Universitat Rovira i Virgili (URV) y Universitat Oberta de Catalunya (UOC)

Master in Computational and Mathematical Engineering

FINAL MASTER PROJECT

Area: Cloud Computing

Painless Data Analytics in the Cloud

Grouping data in serverless architectures

Autor: German Telmo Eizaguirre Suarez

Tutor: Marc Sánchez Artigas

Tarragona, June 19, 2021

Dr. Marc Sánchez Artigas, certifies that the student German Telmo Eizaguirre Suárez has elaborated the work under his direction and he authorizes the presentation of this memory for its evaluation.

Director's signature:

Credits/Copyright

FINAL PROJECT SHEET

Title:	Painless Data Analytics in the Cloud
Autor:	German Telmo Eizaguirre Suárez
Tutor:	Marc Sánchez Artigas
Date (mm/yyyy):	06/2021
Program:	Master in Computational and Mathematical Engineering
Area:	Cloud Computing
Language:	English
Key words	Cloud Computing, Big Data, Serverless

Dedicatory

For Jurgi, Borja, Endika, Iker, Jon, Koldo, Jon, Mikel, Gorka, Raúl, Urbil and Mikel.

Acknowledgments

I would like to thank the whole CloudLab group for their support and good vibes during the completion of this work, specially my thesis director Dr. Sánchez Artigas, for putting up with my stumbles and my lack of concentration and guiding me on the right path.

Abstract

Big Data analysis is ubiquitous in every major research and industry domain. To exploit Big Data's vast information disponibility, analysis technologies have moved to the Cloud, which provides the necessary escalability for workloads of extremely variable size. However, most cloud technologies are not usable by the average programmer, as they entail complex resource selection and management tasks.
The *serverless* paradigm, with FaaS as its basic computation unit, brings enough transparency to bridge untrained cloud programmers and exigent cloud computations. During the past years, the progress on *serverless* analytics platforms has been remarkable. Unfortunately, users are still involved in underlying lower level responsibilities, such as auxiliary resource provisioning or specifying the scale of parallelism of distributed applications. In this work, we validate a fully transparent and *serverless* framework for I/O intensive data analytics operations with practically no intervention by the user. We present an efficient groupBy operator capable of operating on large scale datasets in a completely *serverless* architecture. Finally, we propose some performance optimizations for FaaS applications and we evaluate them on the Terasort benchmark with encouraging results.

Keywords: Cloud Computing, Big Data, Serverless

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Big data analysis in the Cloud

Big Data is omnipresent in present-day industry and research. The arises of the Internet of Things (IoT) and the smart devices or the rapid escalation in the resolution of research equipment have vastly increased the data volumes available. Nowadays, big data is used within social networks, healthcare systems, retailing companies and even governments [32]. Not only from a research and development perspective, but also economically, the presence and the potential of Big Data is immense. In 2020, the Big Data market was forecast to produce more than US$121.4 billion[39]. Data volume, or the quantity of information, however, is not separately sufficient for the development of more intelligent and precise systems. Its gathering, storage, analysis and utilization are essential for taking advantage of Big Data[41].

The behavior of current systems is far from being homogeneous. In real-time applications, data flows vary in size, latency and continuity. The format of datasets is also greatly variable, and so is their volume, ranging from MB to even PB magnitudes. For instance, Walmart generates data from customer transactions at a rate of 2.5 PB of data per hour[39]. Hence, the processing of big data requires great scalability, flexibility and performance. Traditional technologies fit poorly in such scenario, due to limited storage capacity and explicit and costly management of resources. Fortunately, in synergy with the enlargement of data volumes, the development of cutting-edge technologies have boosted and broadened the tools available for their analysis.

Distributed and scalable systems are crucial for the appropriate manipulation of Big Data, and Cloud computing suits utterly such preconditions. Cloud computing encompasses web application, software and hardware services that are provided through the Internet, remotely accessed by the user and typically hosted in data centers [16]. Specially public clouds, which make resources available on demand, establish a suitable context for big data processing and analysis. Since the appearance of first cloud services, the cloud computing paradigm has evolved, and now providers implement shared pools of heterogeneous computing resources, with extensive catalogs of inter-connectable and distributed services [39]. Services are classified into types based on their scope and the resource they provide.

- **Software as a Service (*SaaS*):** Frameworks or web applications for data processing and analysis, maintained by the cloud provider.
- **Platform as a Service (*PaaS*):** Application development and deployment frameworks, with the infrastructure being managed by the cloud provider.
- **Infrastructure as a Service (*IaaS*):** Configurable resources, such as servers, storage systems or networks.

Big Data Cloud architectures typically integrate resources of the different categories vertically or modularly; one can provision or release resources on-demand, and build complex systems capable of scaling horizontally based on workload sizes and latencies. By virtue of its flexibility, cloud computing has naturally housed big data processing. From a pure theoretical perspective, big data has been defined with "five V's" -Volume, Variety, Velocity, Value and Veracity- based on its characteristics and challenges [24]. All of the 5 V's are addressed with the features of cloud computing technology [41].

1. **Extensive pooling of computing resources**: permits the horizontal escalation depending on the volume of the data.
2. **Elasticity**: cloud systems can rapidly adapt to input volumes and types through fast incorporation of resources.
3. **On-demand provisioning:** the user is offered a broad window to decide the amount of resources, the usage limits and the output format of the systems.
4. **Self-service**: variety and veracity of big data can be administered by choosing the appropriate resources for each case from the cloud provider catalog.
5. **Billed according to resource reservation**: the analysis resolution, the performance or the capacity of the system can be adjusted adapting the resource provisioning to the budget of the user.

1.2 FaaS for user-friendly Cloud computing

Cloud computing definitely provides enough scalability and flexibility for Big Data workloads. Unfortunately, many cloud users are not capable of exploiting the Cloud to the maximum, or even making an efficient use of it. Cloud computing in public clouds involves selecting resource type, size and number, setting up clusters and administrating costs, besides that Big Data jobs are usually embarrassingly parallel and hard to optimize. The classical Cloud computing approach, also known as *serverful* cloud computing [26], entails such complex management tasks that narrow the usability of the Cloud. Yet, giving accessibility to Cloud computing is the way to interdisciplinary collaboration, one of the leading initiatives of Big Data in research[41]. The collaboration between domains is necessary for a cohesive analysis of Big Data and a coordinated development of universal knowledge, but not even close to all data analysts or computer scientists have a sufficient level of expertise in the cloud. The implementation of high-level cloud computing frameworks, masking the inner technical complexities of cloud systems, is key to keep up with the exploration of Big Data.

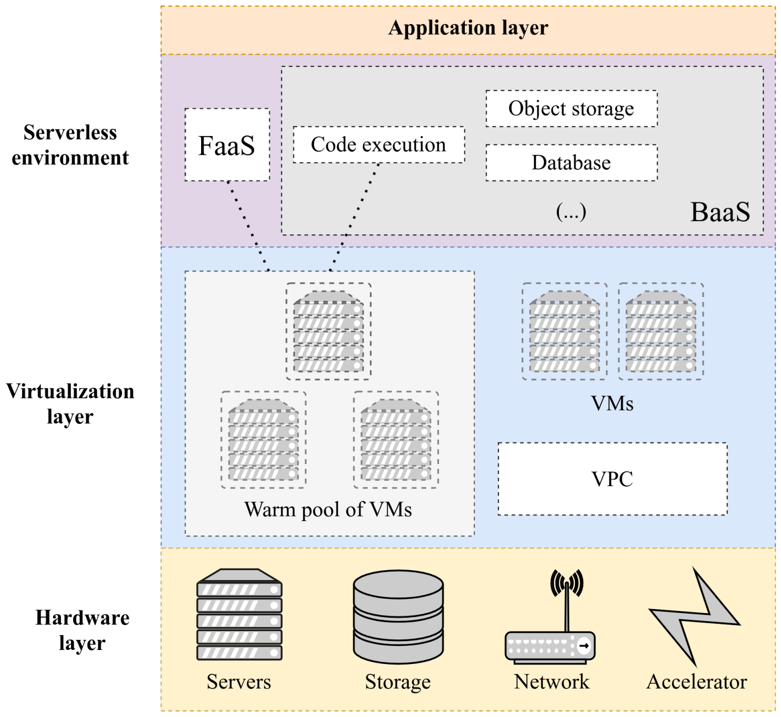


Figure 1.1: Schema of a common *serverless* architecture in public cloud providers. VM: Virtual Machine. VPC: Virtual private Cloud. FaaS: Function as a Service. BaaS: Backend as a Service. (based on [25, 38]).

As both a response to *serverful* computing and an evolution of cloud computing emerges the *serverless* cloud computing. The *serverless* paradigm hides "low-level" cloud platform services from the user (such as virtual machines or networks) and brings higher level development abstractions [38]. Behind the curtain, *serverless* environments rely on warm pools of virtual machine instances on multi-tenant physical servers[26] (see Fig. 1.1). Users, per contra, can work free from resource management issues. The *serverless* paradigm relies on the following principles [26, 38].

1. Resources scale rapid and automatically. Storage and computation resources are decoupled, and their billing and scaling is independent.

2. Users only provide the code to execute, and it is the cloud provider which performs the resource provisioning automatically.

3. The billing is performed in a pay-as-you-go manner, that is, resources are charged *a posteriori*, based on their usage, and must not be previously allocated.

Cloud functions, named *Function-as-a-Service* (FaaS), are the basic and general purpose unit of *serverless* environments. In the FaaS model, the user can deploy code that is executed in isolation at automatically provisioned, remote and virtual containers. Cloud functions are *stateless* and their runtime memory and execution time are limited. Users can configure some features about Cloud Functions, such as the maximum execution time, the available physical memory or the libraries of a custom runtime environment, under certain limits. Along with FaaS, *serverless* environments are composed of a set of specific *Backend-as-a-Service* (BaaS) utilities (Fig. 1.1). BaaS provides higher level utilities to substitute server-side components and bring development closer to the front-end [31]. BaaS offerings include, for example, cloud storage, key-value database and authentication systems [1]. Integrating FaaS and decoupled BaaS utilities allows the development of automatically scalable systems that can be charged only based on usage.

