized for high production quality and scalability. Consequently, we use that storage engine in *MongoDB* experiments.

MongoDB stores related documents in logical containers named collections. Each document must have a field called (*_id*) to hold the document inside its collection. Indeed, this field can be automatically added into documents if needed. At the storage level, *WiredTiger* uses key-value pairs to store documents as *btree records* where each key is assigned by a unique *_id*. We could have some empty spaces between the records because *WiredTiger* uses unspanned allocation, as shown in Figure 4.3. This behavior wastes storage resources where *WiredTiger* reserves more storage resources than the size of data at the expense of optimizing the records retrieval. Indeed, doing so forces records to be stored at beginnings of physical blocks to be easily spotted in case of data retrieval.

WiredTiger tries to allocate contiguous physical blocks on disks for each collection not only to avoid additional mechanical movements in case of sequential access but also to avoid using linked lists for connecting data. Thus, offsets could be calculated directly instead of dealing with pointers. Eventually, all *MongoDB* workloads are served using the classical I/O mechanism represented by *read* & *write* system calls in Linux.

4 *MongoDB* & *Cassandra* are selected for this study as they are interchangeable regarding their usage despite the differences in their data models and architecture

5 https://docs.mongodb.com/manual/core/wiredtiger

SECONDARY INDEXES IN MONGODB

MongoDB supports many types of indexes such as textual, multiple, and spatial. They can be created over collections. If the index is to be created over a distributed Cluster, the router process of MongoDB (called mongo) directs the request to the related primary machines in parallel, so every involved machine holding a portion of collection's data starts the indexation process separately. The router then waits to receive the acknowledgments from all the involved machines to announce that the indexation process is succeeded.

4.3.1.2 *Cassandra*

Cassandra is a columnar data store. Its data model is partially similar to the relational data model in terms of schemes. Cassandra uses a complex I/O system for storing and retrieving data. The endpoint of this I/O system is the *Sorted Strings Table* (*SSTable*), a key-value data store which provides a persistent and immutable storage on-disks.

*SSTable*s store many kinds of metadata as well as the data itself. The recent version of *SSTable* stores information about where the data could be located, compressed or not, and other statistical information about data filtering and contents. Data could be separated among different *SSTables* on disks; this can be done through a manual flush of data into the disk, or automatically when a memory size threshold is exceeded. This action does not affect the I/O performance since the metadata helps *SSTables* to access data efficiently.

Cassandra uses *mmap* mechanism to access data stored in *SSTables*. When *Cassandra* starts up, it maps all the data portions of *SSTables* into its virtual memory. Therefore, it is not involved anymore for sending I/O requests as it considers that the data resides in its address space. Given that, page faults will be fired if *Cassandra* wants to access specific data which is not yet loaded into memory. Hence, the kernel will retrieve the data from the underlying storage, and it will notify the waiting CPU by the address of the retrieved data. The CPU then will map this main memory address into the private address space of *Cassandra*.

SECONDARY INDEXES IN CASSANDRA

Creating a secondary index in Cassandra involves scanning all corresponding *SSTables*. *Cassandra* tackles this mission by reading at first the indexes files of *SSTables*. These files have nothing but pointers to the data rows inside data files. Every read request on an index file is followed by a read request of the corresponding data from the same *SSTable*. Similarly to *MongoDB*, performing an indexation operation on a Cluster requires from the node that receives the command to send the requests to the machines that held the target *keyspace* in parallel. The process terminates when all the data portions are indexed.

4.3.2 *Experimentation on MongoDB & Cassandra*

This section describes a set of experiments for which *IOscope* is used to analyze I/O patterns. It starts by presenting the environmental setup, and the applied

experimental scenarios. It then describes the experiments done on *MongoDB* and its revealed performance issue before describing how *IOscope* is used to explain that issue. It ends by describing the experiments done on *Cassandra*.

4.3.2.1 *Setup, datasets, and scenarios*

ENVIRONMENTAL SETUP

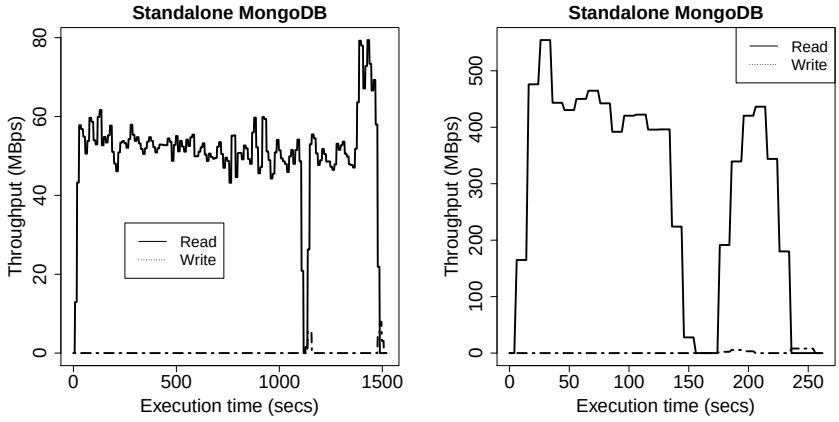
Several machines from the *Grid'5000* testbed are used to perform these experiments. Each machine has two Intel Xeon E5-2630 v3 CPUs (8 cores/CPU), 128 GB of RAM, and a 10 Gbps Ethernet. Additionally, every machine is equipped with an *HDD* of 558 GB, and an *SSD* of 186 GB, connected as a *just a bunch of disks (JBOD)* using Symbios Logic MegaRAID SAS-3 3008 (rev 02). Cencerning the operating sytem, each machine is deployed with *Ubuntu 14.04 LTS* and *Linux 4.9.0* in which a resident *eBPF* virtual machine is enabled. In the I/O stack, the *Ext4* filesystem is used and the *deadline* scheduler is used as an I/O elevator. Indeed, Linux I/O schedulers (*Noop*, *deadline*, and *CFQ*) are interchangeable in our experiments due to the absence of concurrent I/O processes. Concerning the storage device, the *Native Command Queuing (NCQ)* command of disks is configured to its maximum value by default (64); this enables rescheduling groups of concurrent I/O requests inside the storage device, minimizing the mechanical seeks in case of using *HDD*. Eventually, we clean the cache data including the memory-resident data of *MongoDB* and *Cassandra* between experiments. No more than one experiment is executed at the same time.

We use two stable versions of databases. *MongoDB v3.4.12*[6] is used. It is tested with default configuration in which *WiredTiger* is the activated storage engine. As for *Cassandra*, we use *Cassandra v 3.0.14* with its default configuration too. In both databases, nodes hold equal portions of data in case of cluster experiments, thanks to their built-in partitioner/balancer.

DATASETS

Two equally sized datasets are created (each has 71 GB) to perform our experiments on *MongoDB* and *Cassandra*. Their contents are randomly generated. Each dataset has 20,000,000 data elements (called documents in *MongoDB* and rows in *Cassandra*). The sizes of data elements are selected between two boundaries (1 kB & 6 kB) with 3.47 kB as an average size. Each element consists of *one integer with random values*, *a timestamp*, *two string fields*, and *one array which has a random number of fields up to four*. The reason behind the randomization of sizes and contents is to eliminate data-biased results. *MongoDB* stores the dataset as a single file. In contrast, the number of *Cassandras' SSTables* depends on how many times the data is flushed into disks (one or more SSTables).

6 A major version of *MongoDB* (v3.6) has been released during writing the original paper of this work. It suffers from the same performance issue discussed in Section 4.3.2.2, regardless of the optimized throughput



(a) *HDD* (b) *SSD*

Figure 4.4: I/O throughput over execution time of single-sever experiments

WORKLOAD & SCENARIOS

The experiments are executed on either a single server or a distributed cluster of only two shard nodes as the scalability is out of the scope of experiments' context. In both scenarios, one client running on another node performs an indexation workload over an integer field, pushing the target databases to read the entire datasets in order to construct a corresponding index tree. The objective is to look at how each database is accessing data and to see if some hidden issues could be revealed. This does not necessarily mean that the results of both databases are comparable due to the absence of a tuning phase for making an *apple-to-apple* comparison.

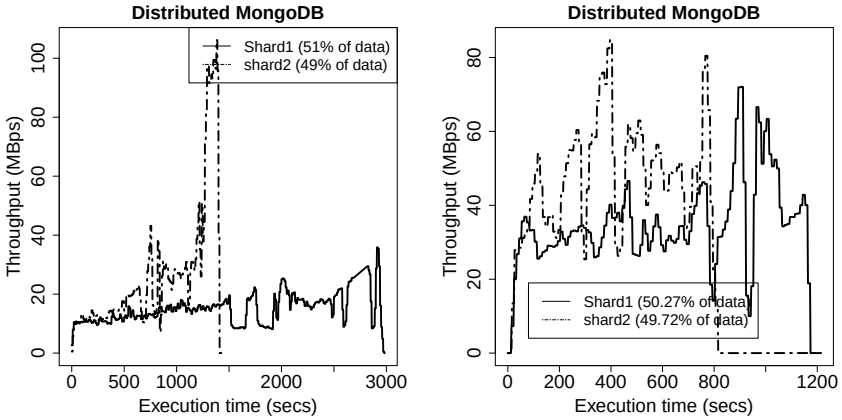
Al experiments cover both technologies of storage: *HDDs* and *SSDs*. They are executed one-time using *HDDs* as principal storage of the involved machines, and another time on *SSDs*. For both storage devices, we test the data contiguity on disks by profiling their physical blocks using filesystem `FIBMAP` command. Results show that each data file resides on about 99.9% of contiguous blocks.

Both databases support a hash-based approach to distribute their data. Hence, this approach is used in case of the distributed experiments. The distribution is achieved over the *_id* field in *MongoDB* and over the primary key in *Cassandra* (*uuid* field), both in order to obtain evenly distributed data.

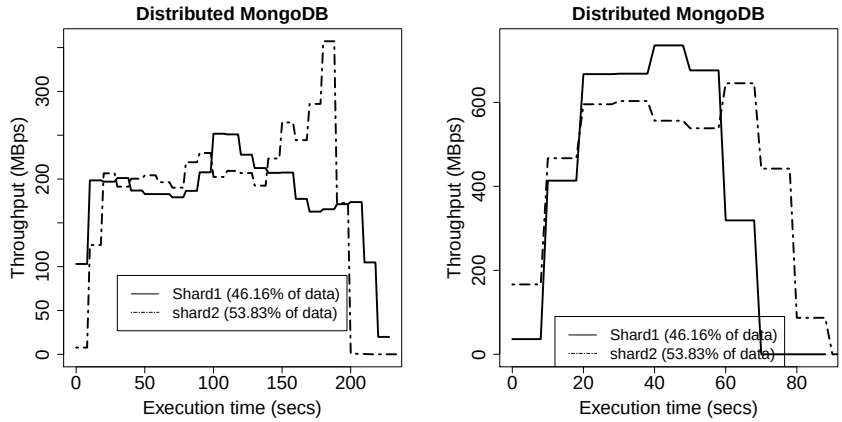
To simplify the understanding of results, we use the term *executions* to differentiate between running the same experiment more than one time. For the distributed experiments, it means that the experiment is re-performed from the step of pushing the dataset into the distributed cluster.

4.3.2.2 *MongoDB experiments*

This section describes the high-level results of executed workloads before showing *IOscope* results. High-level results are presented to convince that they only give insights for understanding I/O performance but cannot explain



(a) Results of two distributed executions on *HDDs*



(b) Results of two distributed executions on *SSDs*

Figure 4.5: I/O throughput over execution time of four independent executions of two-shards cluster experiments

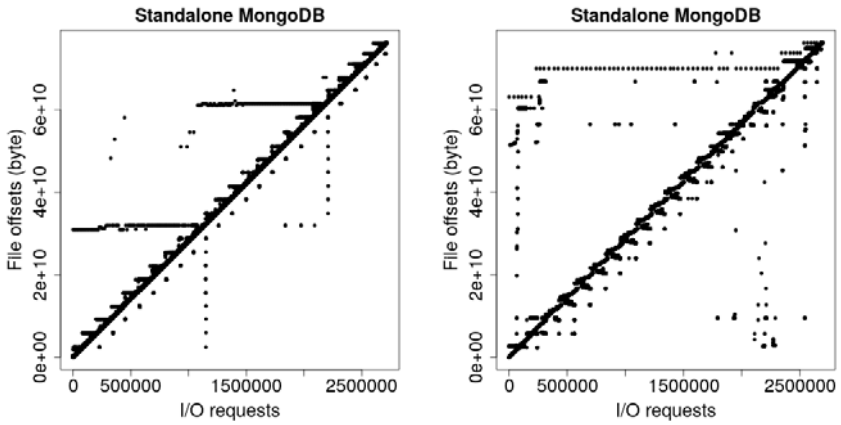
issues. The section ends by describing an *ad hoc* solution to the discovered issue by *IOscope*.

HIGH LEVEL RESULTS

The indexation workload described in Section 4.3.2.1 is executed. The high-level results in terms of I/O throughput and execution time are described below. Figure 4.4 shows the results of single-server experiments. The execution time is reduced by a factor of 6.3 when using SSD as primary storage instead of HDD. This indicates how storage technology can improve accessing data. Moreover, both experiments have a reading gap in the second half of the execution, indicating that a partial commit of the index tree is written back to disks.

Repeating these experiments makes no changes over the execution time and throughput values. Hence, we use them as a reference for the distributed experiments.

For the two-shards experiments, we expect to decrease the execution time by half as an ideal case of linear scalability. However, the performance results of several runs are not as expected, as shown in Figure 4.5. *MongoDB* reports



(a) *HDD* (b) *SSD*

Figure 4.6: I/O patterns of the single-server experiments described in Figure 4.4

variable performance in each execution over both *HDDs* and *SSDs*. Performing more executions over the same dataset brings out variable results too.

Regarding the *HDD* results, Figure 4.5 shows that the first experiment has the worst performance compared with the standalone experiment as its first shard has 3000 s as an execution time while the standalone only has 1500 s! The second experiment has a slight gain of 20% of the standalone execution time. Similarly, the *SSDs* results are variable where the total gain of performance in the first experiment is only 8% compared to the *SSD* standalone results. In contrast, the second experiment has more than a three-fold increase in performance.

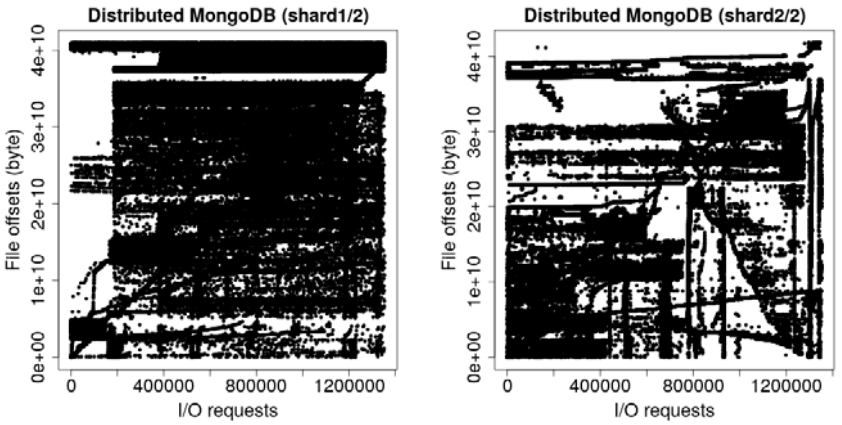
The obtained results argue that there is a hidden issue behind performance variability. However, such an issue could not be explained using high-level metrics as shown above. Without having the exact reasons for the occurred issue, we could not know if it is related to the database itself or if the environment plays a specific role in that performance change. Hence, Hence, we decided to go beyond those results by doing further investigations with *IOscope*.

ANALYZING I/O PATTERNS USING IOSCOPE

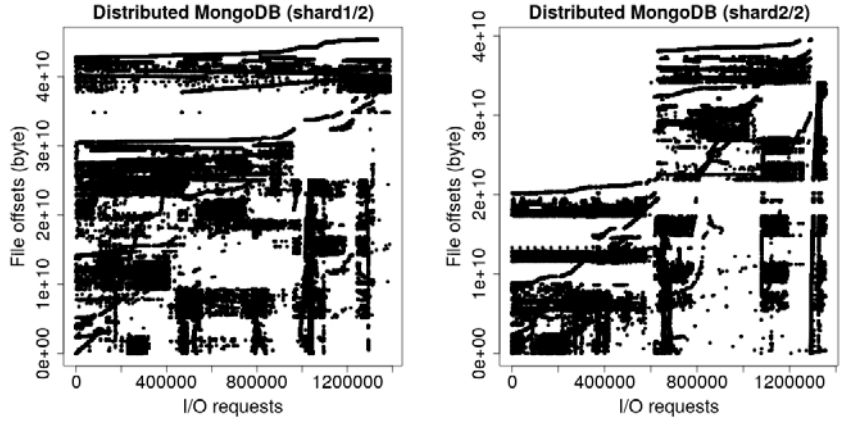
IOscope collects more than 1 GB of traces. We expect that *IOscope* will form a clear diagonal line of file offsets as the data is expected to be accessed sequentially. However, the obtained results reveal noisy access patterns or even shapeless ones in case of distributed experiments.

For the single-server experiments, both I/O patterns over the *HDD* and the *SSD* are acceptable. The files are sequentially accessed as shown in Figure 4.6. The diagonal lines are present in both sub-figures regardless of the tiny noises that might refer to file alignments operations.

IOscope uncovers the reasons behind the performance variability of the distributed experiments. Figure 4.7 and Figure 4.8 show the I/O patterns of these experiments. The same analysis can be done for *HDDs* and *SSDs* experiments. Regarding the results obtained from *HDD* experiments, all I/O



(a) First experiment results



(b) Second experiment results

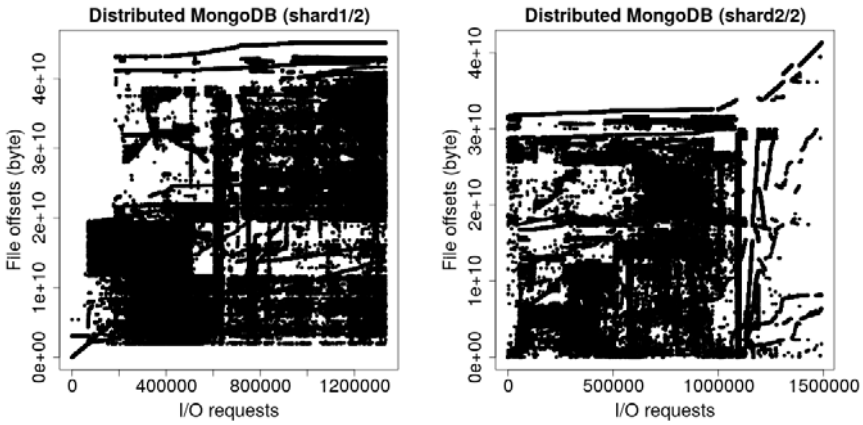
Figure 4.7: I/O patterns of the distributed experiments on *HDDs* that are described in Figure 4.5-a

access patterns in Figure 4.7 are random. Moreover, the obtained I/O patterns of each shard correspond to its execution time.

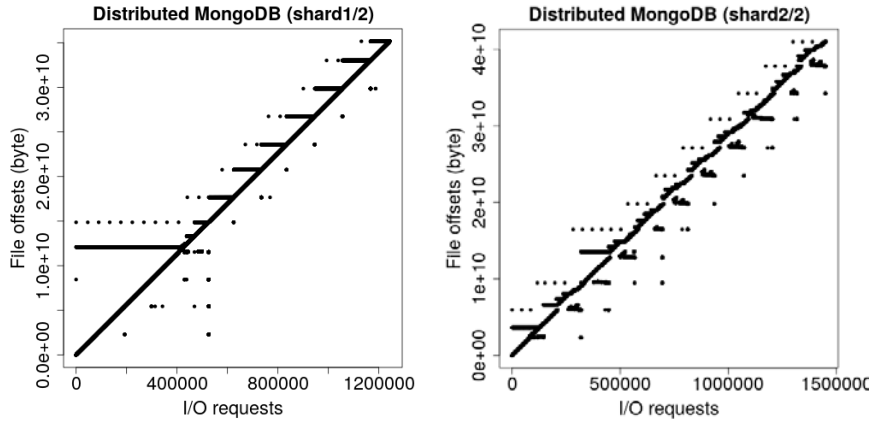
In regard to the *SSDs* experiments, Figure 4.8-a shows that both shards have totally-random I/O patterns[7]. They take about 97%, 82% of the execution time of the single-server experiment. In contrast, the I/O patterns of both shards shown in Figure 4.8-b are sequential. The shards reach the required performance (near 50% of execution time obtained in the single-server experiment). Hence, it is obvious to see the shards patterns as diagonal lines indicating that the data is accessed as it should be. This example shows that *IOscope* is able to explain issues even over recent storage support like *SSDs* and over a fine-grain execution time. This leaves no doubt that the I/O patterns are behind the reported performance variability.

We performed further experiments with three and four sharding nodes, but the random access patterns are still present. As a result, the inefficient way used by *MongoDB* for accessing data is the main reason of obtaining that issue.

7 Further investigation about the sensibility of *SSDs* to I/O access patterns in Section 4.4



(a) First experiment results



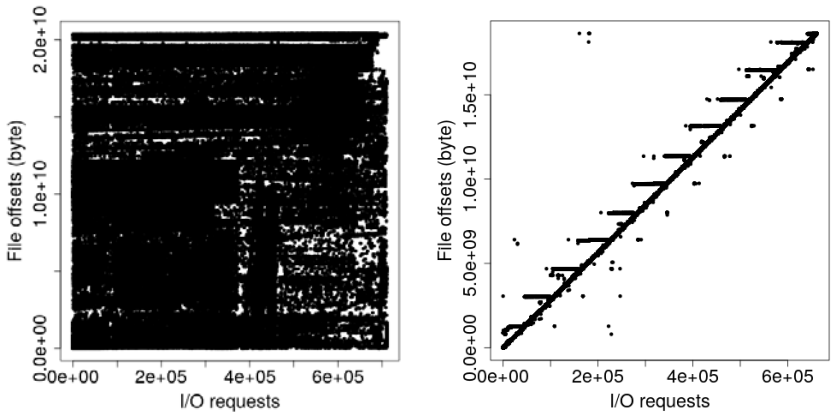
(b) Second experiment results

Figure 4.8: I/O patterns of the distributed experiments on *SSDs* that are described in Figure 4.5-b

AN AD HOC SOLUTION TO FIX MONGODB ISSUE

A mismatch between the order applied by *MongoDB* to retrieve data and the order of stored data is behind the above-described issue. *MongoDB* tries to sequentially traverse the documents based on its view of pre-stored $\_id_s$. This occurs even if its retrieval plan does not follow the exact order of documents in the collection file. As described, the symptoms of this issue are 1) incurring mechanical seeks and 2) having noisy I/O patterns.

The key idea of our *ad-hoc* is to rewrite the shards data locally. This implicitly updates the $\_id_s$ order regarding the order of the documents in the stored file, i.e., the inaccurate traversal plan of *MongoDB* will be replaced. The detailed steps of this solution are as follows. Firstly, we make a local dump of shards' data; this dump will sequentially retrieve the data from the stored file, so it will have an accurate view of the documents' order. Secondly, we re-extract the local dump on the corresponding shard, so *MongoDB* takes into account the novel documents' order. Of course, it is unrealistic to perform this solution every time encountered by a similar issue due to the enormous overhead of



(a) (b)

Figure 4.9: I/O pattern of a *MongoDB* shard with 20 GB of data a) before, and b) after applying our *ad hoc* solution

rewriting data. But it gives insights to *MongoDB* community to fix that issue in upcoming versions.

Figure 4.9 shows a worst I/O pattern obtained on a shared node over an *HDD*. After applying our *ad hoc* solution, the execution time becomes 12.4 times faster (it is reduced from 1341 s to 108 s). On *SSD*, the performance is enhanced with a speedup factor of 2.5 (time is reduced from 89 s to 32 s). This might be related to the nature of the used *SSD* (Toshiba PX04SMQ040) which is optimized for sequential reading workloads.

4.3.2.3 *Cassandra experiments*

This section describes the results of experiments performed on *Cassandra*.

SINGLE-SERVER EXPERIMENTS

Cassandra maintains a stable throughput during the workload execution, as shown in Figure 4.10. However, the workload execution depends more on CPU as the stacked CPU sub-figure shows; the peak CPU reaches more than 150% of a core capacity. We only show the I/O patterns of the biggest *SSTable* in the same figure. In fact, the other *SSTables* have the same clear sequential access (the dataset is represented by five different-size *SSTables*).

The peak value of the disk utilization is near 30%, indicating that the indexing operations are not I/O bounded. Hence, the factor that stresses performance is the amount of used memory. If we limit the available memory for *Cassandra*, the performance in terms of execution time will increase to some extent. This occurs due to an increase in memory operations being performed, such as freeing memory pages. However, the access patterns will not be changed, thanks to the metadata that are used to regulate accessing data.

