BigStorage

STORAGE-BASED CONVERGENCE BETWEEN HPC AND CLOUD TO HANDLE BIG DATA

Deliverable number	D2.2
Deliverable title	WP2 DATA SCIENCE – Final Report
Editor	Gabriel Antoniu (Inria)
Main Authors	Alvaro Brandon (UPM), Ovidiu Marcu (Inria), Pierre Matri (UPM), Yacine Taleb (Inria), Alexandru Costan (Inria), Maria S. Pérez (UPM)

Grant Agreement number	642963
Project ref. no	MSCA-ITN-2014-ETN-642963
Project acronym	BigStorage
Project full name	BigStorage: Storage-based convergence between HPC and Cloud to handle Big Data
Starting date (dur.)	1/1/2015 (48 months)
Ending date	31/12/2018
Project website	http://www.bigstorage-project.eu

Coordinator	María S. Pérez
Address	Campus de Montegancedo sn. 28660 Boadilla del Monte, Madrid, Spain
Reply to	mperez@fi.upm.es
Phone	+34-91-336-7380

Executive Summary

This document provides an overview of the work done until M48 of the Project BigStorage (from 01-01-2015 until 31-12-2018) in WP2 Data Science.

WP2 was dedicated to Data Science, with a particular focus on data processing models, energy-efficient data analytics and data-driven decision making. In the first part of the project, we investigated the limitations and bottlenecks of current frameworks for general workflow/streaming Big Data applications (see deliverable D5.1). We analyzed the application requirements in relation with the use cases studied in WP1, then made progress beyond the state of the art along several directions.

- We investigated the impact of the different architectural choices for Big Data processing engines on end-to-end performance to understand the current limitations faced by streaming engines when interacting with a storage system for holding streaming state. We proposed a set of design principles for a scalable, unified architecture for data ingestion and storage, called **KerA**. Its goal is to efficiently support diverse access patterns: low-latency access to stream records and/or high throughput access to (unbounded) streams and/or objects. This contribution was published **at ICDCS 2018, a CORE A-level conference [Mar18]**.
- We investigated performance vs. fault tolerance trade-offs at storage level and proposed an innovative replication protocol for scale-out in-memory databases in support of Big Data analytics, called **Tailwind**. This contribution was published **at ATC'18, a CORE A-level conference [Tal18]**.
- We investigated and proposed machine-learning techniques to model optimal decisions for task parallelization to process Big Data workloads. This contribution was published in **FGCS, a Q1 impact factor journal [Bra18]**.
- We investigated and proposed a graph-based root cause analysis framework that leverages monitored metrics of the system to facilitate the troubleshooting of problems in microservice architectures. This contribution has been submitted to the **Journal of Systems and Software, a Q1 impact factor journal.**

This report provides an overview of these contributions that we consider representative of WP2. We chose to focus in a more detailed way on the two contributions that were materialized through two software research prototypes (KerA and Tailwind) and we provide a brief overview of the last two contributions (machine learning for task parallelization and graph-based root cause analysis framework).

Document Information

IST Project Number	MSCA-ITN-2014-ETN-642963
Acronym	BigStorage
Title	Storage-based convergence between HPC and Cloud to handle Big Data
Project URL	http://www.bigstorage-project.eu
Document URL	http://bigstorage-project.eu/index.php/deliverables
EU Project Officer	Mr. Szymon Sroda
Deliverable	D2.2 Intermediate Report on WP2
Workpackage	WP2 Data Science
Date of Delivery	Planned: 31.12.2018 Actual: 30.12.2018
Status	Version 0.5 final □ draft ■
Nature	prototype □ report ■ dissemination □
Dissemination level	public □ consortium ■
Distribution List	Consortium Partners
Document Location	http://bigstorage-project.eu/index.php/deliverables
Responsible Editor	Gabriel Antoniu (Inria), gabriel.antoniu@inria.fr, Tel: +33299847244
Authors (Partner)	Alvaro Brandon (UPM), Ovidiu Marcu (Inria), Pierre Matri (Inria)
Reviewers	Alexandru Costan (Inria), María S. Pérez (UPM)
Abstract (for dissemination)	Executive Summary
Keywords	Data processing, data science, Big Data processing, stream processing, workflow processing, Spark, Flink, machine learning