Today, most cloud providers include cloud functions in their catalog (See Tab. 1.1) and their elasticity is greatly appreciated in online commerce and media providers. For instance, Expedia, with 1.2 billion requests per month, Thomson Reuters, with 4000 requests per second, and Vevo [23], among others, have integrated *serverless* functions into their systems.

[1]There is an increasing offer of fully managed BaaS utilities for custom code execution, such as IBM Cloud Code Engine [5] or Google Colab [4].

Cloud provider	*FaaS*	Shared object storage
Amazon Web Services (AWS)	AWS Lambda	Simple Storage Service (S3)
IBM Cloud	IBM Cloud Functions	Cloud Object Storage (COS)
Microsoft Azure	Azure Functions	Blob
Google Cloud Platform	Google Cloud Functions	Google Storage
Alibaba Cloud	Function Compute	Object Storage Service (OSS)
Oracle Cloud	Oracle Functions	Oracle Cloud Infrastructure (OCI)

Table 1.1: *Function as a Service (FaaS) and shared object storage options for some of the leading cloud providers.*

As mentioned earlier, data-analysis pipelines involve embarrassingly parallel workloads, with highly variable scales of parallelism. To support fluctuating parallelism levels elastically, we need a system with great scalability, short startup time and fast process deployment. FaaS seem appropriate for such scenario. On the one hand, they offer the possibility of invoking a large number of functions (up to 1000 parallel functions for IBM Cloud Functions (CF) and AWS Lambda, with a standard account). On the other hand, cloud functions suffer from lower cold start times than VMs, thanks to microVMs and technologies like Firecracker (in AWS Lambda) [17] and warm pooling of VMs [26]. FaaS are particularly interesting for Big Data pipelines with disparate data magnitudes at different stages. In such scenario, allocating VMs for the most demanding stage could imply over-provisioning resources, and consequently, paying for services that remain idle for a great part of the execution time. FaaS elasticity, instead, could adjust the necessary number of cloud functions for each stage, avoiding needless billings for unused resources.

It is not only on paper that *serverless* fits modern data analysis. Previous research has proven the feasibility of using the FaaS model to solve real research problems. Geospatial data analysis, which remains a challenging big data domain -due to its sheer learning curve, implementation complexity and data size- [41], has been successfully tackled with *serverless* functions [19]. Bioinformatics can also be ported to *serverless* architectures, with precedents in proteomics [29] and metabolomics [37].

1.3 Data analysis in the Cloud

The number of available data analysis frameworks is large, and practically all of them are designed for the distributed computing of data. Even computing languages with data analysis features, such as R [11], have been enhanced with distributed data parallelism libraries [32].

With the raise of Big Data and cloud computing, many distributed frameworks have been ported to the Cloud. Users can provision resources and deploy their own data analytics system. As an alternative, cloud providers also include data processing and analysis Platform-as-a-Service (PaaS) utilities in their catalog, such as AWS EMR [1].

Spark [42] is arguably the most common and extended distributed data analysis engine. It is built on the MapReduce execution model [20] and provides SQL-like queries through its module Spark SQL. The MapReduce paradigm was originally designed for processing large data volumes in parallel, using distributed systems. On its basic it is composed of two following stages.

- **map**: A set of parallel workers is called (the mapper functions). Each mapper reads a set of entries from the input data of key-value type, and filters and/or applies a transformation function to the data. The output of each function is transferred to a shared file system.

- **reduce**: A second set of parallel workers is called (the reducer functions). Each reducer reads a set of keys from the mappers' output, applies a reduction function to its data and writes the output to the shared file system.

Despite its usability, Spark must be deployed by the user on the Cloud, with its corresponding management tasks. For example, it relies on a Java Virtual Machine (JVM), and reaching an efficient compromise between the JVM heap and the operating system installation is not straightforward [25]. Even for broadly adopted platform as Spark, we face complex administration tasks that hinders its usage in the Cloud, and once again, we fall into accessibility and transparency flaws. Inspired by the limited usability of the Cloud, some proposals have ported data analytics to *serverless* environments, primarily through the MapReduce model (or solely the map stage of the model) [25, 36, 34, 30, 40]. *Serverless* data analytics frameworks abstract cloud functions as workers, analogously to the nodes of a cluster. Generally, they act as a high-level wrapper for FaaS services, with intuitive programming languages like python: the user specifies the number of workers, writes its custom map, and if applicable, reduce functions and the framework is responsible for invoking cloud functions in the correspondent FaaS service.

1.4 Towards automated, I/O intensive workloads in *serverless* environments

Cloud functions provide fine-grained billing in data analytics workloads, avoid over-provisioning resources and ease the usage of the cloud. However, everything is not bright for the FaaS. Due to their stateless nature, point-to-point communication between *serverless* functions is not possible. Hence, intermediary services are necessary for the coordination of FaaS applications. Essentially, for two functions to communicate with one another, the first must write a message into a shared storage, and the second must actively read it through polling. Alternatives do exist, for example, invoking functions with custom triggers on shared storage writes [15] or direct communication with the hole-punching technique [40]. Nonetheless, such implementations require higher-level management and resource provisioning by the user, which contradicts *serverless*' promises of transparency and usability.

The naïve engine for the inter-function communication, but also the most comfortable for the end-user, is the use of shared object storage systems (SOSS). SOSS are high bandwidth, low-priced distributed storage systems available at most, if not all, cloud providers (see Tab. 1.1). Internally, they manipulate data as distinct units, each consisting of metadata and a set of fixed size blocks, similar to regular file systems. The provisioning and usage of SOSS tend to be straightforward, only requiring an identification key and a simple API or a Web Console for their management, so they are specially interesting for transparency.

As they are oriented to the medium and long-term storage of information, SOSS provide persistence but suffer from high latency in their operations. Thus, they are not the optimal option in terms of performance, specially in I/O-intensive workloads. Various proposals use Redis [12] as the intermediate storage system [34, 35]. Redis is an in-memory data store based on key-value hash tables, and it provides low latency I/O operations, unlike SOSS. Although using Redis raises the performance of *serverless* applications, in-memory storage cloud services are more expensive that SOSS, and their provisioning requires greater user manipulation. In our will to achieve fully transparent cloud usage, SOSS are presumably a more appropriate option.

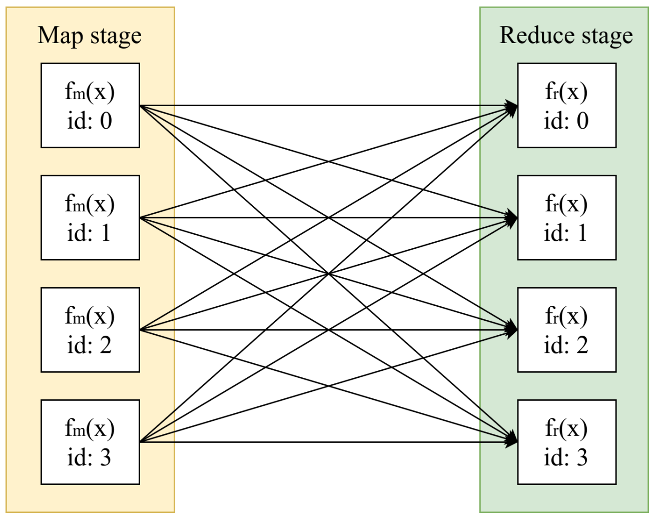
Exploiting transparency with SOSS in detriment of performance aggravates I/O-intensive steps of *serverless* workloads and makes them a bottleneck for the overall performance. The worst case would be the all-to-all shuffling between functions. In the MapReduce model, an all-to-all shuffling implies communicating every mapper function with every reducer function (see Fig. 1.2 (A)), while conserving the volume of the input data (no filtering or reduction is applied at the map stage). All-to-all shuffling is part of many data analysis queries, to say, `sort`, `groupBy` or Spark's `repartitionBy`, and represents a limiter in terms of performance because communications between workers increase quadratically with respect to the number of workers. In a *serverless* MapReduce implementation with SOSS as intermediary (see Fig. 1.2 (B)) such problems are exacerbated, due to high latency requests.

Therefore, steps with all-to-shuffle constrain multi-stage *serverless* data analysis workloads. As the number of requests increases exponentially with the number of workers, choosing the optimal worker number for each case is fundamental to minimize execution time and consequently decrease the pay-as-you-go billing of the analysis. However, *serverless* frameworks have not still integrated the capacity of automatically choosing the optimal number of workers for each workload, and it is the user who has to indicate the level of parallelism. Being realistic, the vast majority of not specialized cloud users would not be able to determinate a close to optimal scale of parallelism based on the data to analyze. Hence, roviding a framework that automated the number of cloud functions to use based on the job features would be a great step towards democratizing the usage of the Cloud for general users.

