NATIONAL TECHNICAL UNIVERSITY OF ATHENS

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

INTERDEPARTMENTAL PROGRAM OF POSTGRADUATE STUDIES
«DATA SCIENCE AND MACHINE LEARNING»

DEPARTMENT OF INFORMATION TECHNOLOGY AND COMPUTERS
COMPUTING SYSTEMS LABORATORY

Development of Large-Scale Big Data System with State Management on Serverless Cloud Architectures

MASTER THESIS

of

Nikolaos Nikitas

Supervisor: Nectarios Koziris
Professor, NTUA

Athens, July 2021

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
«ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ»
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Ανάπτυξη Συστήματος Μεγάλης Κλίμακας Δεδομένων με Διατήρηση της Κατάστασης σε Serverless Αρχιτεκτονικές Υπολογιστικών Νεφών

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Νικολάου Π. Νικήτα

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Ιουλίου 2021.

....	....	....
Νεκτάριος Κοζύρης	Ιωάννης Κωνσταντίνου	Βάνα Καλογεράκη
Καθηγητής	Επικ. Καθηγητής	Καθηγήτρια
Ε.Μ.Π.	Παν. Θεσσαλίας	Οικ. Παν. Αθήνας

Αθήνα, Ιούλιος 2021

. . . .

Νικόλαος Π. Νικήτας
Κάτοχος Μεταπτυχιακού Διπλώματος στον τομέα της
Επιστήμης Δεδομένων και Μηχανικής Μάθησης

Copyright © Νικόλαος Π. Νικήτας 2021
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Τα τελευταία χρόνια, υπήρξε εμφανές ενδιαφέρον για τον τομέα των Δεδομένων Μεγάλης Κλίμακας. Επιπλέον, υπήρξε μεγάλη προσπάθεια για την εφαρμογή και βελτιστοποίηση των κατανεμημένων πλαισίων ανάλυσης Δεδομένων Μεγάλης Κλίμακας, όπως το Apache Spark και το Hadoop. Πιο συγκεκριμένα, δίνεται έμφαση από τη βιομηχανία αλλά και τον ακαδημαϊκό χώρο στη βελτίωση της μεταφοράς μέσω του δικτύου ενδιάμεσων δεδομένων μεταξύ των μονάδων υπολογισμού MapReduce, δηλαδή, στη διαδικασία τυχαίας μεταφοράς δεδομένων, γνωστή και ως διαδικασία shuffling, μεταξύ των σταδίων ενός φόρτου εργασίας τύπου MapReduce, το οποίο είναι υψίστης σημασίας και παραμένει ένα σοβαρό εμπόδιο.

Το Apache Spark είναι ένα ευρέως χρησιμοποιούμενο σύστημα επεξεργασίας Μεγάλων Δεδομένων. Ωστόσο, η shuffling διαδικασία έχει κάποιες προκλήσεις που πρέπει να αντιμετωπιστούν: η συμφόρηση I/O που δημιουργείται από την περιορισμένη απόδοση των σκληρών δίσκων όταν τα τυχαία ενδιάμεσα δεδομένα είναι μικρού μεγέθους και πρέπει να διαβαστούν, η έλλειψη ανοχής σε σφάλματα σε περίπτωση βλάβης Worker κόμβου ή απομάκρυνσης κόμβου από το cluster καθώς τυχόν αποθηκευμένα ενδιάμεσα δεδομένα τοπικά θα χαθούν μόνιμα, και η απουσία προσαρμογής σε περιβάλλοντα με containers με μεμονωμένες και χωρίς διατήρηση της κατάστασης διαδικασίες, που εφαρμόζεται συχνά στο cloud.

Σε αυτήν την μεταπτυχιακή διπλωματική εργασία, στόχος μας είναι να υιοθετήσουμε τη serverless αρχιτεκτονική σε ευρέως αποδεκτά πλαίσια ανάλυσης Δεδομένων Μεγάλης Κλίμακας και να χειριστούμε αποτελεσματικά την ενδιάμεση κατάσταση. Έτσι, υλοποιήσαμε μια κατανεμημένη αρχιτεκτονική αποθήκευσης που είναι απομακρυσμένη από την υπολογιστική μηχανή, και θα διατηρήσει τυχόν ενδιάμεσα shuffle δεδομένα από την εκτέλεση εργασιών για επεξεργασία δεδομένων, και χρησιμοποιούμε το Spark για την εκτέλεση αυτών των εργασιών. Η υλοποίησή μας ονομάζεται Cherry, η οποία είναι μια κατανεμημένη υπηρεσία που αποθηκεύει και σερβίρει shuffle δεδομένα, είναι ανοιχτού κώδικα, διαθέτει ένα μηχανισμό για caching των δεδομένων, επεξεργάζεται τα shuffle δεδομένα σε επίπεδο task, και χρησιμοποιείται για serverless εργασίες ανάλυσης δεδομένων.

Το Cherry επιτρέπει την εκτέλεση μίας de-facto statefull εργασίας απρόσκοπτα με serverless τρόπο, εκμεταλλευόμενο μια απομακρυσμένη μηχανή αποθήκευσης, ενώ βελτιστοποιεί τον χρόνο εκτέλεσης της shuffle διαδικασίας χρησιμοποιώντας ένα μηχανισμό αποθήκευσης στην cache για τα ενδιάμεσα αυτά δεδομένα σε επίπεδο task. Η shuffle υπηρεσία που υλοποιήσαμε είναι υλοποιημένη πάνω στον πηγαίο κώδικα του Spark και επεκτείνει την υπάρχουσα shuffle διαδικασία, επιτυγχάνοντας μία ανεκτική σε

σφάλματα και ελαστική εκτέλεση αποθηκεύοντας την ενδιάμεση κατάσταση μεταξύ των σταδίων απομακρυσμένα. Αυτό διασφαλίζει ότι δεν θα υπάρξει απώλεια shuffle δεδομένων λόγω διακοπής ή σφάλματος κάποιου Worker κόμβου. Εκμεταλλευόμαστε αυτόν το διαχωρισμό της κατάστασης από την υπολογιστική μηχανή και παρουσιάζουμε επίσης έναν μηχανισμό εκ των προτέρων αποθήκευσης των shuffle δεδομένων στην cache σε επίπεδο task, ο οποίος «προθερμαίνει» προληπτικά το διαθέσιμο caching επίπεδο αποθήκευσης με μόνο τα απαραίτητα shuffle δεδομένα που πρόκειται να ζητηθούν με αποτελεσματικό τρόπο ώστε να αποφευχθούν καθυστερήσεις των εργασιών λόγω σημείων συμφόρησης I/O. Θα χρησιμοποιήσουμε επίσης το Kubernetes ως ενορχηστρωτή για containers, έτσι ώστε να προσεγγίσουμε τα ρεαλιστικά περιβάλλοντα μεμονωμένων διαδικασιών που υπάρχουν στο cloud.

Η αρχιτεκτονική και η υλοποίηση του Cherry αναλύονται διεξοδικά και απεικονίζεται με σαφήνεια πώς αλληλεπιδρά με το επιλεγμένο σύστημα επεξεργασίας Δεδομένων Μεγάλης Κλίμακας στην περίπτωσή μας, αποδεικνύοντας την πιθανή εύκολη ενσωμάτωσή του σε διαφορετικά πλαίσια ανάλυσης δεδομένων τύπου MapReduce. Η υλοποίησή μας δοκιμάζεται τόσο σε συνθετικούς όσο και σε πραγματικούς φόρτους εργασίας δεδομένων, έναντι του Apache Spark και του External Shuffle Service που ήδη διαθέτει, και παρουσιάζονται τα αποτελέσματα. Στο τέλος της διπλωματικής, συνοψίζουμε το έργο μας και παρουσιάζουμε πιθανές μελλοντικές επεκτάσεις αυτού.

Λέξεις κλειδιά

Big Data Analytics Frameworks, Apache Spark, Distributed Systems, Cloud Computing, Serverless Architecture, MapReduce, Shuffle Service, Kubernetes, DevOps

Abstract

Over the last years, there has been apparent interest in Big Data. Additionally, there has been great effort in implementing and optimising distributed Big Data analytics frameworks, such as Apache Spark and Hadoop. More specifically, there is emphasis by industry and academia on improving the all-to-all transfer over the network of intermediate data between the MapReduce computation units, i.e., the shuffle data operation between the stages of a MapReduce-like workload, which is of paramount importance and still remains a serious bottleneck.

Apache Spark is a widely used Big Data processing system. Nevertheless, its shuffle operation has some challenges that need to be addressed: the I/O bottleneck that is defined by the limited throughput by HDDs when shuffle intermediate data are small in size and need to be accessed, the lack of fault tolerance in case of a Worker node crash or deallocation since any intermediate data stored will be permanently lost, and the absence of adaptation to containerized environments with isolated and stateless processes that is common in the cloud.

In this specific master thesis our goal is to adopt the serverless paradigm in widely embraced large-scale data analytics frameworks and handle intermediate state efficiently. Thus, we implemented a distributed dissaggragated storage architecture that will maintain any intermediate shuffle data from the execution of analytics workloads and we utilize Spark to execute these workloads. Our implementation is named Cherry, which is an open-source distributed task-aware caching shuffle service for serverless analytics.

Cherry allows a de-facto statefull workload to be seamlessly executed in a serverless manner by exploiting a remote storage engine while optimizing the shuffle execution time by employing a novel caching mechanism for intermediate data in task-level. Our shuffle service is built on top of Spark and extends its existing shuffle operation, achieving both a fault-tolerant and an elastic execution by storing intermediate state between stages remotely. This ensures that there will be no shuffle data loss due to Worker node crashes. We take advantage of this state dissagregation and we also present a look-ahead task-aware caching mechanism which proactively "warms-up" its available caching layer with only the necessary shuffle data that are about to be requested in an efficient way to avoid workload delays due to I/O bottlenecks. We will also utilize Kubernetes as a container orchestrator so as to approach the realistic containerized environments of isolated processes that exist the cloud.

Cherry's architecture and implementation is thoroughly analyzed, and it is clearly

depicted how it interacts with the current Big Data processing system in our case, proving its seamless possible integration with different MapReduce-like analytics frameworks. Our implementation is tested both on synthetic and real data workloads, against Apache Spark and its existing External Shuffle Service, and the conducted results are presented. In the end of the thesis, we sum up our thesis and present possible future expansions of our work.

Key words

Big Data Analytics Frameworks, Apache Spark, Distributed Systems, Cloud Computing, Serverless Architecture, MapReduce, Shuffle Service, Kubernetes, DevOps

Ευχαριστίες

Ολοκληρώνοντας την παρούσα μεταπτυχιακή εργασία θα ήθελα να απευθύνω τις ευχαριστίες μου σε όλους όσουν συνέβαλλαν σε αυτό το επίτευγμα.

Πρωτίστως, θα ήθελα να ευχαριστήσω τον κ. Νεκτάριο Κοζύρη, Καθηγητή Ε.Μ.Π. και επιβλέποντα της παρούσας εργασίας, που με εμπιστεύτηκε και μου έδωσε την ευκαιρία να ασχοληθώ με ένα τόσο ενδιαφέρον και σύγχρονο θέμα που συνδυάζει πολλαπλούς επιστημονικούς κλάδους. Οι γνώσεις που απέκτησα σε όλη αυτήν την πορεία αποτελούν εξαιρετικά εφόδια για τη μετέπειτα πορεία μου.

Επιπλέον, θα ήθελα να ευχαριστήσω ιδιαιτέρως τον κ. Ιωάννη Κωνσταντίνου, Επίκουρο Καθηγητή Παν. Θεσσαλίας, για την πολύτιμη καθοδήγηση και στήριξη που μου προσέφερε καθ'όλη τη διάρκεια εκπόνησης της εργασίας αυτής μέχρι και την ολοκλήρωσή της, καθώς η βοήθεια και οι συμβουλές του είχαν καθοριστικό ρόλο στην επιτυχία της τελευταίας.

Θα ήθελα, επίσης, να ευχαριστήσω θερμά την κ. Βάνα Καλογεράκη, Καθηγήτρια Οικ. Παν. Αθήνας, για την υποστήριξή της και τις ιδέες της για την συνεχή βελτίωση της εργασίας αυτής.

Τέλος, θα ήθελα να ευχαριστήσω θερμά τους γονείς μου, Παναγιώτη και Αναστασία, και όλη την οικογένειά μου αλλά και τους φίλους μου, για την αδιάκοπη υποστήριξη και κατανόηση που έδειξαν καθ'όλη τη διάρκεια φοίτησής μου στο Εθνικό Μετσόβιο Πολυτεχνείο. Αυτή η στήριξη αποδείχθηκε καθοριστική για την ολοκλήρωση των σπουδών μου και ευελπιστώ να τους δικαίωσα.

Νικόλαος Νικήτας
Αθήνα, 30η Ιουνίου 2021

Contents

List of Figures 11

List of Tables 13

1 Introduction 15
- 1.1 Thesis subject 15
- 1.2 Structure of the thesis 17

2 Theoretical Background 19
- 2.1 Cloud and Serverless Computing 19
- 2.2 Apache Spark 22
  - 2.2.1 Architecture 22
  - 2.2.2 Shuffle Data Management 23
  - 2.2.3 Current Challenges 25
- 2.3 Containers and Container Orchestration 26
- 2.4 Monitoring Tools 28
- 2.5 Automation Tools 29

3 Related Work 31
- 3.1 Serverless Architecture 31
- 3.2 Shuffle Operation Optimization and Apache Spark 33
- 3.3 Elasticity 34

4 Implementation Analysis 37
- 4.1 System Overview 37
- 4.2 Orchestration of Cherry 39
  - 4.2.1 Pushing Shuffle Files 39
  - 4.2.2 Look-Ahead Task-Aware Caching Policy 42
    - 4.2.2.1 Caching shuffle blocks instead of shuffle files 42
    - 4.2.2.2 Implementing the Caching Policy 43
  - 4.2.3 Efficient Memory Usage and Low Cache Miss Rate 47
  - 4.2.4 Fault Tolerance of Cherry 48
- 4.3 Cluster Setup 48
  - 4.3.1 Use of Docker and Kubernetes 48
  - 4.3.2 Automating the K8S Cluster Creation 51
  - 4.3.3 Monitoring 51
- 4.4 Autoscaling Module 52

5 **Experimental Evaluation** 55
5.1 Evaluation Setup 55
5.2 Synthetic Workload Evaluations 55
5.2.1 Stage and Completion Time 56
5.2.2 Fault Tolerance in Spark Workloads 57
5.2.3 Scalability and Resource Efficiency 58
5.2.4 Data Skewness 60
5.3 Real Workload Evaluations 61
5.3.1 TPC-DS Queries 61

6 **Conclusions and Future Work** 63
6.1 Summary and Conclusions 63
6.2 Future Work 64

A **Source Code** 67

B **List of Abbreviations** 69

Bibliography 71

List of Figures

2.1 Different cloud service models vs On-premises. 21
2.2 Spark architecture [25]. 23
2.3 DAG: A visual representation of RDDs and the operations being applied on them for a Spark job. 24
2.4 Kubernetes components within a cluster [50]. 28

4.1 Topology of Cherry. 39
4.2 Cherry orchestration and data movement. 40
4.3 Pushing shuffle files Mechanism. 41
4.4 Reduce stage of a Spark job. At each moment Cherry aggressively caches the shuffle blocks of the upcoming tasks. After the fetch completion, each block gets evicted. 43
4.5 Look-Ahead Task-Aware Caching Policy. 45
4.6 Spark Cluster Monitoring with Kubernetes, Prometheus and Grafana. .. 52

5.1 Experiment results of our benchmark with synthetic workload. Smaller block sizes affect severely the I/O performance of Spark with ESS. On the other hand, Cherry achieves a better overall performance on the completion of the reduce stage. 57
5.2 Additional time needed for a map stage to complete when a Spark Worker crashes, depending on the percentage of the stage that has been already completed. 58
5.3 CPU usage per Cherry Pod while varying the number of them in the cluster. 59
5.4 Normalized Cherry Cache Usage compared to fetched shuffle blocks by executors through time. 60
5.5 Performance of Spark using Cherry on skewed data against Vanilla Spark. 61
5.6 Execution results of TPS-DS queries on Spark and Cherry. 62

List of Tables

5.1 Experiment configurations. Each row shows the total size of the synthetic dataset, the number of map or reduce tasks, and the size of each block. 56

List of Algorithms

1 Caching Shuffle Blocks 47

Listings

4.1 YAML file for a Spark Worker Deployment. 49
4.2 YAML file for a Spark Worker Service. 50

Chapter 1

Introduction

1.1 Thesis subject

Nowadays, cloud computing is widely used from software companies for providing services and applications through the internet. Cloud vendors offer diverse services from the infrastructure layer up to the software layer, exchanging their capital expenses by charging their amenities accordingly. A cloud computing model that has emerged in recent years and is being researched thoroughly is serverless computing. The serverless paradigm is suggested to support elastic and scalable execution of applications and adapt to sudden bursts of load, thus achieving guaranteed service up-time and great performance. Additionally, serverless offerings from cloud vendors are charged only based on time of execution, and as a result, they represent a relatively affordable solution for several types of applications. The serverless architecture is based on the run time environments named functions, and the latter can perform stateless execution of program logic for a short period of time, alleviating the customers from having to manage the underlying infrastructure and over-provision hardware resources on a cluster. However, there are several challenges that need to be addressed and remain an intriguing area for exploration and innovation, such as the management of intermediate state of applications. As demands for compute and storage resources continue to grow, data centers keep on expanding their hardware infrastructure to adapt to these requirements. More specifically, the urge for disaggregated storage and compute engines is becoming a common phenomenon in cloud-based environments, due to the fact that optimization of each hardware infrastructure can take place seamlessly. The serverless functionality that the function invocation model offers, requires a separate storage system that must be designed in a certain way to maintain the program state, and thus, can support the disaggregated architecture of cloud environments.

The fast pace of internet usage around the globe through the years had as a result the rapid increase in available information for everyone to access and process. Additionally, the majority of companies focus on building applications and extract as much customer data as possible in order to process, visualize, and consequently analyze them. Data Analytics is without a doubt a major factor for a business in order to minimize risks from data reports as well as find the preferences of customers and attract even more. Another area that exploits data is Machine Learning, which requires extensive hardware resources and is used in cases such as finance, retail, energy, etc. . Thus, it is evident that applications tend to become more data-centric. Big Data analytics frameworks that re

available either on-premises or as fully managed solutions, such as Apache Spark [71], Hadoop[1], Flink [29], Presto [27] and Google's Cloud Dataflow [30], have emerged to be utilized in data-intensive circumstances, offering a scalable, distributed and performant solution that can seamlessly be adapted to cloud-based environments. More specifically, there is research and work available around data analytics on the cloud, and especially making it feasible using the serverless architecture.

The architecture of the aforementioned frameworks is influenced mainly by the MapReduce [41] paradigm. More specifically, they all provide an engine that is fully-distributed and supports data-parallel computation, while the execution of the workloads is split into a pipeline of separate stages and each stage includes some computation units (i.e., tasks) which are assigned to specific data chunks by a centralized scheduler that has the role of the orchestrator. When intermediate data partitions (i.e., shuffle data) occur between certain stages, they are required to be locally stored by the computation units and later exchanged. These frameworks employ an internal data addressing and transferring solution in order to facilitate the shuffling and sorting operation and handle the intermediate temporary soft state [28, 32, 35, 33, 34]. One of the possible approaches to deal with these shuffle data is to disaggregate it from the compute infrastructure by implementing an external "Shuffle Service" that manages the intermediate state through a separate remote engine. This solution entirely removes state from the execution pipeline, thus transforming a de-facto statefull workload into a completely serverless job. Furthermore, it leverages the possibility of targeted optimizations both on intermediate state management and the framework implementation that is being used.

