Communauté
UNIVERSITÉ Grenoble Alpes

## THÈSE

Pour obtenir le grade de

**DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES**

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

**Matthieu CANEILL**

Thèse dirigée par **Noël De Palma, Professeur, Université Grenoble Alpes**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

# Contributions to large-scale data processing systems

# Contributions aux systèmes de traitement de données à grande échelle

Thèse soutenue publiquement le **5 février 2018**,
devant le jury composé de :

**Madame SIHEM AMER-YAHIA**
DIRECTRICE DE RECHERCHE, CNRS/UNIVERSITÉ GRENOBLE ALPES, Examinatrice, Présidente du jury
**Monsieur DANIEL HAGIMONT**
PROFESSEUR, INP TOULOUSE - ENSEEIHT, Rapporteur
**Monsieur JEAN-MARC MENAUD**
PROFESSEUR, IMT ATLANTIQUE BRETAGNE-PAYS DE LA LOIRE, Rapporteur
**Monsieur NOËL DE PALMA**
PROFESSEUR, UNIVERSITÉ GRENOBLE ALPES, Directeur de thèse

To my family.

# Contents

# List of Figures

# Acknowledgments

It takes a lot of energy to write a thesis, an energy I wouldn't have found without the help of many people, towards whom I'm infinitely indebted. The following list is by no means exhaustive.

My first thoughts go to my PhD advisor, Prof. Noël De Palma. I want to express my gratitude for giving me this opportunity, guiding me in the field during all these years, and moreover for encouraging me to take initiatives and helping me refine my ideas. I really enjoyed the challenge thanks to the friendly and trustful atmosphere.

Many thanks go to the defense jury: my two reviewers, Prof. Daniel Hagimont and Prof. Jean-Marc Menaud, for taking the time to read this thesis and providing insightful comments, and Sihem Amer-Yahia (Directrice de Recherche) for accepting to be the president of the jury.

I want to thank my co-authors, Ahmed El Rheddane, Vincent Leroy, Noël De Palma, Ali Ait-Bachir, Bastien Dine, Rachid Mokhtari, and Yagmur Gizem Cinar. It was a pleasure to work together, and I learned a lot throughout the process.

A special note goes to my colleagues at the ERODS team, for all the passionate debates in the coffee room, the shared meals, the debugging help, and more importantly the friendships which arose.

I'm grateful to Stefano Zacchiroli, for initiating me to academic research and article writing, and confirming my inclination towards free software. I probably wouldn't have started a PhD without his valuable input.

The music artists Ratatat, Griz, Gramatik and Pretty Lights, the Volkshaus Biergarten in Leipzig, and all the PhD comics by Jorge Cham have helped a lot in setting up my mood for writing, and they more than deserve their place here.

Last, but not least, I'm very, very grateful to my parents, my brothers and sister, my friends, and Lena. Thank you very much for all the support, without which nothing would have been possible. *Merci infiniment.*

# Abstract

Decades after the foundations of the Internet have been layered, both on technical solutions (open protocols, exponential hardware improvements) and social values (net neutrality, global connectivity), its ability to shift to different paradigms continuously give researchers new and exciting challenges to solve.

The move to cloud computing is not new; yet the field is expanding, and it is likely this is only the beginning. As more and more devices get an IP address assigned (ranging from personal computers, to commodity servers, supercomputers, smartphones, or the huge range of IoT devices), and produce and consume data in complex systems, the needs arise to store, analyze, aggregate, and re-distribute this data. Thousands of datacenters around the world, often comprised of hundreds of thousands of servers, are getting more powerful and complex.

We propose in this thesis to analyze specific sub-fields of datacenter systems, at different abstraction layers, but for the same purpose: improving the efficiency of large scale data processing.

The first study we conduct answers the following question: how to design and integrate a scalable system, able to analyze and store millions of monitoring metrics, while making real-time predictions on their future behaviour using machine learning algorithms. This is driven by an industrial challenge, and is the general scope of the Smart Support Center project. Its strength lies with its direct connection to the industry, and the insights we provide about the research challenges we studied have a direct industrial impact, as the developed solutions are in production.

The second study provides a low-level optimization of how real-time data is ingested in a datacenter, and co-located within relevant data for its processing, while avoiding a network bottleneck. Its implementation does statistical analysis of the relationships between smaller pieces of the incoming data, and an oracle decides how they should be handled. It stays up-to-date by continuously monitoring the evolution of data correlations and re-generating optimized routing tables.

Finally, our third study dives in programming models for large scale data analysis. We have realized it is difficult to write, execute, maintain, share and improve distributed programs to extract intelligent information from scattered data. While many frameworks add abstraction layers to make this task easier, we went one step further, by defining a way to write data processing computations in a descriptive, rather than programmatic, way. This is done by assembling blocks of code in a directed graph, which brings many advantages over writing conventional source code, such as the possibility of manipulating programs in a high-level fashion. The data model, along with the reference framework we developed, can be a foundation for innovative ways to write and execute programs, especially for non-specialists.

# Résumé en Français

La tendance des organisations à se tourner vers l'infonuagique s'est progressivement développée ces dernières années, au point de voir se déployer des milliers de centres de données à travers le monde. Ces centres peuvent contenir plusieurs centaines de milliers d'ordinateurs, qui produisent et traitent en permanence une énorme quantité de données. Ainsi, les systèmes qui les régissent sont de plus en plus puissants et complexes, ouvrant la porte à de nouveaux défis.

Nous proposons dans cette dissertation d'en analyser quelques-uns, à différents niveaux de la pile logicielle dont sont compris les systèmes de traitement de ces données massives, dans le but d'améliorer leurs performances et leur accessibilité.

## Prédiction de métriques de supervision

La première étude que nous menons à bout s'inscrit dans le projet Smart Support Center. Ce projet de recherche est partagé entre plusieurs entreprises, dont la spécialité est la supervision d'infrastructures informatiques et les centres de support technique, et deux équipes de recherche spécialisées en apprentissage automatique, extraction de connaissances, et systèmes. Son but est de développer un ensemble de logiciels et méthodes, sur une infrastructure distribuée, capable d'ingérer l'ensemble des métriques relevées par des agents de supervision, afin de les stocker et de les analyser. Leur traitement en temps réel permet d'effectuer des prédictions, afin de déceler en avance les serveurs susceptibles de tomber en panne, dans le but de soulager les équipes techniques et de maintenir les accords de niveau de service.

Pour ce faire, nous proposons et testons une architecture distribuée. Les agents de supervision, qui surveillent en permanence l'état des machines et des services qui fonctionnent dessus, reportent des métriques, organisées en séries temporelles, tels le nombre de processus en cours d'exécution ou encore le taux de remplissage d'une partition sur un disque dur. Ces mesures sont calculées selon une période prédéfinie, en général comprise entre une et plusieurs minutes, et représentent ainsi l'évolution dans le temps de métriques particulières. Dès qu'elles sont calculées, ces mesures sont transmises à un courtier de messages, qui va se charger à la fois de les stocker dans une base de données Apache Cassandra, et les transmettre à Apache Spark.

Cassandra est une base de données distribuée, orientée colonnes. Grâce à un modèle de communication et de synchronisation en pair-à-pair, elle n'a pas de point individuel de défaillance. De plus, elle exhibe de hautes performances, et passe à l'échelle au moins jusqu'à plusieurs dizaines de milliers de nœuds. Nous utilisons donc Cassandra comme base résiliente et autoritaire pour le stockage de toutes les métriques, et des paramètres des modèles de prédiction appris.

Afin de prédire le futur comportement des métriques, et après analyse de leurs tendances habituelles, nous avons sélectionné la régression linéaire comme premier algorithme d'apprentissage automatique, pour sa simplicité et ses performances. Ce choix nous permet d'extraire des tendances générales, tout en évitant un grand nombre de faux positifs dus aux pics, habituels dans le cas de la supervision de systèmes. Pour l'implémenter, nous utilisons Spark, dont l'intérêt principal est de distribuer des tâches sur un grand nombre de machines, tout en gérant les pannes.

Après moult optimisations, nous obtenons des performances qui passent à l'échelle de manière linéaire, et une métrique est prédite sur un cœur de processeur en environ une seconde. Cette solution robuste permet de superviser et prédire un grand parc de machines avec peu de ressources. Les détails de cette architecture et son évaluation sont donnés dans le chapitre 3.

## Routage de données et localité

Dans cette étude, nous nous intéressons aux données qui atteignent des systèmes de traitement distribués en temps réel. Nous constatons que ces données présentent souvent de fortes corrélations entre elles, ce qui permettrait d'exploiter au mieux la colocation de messages traitant d'un même sujet, au moment du choix de leur routage. Par exemple sur Twitter, les mots-clics ont tendance à se développer autour de régions géographiques, souvent en rapport avec des évènements extérieurs. Ainsi, un système effectuant des analyses de données sur les micromessages, aura un intérêt à colocaliser les tâches traitant un mot-clic particulier avec les tâches traitant une région géographique particulière, afin de diminuer la charge sur le réseau.

Nous proposons d'identifier ce type de corrélations en temps réel (car elles évoluent dans le temps), d'implémenter un équilibreur de charge pour acheminer les messages vers les bons nœuds en fonction de leur contenu et des corrélations détectées, et d'écrire un algorithme de reconfiguration, qui permet de conserver l'état des nœuds lorsque les tables de routage sont détectées.

Nous implémentons notre solution dans Apache Storm, un moteur de traitement distribué de données en temps réel, et mesurons les gains : le nombre de messages capables d'être traités par seconde (le débit) est considérablement augmenté, jusqu'à 150%, avec des jeux de données réels reproduisant des entrées de Twitter et Flickr. Les détails des algorithmes, de leur implémentation, des jeux de données et des résultats obtenus sont décrits dans le chapitre 4.

## $\lambda$-blocks

Notre dernière étude porte sur les modèles de programmation de traitement de données à grande échelle. Nous constatons qu'il est difficile d'écrire des programmes distribués, même en utilisant des environnements de développement spécialisés, peu accessibles aux non-spécialistes. De plus, maintenir et améliorer ces programmes tout en évitant la duplication de code n'est pas tâche aisée.

Nous proposons ainsi $\lambda$-blocks, un environnement de développement pour écrire des algorithmes de manière descriptive, et non programmative. Grâce à une librairie de blocs (morceaux de code implémentant des tâches courantes, ou faisant appel à des libraries spécialisées tel Spark), il est possible grâce à un simple modèle de données de décrire un graphe orienté, dont les sommets sont des blocs et leurs paramètres, et les liens les connexions entre les blocs, représentant effectivement les flux de données.

Écrire un programme en décrivant un graphe présente un grand nombre d'avantages. En tant que modèle de composants, $\lambda$-blocks exhibe des propriétés tels la boîte noire (nul besoin de connaître les détails d'implémentation d'un composant pour l'utiliser), la réutilisabilité (un composant peut aisément appartenir à plusieurs graphes, et un graphe lui-même peut être réutilisé en tant que sous-graphe d'un programme plus large), et le remplacement (si un composant possède la même interface qu'un autre, il peut le remplacer, par exemple s'il exhibe de meilleures performances). Ce simple modèle de graphes, accompagné d'une riche librairie de composants pré-programmés, permet ainsi d'écrire des algorithmes de transformation de données sans écrire de code source.

Nous implémentons également un système de modules d'extension, qui permet, avant ou pendant l'exécution, de manipuler le graphe, afin de le déboguer ou de l'optimiser par exemple. Nous ouvrons ainsi la porte à une manière de raisonner sur un programme de transformation de données.

Nous comparons, en termes de performances, l'utilisation ou non de $\lambda$-blocks, et obtenons une différence maximale de 50 ms, ce qui est négligeable par rapport à la durée moyenne de ce type de programme, variant de quelques secondes à quelques heures (voire beaucoup plus dans certains cas). Tous les détails de $\lambda$-blocks, des exemples d'utilisation, et son évaluation sont dans le chapitre 5.

# Chapter 1

# Introduction

*« Two of the most famous products of Berkeley are LSD and Unix. I don't think that is a coincidence. »*

The UNIX-HATERS Handbook [115]

## Contents

## 1.1 Context

This thesis is the result of my PhD studies in University of Grenoble Alpes, more precisely in the ERODS team (Efficient and RObust Distributed Systems) of the Laboratoire d'Informatique de Grenoble. As its name suggests, this research team focuses on computer systems, and more particularly distributed systems. While many areas are covered by the different team members (such as multi-core systems, high-performance computing, kernels, and others), I have focused my research on the programming of heterogeneous clusters comprised of commodity servers.

This is a large area, especially when taking into account the general move of the industry towards cloud computing, and the many different research teams exploring these new possibilities. However there are recurrent general questions that arose: the improvement of the performances of datacenters; and the improvement and accessibility of programming models to perform distributed computing and large scale data processing.

These open questions are trimmed down to smaller problems, observed through the lenses of the Smart Support Center [14] project. It is a research project, involving actors from both industry and academia, which aims to develop software and methods for a scalable platform to handle issues in datacenters, through monitoring metrics collection, predictive analysis, ticketing, and all their interactions. This project framed my thesis, and gave me opportunities to work on the design and infrastructure of a distributed system able to process, store, and predict time series; a low-level routing algorithm to maximize data locality on a load-balancer; and finally a novel programming model to reduce the complexities of processing large amounts of data in complex systems.

## 1.2 Motivation

In 2013, a study reported that 90% of the data in the world had been generated only during the previous two years [122]. The exponential growth of produced data and the novel techniques to analyze it and extract meaningful information has triggered an evolution of how data is used in many sectors. Computer systems able to process this data get more and more complex, and building end-to-end processes for these purposes usually requires deep knowledge from specialists coming from different fields.

As of today, building a large user-facing service running in a datacenter requires, on the technical side, knowing the intrinsics of the service logic, operating systems, networks, monitoring, data storage, real-time data management, distributed systems, and more and more often, data analytics.

For example, the Smart Support Center project is building a system receiving monitoring data from thousands of machines, constantly reporting about the state of their services and their use of hardware resources. The platform must store and reliably replicate this data, analyze it in real-time to extract relevant knowledge about machine failures and performances, predict potential future problems with machine learning prediction algorithms, correlate the issues with a ticketing system, while constantly staying online and providing guarantees about scaling and latency. To build such a system, it is necessary to aggregate the combined knowledge coming from different disciplines, and while it gets more and more complicated, efficient layers must be added, providing properties such as fault-tolerance, latency, and simplicity, on which business logic can be built.

Ideally, these layers must be independent from the rest of the system, so that they can be used elsewhere, and replaced when necessary. They become more apparent when the problem is divided into smaller ones, which we describe in the next sections.

## 1.3 Problems description and challenges

The problems we tackle in this thesis are blockers in the construction of large systems to process data in a reliable way, while providing guarantees about system properties. More precisely, we answer the following questions:

- **How can we build a system to process real-time monitoring metrics and give predictions about future issues?** While some traditional monitoring systems have measures to raise warnings ahead of failures occurring, they are often unreliable. Moreover, they suffer from scaling issues, and are often centralized, leading to single points of failure. Building an end-to-end pipeline for processing these metrics and giving reliable information as output requires a mix of distributed databases, machine learning, messaging, and workload distribution. To be useful, it needs to scale linearly and be sufficiently fast to let system administrators fix problems without raising any service interruption. As the first (and sometimes only) health metric of a datacenter, a monitoring system also needs to be very reliable.

- **How can we improve the latency and throughput of a distributed real-time processing engine?** When real-time data hits a distributed processing system, its load needs to be balanced across different machines. More than often, different data points show strong correlations between them, and benefit from being handled on the same computer. This is a hard task, because these correlations are not static and need to be dynamically discovered. This is a low-level problem involving correlations discovery, dynamic generation of routing tables, graph clustering, and hot reconfiguration involving state relocation.

- **How can we abstract the complexity of programming large scale data processing applications?** In order to develop a system which reads data, transforms it, and saves it, it is often necessary to use multiple programming frameworks and libraries, especially if it is a distributed application. This requires knowledge in different areas, and leads to writing programming patterns that are repeated across different programs. This is also error-prone and difficult to debug. One solution to this problem is to build an abstraction layer which brings to developers the building blocks they need, leaving them the only responsibility of assembling and configuring them according to their needs. Adding such a layer hides complex code which can be seamlessly reused in different applications.

## 1.4 Contributions

My research contributions belong to bigger projects, which include different researchers and engineers, from academic research teams and industry. They can be summarized as follows:

- *Online Metrics Prediction in Monitoring Systems* (Chapter 3): This is a joint project between different companies in Grenoble (Coservit, HPE), and two research teams of the LIG laboratory (AMA, ERODS). Its contribution is the description of a scalable, distributed architecture, for issuing failure predictions based on monitoring time series. It has been carefully optimized and evaluated. An article summarizing the system and the results obtained has been accepted for publication in the DCPerf workshop of the Infocom conference [44].

- *Data-aware routing* (Chapter 4): this is also a joint project, between the research teams SLIDE and ERODS. Its contributions are an algorithm to detect correlations in the fields of real-time data, a way to route it to favor physical locality, and a reconfiguration algorithm for the routing tables, to keep the system dynamic. This chapter is very similar to our published article in the Middleware 2016 conference [45].

- *Lambda-blocks* (Chapter 5): the contribution is a programming framework, which allows to write data processing programs in a novel, more efficient manner, while taking advantage of all the specialized tools which exist for distributed processing. An article has been submitted to an international conference.

## 1.5 Outline

- **Chapter 2**. We first present a general overview of cloud computing, datacenter programming, large-scale data processing and components models, in order to give the big picture in which our contributions find their context. The presented topics and paradigms are used throughout this thesis, and when relevant more details are given. Our contributions are compared with their related work in their own chapters.

- **Chapter 3**. We describe and evaluate the system developed to predict machine failures with monitoring metrics, at the heart of the Smart Support Center project. It is a scalable platform, able to process millions of data points, evaluated with production data, and currently in use at Coservit.

- **Chapter 4**. We then dive in load-balancing real-time data. We present an algorithm to find correlations between the different fields of real-time messages, in order to route them to favor data locality and avoid network bottlenecks. We implemented this system for the Apache Storm engine, and evaluated it with Twitter messages and Flickr data.

- **Chapter 5**. We describe a programming model to write data processing applications without writing code, by assembling blocks. Its implementation as a framework is very fast, has a built-in plugin system, is extensible through simple decorators to write new blocks, and proposes a novel way to programmatically manipulate computation graphs. We evaluated it with different algorithms on Wikipedia datasets.

- **Chapter 6**. Finally, we give our conclusions and present some future work opportunities.

Chapter 2

# Background overview

« *The problem with distributed systems, is that no matter what the question is, the answer is inevitably 'It Depends'.* »

tef [108]

## Contents

This background shows the evolution of computer systems, and gives insights on how scientists and developers reached the point of managing petabytes of data on large datacenters, providing programmable interfaces to interact with stored and real-time data, in a reliable and fast manner. We dive in some architectures, explore some challenges and solutions, and give details about a few selected and well-recognized ecosystems. In the last part, we describe software architectures which promote components and their desirable properties for designing systems, which we will use as a foundation for $\lambda$-blocks. Although some presented systems seem not to have obvious connections with the original work described in the further sections of this thesis, they are either connected in the greater ecosystem of data processing software and practices, or have design ideas that inspired ours.

## 2.1 Cloud computing and datacenters

In the last decade, we have witnessed a move from in-house servers to cloud computing, leading to many changes in how organizations handle their IT infrastructure and their data. We describe in this section some technical foundations that made this possible, foundations upon which we built our solutions.

### 2.1.1 The emergence of cloud computing

Within the last years, the servers infrastructure of organizations has seen a large growth, due to the ubiquity of Internet access points for end users, diminution of hardware costs, and proliferation of data. Maintaining such an infrastructure has a cost, and requires a deep knowledge in systems and networks. This has led to a move to cloud computing [27]: computers and services available on demand, tailored to the needs of small servers as well as large distributed systems.

A cloud is a set of servers running special software, able to deploy and configure virtual machines, virtual networks, and necessary operating systems. Cloud computing leverages *time sharing*, based on two premises:

- A server is often not using all its resources;
- An application load evolves with time.

To cope with this, cloud infrastructures make a heavy use of virtual machines (many VMs running on a physical server), and propose elastic computing: the automatic increase and decrease of resources depending on the service load. This way, cloud users only pay for the resources they use, because servers are shared with others. The economic advantages it provides, as well as the disappearance of the technical burden associated to managing their own servers infrastructure, has led organizations to quickly adopt cloud computing.

This in turn has given researchers and developers new challenges to solve, due to this new paradigm (resources on demand) and its properties, the always-increasing size of datacenters and their energy consumption, clients data privacy, and cloud availability through service-level agreements.

Different cloud models have emerged [86], giving users the choice on how they want to run their infrastructure. They include:

- **Infrastructure as a service (IaaS)**. With this model, the provider gives access to virtual machines, and it is up to the clients to provision them with operating systems, and configure and run their applications. Often, many services are proposed, such as storage nodes, virtual networks, IP addresses, load balancers, etc.

- **Platform as a service (PaaS)**. PaaS is more restricted than IaaS, in that users are given a pre-configured platform on which they run their applications. Databases, storage, network, and other layers are included in the model and can be relied upon. It is meant to let developers focus on the application alone and not the deployment and management operations.

- **Software as a service (SaaS)**. On SaaS platforms, the execution of a particular software is proposed to users. This removes the needs to run this software on dedicated machines, and effectively outsources a part of an application, for example its database or even its continuous integration environment. Advantages include the optimizations generally provided by specialists whose core of business if a certain software, or auto-scaling resources according to the clients' needs.

Clouds also present different deployment models. Private clouds are operated by the organizations which use them for their own applications, adding the costs of maintenance but removing the dependency upon a third-party provider. On the other hand, public clouds are proposed as an external service by specialized companies, which take the responsibility of running the physical infrastructure. It is also possible for organizations to use a mix of these two models, known as hybrid cloud: this allows for example to store sensitive data on a private cloud, while still benefiting from the external infrastructure for other tasks. Another use case is the elasticity: if resources in a private cloud become too busy, a public cloud can take some of the application load.

An example of a cloud management software is OpenStack [9]: it is a complete suite of tools to run virtual machines on many machines. It supports different hypervisors, can be controlled by an API and a graphical interface, manages virtual networks and virtual disks, etc. Many of the experiments shown in the later sections have been run on machines operated by OpenStack.

### 2.1.2 Virtual machines and hardware virtualization

Virtual machines (VMs) are systems embedded in others, in other words an emulation of an operating system. By the means of specialized software (and sometimes hardware virtualization capabilities), a system can create many smaller sub-systems, providing them a subset of the hardware capacities of the host machine, and mapping the input and output devices. Some systems emulate CPUs, which enables to run operating systems compiled for different CPU architectures.

This provides some advantages:

- **Isolation**: a virtual machine is isolated from the host system and the other VMs, and can only access what the host OS allows it to access. This adds a layer of security: when a guest system is compromised, it is hard for an attacker to escalate their access to the host system.
- **Resource-sharing**: a host server can divide its resources between different guests systems.
- **Reproducibility**: it is easier to create a well-defined computer (with regards to hardware specifications, operating system, and installed software) in a virtual machine, allowing applications to run in a known, reproducible environment.
- **Systems on demand**: with virtual machines, it is easier to create and throw away new systems (for example to run short-lived applications).
- **Different hardware**: by emulating different CPU architectures, it is possible to run systems compiled for different hardware without owning it. It induces a performance overhead, but is nonetheless useful.
- **Consumption measurements and billing**: since the guest system is supervised by the host system, it is easy to measure how much CPU time and memory it is consuming, and adjusting its provided resources accordingly. This can also be used to bill VM users only for what they consume.

To efficiently dedicate a computer to running multiple virtual machines, special systems called hypervisors have emerged. Hypervisors are starting, monitoring and killing virtual machines on demand. Two types of hypervisors exist: special software running on an operating system (for example QEMU [36]), or entire dedicated operating systems (for example Xen [32]).

### 2.1.3 Containers

Containers are similar to virtual machines in that they make it possible to run many guest systems within a host operating system. However, their implementation differ in the way the virtualization is achieved: instead of emulating hardware for an entire OS, the host kernel is used to containerize a file system and the running processes.

The origin of containers comes from the FreeBSD `jails` [75], a mechanism to isolate a set of files and processes from the rest of the system. This adds a layer of security, and provides a way to effectively share a system between different applications. Some benefits of virtual machines are lost (total isolation, hardware virtualization), but the performance overhead is much smaller.

The Linux kernel implements `cgroups`, a similar system, that makes it able to isolate processes, files, and resources (CPU cores, memory, network. . . ). Containers recently gained popularity when Docker [88] was introduced: based on Linux' cgroups, and adding a few bricks such as a union filesystem and a way to build a container by describing the required steps in a `Dockerfile`, it became an ecosystem for managing containers at scale.

### 2.1.4 Modern datacenters

Datacenters are the physical warehouses where a great number of servers and network capabilities are hosted. Dynamics and constraints of distributed systems running at the scale of hundreds of thousands of servers (sometimes across different geographical locations) have uncovered new challenges and new paradigms such as *datacenter programming*. Some of these challenges are introduced in the book "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines" [33]. A key point underlined in this book is the machine failure rate at scale, which constantly happens. In these environments, fault-tolerance is necessary, both for data and running computations, and as such must be part of the design of every system.

Some researchers have pointed the need for a datacenter operating system [121]. Arguing it is hard for scholars to work on such systems (because they lack hardware and operations teams), they raise interesting points, notably the fact that datacenter programming is hard and still in its inception, and that there is a need for new paradigms and abstractions to bring complex software development (able to run on large clusters) to non-experts.

### 2.1.5 Cluster schedulers

Cluster schedulers are an important layer of the foundations of datacenter operating systems. Their role is to allocate different computers (often virtual machines or containers) to the tasks that need them, such as long-running services or batch jobs. We dive in this section into three different schedulers, which present different design ideas.

The first one is Mesos [69]. It is a brick of the Berkeley Data Analytics Stack [59], a set of various software written for large-scale data processing, storage, scheduling, etc. The goal of Mesos is to divide the resources of a datacenter between different computation frameworks, while maximizing the resource utilization, in a way that is compatible with current and future frameworks.

Towards that end, Mesos proposes a two-step-scheduling: it first offers resources to frameworks, which can accept or reject them. If they accept, they in turn schedule their tasks on these resources. This has the benefit of allowing frameworks to manage fault-recovery and data locality, which would be hard to implement in a global scheduler not aware of frameworks' specificities.

On top of this abstraction, Mesos offers several other features: different allocation modules (fair between frameworks, or with priorities), fine-grained resource allocation (task isolation is implemented via an isolation module, for example containers, which makes node re-allocation fast), fault-tolerance for the master node via Zookeeper [70], signalling of the failed nodes to framework schedulers, etc.

It however has some limitations, for example it's hard to allocate tasks in an heterogeneous environment, where the biggest tasks which need a consequent number of nodes might starve (wait indefinitely). Since it is framework-agnostic, it doesn't know the dependencies between frameworks, which could leverage co-locality if they communicate data between each other. Finally, the compatible frameworks need to make their own scheduler compatible with Mesos' API and design.

In conclusion, Mesos provides a way to share a cluster between frameworks, and its resource offering model has been proved to work well in practice, improving cluster utilization and speeding frameworks jobs.

Omega [104], developed at Google, has the same goals of Mesos, maximizing resource usage in a datacenter between different frameworks, while trying to be more flexible than the two-step-scheduling abstraction. More precisely, Omega overcomes the fact that framework schedulers don't have an overall view of the cluster, and can't preempt tasks, which could help for instance with data locality when necessary.

Omega uses a different abstraction: shared-state scheduling. The entire state of the cluster is presented to all schedulers, which then make scheduling attempts. Different policies can be implemented in the scheduler to answer the queries in case of a conflict: gang-scheduling (all the nodes are allocated, or nothing; this can delay other jobs), or incremental placement (nodes are allocated progressively when they become available; the downside is that it can lead to deadlocks). This permits frameworks to begin their job while not all the nodes are allocated yet, and maximizes resource utilization.