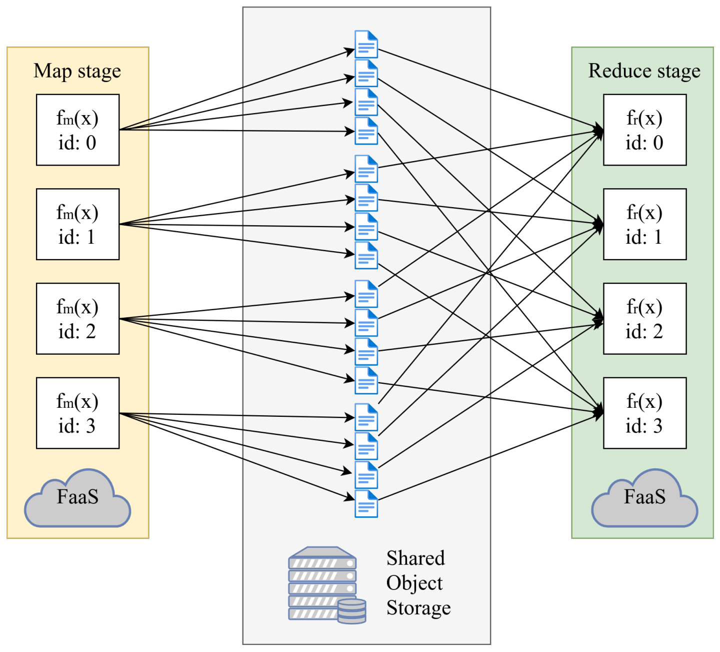
1.5 Contribution

In the present document, we present the following contributions.

- We generalize an analytical model, proposed in our precedent work [21], to infer the optimal number of workers for general all-to-all shuffle operations in environments with uncoupled computation and storage.
- We present an efficient `groupBy` operator for *serverless* architectures.
- We propose and validate a set of optimizations for distributed data analytics in cloud functions.



(a)



(b)

Figure 1.2: (A) All-to-all shuffling in the theoretical MapReduce model. (B) All-to-all shuffling in a *serverless* MapReduce, using a shared object storage system as communication intermediary. In the *serverless* MapReduce, the number of intermediary files in the storage system and the number of GET and PUT requests grows quadratically with respect to the number of workers.

Chapter 2

Related Work

The idea of tackling data analysis with FaaS is not new. Pywren first, with AWS Lambda [25], and later Lithops, with IBM Cloud Function [36], have adapted the MapReduce model to be run over FaaS using high-level programming languages. They both provide `map` calls to process an iterable object across cloud functions using data-level parallelism. Lithops outperforms Pywren in functionalities, giving support to custom runtime configuration and simple multiple-map single-reduce jobs among other features. In addition, it has been recently upgraded with more complex interprocess communication objects using Redis and transparent multi-cloud access [35]. Crucial [18] and `gg` [22] also focus on *serverless* analytics from different approaches. However, neither of the frameworks implements general purpose shuffle operation, and choosing the number of workers to use still corresponds to the user.

Some proposals include *serverless* environments with inherent all-to-all shuffling support, but partially built on *serverful* components. Pocket [28] implemented an elastic and automated *serverless* storage system on top of VMs. Maybe the closest option of transferring Spark to the *serverless* would be Flint [27], which adapted Spark execution plans to AWS Lambda and gave the possibility of running PySpark over FaaS. In the case of Flint, shuffling was performed on AWS SQS, a poorly scalable message queuing service.

Specific research on *serverless* shuffling also falls into not transparent solutions. Locus [34] implemented a greatly scalable sort with good performance integrating low latency (Redis, through AWS ElastiCache) and high latency (SOSS, with AWS S3) storage systems. For the first time, it included an analytical model to predict the theoretically optimal number of workers for the *serverless* shuffling, but forced the user to allocate Redis servers manually. Lambada

[30] offered a shuffling operator with a completely *serverless* architecture, but did not infer the optimal scale of parallelism automatically.

Boxer [40] is the most recent solution addressing the communication between cloud functions. Boxer implements point-to-point communications between functions using a TCP/IP protocol and the hole-punching technique, with good performance results in the TPC-H benchmark. Nevertheless, it depends on an external coordinator for the hole-punching, such as a a VM, and does not give detailed information about its performance in all-to-all shuffling operations. Caerus [43] extends Locus with task scheduling and pipelining to minimize job cost and execution time, but as Locus, lacks enough transparency for users with no cloud expertise.

In our precedent paper, Primula [21], we integrated transparency and a fully *serverless* architecture in a single framework. Primula presented a FaaS+SOSS sort primitive and the capability of automatically infering the scale of parallelism for each job, responding to the shortfalls of the existing proposals.

Chapter 3

Planning

3.1 Objectives

We focused the present research as a generalization of our previous paper, Primula [21]. The main objectives for the final results were the following.

1. **OM1:** Concordance between our theoretical predictions and the results for the `groupBy` operator.
2. **OM2:** Improvement of our benchmarked results over Primula.

To tackle our main objectives and plan our milestones, we set the following secondary objectives.

1. **OS1:** Obtain no greater execution times for our new `groupBy` operator than those for Primula's `sort` with the same medium scale datasets. The improvement margin is modest with medium scale datasets, so the optimizations resulted from our work should at least give the same execution time for the sort and the `groupBy`.
2. **OS2:** Execute the 100GB Terasort [33] with lower execution time than Primula. We set the 100GB Terasort as our main benchmark for the performance of our final implementation because (i) it is an standardized benchmark addressed in many previous papers (ii) its scale is sufficiently big to constraint its execution in general purpose devices, so it is a good option to be executed in the Cloud.

3. **OS3:** Decrease memory utilization in cloud functions. Using less memory per function increases the range of partition sizes that can be assigned per worker.

3.2 Development scheme

Our development followed an iterative pattern. On its basic, each iteration had a related task (see the next section) and consisted on the following steps.

- **Research:** We studied the existing bibliography on the specific task to address. We consulted both academic publications and informative articles. For journal articles, we gave preference to journals at the first quartile of computer science-related domains (based on the Journal Citation Report (JCR) [1]), or alternatively, highly cited articles in Google Scholar [2] and the Web Of Science [3]. For congress publications, we considered congresses with at least A or A* score in the CORE ranking [4].

- **Implementation:** We implemented different options for the correspondent task.

- **Evaluation:** We evaluated the correct functioning of our implementations. We tried to minimize the executions in the Cloud to avoid needless billings, so we first executed every extension locally with threads instead of cloud functions and an abstraction of IBM COS over the local file system. Once ensured the results were correct, we executed small-scale experiments on the Cloud (150MB maximum dataset size).

- **Validation:** The perfect validation per iteration would be checking the completion of OS1 and OS2. However, as the execution of the 100GB Terasort benchmark (OS2) in the Cloud is too costly to be repeated at each iteration, we decided to only perform its evaluation after the final iteration. Experiments from OS1, instead, are more affordable and can be repeated with relative frequency. We decided to use the condition from OS1 as our evaluation method at each iteration. Also, each task had its specific validation method, which we present in the 4 chapter (for instance, using Python's `memory-profiler` to monitor memory optimizations).

[1] JCR impact search: https://www.recursoscientificos.fecyt.es/servicios/indices-de-impacto
[2] Google Scholar: https://scholar.google.com/
[3] Web Of Science: http://wos.fecyt.es/
[4] CORE Conference Portal: http://portal.core.edu.au/conf-ranks/

3.3 Task definition

We divided our planning into the following incremental tasks. For each tasks, we performed the steps described above.

T1: Basic `groupBy` implementation.

The goal was to implement a naïve `groupBy` based on the MapReduce model, using IBM Cloud Functions as the computation engine and IBM COS as the storage system.

T2: Exchange algorithm optimizations.

The goal was to improve the communication between the map and reduce stages of the execution, studying the best option for concurrent I/O.

T3: Intermediate data optimizations.

The goal was to find the best data format for intermediate files in the shuffle, concerning complexity, memory usage and in/out transformation performance.

T4: Memory usage and `groupBy` algorithmic optimizations.

We intended to find additional optimizations for the code in terms of efficient memory usage. For that, we studied different options for the inner functioning of the `groupBy` and the best model for data management.

T5: Additional performance optimizations.

For our final goal we tested code optimizations such as using compiled languages for certain sections, instead of Python.

Chapter 4

Design and architecture

4.1 `groupBy` operator

Grouping data by key is a common operation is data analysis, and every major data analysis framework provides a `groupBy` operator (so is the case, for example, of Spark [42], R [11] or Python's `pandas` [8]). It is included in performance benchmarks such as TPC-H [14] (query 3, for instance) and TPC-DS [13] (query 1, for instance). In a general `groupBy` operation, input data entries are arranged into groups based on certain criteria. For research purposes, we simplify such definition to the most common case of `groupBy`, in which rows with the same key are grouped together. The algorithm receives the name of the attribute to use as the key, and rearranges the entries to put those with the same key into the same group. The output is a set of groups, each with all the entries with for a certain key (see Fig. 4.1).

4.2 Integration of the `groupBy` with previous contributions

We propose a `groupBy` operator as an extension of our previous work, Primula [21]. In Primula, we presented a simple and standalone primitive in Python to sort a cloud-hosted dataset automatically. Primula's sort primitive decoupled the average cloud user from the responsability of specifying the number of functions to use in a shuffle workload. The execution flow of our `groupBy` operator is depicted in Fig. 4.2. Basically, the user only needs to specify the path of a

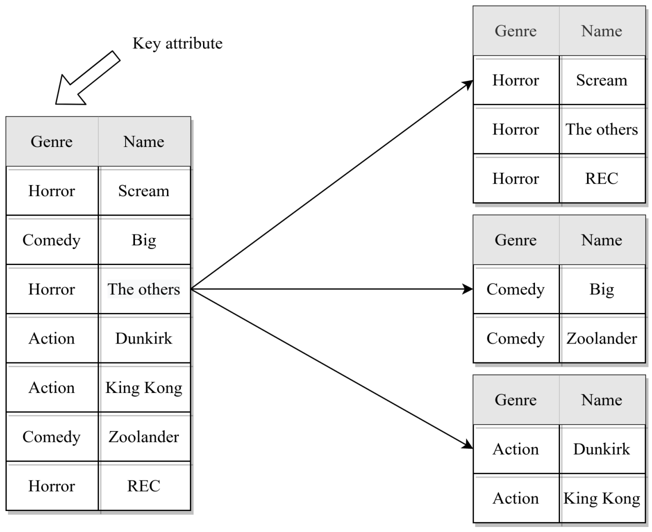


Figure 4.1: Graphical representation of the `groupBy` algorithm.

dataset in the Cloud and the column on which the grouping is applied, and it is the framework which orchestrates the whole `groupBy` job. To give a honest continuation to our research, we maintain the base cloud services used in Primula; IBM Cloud Functions as our FaaS and IBM Cloud Object Storage (COS) as our SOSS.