Version	Modification(s)	Date	Author(s)
0.1	Initial template and structure	22.12.2018	Gabriel Antoniu, Inria
0.2	Sections from all authors	26.10.2018	All authors
0.3	Internal version for review	28.12.2018	Gabriel Antoniu, Inria
0.5	Comments from internal reviewers	29.12.2018	Internal reviewers
0.6	Final version to commission	30.12.2018	Gabriel Antoniu, Inria
0.7	Final version reviewed	31.12.2018	María S. Pérez, UPM

Project Consortium Information

Participants		Contact
Universidad Politécnica de Madrid (UPM), Spain		María S. Pérez Email: mperez@fi.upm.es
Barcelona Supercomputing Center (BSC), Spain		Toni Cortes Email: toni.cortes@bsc.es
Johannes Gutenberg University (JGU) Mainz, Germany		André Brinkmann Email: brinkman@uni-mainz.de
Inria, France		Gabriel Antoniu Email: gabriel.antoniu@inria.fr Adrian Lebre Email: adrien.lebre@inria.fr
Foundation for Research and Technology - Hellas (FORTH), Greece		Angelos Bilas Email: bilas@ics.forth.gr
Seagate, UK		Sai Narasimhamurthy Email: sai.narasimhamurthy@seagate.com
DKRZ, Germany		Thomas Ludwig Email: ludwig@dkrz.de
CA Technologies Development Spain (CA), Spain		Victor Muntes Email: Victor.Muntes@ca.com
CEA, France		Jacque Charles Lafoucriere Email: Charles.LAFOUCRIERE@CEA.FR
Fujitsu Technology Solutions GMBH, Germany		Sepp Stieger Email: sepp.stieger@ts.fujitsu.com

Table of Contents

1 Introduction

1.1 *WP2 Overview*

This subsection provides an overview of WP2 as was presented in the project proposal.

The Data Science has emerged as the fourth paradigm for scientific discovery, based on data-intensive computing [Tan09]. Motivated by the emergence of Big Data applications, Data Science involves several data-centric aspects: storage, manipulation, analysis with statistics and machine learning, decision-making, among others. During the recent years, the MapReduce [Dea04] programming model (e.g., Amazon Elastic MapReduce, Hadoop on Azure - HDInsight) has emerged as the de facto standard for Big Data processing. However, many Data Science applications do not fit this model and require a more general data orchestration, independent of any programming model. Modeling them as workflows is an option [3,4], but current cloud infrastructures lack specific support for efficient workflow data handling. State-of-the-art solutions rely on high-latency storage services (Azure Blobs, Amazon S3) or implement application-specific overlays that pipeline data from one task to another. However, in large-scale distributed environments consisting of multiple datacenters, this would suffer from latencies when accessing large remote datasets frequently.

Data Science involves unprecedented complexity in the Big Data management process, which is not addressed by existing cloud data handling services. This WP aimed to investigate new models, mechanisms and policies that are needed to support dynamic coordination for data processing, dissemination, analysis and exploitation across widely distributed sites with reasonable QoS levels.

WP2 was organized in 3 tasks, as follows.

Task 2.1. Big Data Processing Models
The objective of this task was to design next-generation data processing models for applications that require general data orchestration, independent of any programming model. Based on the use cases studied in WP1, we investigated workflows composed of many tasks, linked via data- and control-flow dependencies, with a particular focus on stream data processing. We examined the limitations and bottlenecks of state-of-the-art solutions; we defined a set of requirements for efficient real-time data processing, then we designed storage-based techniques enabling fast, efficient stream-based processing.

Task 2.2 Green Big Data Analysis

This task focused on investigating new techniques for energy-efficient Big Data analysis, in particular the energy-efficiency vs. performance trade-offs for in-memory storage systems used for Big Data analytics. As this task is strongly focused on energy-related aspects, for consistency, we chose to describe the related results in deliverable D5.2, which gathers together all contributions related to energy-efficiency.