As with Mesos, this supports many scheduling policies, because frameworks schedule their tasks on the allocated nodes themselves. Moreover, Omega supports long running services (for example user-facing services over HTTP) as well as small batch jobs, and can preempt other tasks towards data locality, which can be more efficient than only using idle resources.

Omega has been tested at Google on their workload with very good performance results.

Borg [113] is another software developed at Google. It is meant to be a cluster manager at Google's scale, for hundreds of thousands of machines. Even though it supports different isolation mechanisms, it generally uses containers, and hence it has been developed to manage a large pool of containers executing tasks, and sometimes communicating with each other. When a job is submitted, it is divided into tasks, which all have properties such as their requirements. On the other side, registered machines have physical attributes such as their number of CPU cores or their amount of memory. Finally, users have quotas for the maximum number of resources they can use in a time window.

The *Borgmasters* then match the jobs requirements with the available resources, and synchronize the *Borglets* (the slaves), which among other features run a web server to monitor their tasks, and restart them if they fail. Borg benefits from many optimizations, and an important lesson learned is that mixing resources (for production and development environments, service jobs or batch jobs, with different users, etc) is more efficient than clustering the machines between teams or job types.

Kubernetes [99] began as an open-source implementation of Borg, and is now a widely-used cluster manager, often along Docker. The evolution of container scheduling and management at Google with Omega, Borg and Kubernetes is described in [42].

### 2.1.6 Monitoring at scale

The last topic we introduce with regards to cloud computing is monitoring. Checking health of machines and services is particularly relevant in the context of distributed systems, where failures often occur.

Monitoring engines often follow Nagios' design [8]. It is based on a set of configuration files, which describe computers, their network hierarchy, and the services running on them. Nagios can then run monitoring plugins where it is necessary, for instance a plugin checking the liveliness of a PostgreSQL node would run on every server using PostgreSQL, periodically, and report the results to the engine.

An important point with plugins is their ability to report metrics, such as the free disk space on a hard drive, or the number of processes on a system. Since these metrics are checked regularly (often from every minute to every few hours), they can be stored and used as *time series*: a series of points indexed by their timestamp, showing the evolution of a metric. This opens a lot of possibilities for time series predictions, using for instance machine learning algorithms, and hence anticipation of machine failures.

This idea has been implemented to various degrees in the monitoring engine Zabbix [16], for hardware failure prediction [48], for capacity planning [7], equipment temperature [83], and many related domains.

## 2.2 Large-scale data processing

Managing data in large infrastructures requires more than shell one-liners. We present in this section the modern paradigms for distributing work across many computers, for both offline and real-time data.

### 2.2.1 The MapReduce paradigm

MapReduce [56] is a programming paradigm proposed by Google in 2004, which abstracts away the complexity of distributed programming, in order to easily distribute tasks on large clusters of machines.

A MapReduce developer must write their data processing algorithm in terms of a *map* function (a map function applies a set of transformations to every member of a list) and a *reduce* function (a reduce function reduces a list into a single element). It is worth noting that it may be complicated to express an algorithm in terms of MapReduce. When this is done, the map and reduce functions are submitted to the engine along with some metadata such as the number of required tasks for each step, and the framework will distribute these tasks, optimizing for data locality and taking care of failed tasks and stragglers (rescheduling their work on other machines). A set of facilities are also implemented in MapReduce, such as optimizations for the reduce functions, which can often be partially computed on the map nodes before the shuffle phase (when usually a lot of data is sent back to the reduce nodes, inducing network latency), the skipping of bad input data if it deterministically makes the node crash, counter functions, etc.

MapReduce gained a lot of popularity, and the model has been implemented in the open-source Hadoop ecosystem. A lot of academic literature has been written on the topic, to improve its programming model and performances. For example it is possible to apply database optimizations and static analysis to MapReduce programs to drastically improve their execution times [71]. Other optimizations are described in [30].

### 2.2.2 The Apache Spark ecosystem

Many different frameworks, building on MapReduce' success and improving its performances, programming model, and range of applications, have been implemented. We particularly focus on Apache Spark, which we have used in the further sections.

Spark [119] leverages the processing of data while keeping it in memory. This is a key difference with MapReduce: it can only run one pass on the input data, and iterative algorithms (which require the output of step $n$ to be the input of step $n+1$, such as PageRank) need to save data to disk and read it back later, adding a large overhead.

Resilient Distributed Datasets (RDDs) are at the core of Spark. They are special data structures, built from input data, and on which one can apply many operations. When methods on RDDs are called, Spark doesn't perform anything besides building a lineage graph: a directed acyclic graph representing the set of operations to apply. When the computation is required (for instance after calling a few transformation methods), Spark performs some optimizations on the lineage graph, and divides it into tasks which are distributed on the cluster. This approach works well with fault-tolerance: whenever a task fails, its input can be re-computed just following the lineage graph. However, this can be costly, that's why Spark implements mechanisms such as data checkpointing, to save data on disk after expensive steps.

This programming model is more flexible than MapReduce, because it doesn't limit the developer to express algorithms in terms of map and reduce functions, but a large set of transformations. Hence, MapReduce, database-inspired query languages, or graph algorithms can be easily implemented on top of RDDs. Moreover, the in-memory approach, the lineage graph recovery and the straggler mitigation give great performances, and it has been measured to be faster than Hadoop for a wide range of data processing applications.

Spark includes a graph library, GraphX [64]. It is meant as a system to unify graph processing and dataflow programming, such that it stays compatible with Spark's data structures. It adds two collections, vertices and edges, includes different graph algorithms, and many optimizations for computing parallel graph operations, for example lazy joins between vertices and edges, to avoid computing them if they are not used.

Spark SQL [28] is another library which is now part of Spark. Its goal is to implement relational data processing on Spark data structures. To that end, it proposes a Dataframe API, which implements relational operations on RDDs and external data sources. A Dataframe roughly corresponds to a table in an SQL schema, on which operations such as `select`, `join` and `groupby` can be applied. Catalyst takes care of optimizing the query plan, and it is an easily extendable component, on which particular optimizations for specific cases can be written.

MLlib [87] is another component of Spark. It is a machine learning library, which implements a range of algorithms on top of Dataframes. Due to the iterative nature of Spark's lineage graphs, machine learning algorithms are a good fit for them, and benefit from all the optimizations built into Spark. MLlib also provides an API for building machine learning pipelines easily.

Finally, Spark have stream-processing capabilities, which are described in Section 2.2.4.

### 2.2.3 Distributed frameworks performances

A lot of work has been done to improve the performances of distributed computing; we present in this section a selection of optimizations methods applied to Hadoop and Spark.

The behaviour of intensive workloads run on Intel processors with Hadoop and Spark is studied in [72]. Observed metrics include the execution times (confirming Spark is faster than Hadoop), disk input/output (the write/read ratios are similar for both frameworks, while Spark has a higher disk access frequency, probably due to its higher computation speed), memory bandwidth (Spark has more stable memory access patterns), page access frequency (for both frameworks, 80% of the requests access 20% of the pages), caches access (with the L2 cache having a high miss rate compared to the L1D and TLB caches), and branch prediction (Spark has a lower branch prediction miss rate than Hadoop). Conclusively, these frameworks can benefit from cache hierarchy optimizations, memory bandwidth optimizations, and disk input/output optimizations.

A study of Spark's bottlenecks is performed in [96]. Assumptions are often made when doing distributed computing: network and disk are bottlenecks, and stragglers are responsible for a high increase of job completion times. This analysis reveals that this is not often the case, and CPU is the bottleneck for diverse data processing computations. For this matter, the authors use *blocked time analysis*, measuring when and why the processes are blocked. The results are unexpected, as they show the network and disk latencies are quite small, and optimizing them leads to small job completion decreases. However, a lot of CPU time is spent serializing and deserializing compressed data; moreover, using the JVM can have a strong overhead compared to

C++. Spark can hence benefit from using carefully chosen serialization and compression algorithms, a choice which can vary according to the type of computation that is performed.

Hadoop has a large space of configuration parameters, which can greatly influence job completion times. A framework has been written to try to profile and optimize them [68]. It consists of a job profiler, which takes into account the program, the data, its configuration, and the cluster setup, able to provide cost estimates. Associated to a What-if engine, able to answer queries such as "if we change this parameter, what will be the impact on execution time?", the framework is able to give an optimized set of parameters. Profiles are created by instrumenting sampled running jobs and estimation/simulation through the What-if engine. The good accuracy of the engine (difference between predicted and actual results) comes from the fact it captures a lot of subtleties of MapReduce at a fine granularity. To enumerate the different parameters, it implements gridding (all the values on a grid are tested) or recursive random search (able to find local optimizations in randomly selected regions). This is done in clusters of the settings space, which are usually independent, such as the settings for the map phase and those for the reduce phase. The results are conclusive, as job speeds of almost an order of magnitude can be obtained.

Finally, [54] looks at the shuffle stage of Spark jobs, which can be a bottleneck because a lot of data is moving on the network. A lot of small files (map tasks × reducer tasks) are created during this phase, which are requested concurrently in a random order by the reducers; moreover, the inode cache can't contain them all, causing an overhead. An implemented solution reduces the number of files (number of cores × reducer tasks): on ext4 filesystems, this scales linearly, as opposed to Spark without this optimization. However, this is slower on ext3 filesystems, due to some differences in how physical data locality is achieved for large files, which works better with ext4. This optimization led to a reduction of 50% of execution time for some jobs, which shows the complex interactions the frameworks have with filesystems. Other potential improvements are suggested, such as an overlap between the map and reduce phases, or a shuffle done completely in-memory.

### 2.2.4 Stream processing frameworks

Stream processing engines are frameworks able to handle real-time data, as opposed to batch processing, the one-time transformation of data already recorded on storage. Different challenges are associated with stream processing: load balancing (the throughput of incoming data can vary a lot, and decisions must often be taken on where to route it), latency (extracting relevant metrics from real-time data must often be fast, to provide timely and relevant answers), and reliability (if incoming data is lost before hitting a disk, it can't be recovered). We introduce in this section a few frameworks which exhibit different designs to manage streaming data.

Spark Streaming implements the Discretized Streams (D-Streams) abstraction [120]. It is part of the core of Apache Spark, and uses some of its data structures. Its principle is as follows: incoming data is grouped in an RDD (on which all RDD operations are available), and the lineage graph associated to it is computed repeatedly, for example every second. This is why it is called Discretized Streams: instead of running continuous queries, Spark Streaming does batch processing on smaller subsets of the data, as it reaches the system. This abstraction is called micro-batch processing. Hence, it benefits from all the features of Spark: fault-tolerance, recovery, straggler mitigation, and the rich API of available transformations. On top of this, Spark Streaming adds specific streaming operators: windowing (the grouping of records from past time intervals on a sliding window), incremental aggregation over a sliding window, state tracking (to transform a list of events into a list of updated states). In order to not lose data in case of a failure, the input records are saved before hitting the system. Moreover, for operators maintaining a state (for example incremental aggregation), it is also stored in an RDD, and can be recovered the same way. Spark Streaming implements "exactly-once" semantics, meaning a record will traverse the lineage graph exactly one time. Finally, it is possible to join stream data with historical data stored in an RDD, to allow computations to be performed combining both sources.

Spark Streaming implements many optimizations and favours data locality between nodes, which makes it fast, and scales linearly. However, by design, there is always a small latency, at least as high as the chosen batch interval; for this reason it doesn't make it suitable for systems that require a very low latency, such as trading.

Apache Storm [111] is a streaming engine which was initially acquired by Twitter. To use it, a developer defines a set of spouts (data sources) and bolts (operators transforming data), linked together in a topology: a graph where each vertice is an operator (or a data source), and each link represents the flow of data between two operators. A message consists of a tuple, a set of records containing arbitrary fields. To efficiently distribute tasks, Storm replicates each of them in different executors, placed on different machines. It then implements different forwarding techniques for the flowing data between the operators: shuffle grouping (a record is sent to a random node), fields grouping (one field of the message is chosen to be a key, and all messages belonging to the same key get routed to the same task; which is often efficient for data locality), all grouping (the messages are duplicated to each task), global grouping (all the messages are sent to a single task), local grouping (when possible, the message is sent to an executor running on the same machine), and others. This makes Storm very extensible for different workloads and types of applications. Storm implements "at most once" or "at least once" semantics: the use of *ackers*, tasks acknowledging received messages to their producers, will make Storm resend data in case of a problem. Since this is optional, it falls back to "at most once" when not used.

Some optimizations have been studied in Storm, for example new schedulers taking decisions on operators placement [26]. It is also the engine we chose for our data locality algorithm described in Chapter 4.

Other stream processing engines exist, but have not been studied extensively in the frame of this thesis. They include Twitter Heron [79], MillWheel [21], S4 [94], Apache Samza [95], Apache Flink [46], Photon [24] and others.

## 2.3 Components and blocks programming

We mix in this section some background about component models, flow-based programming, and blocks programming. They are different concepts, but they share some design ideas in the way units are composed together.

Components and flow-based systems are often used to implement ETL (extract-transform-load) pipelines, which is one of the purposes of $\lambda$-blocks, presented in Chapter 5.

### 2.3.1 Component-based software engineering

Component models and the practice of component-based software engineering take their roots in different software development paradigms, most notably object-oriented programming. The goal of component-based architectures is to design and implement software such that it is easy to maintain, evolve, and adapt; while at the same time leveraging code reuse. To achieve this, it primarily focuses on the separation of concerns, by splitting software modules into components communicating with each other. Properties of components often include:

- **Components have an interface**. The interface describes how to interact with the component, through its inputs and outputs. It doesn't specify how the functions are implemented, de facto making the component a *black box*. This is the principle behind encapsulation: there is no need to know the inner details of a component in order to use it.

- **Components are replaceable**. A component can substitute another one if it provides the same interface (or at least a superset of it) and has the same functionality. This enables developers to easily replace a part of the system without altering the other parts. This can be done at compile time and even at run time: the choice of a component over the other can be influenced by some conditions like the current state of the program.

- **Components are meant to be used in different contexts.** Through its interface, a component must have been designed to be used in different systems, not restricted and only useful to one.

Sometimes, a components framework allows a component to be composed of a set of other components, further leveraging code reuse and allowing a complex function to be broken down in smaller pieces.

Some well recognized component-based frameworks include Enterprise JavaBeans [65], Corba [114] (Common Object Request Broker Architecture), OSGi [23], Fractal [40], and many others.

Components, when linked together in a software architecture, can easily be represented with a graph, which opens possibilities for describing such an architecture with a Domain Specific Language (DSL).

### 2.3.2 Flow-based programming

We focus in this section on dataflow programs meant to perform data transformation. Flow-based programming is a particular type of component-based architectures, where the components are black boxes transforming their input and forwarding their output to their subscribers. Such a program is often represented as a directed acyclic graph (DAG), where the nodes without any vertice pointing to them represent the program inputs, and the nodes without any vertice departing from them represent the outputs. In such a graph, data flow from one node to the other, uni-directionally, and is transformed at every step. Unix pipes implement such a system, with the restriction that the graph is a list (a node can't have two subscribers).

The Orange framework [57] takes this approach for building machine-learning pipelines. In Orange, one defines a DAG to describe the steps to perform to read data, clean it, feed it to machine-learning algorithms, and compare and visualize results. Using a flow-based approach helps users to interactively see how the data is transformed (which also helps for debugging), while keeping the code base more easily maintainable, and maximizing the reuse of components.

StreamPipes [101] is another framework for implementing flow-based programs through a graphical interface. Its particularity lies in how the data is stored, through an RDF ontology. It is meant to wrap existing data processing frameworks such as Apache Storm and Apache Flink, and is able to serialize all the messages between operators with different messaging protocols, including Apache Kafka and MQTT.

Apache Beam [1] is an implementation of the Dataflow Model [22] published by Google. It can seamlessly work with batch- and stream-processing, by defining pipelines of data-transformers in a directed acyclic graph. A particularity of Beam is its ability to distribute jobs to different engines, for example Google Cloud Dataflow, Apache Flink or Apache Spark, making it engine-agnostic. It implements user libraries in Java and Python, making a heavy use of operator overloading to define pipelines with a convenient syntax.

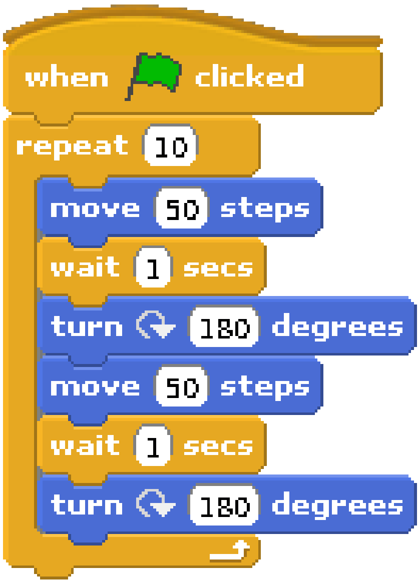
### 2.3.3 Programming with blocks

Programming with visual blocks of code is a paradigm proposed to teach computer programming to beginners. Frameworks propose a graphical interface, which allows to write programs by composing them with blocks, which are meant to wrap code instructions.

Figure 2.1 shows an excerpt of the MIT Scratch [100] graphical interface. The program consists of a `repeat` block, containing smaller blocks giving motion instructions to a sprite. Scratch helps users by hiding the complexities of the inner blocks, allowing only its parameters to be changed, and provides immediate visual feedback: for example it is not possible to construct actions that are semantically wrong, such as inserting a `repeat` block in the condition of an `if` block.

Other similar software exist, for example Blockly [60] and Snap [67].

This paradigm presents a lot of similarities with component-based approaches. A program is written by composing and embedding together different independent components, which are black boxes, are meant to be reused in different programs, can be parameterized, and can be composed of other components.



```
when 🟩 clicked
repeat 10
    move 50 steps
    wait 1 secs
    turn ↻ 180 degrees
    move 50 steps
    wait 1 secs
    turn ↻ 180 degrees
```

Figure 2.1 – MIT Scratch.

Chapter 3

# Online metrics prediction in monitoring systems

*« A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. »*

Leslie Lamport [81]

## Contents

Our first contribution concerns the design of a scalable system, with a direct application related to an industrial problem: handling a high numbers of monitoring metrics, and using them for prediction purposes. Since an entire end-to-end system is developed and presented, this contribution is positioned at different layers of the data processing ecosystem.

Monitoring thousands of machines and services across multiple datacenters produces a lot of time series points, giving a general idea of the health of a cluster. However, there is a lack of tools to further exploit this data, for instance for prediction purposes. We propose to apply linear regression algorithms to predict the future behavior of monitored systems and anticipate downtimes, giving system administrators the information they need ahead of the problems arising. This problem is quite challenging when dealing with a high number of monitoring metrics, given our three main constraints: a low number of false positives (thus blacklisting unpredictable metrics), a high availability (due to the nature of monitoring systems), and a good scalability. We implemented and evaluated such a system using production metrics from Coservit, a French company specialized in IT monitoring.

The results we obtained are promising: sub-second latency per metric per CPU core, for the entire end-to-end process. This latency is constant when scaling the system to 125 cores on 4 machines, and the performances don't decrease with time: during 15 minutes, it is able to predict more than 100 000 monitoring metrics.

## 3.1 Introduction

Monitoring machines ensures a system is running correctly, and triggers human intervention as soon as it is needed. This is especially relevant for user-facing systems, and particularly when downtimes can lead to serious problems, e.g. with hospitals and transportation systems. A lot of monitoring tools exist nowadays, ranging from in-house scripts running various sanity commands to ensure processes are working correctly, to complete suites, distributed on many servers and constantly checking and aggregating thousands of metrics. Apart from the simple "check the existence of a process" scripts, monitoring software usually collect metrics (e.g. CPU load, used memory), check them against defined thresholds (e.g. 80% of the maximum), raise alerts to system administrators when these thresholds are exceeded, and generate reports about resource consumption and error rates.

There are different kinds of collected metrics: system-related (CPU load, remaining empty space on hard drives, network speed, etc.), services-related (database uptime, web server open connections, memory used by processes, etc.), or statuses (up, down, unknown). These metrics are timestamped, and hence are collected as time series.

In order to enhance the current set of practices, we propose to detect failures before they emerge, reducing human intervention and letting administrators plan their solutions.

Coservit [4], a French company providing a "monitoring as a service" infrastructure, has collected and stored about one million different metrics for more than 5 years, across dozens of thousands of physical hosts. We used this dataset to experiment different approaches, aiming to predict the future behavior of these metrics.

The goal of our system is to make predictions about the health of online services in the near future, in order for system administrators to perform preventive maintenance, instead of reacting to problems and fixing them when they occur. This is why we focused on a short time horizon, typically between 1 and 5 hours. For that purpose, we applied linear regressions on the different collected metrics, using historical data to train our system. We found linear regression to be the best fit for this kind of data and this horizon, thanks to its ability to identify local trends.

We added the constraint of creating a scalable system that is not limited to a maximum number of metrics, or a single host. It must also be as CPU-efficient as possible, and for instance not waste resources learning metrics which are too difficult to predict due to their volatile aspect.

The challenges we had to overcome to develop this system were the huge amount of metrics collected by the monitoring engines, the processing time per metric which had to be fast enough to be useful, and the avoidance of false positives, to prevent raising alerts when it is not needed. Our contributions are a scalable system architecture, and the feedback raised after extensively testing it with industrial production data.

We found that more than 50% of the observed metrics are suitable for linear regression prediction; as detailed in Section 3.3.4 the other ones are too volatile to be accurately predicted with this method. The entire process of retrieving measurements, building the learning parameters, predicting the values, and storing the results takes about one second per metric. Moreover, it scales linearly up to all the CPU cores we had at our disposal for evaluating this system.

The rest of this chapter is organized as follows: we first give some background about the problem and the tools involved, before describing our system in details. We then evaluate our results, and present the related work before concluding and giving insights about future ideas.

## 3.2 Background

### 3.2.1 Monitoring

Monitoring, in its simplest form, consists of regularly checking the health of a system or system component, to ensure it is meeting expectations. Most software suites providing monitoring facilities offer convenience features: formal description of monitored hosts and services, data polling or pushing, alerts and alert escalations, aggregated reports, and graphical dashboards. We dive in this section into some interesting properties, describing how metrics are collected and how alert thresholds are defined.

Many monitoring systems such as Nagios [8], Zabbix [19] or Shinken [18] allow system administrators to easily write scripts whose return value and output can be passed to the monitoring engine, provided their standard output conforms to the expected format. One script can return many metrics, and it is usually run regularly: every few minutes or hours, depending on the importance of the component and the relevance of the check. Using the standard output as the message passing channel provides the ability to write scripts with any language using any library, as long as they can write to standard output. The monitoring engine gets the return value, parses the key/value pairs of metrics, stores them along with a timestamp, and can raise alerts if needed. A simple example of a RAM check output is as follows:

```
RAM OK;``|''RAM=1.4GB;3;3.9;0;4
```

A return value of 0 means the state of the service is *OK*, 1 is for *WARNING*, 2 is for *CRITICAL*, and 3 is for *UNKNOWN*. After this run, we get a message, a value and its unit (1.4 GB), the warning and critical thresholds (3 and 3.9), and the minimum and maximum possible values (0 and 4). There's only one collected metric in this example output, but the format allows for more to be concatenated at the end.

Using these simple conventions, anything can be monitored, and the collected metrics can be centralized for further processing.

The warning (resp. critical) threshold, if exceeded, will turn the monitored component into a warning (resp. critical) state. This is used by the monitoring engine to send alerts to the persons in charge, and to display warnings in dashboards.

Two types of alerts exist: reactive and proactive. With the reactive alerts, systems administrators receive notifications that a failure has occurred, for example a disk has crashed. They can then proceed to fix the issue, but downtimes can arise which may have an impact on end users. With the proactive alerts, the notification is sent before the failure occurs, because a threshold was exceeded, for example a disk reached 90% of its capacity. This usually gives administrators some time to get the system back to a sane state.

### 3.2.2 Time series prediction

A time series is a variable evolving over time. Stock market prices, average temperature of a city, consumption of a product and usage of a service are some examples of time series. Time series forecasting is predicting future observations of a variable. As seen in Section 3.2.1, we consider discrete time series, ordered in time, whose values are metrics collected from monitored services. Time series prediction/forecasting has been studied for many years and in many different domains, such as stock market analysis [20] and weather prediction [50]. Over the past few years different machine learning methods have been used to predict time series. Generally, due to the complexity of the problem, in many cases complex models and algorithms have been used to forecast the time series. For instance, many studies, such as [63] and [51], use recurrent neural networks (RNNs) which are basically neural networks adapted for time-dependent data.

Although these models can consider different factors, such as the dependency between data points, and learn based on them via backpropagation through time (BPTT) [116], practically they could be inefficient in some cases. For instance, in the context of a datacenter, where highly dynamic metrics from many devices are considered, the aforementioned methods may not be good candidates due to time and space complexity. In such cases, one may consider simpler methods such as regression approaches. Maybe the most straightforward technique in this domain is the linear regression [66]. Though very simple, the regression techniques are still very powerful and efficient [85], particularly when the data and its intuition (such as critical thresholds on metrics) change very rapidly, they can adapt the new settings of the system very efficiently, due to the simplicity of the model.

## 3.3 System description

### 3.3.1 Architecture

We base our architecture around a Cassandra database, used for storing monitored and predicted values, as well as prediction error rates.

Cassandra [80] is a distributed database with no single point of failure, where all the nodes can answer queries. It provides data replication (including among different datacenters), is fault-tolerant, and is highly scalable. Cassandra is a column-oriented database, and leverages denormalization: since it doesn't support join queries, it is necessary to duplicate some fields into different tables, in order to read them faster.

Figure 3.1 shows the main components of our architecture. We consider the monitoring agents as black boxes, geographically scattered in different datacenters, reporting metrics about the systems they monitor to a monitoring broker. All the metrics are stored in Cassandra for further processing.

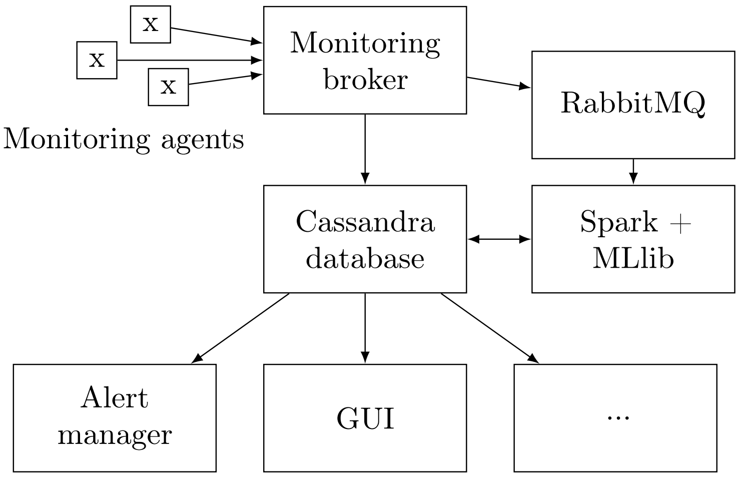


Figure 3.1 – System architecture.

Spark workers find the metric identifiers ready to be processed in a RabbitMQ message queue, read all the data points associated to them from Cassandra, and run prediction algorithms with MLlib (described in Section 3.3.3). The message queue serves as the producer/consumer middleware, and is useful to regulate the flow of predicted metrics and which metrics need to be processed. Once predictions are calculated, they are stored back into Cassandra.

Finally, different end-user applications use the predicted metrics, most notably a web-based graphical interface alerting users about future problems.

This architecture resembles a lambda-architecture [84]: it leverages historical and real-time data to provide up-to-date predictions combining both sources.

### 3.3.2 Data model

We use a Cassandra cluster to store all the collected metrics, as well as the different computed predictions on them. The relevant tables are:

- *metrics*: stores the metric names (e.g. `open_sockets`, `disk_available`).
- *metric_measurements*: each row represents a measurement, and consists of a metric id and a value, as well as a timestamp, a unit, the warning and critical thresholds, and if relevant the minimum and maximum possible values.
- *metric_predictions*: each row stores two predicted points (a point is a value and its timestamp). The first point represents the metric in the

near future (a few seconds ahead), while the second point is the metric in a more distant future, determined by the prediction horizon. It is enough to store two points for a linear regression.