DISTRIBUTED EXPERIMENTS

Cassandra's nodes still reach the same throughput of the single-server config-

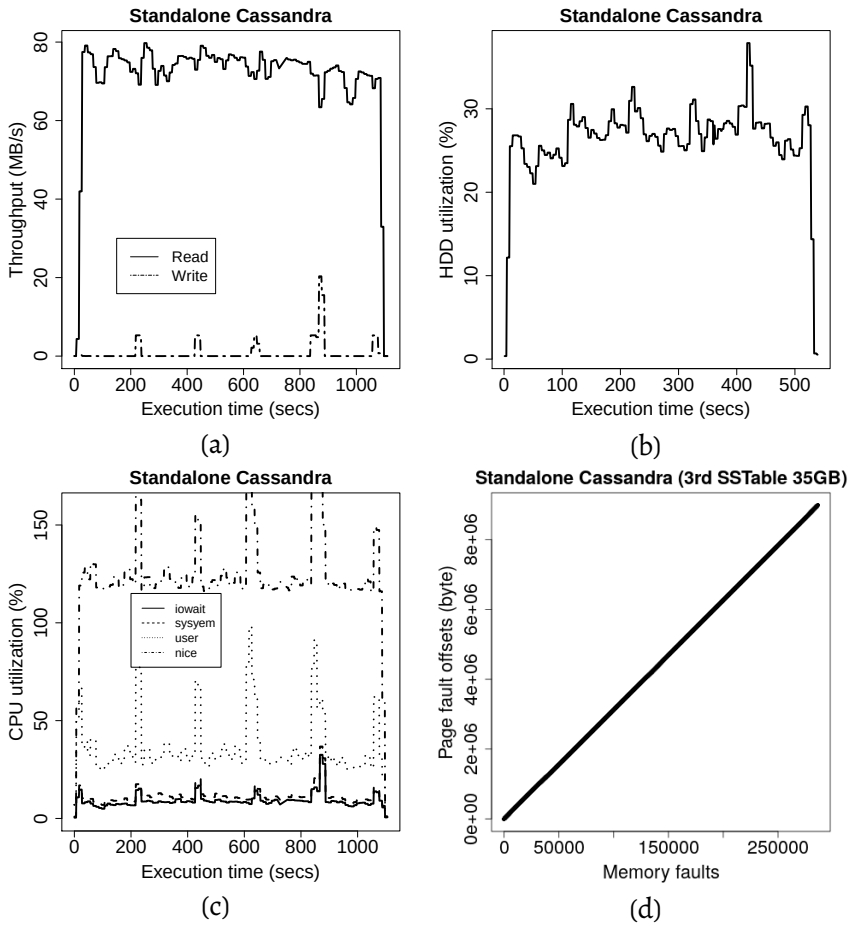


Figure 4.10: Cassandra single-server experiment results on *HDD*. a) shows the I/O throughput, b) shows the disk utilization, c) presents the CPU mode, and d) shows the I/O pattern of the largest SSTable

uration both over *HDDs* and *SSDs*. As a result, the execution time is optimized as expected on both nodes of *Cassandra*. Each node takes near 50% out of the single-server execution time (see Figure 4.11). Moreover, *IOscope* shows sequential I/O patterns for both experiments. (similar results of Figure 4.10-d).

4.4 SSDS AND I/O PATTERNS

Solid-state drives (SSDs) achieve higher performance than HDDs thanks to their complete change in design and architecture. They use performant storage units such as NAND flash memories for storing data, decreasing the performance gap between accessing data stored on RAM and on secondary storage. However, the variety of I/O workloads makes it challenging to leverage the potential of SSDs for all use cases. On the one hand, many I/O workloads still have features that were useful to improve accessing data on HDDs, e.g., having I/O block sizes around 4 KB which is the size of memory pages in Linux. Doing so on SSDs does not make sense as SSDs have an internal page size between 4 KB and 4 MB. Hence, accessing data with small

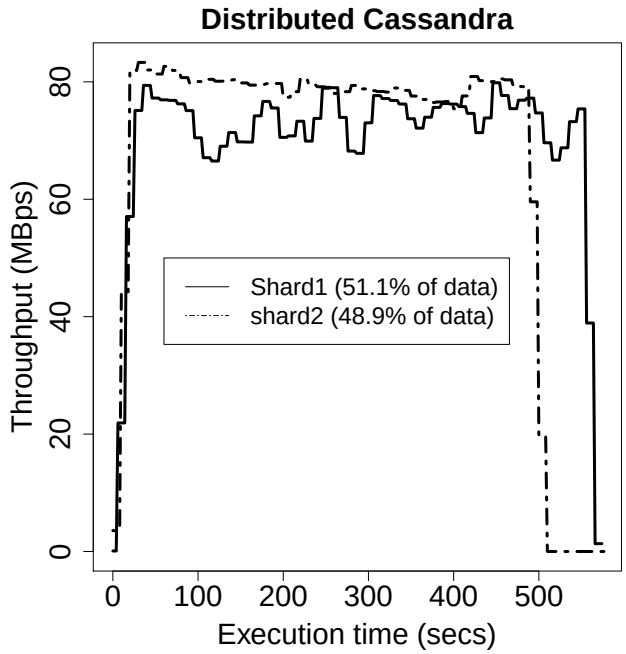


Figure 4.11: I/O throughput and execution time of two-shard cluster experiments on *Cassandra*

block sizes may nevertheless incur performance issues. On the other hand, SSD controllers are still black boxes, so extracting their internal configurations to adjust the workload characteristics accordingly is not feasible. Of course, methods such as extracting information from SSD behaviors [69] might be useful, but trying to do so every time having a different workload is exhaustive. Moreover, systems such as big data storage do not provide support about choosing suitable SSDs for their workloads. Given that, performance issues will occur sooner or later if workloads do not take into account the particularities of underlying storage devices.

Delivering a comparable performance for sequential and random workloads is one of the promising claims of SSDs. Since the majority of SSD devices rely on memory-like storage units, storing data sequentially and randomly should not provoke any issue for SSDs. Indeed, the storage units take almost the same time to store the data in adjacent or far-off storage cells as there is no meaning of adjacency or contiguous allocation on SSDs. However, results from the study described above (see Figure 4.5) shows that the SSD storage in use reports a delayed execution time in case of running random workloads. Even if the experiments' workload has been initiated by only one I/O process that frequently accesses massive data using I/O requests with small size, reporting a degraded performance for random workloads is not logically expected. Hence, we are motivated to do further investigations to explain the leading causes behind this unusual behavior.

In this section, we perform a study over SSD storage to reveal the responses to I/O pattern changes. We perform several experiments that take into account the workload characteristics of the experiments described in the previ-

Table 4.2: Description of three SSD models used in these initial experimentation phase

SSD model	SSD Type	Size	Optimized for	manufacturer	Year
PX02SSF020	MLC	200 GB	High-endurance & write-intensive	Toshiba	2017
PX05SMB040	MLC	400 GB	High Endurance	Toshiba	2016
C400-MTFDDAA064M	MLC	64 GB	—	Micron	< 2008

ous section. Several potential areas that may affect the I/O performance are investigated here, trying to understand what pushes the SSDs under study to report moderate I/O performance for random workloads. These areas are either related to 1) the workload itself such as the size of I/O requests and the concurrency of I/O processes, 2) the I/O stack such as the impact of I/O schedulers, 3) or related to the internal parameters of the storage devices themselves such as the internal `readahead`.

4.4.1 *Initial experiments with artificial workload*

The experiments described above have only one process to perform the I/O activities of the executed workload. That process sends near 93% of I/O requests with 28 KB as a block size. The requests are sent directly to the storage device (synchronous I/Os) via `pread syscall`.

Since the objective here is to investigate only on the behaviors of SSDs, we exclude the systems under test used above to eliminate as much as possible all biases related to components other than the SSDs. Hence, we use *Fio* benchmark (Fio 2.16) to reproduce a similar workload. We create one process to mutually read or write 40 GB of data, testing two block sizes (4 KB & 28 KB) and two access modes (sequential & random). The experiments are to be performed on a machine running *debian9* with *Linux 4.9.0* and *ext4* as a filesystem. Many parameters should be defined in *Fio* script to do so. For instance, having one I/O process to perform a sequential reading over a file of 35 GB and an I/O block size of 4 KB is as follows:

Listing 4.1: Reproducing the workload of IOscope experiments artificially using Fio

```
fio --name TEST1 --filename=test.img --rw=read --size=35G  --
    ioengine=sync
--iodepth=64 --blocksize=4k --direct=1  --numjobs=1 --group_
    reporting
```

Three different models of SSDs are used in this experiments. These models are described in Table 4.2.

Tables 4.3 & 4.4 show the results of reading and writing experiments on different SSDs. For both experiments, increasing the block size of IOs from 4 KB to 28 KB leads to an increase in the performance according to the SSD

Table 4.3: Results of sequential and random reading workloads on three models of SSD

SSD model	Workload	4 KB block size	28 KB block size
PX02SSF020	Read	43.2 MB/s	172 MB/s
	Rand. Read	31 MB/s	148 MB/s
PX05SMB040	Read	136 MB/s	491 MB/s
	Rand. Read	40.1 MB/s	220 MB/s
C400-MTFDDAA064M	Read	61.9 MB/s	143 MB/s
	Rand. Read	25 MB/s	75 MB/s

Table 4.4: Results of sequential and random writing workloads on three models of SSDs

SSD model	Workload	4 KB block size	28 KB block size
PX02SSF020	Write	88.2 MB/s	312 MB/s
	Rand. Write	64 MB/s	292 MB/s
PX05SMB040	Write	123 MB/s	496 MB/s
	Rand. Write	99.2 MB/s	470 MB/s
C400-MTFDDAA064M	Write	58 MB/s	97 MB/s
	Rand. Write	38 MB/s	86 MB/s

model. However, changing the access mode from sequential to random has a significant effect on the reading experiments; the throughput that corresponds to both block sizes is degraded by half for some studied SSD models (*PX055MB040* and *C400*). Similarly, the writing performance is affected by switching the access mode between sequential and random. However, the performance difference does not exceed 30 MB/s which is not comparable with reading experiments where the access mode influences more the performance.

The next section describes the experiments performed to understand why the studied SSDs report degraded performance for random workloads.

4.4.2 *Sensibility of SSDs to I/O patterns*

Determining the primary cause behind the sensibility of certain SSDs to the I/O patterns is not easy since it can be related to several hypotheses. The reason could be either related to 1) the characteristics of the used workload (e.g., small I/O block size), 2) the configuration of I/O stack since some layer such as I/O scheduler can merge, sort, and/or delay the I/O requests for optimizing the performance, 3) the fact of having only one I/O process, or

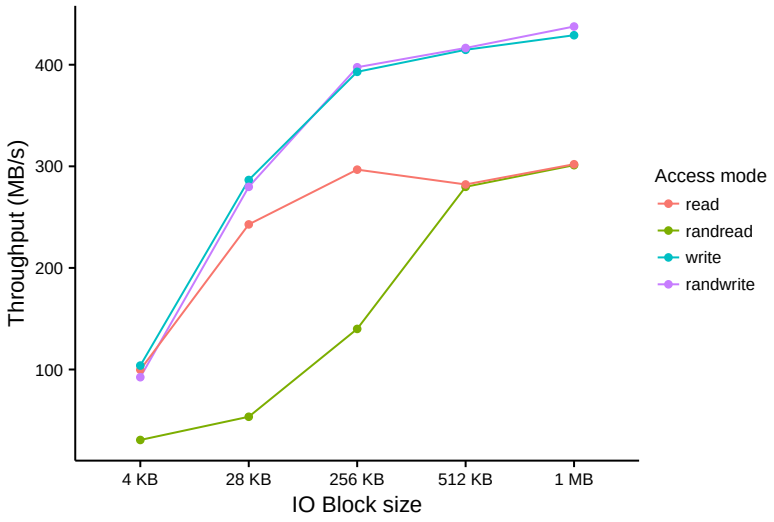


Figure 4.12: Varying I/O block size over the *PX05SMB040* SSD

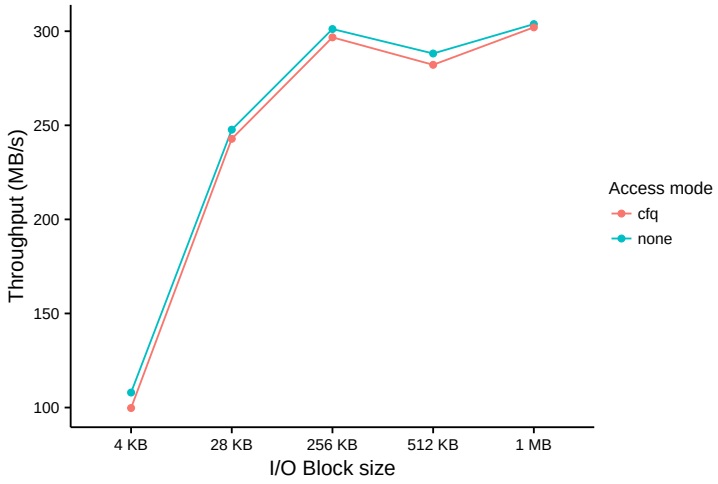
4) the internal behaviors of the SSDs in use as they are not all impacted by random workloads (e.g., *PX02SSF020* model). That I/O pattern sensibility could also be related to all these points together. We narrow our investigation scope to experimenting only on the *PX05SMB040* SSD model since it is the most affected SSD by random reading workloads (see Table 4.3). All results of the experiments described in the following subsections are an average of ten executions.

4.4.2.1 *I/O block size experiments*

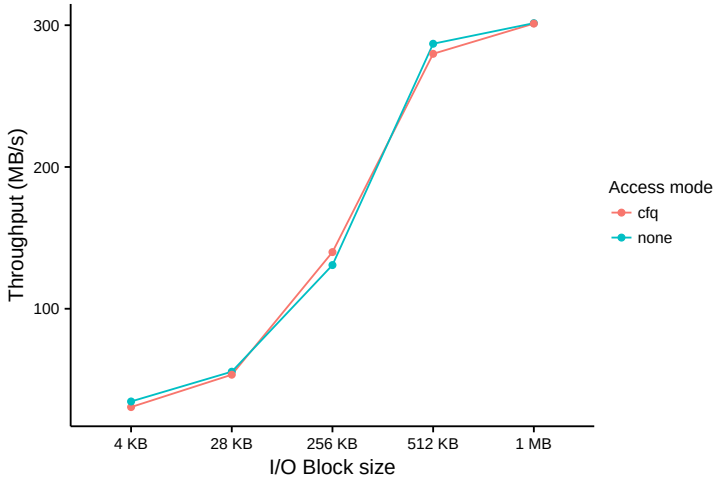
Experiments that vary the I/O block size between 4 KB and 1 MB are performed. Figure 4.12 shows the results of these experiments. One can see that the throughput of the writing workloads is not affected if the data is accessed sequentially or randomly. It just increases by increasing the block size because larger block size means a larger quantity of data written by each I/O request. However, the impact of random workloads is present during the reading experiments. The gap between the throughput of sequential read and random read is shown on the block size points from 4 KB to 512 KB included. Using larger block sizes than 256 KB eliminates the gap of reading data sequentially or randomly. However, these results do not bring answers to understand why that gap is still present with smaller block sizes.

4.4.2.2 *I/O scheduler experiments*

We perform experiments using several I/O schedulers to identify if some of them provoke the performance degradation for random workloads. Linux proposes several I/O schedulers that are basically invented to minimize the HDD's internal seeks as the schedulers are nothing but algorithms to organize and reorder I/O requests. These schedulers are *noop*, *deadline*, and *complete fair queuing (CFQ)* ordered by their level of complexity from the simple to the most



(a) Results of experiments with *sequential reading* workloads



(b) Results of experiments *Random reading* workloads

Figure 4.13: Results of using CFQ scheduler and *blk-mq* with no scheduler over the *PX05SMB040* SSD

complex scheduler that create several I/O queues per process. Although these schedulers are still usable with SSDs despite their HDD-related goals, I/O frameworks such as block multi-queue (aka *blk-mq*) are developed to optimize the I/O for SSDs. *blk-mq* completely bypasses the old I/O block layer and do not have any I/O scheduler so far, relying on the internal schedulers of underlying SSDs. In our experiments, we use the notation of Linux (*none*) to describe the case when *blk-mq* is used. The experiments are performed over the traditional I/O schedulers of I/O block layer (*noop*, *deadline*, and *CFQ*) as well as over *none* of *blk-mq*.

Figure 4.13 only shows the result of using CFQ scheduler and *blk-mq none*. Indeed, the results of *noop* and *deadline* are completely comparable with *CFQ* results due to the usage of only one I/O process in workload execution. In that figure, one can see that varying the I/O schedulers do not have any impacts on the performance as the curves of *CFQ* and *blk-mq none* go hand in hand on

both sub-figures. Additionally, these results do not explain the gap between the sequential and random workloads for small block I/O sizes. For instance, the performance of reading data using a block size of 28 KB is different for sequential and random access. It is near 250 MB/s in case of sequential reading while its less than 50 MB/s in the case of random reading.

4.4.2.3 *Concurrent I/Os*

We narrow our investigation scope to include block sizes between 4 KB & 64 KB since previous experiments show that the issue of random workloads occurs while dealing with small data sizes. We perform experiments with a different number of concurrent jobs, trying to understand if the issue is persistent regardless of the simultaneous access to disks' data.

Figure 4.14 shows the results of varying the number of jobs that access the SSD at the same time. Every job consists of an I/O process that tries to read sequentially or randomly its own 4 GB file. From a, b, and c sub-figures, it is clearly shown that the performance gap between the sequential and random reading workloads occurs when having only one I/O process. Increasing the number of concurrent jobs means sending many I/O requests at a time, exploiting the parallelism of the SSD and eliminating the impact of random access. Indeed, having a large number of jobs indicates that the SSD controller fetches more data each time it accesses the NAND flashes as several files are read simultaneously. Additionally, the larger the block size, the larger the throughput. For instance, the corresponding throughout of 16 jobs in the sub-figures a, b and c are 432 MB/s, 950 MB/s, and 964 MB/s respectively.

Figure 4.14-d shows that the block size is also behind the issue of random performance. Increasing the block size to 64 KB leads to eliminate the performance gap between the sequential and random access even in case of using only one I/O process.

The results recommend having larger blocks size in order to leverage the potential of the storage device. However, it indicates that the SSD behaves abnormally using smaller block size. We expect that the internal usage of buffers and internal page size of SSD is behind that behavior. To emphasize, let us suppose that the internal page size of the used SSD is 64 KB and that a file is allocated on multiple NAND flashes. Randomly accessing that file using a block size of 4 KB means that each read will return 64 KB into the internal buffer, but only the target 4 KB will be sent back to the host. Having millions of I/O requests means that the NAND flashes are always accessed to retrieve the data. In contrast, this always robust the performance of sequential workloads even in case of a single I/O process as the I/O requests often have chances to retrieve data from the buffer instead of reaching the NAND flashes.

In the next subsection, we tune internal parameters of the used SSD in order to understand the performance while having one I/O process and small I/O requests.

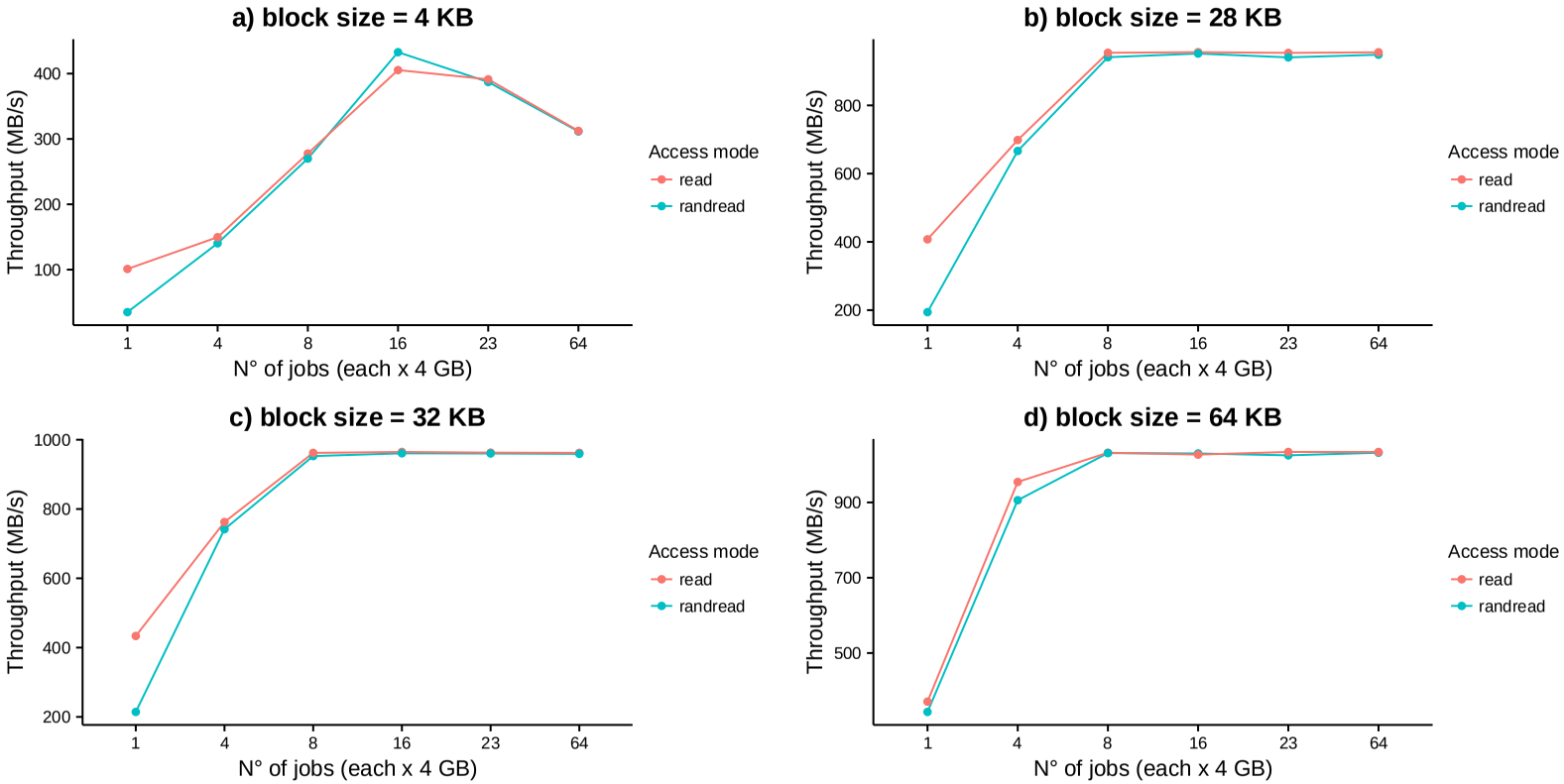
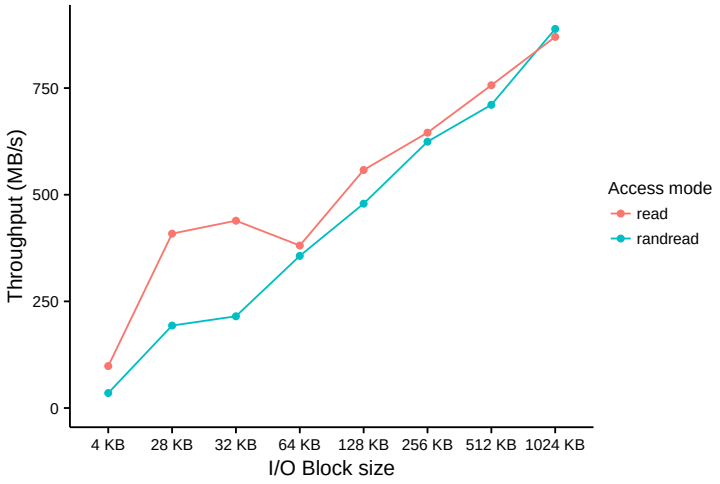
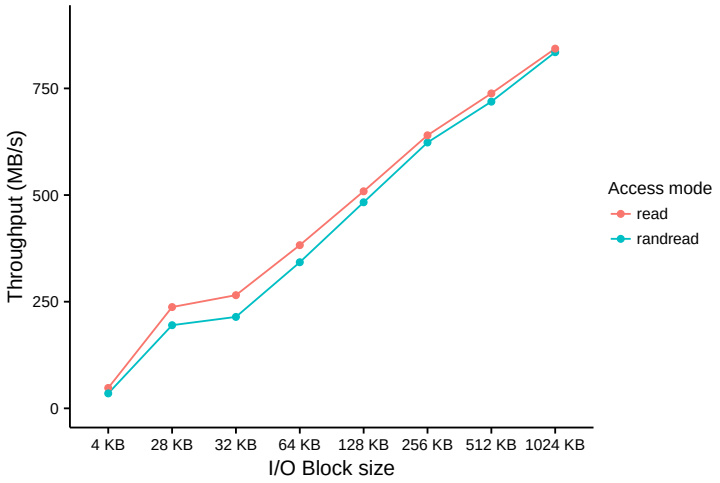


Figure 4.14: Results of using concurrent Fio's jobs over the *PX05SMB040* SSD



(a) SSD's readahead enabled



(b) SSD's readahead disabled

Figure 4.15: The effects of SSD's internal *readahead* on the experiments

4.4.2.4 *Tuning the SSD parameters to understand the performance*

Despite having hints about how to eliminate the performance gap between the sequential and random workloads, we try to tune some internal parameters of the used SSD to understand why that SSD is sensible to random workload in case of having small block sizes and one individual I/O process. We expect that buffering behaviors are behind the distinction of performance between sequential and random workloads.