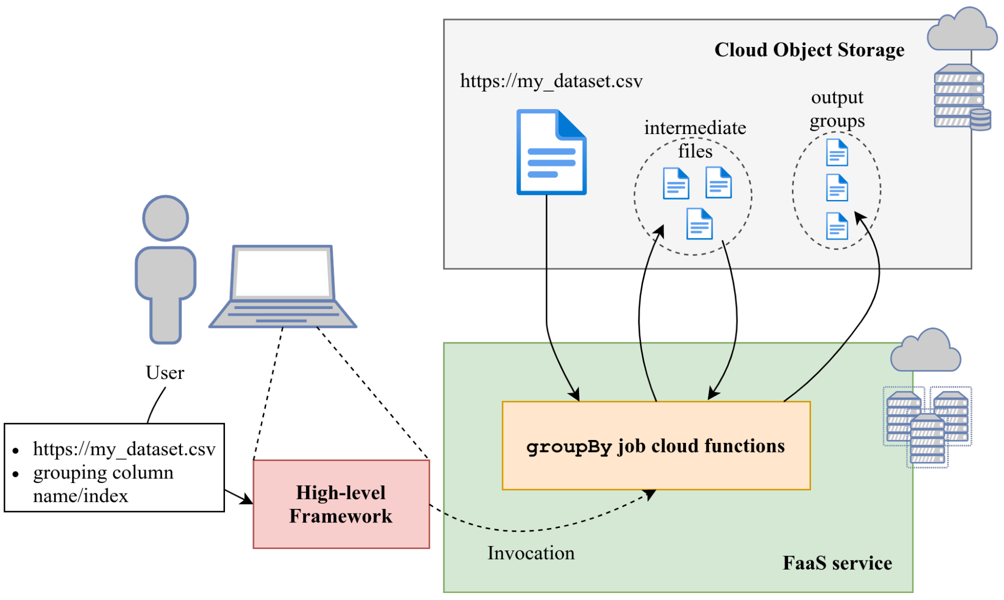


Figure 4.2

In the following sections we briefly mention the contributions and features included in the original Primula paper, and we describe how we have included them in our present study.

4.2.1 Grouping with the MapReduce algorithm

Our adaptation of the MapReduce model to the *serverless* `sort` comprised two preliminary stages and the sorting stage *per se*. First, a parser function performed a characterization of the dataset properties, extracting characteristics such as the data types, or the delimiter, in the case of tabular data. Then we executed a random sampling and finally the MapReduce sorting job, embodying every mapper and reducer in single cloud function.

For the `groupBy` problem, we reuse the initial parser function to characterize the dataset and we then call the grouping job directly. As in Primula, each of the mappers and each of the reducers is assigned to a cloud function. We take Spark's `groupBy` operator as the reference algorithm for our implementation. For the following explanation, we will assume we are using N mappers and M reducers for the `groupBy` execution, and the input dataset has size S.

1. We define N equidistant ranges in S, so that each range covers S/N bytes of data. We will call such ranges partitions. Each mapper reads its corresponding partition of the dataset from the Cloud.
2. To determine the destination reducer of each of the keys in their partition, mappers apply a hash partitioning to the keys to obtain an integer value H. The target reducer for each key will be $H \mod M$. As every mapper uses the same hashing algorithm, a certain key will be directed to the same reducer in every function.
3. Each mapper writes an intermediate file per reducer into the SOSS, containing all the entries in its partition corresponding to that reducer. There will be M intermediate files per mapper, and $N \times M$ in total.
4. Each reducer reads its corresponding intermediate files from the SOSS, and binds them. Finally it regroups the entries based on the keys, setting all the entries with identical key into the same group.
5. If executed segregated, `groupBy` algorithm finishes with each reducer writing the group for each key into the SOSS.

4.2.2 Granulated I/O

High latencies of SOSS requests contribute negatively to the shuffle performance. Most storage systems have an optimal transfer size for their I/O requests, that hinges on the storage system architecture, its network context and its internal management of data blocks. We adopted the bandwidth experiments performed in the Pywren paper [25] for AWS S3 and adapted them for throughput measurements. Our evaluation covered chunk sizes in the range of $[8, 128]$MB, and determined that 64MB is the optimal chunk size for IO operations between cloud functions and IBM COS, both in terms of bandwidth and throughput. Our framework fragments every I/O operation in 64MB chunks to improve data exchange operations.

4.2.3 Barrierless execution

The original MapReduce algorithm [20] executed the map and reduce stages synchronously, that is, reducers where not invoked until all mappers finished their corresponding tasks. In an all-to-all shuffle context, it is not necessary for all mappers to finish to start the reduce stage, as reducers can start reading intermediate files even from the termination of the first mapper. We remove the barrier between the map and the reduce stages, and invoke the reducers when a percentage P of mappers have finished their tasks. Based on empirical analysis, we have set P to 20%. Overlapping mappers and reducers this way improves execution time in two ways.

- Eliminates unnecessary delays in reducer invocation.
- As we will mention in the following sections, we have optimized Primula with concurrent I/O operations for intermediate files. Launching the reducers before the termination of all the mappers avoids all the intermediate files to be transferred concurrently, saturating the SOSS and degrading the overall performance. Asynchronous execution dilates the read of the intermediate files in a wider time lapse, without altering the total execution time.

4.2.4 Speculative execution

In any distributed and highly parallel workloads, outlier functions with abnormally high execution times (known as stragglers) can appear, even if their tasks are load-balanced,. Such functions may emerge due to hardware issues, network congestion or byzantine faults. In Primula, we implemented a straggler mitigation system using speculative execution. A specific monitor functions tracks all the functions in a job, and each of the function communicates its progress in real time through COS. When a relatively slow function is detected, an equivalent function is launched with the same input and output parameters.

4.2.5 Inferring the optimal level of parallelism

As the data exchange step is the bottleneck for most data analysis operation involving an all-to-all shuffle, we can minimize execution time by determining the optimal scale of parallelism of the shuffle. Optimizing the shuffle implies reaching a compromise between bandwidth and throughput limits of the storage system. In Primula we proposed an analytic method for the optimization task. We developed a mathematical model to predict an approximation for the shuffle time for a certain configuration. We will not get into details about the model assumptions and limitations, as they are already explained in our previous paper. Basically, we infer the I/O time of mappers and reducers (eq. 4.1). D is the size of the dataset in bytes. p is the number of workers in a single stage. b_w and b_r are the cloud functions-SOSS bandwidth limits for writing and reading, respectively, in bytes per second. q_w and q_r are the cloud functions-SOSS throughput limits for writing and reading, respectively, in operations per second. c is the I/O chunk size, in bytes.

$$T_{\text{sort}}(p) = T_{\text{map}}(p) + T_{\text{rdc}}(p) \tag{4.1}$$

For each approximation, we calculate the throughput-limited and bandwidth-limited I/O times, an consider the maximum of the both. In plain words, we determine if each operation is constrained by the cloud functions-SOSS throughput or the bandwidth. Eqs. 4.2 and 4.3 approximate the I/O time for the map and reduce stages, respectively.

$$T_{\text{map}}(p) = \underbrace{\max\left\{\frac{D}{b_r \times p}, \frac{\left\lceil \frac{D}{c \times p} \right\rceil p}{q_r}\right\}}_{\text{Reading of input share}} + \underbrace{\max\left\{\frac{D}{b_w \times p}, \frac{\left(\left\lceil \frac{D}{c \times p^2} \right\rceil p^2\right)}{q_w}\right\}}_{\text{Writing of intermediate data}}. \tag{4.2}$$

$$T_{\text{rdc}}(p) = \underbrace{\max\left\{\frac{D}{b_r \times p}, \frac{\left(\left\lceil \frac{D}{c \times p^2} \right\rceil p^2\right)}{q_r}\right\}}_{\textit{Reading of intermediate data}} + \underbrace{\max\left\{\frac{D}{b_w \times p}, \frac{\left\lceil \frac{D}{c \times p} \right\rceil p}{q_w}\right\}}_{\textit{Writing of processed data}}. \tag{4.3}$$

To measure bandwidth and throughput values, we evaluate the aggregate number of 64MB GET and PUT operations that can perform an increasing number of workers in a time lapse.

Once we reach a peak and results decrease, we retain the maximums as our bandwidth and throughput values. For the experiments in this document we used the results in Tab. 4.1.

	Bandwidth	Throughput
GET requests	20.75e3 MB/s	342.19 Ops
PUT requests	23.30e3 MB/s	364.04 Ops

Table 4.1: Throughput and bandwidth measurements between IBM Cloud Functions and IBM Cloud Object Storage in *us-east*. Ops: operations per second.

This work aims to validate the applicability of purely *serverless* architectures on Big Data analysis. In data analysis pipelines, several queries or transformations are concatenated traversing multiple data exchange steps. The volume of data between steps is also variable. For example, the TCP-DS query 94 involves 8 steps, with input data of each step varying between 0.8MB and 66GB [34]. A model that measures the shuffle time in isolation, as an autoconclusive operation, does not properly fit such context. In a data analysis pipeline, it would be of greater interest to infer the number of workers at each step in real time, adjusting the number of workers dynamically based on the data volume to exchange.