Task 2.3. Data-Driven Decision Making

Predictive insights and accurate predictive models from data are essential in nowadays business applications. This task aimed to train researchers in the expertise of data-driven decision, connecting the low-level (scalable data infrastructures) to the real needs of applications. In this context, in-memory cluster computing platforms have gained momentum in the last years, due to their ability to analyse big amounts of data in parallel. One of the key aspects is optimization of the task parallelism of application in such environments. In this task, we focused on machine learning methods for recommending optimal parameters for task parallelization in big data workloads. Additional key technologies that are being widely adopted in Big Data architectures are containerisation and microservice architectures. Establishing the root cause for failures or performance problems in these kinds of architectures can become very complex, since the different parts of an application are split into small units or containers that perform a specific task. To solve this problem, we propose a graph-based root cause analysis framework that leverages monitored metrics of the system to facilitate the troubleshooting of problems in microservice architectures.

1.2 *ESR Participation and contributions*

The full list of ESRs in the current form of the project is:

ESR	Name	Institution	Main advisor
1	Ovidiu-Cristian Marcu	INRIA	Gabriel Antoniu/Alexandru Costan
2	*Alvaro Brandon*	*UPM*	*Maria S. Perez*
3	Pierre Matri	UPM	Maria S. Perez
4	Muhammad Umar Hameed	Mainz	Andre Brinkmann
5	*Rizkallah Touma*	*BSC*	*Anna Queralt/Toni Cortes*
6	Fotios Papaodyssefs	Seagate	Malcolm Muggeridge
7	Linh Thuy Nguyen	INRIA	Adrien Lebre
8	*Athanasios Kiatipis*	*Fujitsu*	*Sepp Stieger*
9	*Fotis Nikolaidis*	*CEA*	*Philippe Deniel*
10	*Dimitrios Ganosis*	*FORTH*	*Angelos Bilas/Manolis Marazakis*
11	*Nafiseh Moti*	*Mainz*	*Andre Brinkmann*
12	*Georgios Koloventzos*	*BSC*	*Ramon Nou/Toni Cortes*
13	Mohammed-Yacine Taleb	INRIA	Gabriel Antoniu
14	Yevhen Alforov	DKRZ	Thomas Ludwig / Michael Kuhn
15	*Michał Zasadziński*	*CA*	*Victor Muntes*

The ESRs participating in WP2 are ESR 1, ESR2, ESR3, ESR13. They are all also involved in other WPs.

Note: This report aims to present the achievements related to Big Data processing in support of data science. As WP5 is fully dedicated to energy-related aspects, to avoid redundancy, we decided not to detail the energy dimension in this report (the results of task 2.2 being described in deliverable 5.2).

2 Big Data processing: challenges

Our intermediate report (D2.1) presented a detailed survey of state-of-the art systems for Big Data processing, which helped identify a set of gaps and corresponding challenges that we addressed in the second phase of the BigStorage project.

We addressed the following goals in WP2:

- Based on the identified limitations and bottlenecks of current frameworks for general workflow/streaming Big Data applications, we analyzed the application requirements in relation with the use cases studied in WP1.
- We investigated the impact of the different architectural choices for Big Data processing engines on end-to-end performance to understand the current limitations faced by streaming engines when interacting with a storage system for holding streaming state. We proposed a set of design principles for a scalable, unified architecture for data ingestion and storage, called **KerA**. Its goal is to efficiently support diverse access patterns: low-latency access to stream records and/or high throughput access to (unbounded) streams and/or objects. This contribution was published **at ICDCS 2018, a CORE A-level conference [Mar18]**.
- We investigated performance vs. fault tolerance trade-offs at storage level and proposed an innovative replication protocol for scale-out in-memory databases in support of Big Data analytics, called **Tailwind**. This contribution was published **at ATC'18, a CORE A-level conference [Tal18]**.
- We investigated and proposed machine learning techniques to model optimal decisions for task parallelization to process Big Data workloads. This contribution was published in **FGCS, a CORE A-level international journal [Bra18]**.
- We investigated and proposed a graph-based root cause analysis framework that leverages monitored metrics of the system to facilitate the troubleshooting of problems in microservice architectures. This contribution has been submitted to the **Journal of Systems and Software, a Q1 impact factor journal.**