The `sdparm`[8] tool is used to alter the state of some internal parameters of the SSD. The study includes several parameters among them (*RCD: read cache disable, WCE: write cache enable, FSW: force sequential write,* & *DRA: disable read ahead*). Our experiments show that all parameters except the *DRA* do not affect our experiments. Figure 4.15 shows the results of enabling and disabling the SSD's internal *readahead* over several block sizes. Figure 4.15-a shows that

8 A tool that can be used to change several control parameters for most of SCSI devices

Table 4.5: A serie of SSD models for investigating the impacts of internal *readahead* on random workload performance

SSD model	SSD Type	Size	Optimized for	manufacturer	Year
MZ7KM240HMHQ0D3	MLC	240 GB	Reliability & heavy demands	Samsung	2017
SSDSC2BB300G4T	MLC	300 GB	Read intensive	Intel	2013
SSDSC2KG480G7R	MLC	480 GB	Read intensive	Intel	2017

the pre-read of consequent segments of data increases the performance of sequential workloads, especially for the small I/O block sizes as most of the requests will end up reading from the buffer. In contrast, submitting random IOs does not benefit from such a feature since the SSD could not predict what data will be read next. Disabling SSD's *readahead* prevents pre-fetching data for sequential reads, resulting by having comparable performance between sequential and random reads.

4.4.3 *Discussion*

Although the experiments of the previous section confirm that the internal *readahead* is behind the performance gap between sequential and random reading workloads, this finding only concerns the studied SSD model (*PX05SMB040*). Hence, we need to confirm that via experimenting on other SSDs, in order to neutralize the hardware effects on results. We perform further experiments on three different models of SSDs which show different performance for sequential and random workloads. The characteristics of these SSDs are described in Table 4.5.

Although the SSDs shown in Table 4.5 report a performance gap between sequential and random workloads, we are not able to confirm that the internal *readahead* is behind that performance gap as we do with the *PX05SMB040* SSD. Unfortunately, manipulating internal parameters of SSDs like the *readahead* is not always possible. It depends on the ability of the target SSD to integrate changes. On the one hand, disabling the internal *readahead* using `sdparm` for Samsung *MZ7KM240HMHQ0D3* is not feasible since the *readahead* parameter on that SSD does not have a changeable state, i.e., we can only read its value. On the other hand, in the case of both *Intel* SSDs (see Table 4.5), the `sdparm` indicates that the parameter is changeable but the desired value cannot be saved into the SSD. It seems like there is an interface mismatch where the `sdparm` considers our parameter as changeable but unfortunately when we try to alter its value, the SSD refuses to integrate the new value. Given that, finding the main reason for the issue of I/O pattern sensibility on such SSDs is not feasible without the ability to tune their internal parameters.

4.5 RELATED WORK

Betke and Kunkel [15] proposed a framework for real-time I/O monitoring. It does not implement a filtering mechanism like *IOscope* during the interception of I/O traces, leading to collecting a huge number of generic traces. Daoud and Dagenais [35] proposed a *LTTng*-based framework for collecting disks metrics. Their framework only analyses generated traces of *HDDs*, and no information is provided about its applicability on *SSDs*. In addition, it does not collect file offsets, which is our main metric for analyzing workloads' I/O patterns. Jeong et. al [63] proposed a tool to generate I/O workloads and to analyze I/O performance for Android systems. Their I/O performance analyzer requires a modified kernel and runs only for custom filesystems (ext4, fat32). In contrast, *IOscope* needs no kernel modification and mainly works on the *Virtual File System (VFS)* layer to support wide number of filesystems. Other tools [24, 81, 82, 143, 152] aim to predict and extrapolate the I/O performance for large scale deployments by analyzing and replaying small set of traces. In contrast, our work focuses on collecting fine-grained traces of I/O workloads under study for discovering and explaining I/O issues.

IOscope performs four activities: profiling, filtering, tracing, and direct analysis of I/O patterns. Its filtering and tracing activities are comparable to several tools of the *BCC* project. In general, they give an instantaneous view of matched events on target instrumentation points, presenting outputs like the `top` command of Linux. BCC *Slower* tools are built to filter the I/O operations with large latencies. They work on higher layers of Linux I/O stack. The *fileslower* tool traces operations on the VFS layer of Linux I/O stack while the *ext4slower* tool works on *Ext4* filesystem. Both bring out fine-grained filtering of I/Os (e.g., reporting I/Os per process), but deal with partial I/O contexts. The *fileslower* does not work with *pvsync*, *pvsync2*, and *mmap* I/Os, while the *ext4Slower* lacks supporting *mmap* I/Os. Extracting I/O patterns is still possible for the supported I/O contexts. However, this requires much post-analysis effort compared with *IOscope* which needs nothing to prepare final results.

Several tools collect traces from the block I/O layer on Linux I/O stack. For instance, BCCs' *biotop*, BCCs' *bIOsnoop*, DTraces' *IOsnoop*. These tools generate traces in terms of *accessed sectors* of disks. They do not link those sectors to workloads' accessed files, being more close to studying hardware issues rather than analyzing I/O patterns. To explain, collecting disks sectors do not specify how applications access data files. The reason is that I/O requests are expected to be re-ordered in intermediate layers (e.g., in the I/O scheduler layer). A modified tracer [85] of *blktrace* addresses that issue by combining traces from block I/O and VFS layers. However, it lacks supporting *mmap* I/O, and it needs an additional effort to analyze I/O patterns. Hence, replacing *IOscope* by any of these tools cannot explain I/O issues. Analyzing I/O flow in terms of disk sectors has no sense as there is no constraints to allocate data on successive or random sectors. *IOscope* addresses that by working with files offsets. Over

a given file, the offsets specify the order of all read/written data throughout workloads' I/Os.

4.6 SUMMARY

Storage systems are getting complex to handle Big Data requirements. This complexity triggers performing in-depth observability to ensure the absence of issues in all layers of systems under test. However, the current activities of evaluation are performed around high-level metrics for simplicity reasons. It is therefore impossible to catch potential I/O issues in lower layers along the Linux I/O stack. Provided that, robust and flexible tools are needed to go beyond such simple evaluations, especially for analyzing I/O access patterns as this can be considered as a costly operation for storage systems in production.

In this chapter, we first admitted the role of technology to reduce the cost of operations such as I/O tracing, selecting eBPF as a foundation for our tool (*IOscope*). Indeed, eBPF shows reduced overhead compared with other technologies used for tracing as well as its adoption by Linux system helps to promote the usage of *IOscope*. We then described *IOscope*, with a series of validation experiments, to assemble I/O patterns over millions of I/O operations, promoting to practice micro-observability through applying a filtering-based tracing technique. *IOscope* has less than 1% overhead and is suitable for analyzing production workloads.

We then showed a use case study over *MongoDB* and *Cassandra* using *IOscope*. The experimental results of this use case study reinforce our hypothesis for going beyond high-level evaluations. To emphasize, *IOscope* was able to report the main reasons behind the performance variability of clustered *MongoDB*, over executed workloads. The issue is raised due to an unexpected mismatch between the order of the allocated data on disks, and the traversal plan used by *MongoDB*, resulting as a random I/O access patterns over both storage devices: *HDDs* and *SSDs*. Based on the insights provided by *IOscope*, we proposed an *ad hoc* solution to fix that issue by rewriting the shards data properly, in a way that eliminates the underlying problem. This allows achieving linear and scalable results of the concerned experiments anew.

Eventually, the unexpected results of *SSDs* in the above-described use case opens a research direction to investigate the sensibility of *SSDs* to I/O patterns. Therefore, we described further investigations on *SSDs* which confirmed, on one side, that the internal *readahead* of certain SSDs is behind I/O pattern sensibility as it does not improve the performance of random workloads as it does with sequential ones. Disabling the *readahead* feature leads to having a similar, but also a moderate performance as there is no more anticipation in data access for sequential workloads. On the other side, the investigations also showed that eliminating the performance gap without performance degradation can be done via either increasing the I/O block sizes or increasing the number of concurrent I/O processes. The higher the

number of concurrent I/O processes, the higher the obtained performance of SSDs no matter if workloads are accessing data sequentially or randomly.

Reason, Observation, and Experience—the Holy Trinity of Science

— Robert Green Ingersoll

5 MACRO-OBSERVABILITY WITH MONEX

5.1 INTRODUCTION

In the previous chapter, we have revealed that the experimental observability can be improved by analyzing micro-activities of systems, constructing an overview of systems' behaviors in lower layers of execution. This was to have a *better understanding of performance* and to uncover potential performance issues impossible to be observed via high-level evaluation tools and metrics. Here, we deal with experimental observability from a wide angle. We look for facilitating the overall activity of experimentation by improving observability. This chapter tries to improve the data collection phase of experiments, pushing towards having methodological tools for monitoring experiments and helping experimenters with results analysis. Indeed, enhancing the data acquisition phase of experiments is crucial to promote reproducible experiments.

Most computer science experiments involve a phase of data acquisition, during which metrics are collected about the system under study. This phase has a central role in the experimental process. First, it is the conclusion of the experiment per-se, after the steps of experiment design, setup, and execution. But the collected raw data is also the starting point for the phase of data analysis that should lead to trustworthy, reproducible, and publishable results. One would expect data acquisition to be performed with well-designed solutions, that fully integrate into the experiment workflow, maximize support for reproducibility of experiments, and limit the risk of user errors. However, in practice, experimenters often resort to ad hoc and manual solutions, such as writing *dumps* or logs, gathering them manually, and parsing them using custom scripts.

Many monitoring tools such as *Ganglia* [88] are in use for monitoring various platforms' infrastructure (e.g., clouds, testbeds). They provide an overview of resources status and usage, raising alerts when things go wrong. Beside their goal of permanently providing infrastructure monitoring, these tools can be used with additional effort for monitoring experiments. However, no framework is found for addressing monitoring experiments, and encompassing all experimentation steps that vary from collecting experiments data to creating publishable figures. Thus, the main concern here is to examine the idea of building *EMFs* on top of the state of the art of infrastructure monitoring tools.

In this chapter, we firstly define in Section 5.2 a minimal list of requirements to be satisfied by *EMF*s. We then link those *EMF* requirements with the current state of the art of monitoring tools in Section 5.3, showing that no existing solution experiences a full coverage. Hence, we are motivated to introduce an *EMF* that covers all defined requirements and nicely inserts into the experiment workflow. Therefore, we describe in Section 5.4 the design of *Monitoring Experiments Framework* (*MonEx*). We then highlight in Section 5.5 the different features of *MonEx* over three use cases for which *MonEx* is employed to collect data and to prepare results. Section 5.6 summarizes.

5.2 REQUIREMENTS OF EXPERIMENT MONITORING FRAMEWORKS

An ideal *Experiment Monitoring Framework* (*EMF*) should meet a number of requirements, which are detailed below.

- EXPERIMENT-FOCUSED. The notion of *Experiment* should be central in the *EMF*. It should keep track of experiments' name or identifier, start time, end time, and list of associated metrics. It should also maintain an overview of the different experiments performed by the same or different users (e.g. in a shared experimental environment).
- INDEPENDENT OF EXPERIMENTS. An *EMF* should support a wide range of experiments, regardless of the number of metrics, the frequency of measurements, or the software or services being monitored. Furthermore, it should not be necessary to alter the system under test for it to be monitored by such a tool. This helps to reproduce the experiment on other testbeds even if the *EMF* is absent.
- INDEPENDENT OF TESTBED SERVICES AND EXPERIMENT MANAGEMENT FRAMEWORKS. Building the monitoring facility into the core testbed services or management framework, as an all-in-one solution, has some advantages. For instance, doing so allows for continuous surveillance of infrastructure, easier access for users, and official support/maintenance from environments' operators. However, an *EMF* should ideally maintain a high level of independence from such services to facilitate porting experiments to other testbeds, or monitoring experiments on federations of heterogeneous testbeds.
- SCALABILITY. An *EMF* should scale to a large number of monitored resources, to a large number of metrics, and to high-frequency of measurements, in order to allow understanding fine-grained phenomena (at the millisecond scale), or phenomena that only occur with hundreds or thousands of nodes.
- LOW IMPACT. The *EMF* should have a low impact on the resources involved in the experiment in order to avoid the *observer effect* (the addition of monitoring causing significant changes to the experiment's results).

- EASY DEPLOYMENT. An *EMF* should not depend on complex or specific testbed infrastructure. It should be easy to deploy over any networking or distributed testbeds without tedious configuration.
- CONTROLLABLE. An *EMF* should be flexible. Users should have the choice to enable or disable the monitoring of their experiments at any time, and to select metrics, e.g. in order to limit or evaluate the impact of the *EMF* on the experiment.
- REAL-TIME MONITORING. The *EMF* should provide real-time feedback during the experiment execution, to allow the early detection of issues in long-running experiments.
- PRODUCING PUBLICATION-QUALITY FIGURES. The *EMF* should integrate the final step of the experiment life-cycle, that is turning results into publishable material, with minimal additional effort.
- ARCHIVAL OF DATA. Saving and exporting the experimental metrics of a given experiment is important to allow for future analysis of the data. It is also a basis for allowing distribution in an open format to enable others to repeat the analysis.

5.3 ALTERNATIVES TO EXPERIMENT MONITORING FRAMEWORKS

To distinguish *EMF*s from existing works, this section presents the state of the art of alternative tools to *EMF*s. These tools are classified into three categories: infrastructure monitoring tools, testbed measurement services, and instrumentation frameworks.

INFRASTRUCTURE MONITORING TOOLS. Infrastructure monitoring is a frequent need for system administrators, to track resources utilization, errors, and get alerted in case of problems. Many tools exist, from the ancestor MRTG [96], to *Munin*, *Nagios*, *Ganglia* [88], *Zabbix* [97], or *Cacti*. They differ in terms of design choices such as protocols used to query resources, use of remote agents to collect and export metrics, or the way of collecting and storing data (*push* vs *pull*). However, they all target the monitoring of long-term variations of metrics, and thus are designed for relatively long intervals between measurements (typically 5 to 10 minutes). They don't scale well to shorter intervals, which makes them unsuitable for monitoring fine-grained phenomas during experiments. Additionally, most of them rely on *RRDtool* to store metrics and generate figures, which is a suitable tool for the infrastructure monitoring use case, but not well suited at all for generating publication-quality figures.

More recently, with the emergence of elasticity and cloud infrastructures, more modern infrastructure monitoring tools were designed, such as Google's *Borgmon*, *Prometheus* or *InfluxDB*. *Prometheus* and *Simple Network Management Protocol (SNMP)* are used in [19] to build an agent-less monitoring system. In the upcoming section we will describe our own effort to base our framework on *Prometheus* and *InfluxDB*.

TESTBED-PROVIDED MEASUREMENT SERVICES. Some testbeds provide services to expose some metrics that would otherwise not be available to experimenters. For example, the *Grid'5000* testbed [9] provides *Kwapi* [32] for network traffic and power measurements, collected respectively at the network equipment and at the power distribution unit. *PlanetLab* [31] used *COMon* to expose statistical information about the testbed nodes and the reserved slices. In general, these services are limited to very specific metrics. Thus, experimenters have no permission to add their own metrics or to otherwise customize these services for their experiments. Those services should rather be considered as potential additional sources of information for an experiment monitoring framework.

INSTRUMENTATION FRAMEWORKS. There are very few attempts at providing frameworks that address the specific needs of experimentation. *Vendetta*[109] is a simple monitoring and management tool for distributed testbeds. It runs an agent code on every node to be monitored to parse the experiment events before sending the results to the central sever which does the visualization mission. *Vendetta* has no mechanism to implement the starting and the ending time of experiments, so the researcher must manually track the experimental timing to extract the collected metrics. OpenStack's *Gnocchi*[1] is a monitoring solution for metrics aggregation. Testbeds like the *Chameleon* testbed uses *Gnocchi* as a monitoring service[2] and provides images with preinstalled *collectd* to forward metrics. Like *Vendetta*, *Gnocchi* cannot precise experiments time boundaries and does not provide visualization. Also, analyzing fine-grained phenomena is not ensured due to metrics aggregation.

Another solution is *ORBIT Measurement Library (OML)*[3] [137], which is closely related to the *OMF* [108] testbed management framework. With *OML*, the experiment components stream their measurements towards an *OML* server. The server creates a SQL database to store the metrics of each experiment. The process of using *OML* is experiment-dependent: several steps are required to modify the experimental code to define the measurement points. The only way to access experiment metrics is to query the experiment database. Overall, *OML* has seen rather low adoption, even if some testbeds like *Fibre* [121], *IOT-Lab*, or *NITOS* [48] support it. Its development seems to have been stalled (last changes on GitHub in 2015).

We perform a comparison of the tools presented here with the requirements discussed in Section 5.2 to determine if all requirements are satisfied. As can be seen in Table 5.1, the existing tools fail to match all requirements to act like *EMFs*. To emphasize, none of the tools recognizes experiments on any experimental environments which is the first requirements to work on *experiment* basis. Therefore, the lack of *EMFs* triggered the design of our own solution, described in the next section.

1 Gnocchi Documentation: https://gnocchi.xyz/
2 Metrics using Gnocchi in the Chameleon tested: https://www.chameleoncloud.org/blog/2018/02/15/metrics-using-gnocchi/
3 https://github.com/mytestbed/oml

Table 5.1: Identified requirements for experiment monitoring frameworks (EMFs) vs related work

	Infrastructure monitoring tools e.g. Munin	Testbed-provided measurement services	Vendetta[109]	OpenStack's *Gnocchi*	OML[137]
Experiment-focused	-	-	-	-	-
Independent of experiments	+	+	-	+	-
Independent of testbeds services	+	+	-	+	+
Scalability	+	+	-	+	+
Low impact	-	+	-	+	-
Easy deployment	+	-	+	+	+
Controllable	-	-	+	+	+
Real-time monitoring	+	+	-	-	-
Producing publication-quality figures	-	-	-	-	-
Archival of data	+	+	-	+	+

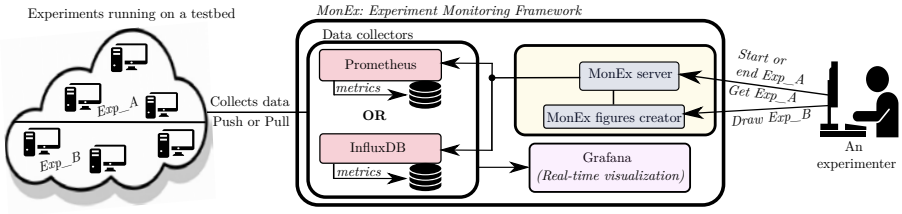


Figure 5.1: Overall design of the *MonEx* framework

5.4 *MONEX* FRAMEWORK

This section introduces *MonEx*, our integrated Experiment Monitoring Framework. Inspired by the *Popper* convention [64], we reuse some off-the-shelf monitoring technologies that fit into *MonEx* design rather than making new ones, and then build on top of them to adjust to the specific requirements of experiment monitoring. Thus, *Prometheus* and *InfluxDB* are used as *data collectors* while *Grafana* is used for *real-time visualization*. But those off-the-shelf components are complemented with custom-built components. First, *MonEx server* brings the experiment process to the monitoring solution, by enabling the experimenter to specify the experiments' start and end time in order to link metrics to specific experiments (allowing the extraction of an experiment's metrics, or to refer to a specific experiment for analysis or comparison purposes). Second, *MonEx figures-creator* makes it possible to automatically extract metrics for a specific experiment, and create publishable figures. Figure 5.1 shows the overall design of the *MonEx* framework.

MonEx is designed as a command line tool. Experimenters can directly interact with the *MonEx-* server or the figure-creator. After setting up an experiment, two calls to the *MonEx server* are issued at the beginning and the end of that experiment, to allow specifying the experiment time boundaries. The server also deals with the experimenters' requests to query their experiments. *Prometheus* and *InfluxDB* are used to retrieve the experiment metrics from the execution environment, representing the main data source of *MonEx*. Hence, experiments can use an appropriate monitoring technique. Indeed, *MonEx* covers *agent or agent-less monitoring, and pull or push monitoring* thanks to these data collectors (details in Section 5.4.2). Furthermore, *Grafana* is used to visualize the experiment metrics at runtime by connecting to the data collectors as a consumer. At last, when the experiment is accomplished, *MonEx server* is able to produce an output file containing the target metrics of a given experiment. Eventually, *MonEx figures-creator* either exploits that file (e.g. in case of running in another environment), or connects directly to the *MonEx server* in order to generate publishable figures.

The components of *MonEx* are described in details in the following sections.

5.4.1 *MonEx server*

MonEx HTTP server is built to handle the time boundaries of experiments as well as manipulating their metrics. It exposes different interfaces to receive notifications about the start and the end time of experiments, to query the experiments metrics, and to remove experiments from its list. Each experiment sends at least two *HTTP* requests: *start_xp* to indicate its intention to use *MonEx*, also bringing a name for this experiment to be distinguished by, *end_xp* to notify the server about the experiment termination time. The requests could be integrated inside the experiment code to be more dynamic when declaring the time boundaries, especially for short-length experiments. *MonEx server* is also used to query the experiment metrics using *get_xp* request. This request asks the *MonEx server* to expose the desired metrics into a *CSV* file. For example, to export a metric named *mymetric* of the experiment *myexp* into a *CSV* file, the following command could be used:

Listing 5.1: Enabling *MonEx* to start monitoring an experiment

```
curl "htttp://MonExServer:5000/exp/myexp"
-X GET -d '{"metric":"mymetric"}' > file.csv
-H "Content-Type: application/json"
```

MonEx server requires a configuration file which describes the data collector in use. The configuration file helps experimenters to send specific commands during communication. The server could also connect to several instances of the data collectors simultaneously, enabling experimenters to interact with multiple data collectors (e.g. experiments running in different physical sites of the same testbed where each site provides its data collector).

MonEx server tasks can be differently achieved by adding a control metric into *Prometheus*. However, this alternative has various limitations. Firstly, it will not scale as every experiment will require an independent instance of *Prometheus* to decode the added control metric. Thus, it will be challenging to have a service for monitoring experiments made available to all testbed users. Secondly, it limits the use of the underlying monitoring system to only *Prometheus*, ignoring the experiments that use other collectors. Thirdly, even if *Prometheus* has a lot of ready-to-use exporters, modifying each of them by adding control-metrics will prevent experimenters from using them directly. Taking these limitations into account, we choose to keep track of experiments via a dedicated server.

5.4.2 *Data collectors used by MonEx*

MonEx supports the use of either *Prometheus* or *InfluxDB* in order to cover all monitoring techniques. *Prometheus* is a monitoring system with alerting and notification services and a powerful querying language which allows creating compound metrics from existing ones. It is optimized to pull numerical metrics into a central server, but not to scale horizontally or to support non-numerical metrics as *InfluxDB* does. *InfluxDB* is a chronological time series

Table 5.2: A comparison of the data collectors employed by the *MonEx* framework

	Prometheus	InfluxDB
Data collection technique	Pull, push also possible via Prometheus-pushgateway	Push
Supported data types	Numerical metrics	Numerical, strings, and boolean metrics
Supported resolution	Up to one second	From milliseconds to nanoseconds
Horizontal scalability	Not really, only using independent servers	Yes, cluster of InfluxDB[4]
Generate derived time series	Yes	No

database for storing experimental metrics with a timestamp resolution that scales from milliseconds to nanoseconds. In *MonEx*, both could be mutually or simultaneously used regarding the experiment need. Indeed, both data collectors provide similar services regardless of their differences. Table 5.2 presents a comparison of *Prometheus* and *InfluxDB* over some selected features.

Although *Prometheus* is the default data collector in *MonEx*, its differences with *InfluxDB* favorite this latter for specific use cases. Firstly, *InfluxDB* fits better for the experiments that send their metrics in variable-time intervals since *Prometheus* still needs to pull the data regarding his scraping interval (even if that makes no sense for the experiment). For example, pulling the metrics every second is not significant for the experiment that generates its data at random time intervals, so pushing them into *InfluxDB* whenever the experiment has new data is more preferable. Secondly, as it follows the *pull-based* approach, *Prometheus* is not able to collect data from the experiments with high-frequency measurements since its scraping interval does not go beyond one second (it is also true if *Prometheus-Pushgateway* is used along with *Prometheus*). Thus, using *InfluxDB*, which supports pushing data at a high scale, is a robust solution to prevent any data loss during such experiments.