One of the most prominent distributed processing frameworks is Apache Spark [71]. Spark workloads are split in stages when data need to travel between running Spark executors. These intermediate data of Spark workloads are served by its executors that are already running and are part of the execution, each of which maintaining a part of these data locally. Another solution for ensuring availability of the intermediate *shuffle* data for external Spark executors is the External Shuffle Service [35] that Spark offers, a process that runs alongside the Spark Worker on the same node. Nevertheless, both vanilla Spark and Spark with ESS present several weaknesses. More specifically, one of the main problems is the procedure of fetching small shuffle intermediate data when the shuffle operation over the network is required to take place. This is due to the I/O bottleneck that is set by the limited read/write throughput by HDDs, making it a serious barrier for optimized performance of workload executions. Another major drawback that needs to be stated is that Spark workloads are not executed in a serverless manner because the Worker nodes or the External Shuffle Service require to be always a part of the Spark cluster. If a Worker crashes, all the intermediate shuffle data will be lost, and hence, making the re-computation of a major part of the workload lineage unavoidable. Even a whole node deallocation is prohibitive due to upcoming data loss. This fact also showcases that Spark is not compatible with the concept of cloud cluster architecture where seamless deallocation of nodes takes place. Finally, another noteworthy downside is that the whole Spark architecture, and more specifically ESS, is not adapted to containerized environments that exist in the cloud infrastructure, like Kubernetes [19] or YARN [26], where processes are required to be isolated and stateless.

The main goal of the thesis is to exploit data analytics systems in the cloud and adopt the serverless paradigm by implementing a distributed disaggragated storage engine that will maintain any intermediate data that may occur from the executed workloads. Thus, we implemented Cherry, an open-source distributed task-aware caching shuffle service for serverless analytics, and we utilize a widely embraced Big Data analytics tool, Apache Spark to execute the workloads in a purely serverless paradigm. This feature will transform Spark workloads in a fault tolerant procedure that will ensure no intermediate shuffle data loss in case of node failures, hardware deallocation or runtime crashes. Moreover, we will try to address the aforementioned challenge of the negative impact that small shuffle data have by causing I/O bottlenecks, by implementing a novel caching policy on task-level that will manage to mitigate the latter and increase the performance of Spark workloads. We will additionally utilize Kubernetes as a container orchestration manager in order to simulate a containerized environment of isolated processes that is a common phenomenon in the cloud. In our implementation we will also combine Prometheus and Grafana to monitor how our cluster and workloads operate, and Ansible to automate some processes of software updates and source code build in all nodes of the cluster.

1.2 Structure of the thesis

The presented master thesis consists of 6 chapters.

In Chapter 1, the main subject and goal of the presented thesis, as well as the complete structure of the thesis are presented.

In Chapter 2, we present the theoretical background of cloud as well as serverless computing and architecture, underlying its positive and negative features. Furthermore, the main Big Data analytics framework that was the motivation of this research, Apache Spark, will be mentioned thoroughly. More specifically, we analyze its architecture, the Shuffle data operation that it offers, as well as the current challenges that need to be addressed. Additionally, we mention the theoretical concepts behind containers and container orchestration, as well as monitoring and automation tools.

In Chapter 3, we present previous work by industry and academia that is related to this thesis and is an inspiration for the development of the latter. More precisely, existing research relative to the serverless architecture, the shuffle operation and Spark, as well as elasticity is mentioned.

In Chapter 4, we analyze our implementation, Cherry, a task-aware caching shuffle service for serverless analytics. More specifically, we make an exhaustive system overview, and we emphasize on its core mechanisms, such as the push operation of shuffle files remotely, as well as an implemented caching policy on task-level, and the advantages they offer. We also present how our cluster is set up in detail, and mention an autoscaling module that we built.

In Chapter 5, the experiments conducted alongside with their results are presented. We firstly analyze our experimental setup, and we thoroughly mention the experiments

on synthetic and realistic benchmarks that took place comparing the existing approaches of executing workloads using vanilla Spark and Spark with its External Shuffle Service against our suggested implementation with Cherry. Furthermore, we present the results from our experiments that showcase the benefits it offers due to its core features.

In Chapter 6, a general summary and some conclusions of this master thesis are presented, as well as some future plans and work whose inspiration will be this thesis.

Chapter 2

Theoretical Background

In this chapter we will emphasize on the required theoretical background that we will rely our implementation on. We will start by introducing the concepts of cloud and serverless computing, how they were comprehended through the years, and some of the benefits and drawbacks they offer. Next, we will analyze the core features and architecture around the Big Data analytics engine, Apache Spark. Additionally, we will present the concepts around containerization and containerization tools like Docker, as well as the main container orchestration tool that is used nowadays, Kubernetes. Finally, we will briefly mention the use of automation tools and mention the basic features of the DevOps automation tool, Ansible, as well as the importance of the utilization of monitoring tools in massive cloud clusters, such as Prometheus and Grafana.

2.1 Cloud and Serverless Computing

Cloud computing is the offering of certain computing resources (i.e., software, analytics, artificial intelligence, storage, database, etc. solutions) over the internet, so that they can be accessed remotely. These services are usually charged based on specific pay-as-you-go models according to the required needs of each customer. Basically, the users, instead of acquiring and maintaining whole physical servers, they exchange the use of the capital expenditure (CapEx) provided by cloud vendors for a specific amount of money based on the consumption of it (i.e., operational expenditure (OpEx)). Additionally, the users via the cloud services have the ability to scale elastically in global level their products, and take advantage of the reliability they offer when it comes to disaster recover and data backup plans. Moreover, cloud vendors provide security through a set of specific policies and optimal performance via optimized hardware and network connections. In recent years, software vendors tend to adopt cloud services and provide their products through these cloud solutions due to flexibility, reliability and scalability benefits.

In 2009, *Above the Clouds: A Berkeley View of Cloud Computing* [42] was published by UC Berkeley, on which cloud computing is described as the software that is provided as services through the internet, as well as the hardware infrastructure that offers these services. Additionally, some novel features in cloud computing that are mentioned are that the customers are only charged for allocated machines and resources e.g. per hour and can easily reduce their bills by removing idle machines, or potentially acquire unlimited resources on application demand based on load spikes, and thus avoiding the

constant underutilization or saturation of servers, as well as the possibility of pay-per-use charging except for charging commitment beforehand. Nonetheless, several barriers that have emerged through cloud computing are mentioned and analyzed. Some representative examples are the costly data placement and transport across the cloud infrastructure and the arbitrary performance of rented Virtual Machines on the same hardware by different tenants on disk I/O and network level. Moreover, other bottlenecks that cloud computing is subject to are the data and vendor lock-in for consumers since cloud vendors user their own internal APIs for storage, network and software services and there is no global standard, constant application uptime, service availability and no single point of failure warranties. An additional problem is the difficulty in the detection and elimination of any bugs and errors that may occur in large-scale parallel systems.

The term *cloud computing* arose and became viral after 2006, when Amazon released the Elastic Compute Cloud (EC2) product [11]. EC2 is a web service that is provided by AWS and offers compute resources to users. More specifically, these are virtual machines that can be spinned up or stopped instantly and there is a wide variety of types based on the requirements of the workload that the user requires to execute, such as compute optimized, memory optimized, accelerated computing, storage optimized, etc. In Figure 2.1 the different cloud service models are illustrated, as well as the on-premises model, where everything is located locally in a private data center of the company. The boxes with light blue color are the components that are managed by the customer, while the boxes with light purple color are the once that are managed by the cloud vendor. The cloud offerings are primarily comprised of the following services:

- **Infrastructure as a Service (IaaS)**: IaaS provides access to virtual machines, bare-metal servers and hardware, storage and operating systems on a pay-as-you-go model. Hence, cloud providers allow users to have the ultimate control on software and workloads that they can execute, but they need to undertake the procedures of any maintaining or upgrading tasks required.

- **Platform as a Service (PaaS)**: PaaS offers an on-demand environment for developing and executing software applications. The application developers that use PaaS offerings avoid the management of the underlying infrastructure of hardware, networking and storage.

- **Software as a Service (SaaS)**: SaaS vendors offer licenses per month or per year for certain end-user software that the own to customers. The cloud providers have the responsibility of maintaining all the infrastructure layers from the hardware to the software, and the users simply use the application over the internet.

- **Functions as a Service (FaaS)**: Serverless computing.

Serverless computing is a model of cloud computing which is similar to PaaS, as the cloud vendors are responsible for maintaining the infrastructure of the execution environment. Additionally, they allocate compute resources based on application demand, and no resources are allocated when there is no software execution. As a result, this offering is charged per use and the overprovisioning of resources, and, thus, over-pricing for customers is restricted. The serverless architecture is meant to be scalable and elastic, since it can cover any short bursts based on load that may take place by increasing

Figure 2.1: *Different cloud service models vs On-premises.*

the number of concurrent so called *functions*. Additionally, it is event-driven because predefined triggers must occur to spin up these functions, and is stateless, since there is no permanent state stored in memory and intermediate data are kept remotely in a dedicated storage system. The platforms that are provided by cloud vendors as serverless environments are also known as Function as a Service (FaaS) platforms. Some popular runtimes by the most well established software vendors are the AWS Lambda [4] and Google Cloud Functions [12]. Nevertheless, there are some drawbacks that are yet to be overcome, such as the limits on resources available that are imposed by cloud vendors, the function execution time restrictions, since the can run for a few minutes, the difficulty in debugging and monitoring serverless environments, and the fact that there is no function-to-function communication.

In 2019, *Cloud Programming Simplified:A Berkeley View on Serverless Computing* [47] was published again by UC Berkeley, on which they evaluate their predictions on their previous paper [42] related to cloud computing, showcase the benefits that serverless architecture offers, outline the limitations and constraints that occur through different types of applications using FaaS, and suggest some solutions to overcome the latter. More specifically, the features of the two approaches of function-based computing and traditional serverful computing are compared. Their key dissimilarities are that the billing is not based on hardware resources reserved but on resources used and execution time, that cloud vendors are responsible for any system administration tasks relative to resource provision, and that the storage engine is decoupled from the compute engine, since the whole computation process is stateless. Subsequently, some benefits that make the serverless cloud functions appealing are mentioned, such as the little time required for development that the latter offer since there is no prerequisite in understanding the cloud concepts in depth and new opportunities for research purposes. Some of the challenges that emerge through different types of use cases are the insufficient options for cloud storage solutions that can support execution of applications

that require state sharing in a serverless manner and the absence of storage systems that support coordination and notification mechanisms for cloud functions. Another challenge that is worth mentioning is the mediocre performance when it comes to specific communication patterns such as broadcast and aggregation that are required in Big Data analytics and Machine Learning workloads. If the latter are met, the serveless computing paradigm can be the de facto computing model for the cloud.

In [46], several benefits and shortcomings of serverless functions are outlined. More precisely, serverless architecture is optimal for several use cases such as embarrassingly parallel functions where independent tasks are required to run and no intercommunication takes place, as well as orchestration functions that manage calls to third party services and function composition where the workloads are event-driven. However, certain challenges of the serverless paradigm is the restricted execution time limit that cloud vendors set (i.e., less than 15 minutes) and disk I/O throughput bottlenecks. Additionally, other restrictions are the latency barrier that an intermediate slow storage has since communication between two cloud functions is not feasible and there is no network address assigned to them, as well as the fact that there is no hardware specialization available as an execution environment. The intriguing challenges that still remain are a great area for innovation and exploration for developers and researchers, and if the latter are overcome, the serverless computing paradigm will reach its utmost potential.

2.2 Apache Spark

In recent years, there has been an explosion of data and information that need to be accessed, organized, processed and visualized for a range of purposes. Thus, the Big Data term made its appearance [61], and a wide variety of Big Data analytics tools and systems have emerged since then, in order to process enormous amounts of data in distributed, parallel and scalable systems. Two extensively used data processing frameworks for large scale systems that tend to become the de facto standard for Big Data companies are Apache Spark[71] and Apache Hadoop[1]. In [65] they compare MapReduce and Spark on different analytics workloads showcasing Spark's better performance on almost all benchmarks due to its aggregation component for combine and the RDD caching that takes place in it.

2.2.1 Architecture

Both Spark and Hadoop MapReduce are open source projects. Additionally, Spark shares a partly similar architecture with Hadoop MapReduce, since both adopt the MapReduce programming model. This model offers the capability of executing an analytics workload in parallel based on the number of available machines, and thus achieving faster completion of the latter. Additionally, a Master has the role of organizing the Worker nodes and assign to them the processing of specific parts of data, labeling them as Mappers or Reducers. When Mappers complete their assigned task, they inform the Master where intermediate data are stored. When Reducers start theirs tasks, they are notified from the Master about the location of these data that they will have to process, based on specific keys that are accounted for them. Finally, they complete theirs tasks

and inform the Master. The key difference between Hadoop and Spark is that Spark takes advantage of in-memory processing, while Hadoop only uses disk for reading and writing data, hence Spark achieves greater performance.

The main structural data component of Apache Spark is the resilient distributed dataset (i.e., RDD), which is a set of data that is read-only and fault-tolerant and can be distributed on several nodes of a cluster. Also, it requires a cluster manager, which can be Standalone, Hadoop YARN [26], Apache Mesos [2] or Kubernetes [19]. Spark can also support execution of Machine Learning algorithms through its library MLlib, as well as streaming processes and querying data through its Spark SQL component. Its architecture consists of a Spark Driver where the SparkContext object is created and the main program is executed, named as job. Furthermore, the cluster manager is also named as Spark Master and is one per cluster. The rest of the nodes are named as Spark Workers, can run multiple processes that are named executors, based on the available resources (memory, CPU, etc.) of each node and configurations defined from the user, and execute the tasks that are assigns to them via the Driver. The DAGScheduler, which is located in the Spark Driver, is responsible for creating a Directed Acyclic Graph (i.e., DAG) from the required computation and the assigned job, and splits the latter in stages, and each stage in separate tasks, according to the available job parallelization. Moreover, the DAGScheduler directs the locations to run each task on, and forwards these to the low-level TaskScheduler, as well as manages failures due to shuffle intermediate data being lost, which may cause a re-execution of a whole stage. The available Workers of the Spark cluster initiate one or more executors that will carry the necessary computations and complete the relative tasks. The data set that requires processing is divided in partitions accordingly. An illustration of the main Spark components is shown in Figure 2.2.

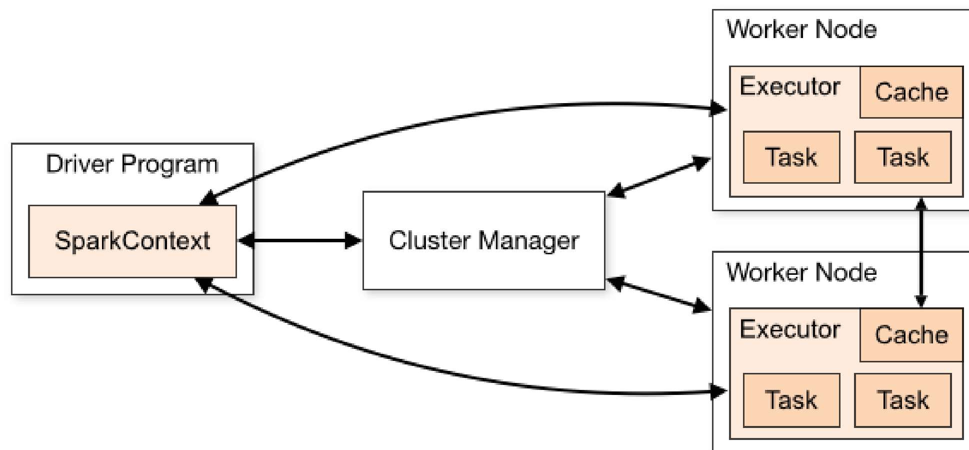

Figure 2.2: *Spark architecture [25].*

2.2.2 Shuffle Data Management

Several map-reduce like frameworks that are implemented for data analytics workloads include a shuffle operation in job executions, where intermediate data are being temporarily stored and later fetched. There are distinct implementations that provide

the aforementioned functionality, such as the External Shuffle Service of Spark[35], a Pluggable Shuffle Service in Hadoop [33] and Apache Flink[29, 32], a Presto Exchange Materialization [28] as well as a Dataflow Shuffle [34]. In our current implementation, we integrated Cherry with Apache Spark, although, as mentioned in section 6.1, we argue that Cherry can be easily integrated with other map-reduce like frameworks, since they follow the same architectural principles which we take into consideration in our modular design.

Two types of data dependencies exist on Spark framework that describe the executing tasks, namely narrow and wide. The partition of the child RDD that is assigned on a task and is dependent on at most one parent RDD, consists a narrow dependency. Common transformations are map, filter and union, there is no data shuffling required over the network of the cluster, and pipelining is feasible. Accordingly, when the partition of the parent RDD of a task is reliant on by many child RDDs, creates a wide dependency. Example transformations are reduceByKey, groupByKey and join. The tasks that require to read intermediate shuffle data are named reduce tasks, while the tasks that create data or read existing data as input are named map tasks. On wide dependencies, the intermediate data (i.e., shuffle data) need to be transferred around different nodes, a procedure that is frequent in the Map-Reduce paradigm. The shuffle data that are generated after each map task, are data files that include sorted data per reduce partition, as well as index files that include offsets and lengths of these data blocks. The shuffle writer that exists in each map task, after processing its respective partition of an rdd that is responsible for, creates the shuffle block data and index files in an explicit format that is the following: shuffle_*shuffleid_mapid_reduceid*. Reduce executors, at first, are required to remotely fetch their respective shuffle data, if the latter are not available in their local filesystem, so as to proceed with a computation that is assigned from the Spark Driver.

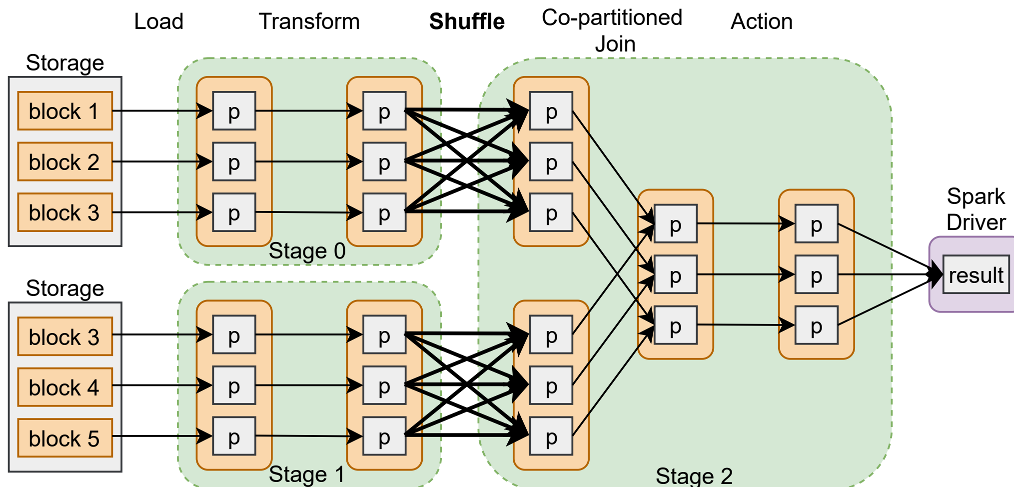

Figure 2.3: *DAG: A visual representation of RDDs and the operations being applied on them for a Spark job.*

In Figure 2.3 a DAG of a simple Spark workload is illustrated, where all stages from the procedure of data loading from the storage engine until the point of sending the final result to the Spark Driver are shown. When a shuffle operation is needed (i.e., wide dependency) the DAG is separated into discrete stages. Other transformations and

actions that constitute a narrow dependency can be pipelined and can be combined into a single stage of the DAG. Spark has two discrete solutions for dealing with intermediate shuffle data that may be created through a certain workload. Firstly, Spark executors that are assigned a reduce task obtain the knowledge from the Driver that includes the nodes' info that the required shuffle data are stored. Thereafter, they can directly retrieve the data from the executors that exist on these nodes, while the latter act as servers for those files. Another solution that Spark offers is the use of the External Shuffle Service (ESS). A Spark cluster can execute a workload by initiating an ESS on each Worker node. It is crucial for the ESS to run continuously and constantly maintain intermediate state. Each Spark executor registers to its local External Shuffle Service. The ESS functions as a proxy through which both node-local and node-remote executors can fetch the needed intermediate shuffle blocks when they are assigned a reduce task at a reduce stage.