We complement Primula's automatic inference system to model only the shuffle stage, excluding the initial read of the input and the final write of the output. Removing the leftmost term of eq. 4.2 and the rightmost term of eq. 4.3 we get an alternative model that only considers the I/O of intermediate data, in a conceptually analogous way to the original. Eq. 4.4 approximates the intermediate data write time for mappers and eq. 4.4 approximates the intermediate read time for reducers. Substituting both expressions in eq. 4.1 we can foresee the execution time of the all-to-all shuffle.

$$T_{\text{map}}(p) = \underbrace{\max\left\{\frac{D}{b_w \times p}, \frac{\left(\left\lceil \frac{D}{c \times p^2} \right\rceil p^2\right)}{q_w}\right\}}_{\text{Writing of intermediate data}}. \tag{4.4}$$

$$T_{\text{rdc}}(p) = \underbrace{\max\left\{\frac{D}{b_r \times p}, \frac{\left(\left\lceil \frac{D}{c \times p^2} \right\rceil p^2\right)}{q_r}\right\}}_{\textit{Reading of intermediate data}} \tag{4.5}$$

Our framework resolves both equations on the fly before each execution, searching a time minimum across the range of possible scales of parallelism. The resolution is performed immediately, without the involvement of the user. Base throughput and bandwidth values are calculated automatically before the first usage of the system in the user machine, and the obtained parameters are saved in a local file for posterior execution.

4.3 Improving I/O performance

We put great effort on improving the performance of the all-to-all shuffle, a common bottleneck for big data analysis pipelines. Our I/O optimizations address two aspects of function communication: performing concurrent requests from each client in an efficient manner and digging into the most productive data formats for the intermediate data exchange.

4.3.1 Concurrent I/O

IBM Cloud Functions, as AWS Lambda, increases the number of CPUs available by a cloud function in proportion to the allocated runtime memory. From an architecture perspective, it would be perfectly feasible to run multiple simultaneous threads inside an IBM Cloud Function. In Python programs, this possibility is restricted by the Global Interpreter Lock (GIL). The GIL is part of CPython, the standard Python implementation, and ensures that only one thread is executed at a time in a Python program. The reasons for the implementation of the GIL are diverse; gives better performance than fine-grained blocking for single-threaded applications and thread-unsafe C extensions are easier to integrate, for example. It is a constraint for CPU-bound multi-threaded applications in favor of single-threaded ones. I/O bound programs can exploit the advantages of threading though, as Python forces the thread switch at blocking operations such as I/O operations. This way, we can overlap the execution time of several threads, switching the CPU across threads as they get blocked in their I/O requests.

The naïve option for multi-threaded applications in Python is the `threading` library. `threading` allows the execution of a pool of threads orchestrated by the operating system. Although I/O performance can be improved, `threading` still carries some drawbacks that avoids reaching the maximum efficiency. First, threads are managed by the operating system explicitly, which appends thread coordination overhead to the execution time. Second, the operating system can switch between threads even if the one owning the CPU has not started a blocking request, through preemptive multitasking.

`asyncio` is an alternative library for concurrent execution in Python, that runs multiple tasks (called co-routines) in a single thread. It is based on a main event loop. Co-routines queue events as they progress, and when the current co-routine gets blocked the event loop switches to the first thread that queued an event. Co-routines communicate events at their start and after finishing a blocking call. `asyncio` reduces the thread-management overhead, makes a more efficient scheduling of concurrent tasks and, as co-routines are lighter than real Python threads, decreases the startup time of concurrent section of code.

Implementation	Reading	Writing
`asyncio`	3.138±0.274s	5.506±0.133s
`threading`	3.454±0.117s	6.580±0.182s
sequential	6.027±0.331s	9.748±0.637s

Table 4.2: `asyncio` vs `threading` vs sequential I/O performance. We performed the evaluation in *us-east*, with a single cloud function per execution and 3 replicas.

We evaluated the time it takes for a IBM Cloud function to read and write 10 objects of 64MB, in a sequential implementation, with `threading` and with `asyncio`. `asyncio` overcomes the performance of `threading` in a 9.1% in the reading and in a 16.3%. Although the improvement may seem humble, we must consider that the evaluation was performed with only 10 concurrent tasks (one per file), and the superiority of `asyncio` should be reinforced as the scale of concurrency increases.

4.3.2 Optimizing intermediary objects

`pandas` [8] dataframes are the core data structure of our framework. However, we evaluated the possibility of using distinct data formats for the intermediary objects of the shuffling. Parquet [2] is a columnar storage data format adapted to big data volumes, unlike CSV, a row-based format. Its performance stands out specially with complex data and structured datasets with non-primitive or lax size data types. In Python, support to the parquet format is given through Apache Arrow's `pyarrow` [9] library.

Apart from choosing performant data formats, we have also focused on reducing the volume of data transferred in the shuffle. From different data compression techniques, `pyarrow`'s integrated `snappy` returned the best results in terms of compression time. We compared the performance of three technologies for our intermediate data, looking at the transformation time from a `pandas` dataframe and the size of the resulting bytes object: Python's `pickle` library, `pyarrow`'s sole parquet conversion and `pyarrows` parquet conversion with `snappy` compression. We performed our evaluation onf the `Brain02_Bregma1-42_02_v2.csv` dataset of 150MB from EMBL's Metaspace project [7].

Implementation	To `pandas`	From `pandas`	Object size
`pyarrow` + `snappy` compression	0.058±0.009s	0.357±0.088s	29085561B
`pyarrow`	0.073±0.010	0.393±0.082s	-
`pickle`	0.034±0.007s	0.754±0.066s	75657107B

Table 4.3: `pyarrow` vs `pickle` performance. Evaluations where performed in a IBM Cloud Function in *us-east*, with 5 replicas per measurement. The test dataset was `Brain02_Bregma1-42_02_v2.csv`, extracted from EMBL's Metaspace project [7]. `pyarrow` was executed sequentially.

Although `pickle` outperformed `pyarrow` in the transformation from intermediate data to `pandas`, the difference is negligible in magnitude compared to an speedup of 2.112 in the conversion from `pandas` to intermediate data with `pyarrow` & `snappy`. In terms of volume to transfer, the resulting object size decreased in a 160% using the compressed parquet format.

We also evaluated the possibility of performing the conversion between `pandas` and the intermediate object format (using `pyarrow` and `snappy`) inside the `asyncio` co-routines, or instead executing the conversion sequentially before the I/O stage and only performing I/O requests from the co-routines. For a partition size of 1GB per worker, during the map stage, converting the data sequentially before the I/O stage got an speedup of 1.15 against the concurrent I/O and conversion. This is coherent with the previously explained GIL limitation, as the conversion to/from a `pandas` dataframe is a completely CPU-bound task.

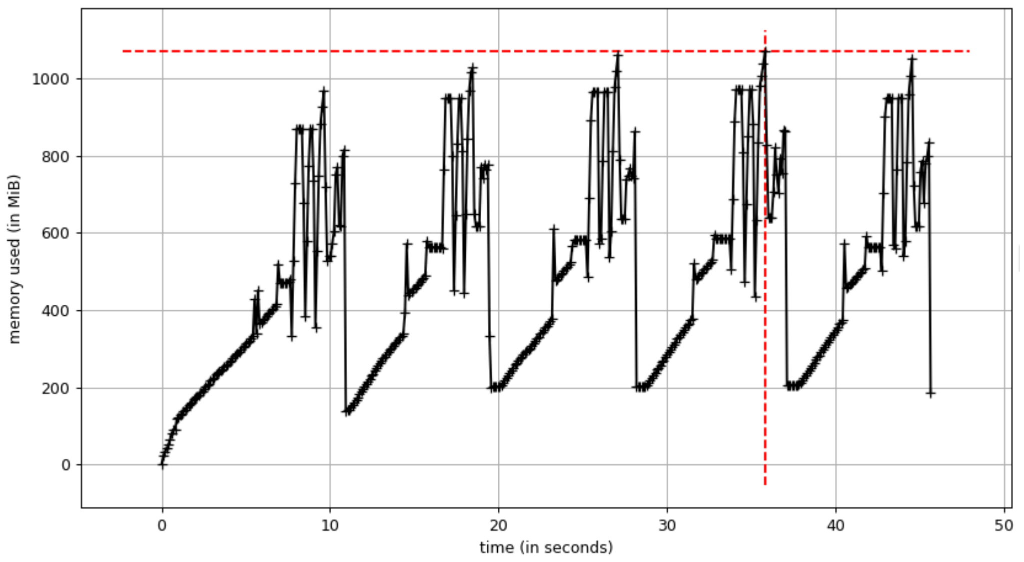
4.4 Memory usage in cloud functions

Memory is a limited resource in cloud functions. IBM Cloud Functions, for instance, can be a assigned a maximum of 2048 MB runtime memory. In addition, detecting memory overflows in cloud functions is not always evident, and the logging service can sometimes provide ambiguous information. An efficient management of the memory usage in cloud functions (a) permits processing a greater partition in each function, which in some cases can enhance the cost-effectiveness of an application, (b) diminishes the possibility of running over memory and eases function monitoring.