5.4.3 *MonEx figures-creator*

This component is essential to exploit the monitoring results for creating publishable figures. It is a tool that deals with the export of an experiment's data from MonEx into a format (CSV) that is widely supported by tools typically used to prepare figures (*R*'s *ggplot*, *gnuplot*, *matplotlib*, *pgfplot*, etc.). It

4 This feature is provided in the commercial version of InfluxDB

also includes direct support for generating figures using R, covering a wide range of standard figures (e.g. X-Y figures, stack figures, multiple-Y figures, ..., etc).

5.4.4 *Real-time visualization*

MonEx uses *Grafana* for real-time visualization which provides a modern web-based interface. *Grafana* consumes the available metrics originating from the active data collecter (*Prometheus* and *InfluxDB*), and then tries to show the metrics on its interface as a time series. Doing so can lead to revealing unexpected behaviors in the experiment as faster as possible. However, the experiments with high-frequency measurements trigger a trade-off with real-time visualization as they might impact the experiment resources by producing a massive volume of data. Therefore, such experiments should be configured in one of two possible ways. First, it can either push its metrics entirely at the end of the corresponding experiment, making this service totally unusable. Second, it can also push the metrics over periodic chunks to still benefiting from this service with reduced precision.

5.5 USING MONEX IN REALISTIC USE CASE EXPERIMENTS

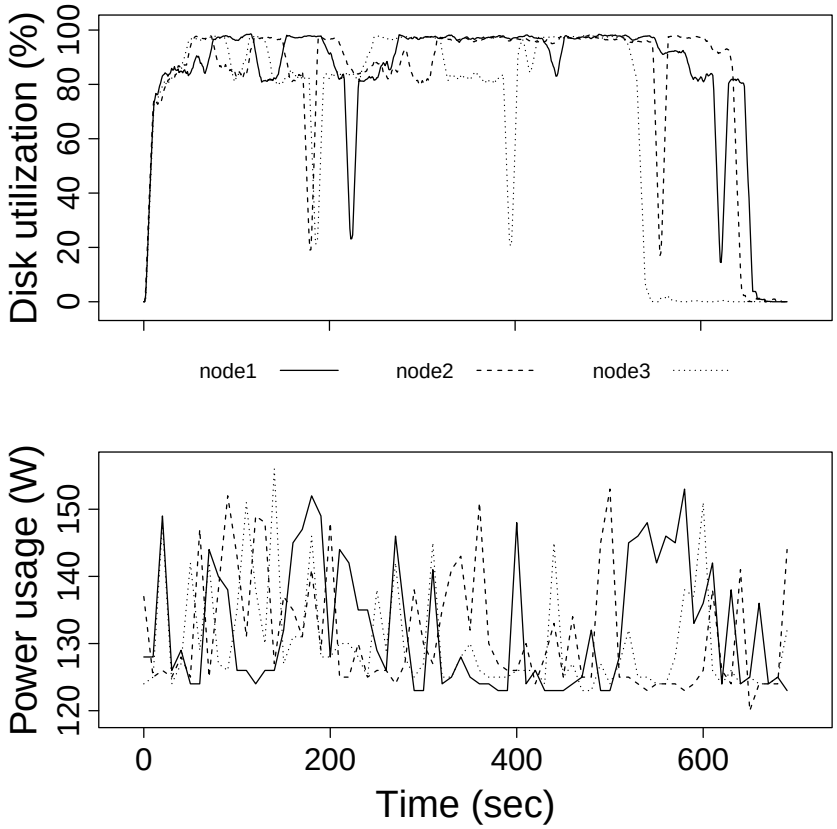
This section highlights through experimentation the ability of *MonEx* to cover all the requirements listed in Section 5.2. It describes three use case experiments while each experiment stresses partial requirements due to its specific objectives and nature. Indeed, the focus of the first experiment is on the flexibility of *MonEx* to pull data of diverse metrics using different exporters, while the second experiment focuses more on scalability and using custom exporters. Although the monitoring technique used in these two experiments is *pulling* experiments' data regularly over precise time units, the third one brings more focus on time-independent metrics with high-frequency of measurements impossible to be monitored similarly as the others. In this case, the experiment takes in charge the responsibility to push its data to the *MonEx* framework. Ultimately, all experiments are performed on a homogeneous cluster of the *Grid'5000* testbed.

5.5.1 *Experiment 1: Cluster's Disk and Power Usage*

The goal of this experiment is to evaluate the *disk utilization* and the *power usage* of a three-shard cluster of *MongoDB*, while performing an indexation workload over a Big Data collection with 80 GB.

Prometheus SNMP Exporter & Prometheus node exporter are used to tackle the cluster power consumption and the disk utilization metrics, respectively. On the one hand, *Prometheus SNMP Exporter* is using *SNMP* on the power distribution units (PDUs) to obtain the power per outlet. It involves adding the *PDUs* addresses to the exporter configuration file for being able to query all nodes outlets. Given that, the exporter becomes able to get the power of the

(a) Real-time monitoring using *Grafana*. The dotted, vertical line indicates the start time of the experiment. Obviously, such figures would not meet the expectations of scientific publications



(b) Figure produced by *MonEx figures-creator*

Figure 5.2: Disk utilization affecting the power consumption while indexing data over a three-nodes cluster of *MongoDB*

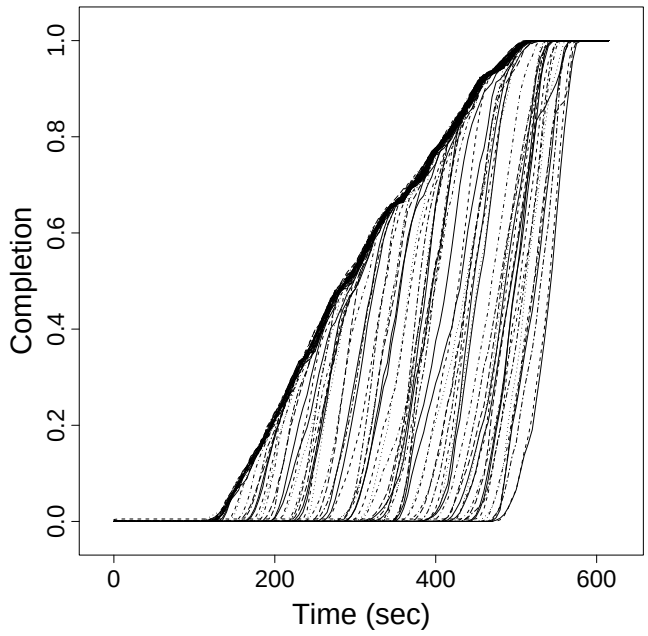


Figure 5.3: Torrent completion of a 500 MB file on a slow network with one initial seeder and 100 peers entering the network over time (one line per peer)

cluster machines regardless that it is only installed on one machine (*agent-less monitoring*). This reduces the impact on the environment as it asks a *PDU* for a bunch of outlets rather than issuing one request per outlet. One the other hand, *Prometheus node exporter* is installed on each machine (agent monitoring) to report the *disk* usage per machine.

To allow using *MonEx* for monitoring this experiment, only two instructions are added into the experiment script in order to notify *MonEx* about its start and the end time. As shown in Figure 5.2-a, we can check in real-time how our experiment is behaving. This helps as a safety step for detecting issues that might require to restart the experiment. In addition, we obtain our target metrics by sending a customized *get_xp* request to the *MonEx server*. *MonEx figures-creator* is then used to generate a publishable figure that contains the target metrics. Figure 5.2-b contains three colored curves that represent the disk utilization and the power usage of the three-nodes cluster.

5.5.2 *Experiment 2: Many-nodes Bittorrent download*

We revisit the torrent expriment covered in [95] using *MonEx*. Monitoring the torrent completion of a given file is the main metric of this experiment. A seeder with a 500 MB file is created and multiple peers seek to download the target file. A mesh topology is used for connecting the seeder/peers, while *Transmission* is used as a torrent client for the peers.

The network and the peers are emulated by *Distem* [123], so the experiment runs independently from the testbed topology. The seeder bandwidth is lim-

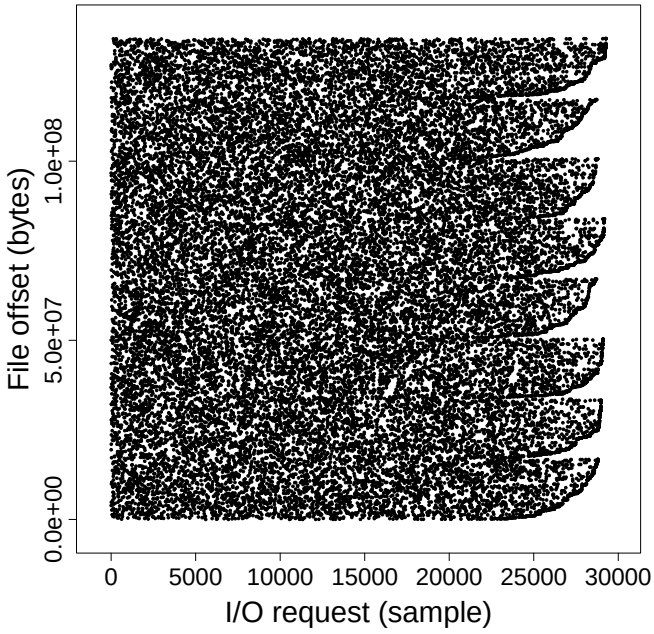


Figure 5.4: I/O access pattern of a 140 MB file read using *randread* mode of the *fio* benchmark. Each access offset is recorded and returned using *IOscope* tool described in the previous chapter

ited to 5 KB/s while this of peers is limited to 30 KB/s. Each peer resides on a virtual node, and all nodes are increasingly connected with a constraint that a new peer is entering the network every 4 seconds until the number of peers reaches its maximum (100 peers).

The experiment begins when the seeder shares the target file by notifying the tracker. It terminates when all peers have that file data. To obtain the completion of the target file, we query the *Transmission* API using our own metrics exporter (about 10 lines of *Python*). The exporter is instantiated to run on each virtual node, while *Prometheus* pulls periodically their data. *MonEx* has a minimal impact on the experiment resources as there is another dedicated *VLAN* in use for the monitoring traffic. This experiment shows how *MonEx* is scalable, as *Prometheus* pulls the experiment metrics from about a hundred of nodes without any overflow.

MonEx is then used either to produce a *CSV* file containing the experiment metrics selected by the experimenter, or to obtain directly a ready-to-publish graph, as shown in Figure 5.3.

5.5.3 *Experiment 3: Time-independent metric*

This experiment evaluates how a data file is accessed during a workload execution. We use the *Fio* benchmark to generate an I/O access pattern over a given file. In parallel, we resue*IOscope* tool (see previous chapter) to uncover this access pattern. The target metric is the file offsets. If the access is random,

we are expecting to see a shapeless view of file offsets versus the sequences of the I/O requests.

This experiment is challenging in both scalability and controllability. Firstly, a pull-based monitoring cannot be used since the experiment has high frequency measurements. Hence, a scraping interval even of one second might not catch all events (data exposed to be lost). Secondly, the target metric does not rely on the timestamps, but rather on the order of I/O requests accessing the file. Hence, every I/O request is significant for understanding the overall access pattern. For these two reasons, we use *InfluxDB* rather than *Prometheus*. We locally collect the massive I/O requests before pushing them at once to *InfluxDB* at the end of experiment. Figure 5.4 shows I/O patterns of a random read workload. The file offsets are totally shapeless regarding the sequences of the issued I/O requests.

5.6 SUMMARY

Most computer experiments including big data experiments include a phase where metrics are gathered from and about various kinds of resources (e.g., IoT sensors, social media websites, records, and logs). This phase is often done via ad hoc scripts, manual and non-reproducible steps, a time-consuming and error-prone process. On experimental environments like testbeds, infrastructure monitoring tools facilitate collecting experiments' data to some extent. However, there is no conventional way for so doing, and there is still much work to be done to leverage the monitoring activity for monitoring experiments. For example, observability tools still require to allow capturing user experiments and helping experimenters to analyze and prepare results for the publication process. Hence, there is a need to improve observability in order to fit the experiments workflow.

In this chapter, we first defined the needed requirements to build an *experiment monitoring framework* (*EMF*) with the aim to extend the infrastructural monitoring activity for working on experiments basis. We then leveraged these requirements and some recent infrastructure monitoring solutions to introduce the *MonEx* framework. Through use case experiments, we showed how *MonEx* reduces the experimenters' effort by encompassing all the steps from collecting metrics to producing publication quality figures, leaving no places for manual and ad hoc steps that were used to be performed in practice.

MonEx has two impacts on the way we perform experiments. Firstly, it pushes towards the repeatability of experiments' analysis and metrics comparison. That is thanks to its abilities to both, separating the phase of collecting metrics from experiments and archiving them per experiment. Secondly, as *MonEx* puts the experiment at the center of the monitoring workflow, focusing on how the experiment results are obtained rather than where the experiment runs, we are a step closer to better experiment portability and reproducibility.

Part IV

IMPROVING THE DESIGNS OF BIG DATA EXPERIMENTS

He uses statistics as a drunken man uses
lamp-posts—for support rather than for illumination

—Andrew Lang

6 REDUCTION OF EXPERIMENTAL COMBINATIONS

6.1 INTRODUCTION

Practicing the concepts of *Design of Experiments* (*DoE*) is not highly used in computer science, unlike other domains such as operational research. Although statistics can be a useful tool, to help validate results by excluding the effects of chance or other unknown factors, and ensuring that all results are generally coherent, experiments in computer science articles are designed and analyzed with only limited use of statistics like employing computing averages and medians. Moreover, when it comes to design experiments, the majority of researchers still use the naive approach represented by the full factorial design, which, of course, needs more experiments to accomplish an ongoing investigation. Hence, continuing to rely on such a design for experiments with a high number of experimental factors[1] is like swimming upstream. It costs time, efforts, and experimental resources yet before reaching a coherence design with the factors that have the most critical impacts on results.

In parallel, the era of big data makes it unimaginable to understand the results of big data experiments without relying on statistics [20, 34]. We start to see more exploitation of statistics in big data research, but the majority of them are restricted to solving issues related to big data analysis. To name a few challenges, improving the data analysis phase by statistical design [42], introducing models to cope with the variety of data [126], and trying to eliminate the bias during experimentation [103]. However, there is not work to tie together the importance of experimental designs to reduce resources utilization. Doing so is not only advantageous to saving experimental resources; this also reduces experimenters' time, and efforts and above all obtains a final experimental design as fast as possible. Provided that, we are motivated to studying the feasibility of employing experimental designs for minimizing resources utilization.

In this chapter, we explore how statistical methods such as a fractional factorial design[8, 18, 55, 61] can help to drastically reduce the number of required experiments while still providing a similar amount of information about the impact of the experimental factors on results. Indeed, we do not introduce a new experimental method, but rather encouraging to use such a

1 Also known as experimental variables and parameters in the literature

practice in the big data community. In section 6.2 we introduce the fractional design and the aspects that allow fitting into obtaining faster designs. In section 6.3 we provide the core of this chapter which is the experimentation part, highlighting the advantages of such a design over realistic experimental configurations and scenarios. Section 6.4 states the related work while Section 6.5 brings out a closing summary.

6.2 FRACTIONAL DESIGN FOR EXPERIMENTATION

The fractional factorial model is originated from the full factorial design. It takes fewer number of experiments than its predecessor thanks to offering flexibility to experimenters to change the number of involved experiments. For simplicity reasons, we limit the illustration here to a two-level fractional design 2^{K-P}, where K is the number of two-level experimental factors in a given experiment, and $K - P$ is the number of required experiments. Having $K = N$ with N as a positive number, the experimenter can assign to P any numerical value larger than zero and lower than N to reduce the number of experiments. For instance, if $K = 10$ & $P = 5$, the number of experiments will be highly reduced compared with the full factorial design (2^K). However, If P equals to zero, the fractional design then backs to the original factorial design where all combinations of experimental factors will be studied. Moreover, having either quantitive or qualitive factors in the study does not imply any modifications. The two-level values of quantitive factors will be represented by whatever two values to serve as high and low [18] (e.g., network throughput factor can be represented by 10 MB/s and 1 GB/s). In contrast, any qualitative factor will be represented by two distinct values that refer to two different types (e.g., system's version factor takes $v1$ and $v2$ as values).

To help to illustrate how the fractional design reduces the number of experiments' runs, we should first discuss the following example of full factorial design. Hence, if we have four experimental factors of two-level A, B, C, and D, the full factorial design 2^4 will analyze the results of the main factors as well as the results of the following interactions:

- Six second-order interactions: AB, AC, AD, BC, BD, and CD
- Four third-order interactions: ABC, ABD, ACD, and BCD
- One fourth-order interaction which is $ABCD$

However, analyzing all those interactions is not always useful, especially when having near-zero dependency between the experimental factors. As a result, the main idea behind reducing the number of experiments by the fractional design is that it ignores analyzing some interactions between the experimental factors under study. Indeed, it creates a set of experiments that is reduced compared to the one obtained with a full factorial design, but that still, by construction, enables the study of the effect of each factor, even if information about the interactions between some factors is lost.

Table 6.1: A design table for a 2^3 experimental design

The average response I	A	B	C	AB	AC	BC	ABC
1	-1	-1	-1	1	1	1	-1
1	-1	-1	1	1	-1	-1	1
1	-1	1	-1	-1	1	-1	1
1	-1	1	1	-1	-1	1	-1
1	1	-1	-1	-1	-1	1	1
1	1	-1	1	-1	1	-1	-1
1	1	1	-1	1	-1	-1	-1
1	1	1	1	1	1	1	1

In Section 6.2.1 we explain how the fractional factorial design does that operation.

6.2.1 *Allocation of Factors over Models' Interactions*

In table 6.1, we show a full factorial design of an experiment with three experimental factors 2^3 which are A, B, and C. The number of lines in that table refers to the total number of experiments to be executed where each line corresponds to an experiment instance. For example, the first experiment requires a combination of the A, B, and C factors where all of them are represented by their low-level values. Moreover, the table should satisfy three matrix properties to validate their values [61]. It is verifiable in the table that 1) the sum of every column except I equals to zero, 2) the sum of the product of any two columns except I also equals to zero too, and 3) the sum of absolute values of any column equals to 2^3 with $K = 3$.

Fractional design cannot reduce the number of experiments without a *confounding*. This latter appears when the effects of some interactions cannot be isolated from the effects of some experimental factors. To give a detailed example, let us suppose that we have an investigation over four two-level factors, namely A, B, C, and D. Instead of performing sixteen experiments with full factorial design (2^4), we can reduce that number of runs to only eight with a fraction design 2^{K-P} with $K = 4$, and $P = 1$. Doing so firstly implies building the design around $K - P$ primary factors, so having the same design as shown in Table 6.1 if the primary factors are selected to be A, B, and C. Secondly, we need to sacrifice any interaction among those presented in the table to put the fourth factor, which is D. Hence, we can say that an interaction is confounded with the factor D i.e., their effects cannot be separated without a full factorial design.

The possibility of allocating the remaining factors on the interactions is not unique since several configurations can be used. In our example, we can have

either $D = AB, D = AC, D = BC$, or $D = ABC$. However, the best design is that which increases the number of confounding terms i.e., the higher the number of confounding terms, the better the proposed model. To illustrate, let us consider two examples. The first one is the design with $D = ABC$. It gives out a confounding equation with four terms $I = ABCD$. The second design is when having $D = AB$ which gives the following confounding equation[2]: $I = ABD$, which has three confounding terms. The best design is the first one as it has a fourth-order confounding. The logic behind that selection can be explained as follows. The more the order of an interaction increases, the more that interaction becomes negligible so better to be confounded with the remaining factor. However, in practice, it depends more on the experimental factors, so irregular cases of that rule still exist.

6.3 USE CASE STUDY: MOVING DATA OVER A WIDE-AREA NETWORK

The Big Data movement has put a lot of focus on the importance of data, and on the importance of proper data management approaches. It encompasses changes in the way we store data, by moving from traditional expensive storage arrays to more cost-effective, scalable and flexible Software-Defined Storage solutions such as Ceph [156], GlusterFS [3] or iRODS [107]. At the same time, there is a push towards moving our computing and storage facilities from small on-premises datacenters to larger datacenters or Cloud offerings in order to reduce costs and increase reliability and elasticity.

This move towards remote and centralized storage resources raises a number of challenges when aiming at maintaining high-performance access. The protocols that were designed to access storage resources over a local network are not necessarily able to perform efficiently when network latency increases due to the need to transfer data over hundreds or thousands of kilometers. Modern storage systems often provide an object-based interface, where storage objects are read or written using GET/PUT methods that are able to perform well over high-latency network. However, most applications still have not been ported to such interfaces, and still rely on traditional POSIX-like directories and files access. Several designs are possible to provide that interface, such as relying on a kernel driver on the client side (which can be a limitation when supporting a wide range of systems), or using a FUSE filesystem (which has performance impacts). As a result, most modern Software-Defined Storage solutions still provide a traditional *Network File System (NFS)* or *Common Internet File System (CIFS)* interface (e.g. GlusterFS), that provides an unintrusive way to access storage from clients.

In this section, we describe the experiments done over NFS while being used in a high-latency *Wide Area Network (WAN)* setup. Besides making a performance evaluation of NFS in such a context in Section 6.3.2, we provide statistical results in Section 6.3.3 to highlight the advantage of using fractional design to decrease the number of experiments.

2 Both equations are verifiable in Table 6.1

3 https://www.gluster.org/

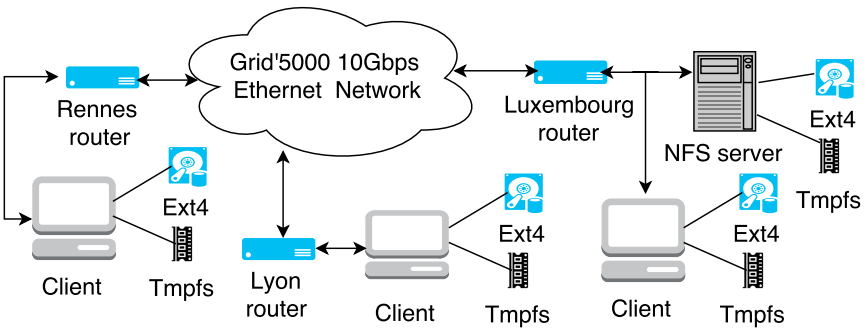


Figure 6.1: Experiment topology of moving data over several physically-distributed sites on the *Grid'5000* testbed

6.3.1 *Experimental Setup*

The experiments are performed on the *Grid'5000* testbed [9] as it has several physically-distributed sites. Among them, three sites are used as shown in Figure 6.1. We intend to deploy an NFS server on only one site in contrary to NFS client which will be deployed in all three sites. Thanks to *Grid'5000* capabilities, we deploy our own software environment, which is Debian *jessie-x64-nfs* image with Linux 3.16.0 on all sites and set up NFS server and clients as described above.

We plan to use two versions of NFS in this experiment. To allow using NFSv4 beside NFSv3, we configured NFS on the clients by enabling the name mapping daemon *IDMAPD*, which is required by NFSv4. Besides, we make sure that the server accepts 1M as I/O size value be verifying it on *nfs_xdr* header file. Moreover, all nodes (server and clients) are connected over a 10 GB/s network (both the Ethernet network on each site and the *Grid'5000* inter-site network). This setup is representative of one where an NFS client accesses a server over the Internet, with a high-bandwidth link. It also useful to investigate the impact of latency on the performance as we can vary network latency on both local and remote machines.

NFS performance can be affected by several experimental factors and configurations. Here, we put the light on six potential factors that can have an impact on NFS performance. All of these factors are manipulated on NFS clients without making any change on the NFS server. Moreover, three factors are NFS parameters which are changeable through the *mount* command. All factors are listed below, starting by the NFS parameters:

NFS Version: [*NFSv3, NFSv4*]. NFSv3 is the stable and widely used version [144]. Besides, NFSv4 is the latest version released with promising specifications [135] such as having parallel high bandwidth access (pNFS). Testing both versions is important to explore if changing from one version to another will lead to change in performance.

Synchronicity of I/O operations: [*sync, async*]. This factor is also known as blocking/non-blocking I/Os. This is used on the client side (see `nfs(5)`)

to control the behavior of the client when issuing writes. In asynchronous mode, the client may delay sending write requests to the server, and return from the `write` request immediately. In synchronous mode, the client will wait until the server replies to return from the `write` request. This option should not be confused with the server-side `sync/async` option (see `exports(5)`) which controls whether the server must wait until data is written to stable storage (e.g. HDD) before issuing a reply to the client.

NFS I/O size: [*64 KB*, *1 MB*]. Those values are used because *64KB* represents the default value of I/O size of several NFS implementations while *1 MB* also represents the maximum I/O value that is used by most NFS servers.