- *metric_ errors*: each row stores the root-mean-square error (RMSE) calculated in the blacklisting process. This permits to later filter out metrics whose measured values don't align with the predictions.

The metrics stored in these Cassandra tables are flattened: they don't represent the hierarchy of hosts, services and metrics found in some monitoring engines. This is on purpose: Cassandra is not used to store this kind of information, which is better managed by specialized tools. In fact, this schema is monitoring engine-agnostic: it is meant to be used by different engines as a sink for their metrics.

### 3.3.3 Linear regression

We first discuss in this section why linear regression makes sense compared to the legacy threshold methods, and then describe how we applied it on our metrics.

#### Comparison with threshold

Most metrics have two thresholds: warning, and critical. A system in the warning state continues to properly run, but it is a signal that some components might break in the future. For instance a CPU whose load is 80% is still working, but approaching its maximum capacity. Most systems will emit a notification when a metric is above its warning threshold, but it doesn't mean the trend will continue and the metric will enter its critical state. In our example, the CPU load can keep increasing, or decrease and leave the warning state.

Looking at the evolution of data points among different kinds of metrics, we identified 6 common scenarios, represented in Figure 3.2. Two situations lead to the *perplexity point*, the point in time when metrics get closer to the warning threshold, and from where three situations can arise. We evaluate the benefits of linear regression versus the warning threshold for each of them:

- *Slow rise* followed by *slow rise*: linear regression is a perfect fit for this situation, as it will easily identify the trend even before the warning zone is reached.

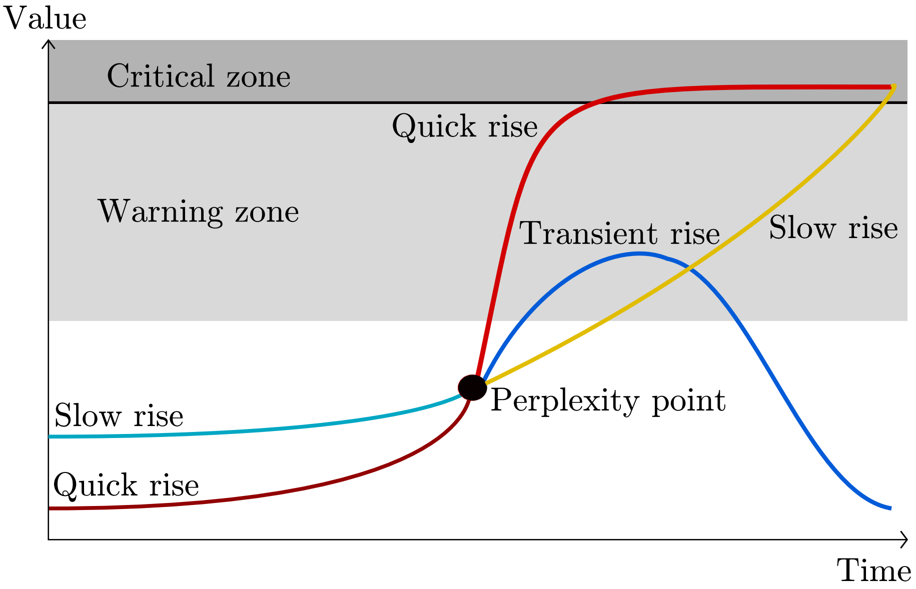


Figure 3.2 – Metric trend cases.

- *Slow rise* followed by *quick rise*: both linear regression and the threshold system will be efficient if they refresh their measurements often enough. If the rise is too fast, they will both predict the issue too late.

- *Slow rise* followed by *transient rise*: if linear regression can predict the future decrease, it will avoid sending a false positive alert.

- *Quick rise* followed by *slow rise*: both systems might give a false positive, or at least predict the issue too early. However, linear regression will be better at predicting the change to normal state again.

- *Quick rise* followed by *quick rise*: depending of the frequency of the measurements, the threshold system might alert too late about a fast arising problem, whereas linear regression can predict it.

- *Quick rise* followed by *transient rise*: again, false positives might be sent by both systems, but linear regression is better at anticipating the return to normal state.

We saw that linear regression is better than the legacy threshold system in 4 out of 6 scenarios, and better or equal in the other 2. That makes it a good candidate for predicting the behavior of many metrics.

### Linear regression on monitoring metrics

We use the machine learning library bundled with Spark, MLlib, to perform a linear regression on data distributed on different machines. The training phase is performed independently for each metric, using a history size of 30 points. We found this value to be optimal: it gives good results (described in more details in Section 3.4) while keeping the training phase to a reasonable time (100 ms). Once performed, we store the prediction results back into Cassandra in the $metric_{predictions}$ table, available for consumption by different front-ends, and by the error evaluation process (Section 3.3.4).

We didn't explore in this work the potential correlations between different metrics. We observed it often happens that multiple errors are reported from the same machine in case of a failure (if it runs multiple monitored services), but this is less obvious when predicting problems in advance. Nonetheless, we keep this exploration of metrics correlation as future work.

### Prediction horizon

The prediction horizon (the estimation of how long the predictions are valid) is complex; as it is generally less and less correct over time. However, we noticed that a maximum horizon of 8 hours is a good metric, and this is the amount needed for reliability engineers to not wake up overnight to fix services. The predictions are given as "best effort": they represent the best values obtained by the system, but can't give guarantees about their veracity. It is important to note they are continuously recomputed, and hence never out-of-date.

## 3.3.4 Metrics selection

Not all metrics are good candidates for prediction. Some metrics don't show any pattern, and never respect their predicted values. We use a blacklisting algorithm to eliminate them, in order to save computing resources and avoid false positives. A weekly batch script performs an error evaluation (RMSE, Root Mean Square Error) of the predicted values, which are compared against the observed values for the week. If the RMSE is higher than a given threshold, the metric is blacklisted. It is of course possible for users to reactivate blacklisted metrics to see if their predictions perform better.

## 3.3.5 Optimizations

This section describes some interesting optimizations that helped us greatly reduce the time and resources needed for the various computations.

### Caching

Spark implements a persistence mechanism, which allows to cache data either on memory, or on disk, or both. By analyzing the data processing chain, one can spot the states where saving data brings computation time benefits, typically when one Dataframe is to be used by many other functions. As sometimes persistence can reduce performances, when the cost of caching data is greater than the benefits it provides, comparing the performances both with and without persistence is rather necessary. In our case, we found that persisting the measurements data for a metric after it was retrieved from Cassandra greatly reduced the processing time.

### Dataframes

Spark provides different APIs to developers, the main one being Resilient Distributed Dataset (RDD). RDDs are suitable to apply transformations on large, unstructured datasets. On the other hand, Dataframes, another container for distributed data, relies on tabular data organized into columns and associated to a schema, like an SQL table. For relatively complex queries, using Dataframes brings consequent speed-ups [17], because they benefit from advanced optimizations such as a query planner (named the Catalyst Optimizer). Since our data is already organized into tables (when it is stored in Cassandra), we could compare both approaches, RDDs and Dataframes, and the latter were the fastest.

### Cassandra optimizations

Apache Cassandra being a distributed, column-oriented database, the key to obtain good performances is to design a schema around the expected types of queries it will get, at the price of duplicating data. Cassandra is optimized for writes, and in order to get good performances for reads, it is necessary to well partition the data. Good practices recommend two goals for that purpose: balance data evenly across machines, and minimize the number of partitions that need to be accessed for one query. It is recommended for a partition to be a few hundreds of megabytes, and contain a few hundreds of thousands of values. Our biggest table containing the measurements as reported by the monitoring engine, and every unique monitored service having up to a few metrics updated at most every minute, we decided to keep monthly partitions per service, which would weigh around 60 MB. It is to be noted that yearly partitions would be sustainable by Cassandra as well.

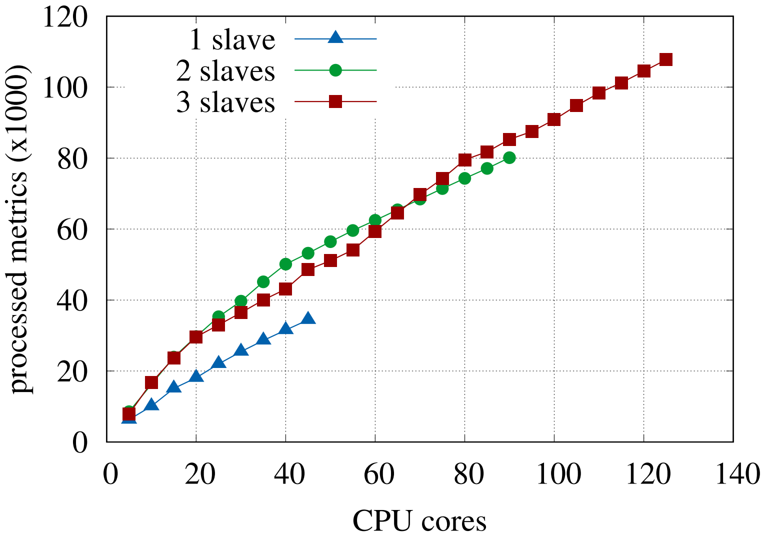


Figure 3.3 – Number of metrics handled in 15 minutes, varying the number of slaves and the number of CPU cores.

# 3.4 Evaluation

## 3.4.1 Setup

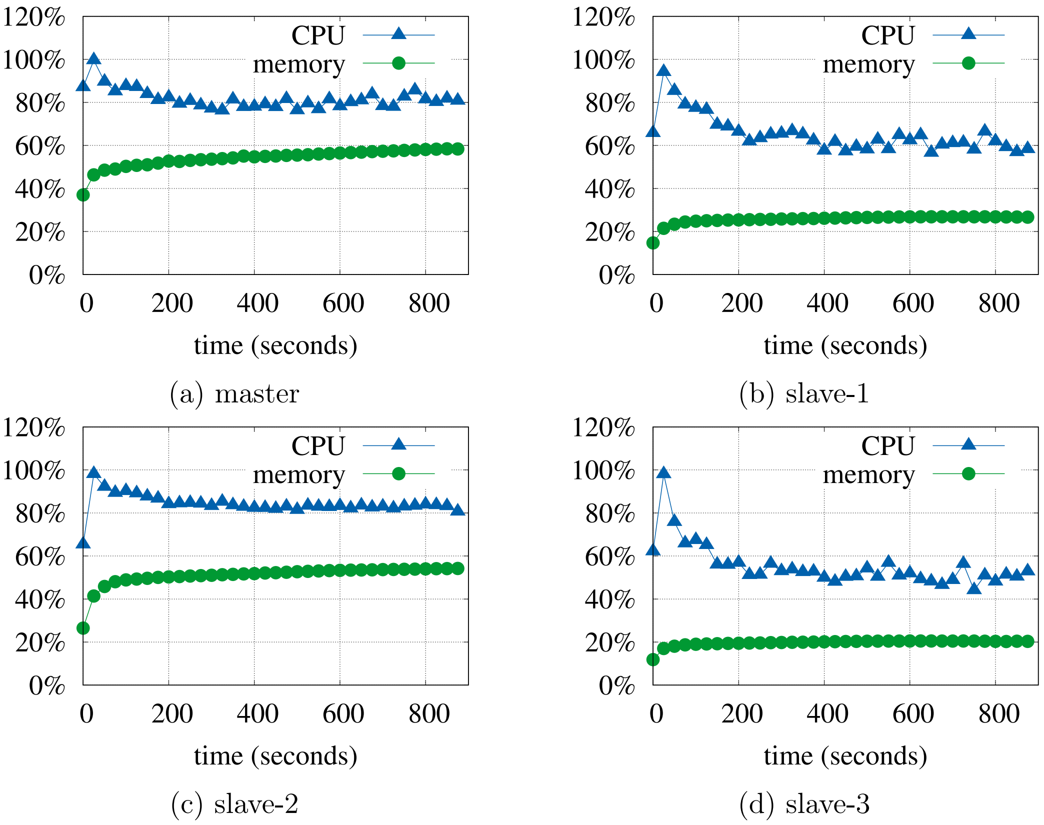
We ran the experiments on 4 physical machines (HPE Proliant DL380), with 16–28 hyper-threaded cores, and 128–256 GB of memory. We installed Debian stretch 9.0, Spark 2.1.0, and Cassandra 3.0.9.

One machine is the master and the three others are slaves. We replayed the production load triggered by the monitoring boxes reporting metrics, and measured different parameters under different conditions.

We replay a dataset made of production data recorded on Coservit's servers. It represents two weeks of data, for 424 206 unique metrics. In total, there are 1 500 335 458 data points, whose size is about 15 GB in total. To get all these data points, 25 070 machines were monitored. An interesting deduction we can make is that, on average, there are about 17 monitored metrics per machine (the standard deviation is about 30 though, so it highly depends on the type of machine).

## 3.4.2 Scaling

To check the system scaling performances, we measured the amount of metrics which can be processed in a 15 minute time range, varying the amount of slaves between 1 and 3, and the number of CPU cores between 5 and 125.



(a) master

(b) slave-1

(c) slave-2

(d) slave-3

Figure 3.4 – CPU load and memory consumption, when running on 100 cores for 15 minutes.

Figure 3.3 shows the results. When using only 5 cores, the system could work on about 7000 metrics (± 1000, depending on the number of slaves) in the given time range. This value scales linearly with the number of CPU cores, for the different amount of slave machines we tested; that's because all metrics are independent from each other and Spark manages this kind of setup very well. The maximum performance obtained is when using the 125 CPU cores at our disposal, and corresponds to about 108 000 predicted metrics in 15 minutes, or 120 per second. In conclusion, one metric takes about one second to be predicted on one CPU core, end-to-end on the processing chain.

Figure 3.4 shows the CPU load and the memory used when running the experiment with 100 cores. The machines are not overloaded, which leaves room for other high-consumption processes such as Cassandra. Two things are worth noting: the CPU load reaches 100% at the beginning, which is due to the Spark jobs start-up. Afterwards, both CPU and memory stay very stable: this is expected given the same work is done for every metric.

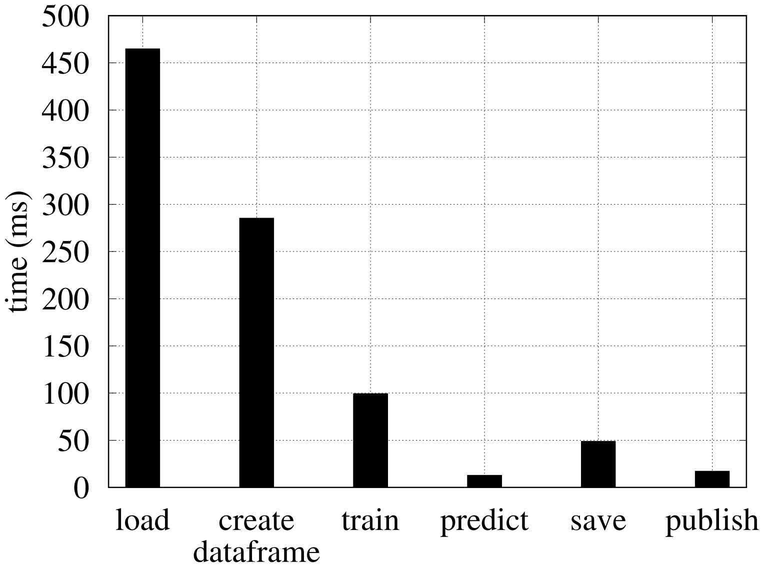


Figure 3.5 – Time repartition of the end-to-end process for predicting a metric (average with 90K metrics).

### 3.4.3 Time repartition

We also instrumented the different components of the processing chain, measuring the time taken by each of them, and averaging it on all the metrics. Figure 3.5 shows the results. Loading the data from Cassandra is what takes most of the time (about half of the total); this is expected since Cassandra is optimized for writes. Moreover, the network adds up to the latency. Creating a Spark Dataframe is quite resource-consuming too, but once it's created it's fast to work on it: the training and prediction times are relatively short. Finally, saving the data back into Cassandra and publishing an acknowledgement message to RabbitMQ is fast. End-to-end, the processing time of one metric on one CPU core is about one second. This is an acceptable time given our requirements, and is way below the prediction horizon (a few hours).

### 3.4.4 Load handling

Using these previous results:

- It takes 1 second to predict a metric (end-to-end);
- There are on average 17 monitored services per machine (in the case of our dataset, a machine is either physical or virtual);

and taking a pessimist average of 1 minute for the metric sampling period (1 minute is usually the minimum period, and most metrics don't gain to be checked that often), we deduce we can handle $60 \times 24 = 1440$ metrics on a 24-core server. That means such a server can handle all the predictions for about 85 machines, which is a very acceptable ratio. Since this system scales linearly, increasing the number of cores will automatically increase the metrics load a server can handle.

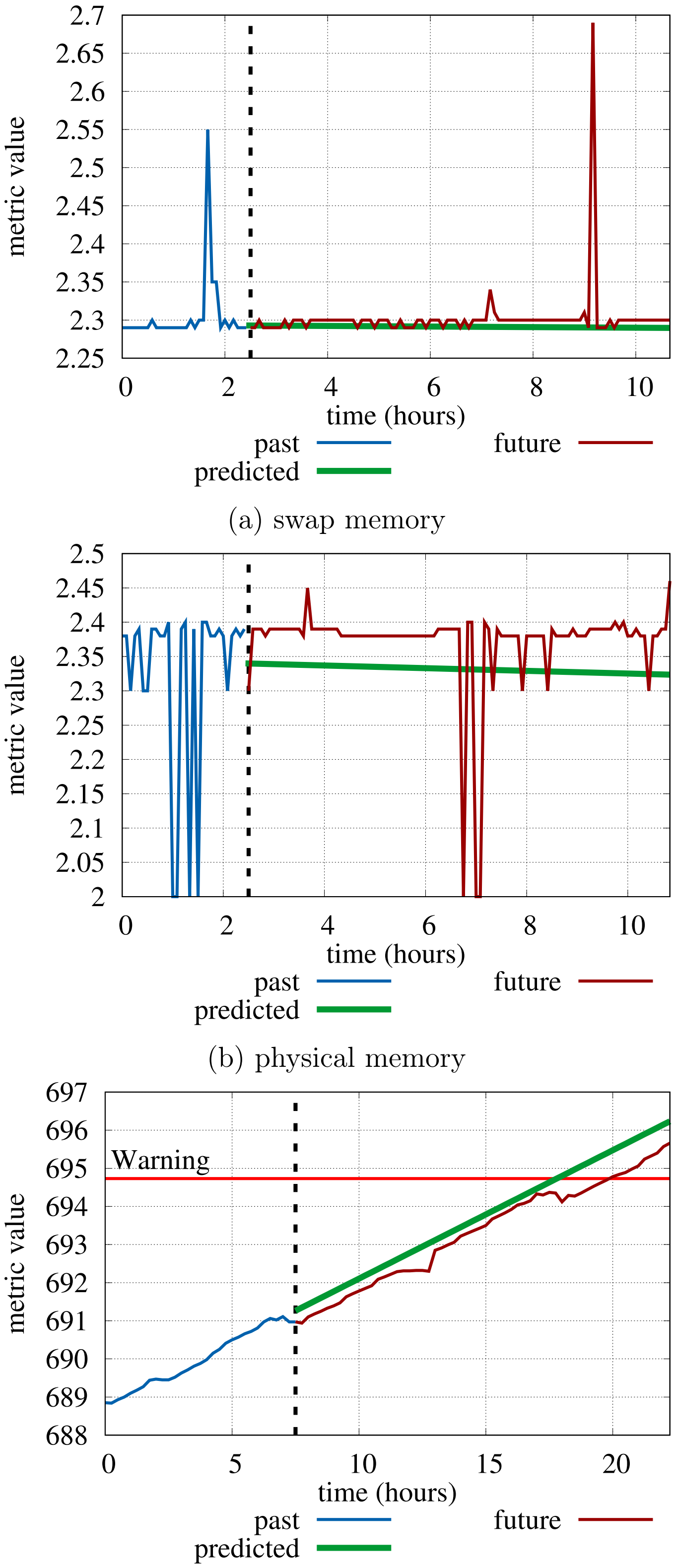
It is important to note that the described servers have monitoring storage and prediction as their only role: they do not run other monitoring software, user interface dashboards, etc. If a 24-core server seems a lot to handle 85 machines (or 1440 metrics), it appears the performances are better in practice: the period is higher than 1 minute for most metrics, and the black-listing process will reduce the load in any case for less predictable metrics.

### 3.4.5 Predictions accuracy

Finally, we measured the root-mean-square error (RMSE) for every predicted metric. Due to outliers and "bad" metrics (which are too volatile to respond well to linear regression), it is quite high: its average is 821.65, with a standard deviation of 23 686.72. However its median is at 0.000 842: this tells us most of the metrics have a very low RMSE. If we decide a good prediction has an $RMSE < 0.02$, when we filter the results to keep only those below this value, we get an average at 0.001 154, and a standard deviation at 0.003 403. More interestingly, 58.5% of the metrics fall within that range, which is promising for the benefits of linear regression over this system.

Figure 3.6 shows three examples of metrics measurements and predictions. A vertical dotted line separates the training values from both the predicted ones and what was actually measured. The first one is the swap memory of a machine, with up-spikes probably due to the kernel swapping memory before freeing it immediately afterwards. The second one represents the physical memory of a machine, with down-spikes. Linear regression can't predict spikes, and that is something we intend to try with other, more complex, machine learning algorithms. Note that the y-scale doesn't begin at 0, hence the spikes are smaller than what they appear. In these cases, not predicting spikes is a feature, as they are sudden increases (or decreases) that are back to normal almost immediately afterwards; hence we avoid raising false positive alerts.

The third figure represents how full a disk partition is. This time an actual problem is detected: the warning threshold is reached. It is predicted a bit sooner than the actual problem occurrence, but this difference would have trimmed down when getting new data.



(a) swap memory

(b) physical memory

(c) disk partition

Figure 3.6 – Measurements and predictions for three different metrics.

## 3.5 Related work

### 3.5.1 Time series

Time series forecasting is essential in many fields such as finance [61], health [62], weather forecast [43] and marketing [35]. Auto regressive (AR) and moving average (MA) are two of the very first linear statistical approaches of time series forecasting [49]. Auto Regressive Integrated Moving Average (ARIMA) is a linear forecasting model, which includes both AR and MA while considering trends in the time series [38]. With advances in probabilistic machine learning, many studies utilize machine learning algorithms for time series forecasting [37] along with statistical approaches [55]. Support vector machines generalize well in high dimension [112]. With regression extension, many studies used support vector regression for time series prediction [91]. Random Forest is an ensemble of weak learners [39] which results in good generalization. They are used in many time series prediction applications [52].

### 3.5.2 Monitoring

Some industrial companies have implemented prediction systems to prevent failures. Zabbix [16] uses different models (linear regression, polynomial regression, etc.) to predict the future value of a given metric, and hence predict when a critical threshold will be reached. Triggers compute the predicted values each time a new metric value arrives. However, the choice of the model and the parameters tuning has to be done manually for each metric, which is resource-consuming and easily leads to errors. In our system, we use cross-validation to automatically tune the parameters and we use blacklisting to remove metrics that are not good candidates for linear regression algorithms.

Chalermarrewong et al. use time series analysis approach for hardware failure prediction task [48]. They use self adjusting multi-step ARMA to predict the future values which updates the model according to paired t-test.

In the capacity planning domain, Microsoft Azure [7] defines a set of machine learning algorithms to compute predicted values. They offer a graphical designer to easily define a web service that performs the learning. However it does not perform predictions in real time.

Thermocast [83] focuses on predicting the thermal parameters of datacenters. Their approach is similar, but it solves a different issue and doesn't look at server problems but rather the temperature of the various equipment.

Many systems (e.g. [82]) leverage elastic computing to predict Service-Level Agreement (SLA) issues and provision enough resources for anticipated workloads. Our solution is orthogonal to this problem, since some monitoring issues can be consequences of changes in the datacenter workload, but not all of them.

Singh et al. describe an architecture for storing monitoring data and update a wiki engine with the collected information [105]. They present some similarities with our work in their choice of distributed engines, notably Apache Spark and MLlib.

## 3.6 Conclusion

Monitoring machines in a datacenter helps determining which services are down, and which resources need to be upgraded. It also generates a lot of time series points: thousands of metrics, as diverse as a CPU load or a database latency, constitute a set of metrics which can be leveraged for prediction purposes. However, predicting the future behavior of monitoring metrics poses a few challenges, the noteworthy ones being *accuracy* (to avoid generating false positive alerts), and *scalability* (as more metrics enter the system, it needs to stay fast).

We described and evaluated a system for leveraging this opportunity, choosing linear regression as the main prediction algorithm, for its simplicity and relevance for most of the observed metrics. We detailed the inner workings of linear regression, as opposed to the threshold system in place within most monitoring software suites; as well as the main components revolving around it: persistence storage in a Cassandra database, work distribution among servers with Spark, and blacklisting of less predictable metrics. The detailed evaluation gave us great insights on this system, notably about its scalability (it scales linearly up to at least 125 cores) and speed (the end-to-end processing chain takes about one second on one CPU core to predict one metric). We believe this system adds a great value to monitoring tools, by giving system administrators information about potential future problems and downtimes, and it allows to plan resources more efficiently.

There are many improvements we can implement as future work. We plan to experiment with deep learning, in order to focus on long term global trends, rather than local ones identified by linear regression. For blacklisted metrics which are not good fits for linear regression, we can implement other machine learning algorithms, and compare the performances obtained. Lastly, we envisage to plug this system to a ticketing mechanism used by clients to report problems, in order to match them with their host errors. This presents a few challenges, because tickets are not labeled and natural language processing will be required; but we're confident it can lead to innovative customer support solutions, such as the automatic deployment of operations in response to tickets reported, as soon as they are matched with the issues raised by the monitoring system.

Chapter 4

# Data-aware routing

*« feature, n: a documented bug / bug, n: an undocumented feature »*
Mario S F Ferreira <lioux@FreeBSD.org>

## Contents

After presenting an end-to-end system for the Smart Support Center project, we focus for our second contribution on a specific layer of the data processing stack: the routing of real-time messages, and how it can be optimized by looking for correlations in data.

Distributed stream processing engines continuously execute series of operators on data streams. Horizontal scaling is achieved by deploying multiple instances of each operator in order to process data tuples in parallel. As the application is distributed on an increasingly high number of servers, the likelihood that the stream is sent to a different server for each operator increases. This is particularly important in the case of stateful applications that rely on keys to deterministically route messages to a specific instance of an operator.

Since network is a bottleneck for many stream applications, this behavior significantly degrades their performance.

Our objective is to improve stream locality for stateful stream processing applications. We propose to analyse traces of the application to uncover correlations between the keys used in successive routing operations. By assigning correlated keys to instances hosted on the same server, we significantly reduce network consumption and increase performance while preserving load balance. Furthermore, this approach is executed online, so that the assignment can automatically adapt to changes in the characteristics of the data. Data migration is handled seamlessly with each routing configuration update.

We implemented and evaluated our protocol using Apache Storm, with a real workload consisting of geo-tagged Flickr pictures as well as Twitter publications. Our results show a significant improvement in throughput.

## 4.1 Introduction

Stream processing engines such as Apache Storm [111] and Apache Spark Streaming [120] have become extremely popular for processing huge volumes of data with low latency. Streams of data are produced continuously in a variety of context, such as IoT applications, software logs, and human activities. Performing an online analysis of these data streams provides real-time insights about data. For instance, the Twitter infrastructure processes up to 150,000 tweets per second, and maintains a list of trending hashtags. This cannot be done using batch jobs, as the processing delay would make the result irrelevant by the time they are produced. Stream processing solves this problem by continuously analysing the new tweets in memory, and updating results within milliseconds.

A stream application consists of a directed acyclic graph in which vertices are operators, and edges represent data streams between operators. To scale horizontally, each operator is deployed as several instances distributed over multiple servers. Stateful applications maintain statistics on different topics, and use *fields grouping* to ensure that data tuples related to a given *key* are always sent to the same instance of an operator. This assignment usually relies on hash functions to obtain a random but deterministic routing. As the application is deployed on a higher number of servers, the likelihood that the recipient operator instance is located on a different server increases. This increases the network consumption of the application, and significantly degrades its performance.