2.2.3 Current Challenges

The process of storing and serving shuffle intermediate is a procedure that takes place widely in Spark analytics jobs. Additionally, the latter process TBs of data, and thus, we believe that it is worth enhancing its performance.

One of the main problems that vanilla Spark and Spark with ESS have is the procedure of fetching shuffle blocks from the local hard disk so as to serve them to other Spark executors. More specifically, this process is subject to huge I/O overhead and limited read/write throughput of HDDs because of random file seeks, especially when intermediate data are small in size. Each shuffle file that is created after a map task consists of a certain number of blocks, i.e., partitions. This number is specified by the number of tasks on the upcoming stage(s) of the Spark workload. Additionally, each data block will be requested only once by a certain reduce executor, but each shuffle data file will be requested by all reduce executors. Therefore, the all-to-all communication among the executors and the shuffle service is inevitable. The aforementioned challenge that takes place in shuffle-heavy big data workloads is evident especially when the shuffle intermediate data that are required to be fetched over the network are small in size (i.e., some KBs of data), and thus, the I/O bottleneck occurs and the performance of Spark applications in large scale will worsen. A probable solution would be to use fewer tasks per stage, and hence, force to have bigger sized shuffle blocks. Nevertheless, this case does not portray the real cloud circumstances where the clusters consist of 100s or 1000s of separate nodes. Thus, these conditions inevitably use small shuffle file sizes in jobs.

Another significant downside is relative to fault tolerance. More specifically, Spark attempts to accomplish fault tolerance by persisting shuffle files on disk instead of memory in case a Spark executor fails or an OOM error occurs, and requiring a shuffle service to be running continuously by each Spark Worker in the cluster. However, this feature does not include the high possibility of a crash of a physical node in the cluster, since all the intermediate data that was stored at the latter will be lost. Consequently, a major part of a lineage of the workload has to be inevitably re-executed, and a shuffle re-computation is really expensive. In cloud environments, the deallocation of whole nodes or Virtual Machines within a cluster are a common phenomenon for maintenance or upgrade of hardware. Nevertheless, this scenario is prohibitive for the existing

implementation of the Spark architecture because of probable data loss. In addition, the requirement that the shuffle services of each Spark Worker must have continuous up time, forces, consequently, the unnecessary allocation of compute resources even when there are no executors running in a specific node.

Last but not least, another problem that needs to be highlighted is that the current Spark implementation, and more specifically its External Shuffle Service, is not adapted to containerized environments, like Kubernetes or YARN, where processes are required to be isolated and stateless. To emphasize on that, in YARN, the ESS keeps on running even when a executor gets lost. With Kubernetes, if an executor crashes, all its respective shuffle data are lost and a part of the application will need to be recomputed. For both options, the Spark Workers that run on the same node have access to its other's shuffle service, violating any isolation prerequisites or policies between components that may be a mandatory feature in a cloud cluster. Additionally, there is a wide interest in disaggregated cluster deployments in cloud environments, where the compute engine is located remotely from the storage engine, and workloads can be executed in a serverless manner seamlessly, without keeping any intermediate state. The same requirement is for containerized environments, where containers should be started or terminated without the danger of losing any state or intermediate data. The current implementation of Spark architecture with Kubernetes does not complete this prerequisite, since the ESS of each Spark Worker that is executed in a single container stores any intermediate state and, consequently, is a single point of failure.

2.3 Containers and Container Orchestration

Most cloud computing software programs, until recently, utilized mainly hypervisors at their core. Hypervisors are used for Virtual Machines (VMs). Some widely used hypervisors are Hyper-V by Microsoft, as well as KVM[51] which is a complete virtualization module based on the Linux kernel. Hypervisors are known for their multitenancy and isolation capabilities, as well as sharing of resources.

Over the last few years, containers have gained popularity as a flexible, portable and lightweight tool to deploy software securely and seamlessly on cloud computing infrastructure[58]. The main difference of a container and a Virtual Machine is that the first offers virtualization on OS level, while the latter on hardware level. A container is a software component that includes all the necessary binaries and libraries of an application so as to run independently from the underlying hardware and OS layer. It uses the host kernel and avoids creating a whole separate OS. The user can simply pull a container image from a repository and execute its including software, avoiding the installation of any libraries at the host OS.

Containers depend on a container engine that is able to initiate them. The first implemented module that offered virtualization functionalities on operating-system-level is LXC[52], offering namespace isolation capabilities using a single Linux kernel. Docker[9] is currently the most widely known open source project that extends LXC and comprises a synonym of container technology. A Docker engine is a process that runs as a daemon and distributes available resources on containers, so as the latter function

smoothly. A Docker image is built on top of a base image that is available at a public repository named Docker Hub[10], or at a private repository of a software company, and consists of separate specific commands that create a set of layers. Those commands can be included in a dedicated file named Dockerfile,that can help automate the process of Docker container creation and deployment.

In case of clustering of processes and orchestration of separate containers, there are several solutions available. Firstly, Docker offers Docker Compose as a tool for executing and connecting individual containers that exist on the same host. For multi-host environment and distributed systems, Docker Swarm is an inherent solution for Docker, which uses its own API and defines two types of roles for the nodes of the cluster, managers and workers.

Another solution that is currently the de facto Developer Operations (also known as 'DevOps') tool for orchestration of Docker containers is *Kubernetes*[19], which was created by Google[53] and is broadly adopted in the software industry. Kubernetes is an open source cluster management tool that can flexibly and easily create, deploy and maintain containers, alongside with the required network infrastructure, utilizing the available CPU and memory resources, on any size of node cluster. Additionally, it mostly suits circumstances where containers are required to be executed in Platform-as-a-Service environments[40], like Google Cloud Platform. Kubernetes' internal components are loosely coupled and make use of its extensive API. A Kubernetes Pod is the fundamental containerized unit of an application that includes one or more Docker containers, has its own IP address within the cluster and is accessible from any other Pod running in the cluster. Other distinct Kubernetes components are Services, Namespaces, Volumes, Labels, Deployments, Pod Autoscaler etc.

The nodes that are part of a Kubernetes cluster can be separated into a master or many masters if high availability is required, as well as workers. The Kubernetes master is the control plane of the cluster, and specific major processes are running on this node. More specifically:

- **API server**: It serves the Kubernetes API on external and internal level, utilizing the JSON format and managing requests.

- **Scheduler**: It is responsible for scheduling the location of the execution of a Pod based on resource requirements of the Pod and resource availability of the nodes, as well as predefined restrictions by the user.

- **Controller manager**: It communicates with the API server and its main role is to make the required changes on Kubernetes resources to get the cluster state to a desired state.

- **etcd**: It is a persistent and highly-available key-value data storage system that keeps all the information relative to the state of the running applications, as well as the configuration of the whole cluster.

Worker nodes are also required to run specific processes to be considered part of the Kubernetes cluster, and those are:

- **Kubelet**: It is accountable for the active state of the node and does health checks on running Pods via heartbeats, ensuring that the desired states of the latter are achieved.
- **Kube-proxy**: It operates as a proxy server, directing the network traffic between different containers within the Kubernetes cluster.

A diagram of a Kubernetes cluster and its main components is illustrated in Figure 2.4.

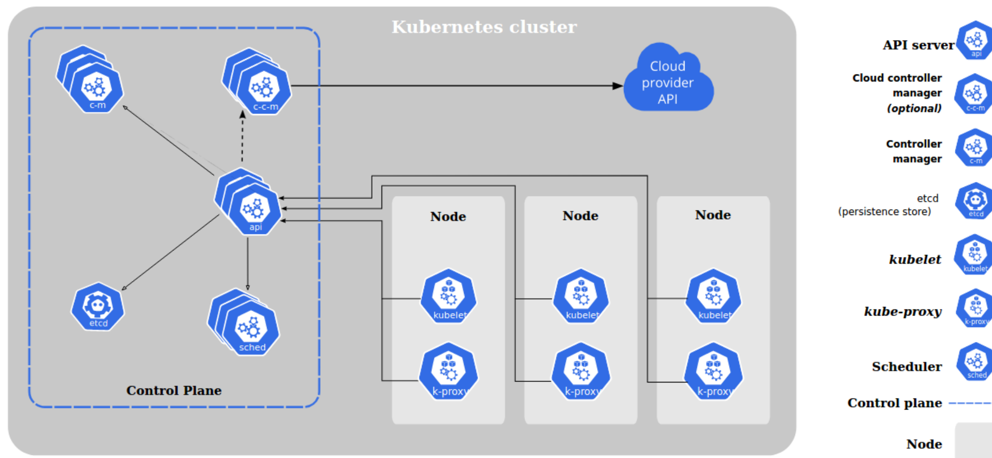

Figure 2.4: *Kubernetes components within a cluster [50].*

2.4 Monitoring Tools

Monitoring for businesses is a procedure of fundamental importance. More specifically, monitoring tools can lower the overall operating costs since infrastructure that requires replacement or upgrade will be easily indicated. The usage of monitoring software that includes reporting and alerting mechanisms will help minimizing the time for any troubleshooting network related errors for IT services that may occur and ensuring more stable up time for the monitored services and software. Moreover, analysis of historical metrics scraped from monitoring systems can aid in showcasing any future trends and make decisions more strategically in order to improve customer reliability towards a provided product.

Cloud computing usually means clusters of machines with different resources and role cooperating in order to support a complex application or complete a specific workload. This is a big challenge and requires optimal resource utilization, performance efficiency and network reliability. Hence, cluster monitoring services are crucial to ensure great interoperability, communication and stable functionality of the whole cluster. Additionally, scraped metrics of the cluster that are visualised clearly in self-explanatory and dynamic dashboards can illustrate a real-time status of the cluster more thoroughly, as well as past circumstances where any problem relative to performance or network degradation existed and requires analysis. Especially when it comes to Kubernetes clusters, it is fundamental to set up a functional monitoring and logging system in order to obtain insights in the intenals of Kubernetes, such as metrics relative to the kube-proxy, the kubelet, the kube-controller-manager, the etcd, any running Pods, the resources

available (e.g. memory, CPU) of the nodes of the cluster, the networking status of the cluster, as well as information acquired from executed workloads and applications.

A widely used monitoring tool by cloud and software vendors is Prometheus [22]. Prometheus is an open-source software package, with its main component being the Prometheus Server that is responsible for scraping data in the form of time series in node, process, microservice or container level and storing them locally. The Prometheus Server monitors and scrapes the required data from the Prometheus Targets that are required to be exposed to it. The targets that are meant to be monitored can be listed in a YAML file. Prometheus, additionally, offers its own query language which is Prometheus Query Language, and users can utilize PromQL to get and process any data metrics they need from Prometheus. Moreover, Prometheus provides its own, rather simple, dashboard web interface, as well as alerting mechanisms based on certain conditions that can be set. One of the most known and embraced visualization solutions that is used for creating comprehensible graphs and dashboards of any metrics acquired is Grafana [15]. Grafana can seamlessly operate with time-series based databases such as Prometheus and use the latter as a data source to query metrics, and provides powerful dynamic dashboards in a web interface as well as alerting solutions.

2.5 Automation Tools

Automation solutions for software development and deployment operations are a significant part of the DevOps ecosystem. They can enhance the productivity of employees since some processes can be automatically completed and, thus, reducing the manual effort. There is a wide variety of open-source and closed-source automation tools available that can aid in automating procedures and repetitive tasks for management and configuration of applications on clusters with numerous nodes and complex environments. One of the tools that has emerged as an efficient and flexible software is Ansible[38]. The main control machine connects to the rest of the nodes of the cluster over SSH and runs small tasks called modules. Ansible utilizes YAML files and adopts an agentless architecture since the target servers are not required to install Ansible, simplifying its use. In addition, developers can combine several modules in a specific order to create an Ansible Playbook in order to do their tasks even faster and prevent monotonous manual work.

Chapter 3

Related Work

The goal of this chapter is to present and compare relative research and work that constitute an inspiration and motivation for our suggested implementation. More specifically, there will be an emphasis on provided services that are relative to serverless computing and serverless architecture on the cloud as well as research on this area. Suggested work and modifications relative to external shuffle service of Spark, elasticity and elastic distributed systems will also be highlighted.

3.1 Serverless Architecture

Serverless computing solutions are already available by major cloud vendors. The first solution that emerged was by Amazon Web Services, named AWS Lambda, which allowed customers to execute software as serverless functions without the need to pre-allocate or manage any server or hardware infrastructure. Serverless runtime solutions with similar functionality that exist nowadays are Google Cloud Functions, IBM Cloud Functions [17], Microsoft Azure Functions [6] and Oracle Cloud Functions [21]. Each function of these services can acquire 128 to 3008 MB of memory, has maximum duration time of 2 to 15 minutes and deployment size of 6 to 500 MB. Thus, there are memory and time constraints that are yet to be overcome. Additionally, in order to maintain state through the execution of the application, external storage services like object or key-value storage are required. Both cloud service solutions using either the disk to maintain data or memory have their drawbacks, since the first is generally slow, while the latter is more expensive. In order to overcome several restrictions that are set from cloud functions, services providing serverless containers were introduced, such as AWS Fargate [3], Google Cloud Run [13], which is a rather hybrid solution between containerization and the serverless approach. Additionally, serverless orchestration offerings, such as AWS Step Functions [5], are provided by cloud computing infrastructure providers that emphasize on orchestrating different services into serverless workflows where the output of one step acts as an input to the next one.

When it comes to research, there is public work relative to serveless platforms and architecture. From a study done in [44] it is depicted that currently serverless computing with AWS Lambda does not provide any guarantee on the available resources and performance stability that correspond to charges on tenants. More specifically, they proved via benchmarks that there is no performance isolation for serverless execution of apps, and the system can be exploited. In [36] some advantages are highlighted through

examples that utilize AWS Lambda functions and serverless computing. More specifically, the first case study showcases that serverless platforms can notably decrease the hosting and operational costs of an application. Through the other case study, it is depicted that lambda offers the possibility to break traditional monolithic applications into microservice oriented granular modules and thus achieving seamless maintenance of the code as well as quicker release of updates and changes for the application. Finally, it is mentioned that circumstances that can embrace serverless architecture at ease are those where tasks of high throughput and short period of execution take place.

Additionally, several approaches through literature have been suggested for exploiting serverless computing. *Cloudburst* [66] is a stateful FaaS platform similar to AWS Lambda Functions. Moreover, it is made sure that consistency guarantees exist among function execution runtimes across the cluster, and, since it is built on top of *Anna* Key-Value Storage [69, 70], it can efficiently leverage its integrated caching capabilities and use mutable caches in order to process key-value storage objects with minimum latency. *Cloudburst* runtime can autoscale independently from *Anna* through heuristic policies, and thus, achieving disaggregation. *FaaSFS* [62] is a file system for stateless cloud functions that utilizes a familiar POSIX API, reaching performance close to what a local file system achieves. It is also mentioned that it is implemented from scratch, exploits inter-process communication mechanisms, its backend service stores any state in memory, and achieves optimistic concurrency control functionality. Another previous work relative to serverless paradigm is *NumPyWren* [63], which is a system for linear algebra that can be utilized to execute several distributed linear algebra algorithms in a serverless manner. For the serverless execution of these algorithms, a specific language was implemented named LAmbdaPACK. There is also a module that has the role of provisioning executors and managing compute resources in order to make these tasks run as cloud functions. Lastly, it is stated that runtime ephemeral data are maintained in fast storage like Redis, and permanent data in slow storage like Amazon S3. Another recent work is *Shredder* [73], which examines multi-tenant isolation on serverless environments and pushes functions into storage. However, it is executed in a single node and does not address any fault-tolerance feature.

Locus [59], which is a serverless analytics system, was created in order to execute map-reduce like workloads as a serverless environment. It integrates both slow and fast storage from cloud vendors to store shuffle intermediate data (i.e., AWS S3 and AWS ElastiCache respectively), named as hubrid shuffle. Hence, this functionality, through benchmarks, seems to accomplish both cost and performance efficiency, since, disk storage is cheap but slow, while memory storage is faster but more expensive. Crucial [39] is a system to program and execute concurrent stateful applications with FaaS serverless architecture. It uses AWS Lambda and is built using an efficient disaggregated shared memory-level data store. Additionally, it is compared with Apache Spark on Machine Learning benchmarks and achieves similar or better results. Another work, named *Lambada* [55], is an implementation of a data processing framework with serverless architecture. Based on some use cases presented, they prove that the serverless paradigm fits perfectly for interactive query workloads on rarely read data. All components of the framework are provided by the cloud vendors. More specifically, the workers of the system run in AWS Lambda as cloud functions and utilize for intermediate shared storage the services Amazon S3, Amazon SQS (Simple Queuing System) or Amazon Dy-

namoDB, depending on the size and type of data. They also use the shared storage to intercommunicate, as well as communicate with the driver of the running application that runs locally. Finally, a concept proposed in literature as a new cloud solution ισ *serverless clusters* [54], which integrates characteristics from Virtual Machines with serverless cloud functions, in order to solve the challenges of each other.

3.2 Shuffle Operation Optimization and Apache Spark

Data analytics systems such as MapReduce and Spark have been studied both by industry and academia, and there has been great emphasis on the shuffle operation that takes place frequently throughout a job. *Magnet* [64] modified the Spark External Shuffle Service and tried to solve the efficiency issues created by disk I/O bottlenecks on read/write operations of intermediate data as well as the all-to-all connection overhead that is required during shuffle data fetching from map to reduce tasks. Hence, they analyzed how the currently available ESS works, and suggested pushing shuffle files from Spark executors to the shuffle services and merging them per shuffle partition in a best-effort approach in order to achieve sequential disk reads of bigger sized shuffle chunks. They, additionally, have both merged and unmerged shuffle files accessible for fetching from reduce tasks, and claim that *Magnet* can be deployed in both cloud and on premises infrastructure. *Riffle* [72] is an optimized shuffle service on top of Apache Spark, that focuses on achieving larger sequential I/O requests when shuffle block fetching takes place. There is a shuffle service running on each physical node of the cluster and tries efficient block merging by pulling the shuffle intermediate data files from map tasks and executing the merging of them. This functionality implemented as a best-effort approach, so as to have both merged and unmerged shuffle files ready for processing by reduce tasks. Additionally, a centralized merge scheduler is responsible for maintaining information about the progress of task executions and the scheduling of merge procedures on distinct Riffle shuffle services. For both implementations, Magnet and Riffle, there is no fault tolerance provided in case of node failures, and compute and storage engine are co-located.