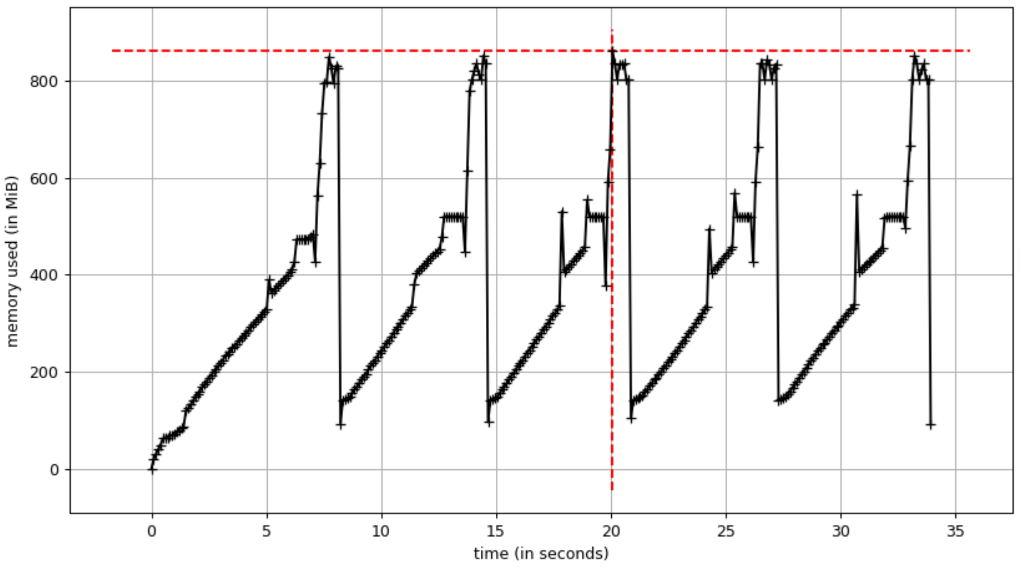
Keeping a single dataframe per function an using `pandas` views for sub-partitioning we avoid generating needless copies of the data at improve memory management. `pandas` views return a *shallow* copy of the data, that is, a different data object with a pointer to the original object's data. Instead of *deep* copies, which return a completely new and independent object, views have not own data (except for the metadata about the original data they are pointing at). Views and the original data are linked: modifications in one instance modifies the other, and vice versa. In the specific context of `pandas` the original dataframe and the *view* dataframe point to the same underlying `numpy` array instances.

For our specific `groupBy` case, we use views both in the map and reduce stages for grouping. In the mapper, we apply the a hash partitioning algorithm on the keys and we generate a `numpy` array with the rows for each of the reducers. Then, per reducer, instead of copying its assigned data to generate its intermediate files, we extract a view of the original dataframe with its corresponding entries, and we serialize the view into the intermediate format. The main objects in memory at its peak are, thus, only the original dataframe and the intermediate compressed parquet byte strings.

Using views we can also avoid creating underlying copies of the data. The straightforward solution for grouping a dataframe is to permutate its rows and rearrange them based on the grouping criteria. In `pandas`, permuting the rows of a dataframe internally generates a complete copy of the dataframe, even if performed in-place. With views, and the system proposed above, we eliminate every permutation from the process and stick to only the original data.



(a)



(b)

Figure 4.3: Memory usage in the `groupBy` using views and permuting rows. (a) Results for a purely views-based implementation. (b) Results for a row permutation-based implementation. Experiments where executed locally, over a triplicated version of the Metaspace `Brain02_Bregma1-42_02_v2.csv` dataset (450MB), and data was grouped into 10 subpartitions. We used Python's `memory-profiler` for monitoring memory usage.

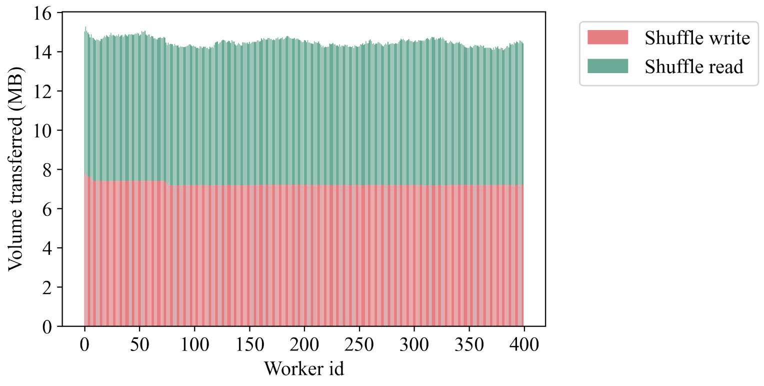
We evaluated the performance (in memory usage and execution time) of a `groupBy` operation using only views and permuting the data, and demonstrated that a views-based implementation outperforms dataframe permutation. Both implementations got as input a `pandas` dataframe from a triplicated version of the Metaspace `Brain02_Bregma1-42_02_v2.csv` dataset (450MB). They performed the partial grouping of the map stage, grouping the keys using hash partitioning into 10 sub-partitions, and serialized every sub-partition into the intermediate compressed parquet format. The *views version* calculated the destination sub-partition for each row, and serialized a view of the data for each sub-partition with its corresponding rows. The *permutation version* permuted the dataframe so that entries with the same destination remained adjacent, extracted a view for each sub-partition (unlike the *views version*, it was composed of consecutive rows) and serialized it.

We executed the experiment locally, and executed the grouping five times per implementation. The pure views version got an average execution time of 6.488s, whereas the permutation version got 8.986s. We profiled the memory usage of both implementations using Python's `memory-profiler` package [6]. Regarding memory consumption, the pure views version shows a maximum memory peak slightly over 800MB, whereas the permutation version surpasses 1000MB memory usage (Fig. 4.3).

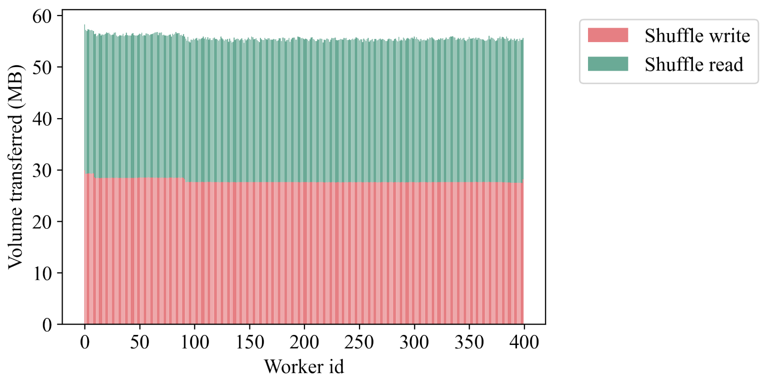
4.5 Using compiled code.

CPython can be extended with C-like compiled code extensions using the `cython` library [3]. Compiled code is specially interesting for CPU-bound tasks with primitive types and basic data structures. We use a `cython` routine for the hash-partitioning algorithm. Hashing is a relatively heavy operation in computational terms, and hashing an array of keys iteratively in high-level languages like Python can be a burden for performance. We considered and compared different alternatives for the hashing. As `pandas` columns are internally `numpy` arrays, we could use `numpy` utilities, which are frequently based on compiled code, to boost performance. `numpy` provides the capability to vectorize an scalar function to work on full arrays, and functions like `apply_along_axis` to apply a scalar function over the elements of the array. Also, we considered using compiled hashing implementations instead of high-level alternative. We evaluated the alternatives on different size and data type key arrays and observed that using compiled code with Python's default `hash` function gave the best performance.

Python's `hash` implementation is derived from object `ids`, a unique identifier that is given to every object and that is not changed during its lifetime [10]. To validate that our chosen hash function does not generate excessive collisions and that data is distributed equitably across reducers, we evaluate the read (reducers) and written (read) data volume in the shuffle. We use two Metaspace datasets of 5.1 and 19.7GB respectively, and 400 workers (Fig. 4.4). As depicted, every reducer reads an comparable size of intermediate data, so we can consider our hash partitioning algorithm is balanced.



(a)



(b)

Figure 4.4: Load-balance analysis of the hash partitioning: written and read shuffle data volume per worker in the `groupby`. (A) CT26_Xenograft (5.1 GB). (B) X089-Mousebrain (19.7GB). We executed the `groupby` from a virtual machine in *us-east*, using 400 workers. Both datasets were extracted from the Metaspace project.

To mention, we also use compiled code for the conversion from csv to `pandas` dataframe, using the integrated C engine in `panda`'s `read_csv`.

Chapter 5

Implementation

5.1 Base framework

Our project extends the serverless framework Lithops [36, 38]. Lithops provides interfaces to implement `map`-like parallel workloads, and is specially suited for data-level parallelism and embarrassingly parallel jobs. Lithops is currently maintained by IBM[1]. Figure 5.1 depicts a diagram of the Lithops execution flow.

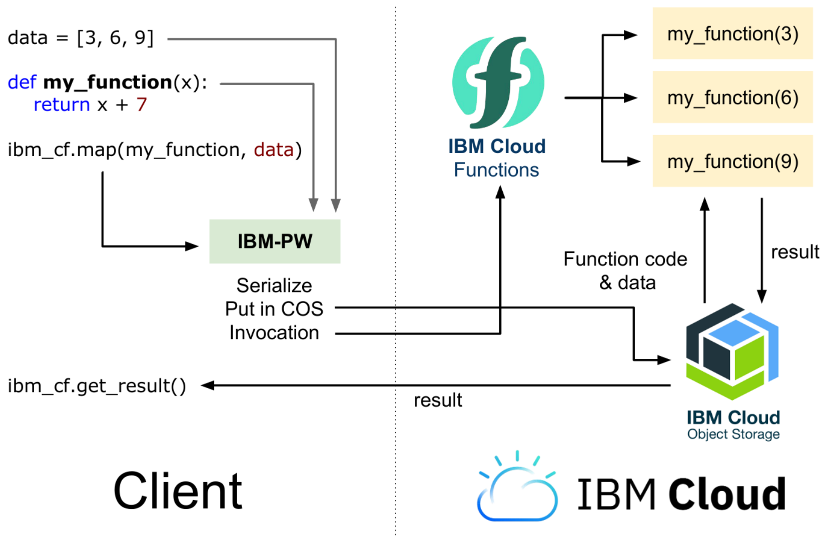


Figure 5.1: Execution flow of Lithops (previously IBM-PyWren), extracted from [36].