In this work, we propose to build optimized routing tables to improve network locality in streaming applications. We analyse the correlation between keys used in successive fields grouping, and explicitly map correlated keys to operator instances executed on the same server. Hence, data tuples containing these keys can be passed from one operator to the next through an address in memory, instead of copying the data over the network. This has the advantage of being faster, and avoids the saturation of the network infrastructure. Hash functions suffer from skews in data distribution, as the most frequent keys cause load imbalance on operator instances responsible for processing them. As we collect statistics on the distribution of keys, routing tables can also be used to ensure that the load remains even, which further improves the throughput of the application.

Data streams fluctuate over time, and association between keys can vary significantly. Consequently, we opt for an online approach to optimize routing and reconfigure key routing without disrupting the application. To this end, we add an instrumentation tool to stateful operators that gather statistics on the frequency of key pairs. A *coordinator* collects these statistics from all operators, and executes a graph partitioning algorithm to divide keys between servers while balancing the load. Finally, an online data migration protocol ensures that the state of reassigned keys is migrated between operator instances without data loss. We evaluate our approach on Apache Storm. Our workloads consist of synthetic datasets with varying degrees of key correlations, and two real datasets from Twitter and Flickr. The results show a significant improvement in throughput.

The rest of this chapter is organized as as follows: Section 4.2 provides background information on streaming applications. Section 4.3 describes our approach for optimizing data routing. We present the evaluation in Section 4.4. We review the related work in Section 4.5, and conclude in Section 4.6.

## 4.2 Background

We present in this section some general stream processing concepts, before focusing on stream routing between operators.

### 4.2.1 Stream processing

#### Application model

As introduced in Section 2.2.4, stream processing was developed to continuously execute operators on potentially unbounded streams of data tuples. Apache Storm [111], Flink [46], S4 [94], Samza [95], and Twitter Heron [79], are examples of popular stream processing engines. Following the dataflow programming paradigm, a stream processing application can be described

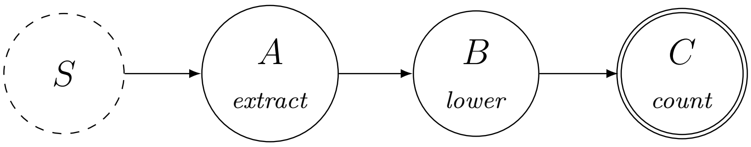


Figure 4.1 – A simple wordcount stream application. $S$ sends sentences, operator $A$ extract words, $B$ converts them to lowercase, and $C$ counts the frequency of each word.

as a Directed Acyclic Graph (DAG). Vertices represent processing operators (POs) that consume and produce data streams, which flow along edges. A source constitutes the entry point of the DAG, and streams data tuples, such as posted tweets or uploaded pictures, to POs. Stream processing implements a share-nothing policy, so operators can be executed in parallel as they manipulate different tuples of data. Figure 4.1 represents a simple wordcount application for streams of sentences. Data enters the DAG at the first PO $A$ and flows from the left to the right. $A$ takes as input sentences and extracts words, thus producing a stream of words. These words then reach $B$ which, for each input word, writes the same word in lower-case format to its output stream. Finally, $C$ counts the frequency of each word in its input stream. POs $A$ and $B$ are stateless, as they do not update any internal state when processing data, while $C$ is stateful as it maintains frequency counts (stateful POs and POIs are represented with double circles in the remainder of the chapter). In a general case, each PO can input and output multiple streams of data. In this chapter, for the sake of simplicity, we present applications consisting of chains of POs, in which each PO has a single input and output stream. The results presented remain, however, valid for more complex DAGs. We also focus on streaming frameworks which implement long-running tasks (such as Apache Storm), as opposed to the micro-batch processing model (as in Apache Spark).

### Scalability

This approach to stream processing scales horizontally by placing different POs on different machines. Moreover, each PO of a DAG can be replicated into different processing operator instances (POIs) to increase its throughput. Figure 4.2 presents a possible deployment of the wordcount application. PO $A$ is replicated twice, with POIs $A_1$ and $A_2$ executing the same code in parallel. In the remainder of this chapter, $X_i$ refers to an instance of PO $X$ executed on server number $i$. In the case of a stateless PO such as $A$ and $B$, this replication is trivial: since no state is maintained, it does not matter which

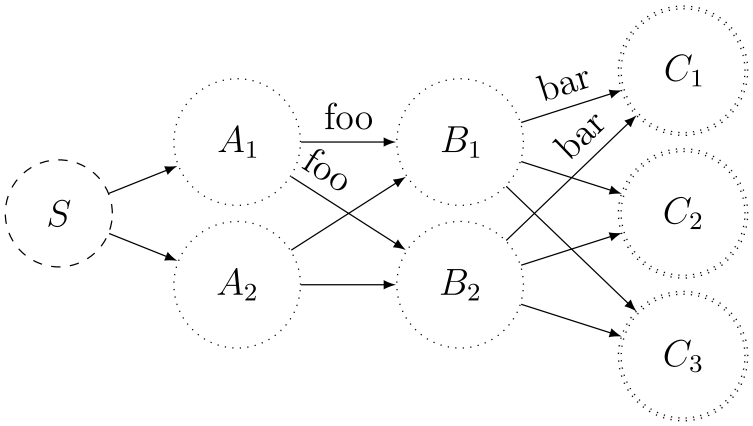


Figure 4.2 – Three components of a DAG having respectively 2, 2 and 3 instances each. $A$ and $B$ are stateless while $C$ is stateful.

instance of the PO processes a given tuple of data. Hence, data streams can be split arbitrarily between POIs of the same PO. However, replicating stateful POs such as $C$ is more complex. If a word $w$ appearing multiple times in the input sentences is processed each time by either $C_1$, $C_2$ or $C_3$, two problems appear. The first one relates to scalability. Each of the 3 POIs of $C$ maintains a state related to the frequency of $w$, so the total memory consumption of $C$ increases linearly with the number of replicas. The second problem is that no single POI of $C$ has the correct information regarding the frequency of $w$. It is impossible for a POI of $C$ to detect when the frequency of $w$ reaches a given threshold, in order to trigger an action for instance. Thus, it is important to ensure that all occurrences of $w$ are routed to the same instance of $C$. While replicating processing operators is key to scale stream processing, it must be used in combination with an appropriate stream routing policy. We detail stream routing policies in Section 4.2.2.

### 4.2.2 Stream routing policies

The specification of the application DAG indicates which PO is the recipient of another PO's output stream. When several POIs are deployed for a given PO it is important to also select which particular POI receives each tuple of data from a stream: this is the role of the routing policy. Each edge of the DAG is labeled with a choice of routing policy indicating how the output stream of a PO is split between the POIs of the recipient PO. When a POI $X_i$ emits data for a POI $Y_i$, both POIs are executed by the server $i$, so passing the data is extremely fast. Only an address in memory is transmitted from a thread to another. However, when $X_i$ communicates with $Y_j$ with $i \neq j$, the two POIs are deployed on different servers, which can be on different racks, or even different locations. The tuple goes through the network, which is less efficient and can constitute a bottleneck for the system. We now review the 3 main stream routing policies.

### Shuffle grouping

Data tuples are sent to a POI of the next PO with a round robin fashion. This ensures a perfect load balancing between the POIs. This can however only be used when routing to a stateless PO, as described in Section 4.2.1. POI co-location is not taken into account, meaning that a message can be routed to another server even if there is an instance of the destination PO on the current server. This incurs a significant network overhead. This is particularly inefficient when successively executing two stateless POs, such as $A$ and $B$ in the example of Figure 4.2. If $B$ is deployed over 5 POIs, then 80%, i.e. 4 tuples out of 5, of the data from $A$ to $B$ goes through the network. This proportion only grows as the system is deployed on larger clusters. To solve this issue, $A$ and $B$ could be combined in a new PO $AB$ executing both operators consecutively. This would avoid network communication, but drastically limits the modularity and reusability of the POs.

### Local-or-shuffle grouping

This grouping is an optimized version of the shuffle grouping: when a POI of the recipient PO is located on the same server, it is selected. This policy opportunistically avoids needless network communication, and mimics the existence of the $AB$ PO mentioned above. The guarantees in terms of load balancing are slightly weaker, but for most applications, if the input of a $A$ is balanced then its output, and thus the input of $B$, remains balanced without the need for a round robin assignment. Similarly to shuffle grouping, this grouping is not appropriate for stateful POs.

### Fields grouping

This policy is used when routing to a stateful PO. The developer selects fields of data tuples as a key to determine the recipient POI, similarly to the reduce function in MapReduce [56]. Hence, all tuples having the same key are sent to the same POI. In the example of Figure 4.2 each word is used as a key when routing from $B$ to $C$. The default implementation of fields grouping uses a hash function on the key to determine the recipient POI, but it is also possible to define and maintain routing tables to explicitly assign keys to POIs. Fields grouping is necessary to ensure the consistency of stateful POs.

However, it can lead to load balancing issues when the distribution of keys is skewed.

## 4.3 Locality-aware routing

In this section, we first state the problem, before providing our solution. In order to do so, we describe how we identify the correlation between keys by collecting statistics from the different POIs. We then focus on the generation of the routing table, reduced to a graph clustering problem. Finally, we describe our protocol that allows the routing tables of POIs to be changed dynamically.

### 4.3.1 Problem statement

In this chapter, we tackle the problem of optimizing stream processing applications. In Section 4.2.2, we explain that network communication constitutes a bottleneck in stream processing. Such bottleneck can easily be avoided when routing to stateless POs through the use of local-or-shuffle grouping. Hence, we argue that the main limitation comes from stateful POs that require fields grouping to ensure consistency. Our objective is to maximize the locality of streams routed using fields grouping.

Figure 4.3 shows a possible deployment of a stream processing application. Each of the 4 POs has two POIs, deployed on two different servers. $A$ and $C$ are stateless, while $B$ and $D$ are stateful, and maintain a state related to the keys of the data tuples they receive, using a hash map for example. Stream routing policies are indicated on the edges between POIs. Routing to $C$ is local to each server, but routing to $B$ and $D$ incurs network traffic between servers. Our goal is to minimize the amount of traffic spanning across servers, i.e. the traffic along $(A_1, B_2)$, $(A_2, B_1)$, $(C_1, D_2)$, $(C_2, D_1)$. More generally, given a stream processing application with POs replicated across different servers, for any two POs $X$ and $Y$ connected through fields grouping, our goal is to minimize:

$$\sum_{i \neq j} \mathit{traffic}(X_i, Y_j)$$

where the *traffic* function indicates the number of data tuples sent from one POI to another. A trivial solution to this problem is to process all data on a single server, which negates the benefits of deploying the application on an additional server. Hence, we add the constraint that the load of the application should remain balanced between POIs: the number of data tuples received by a POI should not be higher than $\alpha$ times the average number

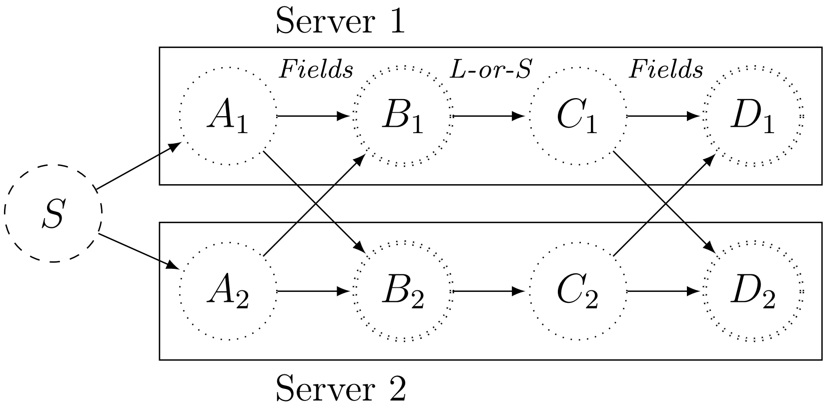


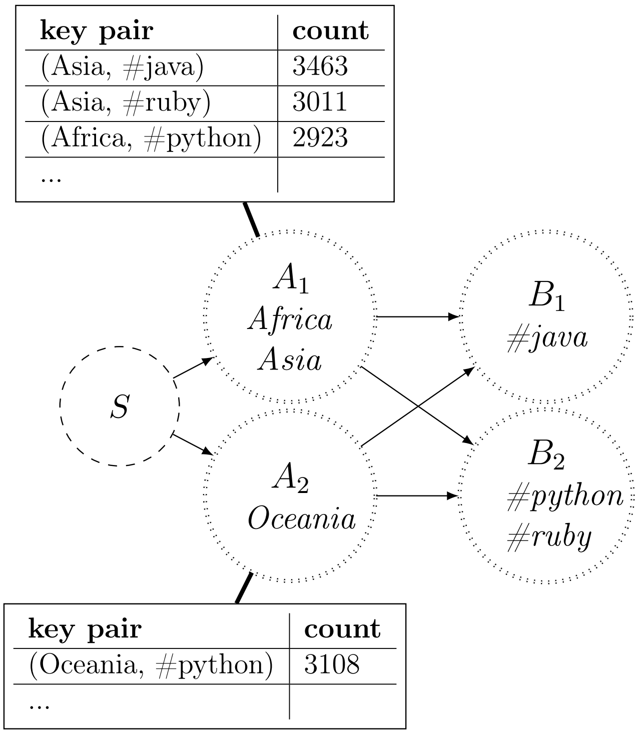
Figure 4.3 – Deployment of a stateful streaming application, with fields grouping linking POs $A$ and $B$, local-or-shuffle grouping linking POs $B$ and $C$, and fields grouping linking POs $C$ and $D$.

of tuples received by POIs of the same PO, where $\alpha \geq 1$ is the imbalance bound.

In this work, we assume that the deployment of POIs on servers is static. Our contribution is a protocol that generates optimized routing tables for fields grouping. We propose to detect at run time the correlations between keys used for consecutive field groupings in order to ensure that those keys are handled by POIs located on the same server. For instance, if the POI $B_1$, after receiving a tuple with key $k$, frequently leads to sending a tuple with key $k'$ to PO $D$, then we should make sure that $D_1$ is the POI responsible for $k'$. As streaming applications are executed on unbounded streams, the characteristics of data may change over time. The routing optimization protocol detects these changes and generates new appropriate routing tables. In addition, as a key is assigned to a different POI, a data migration protocol ensures that the state of the application is preserved.

### 4.3.2 Identifying correlations

Our goal is to leverage the correlation between keys used in consecutive fields grouping to avoid routing streams over the network. The first step consists in detecting candidate pairs of keys that show potential for optimization. Let $X$ and $Y$ be two consecutive stateful POs with instances over a set of $n$ servers. Given two keys $k$ and $k'$ used when routing to $X$ and $Y$ respectively, the probability of the POIs processing a tuple containing $k$ and $k'$ being located on the same server is $1/n$ when using hash functions. Hence, by ensuring that $k$ and $k'$ route to the same server, the expected number of



| key pair | count |
|---|---|
| (Asia, #java) | 3463 |
| (Asia, #ruby) | 3011 |
| (Africa, #python) | 2923 |
| ... | |

| key pair | count |
|---|---|
| (Oceania, #python) | 3108 |
| ... | |

Figure 4.4 – PO instrumentation: every instance counts the key pairs it receives and sends, and keeps the most frequent pairs in memory.

network messages avoided is $f(k, k').(n-1)/n$ where $f(k, k')$ is the number of data tuples containing $k$ and $k'$. This shows that *(i)* routing optimization becomes increasingly important as the number of servers increases, and *(ii)* frequent pairs lead to the highest gains. Indeed, even if two keys always appear simultaneously, and thus have a very strong correlation, the gains are negligible if they are not frequent. Conversely, a loose correlation can lead to significant gains if the keys are extremely frequent.

In the remainder of this section, we use the following application as a running example: geolocated short messages containing hashtags (tweets) are analysed to generate statistics about topics trending in geographical regions [41]. The application contains two stateful POs, and routes first using the region, and then using the hashtag. If the pair *(Asia, #scala)* appears more often than the pair *(Asia, #clojure)*, it means that people in Asia tweet more often about Scala than Clojure. In this case, it is more important to co-locate on the same server the POIs dealing with the keys *Asia* and *#scala* than *Asia* and *#clojure*.

#### Offline analysis

In cases where the workload is stable, correlations between keys are assumed to remain constant over time. Consequently, it is possible to perform an offline analysis on a large sample of the data and to accurately compute the frequency of all key pairs. This information can then be used to compute optimized routing tables that can be used for long periods of time without the need for an update.

#### Online analysis

Data streams often fluctuate over time, particularly when they are generated by human activity. For example, *#breakfast* is associated to *America* and *Europe* at different moments of the day. In addition to diurnal and seasonal patterns, flash events can occur, generating temporary correlations between keys. It is necessary to detect these correlations at run time to perform an online optimization of stream routing without interrupting the execution of the application. For this purpose, we add an instrumentation tool to stateful POs. For each passing message, a POI extracts the input key, which was used to route the data tuple to this instance, and the output key, which decides towards which POI the message is routed next. Pairs of keys are stored in memory along with their frequency, as depicted in Figure 4.4. Computing the frequency of pairs of keys online is a challenging problem, as most of the resources, such as CPU and memory, should be dedicated to the application, and not collecting statistics. To this end, we rely on the SpaceSaving algorithm [90]. Using a bounded amount of memory, it maintains an approximated list of the $n$ most frequent pairs of keys. This limitation on the collection of statistics is, fortunately, not problematic for most large-scale datasets. Indeed, many real datasets follow a Zipfian distribution [25], with few very frequent keys, and many rare keys. Identifying the pairs containing the most frequent keys captures most of the potential for optimization, so the loss compared to an exact offline approach is limited. Whenever the routing of keys is updated, the statistics are reinitialized to only take into account recent data and detect new trends.

### 4.3.3 Generating routing tables

In addition to the streaming application, we execute a *Manager*, responsible for analysing the statistics collected by the POIs and coordinating the deployment of an optimized routing configuration. For each pair of consecutive stateful POs $X$ and $Y$, the manager periodically queries all POIs of $X$ to obtain

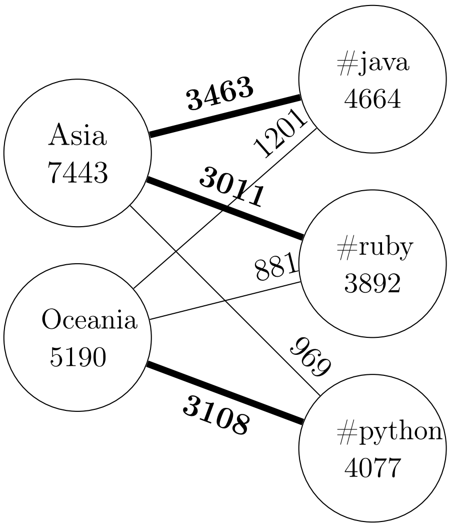


Figure 4.5 – A bipartite graph of the key pairs, showing different weights for the vertices and edges, i.e., pairs.

their statistics on the frequency of associations between keys. These statistics can be represented as a bipartite graph connecting keys of $X$ to keys of $Y$. Each key is a vertex weighted by its frequency, and an edge between keys is weighted by the frequency of their co-occurrence. Figure 4.5 shows the bipartite graph, as it would be constructed with the data in Figure 4.4. The pairs observed the most frequently are represented by bold edges.

Creating the best assignment of keys to servers reduces to a graph partitioning problem. We want to partition the bipartite graph such that pairs of keys that appear together frequently are in the same group. In other words, we try to miminize the added weights of the cut edges. That way, we can co-locate on the same servers the instances which deal with correlated keys and decrease the network utilization. In Figure 4.5, assuming the application is deployed on $n = 2$ servers, *Asia*, *#java* and *#ruby* should be put on the first one, while *Oceania* and *#python* on the second one. Load balancing between servers is handled by balancing the sum of vertex weights in each partition. In practice, we rely on the Metis partitioning library [76]. We construct the bipartite key graph and provide it along with a balance constraint to Metis, which in turn returns the partitioned graph. At this point, keys are assigned to servers, so we can generate routing tables mapping these keys to the POIs hosted on each server.

Each POI preceding a stateful PO uses the optimized routing table to route a key to a POI. When a key is not present in the routing table, it falls back to the standard hash-based routing policy.

### 4.3.4 Online reconfiguration

In the case of an offline analysis, optimized routing tables can be loaded at the start of the application. However, the online approach requires the manager to send the updated routing tables to POIs, while not losing any data tuple of the stream in the process.

The main difficulty when dealing with hot reconfiguration is state migration. Every stateful POI holds the state of the keys to which it is associated. When a key is assigned to a different POI in the updated routing tables, its corresponding state needs to be transferred between POIs. Moreover, after POIs migrate the state of their previous keys, they should no longer receive any message related to this key. This means that preceding POs in the DAG must have proceeded to their reconfiguration first, and route messages according to the new routing tables. To this purpose, we implemented a protocol which orchestrates a progressive reconfiguration following the PO order specified by the DAG. This protocol is shown in Figure 4.6 and described more formally using pseudo-code in Listing 4.1.

The reconfiguration protocol is executed by the manager $M$. The manager first asks every running POI to send the collected statistics (1). Upon receiving them all (2), it builds the bipartite graph of the key pairs (see Figure 4.5), partitions this graph with Metis, and computes the new routing tables. It sends these tables to the respective POIs (3), and waits for all acknowledgements (4). It then enter the propagates phase, and tell the instances of the first PO to proceed to the reconfiguration (5). The two instances update their routing table and exchange the state of the keys whose assignment has changed (6). After this operation, they forward the propagation instruction to the instances of the second PO (5), which in turn update their routing tables and exchange their states if necessary (6).

The computational cost of this reconfiguration is linear in time with respects to the number of POs, and linear in space with respects to the number of key pairs which are counted.

The reconfiguration message sent by the manager to every instance is a data structure containing:

- $\textit{reconfiguration}_{router}$: The new routing table, associating keys to POIs.
- $\textit{reconfiguration}_{send}$: The list of keys whose state must be transferred along with their respective recipients.
- $\textit{reconfiguration}_{receive}$: The list of keys whose states are expected to be received from other instances of the same PO.

```
manager_migration():
  for poi in POIS:
    send(poi, GET_METRICS, nil)                        (1)

  for poi in POIS:
    metrics.add(receive(SEND_METRICS))                 (2)

  reconf = compute_reconfiguration(metrics)

  for poi in POIS:
    send(poi, SEND_RECONF, reconf[poi])                (3)

  for poi in POIS:
    receive(ACK_RECONF)                                (4)

  for poi in POIS:
    if predecessors[poi].is_empty():
      send(poi, PROPAGATE, nil)                        (5)

poi_migration():
  receive(GET_METRICS)                                 (1)
  send(manager, SEND_METRICS, my_metrics)              (2)

  my_reconf = receive(SEND_CONFIGURATION)              (3)
  send(manager, ACK_RECONF, nil)                       (4)

  if my_predecessors.is_empty():
    receive(PROPAGATE) # from manager
  else
    for poi in my_predecessors:
      receive(PROPAGATE)                               (5)

  update_routing(my_reconf[ROUTER])

  for (poi, keys) in my_reconf[SEND]:
    send(poi, MIGRATE, state(keys))                    (6)

  for (poi, keys) in my_reconf[RECEIVE]:
    state.add_all(receive(MIGRATE))                    (6)

  for poi in my_successors:
    send(poi, PROPAGATE, nil)                          (5)
```

Listing 4.1 – Pseudo-code for the reconfiguration algorithm.

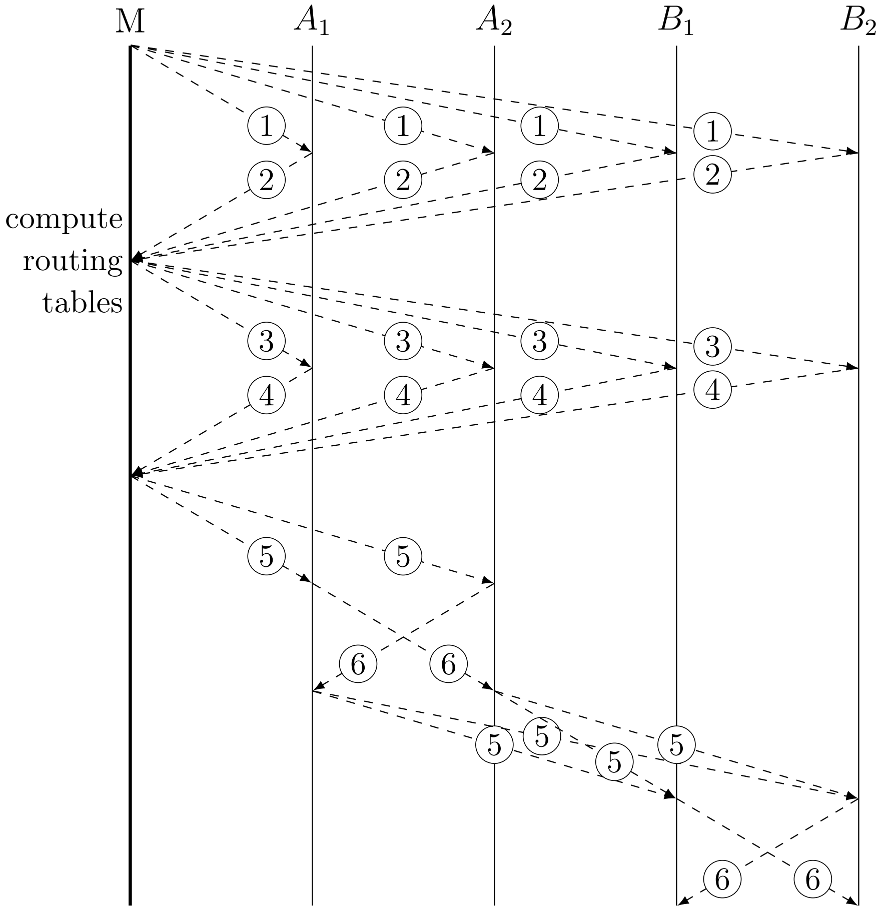


Figure 4.6 – Reconfiguration protocol, forwarding routing tables and key states between POIs. (1) Get statistics. (2) Send statistics. (3) Send reconfiguration. (4) Send ACK. (5) Propagate. (6) Exchange keys.

The data stream is not suspended during reconfiguration, so it is possible that a POI receives a tuple associated to a key while it has not yet received the state associated to it. In this case, tuples are buffered and are only processed once the state of their key is received. This solution is preferable to suspending the stream as some stream sources do not support back pressure and would lose messages. To handle fault tolerance, the manager saves all routing configurations to stable storage before starting reconfiguration. If a POI crashes, the guarantees are the ones provided by the streaming engine and are not impacted by state migration.

## 4.4 Evaluation

In this section, we evaluate the benefits of locality-aware routing in stateful streaming applications by implementing it in Apache Storm. We first present the experimental setup. Then, we perform a thorough evaluation of application throughput when varying a wide range of configuration parameters using a synthetic workload. Afterwards, we switch to a real workload from Twitter to evaluate the benefits of an online approach in the case of a fluctuating workload. Finally, we evaluate the impact of reconfiguration on a real stable dataset from Flickr.

#### Summary of results

We show using a synthetic workload that locality-aware routing significantly outperforms hash-based routing. The difference increases with the number of servers, the size of data tuples, and the amount of locality in the data. We observe on a Twitter dataset that associations between hashtags and locations vary over time. Our approach achieves a 50% locality on this dataset when deployed on 6 servers, compared to 17% for hash-based routing. We demonstrate that online reconfiguration is necessary to maintain this locality, as an offline approach is unable to leverage transient correlations. We also show that only the most frequent key associations are necessary to achieve high locality score, thereby confirming that 1MB of memory per POI is sufficient for collecting statistics. Finally, we observe the impact of online reconfiguration on a stable dataset from Flickr, and notice a significant throughput increase, thus validating our approach.

### 4.4.1 Experimental Setup

We implement locality-aware routing in Apache Storm in order to evaluate its performance. We now describe the testbed and the application deployed.