IBM researchers implemented Apache *Crail* [67], an architecture which utilizes a multi-tiered data store that achieves high performance using specific RDMA interconnected storage resources and RDMA-based network. *Crail* can be used to execute data analytics workloads, and their aim is to execute workloads with Spark or Hadoop and maintain all intermediate data in these. Moreover, it is suggested that storage and compute engine of the cluster can function being both co-located or disaggregated due to the fast storage system that can mitigate the I/O latency. Thus, they created an HDFS adaptor to allow seamless cooperation between *Crail* and other data analytics frameworks, as well as a Spark component that maintains functionality for broadcasting and shuffling procedures. Nevertheless, their hardware is specialized and there is no addressing in the challenges of small-sized intermediate data that occur in shuffle operations. Another research work by Intel [18] has been published that focuses on having the compute engine disaggregated from the storage as well as addressing the current Spark shuffle service issue of fault tolerance. More specifically, they created a remote shuffle manager that has the ability to maintain all intermediate shuffle and cached data from Spark workloads to a remote file system that is compatible with Hadoop API,

without the need of any metadata service. However, there is no mention for any optimization on the process of reading of shuffle files from reduce tasks.

Another implementation on top of Apache Spark that focuses on aggregating shuffle intermediate blocks is *Cosco* [8]. *Cosco* utilizes a remote distributed file system for storing these files, integrates write-ahead buffers for the shuffle services, and uses a metadata service to keep track of the correlations between shuffle files and map tasks, as well as a scheduler that is responsible for orchestrating the shuffle partition merging on the shuffle services. *Sailfish* [60] aims at merging intermediate shuffle blocks of Hadoop MapReduce data analytics workloads and, thus, decreasing the number of block fetches by reducers that is required to take place. They, also, extended an external distributed filesystem similar to HDFS with the use of the i-files, which is a component that they introduced and helps at partitioning, sorting and indexing shuffle chunks. Both *Cosco* and *Sailfish* works depend on existing external storage systems to operate as shuffle services.

Additional work named *iShuffle* [45] emphasizes on the optimization of shuffle phase operation on Hadoop MapReduce workloads. More precisely, it offers the mechanism of predicting the map partition sizes and fairly pushing mapper shuffle data to the reducers, in order to mitigate data skewness and stragglers, and can efficiently operate in a multi-tenant environment. Another relative work presents *Flint* [48], which is a rewrite of the execution engine of Spark, is focused on Spark and explicitly only for pySpark interface, and aims at adding to data analytics workloads the serverless functionality. *Flint* exploits AWS Lambda by running Spark executors as cloud functions, named *Flint* executors. Moreover, tasks use the service Amazon S3 for getting input data and storing output final data, as well as Amazon SQS for storing intermediate output. To overcome the limitation of restricted execution time that Lambda functions have that is set by the cloud vendor that provides the service, *Flint* executors send their current state back to the scheduler if tasks get close to the execution duration limits.

3.3 Elasticity

There is a lot of ongoing effort in elastic data analytics systems and autoscaling infrastructure. In [57], a system was implemented that achieves resource elasticity in distributed Deep Learning workloads. This engine is built on top of the open-source Machine Learning platform Tensorflow and takes into consideration both cost and efficient resource usage in decision making. It also integrates a module that appends or removes workers in specific moments in execution time based on scaling conditions that are always required to be met. In order to keep all application state when addition or removal of workers takes place, the latter store it in memory. Furthermore, the system manages, through real time worker metrics, to detect any worker that functions as a straggler and slows down the whole workload, remove it, and replace it with a new worker. The key-value storage named *Anna* that was previously noted (section 3.1), is an autoscaling system that combines both memory and disk tier for storing data. In further detail, it monitors and analyzes metrics of the cluster, and can elastically change the number of nodes in each tier based on variations of the workload. Moreover, it exploits Kubernetes as an orchestration manager of containers, it has the ability to replicate any

hot keys of the dataset and achieve a coordination-free consistency level. Another work based on the elasticity concept is *Medea* [43]. It emphasizes on long running applications in multi-tenant clusters, and can schedule two types of containers, long-running as well as short-running ones. It is also worth mentioning that *Medea* gives the ability to users to select certain placement constraints and is built on top of Apache Hadoop YARN.

ElasticMem [68] is focused on executing analytical applications, such as relational algebra queries, using the containerization paradigm. There is also implemented a resource manager that modifies the memory resources available to each running container. Machine Learning techniques are also applied to ingeniously predict the memory requirements of the executed queries and, thus, adapt to to this demand and alter the JVM of the application dynamically. *Pocket* [49] is an elastic distributed data store that can be used for serverless data analytics. Its architecture is built on top of Apache *Crail* that was previously mentioned (section 3.2), but utilizes an RPC library instead of the RDMA network protocol. Furthermore, the AWS Lambda service is used as a serverless compute engine, and lambda functions can simply access the ephemeral storage of Pocket nodes in order to store intermediate files from the execution runtime, while the Amazon S3 service is used as a long term storage of input and output files. Pocket's architecture includes a controller which is responsible for constantly monitoring the status of the storage nodes as well as the size of the load (i.e., capacity, existing allocation and throughput), and scales the available resources accordingly. Additionally, some metadata servers exist to make sure that policies for data placement are not violated. For the addition of new storage servers, the latter obtain an IP address from the controller and submit their available resources to the metadata servers in order to connect to the cluster. Accordingly, when a storage node needs to be removed from the cluster, it gets blacklisted from the controller and intermediate data are garbage collected when the respective job executions finish.

As far as Apache Spark is concerned, there are various approaches. First and foremost is the autoscaling mechanism that Spark already provides. More precisely, Spark offers Dynamic resource allocation for changing the size of allocated resources available in order to avoid starvation or underuse of the latter. There are explicit configuration settings that can be modified by the user in order for this mechanism to perform optimally. For scaling up, if many tasks are pending and a certain amount of time has passed, new executors kick off. For scaling down, if many tasks are identified as idle and specified time has passed, they are removed. In order to prevent cached data that may be lost, another parameter can be configured for the duration to wait before destroying those executors. However, the External Shuffle Service of Spark is required to be present in order to maintain the intermediate data by these executors, since they are stored outside of the latter. This technique, however, still has the following drawbacks. Firstly, even if idle executors can be removed from the dynamic resource allocation mechanism of Spark, the latter is not consistent with the concept of the cloud cluster infrastructure where the whole nodes get seamlessly deallocated. This is due to the fact that even if no executors are running in a node, the ESS that runs there still has to operate as a shuffle data service for external Spark executors. Another downside that needs to be highlighted is that the exponential scaling out of executors as well as the delayed scaling of them after a certain period of time causes overprovisioning problems,

with possible increases in costs.

In literature, there is previous research and work on Spark and elasticity. For instance, *iSpark* claims to dynamically modify the number of allocated Spark executors at runtime, based on real-time metrics that are being monitored. The problem of resource provisioning is transformed to a bin packing problem and is solved using several algorithms available. In order to avoid the loss of intermediate data when a running executor needs to be removed, *iSpark* manages to preserve them in memory with the use of the DAG information that has been generated. Another relative work is *Elasecutor*, that aims on allocating dynamically resources for Spark executors and uses time series predictions to limit underutilization of resources. This operation is utilized after a number of runs of the same workload have taken place in order to get the required metric and be able to do the optimal predictions. *Elasecutor*, additionally, modifies the memory thresholds of the executors elastically at runtime. intermediate data Finally, *Prophet* is another work that focuses on scheduling executors dynamically according to requirements that arise as workloads are being executed. It is built on top of YARN and Spark, ackwnoledges demands of resources on executor level in order to take an allocating decision, and uses Machine Learning techniques to create a profile of settings for new workloads. Moreover, its authors claim that one can easily predict the resource requirements accorind to the DAG structure. However, for both *Elasecutor* and *Prophet*, the process of how shuffle intermediate data are managed is not mentioned.

Chapter 4

Implementation Analysis

In this chapter our suggested implementation is presented. Firstly, our thought process will be described and a detailed overview of the topology of our system will be made. Subsequently, the features it offers will be highlighted and fully analyzed along with the orchestration of our implementation.

4.1 System Overview

We decided to work on Apache Spark as it is widely used for Big Data analytics workloads and achieves impressive performance in distributed computing. Therefore, our implementation is built on top of Spark with the aim of enhancing its capabilities and improving its job execution time. We use Spark version *3.0.1*, and can integrate our shuffle service seamlessly with Spark, without requiring any modifications to existing Spark workloads (i.e., existing client code). Although our proposed shuffle service is Spark-specific, we believe that its architecture and attributes can be effortlessly ported on similar big data systems, as we mention in chapter 6.

One of the main challenges that users and developers of Spark face and we focused on overcoming, is the fact that Spark workloads are not executed in a disaggregated manner. To put it differently, intermediate data are stored in local disk of Spark workers and not remotely, and as a result, in case of node failures massive amount of shuffle data will be lost, causing a significant part of the lineage graph to inevitably be executed, and delaying its completion time crucially. Another obstacle that arises in Spark and was mentioned in chapter 3 by relative work takes place in shuffle read phase of reduce tasks due to the limited read/write throughput that HDDs set. This constraint is of great importance when Spark executors are reading shuffle blocks of small sizes and IOPS are restricted, causing the degradation of execution time of workloads significantly. Finally, the architecture of Spark and implementation with Kubernetes or YARN is not fault tolerant in case of crashes, and there is no isolation of processes guaranteed since intermediate data are shared between Workers that run on the same node.

Due to the aforementioned reasons, we implemented Cherry, which is a distributed task-aware Caching sHuffle sErvice for seRveRless analYtics. Cherry exploits Kubernetes, and each module of our Spark system is deployed in a discrete K8s Pod to achieve isolation between processes. The Spark Driver is executed in client mode in its own K8s Pod and not in cluster mode, so as to control its execution location and avoid having

it running within a Worker Pod. We decided to deploy the Cherry Pods in separate - **disaggregated** - nodes in order to store all intermediate shuffle blocks and state of every Spark workload. Additionally, there are specific nodes that are labeled as Worker nodes, where Spark Worker Pods with their respective executors are running, based on the available resources and the user configurations. On a separate node, labeled as Master node, the deployment of the Spark Master Pod and the Spark Driver Pod take place. Another process that is executed on the specific node is a Metadata Service, whose role is to keep track of the alive Cherry Pods that exist in the cluster. More precisely, Cherry Pods are registered on the Metadata Service when they are initiated, and the Metadata Service stores their IP and Ports in an Array. Additionally, the Metadata Service must respond to the active Spark executors, since when they are initiated from their respective Spark Workers, they firstly request the list of the aforementioned Pods in order to know the available locations to push and store their produced intermediate shuffle blocks.

The requests in Spark for intercommunication between components and processes are implemented through the Remote Procedure Call, which is a request–response communication protocol. The requests are split in two types, those that are one-way messages without expecting any return value, and those that require a return value. The main component of RPC is the RpcEndpoint that is an interface for receiving and processing requests by clients. Henceforth, additional precise PRC endpoints have been implemented and strictly specified ports have been set open for accepting connections between Pods. This was made in order to allow RPC calls from executors to push shuffle blocks accordingly, as well as to create communication channels between the Cherry shuffle services and the Spark Driver. Additionally, specific RPC messages are created that will be exchanged among the Cherry Pods and the Metadata Service, but also among the latter and the Spark Workers. In more detail, the implemented RPC messages for the interaction with the Metadata Service are the following:

- **RegisterNewCherryNode**: This message is used by each Cherry shuffle service in order to send their IP and Port they use to the Metadata Service so as to store them.

- **GetCherryNodesInfo**: This RPC message is sent by the recently initiated Spark executors to the Metadata Service, due to the fact that need to learn the available Cherry shuffle services in the cluster to push their shuffle data between stages of a Spark workload. The Metadata Service also stores information of alive Spark executors that send the specific RPC message. Hence, when a new registration of an added Cherry Pod takes place, it will be able to inform them.

- **DeregisterCherryNode**: When a Cherry Pod is being stopped, it informs the Metadata Service to erase its information from the list of available shuffle services in the cluster. This must take place, however, only when no workloads are being executed to ensure no intermediate data loss.

- **DeregisterExec**: This message is sent after Spark executors are ordered to shut down in case of a failure or completion of the Spark job, and hence, the Metadata Service will remove their information from the list with the alive Spark executors in the cluster.

The RPC messages that are implemented for the process of pushing shuffle blocks from the Spark executors to the Cherry shuffle services based on the execution pipeline, as well as the receiving and processing the information of the launching of upcoming tasks from the Spark Driver to the Cherry nodes and utilizing its caching policy on task-level, will be analyzed in subsections 4.2.1 and 4.2.2 respectively. The Cherry's main topology is illustrated below, in Figure 4.1.

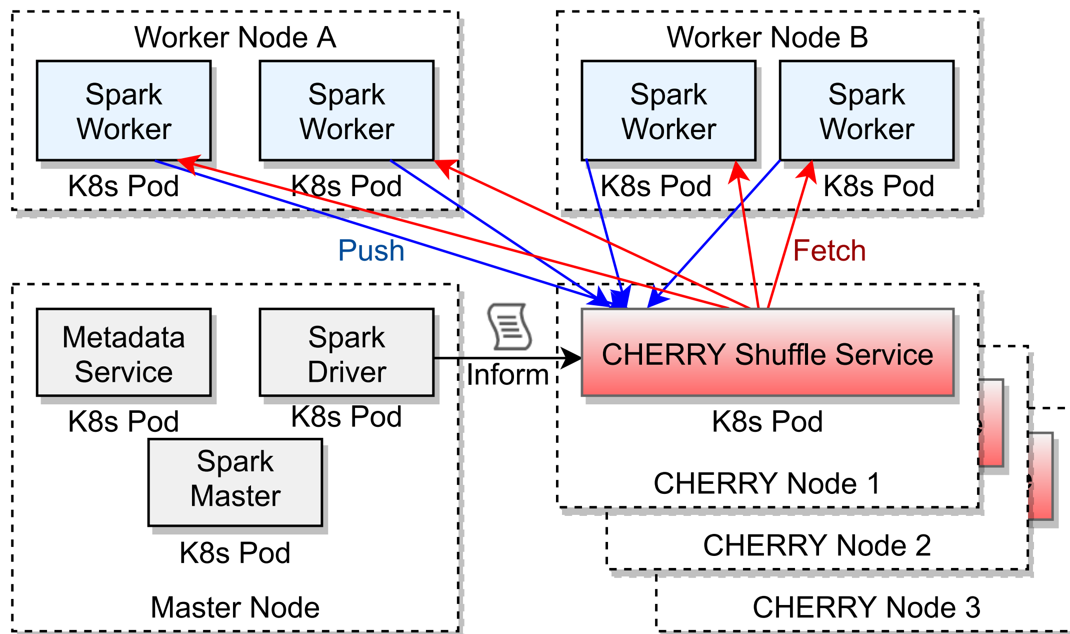

Figure 4.1: *Topology of Cherry.*

4.2 Orchestration of Cherry

In this section we will emphasize on the the data movement that takes place in a Spark cluster with the Cherry shuffle services, as well as present thoroughly how the cluster is orchestrated. This information is illustrated in Figure 4.2. With light red we represent the additions that we implemented so as to smoothly integrate Cherry into Spark. We will analyze the data movement in the following subsections.

4.2.1 Pushing Shuffle Files

We followed a disaggregated approach in order to implement the distributed Cherry shuffle services, with the aim to store all intermediate shuffle blocks from map tasks there. This mechanism makes Spark workloads more fault tolerant, since, on node failures where Spark Workers are executed no shuffle data will be lost. In addition, the feature of maintaining a shuffle storage remotely allows the Spark Worker Pods to run seamlessly in a stateless manner. The flexibility that Kubernetes offers in deploying, and removing those Pods with preconfigured allocated resources, as well as increasing and decreasing the number of those instances that are running at any point in time, transforms Spark into an actual serverless analytics engine. The Spark Workers push their shuffle data in a round-robin pattern to all the available Cherry Pods in the cluster, and

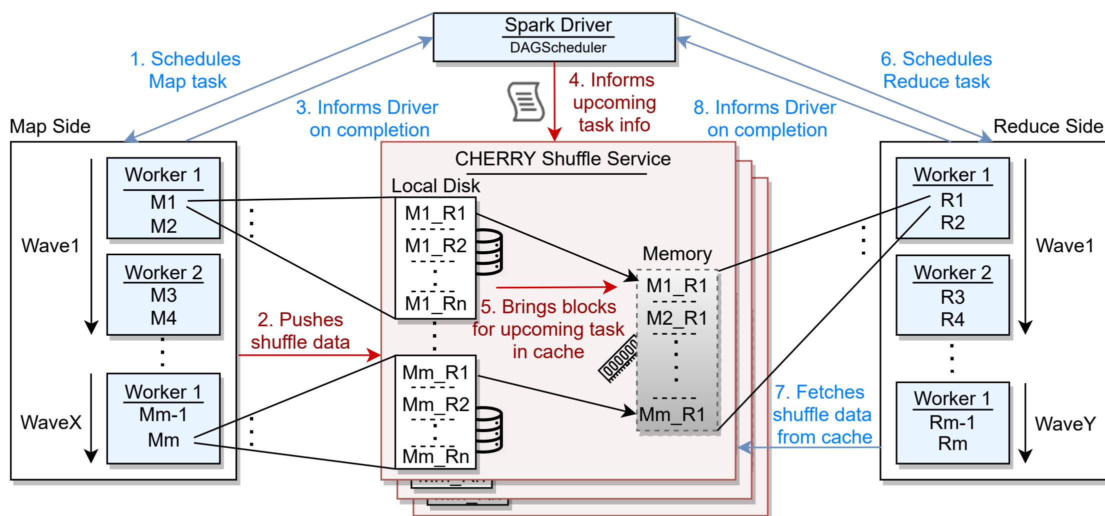

Figure 4.2: *Cherry orchestration and data movement.*

therefore the latter end up having an approximately equal amount of intermediate data.

When an executor is initialized from a Spark Worker, it registers with the available Cherry shuffle services in the cluster, in a similar manner as it occurs with the Spark ESS. The information sent from executors is their IP and Port, as well as an instance of the ExecutorShuffleInfo class, which includes the local directories and subdirectories that the executor uses to store its shuffle files in. The network details of the Cherry services are received from the Metadata Service as mentioned in section 4.1. At the beginning of the map stage, the DAGScheduler that exists in the Spark Driver splits the latter in map tasks and the corresponding data in partitions, and then schedules the map tasks in specific executors (Step 1, Figure 4.2), based on the availability of resources of Spark Worker Pods. When a map task completes its assigned computation, the executor creates a shuffle data file and a shuffle index file that stores in the local disk, and then pushes them to the Cherry Pod that has been selected by the round-robin pattern through explicit RPC messages (Step 2, Figure 4.2). When the procedure is finished, the mapper executor informs the Spark Driver about the completion of the task (Step 3, Figure 4.2). The Cherry Pods store the shuffle data locally by cloning the file path names of the executors, which are known from when the registration of the latter took place. With this mechanism the retrieval of the requested blocks to be fetched by reducer executors is straightforward. We observed through measurements that the time needed to successfully push intermediate shuffle data from map tasks to Cherry is trivial and does not add any additional overhead.