[1]https://cloud.ibm.com/docs/codeengine?topic=codeengine-lithops

As mentioned in chapter 2, Lithops lacks all-to-all shuffle operators, `sort` primitives or `groupBy` primitives. In Primula [21] we extended Lithops with a shuffle operator and a sort primitive. For the current work, we augment the interface with an automatic `groupBy` operator. We take advantage of Lithops primitives to execute our map and reduc stages, and reuse our barrierless execution and speculative execution patches from Primula. To demonstrate the simplicity and transparency of our `groupBy` call, we include an example snippet in code 5.1. In the existing serverless frameworks, the user would have to provision resources in advance, write the `groupBy` code explicitly and input the number of workers to use. With our Lithops extension, the user only has to put its IBM Cloud API Keys into the JSON configuration file (see Annex B), call the `groupBy` operator and the frameworks completes the workload automatically.

Listing 5.1 `GroupBy` operation in our framework.

```
# We import our modified lithops library with sort and groupby.
import lithops
import json

config = json.load(open("my_config.json"))
lh = lithops.ibm_cf_executor(config=config, runtime_memory=mem)
lh.groupby("cos://us−east/my−bucket/my−data.csv", primary_key_column=0)
```

5.2 I/O and memory optimizations

We intended to keep our I/O implementation simple, to keep to door open to future extensions of the framework. Codes 5.2 and 5.2 show the code for the conversion from `dataframe` to compressed parquet and vice versa. To generate views from dataframes we use `pandas`' `iloc` accessor, which returns a view of a set of dataframe rows based on their row index. For from/to parquet conversion we also use `pandas`'s integrated `pyarrow` and `snappy` utilities, the sames we have used in the evaluations described at chapter 4.

Listing 5.2 Dataframe partitioning using views an conversion to the intermediate byte string with `pyarrow` and `snappy`.

```
# ed is the partition dataframe.
# pointers_ni contains, from lower_bound to upper_bound, the row ids
# correspondent to a certain reducer.
# f is a BytesIO object.
```

```
ed.iloc[pointers_ni[lower_bound:upper_bound]]
  .to_parquet(f, engine="pyarrow", compression="snappy")
```

Listing 5.3 Read of intermediate compressed parquet data.

```
# Read object is the intermediate byte string wrapped in a
# BytesIO object.
import pandas as pd
df_ = pd.read_parquet(read_obj_)
```

In Code 5.4 we show the intermediate data writing process by a mapper using `asyncio`. As serializing is performed sequentially before the write operations, we can isolate the I/O portion of the code, providing modularity and readability for upgrades. We minimize co-routines to the minimum, only including the I/O call from the IBM COS client and reducing CPU usage.

Listing 5.4 Writing intermediate data with `asyncio`.

```
# bounds is a list of (reducer_id, serialized_data, chunk_number)
# tuples.
async def _writes(bds):
  loop = asyncio.get_event_loop()

  def _write_func(Bucket, Key, Body):
    ibm_cos.put_object(Bucket=Bucket, Key=Key, Body=Body)

  objects = await asyncio.gather(
    *[
      loop.run_in_executor(None, functools.partial(
        _write_func, Bucket=output_bucket,
        Key="{}/{}/{}.pickle".format(my_output_path, b[0], b[2]),
        Body=b[1]))
      for b in bds
    ]
  )
  return objects

loop = asyncio.get_event_loop()
tts = loop.run_until_complete(_writes(bounds))
```

5.3 Integration of `cython`

`cython` extensions are written in its own *pythonish* code at `.pyx` files, which are compiled in two general steps. In Code 5.5 we show our hash partitioning in `cython`. First, `cython` converts its pythonish code into standard C code. Then, it calls the default C compiler of the system (`gcc`, for instance) and performs the compilation the usual way from the source C code.

There are different ways for integrating `python` libraries with `cython` code. For consistency and portability reasons, we decided to extend Lithops with the intermediate C files generated by `cython`. We adapted the `setup.py` file to compile the `.c` files automatically during the library installation, according to the system's architecture.

Listing 5.5 Hash partitioning in `Cython`.

```
@cython.boundscheck(False)
@cython.wraparound(False)
def chash(np.ndarray key_array,
  unsigned int row_number, unsigned int reducer_number):
  cdef int i
  # allocate number * sizeof(double) bytes of memory
  cdef unsigned int * index_array =
    <unsigned int *> malloc(row_number * sizeof(double))
  if not index_array:
    raise MemoryError()

  try:
    for i in range(row_number):
      index_array[i] = hash(key_array[i]) % reducer_number
    return [x for x in index_array[:row_number]]

  finally:
    # return the previously allocated memory to the system
    free(index_array)
```

Chapter 6

Evaluation

The evaluation tackles the two overriding aspects of our proposal. First, if our model is capable of predicting a close to optimal number of workers for the all-to-all exchange operation with an acceptable error, in the `groupBy` context. Second, if the performance of the system suits data analysis. We executed all the experiments from a VM located at *us-east*, and 2048MB runtime memory per cloud function.

6.1 Is the inference model generalizable?

Probably the main uncertainty after validating our model for the `sort` in Primula [21] was its generalizability to other operations. For two different datasets, we compared the real execution and shuffle times with different scales of parallelism to their theoretic counterparts using our inference equations. We chose two datasets from the Metaspace project [7], CT26_Xenograft.csv, of 5.1GB, and X089-Mousebrain_842x603.csv, of 19.7GB.

We calculate the shuffle time of each execution by measuring the time lapse between the first mapper that initiates the intermediate data writing and the last reducer that reads its correspondent intermediate data. We perform this measurement by making each function signal its corresponding milestone -start of writing for mappers, end of reading for reducers- with a unique IBM COS object, and subtracting the time stamp of the latest object with that of the earliest one.

An important consideration about the results is that our model is not an exact execution time predictor, as it does not consider many factors such as computation or intermediate data serialization. Instead, its objective is to foresee which worker configuration will give the best performance, in terms of execution time, based on the data exchange.

Fig. 6.1 represents our results for CT26_xenograft. In both time calculations, the theoretical and empirical number of workers that give minimum execution time coincide. Surprisingly, full execution time results are more concordant with the theoretic forecast than shuffle results. The shuffle time does not show great variation for different scales of parallelism, probably because 5.1GB is still a modest size for data analysis and we have not reached IBM COS congestion.

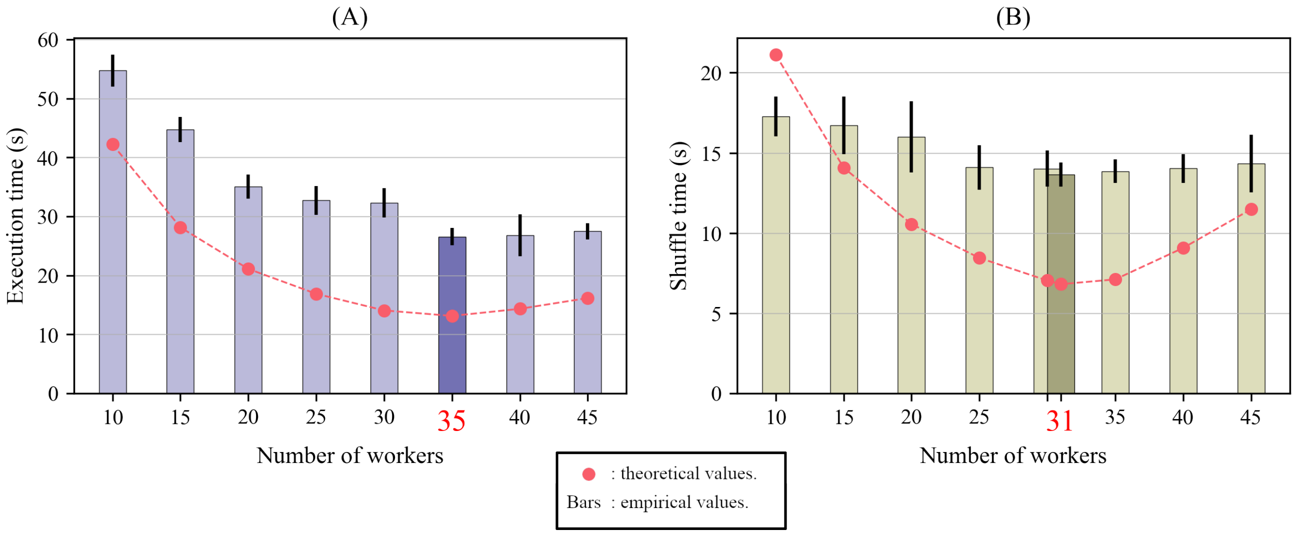


Figure 6.1: Empirical vs theoretical results for the 5.1GB dataset. (A) Results for the total execution time. We used the complete version of the inferencer equation. (B) Results for the shuffle time. We used the shuffle-narrowed version of the equation.

Fig. 6.2 represents our results for X089-Mousebrain_842x603. Empirical results show a flattening of the results once a value close to the minimum is reached in execution and shuffle times. The appropriate scale of parallelism to minimize execution time is predicted correctly. In the shuffle time, we get an absolute minimum, still similar to the predicted minimum, at 60 workers, more than the predicted minimum point. For a similar execution time, minimizing the number of cloud functions is favorable to decrease the billing related to function run time, so we consider the model to be valid also with the 19.7GB dataset.