#### Servers

Our platform for the experiments is a cluster of 9 physical servers. One of them runs Nimbus, i.e. the Apache Storm master, and the other 8 are Apache Storm workers. The workers are HPE Proliant DL380 Gen9 servers with the following specifications:

- **RAM** 128 GB of DDR4 memory.
- **CPU** 2x 10-Core Intel Xeon E5-2660v3@2.6 GHz

- **Network** The machines are linked through a 10 Gb/s network, using Jumbo frames (MTU 9000). We also experiment 1 Gb/s network speed by using a software to throttle bandwidth.
- **Disk** 15 × 6TB SATA disk.
- **Software** Debian 8.3.0 (stable), and Apache Storm 0.9.5.

### Application

For evaluation purposes, we consider the case of a streaming application closely related to the one described in Section 4.3. It consists of one source, $S$, and two stateful POs, $A$ and $B$. The first PO computes statistics based on the first field of the tuples by counting the number of occurrences of its different values, and the second PO executes the same operation on the second field. Hence, fields grouping is used to route data tuples to both POs. Each PO is deployed as $n$ instances, $n$ varying between 1 and 6. We later refer to the number of instances as the parallelism of an experiment. We use the same parallelism for both POs to ensure that every instance of the first PO has a local instance of the second. For each PO $X$, the POI $X_i$ is hosted on server $i$. Hence, each server hosts both an instance of $A$ and an instance of $B$.

## 4.4.2 Locality impact using synthetic workload

To assess the impact of locality on the performance of streaming applications, we first consider the case of a synthetic workload. The source generates tuples containing three fields: $(integer, integer, padding)$. The first two fields, the integers, vary between 1 and $n$. In this experiment, the *locality* parameter controls the number of tuples in which these integers are equal. The last field is here to allow experimenting with different tuple sizes, to simulate workloads ranging from single words to small texts. We vary the padding between 0 and 20kB.

We evaluate the performance of the application on this workload using 3 different versions of fields grouping.

- *Hash-based:* tuples are assigned to a POI using a hash function on the key. This assignment is random but deterministic, and represents the default implementation of fields grouping in Storm.
- *Locality-aware:* each tuple $(i, j, p)$ is routed to the $i$-th instance of $A$, $A_i$, and then to the $j$-th instance of $B$, $B_j$. Doing so, all the tuples of type $(i, i, p)$ are routed on the same server $i$. This implementation

of fields grouping represents the approach advocated in this chapter. These are the routing tables that would be generated by analysing the data.

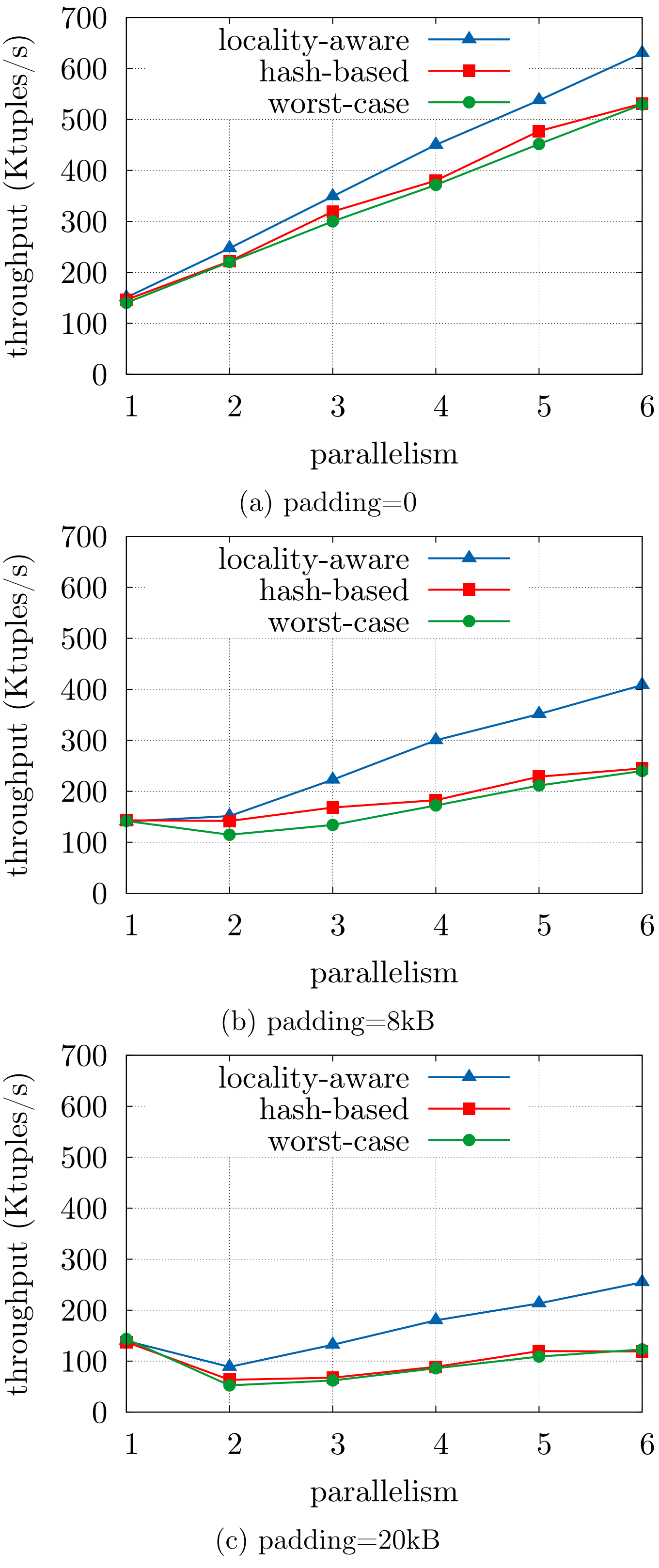
- *Worst-case:* tuples of type $(i, i, p)$ are always routed through the network. This represents a worst case scenario that has negative synergy with locality. This allows us to obtain a lower bound on the performance of the application.

We iterate over different values of following parameters: *parallelism*, *padding size*, and *locality*; and run the application with each of the 3 versions of fields grouping presented above.

Figures 4.7 and 4.8 depict the throughput of the application when varying parallelism. With a locality of 60%, all versions of fields grouping send messages over the network. We observe that, as the size data tuples (padding) increases, the gains of adding additional servers decrease. When padding is at 20kB, we even notice a decrease of performance when switching from 1 to 2 servers. This behavior is symptomatic of stateful streaming applications: network constitutes a bottleneck, and the proportion of tuples transiting through the network increases with parallelism. For all padding sizes, *locality-aware* clearly outperforms the other options, as it is the only option that scales linearly for a parallelism higher than 2. A locality of 100% constitutes an ideal case, in which *locality-aware* avoids all network communications. In this setup, padding has no effect on the throughput as tuples are transferred in memory. This experiment highlights the impact of network communications on the throughput of streaming applications. Even when tuples are extremely small (padding $= 0$), routing through the network lowers the performance by 22%. As expected, this value increases with the size of tuples.

Figure 4.9 presents results for varying values of locality. *Hash-based* is not affected by locality, as it do not leverage it. *Locality-aware* gains performance as the locality of the dataset increases, since the amount of network communication decreases linearly with locality. We notice a plateau above 90% of locality.

Figure 4.10 illustrates the variation of throughput for varying padding sizes. The difference between *locality-aware* and the other options increases both with the size of padding, and with parallelism. This behavior is explained by the fact that in this experiment, *locality-aware* is able to preserve a ratio of 80% local communications, while *hash-based* sends more network messages as parallelism increases. Furthermore, network saturates faster when padding is high. We note that in the most challenging configurations, the performance of *hash-based* and *worst-case* are very similar.

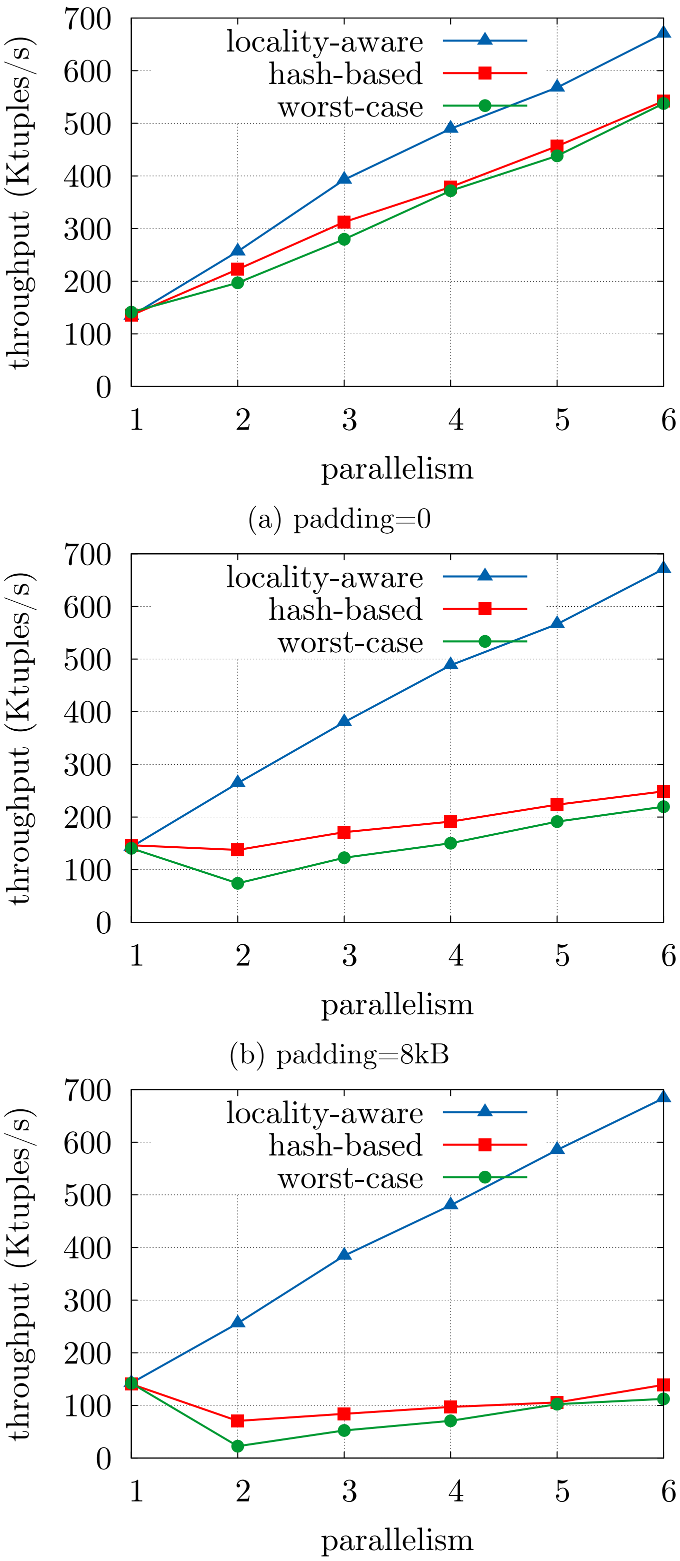


(a) padding=0

(b) padding=8kB

(c) padding=20kB

Figure 4.7 – Throughput when varying parallelism for 60% locality

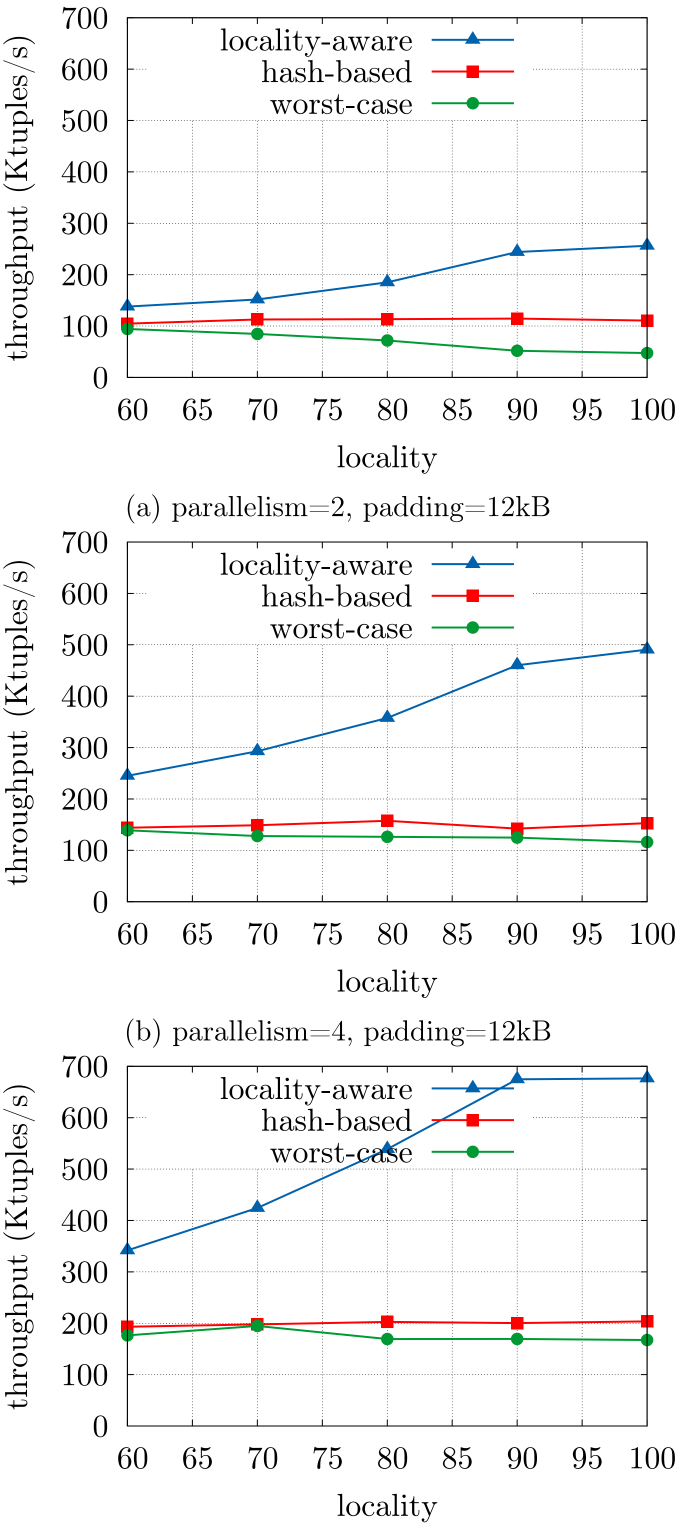


(a) padding=0

(b) padding=8kB

(c) padding=20kB

Figure 4.8 – Throughput when varying parallelism for 100% locality

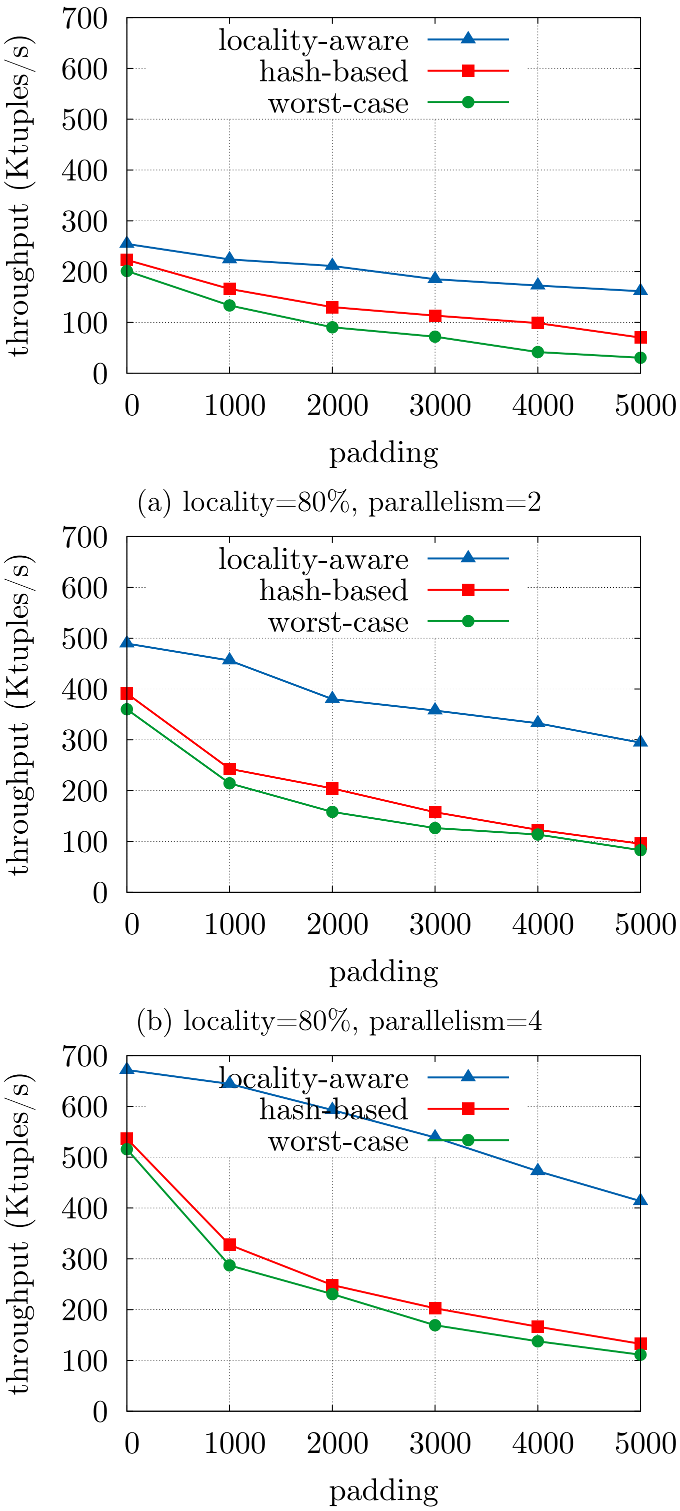


(a) parallelism=2, padding=12kB

(b) parallelism=4, padding=12kB

(c) parallelism=6, padding=12kB

Figure 4.9 – Throughput when varying locality, with a message size of 12kB and different parallelisms



(a) locality=80%, parallelism=2

(b) locality=80%, parallelism=4

(c) locality=80%, parallelism=6

Figure 4.10 – Throughput when varying tuple sizes, with a locality of 80% and different parallelisms

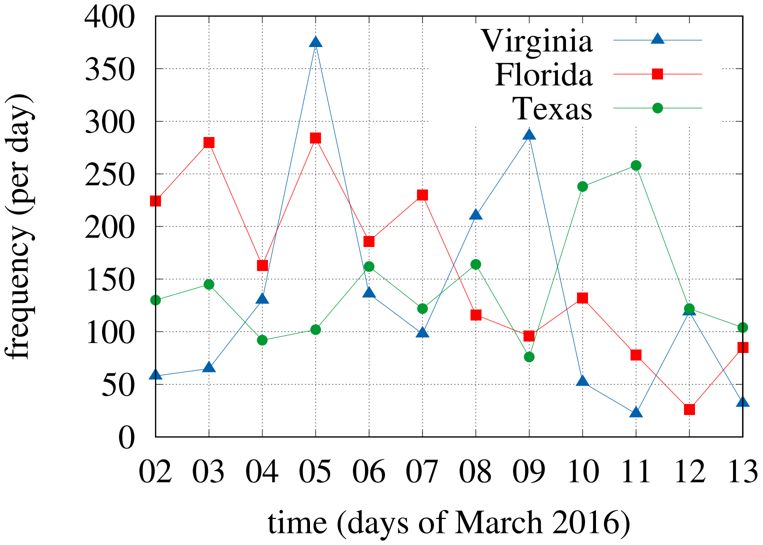


Figure 4.11 – Occurrences of the hashtag #nevertrump in different states in the USA.

### 4.4.3 Impact of online optimization

The approach presented in this work is online. It can detect correlation between keys on a running application, and optimize locality even when the characteristic of data vary over time. In this section, we evaluate the benefits of online optimization over offline optimization. To this end, we rely on a real dataset from Twitter containing timestamped content.

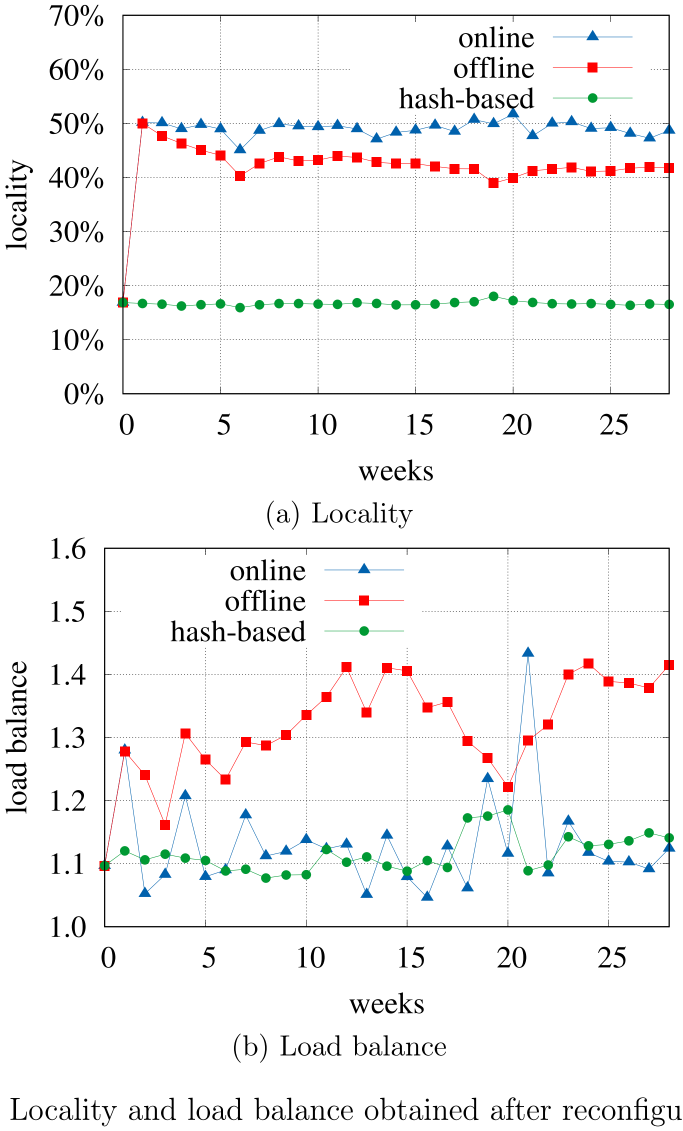
#### Twitter dataset

We crawl tweets from October 2015 to May 2016 using the API of Twitter. Twitter provides for each tweet a location identifier which can be either the location of the user at the moment of the tweet, or a location associated to the content of the tweet. Locations can be countries, cities, or points of interests. Overall, our dataset contains 173 million associations between locations and hashtags. We set our application to first route using the location, and then the hashtag.

#### Evolution of correlations over time

In social media, trends vary constantly and are often linked to events. Figure 4.11 shows the frequency of a popular hashtag on Twitter for different locations. While the tag appears on all of the three locations, it is clearly more correlated with Florida on March $3^{rd}$, with Virginia on the $9^{th}$ and with

Texas on the $11^{th}$. In the context of our application, this means that, to optimize performance, the same hashtag should be co-located with 3 different locations at different periods of time. This justifies the online nature of our reconfiguration protocol: it is important to stay up-to-date regarding volatile correlations. The next experiment shows how reconfiguration affects the processing locality.



(a) Locality

(b) Load balance

Figure 4.12 – Locality and load balance obtained after reconfiguration with a parallelism of 6, and period of one week. Online: reconfiguration every week. Offline: one reconfiguration after one week. Hash-based: no reconfiguration.

### Online and offline optimization

In this experiment, we compare the effectiveness of the *offline* approach, that computes a single configuration using a sample of data, and the *online* approach, that continuously updates the configuration. We use 1 week of data for the *offline* approach, while the *online* approach updates the configuration every week. We also present the performance of *hash-based* routing as a reference. We run this experiment with a parallelism of 6.

Figure 4.12a shows the evolution of locality over time, *Hash-based* achieves a locality of 16.6%, which corresponds to a random assignment with 6 servers. After one week, *online* and *offline* both obtain a sufficient amount of data to perform locality-aware routing, which raises the locality to 49%. However, this value decreases over time in the case of *offline*, and stabilizes around 40%. *Offline* preserves locality for stable associations, but fails to leverage transient ones. *Online* however maintains a locality in the vicinity of 50%. This shows that to capture volatile correlations, reconfiguration should be triggered on a regular basis. The experiments of Section 4.4.2 indicate that a 10% difference in locality can lead to a throughput gain of 25%. This demonstrates the benefits of the online optimization process in the case of fluctuating workloads. Note that when generating routing tables, Metis reports an expected locality of 75%. However, this locality is only achievable by running the exact dataset that was used to compute the configuration. In practice, data of the next week contains a significant proportion of new hashtags and locations that were not observed previously and are thus routed using hash functions.

Now that we have established the need for regular reconfiguration in order to optimize locality, we focus on the impact of reconfiguration on load balancing. As shown by Figure 4.12b, *hash-based* distributes the load fairly evenly, with an average of 12% additional traffic for the most loaded POI. *Online* and *offline* both start with a balanced load, thanks to the statistics collected on the frequency of keys. As the workload fluctuates, some hashtags and locations become more frequent in the following weeks. This causes the distribution of the load to deviate significantly. When optimizing for locality, correlated keys are assigned to the same server. While this contributes to diminishing network consumption and increasing throughput, this has a potential drawback: correlated keys have a higher probability to have peaks of activity simultaneously. *Online* is able to immediately correct these spikes of unbalance, while *offline* stabilizes around 40% imbalance. This experiment confirms that locality-aware routing is able to preserve load balance, and that reconfiguration needs to be carried out regularly on fluctuating workloads. We deliberately use long intervals of time (1 week) between reconfigurations for the online approach to highlight deviations from optimal behavior (imbalance

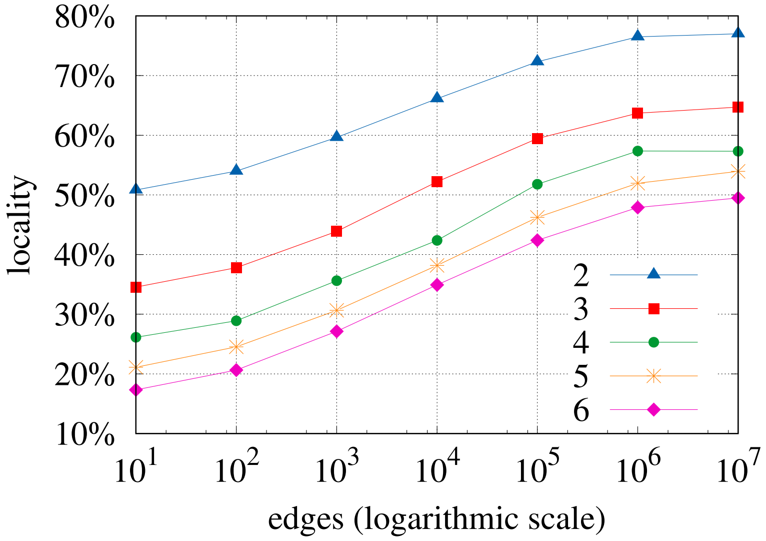


Figure 4.13 – Locality achieved when varying number of considered edges, for different parallelisms.

spikes). In practice, as we show in Section 4.4.4, reconfiguration is extremely fast and can be triggered much more frequently to account for deviations in the frequency of keys.

Please note that the imbalance parameter $\alpha$ specified in Section 4.3.1 is indeed used and set to 1.03, which is Metis default value. However, while it is respected for the collected data on which the partitioning is achieved, its impact on the future data cannot be predicted, although the reconfiguration greatly improves the load balance as shown in Figure 4.12b.

### Statistics collection

As stated in Section 4.3.2, our online protocol uses a bounded amount of memory to collect statistics, and thus only retrieves information on the most frequent pairs of keys, or edges. This experiment aims at quantifying the impact of the number of considered top edges on the quality of the reconfiguration. Figure 4.13 shows how the achieved locality varies with the number of edges that we consider when performing the graph partitioning for different parallelisms. Naturally, having information on more pairs of keys results in better locality. However, considering the logarithmic scale of the figure, we can double the locality for parallelism 6, for instance, with only 0.1% of the total edges. Therefore, a quality/capacity trade-off is to be made when choosing the number of edges for the reconfiguration protocol, depending on the application and environment on which it will run. On this dataset, we find $10^6$ edges is sufficient, so collecting statistics only occupies a few MB of memory of each POI.