The following three sections provide an overview of these contributions that we consider representative of WP2. We chose to focus in a more detailed way on the two contributions that were materialized through two software research prototypes (KerA and Tailwind) and we provide a brief overview of the last two contributions (machine learning for task parallelization and graph-based root cause analysis framework).

3 KerA: Scalable Data Ingestion for Stream Processing

3.1 *Context and problem*

Big Data real-time stream processing typically relies on message broker solutions that decouple data sources from applications. This translates into a three-stage pipeline described in Figure 1. First, in the production phase, event sources (e.g., smart devices, sensors, etc.) continuously generate streams of records. Second, in the ingestion phase, these records are acquired, partitioned and pre-processed to facilitate consumption. Finally, in the processing phase, Big Data engines consume the stream records using a pull-based model.

Since users are interested in obtaining results as soon as possible, there is a need to minimize the end-to-end latency of the three stage pipeline. This is a non-trivial challenge when records arrive at a fast rate and create the need to support a high throughput at the same time. To this purpose, Big Data engines are typically designed to scale to a large number of simultaneous consumers, which enables processing for millions of records per second [Ven17], [Mia17]. Thus, the weak link of the three stage pipeline is the ingestion phase: it needs to acquire records with a high throughput from the producers, serve the consumers with a high throughput, scale to a large number of producers and consumers, and minimize the write latency of the producers and, respectively, the read latency of the consumers to facilitate low end-to-end latency.

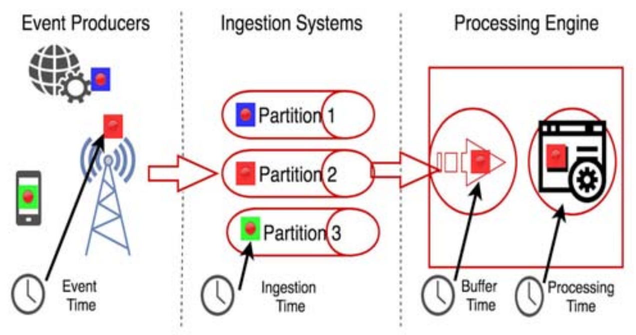

Figure 1. Stream processing pipeline: records are collected at event time and made available to consumers earliest at ingestion time, after the events are acknowledged by producers; processing engines continuously pull these records and buffer them at buffer time, and then deliver them to the processing operators, so results are available at processing time.

Achieving all these objectives simultaneously is challenging, which is why Big Data applications typically rely on specialized ingestion runtimes to implement the ingestion phase. One such popular runtime is Apache Kafka [Kafka]. It quickly rose as the de-facto industry standard for record brokering in end-to-end streaming pipelines. It follows a simple design that allows users to manipulate streams of records similarly to a message queue. More recent ingestion systems (e.g. Apache Pulsar [Pulsar], DistributedLog [DLog]) provide additional features such as durability, geo-replication or strong consistency but leave little room to take advantage of trade-offs between strong consistency and high performance.

State of art ingestion systems typically achieve scalability using static partitioning: each stream is broken into a fixed set of partitions where the producers write the records according to a partitioning strategy, whereas only one consumer is allowed to access each partition. This eliminates the complexity of dealing with fine-grain synchronization at the expense of costly over-provisioning (i.e., by allocating a large number of partitions that are not needed in the normal case to cover the worst case when the stream is used by a high number of consumers). Furthermore, each stream record is associated at append time with an offset that enables efficient random access. However, in a typical streaming scenario, random access is not needed as the records are processed in sequential order. Therefore, associating an offset for each single record introduces significant performance and space overhead. These design choices limit the ability of the ingestion phase to deliver high throughout and low latency in a scalable fashion.