Storage type: [*Hard disk – ext4*, *In-memory – tmpfs*]. We use normal *Grid'5000* compute nodes with a single hard disk drive and not nodes which would provide good I/O performance by spreading data over several disks, as such nodes are not available on all needed sites. Thus, the local hard disk drives of our nodes are likely to be a significant performance bottleneck. In order to focus on the protocol aspects of performance rather than just on the performance of local storage devices, we do not limit ourselves to experiments using data stored on *ext4* filesystems backed by hard disk drives, but we also perform experiments using in-memory data, by setting up a *tmpfs* filesystem on nodes.

File size: [*100 MB*, *5 GB*]. The file size is often significant during data transfer in a wide-area network. Indeed, the performance may change negatively with small files as they may be spread out the disk which requires more seeks to retrieve them than retrieving big files. In the context of big data experimentation, we use both sizes above mentioned to represent small and big files.

Latency/Latencies: [*0.027 ms*, *6.87 ms*, *13.9 ms*]. Including many physically-distributed sites in the experiments implies that each site may be reached with different latency depending on the network implementation and configuration. The latency values here represent the measured one-way latencies between the server, which is located in Luxembourg site (see Figure 6.1), and Luxembourg, Lyon, and Rennes clients, respectively.

Consequently, we have a total of 96 possible combinations of factors [4]. This combination does not incur any special treatment when performing experiments with full combinations of factors to study the performance. However, this may require to unify the statistical model or using a mix-level model when applying a fractional factory design (see Section 6.3.3). A possible combination of an experiment looks like: *[NFSv3, sync,* 64 kB, *tmpfs,* 5 GB, *Lyon client]*. Furthermore, each combination will be tested with sequential

4 Five two-level factors & one three-level factor ($2^5 \times 3^1$)

reading and *writing* access patterns. We use the *dd* command with */dev/zero* as source and 1 MB as a block size to perform the writing operations. Similarly, the *cp* command is used to perform the reading operations.

The experiments are executed sequentially by changing the access pattern of each experiment, i.e., every experiment performs a write operation before reading the written file again. Then, the next experiment is similarly done, and so on. Moreover, to determine the exact read/write time of each experiment, all of them are executed with (1) preventing the cache effect on the server as well as on the clients, and (2) re-mounting the corresponding NFS directory after performing the pair (read and write) operations of each experiment. Indeed, this cleans out all historical operations on the server side, which means that every experiment is performed as it is the first one. Lastly, every experiment is performed five times to increase the results' reliability. Hence, the total number of experiments is 960 executions (96 experiments x 2 access patterns x 5 replications).

6.3.2 *Performance Results*

This section shows the performance results. They are presented by separating reading and writing results, as shown below.

6.3.2.1 *Reading results*

Figure 6.2 shows the reading results. It is plain to see that *tmpfs* provides much better throughput than *ext4* in all cases. This is expected as the moderate speed of HDD is behind the performance degradation when using *ext4*. Limiting the discourse to the results of *tmpfs* only, one can see that the best performance occurred when the files are read from the local client. That can enable us to confirm that the high latencies environment – represented by Lyon and Rennes sites – strongly affect NFS reading performance. As shown, when a client is changed from Luxembourg to Lyon, the throughput is reduced on all NFS versions by at least 8.6X, 7X on sub-figures 6.2-A and 6.2-B, respectively.

It is noticeable that not all of the experimental factor incur a change in performance results. For instance, changing NFS versions does not have a huge impact on the obtained results. We can assert that those versions have approximately the same level of sensitivity against the applied latencies despite having a slight decrease in performance in case of using NFSv4. In contrast, we found that the performance is slightly improved on *tmpfs* by increasing I/O size because larger I/O requests reduce the number of I/O requests, and thus the penalty induced by network latency.

6.3.2.2 *Writing Results*

The results of writing experiments are presented in Figure 6.3 and Figure 6.4 to separate the synchronous and asynchronous results.

As with reads, the results of sync writing (see Figure 6.3) also exhibit reduced performance when using an HDD, compared to when using *tmpfs*.

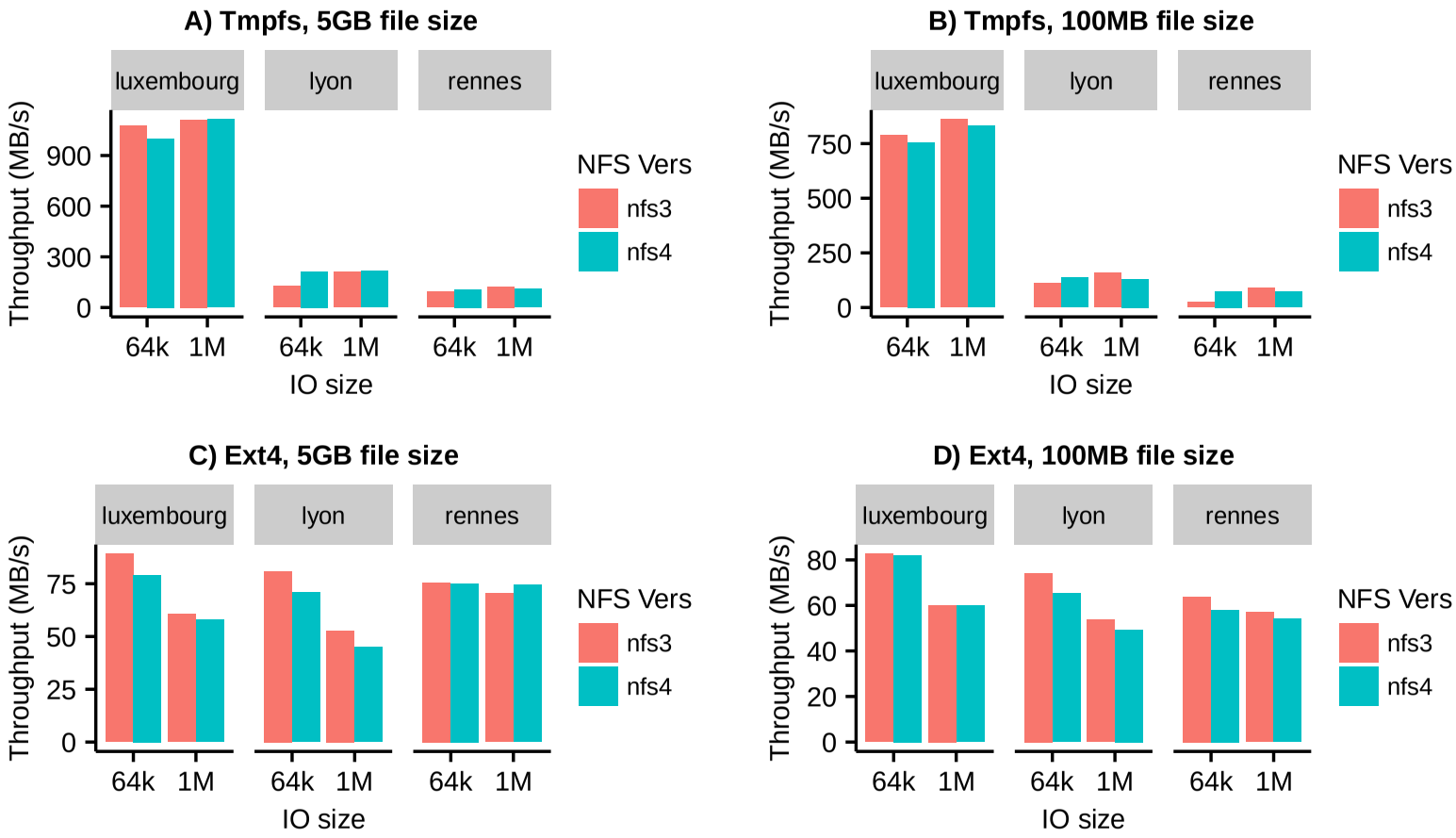


Figure 6.2: Performance results of reading data synchronously over a wide-area network using NFS. Sub-figures are distinguished by the used filesystem and the size of the retrieved data file

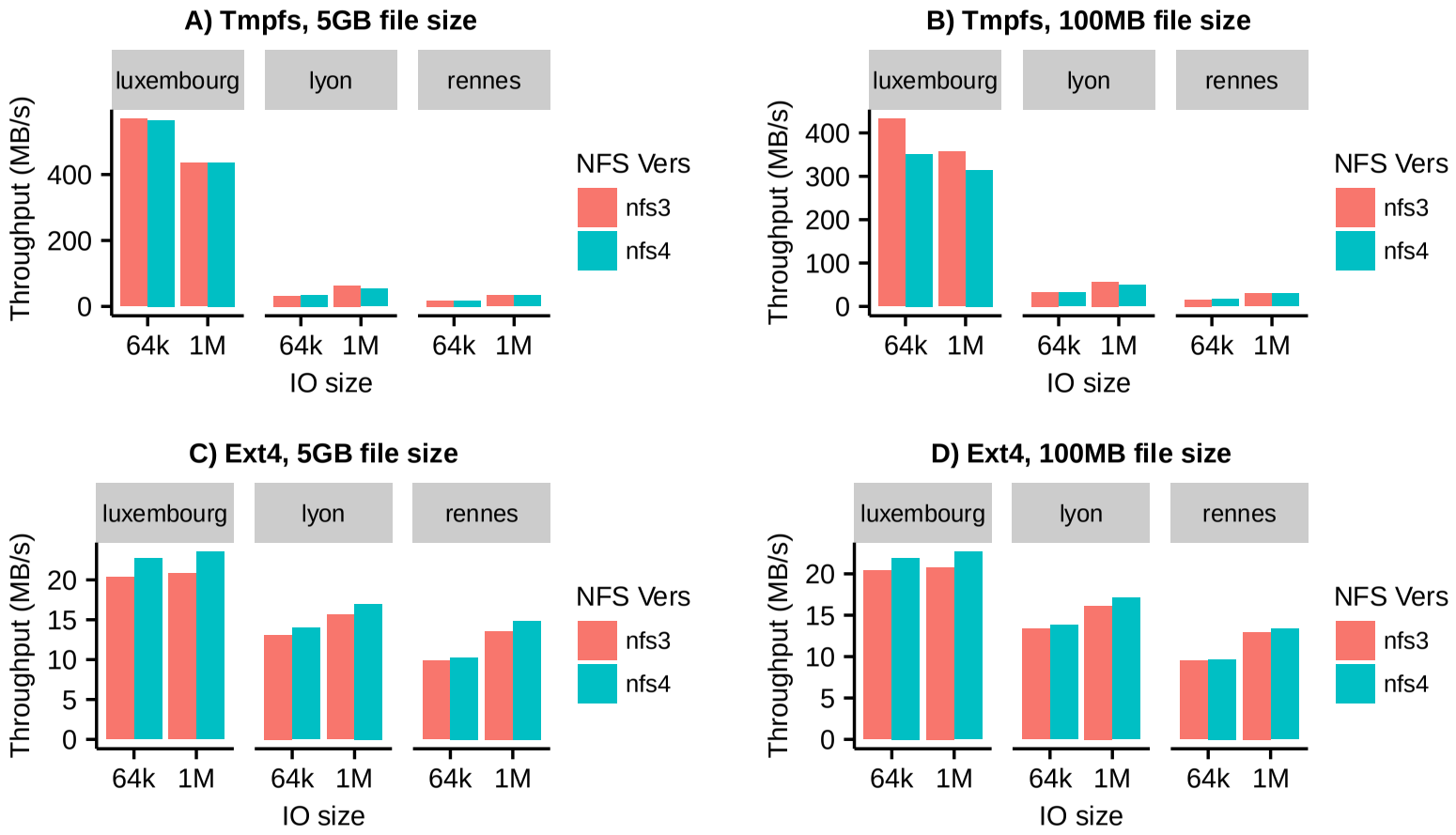


Figure 6.3: Performance results of writing data synchronously over a wide-area network using NFS. Sub-figures are distinguished by the used filesystem and the size of the retrieved data file

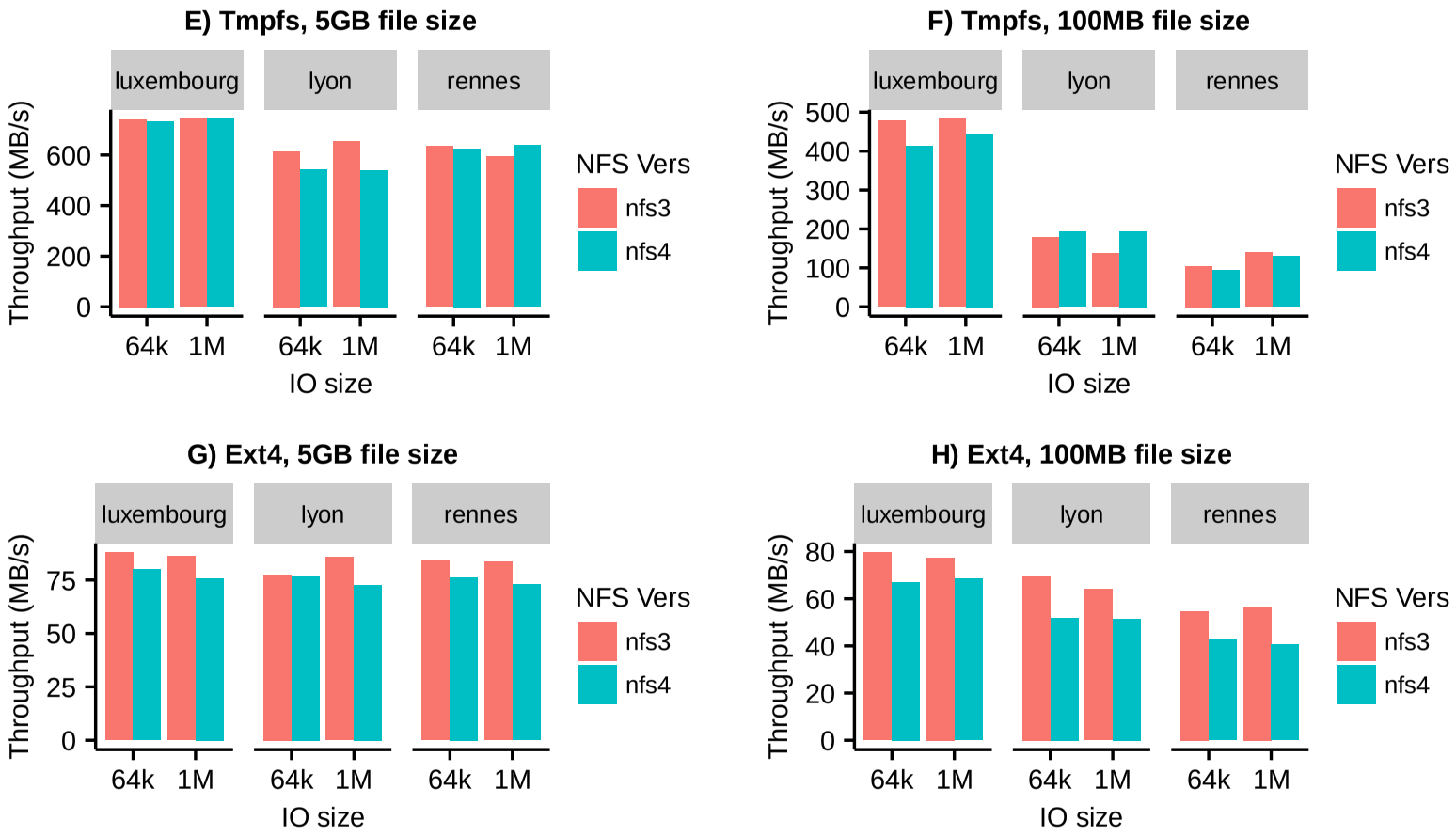


Figure 6.4: Performance results of writing data asynchronously over a wide-area network using NFS. Sub-figures are distinguished by the used filesystem and the size of the retrieved data file

Moreover, varying the latency shows a strong impact on results either when directly using the HDD through the *ext4* or when using a *tempfs*. In regard to the results of *tempfs*, the cost of changing the client from the local site (Luxembourg) to another one with *6.8 ms* as latency can decrease the throughput by 17.5x. We also found that NFSv3 shows better performance over NFSv4 when the local client writes 100 MB file on server memory. We hypothesize that this is an inconvenient of NFSv4's compound *Remote Procedure Calls (RPCs)* when performing write operations on a low latency connection and a high-speed storage material. In contrast, NFSv4 shows better performance when writing on HDD.

On the other hand, asynchronous writing results show an increased performance over the synchronous results on all experiments. The most significant impact is shown on remote clients. For instance, the highest throughput of Lyon client is raised to 18.9x using the *async* option on *tmpfs* and a 5 GB file. By having `write` requests return immediately, this mode can hide the latency between the server and the client. Of course, this could have an impact on data reliability as the notifications are sent back when the data is still on the server's memory and yet became persistent on the disk.

6.3.3 *Statistical Designs & Results*

Statistics can also contribute to a deeper understanding of the results by providing a way to identify the role of each factor and their interactions. It can filter out the experimental factors based on their impact on the experiment's results.

In this section, we show how factorial designs [8, 55, 61] could be used to plan experiments, and identify the relative importance of factors, and their interactions. The same experimental factors described in Section 6.3.1 are reused here with one slight modification. Instead of using three values for the experimental factor of *latency*, only the values of Luxembourg and Rennes sites are used here to allow using a unified design of two-level factors. This choice is utilized here to continue focusing on the benefits of fractional models for big data experiments with simple implementation. Otherwise, a factorial model with mixed-level designs should be used.

The factors applied in this statistical study are notated as follows:

- *A*: NFS version with (-1) NFSv3 , (+1) NFSv4
- *B*: Storage type with : (-1) *tmpfs*, (+1) *ext4*
- *C* : Sync option with : (-1) sync , (+1) async
- *D* : I/O Size with : (-1) 64 kB , (+1) 1 MB
- *E* : File size with : (-1) 100 MB, (+1) 5 GB
- *F* : Client Latency with : (-1) 0.027 ms , (+1) 13.9 ms

The results are described in two sub sections. In Section 6.3.3.1, we describe the results obtained by using a *full factorial design* i.e., results of evaluating

all combinations of experimental factors. This design is applied over two-level factors in order to measure the impact of each factor when its value changes from a *low level* to a *high level*, providing a way to make the first pass over all factors and decide which ones to exclude from further experiments because their impact would be limited. This requires 320 experiments, as will be explained later. In Section 6.3.3.2, we use a *fractional factorial design* to analyze the same factors using a reduced number of experiments (only 45 experiment), at the expense of not obtaining information about interactions between all factors.

Ultimately, we use the Student's t-distribution to calculate a 95% confidence interval of the effects of experimental factors and their interactions.

6.3.3.1 *Results of Full Factorial Design*

In this section, the full factorial design is used over: $2^k * r$ experiments where k is the number of factors ($k = 6$) and r is the number of replications ($r = 5$), that is, 320 experiments. We calculate the estimated behavior of the response variable y, which is the estimated time to read/write a file in a given experiment, using the following model:

$$y = q_0 + q_i x_i + q_j x_j + q_k x_k + q_{ij} x_i x_j + q_{ik} x_i x_k + q_{jk} x_j x_k + \ldots + q_{ijk} x_i x_j xk + \ldots + e$$

where q_0 is the mean response of the model, x_i represents the corresponding level of factor i (-1 or +1), $q_i \; with \; i \in \{A, B, \ldots, F\}$ is the impact of the factor i when changing its value from -1 to +1, $q_{ij} \; with \; i, j \in \{A, B, \ldots, F\} \wedge i \neq j$ represents the impact of interaction between the factors i and j, and e represents the model errors. The higher the absolute value of factor's or interaction's impact, the more the corresponding factor or interaction is critical for the experiment.

The result of applying this model on the writing experiments is shown in Figure 6.5. It shows the effects of all factors under study, and of their interactions. The x-axis of this figure represents the factors and all their different-order interactions. For example, *A*, *AB*, *CDE* represent *Factor A (NFS versions), the interaction between factors A and B (NFS versions, storage type), and the interactions between factors A, B and C (NFS versions, storage type and Sync option)*, respectively.

We found that the four main factors that make the most variation on the results are: 1) the file size, 2) the storage systems, 3) the sync options, and (4) the client latency. For example, the first case can be interpreted as follows: when the transferred file is changed from -1 (100 MB) to +1 (5 GB), it makes the most variation on the transfer time results and so on. Moreover, the results show that some interactions may have a stronger impact on results than primary factors. Indeed, this is the case with the interactions *BF*, *CF*, and *BCF*, which are more crucial than the factor of NFS version.

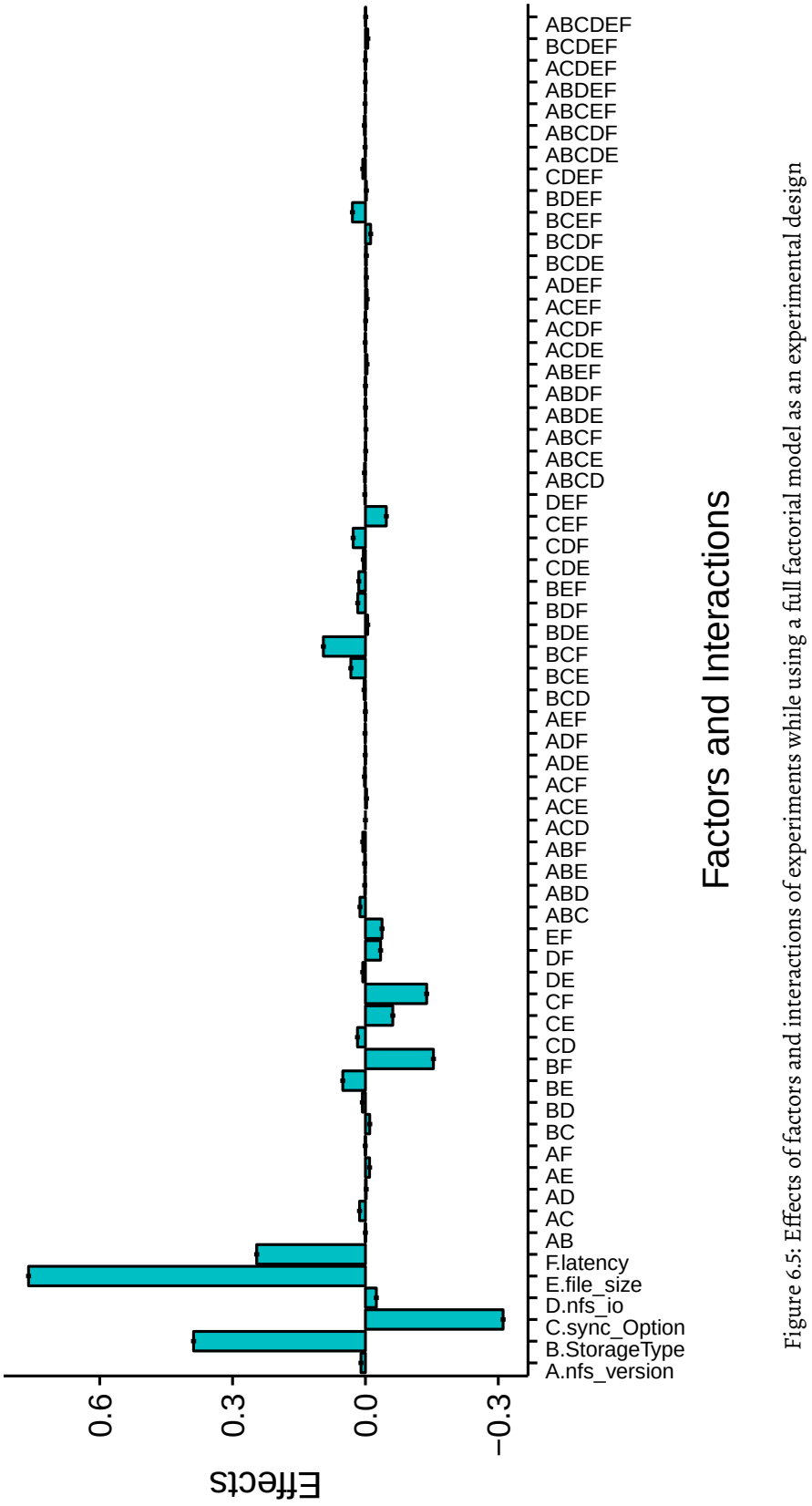


Figure 6.5: Effects of factors and interactions of experiments while using a full factorial model as an experimental design

Table 6.2: A design table for a fractional design with $K = 6$ & $P = 3$, and with three confounded factors

Basic factors			Confounded factors			Rest of interactions
NFS Version	Storage type	Synchronicity	I/O Size	File Size	Latency	-
A	B	C	D_{AB}	E_{AC}	F_{BC}	ABC
-1	-1	-1	1	1	1	-1
-1	-1	1	1	-1	-1	1
-1	1	-1	-1	1	-1	1
-1	1	1	-1	-1	1	-1
1	-1	-1	-1	-1	1	1
1	-1	1	-1	1	-1	-1
1	1	-1	1	-1	-1	-1
1	1	1	1	1	1	1

6.3.3.2 *Results of Fractional Factorial Design*

We prepare a design that requires only 12,5% of experiments compared with the number of experiments of full factorial design. That is done via assigning the value of three to P in fractional design annotation 2^{6-3}. Hence, the design table is fixed around three basic factors, which are selected to be A, B, and C. To determine what are the experiments maintained by our design, we fill out the truth table of A, B, and C factors, and then calculate their interactions of all orders. We then put the rest of factors D, E, and F over an equal number of interactions as shown in Table 6.2. In that design $A = ABD$, we decide to use the AB interaction for the factor D, AC interaction for the factor E, and BC interaction for the factor F. Note that, the effects of factors D, E, and F will be confounded with the interactions AB, AC, BC, which means that their respective effects on results cannot be separated. As a result, each row in that table represents an experiment. For instance, the first row involves running the experiment that has the factors A, B, and C at *NFSv3*, *tmpfs*, *sync* levels and the factors D, E, and F at 1 MB, 5 GB, *Rennes* levels, respectively.