In order to illustrate thoroughly this approach, a sequence diagram in class level is depicted below, in Figure 4.3. More specifically, in the latter a specific part of a map task is shown, and we represent the RPC messages that are exchanged between a Spark executor and the Cherry shuffle service in red color (thick lines for data exchange and thin lines for control exchange), whereas the sequence inside a component (i.e., the executor and a Cherry Pod) in black color. The selected classes shown for the Spark executor are the *LocalDiskShuffleMapOutputWriter* that has the functionality of persisting shuffle data files alongside their index files, the *IndexShuffleBlockResolver* that generates and maintains the intermediate shuffle blocks' mapping between the physical file location

and the logic block, the *NettyBlockTransferService* that extends the BlockTransferService class and uses the Netty protocol to fetch a set of blocks at a time, as well as the *TransportClient* class that implements a client for making a request to the server, i.e., to fetch consecutive pre-negotiated chunks in a stream manner. For Cherry, the shown classes are the *TransportRequestHandler* that is a handler attached to a Netty channel to process requests from clients, as well as the *RemoteBlockHandler* that is modified and based on the Spark *ExternalBlockHandler* class, a RPC handler which can handle registering executors, opening shuffle blocks and serving them to clients.

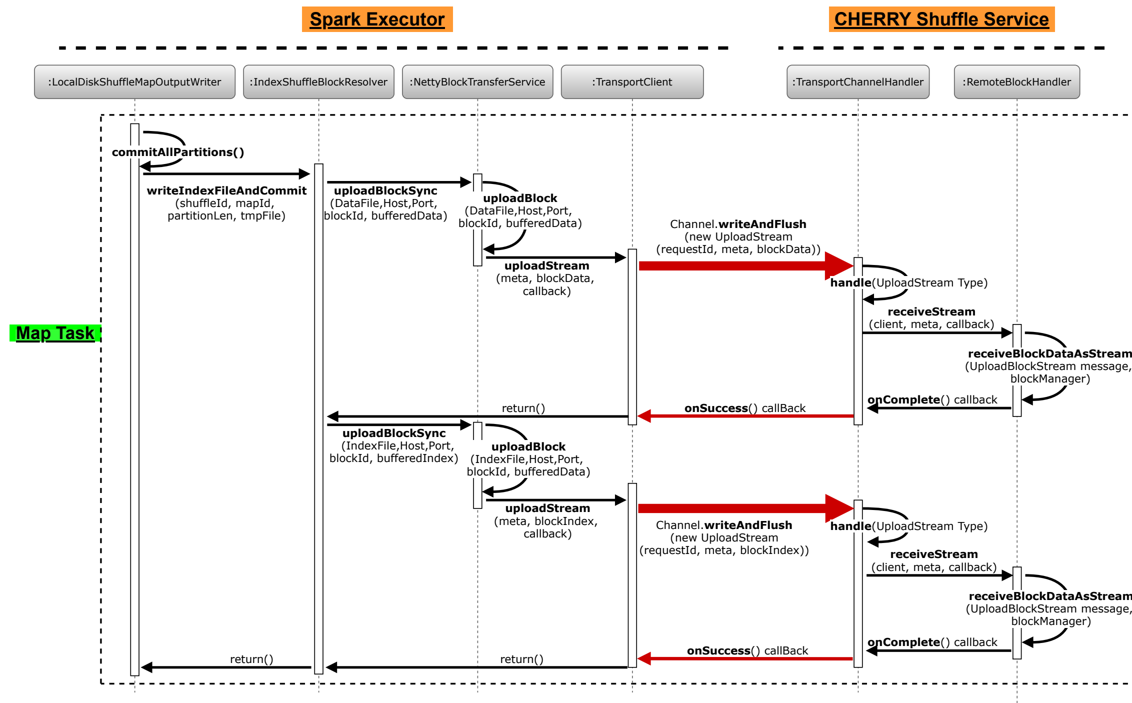

Figure 4.3: *Pushing shuffle files Mechanism.*

After the required computation from the executor has been completed and before the latter informs the Spark Driver that the task is completed, the *commitAllPartitions()* method is called from the LocalDiskShuffleMapOutputWriter class. This method calls the *writeIndexFileAndCommit()* method that is implemented in the IndexShuffleBlockResolver class, alongside with the shuffle Id, the map Id, the partition length, and the name of the temporary file. This is where the procedure in default Spark stops. When our system reaches this point, the NettyBlockTransferService's instance is used and its method *uploadBlockSync()*, and consequently, the modified method *uploadBlock()* is called that initiates the procedure to upload the respective shuffle data and index files through the TransportClient.*uploadStream()* method. Firstly, we initiate a connection to upload as a stream the shuffle data file, and we combine the connection by passing its path file that exists in the local disk of the executor as a metadata message, The Cherry shuffle service accepts the connection through its TransportHandler.*handle()* method, with the explicit type UploadStream. For this process, network protocol (i.e. a class) is implemented that is named *UploadBlockStream* and extends the class BlockTransferMesage class. There are actually many BlockTransferMessage instances that are used for different occasions by the BlockTransferService class that uses the Netty framework to fetch a set of blocks at a time. Thus, the header of the messages from the executors

are sent to the Cherry services as an UploadBlockStream type message, and the latter know immediately that a request to upload a block arrived, and expects to receive the block as a stream.

Our RemoteBlockHandler instance receives the shuffle data file as a stream (through the *receiveBlockDataAsStream()* method) and writes them down to the corresponding path from the metadata message until they are completely received. Then, it can inform the TransportChannelHandler, and the latter can activate the *onSuccess()* callback function to notify the executor. Secondly, we upload the shuffle index file accordingly, following a similar manner. The MapStatus class of Spark is important at this point where the push of shuffle files is complete and when the tracing of shuffle blocks for caching is required before fetching. To emphasize on that, the MapStatus class of Spark has a crucial role for matching the created shuffle files of map tasks with their location, in order for the reduce tasks to consume this information. More specifically, it is the result that is returned by a map task to the DAGScheduler after task completion, and involves the address of the block manager that the intermediate files are maintained alongside the sizes of outputs per reducer. Thus, when both the data and index files have been received, the executor updates its MapStatus instance and informs the Spark Driver about the completion of its assigned task. To put it differently, map executors write to the MapStatus data stracture the location of the intermediate shuffle files they created, and reduce executors read from them.

4.2.2 Look-Ahead Task-Aware Caching Policy

The challenge that Spark has by default on the process of fetching small shuffle intermediate blocks from disk due to great I/O overhead and random file seeks takes place using both Vanilla Spark as well as Spark with its provided External Shuffle Service. A map task creates a shuffle file that consists of a certain number of partitions. This quantity is determined by the number of total tasks that constitute the next stage in the Spark job. Moreover, each block (i.e., partition) of data will be required only one time for a certain reduce task. To put it differently, a wide-dependency stage that consists of M map tasks on the map stage and N reduce tasks on the reduce stage and requires shuffling, will create M shuffle files on the map side, and MxN connections will be created due to MxN shuffle blocks required to be fetched. Thus, the all-to-all communication over the network of the cluster in Spark's architecture can't be avoided. This means that many random file seeks will be required, and as a result, the performance of the system will degrade on heavy shuffle workloads with small sized blocks. Thus, we decided to create a policy that is task-aware.

4.2.2.1 Caching shuffle blocks instead of shuffle files

We firstly thought of adding in Cherry's memory the whole shuffle data file every time that it would be requested. However, OOM errors would occur, since, on real workloads with big datasets of TBs of data each shuffle file is large in size and we would not be able to proactively fit many shuffle intermediate files in cache. Additionally, each shuffle file would be required as many times as the partitions that it contains, and many random evictions would occur, adding overhead to the performance of the shuffle service. Therefore, we ended up aggressively caching only the needed blocks of the shuffle

files per Spark task, utilizing the informed MapStatus data structure that we proactively retrieved from the DAGScheduler of the Spark Driver.

In order to explain thoroughly what happens in a reduce stage of a Spark job with Cherry, we present Figure 4.4. It consists of 7 tasks and is executed by 5 executors. Using this as an example, we outline Cherry's task-aware caching policy. At the moment T_0, Cherry will have already acquired and processed the appropriate information from the Spark Driver about the first task of the executor 4, and will have cached the blocks that the latter will require to fetch. Cherry will evict each block that will successfully get fetched. At the moment T_1, Cherry will be serving shuffle blocks from its cache to the executor 2 for its second task (previous shuffle files from older tasks that were running on executors 1, 3-5 have already been evicted by Cherry). Finally, at T_2, Cherry will only have in memory the blocks that were not fetched because they were located locally in the executors. In a more general note, at any specific moment in time (i.e., vertical lines in Figure 4.4), only the shuffle blocks of the active reducers that are currently fetching data (i.e., red rectangles) will be stored at Cherry, thus limiting the maximum amount of required cache memory irrespective of the entire intermediate data size.

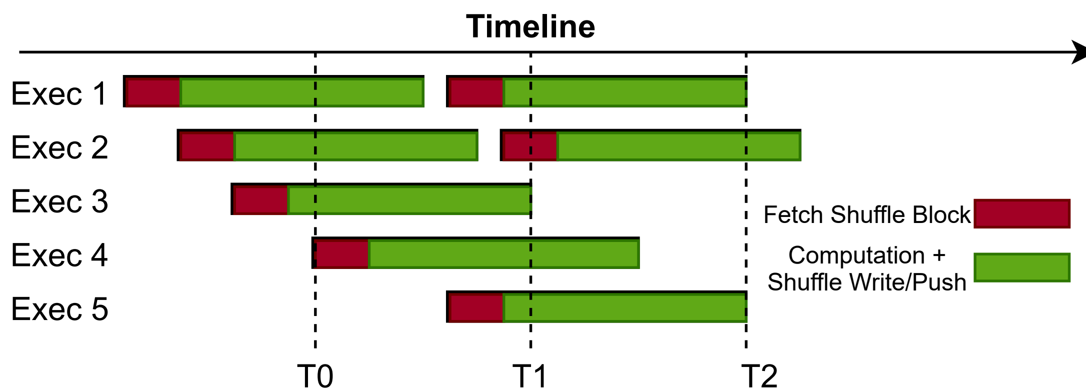

Figure 4.4: *Reduce stage of a Spark job. At each moment Cherry aggressively caches the shuffle blocks of the upcoming tasks. After the fetch completion, each block gets evicted.*

4.2.2.2 Implementing the Caching Policy

The DAGScheduler is located on the Spark Driver and is responsible for the following: it manages the task assignment to executors and transmits all the appropriate information to them in 'waves', based on maximum executor parallelization possible, if there is no data skew. Hence, we modified the DAGScheduler as follows. At the (reduce) task creation phase, where the Spark Driver predefines the exact task creation phase and the data that the latter will process, Cherry consumes this information (Step 4, Figure 4.2), and can detect the exact block IDs that will be requested beforehand (i.e, look-ahead). By acquiring this knowledge, the latter is able to aggressively cache only the necessary blocks of each shuffle file on task-level by buffering them in memory, and have them ready for fetching (Step 5, Figure 4.2). The requested blocks and their belonging shuffle files can be easily detected since we know the sequence of the upcoming reduce tasks that will be executed in the Spark cluster. A Cherry shuffle service provides the requested partitions to an executor in a best-effort manner, since, when a shuffle block is not in memory, it can simply get it from its local disk, which is how

Spark normally works. When the remote shuffle block fetching process finishes from the reduce task after it has been initialized (Steps 6-7, Figure 4.2), Cherry can discard the cached blocks, since they will not be needed again by the upcoming tasks, to avoid cache overflow. Then, it can simply fill the cache with the next partitions according to the expected tasks required to be executed. In case the whole memory space is used, FIFO eviction of shuffle blocks takes place if necessary. At the end of the reduce tasks, the executors inform the Spark Driver that their task is completed (Step 8, Figure 4.2).

To show in depth how this mechanism functions, a sequence diagram in class level is depicted below, in Figure 4.5. For brevity, we will illustrate only the required parts of the sequence that are enough to understand its functionality. More specifically, the depicted classes of the Spark Driver are the *CoarseGrainedSchedulerBackend* which is a backend process that implements a Driver RPC endpoint and waits for executors to connect and exchange messages, and the *DAGScheduler* that is responsible for computing the DAG of a job, submitting stages and tasks, as well as tracking possible failures and selecting preferred locations for task execution. One of the selected classes shown from the Cherry shuffle services is the *ShuffleServiceBackend* which extends the RpcEndpoint class and implements a specific RPC endpoint for the receiving and processing of explicit RPC messages about information of the launching of upcoming tasks from the Spark Driver. Additionally, other classes that are depicted are the *MapOutPutTrackerShuffleService* that is an instance of the MapOutputTracker class and has the role of decoupling map output information from the MapStatus component that is already received from the Spark Driver, and the *ShuffleLocalBlockRetrieverIterator* which is an iterator for the requests blocks that appends the latter in a specific queue. Finally, other shown classes are the *OneForOneLocalBlockRetriever* which explores the structure of the blockIds and creates FetchShuffleBlocks message for each block, the *RemoteBlockHandler*, and the *RemoteShuffleBlockResolver*, which is a modified version of the ExternalBlockHandler class of Spark that contains the data structure of cached blocks, brings them in cache and serves them when they are requested.

In a Spark job, the last stage is labeled as ResultStage, whereas the others are labeled as ShuffleMapStage. Accordingly, the tasks that belong in a ShuffleMapStage are labeled as ShuffleMapTasks, and the tasks in a ResultStage are labeled as ResultsTasks. Each executor that is assigned a task firstly requires a message of *LaunchTask* type that contains the description of the assigned task (i.e., TaskDescription) to decompose and process the necessary information. Additionally, the executor requires a message, named taskBinary, from the DAGScheduler that is broadcasted and contains info about the RDD that corresponds to the stage, as well as another object based on the stage type. More specifically, on ShuffleMapStages this field is named 'shuffleDep' and describes the shufffle that this stage is part of, while on ResultStages this field is named 'func' is the function to apply in each partition. Another information that is required once per stage from every Spark Worker and is received by the Spark Driver is the array of MapStatus instances of the shuffle that the running stage is dependent on, in a serialized format. The MapOutputTrackerMaster that is instanciated in the Spark Driver extends the MapOutputTracker class and has the role of bookkeeping the information of mappping a map index to a MapStatus instance for each stage, as well as its serialized form. The MapStatus array of the parent stage(s) will be complete after the latter is finished, and it will include the map output locations of shuffle blocks in an ef-

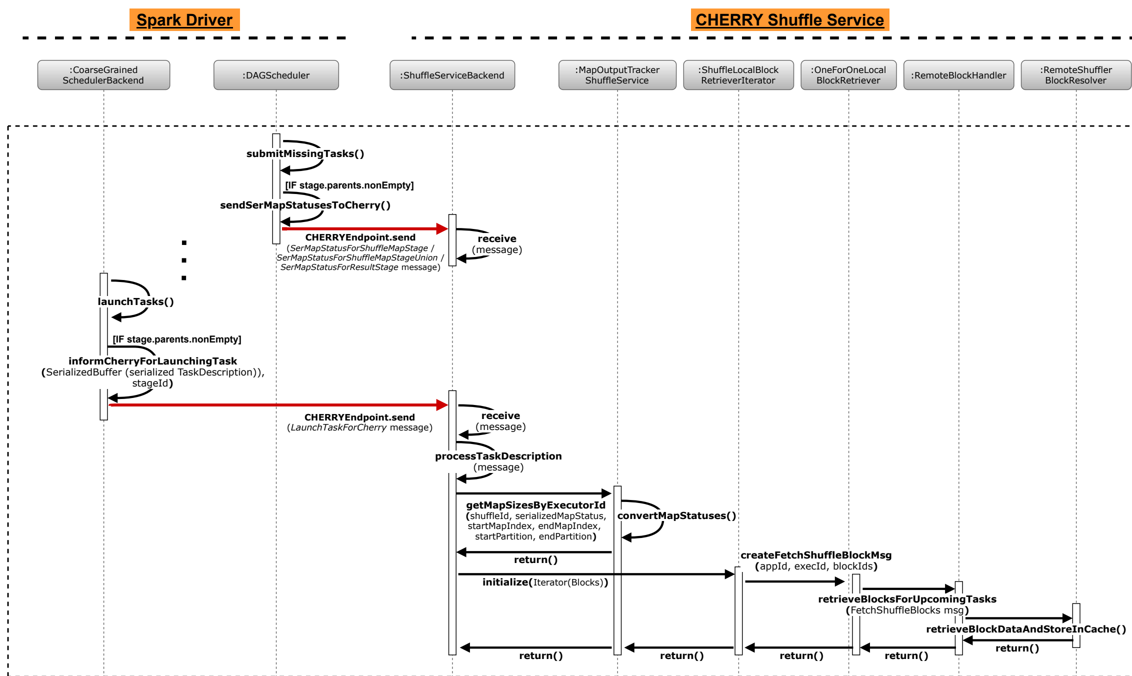

Figure 4.5: *Look-Ahead Task-Aware Caching Policy.*

ficient byte format ready to be sent to the reduce tasks of the child stage. However, we found essential only the MapStatus array and the description of each task for a Cherry shuffle service to track, to find and proactively store in cache the shuffle partitions of each upcoming task.

After the DAGScheduler computes the DAG of stages for a job, it submits each stage sequentially based on dependencies and shuffles required, broadcasts the aforementioned value taskBinary, computes the sequence of the required tasks combined with the necessary information, and finally submits them to the TaskScheduler. Thus, after the broadcast, the DAGScheduler examines if the running stage has parent stages. This is used since the first stage of the DAG requires no shuffling, and we avoid sending any unnecessary messages to the Cherry services. Subsequently, it calls the *sendSerMapStatusesToCherry()* function. In this function it is examined if the running stage is dependent in one or more parent stages. If it has more than one parent stages, it means that this reduce stage is after a join or union transformation of 2 or more rdds, while if it has one parent, this reduce stage is after e.g. a groupByKey or sortByKey transformation. After tracing the source code of Spark we found that the most used rdd types are the UnionRDD and the ShuffledRowRDD classes, based on the number of parent stages. Hence, we emphasized on these classes. If the running stage is dependent on one only stage, we collect the shuffle Id that this stage is dependent on and we serialize the MapStatus array of the parent stage. Then, we send an implemented RPC message asynchronously named **SerMapStatusForShuffleMapStage** or **SerMapStatusForResultStage** if the running stage is a ShuffleMapStage or a ResultStage, to the ShuffleServiceBackend of each Cherry Pod that includes the serialized array of MapStatuses for each partition, the stage Id, the number of partitions for the running stage, and the shuffle Id. The knowledge of the MapStatus array will make the Cherry capable of tracing the shuffle files that are about to be requested and, consequently, bring them in memory for faster access. When the Cherry's ShuffleServiceBackend instance

receives this message, it stores the acquired information in a HashMap based on the stageId. If the running stage is dependent on more than one stage, an array of the Partition class is included in a new RPC message asynchronously named **SerMapStatusForShuffleMapStageUnion**, as well as an array of the parent shuffle Ids and an array of the parent MapStatuses in a compressed format. This will later allow Cherry to trace and bring in cache shuffle blocks based on different parent rdds and stages. The DAGScheduler, then, continues to task submissions. This message is a One-way-message and the DAGScheduler waits for no response.