We introduce KerA, a novel ingestion system for scalable stream processing that addresses the aforementioned limitations of the state of art. Specifically, it introduces a dynamic partitioning scheme that elastically adapts to the number of producers and consumers by grouping records into fixed-sized segments at fine granularity. Furthermore, it relies on a lightweight metadata management scheme that assigns minimal information to each segment rather than record, which greatly reduces the performance and space overhead of offset management, therefore optimizing sequential access to the records.

3.2 *Stream ingestion: background and state of the art*

A stream is a very large, unbounded collection of records, that can be produced and consumed in parallel by multiple producers and consumers. The records are typically buffered on multiple broker nodes, which are responsible to control the flow between the producers and consumers such as to enable high throughput, low latency, scalability and reliability (i.e., ensure records do not get lost due to failures). To achieve scalability, stream records are logically divided into many partitions, each managed by one broker.

Static partitioning

State-of-art stream ingestion systems (e.g., [Kafka], [Pulsar], [DLog]) employ a static partitioning scheme where the stream is split among a fixed number of partitions, each of which is an unbounded, ordered, immutable sequence of records that are continuously appended. Each broker is responsible for one or multiple partitions. Producers accumulate records in fixed-sized batches, each of which is appended to one partition. To reduce communication overhead, the producers group together multiple batches that correspond to the partitions of a single broker in a single request. Each consumer is assigned to one or more partitions. Each partition assigned to a single consumer. This eliminates the need for complex synchronization mechanisms but has an important drawback: the application needs a priori knowledge about the optimal number of partitions.

However, in real-life situations it is difficult to know the optimal number of partitions a priori, both because it depends on a large number of factors (number of brokers, number of consumers and producers, network size, estimated ingestion and processing throughput target, etc.). In addition, the producers and consumers can exhibit dynamic behavior that can generate large variance between the optimal number of partitions needed at different moments during the runtime. Therefore, users tend to over-provision the number of partitions to cover the worst case scenario where a large number of producers and consumers need to access the records simultaneously. However, if the worst-case scenario is not a norm but an exception, this can lead to significant unnecessary overhead. Furthermore, a fixed number of partitions can also become a source of imbalance: since each partition is assigned to a single consumer, it can happen that one partition accumulates or releases records faster than the other partitions if it is assigned to a consumer that is slower or faster than the other consumers.

For instance, in Kafka, a stream is created with a fixed number of partitions that are managed by Kafka's brokers, as depicted in Figure 2. Each partition is represented by an index file for offset positioning and a set of segment files, initially one, for holding stream records. Kafka leverages the operating system cache to serve partition's data to its clients. Due to this design it is not advised to collocate streaming applications on the same Kafka nodes, which does not allow to leverage data locality optimizations.

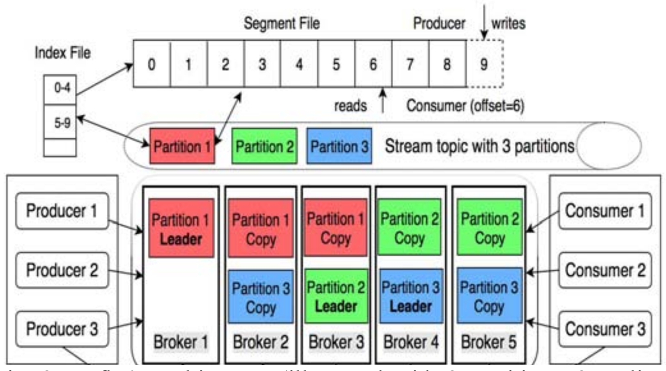

Fig. 2. Kafka's architecture (illustrated with 3 partitions, 3 replicas and 5 brokers.). Producers and consumers query Zookeeper for partition metadata (i.e., on which broker a stream partition leader is stored). Producers append to the partition's leader (e.g., broker 1 is assigned partition 1 leader), while exclusively one consumer pulls records from it starting at a given offset, initially 0. Records are appended to the last segment of a partition with an offset being associated to each record. Each partition has 2 other copies (i.e., partition's followers) assigned to other brokers that are responsible to pull data from the partition's leader in order to remain in sync.