### 4.4.4 Reconfiguration protocol validation

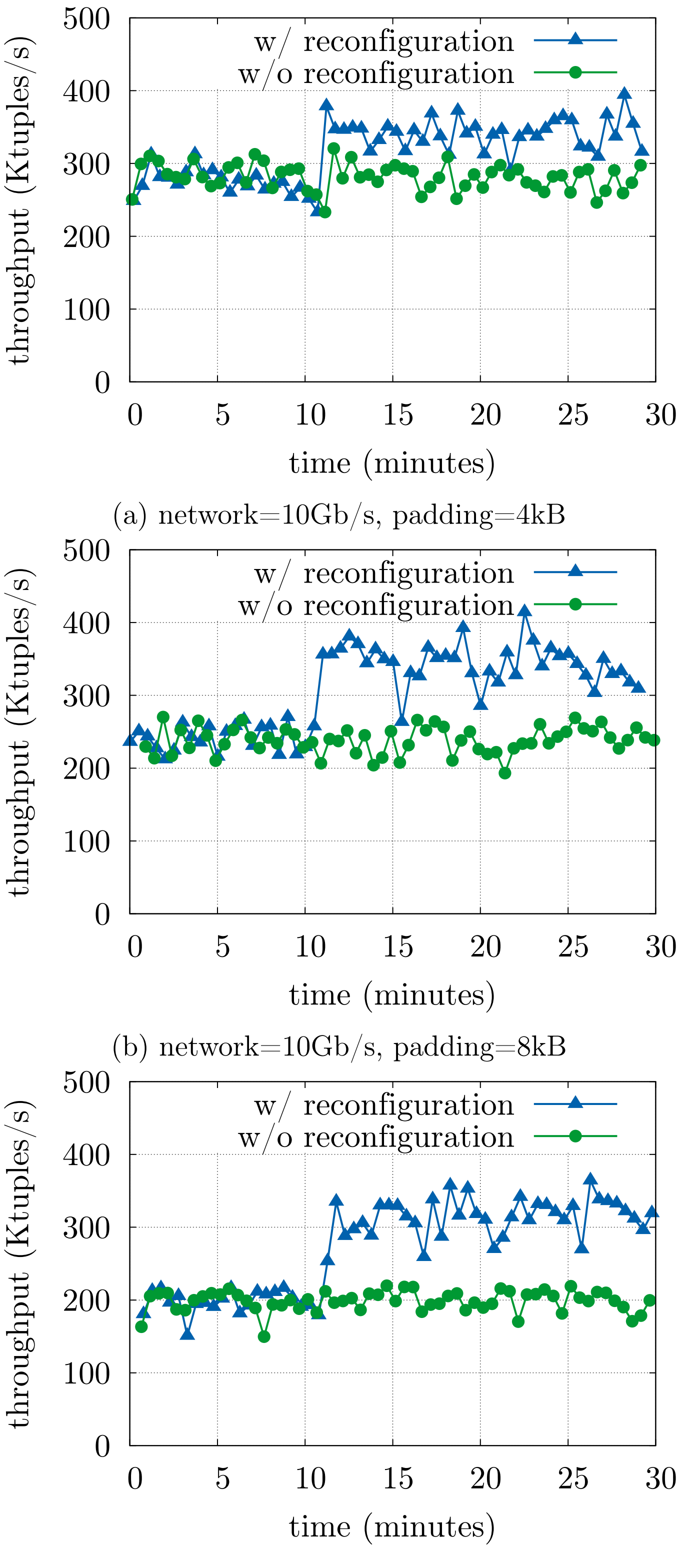
The following experiments launch the application with and without our reconfiguration protocol and see how the reconfiguration affects the application's performance. The tuples processed by our streaming application are of type $(tag, country, padding)$ which come from a dataset provided by Flickr and described below.

**Flickr dataset**

This dataset [5] contains metadata about 100 million pictures posted on Flickr. Among other fields, there is a geolocation and a list of user tags for every picture. The geolocation is mapped to a country using data from OpenStreetMap. This dataset represents a stable workload as there is no temporal information and images are not ordered.

**Throughput analysis**

In this set of experiments, we launch our streaming application either without reconfiguration or with a reconfiguration every 10 minutes, each run lasts for 30 minutes. We do so for different tuple sizes (paddings) and using two different network settings, 10Gb/s and 1Gb/s. As shown by the different plots of Figures 4.14 and 4.15, a significant improvement of throughput follows the first reconfiguration at $t = 10min$ and is maintained throughout the run. This proves that the real-life correlation of Flickr data is sufficient to enhance performance through locality. As established in Section 4.4.2, the performance gain does indeed increase following the size of the tuples. By comparing results at 10Gb/s and results at 1Gb/s, we can see that the effect of tuple size is even greater in the case of a more limited bandwidth. As for the effect of parallelism, i.e., the number of instances of each PO, Figure 4.16 shows that the gap between throughput with and without reconfiguration is more important for higher numbers of instances. These executions on real workloads confirm the results obtained on synthetic workloads presented in Section 4.4.2. We also notice that deploying an updated configuration and migrating data is extremely fast and does not impact performance negatively, as the throughput increase is noticeable immediately after $t = 10min$.

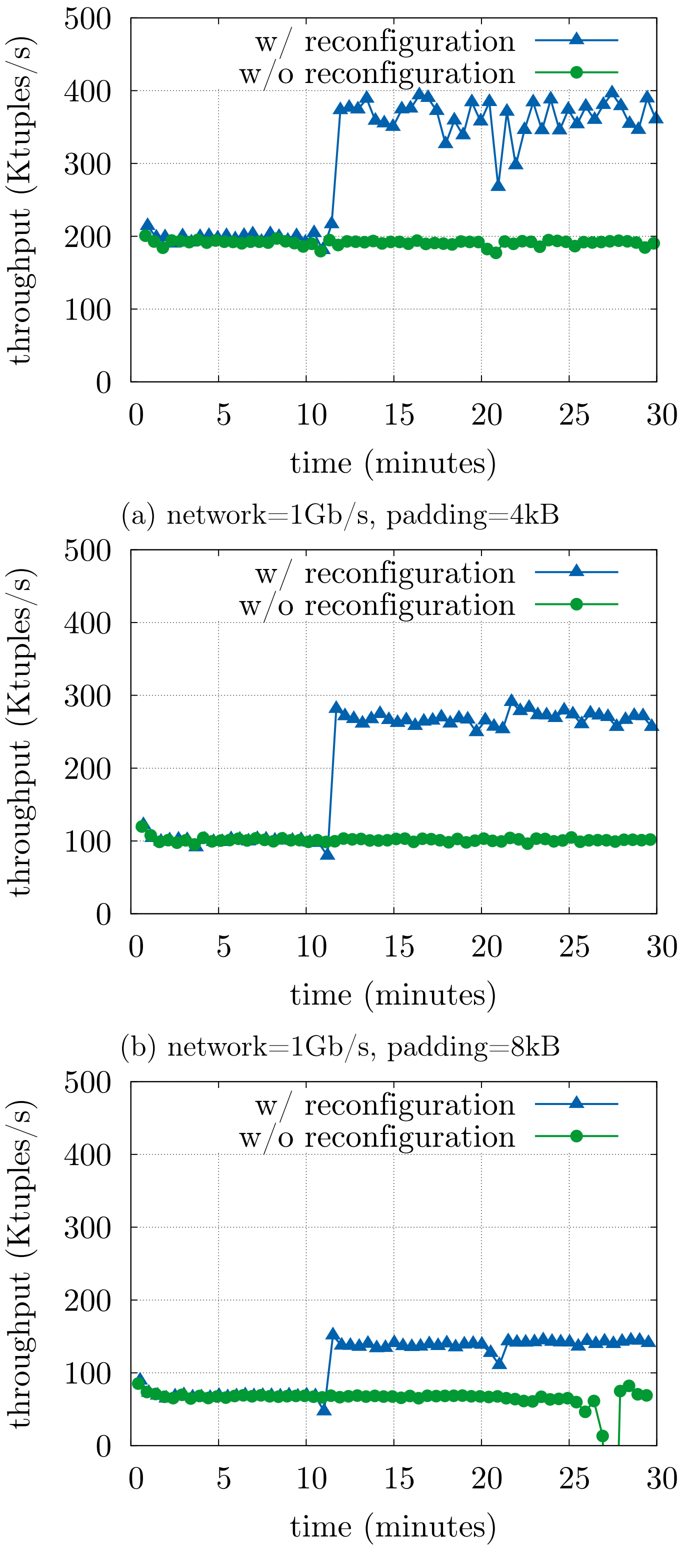


(a) network=10Gb/s, padding=4kB

(b) network=10Gb/s, padding=8kB

(c) network=10Gb/s, padding=12kB

Figure 4.14 – Evolution of the throughput with or without reconfiguration, for a parallelism of 6, different padding sizes and a 10Gb/s bandwidth



(a) network=1Gb/s, padding=4kB

(b) network=1Gb/s, padding=8kB

(c) network=1Gb/s, padding=12kB

Figure 4.15 – Evolution of the throughput with or without reconfiguration, for a parallelism of 6, different padding sizes and a 1Gb/s bandwidth

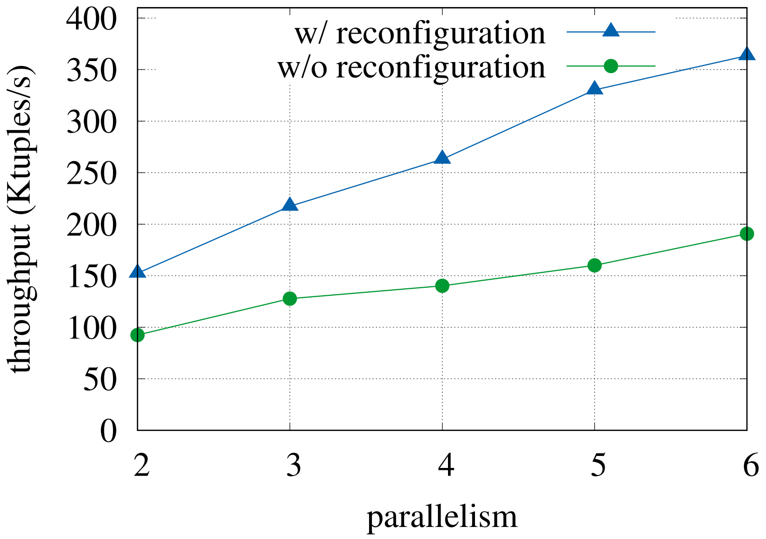


Figure 4.16 – Average throughput for different parallelisms, and a padding of 4kB (on the 1Gb/s network). With reconfiguration, the average is measured after the first reconfiguration.

## 4.5 Related work

### 4.5.1 Operator instance scheduling

When executing a streaming application, the scheduler deploys POIs on servers. The assignment of POIs to physical servers has a significant impact on the performance of the application. A first optimization criterion is ensuring that each server is assigned an even share of the computational load. A second objective is to locate tasks communicating frequently on the same servers to avoid network saturation.

In the stream processing engine *System S* [77], several operators are fused into a single processing element to achieve a good trade-off between communication cost and execution parallelism. The approach proposed is top-down, and starts by considering that all operators are part of the same processing element. This processing element is then recursively divided using graph partitioning algorithms.

Aniello et al. [26] proposed two schedulers for Storm. The offline scheduler only considers the topology of the application and tries to place consecutive POs on the same server. The online scheduler measures CPU and memory consumption of POIs, as well as communication patterns between POIs. POIs are then assigned to servers using a greedy algorithm, starting with the pair of POIs that communicate the most. The topology used for evaluation alternates shuffle grouping and fields grouping for routing, and fields grouping relies on hash functions only. The addition of local-or-shuffle grouping would significantly contribute to reducing network communications.

Fisher et al. [58] solve the scheduling problem using graph partitioning. POIs are vertices of the graph, and are weighted by the computational resources they consume. Edges represent communications between POIs, and are weighted by the amount of data that transits. In practice, the authors use Metis to obtain high quality partitions while preserving load balance.

R-Storm [98] is a scheduler for Storm that aims at maximizing resource utilization while minimizing latency. The developer declares the memory and CPU utilization of each PO using a specific API. Then, a Knapsack-based heuristic performs the POI assignment. The authors describe two topologies typical of applications used by Yahoo in production. One of them is a chain, while the other starts with a transformation PO branching into two chains.

Cardellini et al. [47] proposed to perform an embedding of servers into a 4-dimensional cost-space. These dimensions represent network characteristics (latency and throughput), as well as processing power (utilization and availability). Coordinates are then used to optimize scheduling, using a spring-based formulation. The main novelty of this work is the use of the P2P algorithm Vivaldi to assess network latency in a distributed manner.

The problem of scheduling POIs is orthogonal to our contribution. We assume the existence of a scheduler that assigns POIs to servers, and take this assignment as an input constraint. While schedulers see fields grouping as a black box that cannot be optimized, our approach is able to improve data placement to further reduce network usage. Any online scheduler that actively measures communication between POIs can then notice the improvement and re-visit the POI placement decision, leading to even better performance. Our approach is similar to [58] as it relies on Metis for graph partitioning. Instead of considering a graph of POIs communicating, we consider a graph of keys that co-occur in the data.

### 4.5.2 Load balancing for stateful applications

Load balancing consists in ensuring that each server involved in a distributed system receives an even share of the total load to avoid bottlenecks. As explained in Section 4.2.2, stateless streaming applications rarely suffer from imbalanced load, as data tuples can be sent to any instance of a given PO. However stateful application use fields grouping to ensure that data tuples containing the same key always reach the same POI. In the case of skewed data distribution, POIs responsible for keys occurring frequently receive more tuples to process than other POIs, and become bottlenecks. Several solutions have been proposed to limit the impact of data skew.

Nasir et al. [92] propose to use *partial* key grouping, where a key can be sent to two different POIs instead of one. Each POI locally estimates the load of its successors, and sends the data tuple to the least loaded of the two options. This solution is elegant, as it relies on two hash functions and does not require maintaining routing tables. However, it leads to additional memory consumption as the state associated to a given key is maintained by two POIs. This state is aggregated downstream, which limits the use of this solution to associative operators and introduces latency. Partial key grouping was improved to handle extremely frequent keys [93]. A list of the most frequent keys is maintained using the SpaceSaving algorithm [90]. These keys can be routed to any POI, instead of just two. This further increases memory consumption but improves load balancing in extreme cases.

Similarly to [93], Rivetti et al. [102] proposed DKG, an algorithm that maintains a list of the most frequent keys that cause load imbalance. These keys are then explicitly mapped to POIs using routing tables, while less frequent keys are routed using hash functions. This approach also relies on the SpaceSaving algorithm for estimating frequencies.

Skewed key distribution also cause load imbalance in database systems. E-store [107] keeps track of frequently accessed data tuples using SpaceSaving and migrates them in order to balance load between servers. A routing layer maintains the assignment of keys to servers using routing tables.

Our approach is similar to [93] as it relies on the SpaceSaving algorithm [90] to obtain an online estimation of frequency in data streams. However the statistics we collect are richer, as they involve pairs of keys. Hence, we are able to use them for network locality optimization in addition to load balancing. Moreover, our algorithm handles data migration in the case of a reconfiguration. This important problem is left to the application developers in [93].

### 4.5.3 Co-locating correlated keys

Workload-driven optimization consists in analysing the workload of an application in order to tune the system to its specific activity. Schism [53] analyses query logs of shared-nothing databases. Keys accessed by queries are represented as a graph, with edges weighted with the number of co-occurrences. This graph is partitioned using Metis to obtain a new assignment of keys to servers, such that each query can be answered with as few partitions as possible. Dynasore [31] performs similar optimizations for building social feeds. User profiles that are frequently accessed simultaneously are hosted on the same server. The offline partitioning relies on Metis, while an online optimizer reacts to workload changes dynamically. A streaming application can be interpreted as a continuous query, where operators (POIs) are placed on servers and data streams through them. We aim at ensuring that all POIs impacted by an execution query on a tuple of data are located on the same server. Hence, our approach is similar to [31], as we analyse the workload to uncover correlations between keys, but is optimized for stream processing.

## 4.6 Conclusion

Stateful streaming applications suffer from increasing network consumption as they scale to multiple servers. To alleviate this drawback, we propose to increase data locality by explicitly routing correlated keys to the same servers. Our approach relies on lightweight statistics about co-occurrence of keys collected by stream operators. A manager gathers all statistics and performs a partitioning of the graph of keys to assign correlated keys to the same server while enforcing load balancing. Data related to reassigned keys is migrated following the order of operators in the application, which ensures that the state of a key is preserved. While in this work we only consider chains of POs, the same graph partitioning technique can be applied to more complex DAGs. When measuring association between keys, successor keys can be assigned to different POs, without changing the formulation of the problem. We demonstrate the effectiveness of our approach on synthetic and real datasets, by throughput increasing up to ×2 with relatively small tuples. We prove the gain increases with the tuple size.

Our approach is able to deal with fluctuations in the correlations between keys by continuously optimizing routing. When the workload is very volatile, it is important to avoid triggering reconfigurations for ephemeral correlations, as the cost of reconfiguring would not be amortized. As future work, we will design estimators able to predict the impact of a reconfiguration to provide more fine-grained information to the manager. Another promising area of research is adapting this approach to hierarchical network structures. Instead of having a binary model in which keys are co-located or not, distances between servers can be taken into account to leverage rack locality when load balancing prevents server locality. This could be done by using hierarchical clustering, similarly to [31]. This however requires a larger testbed for validation.

Chapter 5

# λ-blocks

*« Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. »*

Donald Knuth [78]

*« The best code is no code at all. Every new line of code you willingly bring into the world is code that has to be debugged, code that has to be read and understood, code that has to be supported. »*

Jeff Atwood [29]

## Contents

We've seen the integration of an end-to-end data processing system, and an optimization of a low level routing algorithm. Our third contribution is positioned higher in the data processing stack: it concerns the programming abstractions available to developers when writing data processing computations, and integrates well with lower level systems and frameworks.

For that purpose, we present and evaluate $\lambda$-blocks[1], a novel framework to write data processing programs in a descriptive manner. The main idea behind this framework is to separate the semantics of a program from its implementation. For that purpose, we define a data schema, able to describe, parameterize, compose, and link together blocks of code, storing a directed graph which represents the data transformations. Along this data schema lies an execution engine, able to read such a program, give feedback on potential errors, and finally execute it. In our reference implementation, a computation graph is described in YAML, linking together vertices of Python code blocks defined in separate libraries.

The advantages of this approach are manyfold: faster, less error-prone programming; reuse of code blocks; language- and framework-agnostic representation of data processing programs; computation graph manipulations; mixing of different specialized libraries; and finally middleware for potential front-ends (such as graphical interfaces) and back-ends (other execution engines). We notably aim to bring complex data processing computations to non-specialists.

Our contributions lie within a description of the schema, and an analysis of the reference execution engine. For that purpose we describe $\lambda$-blocks internals, show some applications and evaluate the framework performances. We measured the framework overhead to have a maximum value of 50 ms, a negligible amount compared to the average duration of data processing jobs.

## 5.1 Introduction

Within many frameworks and systems, data analysis can be summed up to a set of high-level operations: connect to a data store; fetch, clean and transform data; save the obtained result. Data is flowing from one operator to another, and the program can be easily represented with a directed graph, where vertices are operators and edges connect them together. For example, Apache Storm [111] allows to explicitly define such a graph when linking together its agents (spouts and bolts), and Apache Spark [119] automatically builds a lineage graph inferred from the successive methods called on its

[1] $\lambda$-blocks has been released under the Apache License version 2.0, and is available at `https://github.com/lambdablocks/lambdablocks`.

data structures (resilient distributed datasets). Many operations have been standardized in the fields of relational algebra or functional programming: *map*, *reduce*, *filter*, etc. Specialized libraries apply these operations to different data containers, sometimes on distributed clusters of machines, with different levels of optimization.

We argue these programs can be written in a higher-level fashion. By writing one or more of these operations in a "code block", we abstract out the functional code of this block, in the same manner than a library function. Having inputs and outputs, a block can then be a vertex of an oriented graph, which can be reused in different computations.

We propose to write such a graph in a descriptive fashion, rather than programmatic. Using for example YAML, a data serialization format particularly easy to read and parse, we can describe the vertices (linked to code blocks with unique names) and their edges (an edge exists when one block's output is another block's input). Moreover, a graph can itself be a sub-graph of another graph, leveraging code-reuse one step further. Some advantages of this approach include:

- strict separation of low-level data operations and high-level data processing programs;
- direct manipulation of the computation graph, for optimization, instrumentation, etc;
- reusability of code, with blocks being used in many programs, and computation graphs being composed of other graphs;
- easier reading, understanding, sharing, evolution and maintenance of a data processing program;
- seamless mixing of different frameworks and libraries together;
- as a middleware layer, room for front-ends such as graph visualization tools, and back-ends such as more optimized execution engines.

$\lambda$-blocks aims to bring framework-agnostic dataflow programming to large-scale data processing, without losing any of the benefits provided by specialized and well-optimized libraries implementing data operations.

The rest of this chapter is organized as follows: we first introduce some background on data processing with graphs and component-based software engineering, before diving into $\lambda$-blocks' design, its graph description format and its execution engine. We then present examples of graph manipulations and data processing applications, before evaluating the framework's performances. We finally introduce some related work and conclude.

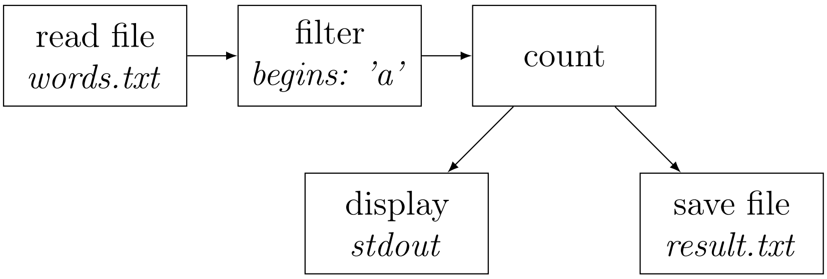


Figure 5.1 – A graph representing a program counting words in a file beginning with 'a'.

## 5.2 Background

### 5.2.1 Data processing with DAGs

Directed acyclic graphs (DAGs) are often used to represent a finished list of successive transformations to apply to input data in an efficient manner. Vertices represent transformations, while oriented edges represent the links between them, orchestrating the order in which the transformations must be applied. The field of dataflow programming deals with the abstractions behind this idea. In a dataflow-expressed program, the graph is not always explicitly written by the developer, but used as the internal structure. Famous examples include tabular spreadsheets, Makefiles, or large-scale processing frameworks such as Apache Spark and Apache Storm.

DAGs are an efficient data structure for representing a flow of data transformations, because it gives an abstraction over them. Moreover, a vertex can have one or many incoming edges, thus allowing the combination of different data sources together, and one or many outgoing edges, thus maximizing data reuse.

An example of a simple dataflow program is shown in Figure 5.1. In every vertex of this directed graph, an operation is shown, along with the arguments it takes (except for `count`, which doesn't take any). After a vertex has executed its function, it passes the result to the vertices that subscribe to it: for example, both `display` and `save file` read the result of `count` to act upon it. The complete program counts the number of words beginning with the character 'a' in the file words.txt, shows the result in the console and saves it in the file result.txt.

This example is as simple as it can be, but illustrates the principles of dataflow programming: it links together operators transforming data, and can be represented with boxes and arrows without losing its semantics.

### 5.2.2 Component-based software engineering

As introduced in Section 2.3.1, component-based architectures leverage the separation of a software system between different independent elements, called components. More specifically, their properties we're interested in mirroring in λ-blocks are:

- Black box: an observer doesn't need to know how a component works in order to use it.
- Clear interface: components provide one (or multiple) defined interface to interact with them, just as an object does in object-oriented programming. A component interface can be for example defined by its parameters, inputs and outputs; and the output of one component can be the input of another one.
- Standalone: a component doesn't need other components in order to work, and can be used in different systems, given its interfaces are honored.
- Composition: a component can be composed of multiple smaller components, leveraging code reuse.

Components bring a clear separation of concerns to a system, and compartmentalize its different actors. Many component frameworks embed a domain-specific language to describe the different components and how they interact with each other. Some of these ideas are borrowed in λ-blocks, but for a different kind of system: whereas components are different actors communicating with each other in both directions to transmit messages and instructions, in λ-blocks the components are only operators, transmitting transformed data in one direction. The similarities comprise of the description of actors and their relationships with a domain-specific language (when they are themselves written in classical programming languages), the separation of concerns, and the room for optimization which naturally exists when accessing a high-level view of such a system.

## 5.3 λ-blocks

### 5.3.1 Terminology

We describe here the different abstractions used in λ-blocks. These objects need to exist in any engine implementing the system:

- **Block**: a standalone piece of code, able to provide information about its behavior, either using introspection (with languages supporting it) or embedded declarations (e.g. through decorators or class attributes). A block needs to describe at least its ports and arguments (explained below).

- **Component**: an *instance* of a block, i.e. a block with its arguments, ready to run its functional code.

- **Port**: a named input or output of a block. The inputs are the data provided to the block as arguments, and the outputs are its results.

- **Argument**: a runtime option of a block, such as a configuration value. It is different from an input port, because it is not meant to carry flowing data, but rather a variable to parameterize the block, initialized in the topology.

- **Registry**: a catalog of blocks, providing their functional code along with their metadata (a list of key-value pairs used to classify blocks) and documentation.

- **Topology**: a computation graph, i.e. a high-level representation of a data processing program, defining components and linking them together as a DAG. Sometimes simply called graph.

- **Sub-topology**: a topology used as a component of another topology (i.e. when using a graph as a vertex of a bigger graph). Sometimes called sub-graph.

### 5.3.2 Architecture

Figure 5.2 shows the architecture of the system. Its different components are as follows:

- The graph engine is the main controller, it is responsible for parsing topologies, matching the vertices with code blocks, building the corresponding graph, running different checks against it, and finally executing it.

- Graph plugins can add functionalities, such as graph manipulation, instrumentation, etc. They are plugged to the graph engine.

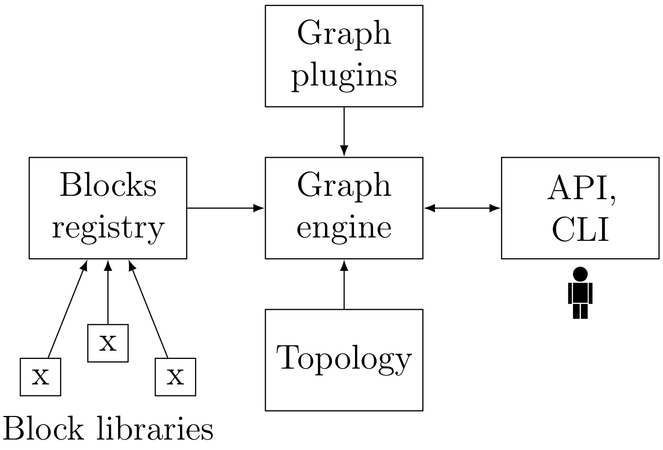


Figure 5.2 – System architecture.

- The blocks registry aggregates the code blocks defined in different blocks libraries, and extracts their metadata, either through introspection or through decorators or tags.

- A topology is the main input of the system, defined in one or more files.

- Finally, the system can be driven by an API and a CLI, and more front-ends can be plugged, for instance graphical interfaces.

We kept the architecture modular, so that it is easily extendable, and the different components can be replaced independently; this can be useful for instance to extend the supported description and programming languages (beyond YAML and Python).

### 5.3.3 Topologies format

The main design goals of the topology schema are simplicity and extensibility. It is meant to be easily written by hand, even by non-programmers, who would simply need to know the high-level concepts of data transformations and the YAML syntax.