Later, when the sequence control is in the CoarseGrainedSchedulerBackend instance of the Spark Driver, where the *launchTasks* method is called, if for the aforementioned reason the running stage of the launching task has parent stages, the method **informCherryForLaunchingTask** is called. This happens for each and every task of a Spark job. More specifically, it is sent just before informing the pre-selected (based on the already defined DAG of the Spark job and the available resources) executor that will take on a task. In this method, we send to the available Cherry RPC endpoints in the cluster the description of the upcoming task, named as TaskDescription, and the stage Id that the launching task is a part of, as a *LaunchTaskForCherry* RPC type message. When the ShuffleServiceBackend instance receives the information, the *processTaskDescription()* method is called, and continues on processing it and consequently caching the upcoming shuffle blocks that will be requested shortly. More specifically, we process the message and inspect if the upcoming task is a ResultTask or a ShuffleMapTask, as well as how many parent stages the stage of the current task has, using the already acquired message from the DAGScheduler at the beginning of each stage.

Then, the *getMapSizesByExecutorId()* method is called from the MapOutputTrackerShuffleService instance where the shuffleId, startMapIndex, endMapIndex, startPartitionId, endPartitionId and serialized mapStatus that is already stored are passed. Subsequently, the *convertMapStatuses()* method is called that returns an iterator of the shuffle block IDs and corresponding shuffle block sizes based on the block manager ID. Each Cherry Pod stores a block manager ID for every shuffle block that it receives from the Spark executors, that includes the Cherry's IP and Port and the executor Id that pushed each block. Thus, we then filter the iterator to keep only the requested blocks that are stored locally on each Cherry. Later, the *initialize()* method of the LocalBlockRetrieverIterator instance is called on which the filtered iterator of blocks is passed. In this function the blocks from the iterator are added in a queue and are forwarded to the *createFetchShuffleBlocksMsg()* method of the OneForOneLocalBlockRetriever instance to analyze the pass in the blockIds and create FetchShuffleBlock components. The ShuffleLocalBlockRetrieverIterator and OneForOneLocalBlockRetriever classes are similar in functionality as the ShuffleBlockFetcherIterator and OneForOneBlockFetcher classes of Spark accordingly. Each FetchShuffleBlock message is forwarded to the *retrieveBlocksForUpcomingTasks* method of the RemoteBlockHandler instance, where the blocks included in the message are transformed to an iterator, and each object is forwarded to the *retrieveBlockDataAndStoreInCache()* function of the RemoteShuffleBlockResolver instance. At this point, the parameters of the method are the appId, execId, shuffleId, mapId, startReduceId and endReduceId. The data structure in the RemoteShuffleBlockResolver instance that we proactively store the shuffle blocks is a LinkedHashMap. Each key is a unique string that includes the path of the shuffle data file in the local disk, the

length and the offset requested. There will be no key collisions since each partition will be requested only once. The value of our data structure is a FileSegmentManagedBuffer object of each block. We also tested different data structures like a separate HashMap for each executor to store its intermediate data, and the cache interface from the Google Guava interface [14]. Nevertheless, there was no improvement in shuffle reduce time or job completion time. The messages exchanged when the procedure of fetching shuffle block data takes place are the same with Vanilla Spark, and each shuffle block is returned as a FileSegmentManagedBuffer object. Moreover, if a shuffle intermediate block has not been already proactively cached when requested, it is served as native Spark works. The shuffle blocks that were cached but not fetched because they were located locally in the respective executors get evicted at the end of the Spark workload, so as to free up all available memory resources for new upcoming jobs. The procedure of how Cherry caches the shuffle blocks on task-level is also described in Algorithm 1.

Algorithm 1: Caching Shuffle Blocks

taskDesc: description info for each task that is launched
stageId: stage Id of the launching task
mapStatuses: array of MapStatus instances that keeps mappings from map index to output locations for each partition of a stage
shuffleId: Id of the shuffle that a task depends on
blocksIter: iterator of blocks to be fetched for a task
q: queue to store blocks before processing their data
cache: cache to store shuffle blocks

```
1  begin
2      blocksIter = ConvertMapStatuses
           (shuffleId, stageId, mapStatuses, taskDesc)
3      blocksIter = Filter(blocksIter)
4      q ← {}
5      for all block in blocksIter do
6          q ← q ∪ block
7      for i ← 1 ... len(q) do
8          blockData = GetBlockInfo(q(block_i))
9          blockKey = GetUniqueKey(blockKey)
10         blockValue = ManagedBuffer(blockData)
11         cache.put(blockKey, blockValue)
12 end
```

4.2.3 Efficient Memory Usage and Low Cache Miss Rate

As explained in Figure 4.4, Cherry's caching mechanism performs expeditiously while the required allocated memory resources throughout a big data workload are low, since only a small portion of tasks from a stage is being executed at each moment in time. This happens due to the fact that the available executors in a Spark cluster are usually less than the number of these tasks. One of the main bottlenecks that may occur when utilizing memory resources are the cache misses, which can cause significant delays. We wanted to benchmark if there are many cache misses when trying to locate

shuffle blocks in memory, in order to serve the latter to executors, based on the size of available memory.

Our implemented benchmark included an execution of a shuffle synthetic workload of 40GB with 10KB shuffle block size, 10 Spark executors and 10 available Cherry Pods in the cluster. We also examined using different percentages of total available cache relative to the total shuffle data volume (i.e., percentage of data that fit simultaneously in cache), which were from 10% up to 100%. The outcome of our experiments was that the cache misses that arose are below 1% for all test cases, even if all shuffle data could presumably fit in memory. This happens because only a part of the shuffle blocks required is being fetched at a time, and Cherry's caching policy optimally leverages any acquired information from the Spark Driver for a few upcoming reduce tasks by caching only these shuffle blocks and maintaining its memory footprint low. Therefore, the implementation and integration of Cherry as a shuffle service with a data analytics framework such as Spark does not require massive memory resources to perform efficiently, a feature which is important to reduce the total costs of allocating cloud resources.

4.2.4 Fault Tolerance of Cherry

The crashes and failures of nodes in large-scale cloud systems is a pretty usual phenomenon due to out of space errors from hard disks, software issues, faulty hardware, etc. . Accordingly, in a Spark job there is a high likelihood of loss of a Spark Worker and any state that is stored locally. Cherry's mechanism of pushing and storing remotely any ephemeral shuffle data from executors offers fault tolerance in Spark workloads. More specifically, Cherry leverages this attribute and transforms Spark into a serverless framework, since, in case of Worker node failures or deallocations, no shuffle state will be lost, and any task re-computation of the lineage of the job will be avoided.

Moreover, our implementation of Cherry provides resilience even in case of failures of the Cherry Pods. More specifically, the Metadata Service has the role of keeping track of the alive Cherry shuffle services in a cluster. In case it realizes that a Cherry Pod crashes, it immediately informs the Spark Workers. Thus, on upcoming map tasks the executors push their shuffle intermediate data to the rest of the available Cherry pods with a round-robin selection. Also, on an upcoming reduce task, if the required shuffle blocks were lost due to a failure of Cherry, the executors will fetch the intermediate data that were stored in the lost shuffle service directly from the rest of the Workers of the Spark cluster. This is possible since the Spark Driver also tracks this knowledge of the map output locations for the running job, and this is how Spark natively works. The fault-tolerant behavior of Cherry will be better displayed in a benchmark subsequently in section 5.2.2.

4.3 Cluster Setup

4.3.1 Use of Docker and Kubernetes

Cherry uses Kubernetes (version *v1.19.0*) as a cluster orchestrator because it can flexibly and seamlessly allocate and deallocate nodes from our cluster, deploy Pods and

ensure the liveness of containers via receiving heartbeats. Kubernetes, in our case, is exloited to facilitate and automate the process of deploying and terminating containers and services, and set up all the necessary network communication paths precisely. For the Kubernetes installation to take place, the Docker software package (version *20.10.5*) was required. Additionally, the *kubectl* command-line tool of Kubernetes was needed in order to run commands against the Kubernetes cluster. Another installation that took place was *Calico* [7] that is an open source networking solution for virtual machines and containers and operates seamlessly with Kubernetes.

Each of the available components in our cluster are executed inside a Docker container that is included in a Kubernetes Pod. Thus, we created and Dockerfile that includes all the required libraries and binaries as well as the compiled source code. Moreover, we created a Docker Image that is the local built of the aforementioned Dockerfile, and pushed it to Docker Hub to a private repository, so as to easily retrieve it in the cluster. In order to build and compile the modified Spark source code alongside its required dependencies, we used Maven [20], which is a build automation tool.

In our Kubernetes cluster, all the components of Spark belong in a separate namespace in order to establish a first level of isolation between the latter and other components operating in the cluster. Moreover, each of the Spark Driver, Worker, Master and Cherry shuffle Service components has its own Kubernetes Service. With Kubernetes Services, a set of Pods can be defined and grouped in a higher level abstraction. Each Pod has its own unique cluster-private IP, but a set of Pods that are targeted by the same Service share the same DNS label name that can be easily identified by the rest of the cluster, and Kubernetes manages the load-balancing of requests towards the latter. In our case, we use the Services to easily port forward and expose in groups the necessary ports of Pods to the cluster for interconnection of the Spark components, as well as allowing Spark Workers to communicate with the Spark Master, the Spark Driver, the Metadata Service, and the Cherry Pod (if distributed mode is not enabled) by DNS names without knowing any IP beforehand, since they are bound dynamically. Additionally, we have implemented the Spark Workers and the Cherry Shuffle Services as Kubernetes Deployments of Pods, so as to seamlessly scale in or out the number running Pods in a deployment. We, also, decided to execute the Spark Driver as a Kubernetes Job since we want to reliably run a Spark program to completion. All Kubernetes components can be expressed in a .yaml format. Below, we present an example YAML file for the creation of a Spark Worker Deployment (4.1), as well as a YAML specification for the service that the latter will be targeted (4.2).

Listing 4.1: *YAML file for a Spark Worker Deployment.*

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: spark-worker
5    labels:
6      component: spark-worker
7  spec:
8    replicas: 1 # number of replicas in a deployment
9    selector:
10     matchLabels:
```

```yaml
      component: spark-worker
  template:
    metadata:
      labels:
        component: spark-worker
    spec:
      containers:  # containers that run in each Pod
        - name: spark-worker
          image: nikoshet/k8s-spark:spark-Cherry
          command: ["/spark/spark-worker.sh"]
          ports:
            - containerPort: 8081
          resources:
            requests: # set requests for each Pod
              cpu: 1000m
              memory: 2Gi
            limits: # set limits for each Pod
              memory: 2.5Gi
          volumeMounts:
            - mountPath: data
              name: csv-path
      volumes:
        - name: csv-path
          hostPath: # directory location on host
            path: /home/ubuntu/my-data
            type: Directory
      nodeSelector:
        type: worker
```

Listing 4.2: *YAML file for a Spark Worker Service.*