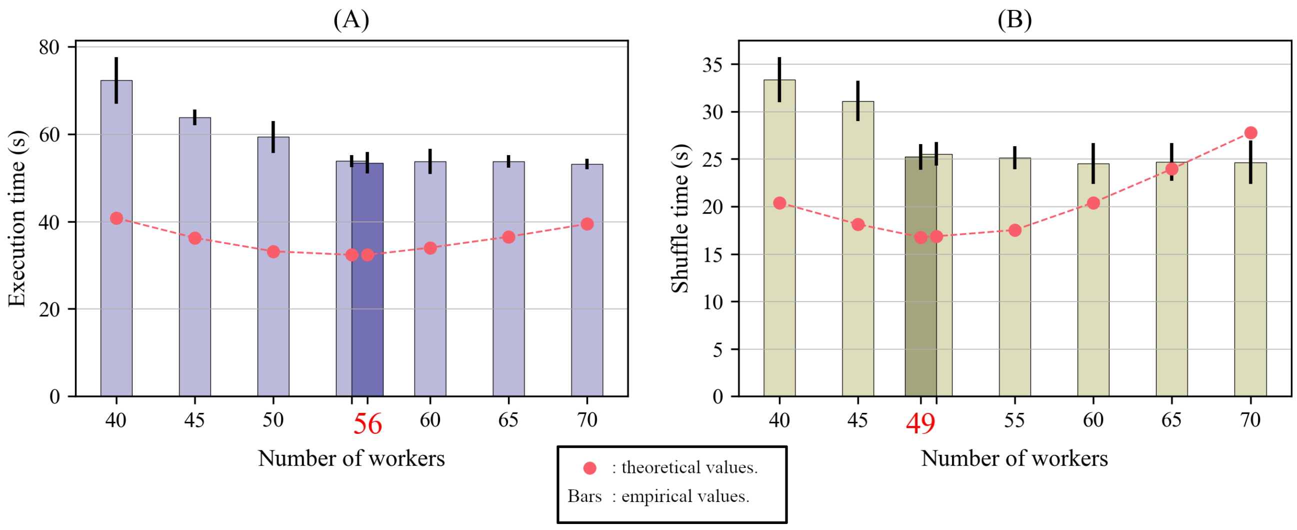


Figure 6.2: Empirical vs theoretical results for the 19.7GB dataset. (A) Results for the total execution time. We used the complete version of the inferencer equation. (B) Results for the shuffle time. We used the shuffle-narrowed version of the equation.

Overall, we see that both prediction models have a good inference capacity. Although the divergence between the theoretic and the real shuffle and execution times is evident, we insist on the idea that our model does not intend to reach great precision in its inferenced times. From a practical perspective, we can affirm than the optimal scale of parallelism determined by the predictor and the real configuration are sufficiently proximal. We can thus confirm our hypothesis that the model is generalizable to distinct shuffling operations. Supposing a future integration of the model into complex data analysis pipelines, as cloud functions can be launched dynamically at runtime without previous provisioning of resources, we could execute the model at each shuffle stage and scale the parallelism accordingly.

6.2 Performance validation

To assess the performance improvements consequence of our optimizations, we executed a 100GB Terasort [33] and compared our results with our previous achievements. In Primula, we accomplished a minimum execution time of 195.4s for the 100GB Terasort with 200 workers. With the enhancement presented in this work, we achieved a 89.9s performance for the total execution time and 50.83 for the shuffle time with 225 workers. The execution time speedup was 2.17.

In Tab. 6.1 we include `groupBy` performance results for large scale datasets, that validate the applicability of our operator in greater size analytics. Datasets were generated through fragmentation of X089-Mousebrain_842x603, replication of fragments and random rearrangement and concatenation of the fragments.

Dataset size	Number of workers	Execution time	Shuffle time
39.4GB	169	62.304±6.753s	35.261±2.856
59.1GB	225	71.957±5.898	46.497±7.896s
78.8GB	324	87.105±3.615s	59.902±3.118s

Table 6.1: `groupBy` performance with large scale datasets. Datasets were generated replicating the X089-Mousebrain_842x603 dataset from the Metaspace project.

Chapter 7

Conclusion

We have tackled one of the main issues of the FaaS model successfully -the execution of I/O-intensive workloads- and we have demonstrated that their performance can be modeled and optimized. Still keeping a simple FaaS+SOSS architecture we overcome unnecessary resource management issues and we are able to optimize bottleneck shuffle stages analytically. We also contradict previous suggestions about the incompatibility of cloud functions with Big Data analysis [23] and the inappropriateness of using SOSS as the communication intermediary [34].

Enabling the transparent access to remote resources for not specialized programmers is one of the final milestones of cloud computing. Our present results, along with our preceding work [21], suggest that automatically managed data analytics in serverless environments are possible and could be integrated into research and industry domains in the future. However, we have only focused our evaluation on the IBM Cloud, and the performance of parallel workloads is still highly variable between cloud platforms [17]. As future work, our model should be validated in different platforms and at different scales. Currently, it is the user's responsibility to select the appropriate cloud provider for its workloads, but the cloud research community is already working on utilities for the transparent switch between cloud providers [35].

Regarding performance and applicability, we implement a `groupBy` operator on top of a completely *serverless* architecture with competitive performance. We propose an innovative solution for Big Data analytics, giving the user the possibility to process datasets with a magnitude of hundred GB from any kind of laptop and with just an IBM Cloud user account.

A future step of our research track is the integration of the shuffle inferencer into multi-step data analysis pipelines so that the parallelism level can be scaled dynamically. In this work we have generalized the model to overall all-to-all shuffle operations, but still remains to demonstrate if it fits alternative distributed architectures and emerging serverless communication technologies. The data exchange performance of intermediary-less cloud function communication systems [40] and partially *serverful* frameworks [27], for instance, could potentially be predicted analytically with a similar methodology.

Bibliography

[1] Amazon EMR. https://aws.amazon.com/emr. Accesed: 2021-06-05.

[2] Apache Parquet. https://parquet.apache.org/. Accesed: 2021-06-07.

[3] cython. https://cython.org/. Accesed: 2021-06-07.

[4] Google Colab. https://colab.research.google.com. Accesed: 2021-06-05.

[5] Ibm Cloud Code Engine. https://www.ibm.com/cloud/code-engine. Accesed: 2021-06-05.

[6] memory-profiler. https://github.com/pythonprofilers/memory_profiler. Accesed: 2021-06-07.

[7] Metaspace Project. https://metaspace2020.eu/group/EMBL. Accesed: 2021-06-07.

[8] Pandas. https://pandas.pydata.org/pandas-docs/stable/. Accesed: 2021-06-07.

[9] PyArrow. https://arrow.apache.org/docs/python/. Accesed: 2021-06-07.

[10] Python `hash`. https://docs.python.org/3/glossary.html#term-hashable. Accesed: 2021-06-07.

[11] The R Project for Statistical Computing. https://www.r-project.org/. Accesed: 2021-06-07.

[12] Redis. https://redis.io/. Accessed: 2021-06-06.

[13] TPC-DS. http://www.tpc.org/tpcds/. Accesed: 2021-06-07.

[14] TPC-H. http://www.tpc.org/tpch/. Accesed: 2021-06-06.

[15] Aitor Arjona, Pedro García López, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow: Trigger-based orchestration of serverless workflows. *Future Generation Computer Systems*, 124:215–229, 2021.

[16] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[17] Daniel Barcelona-Pons and Pedro García-López. Benchmarking parallelism in faas platforms. *Future Generation Computer Systems*, 2021.

[18] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Sujit Bebortta, Saneev Kumar Das, Meenakshi Kandpal, Rabindra Kumar Barik, and Harishchandra Dubey. Geospatial serverless computing: Architectures, tools and future directions. *ISPRS International Journal of Geo-Information*, 9(5), 2020.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[21] Germán T. Eizaguirre. *A Serverless Sort for Scalable Analytics in the IBM Cloud.* Bachelor thesis, Universitat Rovira i Virgili, 2020.

[22] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[23] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research. *arXiv e-prints*, page arXiv:1708.08028, August 2017.

[24] Reihaneh H. Hariri, Erik M. Fredericks, and Kate M. Bowers. Uncertainty in big data analytics: survey, opportunities, and challenges. *Journal of Big Data*, 6(1):44, Jun 2019.

[25] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[27] Youngbin Kim and Jimmy Lin. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455, 2018.

[28] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[29] Mariano Ezequiel Mirabelli, Pedro García-López, and Gil Vernik. Bringing scaling transparency to proteomics applications with serverless computing. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, page 55–60, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery.

[31] J. Nupponen and D. Taibi. Serverless: What it is, what to do and what not to do. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 49–50, Los Alamitos, CA, USA, mar 2020. IEEE Computer Society.

[32] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait, and Samir Belfkih. Big data technologies: A survey. *Journal of King Saud University - Computer and Information Sciences*, 30(4):431–448, 2018.

[33] Owen O'Malley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pages 1–3, 2008.

[34] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 193–206, USA, 2019. USENIX Association.

[35] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona. Toward multicloud access transparency in serverless computing. *IEEE Software*, 38(01):68–74, jan 2021.

[36] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 1–8, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. Primula: A practical shuffle/sort operator for serverless computing. In *Proceedings of the 21st International Middleware Conference Industrial Track*, Middleware '20, page 31–37, New York, NY, USA, 2020. Association for Computing Machinery.

[38] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, April 2021.

[39] Jin Wang, Yaqiong Yang, Tian Wang, R. Simon Sherratt, and Jingyu Zhang. Big data service architecture: A survey. *Journal of Internet Technology*, 21(2):393–405, 2020.

[40] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. Boxer: Data analytics on network-enabled serverless platforms. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[41] Chaowei Yang, Qunying Huang, Zhenlong Li, Kai Liu, and Fei Hu. Big data and cloud computing: innovation opportunities and challenges. *International Journal of Digital Earth*, 10(1):13–53, 2017.

[42] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[43] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 653–669. USENIX Association, April 2021.

Appendix A

Annex: Code availability

The code for our framework, specially our `groupBy` operator, is available at the private Gitlab repository of the CloudLab group (universitat Rovira i Virgili, Tarragona). Access will be conceded under request.

https://git.cloudlab.urv.cat/geizaguirre/primula_original.

Appendix B

Annex: Lithops configuration

To configure Lithops, refer to its official GitHub page, which includes intuitive and simple guides with all the necessary steps.

https://github.com/lithops-cloud/lithops