YAML is a data-serialization language, focusing on readability. It can easily define lists and associative arrays, and supports typing. We chose it for the simplicity of reading and writing data with it, and because it is fast to parse.

There are two types of objects in a topology schema, *blocks* and *sub-topologies*, described below.

### Defining and linking components

A topology consists of two YAML sections: the first one is a simple dictionary, which permits to assign a name and various metadata to the topology. It is useful when the number of maintained topologies grows within an organization, and it permits to easily retrieve them through a search engine for example. No key is mandatory in this first section, except the `name` when this topology is to be composed with other ones, as it needs a unique identifier.

The second section lists the different components and links them together. Some keys are defined in the reference implementation, and it is easy to add new ones to further customize the topology. These keys are:

- `block`: the name of the block (from the blocks registry) that is to be used;
- `name`: a unique name for this component;
- `args`: optional, it allows to give arguments to the block, to customize its behavior;
- `inputs`: absent for entry-level blocks (the data sources), it permits to link components together.

Both `args` and `inputs` are defined with a dictionary, because they are always explicitly named. A block can have zero or more inputs, and zero or more outputs. At the topology level, only the inputs are defined; its outputs are inferred from the other blocks consuming them, and can sometimes remain unused (if no other block subscribes to them).

Listing 5.1 shows a simple topology, which counts the users of a Linux-based system (note it is incorrect since it will also count daemons and other system users, but this is out of the scope of the example). The first block will read a file line by line, and it knows which file to open through the `args.filename` value. The second block will simply count the length of the data structure it receives: it has one input, named `data`, which is linked to the output `result` of the named block `my_readfile`. This block produces another result, accessible through `my_count.result`, which could be displayed on a console, saved to a file, or used as an input of other different blocks.

### Encapsulating other topologies

The second type of component which can be defined in a YAML topology is a topology itself, encapsulated and linked to blocks or other sub-topologies. This poses a few challenges, mainly to keep the outer links simple to define.

```
---
name: count_users
description: Count number of system users
---
- block: readfile
  name: my_readfile
  args:
    filename: /etc/passwd

- block: count
  name: my_count
  inputs:
    data: my_readfile.result
```

Listing 5.1 – A simple topology.

An example is shown in Listing 5.2, along with its graph representation in Figure 5.3. On the left column, the main topology is defined, and it instantiates a sub-topology block, `count_pb`, and linked to two blocks, `my_file` and `my_print`. The `bind_in` dictionary permits to give inputs to this sub-topology, while `bind_out` permits to use some of its outputs as inputs to other blocks. The sub-topology is displayed on the right column.

Any block output can be used as a sub-topology input, and once they are defined, they are accessible in the encapsulated topology through the special dictionary `$inputs`. We keep the $ sign as the only reserved symbol, which could be used in the future for different conveniences, for example to give command-line parameters to a topology.

Similar to the bound inputs, any output of any block of the sub-topology can be linked to a block. There is no need to use a special dictionary for this purpose, since outputs are never explicitly declared. In this example, the value `count_pb.result` means the output `result` of the block `count` of the encapsulated topology `count_pb`.

The result of this is a program filtering errors in a log file, counting them, and displaying the sum. It makes a great use of encapsulation, because the sub-program taking care of filtering and counting could be used in other topologies, for example with other log files as bound input, or with a block saving the result in a database as bound output.

```
---
name: foo_errors
---
- block: readfile
  name: readfile
  args:
    filename: foo.log

- topology: count_pb
  name: count_pb
  bind_in:
    data: readfile.result
  bind_out:
    result: count.result

- block: print
  name: print
  inputs:
    data: count_pb.result
```

```
---
name: count_pb
---
- block: filter
  name: filter
  args:
    contains: error
  inputs:
    data: $inputs.data

- block: count
  name: count
  inputs:
    data: filter.result
```

Listing 5.2 – Encapsulation example. Left: the main topology; right: the encapsulated topology. Reports the number of errors found in a file `foo.log`.

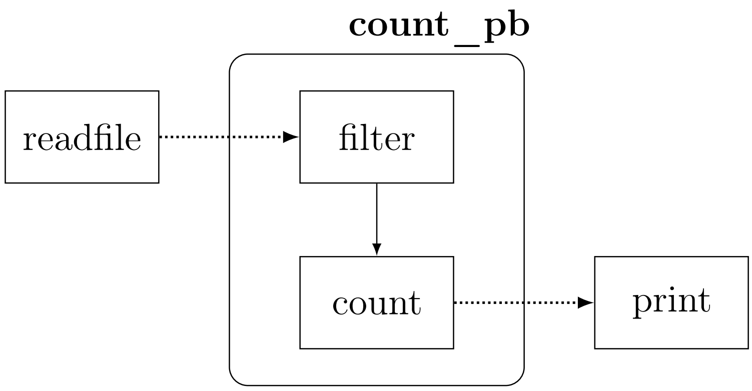


Figure 5.3 – The DAG associated to the program counting errors. The *bind_ in* and *bind_ out* links for the sub-graph are dotted.

### 5.3.4 Block internals

Blocks are the individual components of the topologies. They are independent and reusable: they know nothing about a data processing program, except their inputs and outputs. Most of them only take care of transforming data, and hence don't have side effects, in a purely functional manner. Some blocks read data from stores (they are the entry points of the computation graph), and some save back their results on storage. There is no restriction about the programming language used to write the blocks, as long as they can be called from the engine manipulating them. We chose Python for the reference implementation for a few reasons:

- Simplicity: it is a design goal of λ-blocks to be as simple as possible, and Python has been known for being very accessible to novice programmers.

- Variety of libraries: since blocks can wrap any library function, Python is a good choice for combining distributed computing (for example through pyspark, the Python package to interact with Spark), machine learning (MlLib, scikit-learn), plotting (matplotlib), etc.

- Introspection: it is straightforward to inspect functions in Python, hence the engine can infer a lot of metadata about a block (its arguments, inputs, outputs, and documentation) without them being declared explicitly. Any other metadata can be added with function decorators.

Two block examples are shown in Listing 5.3. A block named `take` is registered through the `@block` decorator, which takes any pair of key/value for tagging it, for example to categorize it. We then create a closure: the outer

```
@block(engine='localpython')
def take(n: int=0):
    """
    Given a list of integers, returns the n
    first items.
    """
    def inner(data: List[int])->ReturnType[List[int]]:
        assert n <= len(data)
        return ReturnEntry(result=data[:n])
    return inner

@block(engine='spark')
def spark_distinct(numTasks=None):
    """
    Spark's distinct
    """
    def inner(data: pyspark.rdd.RDD) \
          ->ReturnType[pyspark.rdd.RDD]:
        o = data.distinct(numTasks)
        return ReturnEntry(result=o)
    return inner
```

Listing 5.3 – Two typical block structures.

function takes the block's arguments, while the inner function takes the block's inputs. We use Python's type annotation capabilities to give types to the arguments, inputs and outputs. The special `ReturnType` and `ReturnEntry` give us the ability to properly define the block's outputs, to overcome some limitations of the dynamic manipulation of Python's typing annotations. This way, the arguments, inputs and outputs can all be documented and verified.

What happens in the inner function is the responsibility of the block developer, and can be anything Python can do, such as direct data manipulation, library function wrapping, or input/output (to retrieve and store data).

The second function, `spark_distinct`, is a wrapper around the Spark method which implements *distinct* on an RDD. Writing a wrapper around other library functions allows to use those in λ-blocks along with the other blocks.

### 5.3.5 Execution engine

The execution engine is the glue between the topologies and the Python blocks of code. Upon initialization, it will parse a topology, build the associated DAG (recursively when sub-topologies are involved), and associate each vertex with a named block of code. It can then run some DAG manipulations, do all the necessary checks, and execute the graph, giving each component its inputs after they have been computed.

The execution engine must be fast, to reduce the overhead of the system as much as possible, and also easy to extend, to leave the possibility of adding graph manipulations. For the latter, atop the internal API to manipulate edges, vertices, and the engine itself, it provides a plugin system, which makes it easy to register hooks at the different steps of the graph execution. When these hooks are called, they receive relevant parameters, such as the current value of the flowing data for a certain block. Some possible manipulations are described in the next section, showing both the use of the internal API and the plugin system.

In the reference implementation, the engine is single-threaded. However, each component can easily leverage parallelization by spawning multiple threads. While this is not optimal for building a proper distributed system, it is not λ-blocks' role: distributed data processing frameworks such as Apache Spark do it better. λ-blocks is meant to wrap their instructions in order to combine their benefits with its own.

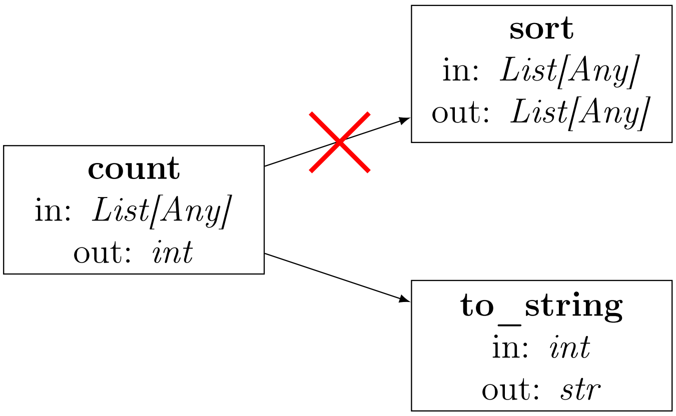


Figure 5.4 – Type checking.

## 5.4 DAG manipulations

As stated earlier, having a high-level representation of the processing graph can bring many benefits, among them the possibility of reasoning on and optimizing a data processing program. We describe two examples we implemented, and it is easy to build more using λ-blocks' internal API or plugin system.

### 5.4.1 Type checking

Python does not benefit from compile-time type safety. However, it supports type annotations for variables and functions, and these annotations include base types as well as more complex ones, such as generic lists and dictionaries, unions, callables, etc. Type checking can only happen statically, with the Mypy [74] static analyzer. The links between the different blocks being computed dynamically (from their YAML description), we implemented a type checker, which runs right after the DAG construction. The types of every vertex input and output have already been introspected, hence it is enough to check that the types of both ends of an edge are compatible.

An example of type checking is shown in Figure 5.4. The block `count` takes a list of elements, and returns the length of the list, as an integer. This is fine for the block `to_string`, which takes an integer as an input, however this doesn't make sense for the block `sort`, which takes a list as an input. The type checker, when verifying the edge between the blocks `count` and `sort`, will see two incompatible types at its extremities, and will raise an error.

This process was easily implemented thanks to the high-level DAG manipulation features λ-blocks provides: iterating through vertices and edges, accessing blocks' details, and accessing the registry's objects (functions, input types, etc). This feature is useful on its own, but can also be leveraged when writing a graphical interface: an edge could simply not be created between two vertices if their types were not compatible. This reduces potential errors while writing data processing programs, giving an immediate feedback to the user.

### 5.4.2 Instrumentation

DAG manipulations can also be dynamic and happen at execution time. For example, it is easy to instrument a program by measuring the time taken by every of its components to execute. We implemented a plugin for that purpose, using three hooks:

- before block execution: stores a timestamp associated to this block;
- after block execution: computes the time it took for the block to execute, using the previous timestamp;
- after graph execution: sorts and displays all the recorded durations.

This can be done with a few lines of Python, and shows the easiness with which one can add features to λ-blocks, when reasoning with a computation graph as a top level object. An excerpt of the plugin code is shown in Listing 5.4. The hooks are registered with the Python decorators `@before_block_execution`, `@after_block_execution`, and `@after_graph_execution`, and the plugin is activated with the help of the plugins manager, through command-line parameters. The first hook is called before each block is executed, and stores a timestamp. The second hook is called after each block is executed, and with the help of the first timestamp stored, it can deduce the execution time for every block. We simply display the summary for every block with the third hook, once the whole DAG has finished its execution.

An example of this plugin output is shown in Table 5.1 (durations and timestamps have been truncated to the millisecond):

This example is simple: it downloads a list of words over http, splits the result to generate a list of words, filters them to keep only those containing the letter 'e', and stores the result in a file. Unsurprisingly, the network call is the longest, followed by the block writing the result on the disk.

```python
by_block = {} # timing by block: begin, end, duration

@before_block_execution
def store_begin_time(block):
    name = block.fields['name']
    by_block[name] = {}
    by_block[name]['begin'] = time.time()

@after_block_execution
def store_end_time(block, results):
    name = block.fields['name']
    by_block[name]['end'] = time.time()
    by_block[name]['duration'] = \
      by_block[name]['end'] - by_block[name]['begin']

@after_graph_execution
def show_times(results):
    longest_first = sorted(
        by_block.keys(),
        key=lambda x: by_block[x]['duration'],
        reverse=True)
    for blockname in longest_first:
        print('{}\t{}\t{}\t{}'.format(
            blockname,
            1000 * by_block[blockname]['duration'],
            by_block[blockname]['begin'],
            by_block[blockname]['end']))
```

Listing 5.4 – Excerpt of the instrumentation plugin.

Table 5.1 – Block instrumentation results.

| block | duration (ms) | begin | end |
|---|---:|---:|---:|
| read http | 818 | 1509717620.416 | 1509717621.235 |
| write lines | 54 | 1509717621.305 | 1509717621.360 |
| grep | 49 | 1509717621.256 | 1509717621.305 |
| split | 20 | 1509717621.235 | 1509717621.256 |

### 5.4.3 Debugging

Another dynamic DAG manipulation implemented with hooks is the debug plugin: it prints the output of a block after it has executed, which gives the developer an easy way to follow the transformation of data and spot mistakes. It could easily be leveraged in a graphical interface, giving instant feedback about misconfigured blocks or broken graphs. For that purpose, it is enough to register a hook `@after_block_execution`, which receives a block and its results as parameters. The only difficulty is to display a sample of the result, to give a concise idea of the transformation being done.

Using the same example as in the previous section, we obtain:

```
For block read_http
  result
    "2\n1080\n&c\n10-point\n10th\n11-point\n12-point\n...

For block split
  result
    ['2', '1080', '&c']

For block grep
  result
    ['Aaberg', 'Aachen', 'aahed']

For block write_lines
  No results for this block.
```

Each block (except the last one) has only one output field, named `result`. We can see how the DAG transforms a string (the answer of an http request) containing newline characters, into a list of words, into a list of words containing 'e'.

### 5.4.4 Other graph manipulations

The examples above showed the advantages gained when developers can manipulate their programs, and plug hooks into their execution paths. We list some other possibilities this API brings:

- **Optimizations**. Analyzing the graph could reveal patterns which have room for optimization, for example two successive *map* functions can be combined together.

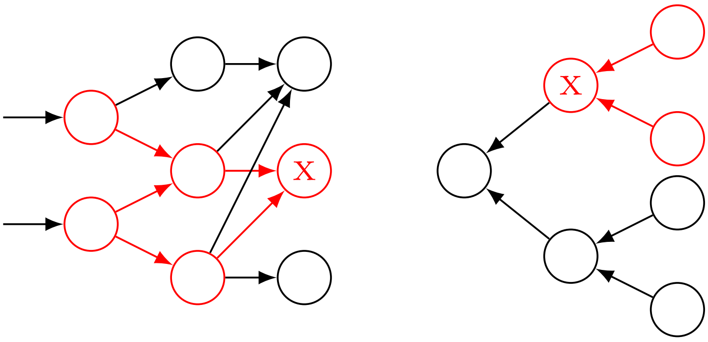


Figure 5.5 – Signature dependencies for a directed graph and a Merkle tree.

- **Complexity analysis**. Predicting the complexity of an execution graph, and mixing it with the input size, could lead to interesting execution time estimations.

- **Reasoning**. Having a high-level view of an execution graph opens the door for automatic reasoning about the program: is it trying to count, to filter, to aggregate? It would be interesting to classify programs according to their type or the presence of recurring patterns.

- **More developer tools**. The API makes it easy for developers to add and use plugins, which they can adapt for their uses: automatic sample of the input data, caches (see Section 5.5), advanced debugging tools tailored for their specific data types, etc.

## 5.5 Caching/Memoization

In order to cache certain computations, we implement a signature algorithm, whose role is to uniquely identify a block instance within a DAG. To define the uniqueness of a block instance, we declare that two blocks instances are the same if:

- They are instances of the same block (identified by a unique name).

- They have the same parameter values.

- They have the same inputs.

Our signature algorithm is similar to a Merkle tree [89], in that a block signature recursively depends on the signatures of its inputs.

Figure 5.5 shows the similarities shared in the signature dependencies for a directed graph and a Merkle tree. The hash (signature) of a node depends on the hash of all the nodes pointing to it (the successors, in a tree). We show in red the nodes whose hash the node $x$ is dependent on to compute its own hash.

We define a unique signature this way. Let $H$ be the signature function, $B$ a block instance, and $h$ a secure hash function:

$$\begin{aligned} H(B) = h(&B.\mathit{name}, && \text{block name (not instance name)}\\ &B.\mathit{args}, && \text{list of (name, value) tuples}\\ &B.\mathit{inputs}) && \text{list of (name, H(block), connector) tuples}\end{aligned}$$

$B.\mathit{args}$ is the list of arguments and the block accepts, along with the values provided. To use a canonical version of this list, we order it alphabetically by argument name.

$B.\mathit{inputs}$ is the list of inputs the block accepts, where each item of this list is a tuple containing the input name, the signature of the input block, and the name of the port this input corresponds to. This is where the recursion applies.

With this function, if an input changes in any predecessor of the block, it will have a different signature. This gives us a way to associate a block instance with a key, which can be used in a caching mechanism. With that in place, two programs which share the same inputs and possibly input transformations will benefit from the cache and improve their processing times.

We implemented the signature function in λ-blocks using `SHA-256` as the hash function, along with a cache provider abstract class, meant for plugins to build upon. We also wrote a plugin that implements a cache on disk, marshalling the Python results for that purpose using the `pickle` library.

It is important to note that this doesn't replace the caching mechanisms present in frameworks such as Spark, which are better optimized for their data types and applications, and as such recommended to be used in place of λ-blocks' caches.

## 5.6 Examples

### 5.6.1 Wordcount

As a first example, Listing 5.5 shows a traditional wordcount program: its goal is to output the 5 most used commands (without their parameters) in a

```
---
name: zsh-history-wordcount
description: The top 5 zsh commands
---
- block: cat
  name: readfile
  args:
    filename: /home/foo/.zhistory

- block: cut
  name: cut
  args:
    sep: ' '
    fields: [1]
  inputs:
    data: readfile.result

- block: group_by_count
  name: reduce
  inputs:
    data: cut.result

- block: sort
  name: sort
  args:
    key: "lambda x: x[1]"
    reverse: true
  inputs:
    data: reduce.result

- block: head
  name: head
  args:
    n: 5
  inputs:
    data: sort.result

- block: show_console
  name: show_console
  inputs:
    data: head.result
```

Listing 5.5 – Wordcount over a terminal command history.

terminal emulator, along with their number of occurrences. For that purpose, we apply these actions:

- read the zsh history file (block `cat`);
- extract the first field of each line, the program name (block `cut`);
- group every program name together, and count their occurrences, similar to SQL's *SELECT COUNT(word)... GROUP BY word* (block `group_by_count`);
- sort the result according to the second field (the number of occurrences), in reverse order (block `sort`);
- take the first 5 results (block `head`);
- finally display the result to the user (block `show_console`).

Most of these block names are similar to UNIX commands, because they do the same job (on Python datastructures rather than text streams), and come from the blocks library *unixlike.py*, bundled with $\lambda$-blocks. After executing this program, we obtained:

```
[('git', 506),
 ('rgrep', 125),
 ('cd', 121),
 ('less', 117),
 ('sudo', 89)]
```

This example shows the simplicity with which one can implement a data processing program, by linking pre-written blocks with each other. While in essence it carries the same ideas than writing a regular program using libraries, the main difference is that we have in the resulting YAML file a different representation of the program, a graph which can be manipulated and optimized by any engine following its conventions. Moreover, it shows a clear separation of concerns: the semantics of a data processing program on one side, and the building blocks which actually transform the data on the other side. Such a graph, which is language-agnostic (except for the lambda function), can easily be written by any tool speaking YAML.

### 5.6.2 Twitter API and encapsulated wordcount

The previous example showed a computation graph counting the occurrences of commands in a terminal session history. Its input was a text file, and its output the user console. The abstraction can go further, by encapsulating the actual wordcount computation (group by, count, sort, head) in a sub-graph, which can then act as black-box, accepting any list as input, and giving as output a list of (item, count), like shown in Figure 5.6.

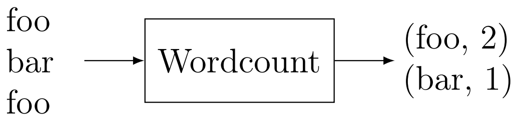


Figure 5.6 – Wordcount as a black box.

To implement this, we separate the blocks doing the wordcount computation in a different file, which will be used as a sub-graph. In order to be referenced later, we give it a unique name, `topology-wordcount`. Listing 5.6 shows a use-case of this sub-graph, while Figure 5.7 shows its representation.

Its goal is to compute the list of the most used hashtags on a news timeline, AFP in this example. For that purpose, it uses a block `twitter_search`, whose responsibility is to authenticate to Twitter with the provided crendentials, submit an http query, parse the JSON response to a Python datastructure, and return it as its result. This job is abstracted in the block source code, which makes it easy to further analyze any Twitter query using it as a data source.

The next challenge is to extract the hashtags from the list, which contains all kinds of metadata along with the tweet content. This is done with a `flatMap`, a block that will apply a map function and then flattens the result into a one-dimensional list: this is necessary, because a tweet may contain zero, one, or many hashtags.

Once this is done, our data stream has the structure expected by the wordcount sub-graph: a list of words. We use the `topology` entity, binding-in this list, and binding-out the result, which we display as usual. We obtain the following list, showing the 5 most trending tweets on the AFP timeline:

```
[('BREAKING', 10),
 ('UPDATE', 9),
 ('AFP', 5),
 ('AppleEvent', 2),
 ('Airbus', 1)]
```

```yaml
---
name: twitter-wordcount
description: Extract the most used hashtags on the
             recent AFP timeline.
---
- block: twitter_search
  name: twitter_search
  args:
    query: "from:afp"
    client_key: xxx
    client_secret: xxx
    resource_owner_key: xxx
    resource_owner_secret: xxx

- block: flatMap
  name: extract_hashtags
  inputs:
    data: twitter_search.result
  args:
    func: "lambda x: [y['text'] for y in \
             x['entities']['hashtags']]"

- topology: topology-wordcount
  name: wordcount
  bind_in:
    data: extract_hashtags.result
  bind_out:
    result: head.result

- block: show_console
  name: show_console
  inputs:
    data: wordcount.result
```

Listing 5.6 – Wordcount over the most recent hashtags from the press agency AFP, using a sub-graph.

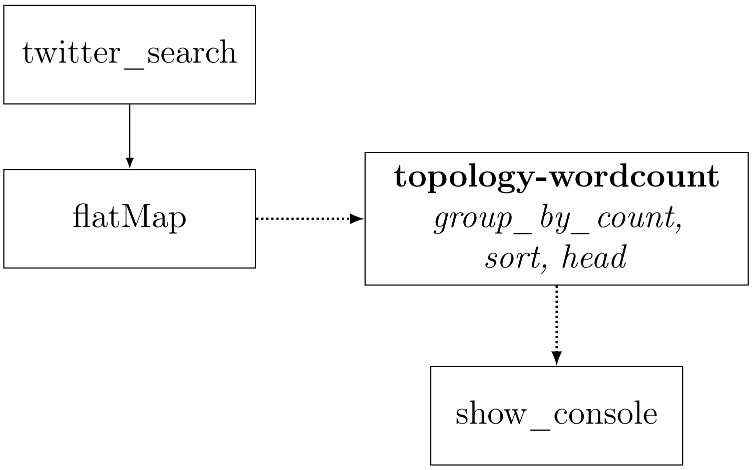


Figure 5.7 – Representation of the Twitter Wordcount with a topology-wordcount sub-graph.

The sub-graph `topology-wordcount` acted as a black-box, effectively counting unique words and giving their number of occurrences. It can be used with any input and its output can be fed anywhere, as long as the data structures are respected and the bindings correctly done.

## 5.7 Evaluation

### 5.7.1 Performances

The first metric we want to calculate is the overhead of using λ-blocks' engine, compared to a regular Python program. We run 3 different programs, and compare their execution times on different setups:

- with λ-blocks;
- with λ-blocks and two plugins: *debug* and *instrumentation*;
- without λ-blocks, writing the equivalent code in a regular Python fashion.

The three programs we run show different patterns of latency and complexity:

- Wordcount on trending Twitter hashtags: we run the example shown in Listing 5.6, which extracts hashtags from the Twitter API, and groups and counts them in a wordcount sub-graph. This program has a non-negligible network overhead, since it needs to wait for the http queries to complete before continuing. It is single-threaded.

- Wordcount on a local file (without network queries), over a Wikipedia dataset [12] which we trimmed to consist of 10 million words. This dataset is an HTML dump of the English version of Wikipedia. The program is very similar to Listing 5.5, and is single-threaded.

- PageRank on an Apache Spark cluster, over a dataset of internal Wikipedia hyperlinks [118]. For this purpose, we use the Spark wrapper code blocks in $\lambda$-blocks' `lb.blocks.spark`. The DAG has two entry points (the file containing the links, and the one containing the page names, both stored in HDFS), and the blocks use various Spark functions. The YAML code for this program is shown in Appendix A. It is run on a bare-metal Spark server.% with 24 CPU cores and 1GB of RAM.

To obtain more precise results, we run the Twitter Wordcount 10 times for each setup (with the program latency, that's the limit to not reach the API rate limits), the Wikipedia Wordcount 1000 times, and the Spark PageRank 10 times. We kept the average of the obtained values as our reference.

The times are measured with `/usr/bin/time -p`: *real* is the time taken by the program to complete; *user* is the CPU time consumed in user mode by the program, and *sys* is the CPU time consumed in kernel mode. If *user* and *sys* don't add up to *real*, it means the program was blocked during execution, generally waiting for disk or network.

Figures 5.8, 5.9 and 5.10 show the results obtained. The first program, in Figure 5.8, confirms there is indeed a network overhead, during which the program is waiting (in the three cases), and is not consuming CPU cycles. More importantly, by substracting times, we measure the overhead of using $\lambda$-blocks: about 40 ms per run (we see in Section 5.7.2 how it can be reduced further). The last interesting point is the negligible difference between $\lambda$-blocks with and without plugins: inspecting the graph vertices and instrumenting their computation times comes almost for free (the difference is smaller than the standard deviation of multiple runs of the same setup).

Figure 5.9 shows almost the same execution times for the three setups, and the first one (with $\lambda$-blocks) is even faster. This is not supposed to be the case, because it executes more code by design, but comes from the non-determinism of disk (and kernel cache) input/output: the speed varies with time, hence the imprecision of the calculation. The key point here is

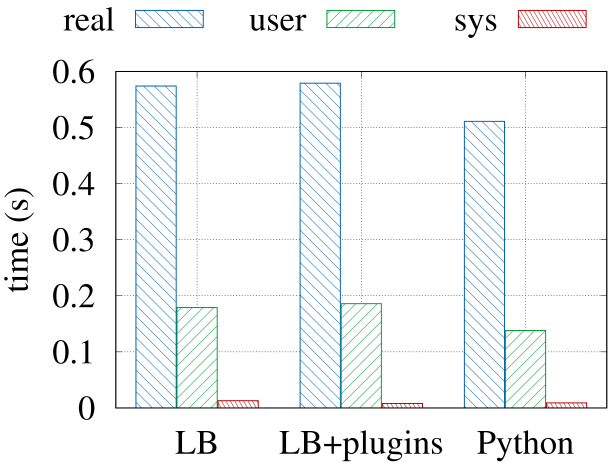


Figure 5.8 – Twitter hashtags Wordcount.