```yaml
kind: Service
apiVersion: v1
metadata:
  name: spark-worker
  labels:
    component: spark-worker
spec:
  clusterIP: "None"
  ports:
    - name: spark-ui
      port: 8081
      targetPort: 8081
  selector:
    component: spark-worker
```

In Listing 4.1, firstly we define the Kubernetes API version that will be used, and that we want to create a specific Kubernetes object, a Deployment. Additionally, the segment of metadata helps us identify the Deployment in the Kubernetes cluster and

attach it to a Service. In the part of spec we specify what we want to execute. We can define the number of the replica set of Pods that will be part of the deployment. Using the Kubectl command tool we will then be able to scale the number of Spark Worker replicas at ease. The .spec.selector field describes how a Deployment discovers which Pods it has to manage. Moreover, the .spec.metadata is used for labeling the Pods in a Deployment, and in the spec.template.spec.containers we can list all the containers that will run in every Pod of the Deployment. Thus, we define the name spark-worker for our container, we set the Docker image that it will be pulled and run as well as the command with the bash script that will initiate the launch of the Spark Worker. Moreover, in the spec.template.spec.containers.ports.containerPort we set the port that will be exposed to the cluster from the Kubernetes Service for the Spark Worker. We also define the requests and limits on CPU and memory resources. Finally, we mount a directory of data from our node inside the Pod at a defined path in order to use them for a Spark workload, and select the nodes that the Deployment will take place, with the condition that the nodes of the cluster have already been labeled accordingly. Respectively, for the Service we define the same metadata and spec.selector fields to target the Spark Worker Deployment, and select a name and port to expose the Ports of the Spark Workers that will allow the access to their Spark Web UI. The spec.clusterIP field that is set to None is for creating a 'headless' Service that does not allocate an IP and does not require load balancing.

4.3.2 Automating the K8S Cluster Creation

To accelerate the procedure of the Kubernetes Master and Worker nodes configuration, the initialization of the monitoring instances, as well as the required management so as to achieve seamless start and execution of Spark workloads we use Ansible with version *2.9.19*. To be more precise, we installed Ansible only on the local machine since there is no installation of Ansible agents required to the remote cluster, due to its flexible architecture. Additionally, we created the necessary Ansbile playbooks in order to install all dependencies and create the required networking, Services and Deployments in the cluster, as well as facilitate the procedure of clearing the RAM, copying data from the master to the worker nodes or updating the Docker Images to all the nodes. This automation helped us save a lot of time while making changes on the code, recompiling the source code of Spark and testing with different configurations.

4.3.3 Monitoring

Our implementation provides monitoring capabilities in order to observe and collect real-time metrics of the performance of Spark workloads when they are being executed. More specifically, we integrated the Prometheus Operator [23] in our Kubernetes cluster. The Prometheus Operator is a software that utilizes Kubernetes custom resources to simplify the deployment and configuration of Prometheus instances, Grafana and other services. It was installed using Helm [16], which is a package manager for Kubernetes. It can scrape metrics through the Kubernetes API and automatically deploy multiple *Node Exporter* Pods on each node of the cluster, in order to get analytics on node level. In Cherry we utilize Prometheus of version *2.18.2* and Grafana of version *7.0.3*.

We additionally enabled Spark to export default Prometheus metrics via its configuration and modified the *PrometheusResource* class to scrape additional metrics. In order for Prometheus to have access to the metrics of the Spark Master, Worker and Driver Pods it is necessary to primarily create a ServiceMonitor to configure how the corresponding Spark services should be monitored as metric endpoints. Thus, in a YAML file, we set the scrape interval, the service that the specified ServiceMonitor will have to target, and finally the port and the path for Prometheus to monitor. The additional Spark metrics are the number of waiting and running stages of a Spark job, the number of total and completed tasks as well as the execution time for each running stage. For the Cherry Pods we did not create any Prometheus targets for scraping since we will acquire metrics from the Node Exporters. The architecture diagram of how a Spark workload is monitored in a Kubernetes cluster is shown in Figure 4.6. With this implementation, the user can observe real-time metrics of the system in node, Spark, and Kubernetes level through a web interface at ease. An example use case scenario is that the user can easily notice when the aggressive caching takes place in Cherry due to our task-aware policy, when there is a sharp increase in the memory usage of the Cherry shuffle service Pod.

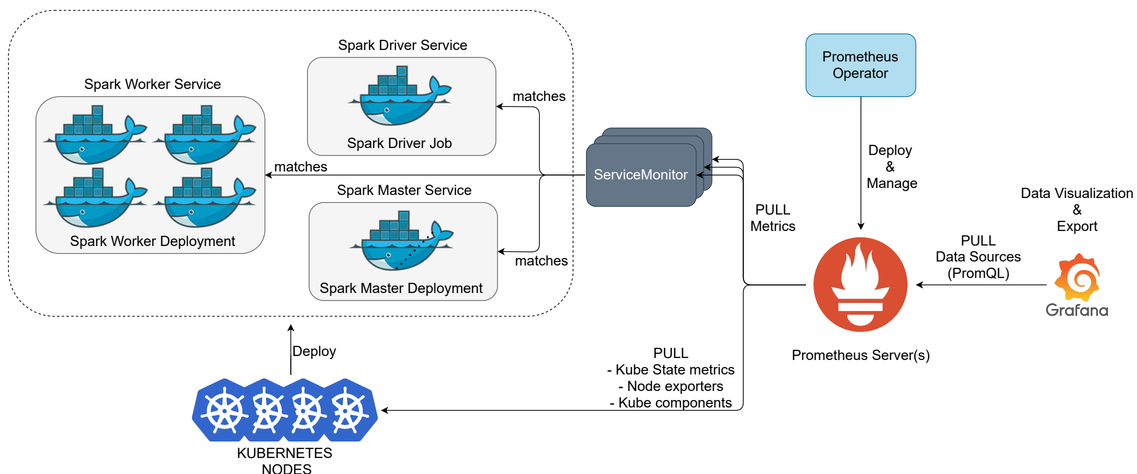

Figure 4.6: *Spark Cluster Monitoring with Kubernetes, Prometheus and Grafana.*

4.4 Autoscaling Module

We also implemented a module, named Metrics Monitor, that achieves automatic horizontal scaling of the number of deployed Spark Worker Pods that participate in a Spark workload. More specifically, we implemented a Python program that queries specific custom metrics constantly every several seconds, collects the results and decides if in or out autoscaling is required based on some well defined logic criteria. This mechanism works as follows. Since all metrics are firstly collected from the Prometheus Server, we send HTTP GET requests to its available HTTP endpoint that is at the url http://localhost:9090/api/v1/query. Moreover each Prometheus expression query string is passed as a parameter in the URL GET request, and the result is returned from the Prometheus Server in JSON format. Additionally, we created a separate Dockerfile that includes all the necessary libraries and binaries for the latter to run seamlessly in a K8S Pod. Current collected metrics are the CPU and Memory usage of Worker Pods, the

available Memory per Kubernetes node, as well as the CPU usage per node.

We implemented a simple mechanism that examines if autoscaling is required. More explicitly, our Metrics Monitor, every time that it collects these values, it checks if CPU usage of Worker Pods is below or above 90%, and removes or adds one Pod respectively. This takes place by initiating the execution of a separate bash script that sends a request to the Kubernetes API. In order to achieve that, since our module is running in a Pod, it requires specific privileges and authorization. To emphasize on that, we enabled and utilized Role-based access control (RBAC) [24], in order to dynamically configure the number of alive Spark Worker Pods in the cluster based on the aforementioned metrics, through the K8s API. In order to achieve that, we also created a ClusterRole that gives permission for actions on deployments. Furthermore, we initialized a ServiceAccount for the Metrics Monitor Pod, and configured a RoleBinding to bind the the ClusterRole with the ServiceAccount and authorize the Pod to communicate with the K8s API. Finally, we executed some simple Spark workloads and confirmed that our Autoscaling feature works correctly based on CPU load of Spark Worker Pods. Although the main goal of this diploma thesis is not the autoscaling capability of a Spark cluster, with this module we wanted to illustrate that our research can easily be extended towards this direction seamlessly by developing a smart logic criteria.

Chapter 5

Experimental Evaluation

In this chapter our evaluations alongside their results are presented. We examined different types of benchmarks, synthetic and real, which showcase that Cherry can decrease the disk read/write latencies of small shuffle partitions by proactively caching the shuffle blocks on task-level, and, consequently improve the performance of Spark workloads. Additionally, Cherry offers fault tolerance due to its serverless architecture and maintains low resource footprint.

5.1 Evaluation Setup

We have evaluated the performance of Cherry on both realistic and synthetic benchmarks against Apache Spark. Our evaluation setup consists of 11 virtual nodes, each of which has 4 threads from Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz and 8GB DDR4 RAM @2133MHz, and are connected with a 10 Gbps Ethernet link. Every Spark Worker Pod allocates 1 thread and 2GB RAM, and has 1 executor. These Pods are completely stateless, since, only soft state is stored temporarily and shuffle intermediate data are maintained remotely with the Cherry Pods which are the storage engine of the system. Each Cherry Pod has 1 thread and 2GB RAM available. Additionally, the Spark Master, Spark Driver and Metadata Service Pods allocate 1 thread and 1GB RAM each. A separate physical node is allocated for the execution of the Spark Master, Spark Driver and Metadata Service Pods, while five are dedicated to the Spark Workers and five for the Cherry shuffle services. We believe that a fair comparison of Spark with our implementation is the same number of External Shuffle Services versus Cherry shuffle services. Thus, we decided to run 10 Spark Worker Pods and 10 Cherry Pods for all our experiments. Additionally, since both Spark with its ESS as well as vanilla Spark perform similarly, we will mainly compare our implementation with Spark with its ESS, and use Vanilla Spark only for our synthetic shuffle workload.

5.2 Synthetic Workload Evaluations

We firstly wanted to observe how Cherry performs with Apache Spark on a synthetic benchmark, and emphasize on the shuffle operation of Spark. The benchmark that we created includes a map and a reduce stage of a shuffle heavy synthetic workload, which is similar to [64]. Through this benchmark we can compare the performance of Cherry

against Vanilla Spark and Spark using its ESS, and consequently evaluate them on the reduce stage time, fault tolerance, scalability and resource consumption.

5.2.1 Stage and Completion Time

The shuffle phase of workloads where the reduce tasks require to fetch intermediate shuffle data remotely is the period that depicts the optimized performance of Cherry. Thus, we monitored the total reduce stage time of our synthetic Spark workload. It needs to be highlighted that the completion time of the map stage is similar for all variations since there is no optimization there and the time spent on the process of pushing shuffle data on Cherry Pods is insignificant. We decided to examine out benchmark on different number of tasks per stage and block size, and used the same number of map and reduce tasks on each stage. The number of map tasks equivalents to the number of shuffle files created, while the number of reduce tasks is the same with the number of blocks per shuffle file. Table 5.1 shows the configurations of the conducted experiments.

Table 5.1: *Experiment configurations. Each row shows the total size of the synthetic dataset, the number of map or reduce tasks, and the size of each block.*

	Size	**# M/ # R tasks**	**Block size**
1	50 GB	400	250 KB
2	50 GB	2000	10 KB
3	50 GB	4000	2.5 KB

The results of our experiment are illustrated in Figure 5.1, where we compare our optimized shuffle service against Spark with ESS, Vanilla Spark, and Cherry without caching, and the shuffle data size is 50GB. When it comes to small shuffle block sizes of 2.5KB, the disk I/O bottleneck is apparent and degrades the performance of the workload execution on Spark with ESS and Vanilla Spark. On the contrary, Cherry manages to overcome this problem by utilizing its look-ahead caching policy on each task and have its requested shuffle block in memory before the fetching operation, since any I/O will take place beforehand and not in the critical moment of shuffle block fetching. Moreover, Cherry without its caching mechanism has a analogous performance as Vanilla Spark. However, Cherry without caching can leverage its other features on large-scale data analytics workloads, such as disaggregation and fault tolerance.

With the size of 2.5KB for the intermediate shuffle blocks, Cherry achieves an almost 23% reduction in completion of the reduce stage time against Spark with ESS, and an almost 39% reduction against Vanilla Spark. More specifically, Spark with ESS needs 196 seconds for the completion of the reduce stage that includes reading the intermediate data with a 2.5KB shuffle block size, Vanilla Spark requires 248 seconds, Cherry without caching needs 240 seconds, while our optimized shuffle service needs only 152 seconds. As the block size increases for the other 2 experiments of 10KB and 250KB sizes, the performance of Spark with ESS, Vanilla Spark, Cherry without caching and Cherry improve similarly. In more detail, for 10KB shuffle data size, Cherry scores 118 seconds, Spark with ESS needs 112 seconds, Vanilla Spark requires 132 seconds, and Cherry without its look-ahead caching policy needs 130 seconds. Additionally, when the size of each shuffle block of the synthetic workload is set to be 250KB, the required time is 95 seconds,

103 seconds, 105 seconds and 103 seconds respectively. Thus, Cherry offers a great advantage in the reduce phase when it comes to reading small shuffle block sizes, a feature that is frequent in large cloud environments with numerous separate nodes.

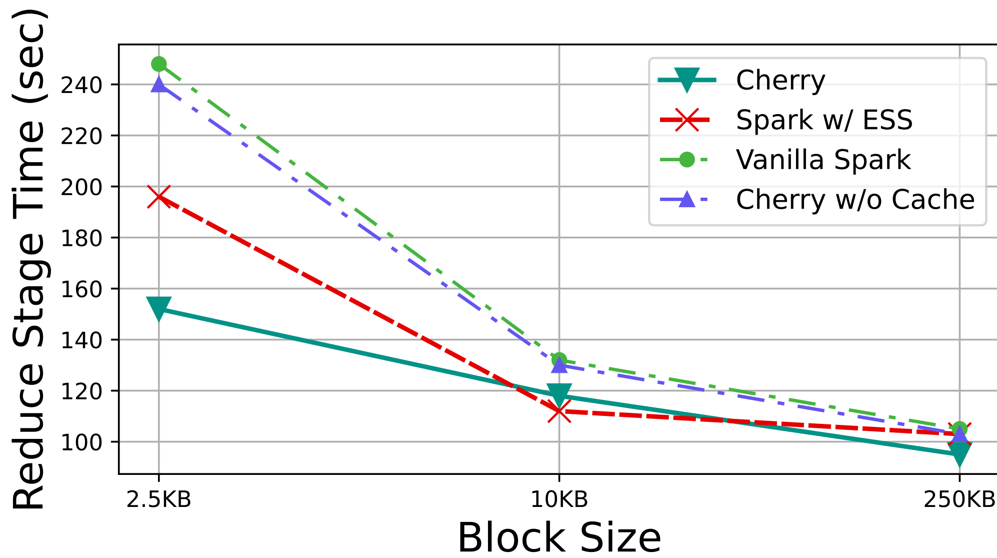

Figure 5.1: *Experiment results of our benchmark with synthetic workload. Smaller block sizes affect severely the I/O performance of Spark with ESS. On the other hand, Cherry achieves a better overall performance on the completion of the reduce stage.*

5.2.2 Fault Tolerance in Spark Workloads

The implementation and integration of Cherry in Spark workloads offers great fault tolerance for the latter, since, all intermediate data are stored remotely and, thus, each Spark workload turns into a serverless job where the execution of the Spark Worker Pods and executors is ephemeral, and, in case of node failures where the Spark Workers are running, no shuffle data will be required to be re-executed. We estimate that the approximate additional time that is required on a specific stage of a Spark job without Cherry is given by the Equation (5.1). In this respect, T_{Spark} is the additional time needed, p is the percentage of the completed tasks in the stage, t is the total number of tasks of the current stage, c is the completion time for a task, e is the total number of executors in the Spark cluster and l is the number of lost executors when a Spark Worker crashes. With Cherry, the additional re-computation overhead is constant, and equal to the completion time of the tasks that were running at the failure time, which is c (i.e., $T_{Cherry} = c$).

$$T_{Spark} = \frac{ptl}{e(e-l)} * c \tag{5.1}$$

In order to prove its fault tolerant feature, we conducted the following experiment. We run our shuffle synthetic workload with 10 Cherry Pods and 10 Spark Worker Pods, created 20GB with 1000 mappers and 1000 reducers and a block size of 20KB. Furthermore, we forcibly killed a Spark Worker Pod when different percentages of the entire map stage have been completed, and measured the additional required re-computation time for the tasks of which the shuffle data were lost. Figure 5.2 illustrates the results

of the aforementioned experiment. In our case, the completion time of a task is c=25 seconds. For all different cases conducted, Cherry's stateless architecture allows the re-execution of the task that was currently running when the failure occurred, achieving a minimal constant overhead of around c=25 seconds. On the contrary, with vanilla Spark, all the tasks that were computed by the killed executors by the time that the failure occurred have to be re-executed, adding a great overhead in the overall Spark workload completion time.

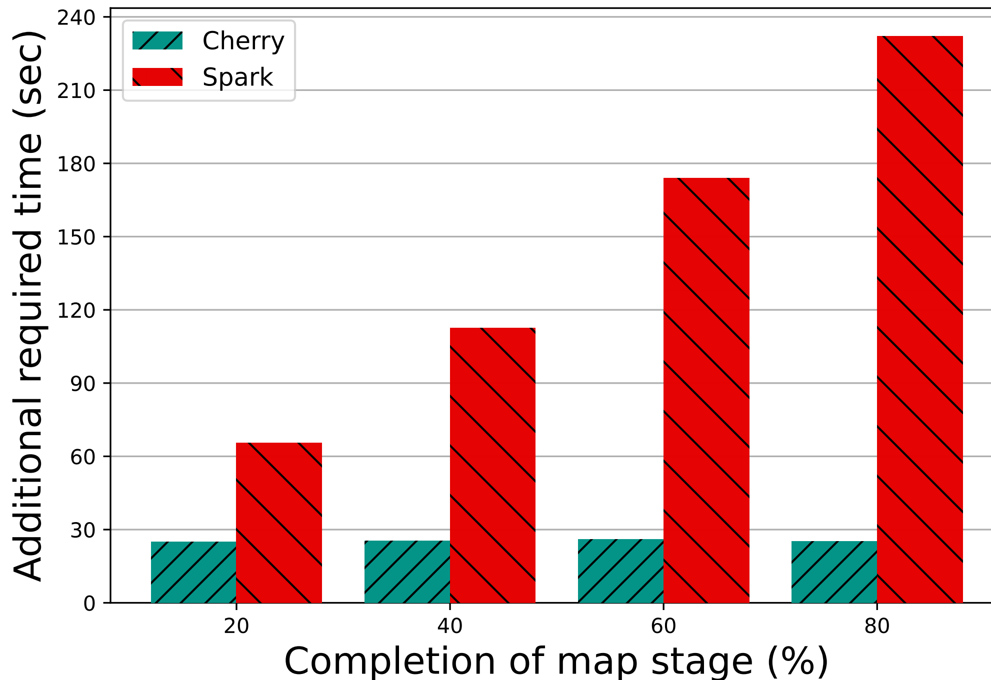

Figure 5.2: *Additional time needed for a map stage to complete when a Spark Worker crashes, depending on the percentage of the stage that has been already completed.*

Using different values of the parameters on the Equation (5.1), there will be divergent additional overhead. For example, in one of our cases with p=40%, t=1000, c=25 seconds, e=10 and l=1, the additional time is T_{Spark}=111.1 seconds, which is close to the measured value from our experiment, which is 112.5 seconds. Moreover, in another case with p=80%, t=1000, c=25 seconds, e=10 and l=1, the required added time based on the Equation (5.1) is T_{Spark}=222.2 seconds, while the corresponding measured time is 232 seconds, concluding that the values are almost similar. However, using Cherry, we manage to have an overhead reduction of 88.7%. On a more abstract level, a theoretical example is as follows: for p=75%, t=4000, c=500 seconds, e=20 and l=2 (usually, several executors are running in each physical node in the cluster), that results in T_{Spark}=8333.3 seconds, which is $\approx$ 2.31 h of additional time, while with Cherry we will need only T_{Cherry}=500 seconds. Therefore, Cherry's disaggregated architecture and serverless manner of execution of Spark workloads achieves great fault tolerance regarding Spark in real production environments with minimal additional overheads.

5.2.3 Scalability and Resource Efficiency

The use of Kubernetes as a container orchestrator, alongside Cherry's architecture enables its seamless scalability in a Spark cluster. As shown in section 4.4, our imple-

mented autoscaling module is capable of dynamically updating the active number of Spark Worker Pods based on specific metrics critetia. Accordingly, due to the bookkeeping role of the Metadata Service, the number of the available Cherry shuffle services can easily change, since each one registers and deregisters from the Metadata Service. Consequently, when there are additions or removals of the Cherry instances in the cluster, the Metadata Service informs each Spark Worker.

Moreover, Cherry has a low resource usage, both on CPU and memory end. To illustrate that, we run our synthetic workload with different number of Cherry shuffle services available within the cluster and 10 Spark Worker Pods, and monitored the CPU usage of each Cherry Pod, when the shuffle block fetch operation of the reduce stage takes place. Figure 5.3 shows the results of the conducted experiment. As the number of Cherry Pods is increased, the latter uses less amount of a CPU core. This happens since the amount of requests from executors that have to be handled per Cherry in a certain time frame is reduced. Additionally, when the cluster has 10 Cherry shuffle services, each one uses about 20% of a CPU core. We, also, monitored the CPU utilization of one of the 10 Spark Workers utilizing its External Shuffle Service instead of Cherry, and measured that it needs about 35% of a CPU core. This is due to the fact that the Worker's executor has to fetch its local intermediate data for its respective assigned tasks, as well as its ESS has to respond to requests and serve the shuffle data accordingly.

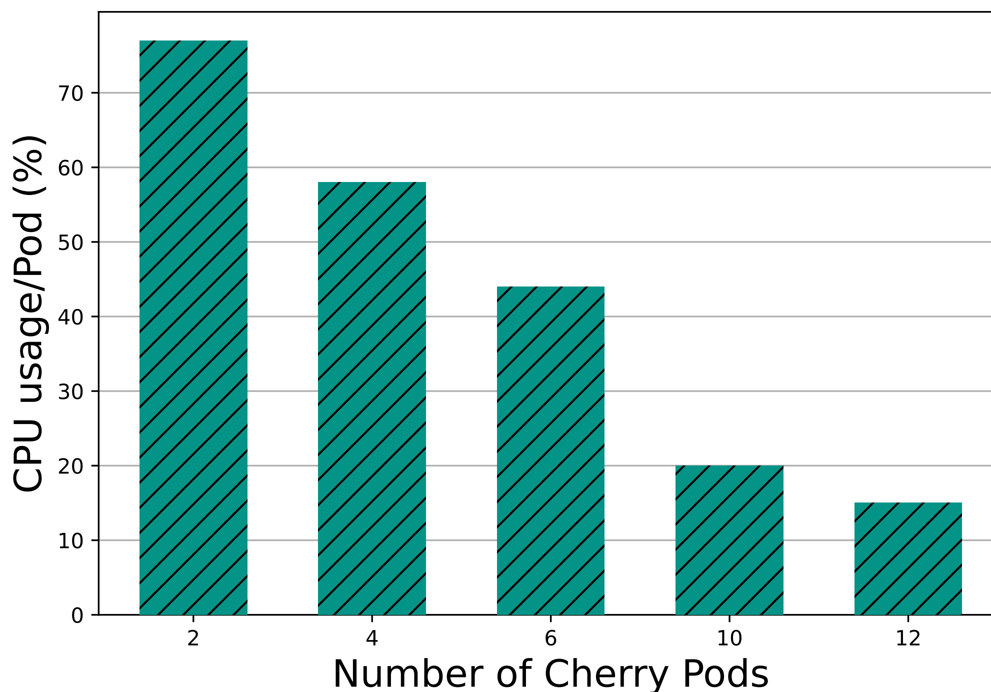

Figure 5.3: *CPU usage per Cherry Pod while varying the number of them in the cluster.*

When it comes to memory consumption in our Spark cluster, we examined how cache is utilized in all Cherry shuffle services for shuffling 50GB of data. Figure 5.4 illustrates how cache utilization changes through time, compared to the normalized shuffle blocks that are fetched by executors in the workload. Since Cherry caches and evicts immediately the blocks that are fetched from the Spark executors, we can see that the cache consumption increases slowly, compared to the intermediate data that are served. At the end of the workload, it ends up containing in cache around 18% of the

shuffle data fetched. This slow and steady increase is due to the blocks that are cached but are not fetched, since they are located locally in their respective executors. The percentage of the shuffle data that will remain cached and not served depends on the number of Spark workers and Cherry shuffle services in a Spark cluster. Additionally, through our measurements, we found out that only 10% of the total shuffled data at most remains in the total cache of all Cherry shuffle services after a job completion. In order to retrieve the maximum available memory for any upcoming workloads, all of these shuffle blocks are evicted at the end of of the Spark job.