The result of applying this model on writing experiments is shown in Figure 6.6. This result shows that the first four crucial factors that have a significant effect on the execution time of the performed experiments are the same factors that were obtained using the full factorial model in the previous section. As a result, this model gives the same conclusion but by reducing the number of experiments by a ratio of 87.5%.

6.4 RELATED WORK

The Network File System (NFS) was introduced in 1984 to allow sharing resources in a network for heterogeneous client machines [122]. Its first public version (NFSv2) had several limitations (no asynchronous write operations). NFSv3 (1988) overcomes this and added support for the TCP protocol. NFSv4 (2000) came with additional improvements such as compound RPCs to divide

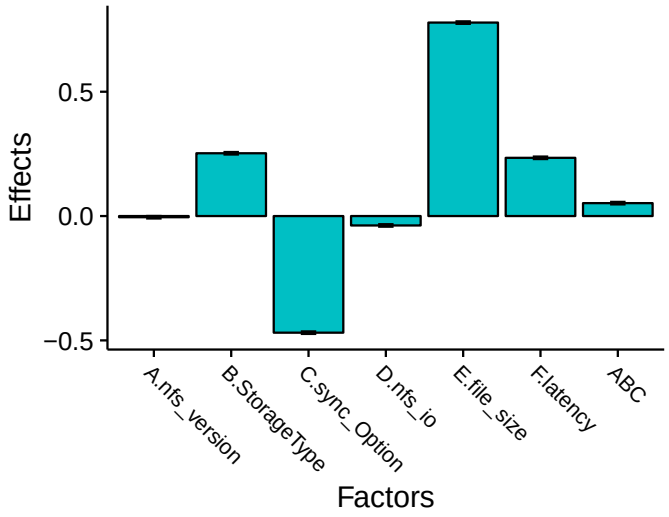


Figure 6.6: Effects of factors and interactions using fractional factorial design. The results are comparable with the results shown in Figure 6.5, but only obtained with a set of 12.5% of experiments

any operation into smaller ones, additional security features, client delegations to authenticate the clients to do some operations without contacting the server [25, 134], all together with other features in order to improve the overall performance.

Many studies have focused on the evaluation of the performance of NFS. Chen et al. [30] made an intensive comparison of NFSv3 and NFSv4.1. They also tested the newly implemented features of NFSv4.1 as well as tuning NFSv3 & NFSv4.1 to increase the throughput performance. They show that the performance of NFSv3 and NFSv4.1 depends on the network latency, i.e., NFSv4.1 was faster on a high latency network and slower than NFSv3 on a network with a lower latency. The same authors additionally focused on the performance of NFSv4.1 [29] by studying its behavior against several factors in Linux as the kernel version and the memory management. Radkov et al. [106] compared several versions of NFS (v2, v3 and v4) with iSCSI as representatives of file and block storage systems, and concluded that NFS is comparable with iSCSI in terms of performance. Furthermore, Martin et al. [87] tested NFS sensitivity against latency. They showed that NFSv3 can tolerate high latency more than NFSv2. Others have focused on performance evaluation of client-side tuning. Ou et al. [98] tested NFSv4 besides other storage file systems such as iSCSI and Swift in the context of cloud storage systems. Their important finding, which is related to our context, shows that the client-side configuration plays a significant role over the performance of IP-based storage systems like NFS. Lever et al. [76] uncovered several performance and scalability issues in the Linux NFS client that affected write latency and throughput.

The main difference between the experiments presented in this chapter and the ones presented in previous works is the focus on a high latency envi-

ronment, using a realistic distributed testbed instead of using an emulated network. Also, we use statistical methods (fractional factorial and full factorial designs). Those are rarely used in Computer Science, but we could find a few examples: Videau et al. [151] used a fractional factorial model to design their multi-parameter experiment in order to determine the impact of each parameter on the results. Khoshgoftaar and Rebours [68] used a full factorial model to evaluate several learning algorithms.

6.5 SUMMARY

The number of experimental factors at the beginning of an experimental study can be high, especially when the experimental context was not clear or when the correspondent research questions suggested them. This indirectly means running a colossal number of experimental combinations, even without a guarantee that the selected factors are crucial for the study. Hence, much time, effort, and resources could be wasted only for defining a coherent core of our experiments. We claim that statistical approaches can be leveraged here to enhance the designs of our experiments.

In this chapter, we explained the principles of fractional design and then demonstrated through experiments its ability to dramatically reduce the experimental combinations, compared with full factorial design. Both statistical designs — factorial and fractional — allow to draw similar conclusions at the end of study, but with the omission of analyzing factors' interactions in case of fractional design. Indeed, fractional designs encapsulate the effects of some selected factors with an equivalent number of interactions to gain more combinations. Although we recommended using that model in order to refine initial designs of big data experiments, it may be considered as a limitation in other domains, especially when all factors' effects should be available. Moreover, the fractional design is so sensitive to change, i.e., having experimental factors with an odd number of levels always incur a modification in the model equation, which is not always an attractive option.

The advantage of using fractional design is then demonstrated over an original set of real and complex experiments on moving data on experimental testbeds. Experiments with a wide range of parameters over several physically-distributed sites of the *Grid'5000* testbed were executed. We evaluated two related performance metrics — *average throughput and execution time* — which are meaningful for system engineers and clients, respectively. From one side, the results showed that, instead of doing full factorial experimentation, we could have done a fractional design which would have required only 12.5% of full factorial experiments. From the other side, the performance evaluation results showed that the network latency has a significant impact on the performance of the used data transfer protocol (*NFS*). With tuning some parameters (e.g., blocking/non-blocking I/O), the performance of a remote client can increase up to 20%, 83% of the performance of a local client, which makes the use of *NFS* over high-latency networks an acceptable option.

Part V

CONCLUSIONS

The strongest arguments prove nothing so long as the conclusions are not verified by experience. Experimental science is the queen of sciences and the goal of all speculation

— Roger Bacon

7 CONCLUSIONS AND PERSPECTIVES

Many research studies demonstrate the non-stop increase of using big data systems in various applicative domains to store and to analyze heterogeneous and scalable data. Given the considerable impact of such systems on the overall performance of on-top services and ecosystems, there is a tendency in academia and industry to understand their behaviors, in order to limit severe consequences when something goes wrong and to put their complexity under the microscope. Indeed, traditional experimental challenges are revived again with big data systems, in parallel with a convention inside the community that the legacy experimental methods are not relevant enough to cope with big data systems' particularities and requirements [92]. Hence, all phases of experiments' life-cycle should be reconsidered by revising current experimental methods and inducing new ones if needed. In this context, the goal of this thesis was to study how to improve the experimentation activity with big data systems. Among several experimental challenges that surround big data experiments, this work only focuses on addressing specific challenges related to the state of experimental context and experimental observability.

The general conclusion of this thesis affirms that the experimentation activity on big data systems can be improved by introducing specific experimental methods. This does not, by the way, mean that all actual methods, such as benchmarking and emulation, are obsolete. Indeed, they need to be supported by complementary methods that facilitate the experimentation activity from the viewpoint of researchers. The improvements are not limited to a specific phase of big data experiments' life-cycle; instead, our contributions are spread out over all experimental phases— *design, execution, and analysis*. We summarize below the main achievements of this work and present many perspectives of our contributions.

7.1 ACHIEVEMENTS

This work made four principal achievements in the context of improving and facilitating experimentation activity with big data systems.

Analyzing the literature of conducting big data experiments in Chapter 2 showed that the role of experimental design in the existing big data exper-

iments has not been paid enough attention. The design phase of analyzed experimental papers was not exploited to affine the set of initial experimental factors which are not necessarily coherent at an early stage of experimentation. Moreover, that phase did neither improve the usage of experimental resources knowing that most experiments are greedy at the beginning due to their unstable state. Hence, Chapter 6 has shown how statistics can be leveraged to improve the experimental design of big data experiments, reducing the number of required experimental runs and utilizing fewer experimental resources to reach same experimental conclusions as without using improved designs. The experimental study stated in Section 6.3 has demonstrated that only 12.5% of experimental runs was sufficient to reach its conclusions, intentionally ignoring to analyze tens of factors and interactions that are proved to be nonessential for the study. As a result, many experimental studies can get similar inspiration to go the same way for both, getting faster results and indirectly saving more resources.

We have argued that experimental environments should imitate end environments in a lot of aspects including resources performance to build reliable experimental conclusions. Unlike CPU and memory resources that can be customized using multiple technologies to meet experiments' requirements, storage experiments are forced in practice to use storage resources as provided without any customization (e.g., on networked testbeds). Hence, our storage performance emulation service presented in Chapter 3 allowed for creating heterogeneous storage performance on homogeneous storage infrastructure and vice versa. This did not only allow to perform experiments over specific scenarios of implementation but also led to ignoring the bias of storage devices as *HDDs* and *SSDs* can be altered under specific constraints. Indeed, experiments described in Section 3.4 have produced comparable results when performed on *HDDs* and *SSDs*.

We claimed in Chapter 4 that high-level evaluation techniques are not adequate for understanding micro behaviors of big data systems. Thus, our *IOscope* toolkit have been developed to perform in-depth observations for storage operations, aiming at understanding performance and revealing potential issues. Section 4.3 have brought experimental evidence to confirm this insight as *IOscope* was able to explain a hidden performance issue in one of *NoSQL* storage leaders. Moreover, the revealed I/O pattern-related issue was also raised on *SSD* storage, which was unexpected. As a reaction, the experiments presented in Section 4.4 enabled us to affirm that some SSD storage could be influenced by workload types (sequential or random) in case of having workloads with HDD-like characteristics (e.g., one I/O process and small I/O requests). The bias of storage resources is not negligible, so systems that act as big data storage should be internally tested over various storage technologies as a precaution.

Finally, avoiding manual steps during collecting experiments' data and results did not only facilitate conducting experiments but also led to promote experiments' repeatability and reproducibility. We have shown that the experimental observability could be enhanced through *EMFs* whose re-

quirements have been defined in Chapter 5. Our framework, *MonEx*, which is the first *EMF* implementation, demonstrated through different experiments the advantages of methodological data collection (see Section 5.5). Moreover, it illustrated how the experimenter could rely on such a framework even to analyze and prepare figures that respect publication requirements.

According to the achievements mentioned above, we believe that experiments became easier to be handled either on experimental or production environments thanks to the statistical-based design, the increased controllability over experimental resources, and the observability tools (*IOscope* & *MonEx*) that complement each other by performing in-depth observations and facilitating data retrieval and analysis. These achievements could be seen in experimenter's eyes as bricks of LEGO that have been built to complement each other, in order to facilitate experimental tasks through all experiments' stages from *design* to *analysis*.

7.2 FUTURE DIRECTIONS

The findings of this work can be seen as seeds for many improvements and research directions under the umbrella of improving experimental context and observability for big data experiments. Four research directions are identified here to extend the research presented in this thesis as well as to overcome its limitations.

7.2.1 *Expanding Storage Configurations for Experiments*

Providing customized storage configurations to big data experiments is a big goal which behind the scenes implies working on various storage layers, including storage devices. The work presented in Chapter 3 could be seen as a partial achievement of that goal, enabled to emulate storage performance to provide heterogeneous and complex setups for experiments. However, our emulation service only focused on storage performance. Indeed, it did not deal with storage devices' internals, which in parallel are confirmed in Chapter 4 to have substantial impacts on experiments' results (see Section 4.4). To this end, our storage emulation service should be extended to stretch beyond controlling I/O throughput.

As future work, storage devices should be controlled by big data experiments as a logical addition to the storage performance emulation described in Chapter 3. This will allow for emulating devices with degraded performance or with a specific set of features (e.g., 50% of corrupted allocation space, disabled cache, disabled internal concurrency), enlarging without a doubt the testing configurations for big data experiments. Indeed, further research should identify the flexible characteristics of storage devices that should be manipulated by emulation services. We are convinced that studying systems over such diverse parameters can enable the industry to develop systems that are aware of negative impacts on performance caused in low layers of execution. In the same time, we believe that realizing such emulations is

doable in the era of open-channel storage devices [16, 17] that are totally or partially exposed to the operating system.

As secondary future work, the emulation service described in Chapter 3 depends on Linux's *Cgroups* technology. Hence, its lack of supporting other operating systems can be considered as a limitation even if Linux distributions are heavily used on experimental environments. Provided that, further research should be done to figure out an alternative technology of *cgroups* that will enable to emulate storage performance on other non-Linux operating systems.

7.2.2 *Observability Tools for Uncovering Wide Range of I/O Issues*

IOscope toolkit, which is presented in Chapter 5 was able to reveal potential I/O issues of big data systems. It was adequate for highlighting through experiments (see Section 4.3) the fact that some big data systems may report moderate performance due to unexpected behavior when accessing stored data. That work did not only open a discussion to assess the advantages of using in-depth observability tools beside high-level tools but also did demonstrate that such objective-specific tracing tools are useful for production usage. However, the scope of our toolkit is still restricted as it was only able to uncover I/O issues related to I/O patterns.

We have identified two directions to improve. First, *IOscope* tools can be technically improved to take self-decisions, reducing the number of target data files to those who are mostly visited in case of intensive I/O workloads over multiple data files. Doing so can reduce the time of post-analysis and minimize further the overhead of these tools. Secondary and most importantly from a research viewpoint, the diversity of I/O issues should be investigated, and complementary tools to *IOscope* should be built. For instance, besides of investigating workloads' I/O patterns as done by *IOscope*, issues related to a mismatch between scheduling layers on the I/O stack and storage disks, or caused by the way the data is allocated on storage devices should be studied and covered in future.

7.2.3 *EMFs for Experiments Running on Federated Environments*

Many projects such as *FED4FIRE*[1] have taken in charge the responsibility to group several experimental environments where experiments can utilize multiple resources over federated environments either to benefit from environments with specific kinds of resources (e.g., IoT, WSN) or simply to scale. The heterogeneity of resources and interfaces of federated environments may represent obstacles if we manage to use the *MonEx* framework (see Chapter 5) to monitor such experiments or to provide services on top of it. Many works have been proposed to improve the state of monitoring over federated environments, introducing monitoring ontology [4], and architectures of measurements [3, 5]. However, these propositions can be seen as an initial

1 Fed4Fire website: https://www.fed4fire.eu/

step to provide a fundamental basis for infrastructure monitoring, which is different from experiment-based monitoring in its nature, usage, and restrictions.

The promising direction to extend the research done over the *MonEx* framework can be achieved by restudying the list of requirements of *EMF* presented in Chapter 5, to discuss and to include the challenges of federated experiments. This will lead either to extend the *MonEx* framework, which was the first step to unifying experiments' data collection methods under one framework. Moreover, as none of the experiments presented in Section 5.5 is performed on federated infrastructure, the need to highlight the maturity of the extended *MonEx* framework over complex experiments on federated environments is necessary to show that it is useful for various experimentation contexts and constraints. In contrast, the speculation around the new set of requirements may also lead to building other *EMF*s from scratch.

7.2.4 *Machine Learning-Based Experimental Methods*

The convergence between big data and machine learning is in the growth phases, and it will grow as much as we need intelligent computers to deal with our data [6, 83]. However, the usage of machine learning in big data context is so far limited to analyze and to extract values from data. Indeed, using machine learning approaches to enhance experimental methods around big data is not common as it is the case in other domains [41].

The work stated in Chapter 6 demonstrated that statistics should be reconsidered in the design of big data experiments to minimize resources utilization. That was done by reducing the number of experimental runs, which is in the interests of both experimenters and resources operators. Going further to enhance the phase of experimental factors selection and the amount of saved energy in experimental environments can be beyond the capacity of statistical models. A promising way to doing so is through introducing machine-learning based experimental methods to overcome such challenges. For instance, machine learning approaches can help to decide experimental factors and their suitable values with the aid of archived studies performed on the same infrastructure. They can also direct experimenters to select resources on distributed environments that fit experimental workflow and offer best scenarios regarding energy consumption.

Part VI

APPENDIX

RÉSUMÉ DE LA THÈSE EN FRANÇAIS

A.1 INTRODUCTION GÉNÉRALE

Les domaines d'informatique évoluent rapidement. Il y a plusieurs décennies, les systèmes informatiques étaient conçus pour fonctionner sur un seul ordinateur, faisant leurs tâches basiques de manière isolée. Les évolutions dans les domaines logiciels et matériels ouvrent la voie à des architectures complexes pour les systèmes informatiques. Parallèlement au fait que la courbe de production de processeurs haute vitesse ne peut pas être poussée encore plus loin, les progrès des algorithmes distribués et de réseaux conduisent à envisager d'aller plus vite et plus loin avec la distribution [74, 140]. Aujourd'hui, les systèmes distribués coordonnent un nombre considérable d'ordinateurs indépendants qui effectuent leurs tâches et partagent leurs ressources de manière transparente pour les utilisateurs finaux. Comme on peut le voir, de multiples secteurs de l'économie au divertissement sont inspirés pour construire une infrastructure à l'échelle, fournir des services mondiaux et viser les gens dans le monde entier. Sans aucun doute, les gens entrent rapidement dans le monde numérique en produisant et en consommant une énorme quantité de données. Selon la *société internationale de données - IDC*, les individus sont responsables de la génération d'environ 75% des informations dans le monde numérique [46].

Récemment, le big data est devenu l'un des sujets les plus brûlants de la recherche en informatique. De nombreuses institutions, dont des gouvernements, des entreprises et des médias, manifestent un grand intérêt pour big data et leur perspectives [14, 28, 80]. Bien que le terme «big data» ne donne pas de limites précises sur ce que nous traitons, Wikipédia le définit comme «*des ensembles de données devenus si volumineux qu'ils dépassent l'intuition et les capacités humaines d'analyse et même celles des outils informatiques classiques de gestion de base de données ou de l'information*». Admettant que le mot "volumineux" n'ajoute rien au terme "big data" en termes de précision, cette définition critique les capacités des systèmes actuels utilisés autrefois pour traiter le big data. Bien sûr, les défis du big data sont importants. Ces défis ne sont pas seulement liés au volume de données qui augmente chaque jour mais également à plusieurs aspects connus sous le nom de *big data V's*. Outre le *volume*, les défis touchent également à la *vitesse* de génération et de traitement des données, la *variété* de données, par exemple, textuelles, images, vidéos, journaux et appels téléphoniques rassemblés à partir de multiples sources et la *valeur* à extraire de ces données [60, 66, 67, 73]. Cela explique un peu pourquoi les systèmes traditionnels sont limités pour relever ce type de défis.

Ayant de nouveaux systèmes pour surmonter un ensemble partiel ou tous les défis liés au stockage, à l'analyse et au traitement big data est crucial pour maintenir le contrôle sur ces données émergentes. Nous parlons de systèmes big data.

Le développement de systèmes big data est difficile. Ces systèmes devraient surmonter deux catégories générales de défis. Premièrement, ils font face à des défis associés aux systèmes existants tels que les systèmes d'E/S et les systèmes analytiques car ils font partie du flux de travail des systèmes big data. Deuxièmement, ils sont principalement concernés de répondre aux nouvelles exigences liées aux caractéristiques big data [26, 27]. D'une part, un grand nombre de systèmes big data sont souvent construits sur des idées récentes qui n'avaient pas été exploitées auparavant, en montrant un peu d'utilisation des expériences de recherche construites au fil du temps. Par exemple, les systèmes de stockage de données volumineuses relèvent les défis liés à la conservation de données hétérogènes, en créant de nouveaux modèles de données qui sont largement différents du modèle relationnel utilisé dans les bases de données *SQL*. D'autre part, ils doivent gérer de nouvelles exigences telles que le maintien de leurs services toujours disponibles et le contrôle de la cohérence des données. Cela se produit cependant au prix d'architectures et de conceptions sophistiquées qui peuvent être sujettes aux erreurs et masquer de nombreux types de problèmes (par exemple, des problèmes fonctionnels et de performances). Dans l'ensemble, les points abordés ici montrent la nécessité de mener plus d'investigations et de recherches pour créer des systèmes robustes big data.

Évaluer les systèmes big data est important. En théorie, l'évaluation peut être effectuée à l'aide de méthodes formelles. Ce dernier représente les spécifications des systèmes sous forme de modèles mathématiques à valider via des techniques telles que la vérification des modèles. Ces techniques sont cependant utilisées dans des domaines tels que la spécification formelle [94] et la vérification [119], mais pas pour les systèmes complexes. Dans la pratique, la complexité de conception des systèmes big data peut représenter un obstacle pour l'utilisation des méthodes formelles. En effet, capturer tous les comportements attendus des systèmes big data par un modèle abstrait n'est pas réalisable. De plus, les environnements qui jouent un rôle important dans le cycle de vie des systèmes big data devraient également être pris en compte pendant l'évaluation. Cependant, ce n'est pas le cas avec les méthodes formelles.

La validation expérimentale est l'alternative logique à la validation formelle. Elle est principalement utilisée avec les systèmes informatiques [125, 146, 158]. L'une des raisons, entre autres, de faire une validation expérimentale est l'impossibilité de produire des modèles complets pour faire de validation formelle. Plus la complexité du *système sous test – SUT* est grande, plus l'orientation vers des méthodes de validation basées sur l'observation. En effet, les expériences sont directement confrontées à de réelles complexités qui sont soit liées au SUT, soit à l'environnement expérimental, soit aux deux. Par conséquent, les conclusions sont tirées de cas d'utilisation réels. Entre

temps, il n'y a pas de normes pour identifier de bonnes expériences ou de meilleurs environnements expérimentaux. De plus, il n'est pas étrange d'avoir des méthodologies et des environnements différents pour expérimenter sur un domaine donné de systèmes informatiques.

Pour définir la portée de cette thèse, la recherche expérimentale sur les systèmes big data est le sujet principal à traiter ici. Nous accordons une attention particulière à l'élément central de cette recherche: les expériences. Nous soutenons que le cycle de vie des expériences big data est similaire au cycle de vie hérité des expériences informatiques et au cycle de vie des expériences à grande échelle mentionné dans [89], en ayant au moins trois phases principales – *conception* [1], *exécution* et *analyse*. En revanche, nous affirmons que le les particularités des systèmes big data (*voir la section a.1.1*) entraînent de nouvelles difficultés qui affectent le flux expérimental tout au long du cycle de vie des expériences. Continuer à effectuer des évaluations sans les considérer peut conduire à des résultats insatisfaisants. Par exemple, pratiquer une observabilité en profondeur est essentiel pour comprendre le performance des systèmes big data. Cependant, les pratiques actuelles dans les deux — *l'académie et l'industrie* — pointent vers des outils génériques [51] comme *Yahoo! Cloud Serving Benchmark (YCSB)* [33], YCSB ++ [102], et *BigDataBench* [154] pour évaluer les systèmes de stockage big data. Bien sûr, ces outils sont conçus pour produire des mesures plutôt que pour fournir une compréhension de performance; ils font donc un compromis entre la flexibilité de l'expérimentation et l'obtention de résultats détaillés. Ils effectuent des évaluations sur des couches supérieures pour couvrir de nombreux systèmes car ils pourraient être visés à ce niveau, mais au prix d'ignorer les particularités des systèmes et leurs architectures internes.

Faire face aux défis du big data lors de l'expérimentation implique de traiter en premier lieu les méthodes expérimentales. L'état de l'art des méthodes expérimentales en informatique sont diverses. La sélection d'une méthode appropriée pour les expériences big data dépend de plusieurs aspects, notamment l'environnement et les caractéristiques du système cible. Par exemple, nous simulons souvent les activités d'un système donné si l'expérimentation sur un système réel entraîne des coûts élevés ou est tout simplement impossible. En fonction du type de systèmes cibles (réels ou modèles) et de la nature des environnements (réels ou modèles), les méthodes expérimentales peuvent être classées en quatre catégories [50, 62]. *expérimentation in-situ* (système réel sur un environnement réel), *benchmarking* (modèle d'un système sur un environnement réel), *émulation* (système réel sur un modèle d'environnement) et textslsimulation (modèle d'un système sur un modèle d'environnement). Ces approches sont utilisables pour l'évaluation des systèmes big data. Cependant, la dérivation de nouvelles méthodes qui prennent en compte les défis précis des systèmes big data est fortement requise. En effet, c'est la direction principale de cette thèse.