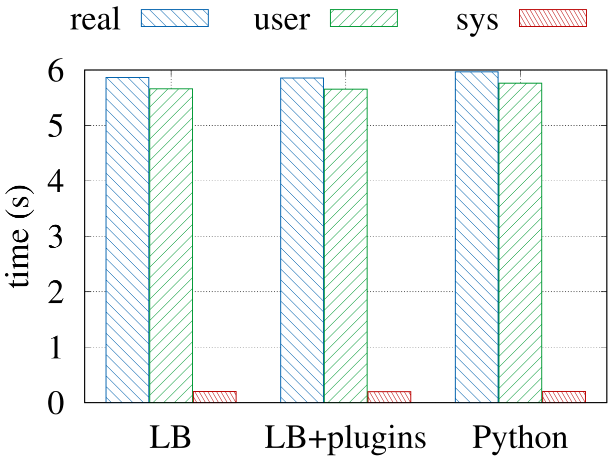


Figure 5.9 – Wikipedia file Wordcount.

that using λ-blocks doesn't add any significant overhead if the job runs over a few seconds.

Finally, Figure 5.10 depicts the execution times for the PageRank computed with Spark. The first thing to notice is the low values of *user* and *sys* times, too low to be visible on the plot. This time it is not due to the majority of the program waiting for IO, but rather because the programs are communicated to and executed by a Spark daemon. Hence the CPU times are not seen by `/usr/bin/time`. However the *real* times are correct, and we can use them to compare the setups. Similar to the previous experiments, using λ-blocks doesn't add any significant overhead; and for such a job duration (about 14 minutes), it is negligible. Hence λ-blocks can easily drive a Spark

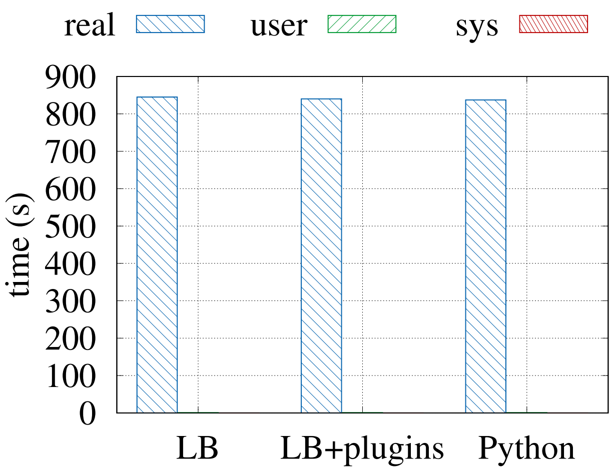


Figure 5.10 – Wikipedia hyperlinks PageRank.

program at a low cost.

### 5.7.2 Engine instrumentation

In order to further reduce the overhead added by the use of $\lambda$-blocks' engine compared to writing regular Python programs, we instrument the framework when running the Wikipedia Wordcount described above. We used a smaller input file in order to have a total execution time comparable to the measured overheads. We instrument three different setups, only changing the command-line parameters of $\lambda$-blocks:

- Loading all the block modules, and two plugins;
- Loading only one block module, and two plugins;
- Loading only one block module, without any plugin.

Figure 5.11 shows the results we obtained, with the different steps followed by $\lambda$-blocks : (1) Python startup, modules import and arguments parsing; (2) Blocks registry creation, block modules import; (3) Plugin import; (4) YAML parsing and graph creation; (5) Graph checks; (6) Graph execution.

We note interesting results:

- Importing and executing plugins doesn't add any visible overhead, which confirms the results described in Section 5.7.1.

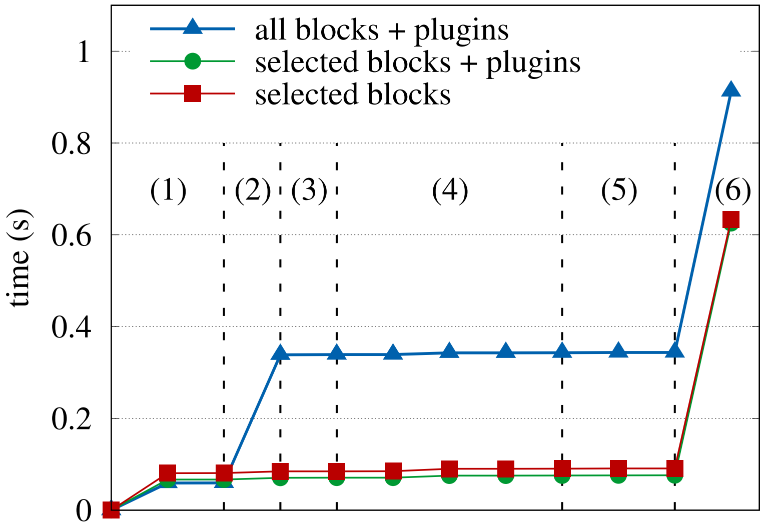


Figure 5.11 – Instrumentation of a Wordcount program running under different setups.

- Importing all the available blocks in the built-in block modules is very costly: between 250 and 300 ms. Python's import mechanisms are known to being slow [13]. This is why we only imported selected block modules in the previous section, to obtain a smaller overhead compared to a standalone Python program.
- Building the computation graph, and running checks against it (correct YAML, type checking, absence of loops, etc), is very fast; this is encouraging to develop more graph manipulation plugins.

Overall, the best optimization we found is to avoid importing block modules if they are not to be used. We target our future work to develop a λ-blocks daemon, in order to load modules only once and execute many computation graphs on demand.

## 5.8 Related work

We introduced related tools in Section 2.3. We compare them against λ-blocks to highlight their differences, and introduce a few others.

*Blocks-based programming* has gained a lot of attention recently [34]. λ-blocks shares similar ideas, for example combining chunks of embedded code to create larger programs. However, block-based graphical interfaces are not oriented towards large scale data processing, and hence do not benefit from distributed libraries such as Apache Spark. We plan to explore some of their innovative features such as *recognition over recall* [34], immediate feedback, and impossibility to link blocks that don't make sense together. We believe it is a path towards bringing data processing and analysis to non-programmers.

*Graphs from configuration* is not a novel concept either. Pyleus [117] and later Storm Flux [110] brought configuration-based topologies to the Apache Storm [111] framework, for stream-processing. They both use a YAML format to define topologies, and inspired $\lambda$-blocks. However, they are limited to link spouts and bolts, the base objects of Storm, which are meant to process online data.

*Dataflow programming with pipelines* has been implemented in numerous frameworks. For machine-learning applications for example, scikit-learn [97] and Apache Spark [109] have a built-in concept of Machine Learning Pipelines, where different data processors are defined and linked with each other. However, the DAGs are created programmatically in their respective library languages, and they are limited to the components of their frameworks.

The Orange framework [57] features a collection of widgets, linkable with each other, to execute and visualize machine learning algorithms. Like $\lambda$-blocks, it has a programming interface to implement new widgets in Python. However, it is specialized in data mining, data exploration and machine learning, and to the best of our knowledge it can't be distributed on many machines.

StreamPipes [101] is a framework for building and executing data stream pipelines, oriented towards distributed real-time processing of data, between sources and sinks. Some of their ideas are similar to those of $\lambda$-blocks (type checking (and other verifications) between operators, independent and self-contained blocks of code), but the framework differs with regard to some design choices: stress towards wrapping external computing engines, formalization of message passing between operators with data serialization formats (no possibility to use direct memory transfers), and RDF as the description and configuration language. Finally, it is oriented towards real-time data, whereas $\lambda$-blocks focuses on offline analysis for now.

Cascading [3] is a layer on top of Apache Hadoop. It permits to programmatically describe MapReduce jobs, by linking components together (sources, pipes, and sinks) in any JVM-based language. It has some similarities with $\lambda$-blocks, but also different goals: restricted to Hadoop and Flink, no graph manipulation, no configuration-oriented description of jobs.

KeystoneML [106] is a framework written in Scala, leveraging the use of high-level operators to build machine learning pipelines. Like the other introduced frameworks, it has different goals than $\lambda$-blocks', but shares the dataflow design and the reuse of components.

Apache Beam [1] also implements pipelines, and is able to run them on different processing engines, including Apache Flink and Apache Spark. Its goals are not exactly the same than those of $\lambda$-blocks, as its main feature is to define a pipeline and have it executed on different runners; whereas $\lambda$-blocks focuses on the expressiveness of programs and their manipulation as graphs. We intend however to study the deployment of a program on different executors, as $\lambda$-blocks would highly benefit from such a feature.

Other related tools [2,6,10,11,15] exist, but to the best of our knowledge, none implements all the features of $\lambda$-blocks, in particular the DAG specifications and the high-level graph manipulation abstractions.

A recent study [73] describes a new way of implementing large-scale data transformations, using serverless architectures and stateless functions, such as AWS Lambda on the Amazon public cloud. Using remote containers to execute stateless functions, along with a distributed storage engine, can reduce the complexity of distributing work on a cluster, by removing many of the complex tasks (such as cluster configuration or task deployment). Although $\lambda$-blocks' approach is not on the same layer, it made us more confident that implementing easy-to-use engines as a proxy between cluster systems and developers is a way to greatly speed-up the writing of large-scale data analysis programs.

## 5.9 Conclusion

We presented $\lambda$-blocks, a framework which permits to define execution graphs for large scale data processing, combining blocks of code with high-level manipulable directed acyclic graphs. Blocks are linked together, and the data flows from the inputs to the outputs, transformed at every step. This approach permits to combine different data processing frameworks, to represent the high-level steps of an algorithm without code, and gives a way to manipulate a program by changing its graph representation, or triggering actions when it is executed, through the use of plugins. We described a reference implementation of $\lambda$-blocks, which uses Python as the language for blocks and YAML as the data-serialization format for topologies. We explained the design choices, the system internals, and showed some practical examples of topologies. $\lambda$-blocks has a very small overhead with regards to a system which doesn't use explicit computation graphs.

As future work, we want to explore how we can further reason about these graphs. $\lambda$-blocks allows to work with a high-level DAG, which opens opportunities for graph complexity analysis, serialization methods comparison, caching optimizations, automatic choosing of data processing libraries, in-depth monitoring, and verification of programs' semantics. Another axis we want to explore is data streaming, and how we can simply and efficiently implement continuous queries on online data; as well as for example implementing triggers with blocks, which could be used for many automation tasks beyond the scope of data analysis. Another open problem is the representation of lambda functions: as of today, we used Python's notation, because the execution engine is written in Python. We plan to further explore the possibilities to stay language-agnostic for this issue. Finally, we plan to look for new ways to mirror library APIs, to simplify the writing and maintenance of block collections.

# Chapter 6

# Conclusions

*« Is the universe computable? If so, it may be much cheaper in terms of information requirements to compute all computable universes instead of just ours. »*

A Computer Scientist's View of Life, the Universe, and Everything [103]

We have presented the ecosystem of large-scale data processing, and three contributions which live at different layers of the software stack.

The platform we introduced for the Smart Support Center project is an example of a scalable architecture, tuned to perform time series predictions to guess system failures in advance. Recording the past history of monitored metrics and applying linear regression algorithms on it proved to be efficient at predicting their future behaviour, while avoiding occasional spikes that trigger false positive warnings on traditional monitoring systems. We've built this at scale, using a Cassandra database to keep all the historical data and the prediction parameters for every metric, along with Spark workers to perform the predictions in real-time. Evaluating this system proved its efficiency: it scales linearly up to (at least) 160 CPU cores, the end-to-end pipeline to compute one prediction takes about 1 second, and it is possible to manage 85 monitored servers (or 1440 monitored services on average) per worker of our system, when using the most pessimistic values.

Our second contribution lies deeper on the stack, and concerns the routing of data in a real-time distributed processing engine. To improve the throughput and the latency such a system can handle, we designed an algorithm to detect correlations in the keys of messages, for example the geolocated country and the hashtags, in the case of tweets. Once the correlations are detected, the messages exhibiting them can be routed such that they hop on fewer machines, because the worker nodes dealing with their keys are co-located. This decreases the network use, allowing for better performances. Moreover, because correlations are dynamic, this system is able to stay up-to-date with a reconfiguration algorithm, in charge of updating the routing tables while transferring state between worker nodes. The gains obtained with this system highly depend on parameters such as the network speed and the correlation potential the data exhibits, but the observed throughput ranged from 25% to 150% higher than without data-aware routing. This was evaluated with datasets coming from Twitter and Flickr.

Finally, higher on the stack, we described $\lambda$-blocks, a programming framework acting as a proxy between developers and complex systems. It is meant to allow non-distributed computing specialists to write data processing applications in a novel manner, by assembling blocks of code in a directed graph. Blocks are meant to process data themselves, or wrap more specialized tools such as distributed processing engines. $\lambda$-blocks comes with many batteries included to build complex ETL (extract-transform-load) pipelines, without writing any line of code. It leverages code reuse through block parameterization and sub-graph embedding, and exposes a simple API to extend its block libraries. Moreover, a plugins system gives developers a way to manipulate the computation graph of their programs, which opens the door to innovative ways to manipulate them. We showed examples of manipulations for instrumentation, debugging, and caching purposes. Using $\lambda$-blocks doesn't impact performances, as a maximum of 50 ms of overhead has been measured when comparing against equivalent programs.

We have mentioned some future work opportunities in the previous chapters. There are many directions we can take to make this work more useful and stable, and many ways to add novel features on top of it. Extending the developed systems would be very helpful:

- Adding other machine learning algorithms to Smart Support Center, and notably deep learning algorithms, would make it easy to compare their performances, and would bring interesting balances between coarse-grained predictions (general long-term tendencies) and fine-grained predictions (spikes, which often trigger false positive alerts). Integrating such a system with the user tickets, the problems manually reported by clients, is also quite challenging, as it involves text mining, and it can trigger privacy and security problems (for example, a lot of tickets contain credentials). The benefits of this automated assistance to system administrators are directly translated to better service-level agreements proposed by companies, and ultimately a better user experience for their clients.

- The routing algorithm which takes into account correlations between the message fields can also be extended. Two natural and compatible paths can be taken to make it more useful. The first one is its extension to a multi-tiered network, where the distance between two virtual machines is considered, instead of the binary assumption "on the same server"/"on a different server". This makes the graph representation and the routing decisions more complex, but is necessary if it needs to be scaled up beyond a rack of servers. The second extension concerns its implementation: it should be decoupled from Apache Storm, and be usable as an independent layer. It requires a deeper abstraction, as the actors of Storm (bolts and spouts) and those of other engines can follow very different designs, but being able to use this routing optimization on different engines and without updating application code would bring high benefits to developers.

- Finally, $\lambda$-blocks' included batteries can be easily extended. A very useful thing to do would be the addition of new block libraries, to read and write from different filesystems and databases, communicate with APIs, and wrap some well-known distributed frameworks. The plugin system for graph manipulation can also be extended to allow plugins to perform more fine-grained computations, both static (before program execution) and dynamic (at any step of the execution, not only before or after each block). The addition of streaming features and the continuous execution of a computation graph would make $\lambda$-blocks useful beyond the offline computations it currently performs. Lastly, a solid iterative system would be a really useful feature. Iterative computations are currently done within blocks, which is a hack in that it is not optimal for code reuse and algorithm representation. Inspiration from block-based programming interfaces can give us elegant solutions to implement loops and conditionals externally from blocks.

Beyond these extensions to our contributions, we conclude this thesis by sharing a more long-term vision of the path $\lambda$-blocks can take to further reduce the gap between programmers and complex data processing systems. We would love to perform experiments on both sides of $\lambda$-blocks' format: on the programmer side, with for example graphical interfaces, automatic graph generation, or even formal tools to verify the correctness of computations; and on the system side, with an easy deployment of the framework through block libraries' dependencies, better execution monitoring and debugging tools, and an execution environment comprised of light containers, automatically configured without the complex knowledge required today for the maintenance of distributed systems. Of course, different tools should work together towards these goals in an integrated ecosystem, and we're far from pretending $\lambda$-blocks should do all that, but it is the general direction we take when thinking of it as a middleware between systems (computers, networks, and data) and programs (algorithms, data transformations, and information extraction).

# Bibliography

[1] Apache Beam. https://beam.apache.org/.

[2] Blaze. http://blaze.pydata.org/.

[3] Cascading | Application Platform for Enterprise Big Data. http://www.cascading.org/.

[4] Coservit. http://coservit.com.

[5] Flickr dataset. https://webscope.sandbox.yahoo.com/catalog.php?datatype=i&did=67.

[6] Flowhub and Noflojs. https://flowhub.io/.

[7] Microsoft cloud azure. https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-algorithm-choice.

[8] Nagios-the industry standard in it infrastructure monitoring. http://www.nagios.org/.

[9] OpenStack. https://www.openstack.org/.

[10] Pipeline.io. http://pipeline.io/.

[11] Pipes.digital. https://www.pipes.digital/.

[12] Puma benchmarks and dataset downloads. https://engineering.purdue.edu/~puma/datasets.htm.

[13] PythonSpeed/PerformanceTips - Python Wiki. https://wiki.python.org/moin/PythonSpeed/PerformanceTips#Import_Statement_Overhead.

[14] Smart support center. http://www.smartsupportcenter.org.

[15] Xplenty. https://www.xplenty.com/is/.

[16] Zabbix prediction triggers. https://www.zabbix.com/documentation/3.0/manual/config/triggers/prediction.

[17] Adsquare. http://www.adsquare.com/comparing-performance-of-spark-dataframes-api-to-spark-rdd/, 2016.

[18] Shinken. http://www.shinken-monitoring.org/, 2017.

[19] Zabbix. http://www.zabbix.com, 2017.

[20] V. Akgiray. Conditional heteroscedasticity in time series of stock returns: Evidence and forecasts. *Journal of business*, pages 55–80, 1989.

[21] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[22] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[23] O. Alliance. Osgi-the dynamic module system for java, 2009.

[24] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 577–588. ACM, 2013.

[25] C. Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, 2006.

[26] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218. ACM, 2013.

[27] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[28] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[29] J. Atwood. The best code is no code at all. `https://blog.codinghorror.com/the-best-code-is-no-code-at-all/`, 2007.

[30] S. Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 137–142. ACM, 2010.

[31] X. Bai, A. Jégou, F. Junqueira, and V. Leroy. Dynasore: Efficient in-memory store for social applications. In *Middleware 2013 - ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*, pages 425–444, 2013.

[32] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[33] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[34] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak. Learnable programming: Blocks and beyond. *Commun. ACM*, 60(6):72–80, May 2017.

[35] S. Beheshti-Kashi, H. R. Karimi, K.-D. Thoben, M. Lütjen, and M. Teucke. A survey on retail sales forecasting and prediction in fashion markets. *Systems Science & Control Engineering*, 3(1):154–161, 2015.

[36] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[37] G. Bontempi, S. B. Taieb, and Y.-A. Le Borgne. Machine learning strategies for time series forecasting. In *Business Intelligence*, pages 62–77. Springer, 2013.

[38] G. Box and G. Jenkins. *Time series analysis: forecasting and control*. Holden-Day series in time series analysis. Holden-Day, 1970.

[39] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[40] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[41] C. Budak, T. Georgiou, D. Agrawal, and A. El Abbadi. Geoscope: Online detection of geo-correlated information trends in social networks. *Proc. VLDB Endow.*, 7(4):229–240, Dec. 2013.

[42] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[43] S. D. Campbell and F. Diebold. Weather forecasting for weather derivatives. *Journal of the American Statistical Association*, 100:6–16, 2005.

[44] M. Caneill, N. De Palma, A. Ait-Bachir, B. Dine, R. Mokhtari, and Y. Gizem Cinar. Online metrics prediction in monitoring systems. In *2018 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2018.

[45] M. Caneill, A. El Rheddane, V. Leroy, and N. De Palma. Locality-aware routing in stateful streaming applications. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 4:1–4:13. ACM, 2016.

[46] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[47] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 344–347, New York, NY, USA, 2015. ACM.

[48] T. Chalermarrewong, T. Achalakul, and S. C. W. See. Failure prediction of data centers using time series and fault tree analysis. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 794–799, Dec 2012.

[49] C. Chatfield. *The Analysis of Time Series: An Introduction, Sixth Edition*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis, 2003.

[50] S.-M. Chen and J.-R. Hwang. Temperature prediction using fuzzy time series. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 30(2):263–275, 2000.

[51] Y. G. Cinar, H. Mirisaee, P. Goswami, E. Gaussier, A. Ait-Bachir, and V. Strijov. Time series forecasting using rnns: an extended attention mechanism to model periods and handle missing values. *arXiv preprint arXiv:1703.10089*, 2017.

[52] G. G. Creamer and Y. Freund. Predicting performance and quantifying corporate governance risk for latin american adrs and banks. *Financial Engineering and Applications*, 2004.

[53] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, Sept. 2010.

[54] A. Davidson and A. Or. Optimizing shuffle performance in spark. *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep*, 2013.

[55] J. G. De Gooijer and R. J. Hyndman. 25 years of time series forecasting. *International journal of forecasting*, 22(3):443–473, 2006.

[56] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[57] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan. Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353, 2013.

[58] L. Fischer and A. Bernstein. Workload scheduling in distributed stream processors using graph partitioning. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 124–133, 2015.

[59] M. Franklin. The berkeley data analytics stack: Present and future. In *Big Data, 2013 IEEE International Conference on*, pages 2–3. IEEE, 2013.

[60] N. Fraser et al. Blockly: A visual programming editor. `https://developers.google.com/blockly`, 2013.

[61] T. V. Gestel, J. A. K. Suykens, D. E. Baestaens, A. Lambrechts, G. Lanckriet, B. Vandaele, B. D. Moor, and J. Vandewalle. Financial time series prediction using least squares support vector machines within the evidence framework. *IEEE Transactions on Neural Networks*, 12(4):809–821, Jul 2001.

[62] M. Ghassemi, M. A. F. Pimentel, T. Naumann, T. Brennan, D. A. Clifton, P. Szolovits, and M. Feng. A multivariate timeseries modeling approach to severity of illness assessment and forecasting in icu with sparse, heterogeneous clinical data. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 446–453. AAAI Press, 2015.

[63] C. L. Giles, S. Lawrence, and A. C. Tsoi. Noisy time series prediction using recurrent neural networks and grammatical inference. *Machine learning*, 44(1):161–183, 2001.

[64] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, 2014. USENIX Association.

[65] G. Hamilton. Javabeans. *API Specification, Sun Microsystems*, 1997.

[66] J. D. Hamilton. *Time series analysis*, volume 2. Princeton university press Princeton, 1994.

[67] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, O. Miller, D. Armendariz, J. McKinsey, Z. Machardy, E. Lemon, S. Morris, and J. Paley. Snap! (build your own blocks) (abstract only). In *Proceedings of the 45th*

*ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 749–749, New York, NY, USA, 2014. ACM.

[68] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4:1111–1122, 2011.

[69] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[70] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9. Boston, MA, USA, 2010.

[71] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011.

[72] T. Jiang, Q. Zhang, R. Hou, L. Chai, S. A. Mckee, Z. Jia, and N. Sun. Understanding the behavior of in-memory computing workloads. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 22–30, Oct 2014.

[73] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. *arXiv preprint arXiv:1702.04024*, 2017.

[74] Jukka Lehtosalo et al. Mypy. `http://mypy-lang.org/`.

[75] P.-H. Kamp and R. N. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[76] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[77] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 16:1–16:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[78] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[79] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250. ACM, 2015.

[80] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[81] L. Lamport. Distribution, May 1987.

[82] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. Monitoring, prediction and prevention of sla violations in composite services. In *2010 IEEE International Conference on Web Services*, pages 369–376, July 2010.

[83] L. Li, C.-J. M. Liang, J. Liu, S. Nath, A. Terzis, and C. Faloutsos. Thermocast: A cyber-physical forecasting model for datacenters. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1370–1378, New York, NY, USA, 2011. ACM.

[84] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[85] A. D. McQuarrie and C.-L. Tsai. *Regression and time series model selection.* World Scientific, 1998.

[86] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[87] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[88] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[89] R. C. Merkle. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 369–378. Springer, 1987.

[90] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 398–412, Berlin, Heidelberg, 2005. Springer-Verlag.

[91] K.-R. Müller, A. J. Smola, G. Rätsch, B. Schölkopf, J. Kohlmorgen, and V. Vapnik. Predicting time series with support vector machines. In *International Conference on Artificial Neural Networks*, pages 999–1004. Springer, 1997.

[92] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In *31st IEEE International Conference on Data Engineering, ICDE*, pages 137–148, 2015.

[93] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *32nd IEEE International Conference on Data Engineering, ICDE*, 2015.

[94] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Dec 2010.

[95] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[96] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, Oakland, CA, 2015. USENIX Association.

[97] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[98] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 149–161, New York, NY, USA, 2015. ACM.

[99] D. K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015.

[100] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, Nov. 2009.

[101] D. Riemer, F. Kaulfersch, R. Hutmacher, and L. Stojanovic. Streampipes: solving the challenge with semantic stream processing pipelines. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 330–331. ACM, 2015.

[102] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 80–91, New York, NY, USA, 2015. ACM.

[103] J. Schmidhuber. A computer scientist's view of life, the universe, and everything. In *Foundations of computer science*, pages 201–208. Springer, 1997.

[104] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[105] S. Singh and Y. Liu. A cloud service architecture for analyzing big monitoring data. *Tsinghua Science and Technology*, 21(1):55–70, Feb 2016.

[106] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 535–546, April 2017.

[107] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic

partitioning for distributed transaction processing. *Proc. VLDB Endow.*, 8:245–256, November 2014.

[108] tef. How do you cut a monolith in half? `https://programmingisterrible.com/post/162346490883/how-do-you-cut-a-monolith-in-half`, 2017.

[109] The Apache Spark developers. ML Pipelines. `https://spark.apache.org/docs/latest/ml-pipeline.html`, 2017.

[110] The Apache Storm developers. Flux. `http://storm.apache.org/releases/2.0.0-SNAPSHOT/flux.html`, 2017.

[111] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156. ACM, 2014.

[112] V. Vapnik. The nature of statistical learning theory, 1995.

[113] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[114] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications magazine*, 35(2):46–55, 1997.

[115] D. Weise, S. Garfinkel, and S. Strassmann. *The UNIX-Haters Handbook.* IDG books, 1994.

[116] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[117] YelpArchive. Pyleus. `https://github.com/YelpArchive/pyleus`, 2016.

[118] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 555–564, New York, NY, USA, 2017. ACM.

[119] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[120] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[121] M. Zaharia, B. Hindman, A. Konwinski, A. Ghodsi, A. D. Joesph, R. Katz, S. Shenker, and I. Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 17–17, Berkeley, CA, USA, 2011. USENIX Association.

[122] Åse Dragland and Sintef. Big data, for better or worse: 90% of world's data generated over last two years. `www.sciencedaily.com/releases/2013/05/130522085217.htm`, 2013.

# Appendix A

# Source code for PageRank in $\lambda$-blocks

```
---
name: PageRank
---
- block: spark_readfile
  name: read_links
  args:
    filename: "hdfs://localhost:9000/wikipedia/wiki-topcats.txt"
    master: "spark://127.0.1.1:7077"

- block: spark_map
  name: split_urls
  inputs:
    data: read_links.result
  args:
    func: "lambda x: tuple(x.split())"

- block: spark_distinct
  name: distinct_links
  inputs:
    data: split_urls.result

- block: spark_groupByKey
  name: group_urls
  inputs:
    data: distinct_links.result

```

```
- block: spark_map
  name: ranks
  inputs:
    data: group_urls.result
  args:
    func: "lambda x: (x[0], 1.0)"

- block: spark_pagerank
  name: pagerank
  inputs:
    links: group_urls.result
    initial_ranks: ranks.result
  args:
    iterations: 10

- block: spark_readfile
  name: read_names
  args:
    filename: "hdfs://localhost:9000/wikipedia/ \
               wiki-topcats-page-names.txt"
    master: "spark://127.0.1.1:7077"

- block: spark_map
  name: split_names
  inputs:
    data: read_names.result
  args:
    func: "lambda x: tuple(x.split(maxsplit=1))"

- block: spark_filter
  name: filter_empty
  inputs:
    data: split_names.result
  args:
    func: "lambda x: len(x) == 2"

- block: spark_join
  name: join
  inputs:
    data1: pagerank.result
    data2: filter_empty.result
```

```

- block: spark_takeOrdered
  name: take
  inputs:
    data: join.result
  args:
    num: 5
    key: "lambda x: -x[1][0]"

- block: show_console
  name: show
  inputs:
    data: take.result
```