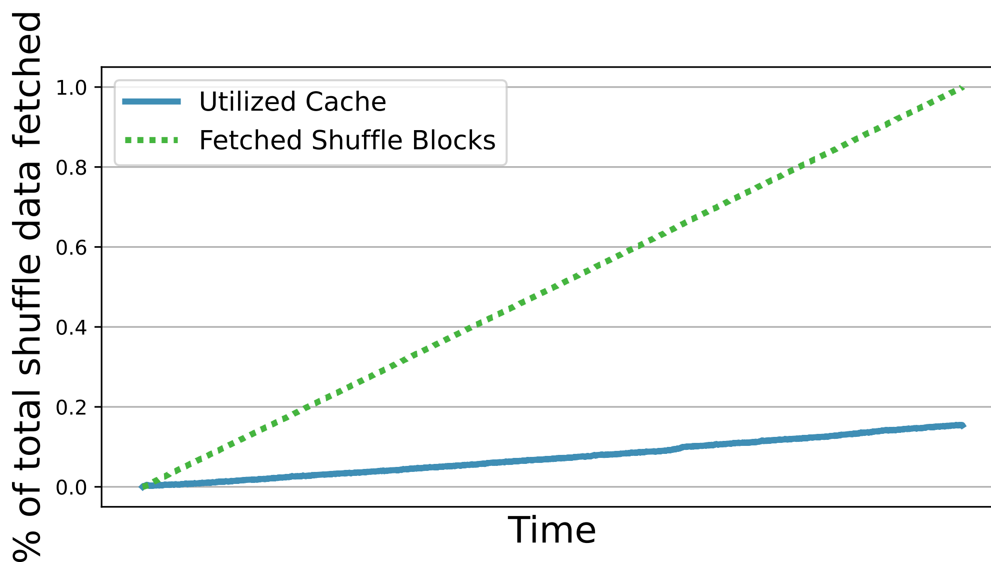

Figure 5.4: *Normalized Cherry Cache Usage compared to fetched shuffle blocks by executors through time.*

5.2.4 Data Skewness

One of the common severe problems that occur in large-scale analytics workloads and can severely impact the performance and completion time are the data skews and task stragglers. The operation of coping with these is of vital importance, and there is recent research that emphasizes on that [37, 31]. Although this area is not our main focus in this work, we wanted to showcase how Cherry performs and deals with skewed data in a Spark job.

In order to examine this challenging aspect, we created a synthetic workload that requires 20GB of data shuffling with 10KB shuffle block size and allows the tuning of the percentage of the data skewness. Consequently, we executed it on our implemented Spark cluster, and emphasized again on the reduce stage time, as in subsection 5.2.1. Figure 5.5 shows the experiment results. We can see that Cherry's caching policy and distributed architecture accomplishes relatively faster completion time of the reduce stage time. More specifically, Cherry achieves from 2% to 8% better performance, with the best score being on 80% skewed data with 1850 seconds, while Spark with ESS requires 1994 seconds to complete its reduce stage. Thus, we believe that on large-scale workloads where TBs of shuffle data are created through a lineage, Cherry's features

can improve the data shuffle operation and reduce the job completion time, even when there is severe data skewness.

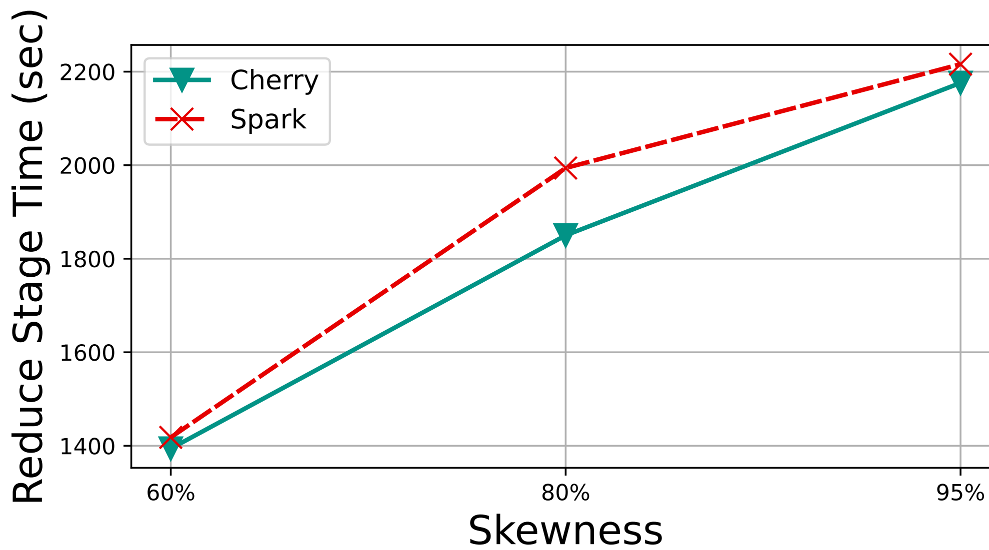

Figure 5.5: *Performance of Spark using Cherry on skewed data against Vanilla Spark.*

5.3 Real Workload Evaluations

We further wanted to examin Cherry on realistic workloads, and evaluate how it performs in a Spark cluster, against using Spark with its ESS.

5.3.1 TPC-DS Queries

We decided to use the TPC-DS benchmark and its available queries [56]. This is a decision support benchmark that models a general decision support system of a retail product supplier and includes a wide variety of SQL queries, such as ad hoc, reporting, data mining and iterative queries. For our evaluation, we use the same hardware resources and number of Pods in our cluster as used in our synthetic workloads (more details in section 5.1), which is 10 Spark Workers as well as 10 disaggregated Cherry shuffle services of 1 CPU core and 2GB RAM each. Furthermore, we generated TPC-DS data with a scale factor of 100, which corresponds to all its respective tables adding up to a total input size of 100GB data, and stored them in a HDFS cluster of 3 virtual nodes.

Figure 5.6 illustrates the completion time of executing the two selected TPC-DS queries on Spark with its External Shuffle Services and Spark with Cherry. Both executed queries, Query 16 and Query 94, are general report type queries that combine different tables and include shuffle-heavy operations. For Q16, Cherry manages to reduce its completion time from 364 seconds to 324 seconds, which is 11% reduction in completion time. Moreover, for Q94, Spark with ESS requires 234 seconds to complete it, while the utilization of Cherry with Spark needs 214 seconds, which corresponds to 8.5% reduction in total execution time. In conclusion, the architecture and features of

Cherry can beneficially execute real workloads and significantly reduce their completion time, comparing to Vanilla Spark and Spark with its available ESS.

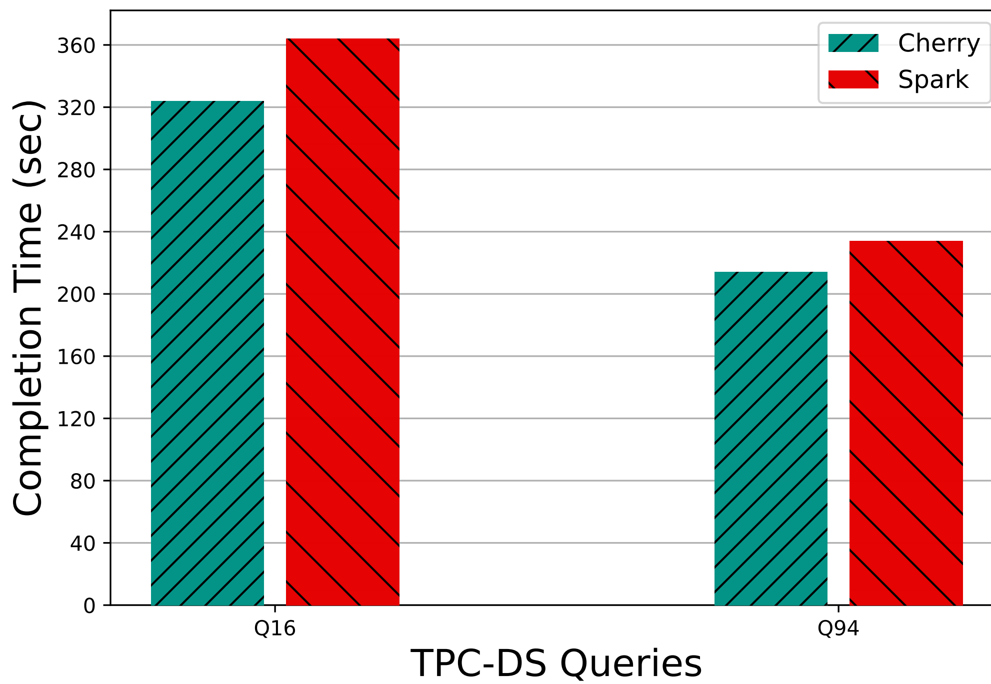

Figure 5.6: *Execution results of TPS-DS queries on Spark and Cherry.*

Chapter 6

Conclusions and Future Work

In this chapter we close our thesis. We shortly present the conclusions that we came to through our research and work. Additionally, we suggest possible future work and directions that could improve our research and suggested implementation in several different ways.

6.1 Summary and Conclusions

The serverless architecture and its adoption in large-scale data analytics systems in cloud environments has been of great interest by industry and academia in recent years. Additionally, the storing and handling of the intermediate small-sized shuffle data in these jobs has a vital role and can cause a great overhead if not dealt with correctly.

In the current thesis we present Cherry, a distributed shuffle service for large-scale analytics workloads that leverages a look-ahead caching policy on task-level that efficiently improves the I/O bottlenecks on the shuffle operation of small shuffle block sizes. Cherry is disaggregated from the Spark components and can store all intermediate data from workloads seamlessly, transforming Spark into a pure serverless analytics engine. It also achieves fault tolerance in case of node failures and has low resource footprint.

Based on the experiments that are presented in chapter 5, we observed that Cherry overcomes the challenges of shuffling in Spark and improves the required time of shuffle fetching in analytics jobs, both synthetic and realistic. This takes place since with its task-level caching policy, any requested shuffle block is proactively maintained in cache and is offered from there for the respective executors. Additionally, Cherry's fault tolerant architecture significantly reduces any additional needed time to re-compute any tasks that were completed by crashed Spark Workers, since all shuffle data are stored remotely.

Moreover, through certain experiments and measurements we proved that Cherry uses less CPU resources than a Spark Worker when shuffle data are requested by the reducer executors. Additionally, we observed how effortlessly we can scale in and out the available Cherry shuffle services in the cluster due to the whole architecture of our system and the implementation of the Metadata Service. This also happens since Kubernetes is used, and we ascertained how beneficial and flexible K8s is in large cloud systems. Finally, when it comes to skewed datasets, Cherry can adequately deal with

these and improve the execution time of these workloads.

The original MapReduce framework and all of its descendants employ an intermediate shuffling mechanism that, at its simplest form, utilizes the local filesystem of the participating computing units (i.e., executors). All shuffling approaches, either local or remote, require a persistent storage engine and an addressing mechanism to maintain and retrieve intermediate data: storing is done by utilizing POSIX read/write system calls in the local case or RPC Put/Get calls through network in the remote case and addressing is done via a combination of executor IP address and filename (i.e., reduce bucket), so as to uniquely identify the exact intermediate data that needs to be fetched from its respective executor.

These attributes (i.e., shuffle storage and addressing) are being taken into consideration by Cherry throughout its design, and a lean approach with well-defined external system interactions is followed, to facilitate its generic applicability: In Figure 4.2 it is illustrated that Cherry interacts with the external big data processing system (i.e., Spark in our case) only in its Put/Get and task info creation steps, thus requiring system specific code to be implemented only there. In fact, code changes to the big data processing system are required only in the respective steps: Cherry requires Spark to adapt only its Put (step 2) call, and task info call (step 4), whereas the adaptation is minimal with around of 1000 lines for the configuration of API messages between Cherry and Spark (i.e., Cherry code inside Spark). On the other hand, Cherry requires around of 1000 lines of additional Spark-specific code in its implementation for processing the aforementioned messages and storing the intermediate data locally, as well as caching the required blocks to be fetched (step 5) and responding to Get requests of Spark executors (step 7).

Generic caching policies (for instance, FIFO, LRU, NFU, etc.) are employed when the data access pattern is not known beforehand, something that we overcome in our case: by exploiting task information from the DAG Scheduler, we know both the exact blocks that are going to be requested and the moment that this is going to take place. Thus, the proposed look-ahead block-based caching policy comes as a natural optimization that Cherry can achieve by utilizing its two distinguishing characteristics, namely the external shuffle storage service (through disaggregation) and the exact data access pattern knowledge. Regarding the caching policy generic applicability, since all those big data frameworks employ a task scheduler that assigns data and tasks to executors, we can effortlessly extend Cherry's architecture by consuming similar system-specific scheduling info.

6.2 Future Work

In this section we suggest several directions and extensions for future work. There is a wide feasibility in large-scale serverles analytics systems that indicates promising research and results.

Experiments on Ultra-Large-Scale Systems. The experiments conducted in the

present thesis are executed in a specific large cloud system with certain available resources, are thorough, and examine different aspects and problems that need to be addressed in data analytics workloads. However, we believe that it is worth evaluating the performance of Cherry with Spark workloads in larger scale relative to available cloud resources as well as shuffle data created. With this approach, we will be able to implement a more stress testing environment where each job requires to process 100s of TB of data, a phenomenon which is common in realistic everyday workloads.

Implementation on Different Frameworks. Cherry's current implementation makes straightforward the way the interaction between the latter and Apache Spark takes place. This is due to the fact that minimal modifications and adaptations are needed for these to co-operate seamlessly. Thus, it is interesting to evaluate Cherry's performance as an ephemeral data storage with other frameworks such as Apache Flink and Hadoop MapReduce.

Cost-Performance Optimization. Large-scale data workloads are usually required to be executed in cloud environments. This means that the costs of the allocated resources need to be taken into consideration. Although Cherry offers a fast and fault-tolerant way of maintaining all the shuffle data of workloads, a smart optimizer that could calculate re-computation expenses in case of node failures and would balance the cloud expenses and resources that Cherry needs seems promising and really feasible.

Autoscaling Policy. We utilized Kubernetes as a cluster manager, that allows seamless initialization, scaling and removing of Spark components based on available resources. We also built a Metrics Monitor that, based on well defined logic criteria, can dynamically scale in and out the Spark Worker Pods in our cluster. Thus, we believe that the development of certain more complicated and smart heuristics that enable the elastic auto-scaling of both compute (i.e., Spark Workers) as well as storage (i.e., Cherry shuffle services) resources would be really beneficial in respect to costs and resources allocated.

Merging Shuffle Blocks. Cherry's current caching mechanism is operating on task-level, achieving low memory footprint, since it aggressively caches only the required blocks per task and then evicts them from memory. A possible optimization approach would be to try and merge these blocks with a certain logic beforehand and cache only one object per task. This would lead to 1 cache access instead of N accesses (i.e., N blocks per task), reducing any possible overhead that this procedure may cause.

Appendix A

Source Code

The source code of the presented thesis can be found in the following link here.

The Docker image of our implementation that can be pulled is also publicly available in the following link here.

Appendix B

List of Abbreviations

API	:	Application Programming Interface
RDD	:	Resilient Distributed Dataset
K8S	:	Kubernetes
ESS	:	External Shuffle Service
DAG	:	Directed Acyclic Graph
AWS	:	Amazon Web Services
EC2	:	Elastic Compute Cloud
S3	:	Simple Storage Service
PaaS	:	Platform as a Service
SaaS	:	Software as a Service
IaaS	:	Infrastructure as a Service
FaaS	:	Function as a Service
HDFS	:	Hadoop Distributed File System
VM	:	Virtual Machine
IOPS	:	Input/output operations per second
OOM	:	Out-Of-Memory
JVM	:	Java Virtual Machine
RPC	:	Remote Procedure Call
RBAC	:	Role-based access control

Bibliography

[1] Apache hadoop. http://hadoop.apache.org [Online].

[2] Apache mesos. https://mesos.apache.org/ [Online].

[3] Aws fargate. https://aws.amazon.com/fargate/ [Online].

[4] Aws lambda. https://aws.amazon.com/lambda/ [Online].

[5] Aws step functions. https://aws.amazon.com/step-functions [Online].

[6] Azure functions. https://azure.microsoft.com/en-us/services/functions/ [Online].

[7] Calico networking. https://www.projectcalico.org/calico-networking-for-kubernetes/ [Online].

[8] Cosco: An efficient facebook-scale shuffle service. https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service [Online].

[9] Docker. www.docker.com [Online].

[10] Docker hub. https://hub.docker.com/ [Online].

[11] Elastic compute cloud. https://aws.amazon.com/ec2 [Online].

[12] Google cloud functions. https://cloud.google.com/functions [Online].

[13] Google cloud run. https://cloud.google.com/run [Online].

[14] Google guava. https://github.com/google/guava/wiki/CachesExplained [Online].

[15] Grafana. https://grafana.com/ [Online].

[16] Helm. https://helm.sh/ [Online].

[17] Ibm cloud functions. https://cloud.ibm.com/functions [Online].

[18] Improving apache spark by taking advantage of disaggregated architecture. https://databricks.com/session_eu19/improving-apache-spark-by-taking-advantage-of-disaggregated-architecture [Online].

[19] Kubernetes. https://kubernetes.io/docs/home/ [Online].

[20] Maven. https://maven.apache.org/ [Online].

[21] Oracle cloud functions. https://www.oracle.com/cloud-native/functions/ [Online].

[22] Prometheus. https://prometheus.io/ [Online].

[23] Prometheus operator. https://github.com/prometheus-operator/prometheus-operator [Online].

[24] Rbac. https://kubernetes.io/docs/reference/access-authn-authz/rbac/ [Online].

[25] Spark achitecture. https://spark.apache.org/docs/latest/cluster-overview.html [Online].

[26] Yarn. https://yarnpkg.com/ [Online].

[27] Presto | Distributed SQL Query Engine for Big Data, 2013. http://prestodb.github.io/[Online].

[28] Presto exchange materialization, 2013. https://prestodb.io/docs/current/admin/exchange-materialization.html[Online].

[29] Flink, 2014. https://flink.apache.org/[Online].

[30] Dataflow, 2015. https://cloud.google.com/dataflow[Online].

[31] Spark sql adaptive execution at 100 tb, 2018. https://software.intel.com/content/www/us/en/develop/articles/spark-sql-adaptive-execution-at-100-tb.html [Online].

[32] Flink shuffle service, 2019. https://cwiki.apache.org/confluence/display/FLINK/FLIP-31%3A+Pluggable+Shuffle+Service[Online].

[33] Hadoop pluggable shuffle, 2020. https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/PluggableShuffleAndPluggableSort.html[Online].

[34] Dataflow shuffle, 2021. https://cloud.google.com/dataflow/docs/guides/deploying-a-pipeline#cloud-dataflow-shuffl[Online].

[35] Spark external shuffle service, 2021. https://spark.apache.org/docs/latest/job-scheduling.html#configuration-and-setup[Online].

[36] Adzic, G., and Chatley, R. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (2017), pp. 884–889.

[37] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective straggler mitigation: Attack of the clones. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)* (2013), pp. 185–198.

[38] Ansible. https://www.ansible.com [Online].

[39] Barcelona-Pons, D., Sánchez-Artigas, M., París, G., Sutra, P., and García-López, P. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference* (2019), pp. 41–54.

[40] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[41] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[42] Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS 28*, 13 (2009), 2009.

[43] Garefalakis, P., Karanasos, K., Pietzuch, P., Suresh, A., and Rao, S. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–13.

[44] Ginzburg, S., and Freedman, M. J. Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing* (2020), pp. 43–48.

[45] Guo, Y., Rao, J., Cheng, D., and Zhou, X. ishuffle: Improving hadoop performance with shuffle-on-write. *IEEE transactions on parallel and distributed systems 28*, 6 (2016), 1649–1662.

[46] Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., and Wu, C. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[47] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[48] Kim, Y., and Lin, J. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)* (2018), IEEE, pp. 451–455.

[49] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 427–444.

[50] Kubernetes components. `https://kubernetes.io/docs/concepts/overview/components/` [Online].

[51] Kvm. `www.linux-kvm.org` [Online].

[52] Lxc. `https://linuxcontainers.org` [Online].

[53] Mcluckie, C. Containers, vms, ku-bernetes and vmware, 2014. `http://googlecloudplatform.blogspot.com/2014/08/containers-vms-kubernetes-and-vmware.html`.

[54] Müller, I., Bruno, R. F., Klimovic, A., Alonso, G., Wilkes, J., and Sedlar, E. Serverless clusters: The missing piece for interactive batch applications? In *10th Workshop on Systems for Post-Moore Architectures (SPMA'20)* (2020).

[55] Müller, I., Marroquín, R., and Alonso, G. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 115–130.

[56] Nambiar, R. O., and Poess, M. The making of tpc-ds. In *VLDB* (2006), vol. 6, pp. 1049–1058.

[57] Or, A., Zhang, H., and Freedman, M. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems 2* (2020), 400–411.

[58] Pahl, C., Brogi, A., Soldani, J., and Jamshidi, P. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing 7*, 3 (2017), 677–692.

[59] Pu, Q., Venkataraman, S., and Stoica, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 193–206.

[60] Rao, S., Ramakrishnan, R., Silberstein, A., Ovsiannikov, M., and Reeves, D. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing* (2012), pp. 1–14.

[61] Sagiroglu, S., and Sinanc, D. Big data: A review. In *2013 international conference on collaboration technologies and systems (CTS)* (2013), IEEE, pp. 42–47.

[62] Schleier-Smith, J., Holz, L., Pemberton, N., and Hellerstein, J. M. A faas file system for serverless computing. *arXiv preprint arXiv:2009.09845* (2020).

[63] Shankar, V., Krauth, K., Pu, Q., Jonas, E., Venkataraman, S., Stoica, I., Recht, B., and Ragan-Kelley, J. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[64] Shen, M., Zhou, Y., and Singh, C. Magnet: push-based shuffle service for large-scale data processing. *Proceedings of the VLDB Endowment 13*, 12 (2020), 3382–3395.

[65] Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B., and Özcan, F. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment 8*, 13 (2015), 2110–2121.

[66] Sreekanti, V., Wu, C., Lin, X. C., Schleier-Smith, J., Faleiro, J. M., Gonzalez, J. E., Hellerstein, J. M., and Tumanov, A. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592* (2020).

[67] Stuedi, P., Trivedi, A., Pfefferle, J., Stoica, R., Metzler, B., Ioannou, N., and Koltsidas, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull. 40*, 1 (2017), 38–49.

[68] Wang, J., and Balazinska, M. Elastic memory management for cloud data analytics. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)* (2017), pp. 745–758.

[69] Wu, C., Faleiro, J., Lin, Y., and Hellerstein, J. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering* (2019).

[70] Wu, C., Sreekanti, V., and Hellerstein, J. M. Autoscaling tiered cloud storage in anna. *The VLDB Journal 30*, 1 (2021), 25–43.

[71] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012), pp. 15–28.

[72] Zhang, H., Cho, B., Seyfe, E., Ching, A., and Freedman, M. J. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–15.

[73] Zhang, T., Xie, D., Li, F., and Stutsman, R. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 1–12.