1 Appelée *composition* dans [89] pour montrer la valeur de la réutilisabilité de conception

A.1.1 *Caractéristiques des systèmes big data*

Le concept de *big data* est volatile. Les chercheurs ont encore des ambiguïtés concernant sa définition précise, ses domaines, ses technologies et ses fonctionnalités citeward2013undefined. De nombreux travaux de recherche tels que [36, 37, 44] apparaissent au fil du temps pour définir le *big data* concernant ses caractéristiques trouvées dans la littérature. Ces travaux à côté d'autres encouragent à décrire le *big data* par ses fonctionnalités, communément appelées *Vs*. La liste de *Vs* qui a été limitée à *Volume* (taille des données), *Vélocité* (vitesse de génération des données) et *Variété* (types de données) est étendue au fil du temps pour inclure *valeur* (les informations extraites) et *véracité* (qualité des données). Cependant, la signification de *Vs* est variable et peut être interprétée différemment en fonction du contexte sous-jacent. Par exemple, *Velocity* signifie par défaut le *rythme de génération de données*, mais il fait référence à *vitesse de mise à jour* pour les systèmes de médias sociaux, et à *vitesse de traitement* pour les systèmes de streaming [52]. Cette controverse de haut niveau pourrait se refléter dans la mise en œuvre de systèmes big data.

Les systèmes big data sont définit comme tous les systèmes informatiques qui collectent, stockent ou analysent des données contenant un ensemble partiel ou complet de *big data Vs*. Ces systèmes peuvent d'un coté utiliser les techniques et technologies existantes pour fonctionner, comme l'utilisation des systèmes distribués pour permettre le passage à l'échelle horizontale. D'un autre côté, les systèmes big data peuvent également avoir leurs propres technologies. Les systèmes de stockage en sont un exemple. Au lieu d'utiliser des systèmes de stockage traditionnels tels que les bases de données SQL, qui ne permettent pas de stocker des données non-structurées ni de les mettre à l'échelle, les systèmes de stockage big data s'appuient sur des technologies de stockage adaptées aux fonctionnalités big data. Ils peuvent être classés en quatre catégories [141], qui sont 1) *Systèmes de fichiers distribués* tels que *Hadoop Distributed File System* (*HDFS*) qui fournit une base de stockage pour l'écosystème *Hadoop*, 2) *Not Only SQL* (*NoSQL*) *bases de données* qui introduisent de nouveaux modèles de stockage tels que les modèles basés sur des documents et des colonnes, 3) *bases de données NewSQL* qui tentent d'améliorer les bases de données traditionnelles à l'échelle tout en conservant leur modèle de données relationnelles, et 4) *Plateformes d'interrogation* qui fournissent des requêtes de stockage front-end pour les systèmes de fichiers distribués et les bases de données NoSQL. Ces systèmes ont également des caractéristiques qui pourraient affecter la façon dont nous les évaluons. D'une part, ils ont des architectures complexes bien qu'ils appartiennent à la même famille (par exemple, les systèmes de stockage). Par conséquent, les interfaces des systèmes peuvent être implémentées différemment, de sorte que les possibilités de construire des outils expérimentaux qui couvrent plusieurs catégories de systèmes sont réduites. D'une autre part, on pourrait penser que la réalisation d'évaluations sur des interfaces de systèmes big data est suffisante. Cependant, ces systèmes ont beaucoup de comportements complexes, tels que des interactions dans les couches inférieures de *OSs*, qui devraient également

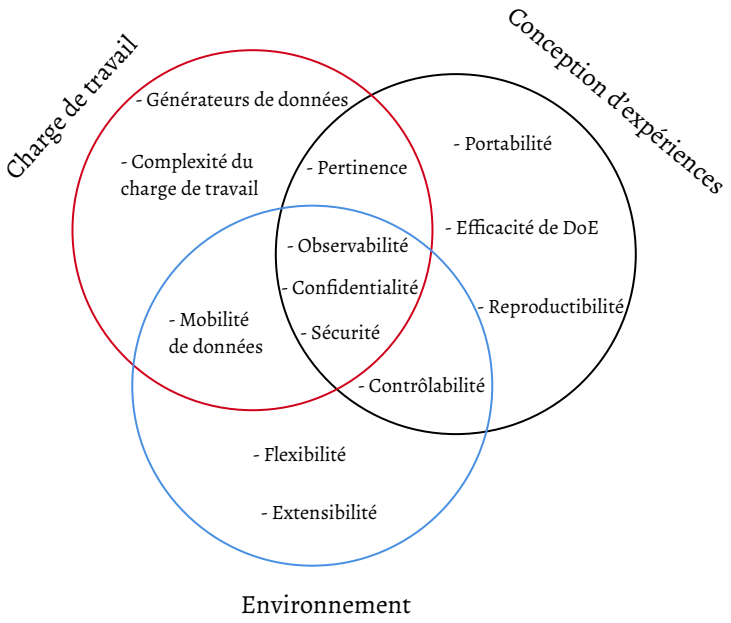


Figure a.1: Une liste des principaux défis des expériences big data

être ciblées par des expériences. Dans l'ensemble, bien que les systèmes big data soient distincts et correspondent à divers sous-ensembles de *big data V's*, effectuer des expériences sur ces systèmes pose des défis communs. Nous les décrivons dans la section suivante.

A.1.2 *Défis d'Expérimentation sur les Systèmes big data*

Comme souligné ci-dessus, les défis auxquels sont confrontées les expériences sur des systèmes big data sont énormes. La majorité d'entre elles sont liées aux principales questions d'expérimentation qui sont *comment, où et sur quelles données les expériences peuvent être effectuées ?* Bien que de nombreuses décennies et recherches soient consacrées à répondre à ces questions concernant d'autres disciplines, les mêmes questions sont à revoir sous l'angle big data. Par conséquent, nous classons les défis communs des expériences big data en trois catégories qui se croisent. Ces catégories sont 1) *charge de travail/données* car la sélection de la charge de travail et les caractéristiques des données sont l'un des principaux facteurs d'expérimentation, 2) *conception* pour englober tous les défis liés aux expériences elles-mêmes, et 3) *environnement* qui joue un rôle important dans le déroulement des expériences. La figure a.1 montre une liste des principaux défis de l'expérimentation sur des systèmes big Data. Ces défis sont brièvement décrits ci-dessous.

- Contrôlabilité. De nombreux environnements expérimentaux, y compris les clouds et les testbeds, offrent des ressources qui ne correspondraient pas aux exigences des expériences des chercheurs par défaut. Par exemple, une expérience à effectuer sur une infrastructure hétérogène en termes de performance de mémoire tandis que

l'infrastructure expérimentale disponible est homogène. En admettant que toutes les expériences ne sont pas similaires, beaucoup d'entre eux nécessitent des configurations spécifiques qui, dans certains cas, ne sont pas réalisables sur les ressources expérimentales offertes. Donner aux expériences davantage de contrôle sur l'environnement est difficile car cela implique de manipuler des ressources à l'échelle et personnaliser leur performance.

- OBSERVABILITÉ. L'observabilité des expériences big data ne se limite pas aux résultats et aux performances des ressources, mais peut également être étendue pour observer des comportements de bas niveau des systèmes big data. L'utilisation de méthodes d'observabilité traditionnelles (par exemple, les systèmes de monitoring traditionnels) qui ne sont pas principalement créés pour les systèmes de dig data peut avoir plusieurs obstacles techniques et de conception (par exemple, limitation du passage à l'échelle et l'incompatibilité avec des systèmes big data spécifiques). Par conséquent, les techniques d'observabilité devraient être améliorées, en facilitant les méthodes de collecte des données d'expérience et en proposant des méthodes adaptées aux scénarios expérimentaux du big data.

- COMPLEXITÉ DU CHARGE DE TRAVAIL. Les charges de travail pour les expériences big data ont deux sources [7]. Ils sont soit générés à l'aide de générateurs de données de benchmarks connus (par exemple, la famille TPC et les benchmarks YCSB) tels que le *Parallel Data Generation Framework* [105], soit à l'aide de générateurs personnalisés adaptés à des expériences spécifiques. Pour les deux types, les charges de travail sont simples et ne reflètent pas les caractéristiques complexes des charges de travail réelles. Par conséquent, la simplicité des charges de travail peut être considérée comme un biais qui falsifie les résultats des expériences en prenant en compte les cas d'utilisation réels.

- EFFICACITÉ DE *DoE*. Les expériences big data ont tendance à avoir des conceptions approximatives à leurs débuts, c'est-à-dire collecter des facteurs expérimentaux par intuition. Ces conceptions approximatives affectent non seulement les résultats expérimentaux, mais augmentent également la consommation de ressources expérimentales. Proposer des *Design of Experiments* (*DoE*) efficaces peut raccourcir le temps expérimental, réduire les scénarios expérimentaux en se concentrant uniquement sur des facteurs expérimentaux significatifs qui minimisent indirectement l'utilisation des ressources.

- MOBILITÉ DES DONNÉES. Les expériences big data nécessitent souvent de s'exécuter sur plusieurs environnements indépendants (par exemple, fédération de testbeds et IoT) qui n'ont pas nécessairement la même infrastructure. Il est difficile de maintenir un déplacement efficace des données entre les nœuds expérimentaux malgré la diversité des technologies, des ressources et des protocoles de transfert, en particulier en

cas de données volumineuses. Même le déplacement de données pour alimenter les nœuds expérimentaux ou enfin pour l'analyse, assurer la cohérence et l'intégrité des données devrait passer par l'élimination de plusieurs sources de pannes d'E/S et la coordination de nombreux protocoles locaux et distribués.

- GÉNÉRATEURS DE DONNÉES. La génération d'ensembles de données réalistes pour les expériences big data est essentielle pour améliorer l'activité d'évaluation et la comparaison des systèmes big data. Cette phase présente encore de nombreux défis, tels que la génération de données à partir d'un modèle de référence complexe et la création de générateurs de données hybrides pour les ensembles de données structurels et non structurels.

- EXTENSIBILITÉ D'ENVIRONNEMENT. Les environnements expérimentaux sont souvent statiques quant à la composition de leurs ressources. Les pousser à adopter de nouvelles disciplines scientifiques associées aux systèmes big data nécessite souvent d'ajouter de nouvelles ressources, de modifier leur liaisons et de créer de nouvelles couches de communication. Cela entraîne de nombreux défis immédiats car les environnements sont toujours résistants au changement.

- PERTINENCE DE LA CHARGE DE TRAVAIL AUX EXPÉRIENCES. Chaque système big data est unique dans sa conception et ses exigences. Par conséquent, l'expérimentation sur des systèmes divers en utilisant une charge de travail similaire ne garantit pas de mettre en évidence des comportements similaires sur ces systèmes.

- FLEXIBILITÉ DE L'ENVIRONNEMENT. Les expériences big data ont des besoins différents de ressources et de services d'accompagnement (par exemple, stockage intermédiaire). Les environnements doivent être flexibles concernant les changements. Ils doivent fournir des ressources à la demande pour les expériences, soutenir la fédération pour les expériences avec une utilisation intense et fournir des services secondaires tels que le cadre d'analyse et les installations de stockage.

- REPRODUCTIBILITÉ. La reproduction des expériences est une exigence majore pour valider les résultats scientifiques. Cependant, c'est l'un des plus grands défis non seulement pour les expériences big data mais également pour les expériences dans toutes les disciplines scientifiques.

- PORTABILITÉ DES EXPÉRIENCES. Il n'est pas possible d'effectuer une expérience sur de nombreux environnements sans tenir compte de la portabilité dans sa conception. Il est difficile de faire des expériences 100% portables. Cela nécessite d'éliminer l'impact des ressources expérimentales de la conception.

- CONFIDENTIALITÉ. Bien que certains environnements expérimentaux tels que les testbeds soient partagés entre plusieurs utilisateurs au fil

du temps, assurer la confidentialité des journaux et des données des expériences peut représenter un véritable défi.

- SÉCURITÉ. De nombreux défis de sécurité sont présents, tels que la garantie de l'intégrité et du chiffrement des données.

A.1.3 *Problématique*

Traiter tous les défis décrits dans la section précédente a.1.2 ne pourrait pas être fait dans le cadre d'une seule thèse. Par conséquent, cette thèse considère un sous-ensemble de ces défis, à savoir la *contrôlabilité* et l'*observabilité*. Sa principale préoccupation est de faciliter l'activité d'expérimentation tout au long du traitement de ce sous-ensemble de défis dans les différentes phases du cycle de vie des expériences. En particulier, cette thèse apporte des réponses aux questions listées ci-dessous.

- *Comment éliminer le problème d'incompatibilité entre les performances des environnements expérimentaux et les exigences sophistiquées des expériences big data ? En particulier, comment satisfaire les besoins d'expériences à l'échelle qui nécessitent des performances de stockage hétérogènes sur des clusters de ressources homogènes ?*

  Cette question envisage de modifier les environnements pour répondre aux besoins des expériences plutôt que d'appliquer des changements à la conception des expériences ou d'envisager d'autres environnements expérimentaux.

- *Comment améliorer l'observabilité des expériences qui ont des mesures à grain fin (par exemple des millions d'opérations dans une petite unité de temps) afin d'avoir une meilleure compréhension de performance?*

- *Comment améliorer l'observabilité des expériences sur des environnements tels que les testbeds, en surmontant des défis tels que i) déterminer le contexte des expériences quelle que soit l'infrastructure sous-jacente et ii) avoir des approches méthodologiques pour collecter leurs données?*

  Cette question de recherche émerge des questions dérivées suivantes:

  - *Quelles sont les exigences pour construire EMFs?*
  - *Comment concevoir et implémenter des EMFs afin de satisfaire les exigences prédéfinies?*

- *Comment améliorer la conception des expériences big data pour réduire leurs combinaisons expérimentales lourdes qui consomment plus de ressources que d'habitude?*

  De nombreuses considérations doivent être prises en compte lors de la conception de telles expériences. À un stade précoce, les expériences ont un grand nombre de facteurs qui peuvent affecter ou non les résultats des expériences. En outre, des contraintes sur l'utilisation des ressources peuvent être données.

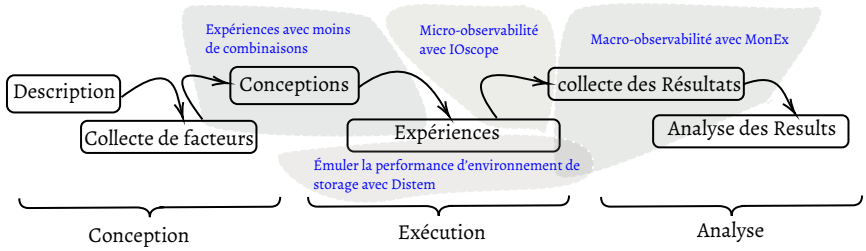


Figure a.2: Présentation des contributions de cette thèse sur le cycle de vie des expériences big data

A.1.4 *Contributions*

La thèse apporte quatre contributions majeures pour répondre aux questions de recherche décrites dans la section précédente. La figure a.2 montre les contributions ensemble sur le cycle de vie des expériences big data. Pour décrire les contributions concernant leur placement sur la figure, les deux premières contributions visent à améliorer la conception expérimentale et le contexte en travaillant sur deux points distincts mais relatifs – *conception* & *exécution*. Tout d'abord, étant donné l'état brut de conception [2] des expériences, nous utilisons des statistiques pour obtenir le plus rapidement possible un plan qui permet de réduire l'ensemble de combinaisons expérimentales qui conduisent indirectement à consommer moins de ressources. Nous poussons ensuite vers de meilleures exécutions en donnant aux expériences une plus grande contrôlabilité sur les ressources pour établir des environnements faciles à configurer à l'échelle. Ces deux contributions conduisent à susciter des questions sur l'observation et l'analyse des résultats des expériences, en particulier pour les expériences ayant des comportements de bas niveau et une infrastructure à l'échelle. Par conséquent, les deux dernières contributions sont consacrées à l'amélioration de l'observabilité des expériences quels que soient leurs conceptions et environnements expérimentaux. Toutes les contributions de thèse sont décrites ci-dessous.

INTRODUIRE L'HÉTÉROGÉNÉITÉ À LA PERFORMANCE DES RESSOURCES DE STOCKAGE

Nous introduisons un service pour personnaliser les environnements expérimentaux de stockage. Cela permet de créer des configurations réalistes d'expérimentation grâce à l'émulation. Le service permet d'avoir des performances de stockage hétérogènes et contrôlables pour des expériences complexes et à l'échelle, en reflétant les caractéristiques des environnements finaux pour renforcer les résultats expérimentaux. Notre service est implémenté dans le plate-forme expérimentale *DISTributed systems EMulator* (*Distem*), qui est une solution open source connue dans la communauté des testbeds tels

2 Une phase initiale de la plupart des expériences big data où la majorité des facteurs expérimentaux sont collectés par intuition à étudier

que textesl Grid'5000, textesl Chameleon et textesl CloudLab, afin d'atteindre plus de personnes dans des communautés diverses.

IOSCOPE POUR RENFORCER LE MICRO-OBSERVABILITÉ

Nous proposons une méthode de profilage basée sur le filtrage pour tracer les opérations d'E/S de bas niveau. Nous encourageant à observer de petits comportements lors de l'expérimentation du big data. Cette méthode est implémentée dans une boîte à outils, nommée *IOscope*, pour couvrir toutes les modèles d'E/S dominantes. L'utilisation d'*IOscope* avec des outils d'évaluation de haut niveau est efficace pour étudier les charges de travail avec des questions de performance ainsi que pour comprendre les principales raisons des problèmes potentiels de performance.

MACRO-OBSERVABILITÉ AVEC MONEX

Nous proposons un outil expérimental, nommée *Monitoring Experiments Framework* (*MonEx*), pour faciliter l'observation des expériences du point de vue des expérimentateurs. MonEx élimine toutes les étapes ad hoc courantes pendant la phase de collecte des données des expériences et est partiellement en charge de la préparation des résultats, tels que la production de figures simples et prêtes à être publiées. Il est construit au-dessus des composants standard pour assurer l'exécution sur plusieurs environnements expérimentaux.

RÉDUCTION DES COMBINAISONS EXPÉRIMENTALES

Nous confirmons l'applicabilité des statistiques pour réduire des combinaisons lourdes d'expériences. En particulier, les expériences qui n'ont pas une portée claire au début de la conception. Cela se fait en utilisant un modèle statistique qui élimine les facteurs expérimentaux non pertinents d'une expérience ainsi que les interactions des facteurs qui n'influencent pas les résultats expérimentaux. Cela ne pousse pas seulement à avoir une conception expérimentale stable le plus rapidement possible, mais aussi à économiser beaucoup de ressources expérimentales et d'efforts par rapport à la méthode classique qui exécute des expériences avec une combinaison complète de facteurs expérimentaux.

BIBLIOGRAPHY

[1] Veronika Abramova and Jorge Bernardino. "NoSQL databases: MongoDB vs cassandra." In: *Proceedings of the international C* conference on computer science and software engineering*. ACM. 2013, pp. 14–22.

[2] Sungyong Ahn, Kwanghyun La, and Jihong Kim. "Improving I/O Resource Sharing of Linux Cgroup for NVMe SSDs on Multi-core Systems." In: *HotStorage*. 2016.

[3] Yahya Al-Hazmi and Thomas Magedanz. "Monitoring and measurement architecture for federated Future Internet experimentation facilities." In: *2014 European Conference on Networks and Communications (EuCNC)*. IEEE. 2014, pp. 1–6.

[4] Yahya Al-Hazmi and Thomas Magedanz. "MOFI: Monitoring ontology for federated infrastructures." In: *Measurements & Networking (M&N), 2015 IEEE International Workshop on*. IEEE. 2015, pp. 1–6.

[5] Yahya Al-Hazmi and Thomas Magedanz. "Towards semantic monitoring data collection and representation in federated infrastructures." In: *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE. 2015, pp. 17–24.

[6] Omar Y Al-Jarrah, Paul D Yoo, Sami Muhaidat, George K Karagiannidis, and Kamal Taha. "Efficient machine learning for big data: A review." In: *Big Data Research* 2.3 (2015), pp. 87–93.

[7] Alexander Alexandrov, Christoph Brücke, and Volker Markl. "Issues in big data testing and benchmarking." In: *Proceedings of the Sixth International Workshop on Testing Database Systems*. ACM. 2013, p. 1.

[8] Jiju Antony. *Design of experiments for engineers and scientists*. Elsevier, 2014.

[9] Daniel Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed." In: *Cloud Computing and Services Science*. Vol. 367. Communications in Computer and Information Science. 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1.

[10] Chaitanya Baru and Tilmann Rabl. "Application-Level Benchmarking of Big Data Systems." In: *Big Data Analytics*. Springer, 2016, pp. 189–199.

[11] Chaitanya Baru, Milind Bhandarkar, Raghunath Nambiar, Meikel Poess, and Tilmann Rabl. "Setting the direction for big data benchmark standards." In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2012, pp. 197–208.

[12] Chaitanya Baru, Milind Bhandarkar, Raghunath Nambiar, Meikel Poess, and Tilmann Rabl. "Benchmarking big data systems and the bigdata top100 list." In: *Big Data* 1.1 (2013), pp. 60–64.

[13] Elisa Bertino, Philip Bernstein, Divyakant Agrawal, Susan Davidson, Umeshwas Dayal, Michael Franklin, Johannes Gehrke, Laura Haas, Alon Halevy, Jiawei Han, et al. "Challenges and Opportunities with Big Data." In: (2011).

[14] John Carlo Bertot and Heeyoon Choi. "Big Data and e-Government: Issues, Policies, and Recommendations." In: *Proceedings of the 14th Annual International Conference on Digital Government Research*. dg.o '13. Quebec, Canada: ACM, 2013, pp. 1–10. ISBN: 978-1-4503-2057-3. DOI: 10.1145/2479724.2479730. URL: http://doi.acm.org/10.1145/2479724.2479730.

[15] Eugen Betke and Julian Kunkel. "Real-Time I/O-Monitoring of HPC Applications with SIOX, Elasticsearch, Grafana and FUSE." In: *High Performance Computing*. Ed. by Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf. Cham: Springer International Publishing, 2017, pp. 174–186. ISBN: 978-3-319-67630-2.

[16] Matias Bjørling, Javier González, and Philippe Bonnet. "LightNVM: The Linux Open-Channel SSD Subsystem." In: *FAST*. 2017, pp. 359–374.

[17] Matias Bjørling et al. "Open-Channel Solid State Drives." In: *Vault, Mar* 12 (2015), p. 22.

[18] George EP Box and J Stuart Hunter. "The 2 k—p fractional factorial designs." In: *Technometrics* 3.3 (1961), pp. 311–351.

[19] Morgan Brattstrom and Patricia Morreale. "Scalable Agentless Cloud Network Monitoring." In: *Cyber Security and Cloud Computing (CSCloud)*. 2017.

[20] Peter Bühlmann and Sara van de Geer. "Statistics for big data: A perspective." In: *Statistics & Probability Letters* 136 (2018), pp. 37–41.

[21] Michael J Carey, David J DeWitt, Michael J Franklin, Nancy E Hall, Mark L McAuliffe, Jeffrey F Naughton, Daniel T Schuh, Marvin H Solomon, CK Tan, Odysseas G Tsatalos, et al. *Shoring up persistent applications*. Vol. 23. 2. design; discussion; shore. ACM, 1994.

[22] Rick Cattell. "Scalable SQL and NoSQL data stores." In: *Acm Sigmod Record* 39.4 (2011), pp. 12–27.

[23] Sheriffo Ceesay, Adam Barker, and Blesson Varghese. "Plug and play bench: Simplifying big data benchmarking using containers." In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 2821–2828.

[24] Dheeraj Chahal, Rupinder Virk, and Manoj Nambiar. "Performance extrapolation of io intensive workloads: Work in progress." In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM. 2016, pp. 105–108.

[25] Aurelien Charbon, B Harrington, B Fields, T Myklebust, S Jayaraman, and J Needle. "NFSv4 Test Project." In: *Proceedings to the Linux Symposium*. 2006.

[26] Hong-Mei Chen, Rick Kazman, and Serge Haziyev. "Agile big data analytics development: An architecture-centric approach." In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. IEEE. 2016, pp. 5378–5387.

[27] Hong-Mei Chen, Rick Kazman, and Serhiy Haziyev. "Strategic prototyping for developing big data systems." In: *IEEE Software* (2016).

[28] Min Chen, Shiwen Mao, and Yunhao Liu. "Big data: A survey." In: *Mobile networks and applications* 19.2 (2014), pp. 171–209.

[29] Ming Chen et al. "Linux NFSv4.1 Performance Under a Microscope." In: *LISA*. 2014, pp. 137–138.

[30] Ming Chen et al. "Newer Is Sometimes Better: An Evaluation of NFSv4.1." In: *SIGMETRICS*. 2015, pp. 165–176.

[31] Brent Chun et al. "Planetlab: an overlay testbed for broad-coverage services." In: *ACM SIGCOMM Computer Communication Review* (2003).

[32] Florentin Clouet et al. "A unified monitoring framework for energy consumption and network traffic." In: *TRIDENTCOM*. 2015.

[33] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB." In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[34] Piet JH Daas, Marco J Puts, Bart Buelens, and Paul AM van den Hurk. "Big data as a source for official statistics." In: *Journal of Official Statistics* 31.2 (2015), pp. 249–262.

[35] Houssem Daoud and Michel R Dagenais. "Recovering disk storage metrics from low-level trace events." In: *Software: Practice and Experience* 48.5 (2018), pp. 1019–1041.

[36] Andrea De Mauro, Marco Greco, and Michele Grimaldi. "What is big data? A consensual definition and a review of key research topics." In: *AIP conference proceedings*. Vol. 1644. 1. AIP. 2015, pp. 97–104.

[37] Andrea De Mauro, Marco Greco, and Michele Grimaldi. "A formal definition of Big Data based on its essential features." In: *Library Review* 65.3 (2016), pp. 122–135.

[38] Elif Dede, Bedri Sendir, Pinar Kuzlu, Jessica Hartog, and Madhusudhan Govindaraju. "An Evaluation of Cassandra for Hadoop." In: *IEEE CLOUD* 2013 (2013), pp. 494–501.

[39] Elif Dede, Madhusudhan Govindaraju, Daniel Gunter, Richard Shane Canon, and Lavanya Ramakrishnan. "Performance evaluation of a mongodb and hadoop platform for scientific data analysis." In: *Proceedings of the 4th ACM workshop on Scientific cloud computing*. ACM. 2013, pp. 13–20.

[40] Mathieu Desnoyers and Michel R Dagenais. "The lttng tracer: A low impact performance and behavior monitor for gnu/linux." In: *OLS (Ottawa Linux Symposium)*. Vol. 2006. Citeseer, Linux Symposium. 2006, pp. 209–224.

[41] Thaer M Dieb and Koji Tsuda. "Machine Learning-Based Experimental Design in Materials Science." In: *Nanoinformatics*. Springer, Singapore, 2018, pp. 65–74.

[42] Christopher C Drovandi, Christopher Holmes, James M McGree, Kerrie Mengersen, Sylvia Richardson, and Elizabeth G Ryan. "Principles of experimental design for big data analysis." In: *Statistical science: a review journal of the Institute of Mathematical Statistics* 32.3 (2017), p. 385.

[43] GC Fox, S Jha, J Qiu, S Ekanazake, and Andre Luckow. "Towards a comprehensive set of big data benchmarks." In: *Big Data and High Performance Computing* 26 (2015), p. 47.

[44] Cecilia Fredriksson, Farooq Mubarak, Marja Tuohimaa, and Ming Zhan. "Big data in the public sector: A systematic literature review." In: *Scandinavian Journal of Public Administration* 21.3 (2017), pp. 39–62.

[45] Andrea Gandini, Marco Gribaudo, William J Knottenbelt, Rasha Osman, and Pietro Piazzolla. "Performance evaluation of NoSQL databases." In: *European Workshop on Performance Engineering*. Springer. 2014, pp. 16–29.

[46] John Gantz and David Reinsel. "Extracting value from chaos." In: *IDC iview* 1142.2011 (2011), pp. 1–12.

[47] Naveen Garg, Sanjay Singla, and Surender Jangra. "Challenges and Techniques for Testing of Big Data." In: *Procedia Computer Science* 85 (2016), pp. 940–948.

[48] Dimitris Giatsios, Apostolos Apostolaras, Thanasis Korakis, and Leandros Tassiulas. "Methodology and tools for measurements on wireless testbeds: The nitos approach." In: *Measurement Methodology and Tools*. Springer, 2013, pp. 61–80.

[49] Mahesh Gudipati, Shanthi Rao, Naju D Mohan, and Naveen Kumar Gajja. "Big data: Testing approach to overcome quality challenges." In: *Big Data: Challenges and Opportunities* 11.1 (2013), pp. 65–72.

[50] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. "Experimental methodologies for large-scale systems: a survey." In: *Parallel Processing Letters* 19.03 (2009), pp. 399–418.

[51] Rui Han, Lizy Kurian John, and Jianfeng Zhan. "Benchmarking big data systems: A review." In: *IEEE Transactions on Services Computing* 11.3 (2017), pp. 580–597.

[52] Rui Han, Zhen Jia, Wanling Gao, Xinhui Tian, and Lei Wang. "Benchmarking Big Data Systems: State-of-the-Art and Future Directions." In: *arXiv preprint arXiv:1506.01494* (2015).

[53] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. "OLTP through the looking glass, and what we found there." In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 981–992.

[54] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. "Mesos: A platform for fine-grained resource sharing in the data center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[55] Klaus Hinkelmann and Oscar Kempthorne. "Design and Analysis of Experiments." In: (2012).

[56] Dan Huang, Jun Wang, Qing Liu, Xuhong Zhang, Xunchao Chen, and Jian Zhou. "DFS-container: achieving containerized block I/O for distributed file systems." In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017.

[57] Karl Huppler. "The art of building a good benchmark." In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2009, pp. 18–30.

[58] Todor Ivanov, Tilmann Rabl, Meikel Poess, Anna Queralt, John Poelman, Nicolas Poggi, and Jeffrey Buell. "Big Data Benchmark Compendium." In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2015, pp. 135–155.

[59] Bart Jacob, Paul Larson, B Leitao, and SAMM Da Silva. "SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems." In: *IBM Redbook* (2008).

[60] HV Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M Patel, Raghu Ramakrishnan, and Cyrus Shahabi. "Big data and its technical challenges." In: *Communications of the ACM* 57.7 (2014), pp. 86–94.

[61] Raj Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 2008.

[62] Emmanuel Jeannot. "Experimental validation of grid algorithms: A comparison of methodologies." In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE. 2008, pp. 1–8.

[63] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. "AndroStep: Android Storage Performance Analysis Tool." In: *Software Engineering (Workshops)*. Vol. 13. 2013, pp. 327–340.

[64] Ivo Jimenez, Michael Sevilla, et al. "The Popper Convention: Making Reproducible Systems Evaluation Practical." In: *Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2017.

[65] Min-Gyue Jung, Seon-A Youn, Jayon Bae, and Yong-Lak Choi. "A Study on Data Input and Output Performance Comparison of MongoDB and PostgreSQL in the Big Data Environment." In: *Database Theory and Application (DTA), 2015 8th International Conference on*. IEEE. 2015, pp. 14–17.

[66] Stephen Kaisler, Frank Armour, J Alberto Espinosa, and William Money. "Big data: Issues and challenges moving forward." In: *2013 46th Hawaii International Conference on System Sciences*. IEEE. 2013, pp. 995–1004.

[67] Avita Katal, Mohammad Wazid, and RH Goudar. "Big data: issues, challenges, tools and good practices." In: *2013 Sixth international conference on contemporary computing (IC3)*. IEEE. 2013, pp. 404–409.

[68] Taghi M Khoshgoftaar and Pierre Rebours. "Improving software quality prediction by noise filtering techniques." In: *Journal of Computer Science and Technology* 22.3 (2007).

[69] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. "Parameter-aware I/O management for solid state disks (SSDs)." In: *IEEE Transactions on Computers* 61.5 (2012), pp. 636–649.

[70] John Klein and Ian Gorton. "Runtime performance challenges in big data systems." In: *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*. ACM. 2015, pp. 17–22.

[71] John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. "Performance evaluation of nosql databases: A case study." In: *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems*. 2015.

[72] John Klein, Ian Gorton, Laila Alhmoud, Joel Gao, Caglayan Gemici, Rajat Kapoor, Prasanth Nair, and Varun Saravagi. "Model-driven observability for big data storage." In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE. 2016, pp. 134–139.

[73] Alexandros Labrinidis and Hosagrahar V Jagadish. "Challenges and opportunities with big data." In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 2032–2033.

[74] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[75] John Lawson. *Design and Analysis of Experiments with SAS*. Chapman and Hall/CRC, 2010.

[76] Chuck Lever and Peter Honeyman. "Linux NFS Client Write Performance." In: *FREENIX Track, USENIX Annual Technical Conference*. 2002, p. 29.

[77] Nan Li, Anthony Escalona, Yun Guo, and Jeff Offutt. "A scalable big data test framework." In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–2.

[78] BPF manual page on Linux. "http://man7.org/linux/man-pages/man2/bpf.2.html." In: 2017.

[79] Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon, Ying Liu, and Hui Joe Chang. "Closing the functional and Performance Gap between SQL and NoSQL." In: *Proceedings of the 2016 International Conference on Management of Data*. ACM. 2016, pp. 227–238.

[80] Steve Lohr. "The age of big data." In: *New York Times* 11.2012 (2012).

[81] Xiaoqing Luo, Frank Mueller, Philip Carns, John Jenkins, Robert Latham, Robert Ross, and Shane Snyder. "Hpc i/o trace extrapolation." In: *Proceedings of the 4th Workshop on Extreme Scale Programming Tools*. ACM. 2015, p. 2.

[82] Xiaoqing Luo, Frank Mueller, Philip Carns, Jonathan Jenkins, Robert Latham, Robert Ross, and Shane Snyder. "ScalaIOExtrap: Elastic I/O Tracing and Extrapolation." In: *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE. 2017, pp. 585–594.

[83] Alexandra L'heureux, Katarina Grolinger, Hany F Elyamany, and Miriam AM Capretz. "Machine learning with big data: Challenges and approaches." In: *IEEE Access* 5 (2017), pp. 7776–7797.

[84] Shiyao Ma, Jingjie Jiang, Bo Li, and Baochun Li. "Custody: Towards data-aware resource sharing in cloud-based big data processing." In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2016, pp. 451–460.

[85] Sushil Govindnarayan Mantri. "Efficient In-Depth IO Tracing and its application for optimizing systems." PhD thesis. Virginia Tech, 2014.

[86] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S Pérez. "Spark versus flink: Understanding performance in big data analytics frameworks." In: *Cluster 2016-The IEEE 2016 International Conference on Cluster Computing*. 2016.

[87] Richard P Martin and David E Culler. "NFS sensitivity to high performance networks." In: *SIGMETRICS Performance Evaluation Review* 27.1 (1999), pp. 71–82.

[88] Matthew L Massie et al. "The ganglia distributed monitoring system: design, implementation, and experience." In: *Parallel Computing* (2004).

[89] Marta Mattoso, Claudia Werner, Guilherme Horta Travassos, Vanessa Braganholo, Eduardo Ogasawara, Daniel Oliveira, Sergio Cruz, Wallace Martinho, and Leonardo Murta. "Towards supporting the life cycle of large scale scientific experiments." In: *International Journal of Business Process Integration and Management* 5.1 (2010), pp. 79–92.

[90] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris performance and tools: DTrace and MDB techniques for Solaris 10 and OpenSolaris*. Prentice Hall, 2006.

[91] Zijian Ming, Chunjie Luo, Wanling Gao, Rui Han, Qiang Yang, Lei Wang, and Jianfeng Zhan. "BDGS: A scalable big data generator suite in big data benchmarking." In: *Advancing Big Data Benchmarks*. Springer, 2013, pp. 138–154.

[92] Ashlesha S Nagdive, Manish P Tembhurkar, and RM Tugnayat. "Overview on performance testing approach in big data." In: *International Journal of Advanced Research in Computer Science* 5.8 (2014).

[93] Denis Nelubin and Ben Engber. "Ultra-High Performance NoSQL Benchmarking." In: *White paper in Aerospike website.* `http://www.aerospike.com/wp-content/uploads/2013/01/Ultra-High-Performance-NoSQL-Benchmarking.pdf` (2013).

[94] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon web services uses formal methods." In: *Communications of the ACM* 58.4 (2015), pp. 66–73.

[95] Lucas Nussbaum and Olivier Richard. "Lightweight emulation to study peer-to-peer systems." In: *Concurrency and Computation: Practice and Experience* 20.6 (2008), pp. 735–749.

[96] Tobias Oetiker and Dave Rand. "MRTG: The Multi Router Traffic Grapher." In: *LISA*. Vol. 98. 1998, pp. 141–148.

[97] Rihards Olups. *Zabbix 1.8 network monitoring*. Packt Publishing Ltd, 2010.

[98] Zhonghong Ou, Zhen-Huan Hwang, Antti Ylä-Jääski, Feng Chen, and Ren Wang. "Is Cloud Storage Ready? A Comprehensive Study of IP-based Storage Systems." In: *UCC15* (2015).

[99] Claus Pahl and Brian Lee. "Containers and clusters for edge cloud architectures–A technology review." In: *FiCloud*. IEEE. 2015.

[100] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. "Cloud container technologies: a state-of-the-art review." In: *IEEE Transactions on Cloud Computing* (2017).

[101] Fawaz Paraiso, Challita Stéphanie, Al-Dhuraibi Yahya, and Philippe Merle. "Model-Driven Management of Docker Containers." In: *9th IEEE International Conference on Cloud Computing (CLOUD)*. 2016.

[102] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. "YCSB++: benchmarking and performance debugging advanced features in scalable table stores." In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 9.

[103] Elena Pesce, Eva Riccomagno, and Henry P Wynn. "Experimental Design Issues in Big Data. The Question of Bias." In: *arXiv preprint arXiv:1712.06916* (2017).

[104] Xiongpai Qin and Xiaoyun Zhou. "A survey on benchmarks for big data and some more considerations." In: *International Conference on Intelligent Data Engineering and Automated Learning*. Springer. 2013, pp. 619–627.

[105] Tilmann Rabl and Hans-Arno Jacobsen. "Big Data Generation." In: *Specifying Big Data Benchmarks*. Ed. by Tilmann Rabl, Meikel Poess, Chaitanya Baru, and Hans-Arno Jacobsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 20–27. ISBN: 978-3-642-53974-9.

[106] Peter Radkov, Li Yin, Pawan Goyal, Prasenjit Sarkar, and Prashant J Shenoy. "A Performance Comparison of NFS and iSCSI for IP-networked Storage." In: *FAST*. 2004, pp. 101–114.

[107] Arcot Rajasekar et al. "iRODS Primer: integrated rule-oriented data system." In: *Synthesis Lectures on Information Concepts, Retrieval, and Services* 2.1 (2010), pp. 1–143.

[108] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. "OMF: a control and management framework for networking testbeds." In: *ACM SIGOPS Operating Systems Review* 43.4 (2010), pp. 54–59.

[109] Olof Rensfelt, Lars-Ake Larzon, and Sven Westergren. "Vendetta – a tool for flexible monitoring and management of distributed testbeds." In: *TridentCom*. 2007.

[110] Cristian Ruiz, Emmanuel Jeanvoine, and Lucas Nussbaum. *Porting the Distem Emulator to the CloudLab and Chameleon testbeds*. Research Report RR-8955. Inria, Sept. 2016. URL: https://hal.inria.fr/hal-01372050.

[111] Abdulqawi Saif and Lucas Nussbaum. "Performance Evaluation of NFS over a Wide-Area Network." In: *COMPAS - Conférence d'informatique en Parallélisme, Architecture et Système*. Lorient, France, July 2016. URL: https://hal.inria.fr/hal-01327272.

[112] Abdulqawi Saif, Lucas Nussbaum, and Ye-Qiong Song. *MongoDB I/O Access Patterns are under the Microscope*. Annual PhD students conference IAEM Lorraine 2017. Poster. Oct. 2017. URL: https://hal.inria.fr/hal-01793531.

[113] Abdulqawi Saif, Lucas Nussbaum, and Ye-Qiong Song. "Performance Evaluation of MongoDB I/O access patterns." In: *CLOUD DAYS'2017*. Action transverse Virtualisation et Cloud du GdR RSD. Nancy, France, Sept. 2017. URL: https://hal.inria.fr/hal-01793479.

[114] Abdulqawi Saif, Lucas Nussbaum, and Ye-Qiong Song. "IOscope: A Flexible I/O Tracer for Workloads' I/O Pattern Characterization." In: *ISC High Performance 2018 International Workshops - WOPSSS'18*. Frankfurt, Germany, June 2018. URL: https://hal.inria.fr/hal-01828249.

[115] Abdulqawi Saif, Lucas Nussbaum, and Ye-Qiong Song. *On the Impact of I/O Access Patterns on SSD Storage*. Research Report RR-9319. Inria, Jan. 2020. URL: https://hal.inria.fr/hal-02430564.

[116] Abdulqawi Saif, Alexandre Merlin, Lucas Nussbaum, and Ye-Qiong Song. "MonEx: An Integrated Experiment Monitoring Framework Standing on Off-The-Shelf Components." In: *P-RECS 2018: 1st International Workshop on Practical Reproducible Evaluation of Computer Systems*. Tempe, AZ, United States, June 2018. DOI: 10.1145/3214239.3214240. URL: https://hal.inria.fr/hal-01793561.

[117] Abdulqawi Saif, Alexandre Merlin, Lucas Nussbaum, and Ye-Qiong Song. "Monitoring Testbed Experiments with MonEx." In: *Grid5000-FIT Spring School*. SILECS project, INRIA. Sophia Antipolis, France, Apr. 2018. URL: https://hal.inria.fr/hal-01793507.

[118] Abdulqawi Saif, Alexandre Merlin, Olivier Dautricourt, Maël Houbre, Lucas Nussbaum, and Ye-Qiong Song. "Emulation of Storage Performance in Testbed Experiments with Distem." In: *CNERT 2019 - IEEE INFOCOM International Workshop on Computer and Networking Experimental Research using Testbeds*. Paris, France, Apr. 2019, p. 6. URL: https://hal.inria.fr/hal-02078301.

[119] Habib Saissi, Péter Bokor, Can Arda Muftuoglu, Neeraj Suri, and Marco Serafini. "Efficient verification of distributed protocols using stateful model checking." In: *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE. 2013, pp. 133–142.

[120] Vikram A Saletore, Karthik Krishnan, Vish Viswanathan, and Matthew E Tolentino. "HcBench: Methodology, development, and full-system characterization of a customer usage representative big data/hadoop benchmark." In: *Advancing big data benchmarks*. Springer, 2013, pp. 73–93.

[121] Tiago Salmito, Leandro Ciuffo, Iara Machado, et al. "FIBRE-an international testbed for future internet experimentation." In: *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. 2014.

[122] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. "Design and implementation of the Sun network filesystem." In: *Summer USENIX conference*. 1985.

[123] Luc Sarzyniec, Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. "Design and Evaluation of a Virtual Experimental Environment for Distributed Systems." In: *PDP2013 - 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*.

[124] Luc Sarzyniec, Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. "Design and evaluation of a virtual experimental environment for distributed systems." In: *PDP*. 2013.

[125] Viola Schiaffonati and Mario Verdicchio. "Computing and experiments." In: *Philosophy & Technology* 27.3 (2014), pp. 359–376.

[126] Elizabeth D Schifano, Jing Wu, Chun Wang, Jun Yan, and Ming-Hui Chen. "Online updating of statistical inference in the big data setting." In: *Technometrics* 58.3 (2016), pp. 393–403.

[127] Jiri Schindler. "I/O characteristics of NoSQL databases." In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 2020–2021.

[128] Jiri Schindler. "Profiling and analyzing the I/O performance of NoSQL DBs." In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 41. 1. ACM. 2013, pp. 389–390.

[129] Jiri Schindler, Anastassia Ailamaki, and Gregory Ganger. "Matching Database Access Patterns to Storage Characteristics." In: *VLDB PhD Workshop*. 2003.

[130] Jiri Schindler and Gregory R Ganger. "Matching application access patterns to storage device characteristics." PhD thesis. PhD thesis. Carnegie Mellon University, 2004.

[131] J Schulist, D Borkmann, and A Starovoitov. *Linux socket filtering aka Berkeley packet filter (BPF)*. 2016.

[132] Gwen Shapira and Yanpei Chen. "Common Pitfalls of Benchmarking Big Data Systems." In: *IEEE Transactions on Services Computing* 9.1 (2016), pp. 152–160.

[133] Suchakrapani Datt Sharma and Michel Dagenais. "Enhanced Userspace and In-Kernel Trace Filtering for Production Systems." In: *Journal of Computer Science and Technology* 6 (2016), pp. 1161–1178.

[134] S Shepler, M Eisler, and D Noveck. "Network file system (NFS) version 4 minor version 1 protocol." In: *The Internet Engineering Task Force (IETF)* (2010).

[135] Spencer Shepler, Mike Eisler, David Robinson, Brent Callaghan, Robert Thurlow, David Noveck, and Carl Beame. "Network file system (NFS) version 4 protocol." In: *Network* (2003).

[136] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. "Clash of the titans: MapReduce vs. Spark for large scale data analytics." In: *Proceedings of the VLDB Endowment* 8.13 (2015), pp. 2110–2121.

[137] Manpreet Singh, Maximilian Ott, Ivan Seskar, and Pandurang Kamat. "ORBIT Measurements framework and library (OML): motivations, implementation and features." In: *Tridentcom*. 2005.

[138] Huaiming Song, Xian-He Sun, and Yong Chen. "A Hybrid Shared-nothing/Shared-data Storage Architecture for Large Scale Databases." In: *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2011, pp. 616–617.

[139] Alexei Starovoitov. "`https://lwn.net/Articles/598545/`." In: 2014.

[140] Maarten van Steen and Andrew S Tanenbaum. "A brief introduction to distributed systems." In: *Computing* 98.10 (2016), pp. 967–1009.

[141] Martin Strohbach, Jörg Daubert, Herman Ravkin, and Mario Lischka. "Big Data Storage." In: *New Horizons for a Data-Driven Economy*. Springer, 2016, pp. 119–141.

[142] Phasakorn Sukheepoj and Natawut Nupairoj. "Adaptive I/O Bandwidth Allocation for Virtualization Environment on Xen Platform." In: *ICIIP*. 2017.

[143] Byung Chul Tak, Chunqiang Tang, Hai Huang, and Long Wang. "Pseudoapp: Performance prediction for application migration to cloud." In: *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*. IEEE. 2013, pp. 303–310.

[144] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. "Virtual Machine Workloads: The Case for New NAS Benchmarks." In: *FAST*. 2013, pp. 307–320.

[145] Mininet Team. *Mininet*. 2014.

[146] Matti Tedre and Nella Moisseinen. "Experiments in computing: A survey." In: *The Scientific World Journal* 2014 (2014).

[147] Rakshit S Trivedi, I Nilavalagan, and Jayant R Haritsa. "CODD: constructing dataless databases." In: *Proceedings of the Fifth International Workshop on Testing Database Systems*. ACM. 2012, p. 4.

[148] Hal R Varian. "Beyond big data." In: *Business Economics* 49.1 (2014), pp. 27–31.

[149] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. "Apache hadoop yarn: Yet another resource negotiator." In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[150] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. "Challenges and Solutions for Tracing Storage Systems: A Case Study with Spectrum Scale." In: *ACM Trans. Storage* 14.2 (Apr. 2018), 18:1–18:24. ISSN: 1553-3077. DOI: 10.1145/3149376. URL: http://doi.acm.org/10.1145/3149376.

[151] Brice Videau, Corinne Touati, and Olivier Richard. "Toward an experiment engine for lightweight grids." In: *MetroGrid*. 2007, p. 22.

[152] Rupinder Virk and Dheeraj Chahal. "Trace replay based i/o performance studies for enterprise workload migration." In: *2nd Annual Conference of CMG India, page Online*. 2015.

[153] Ke Wang, Ning Liu, Iman Sadooghi, Xi Yang, Xiaobing Zhou, Tonglin Li, Michael Lang, Xian-He Sun, and Ioan Raicu. "Overcoming hadoop scaling limitations through distributed task execution." In: *2015 IEEE International Conference on Cluster Computing*. IEEE. 2015, pp. 236–245.

[154] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. "Bigdatabench: A big data benchmark suite from internet services." In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE. 2014.

[155] Jonathan Stuart Ward and Adam Barker. "Undefined by data: a survey of big data definitions." In: *arXiv preprint arXiv:1309.5821* (2013).

[156] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: A Scalable, High-performance Distributed File System." In: *OSDI*. 2006, pp. 307–320.

[157] Yiqi Xu and Ming Zhao. "Ibis: interposed big-data i/o scheduler." In: *Proceedings of the 25th ACM HPDC*. 2016.

[158] Marvin V Zelkowitz and Dolores Wallace. "Experimental validation in software engineering." In: *Information and Software Technology* 39.11 (1997), pp. 735–743.

[159] Jianyong Zhang, Anand Sivasubramaniam, Alma Riska, Qian Wang, and Erik Riedel. "An interposed 2-level I/O scheduling framework for performance virtualization." In: *ACM SIGMETRICS Performance Evaluation Review*. 2005.

[160] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. "Storage performance virtualization via throughput and latency control." In: *ACM Transactions on Storage (TOS)* 2.3 (2006), pp. 283–308.

DECLARATION

I declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research. I also undertake here that this thesis has not been submitted for any other degree or professional qualification.

Nancy, France, October 2019

Abdulqawi Saif

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/