Offset-based record access

The brokers assign to each record of a partition a monotonically increasing identifier called the partition offset, allowing applications to get random access within partitions by specifying the offset. The rationale of providing random access (despite the fact that streaming applications normally access the records in sequential order) is due to the fact that it enables failure recovery. Specifically, a consumer that failed can go back to a previous checkpoint and replay the records starting from the last offset at which its state was checkpointed. Furthermore, using offsets when accessing records enables the broker to remain stateless with respect to the consumers. However, support for efficient random access is not free: assigning an offset to each record at such fine granularity degrades the access performance and occupies more memory. Furthermore, since the records are requested in batches, each batch will be larger due to the offsets, which generates additional network overhead.

3.3 *Design principles for stream ingestion*

In order to address the issues detailed in the previous section, we introduce a set of design principles for efficient stream ingestion and scalable processing.

a) **Dynamic** partitioning using semantic grouping and sub-partitions: In a streaming application, users need to be able to control partitioning at the highest level in order to de-fine how records can be grouped together in a meaningful way. Therefore, it is not possible to eliminate partitioning altogether (e.g., by assigning individual records directly to consumers). However, we argue that users should not be concerned about performance issues when designing the partitioning strategy, but rather by the semantics of the grouping. Since state-of-art approaches assign a single producer and consumer to each partition, the users need to be aware of both semantics and performance issues when using static partitioning. Therefore, we propose a dynamic partitioning scheme where users fix the high level partitioning criteria from the semantic perspective, while the ingestion system is responsible to make each parti-tion elastic by allowing multiple producers and consumers to access it simultaneously. To this end, we propose to split each partition into sub-partitions, each of which is independently managed and attached to a potentially different producer and consumer.

b) **Lightweight offset indexing optimized for sequential record access**: Since random access to the records is not the norm but an exception, we argue that ingestion systems should primarily optimize sequential access to records at the expense of random access. To this end, we propose a lightweight offset indexing that assigns offsets at coarse granularity at sub-partition level rather than fine granularity at record level. Additionally, this offset keeps track (on client side) of the last accessed record's physical position within the sub-partition, which enables the consumer to ask for the next records. Moreover, random access can be easily achieved when needed by finding the sub-partition that covers the offset of the record and then seeking into the sub-partition forward or backward as needed.

3.4 *KerA: overview*

In this section we introduce KerA, a prototype stream ingestion system that illustrates the design principles introduced in the previous section.

Partitioning model

KerA implements dynamic partitioning based on the concept of streamlet, which corresponds to the semantic high-level partition that groups records together. Each stream is therefore composed of a fixed number of streamlets. In turn, each streamlet is split into groups, which correspond to the sub-partitions assigned to a single producer and consumer. A streamlet can have an arbitrary number of groups, each of which can grow up to a maximum predefined size. To facilitate the management of groups and offsets in an efficient fashion, each group is further split into fixed-sized segments. The maximum size of a group is a multiple of segment size P 1. To control the level of parallelism allowed on each broker, only Q 1 groups can be active at a given moment. Elasticity is achieved by assigning an initial number of brokers

N 1 to hold the streamlets M, M N. As more producers and consumers access the streamlets, more brokers can be added up to M.

In order to ensure ordering semantics, each streamlet dynamically creates groups and segments that have unique, monotonically increasing identifiers. Brokers expose this information through RPCs to consumers that create an application offset defined as [streamId, streamletId, groupId, segmentId, position] based on which they issue RPCs to pull data. The position is the physical offset at which a record can be found in a segment. The consumer initializes it to 0 (broker understands to iterate to first record available in that segment) and the broker responds with the last record position for each new request, so the consumer can update its latest offset to start a future request with. Using this dynamic approach (as opposed to the static approach used by explicit offsets per partition, clients have to query brokers to discover groups), we implement lightweight offset indexing optimized for sequential record access.

Stream records are appended in order to the segments of a group, without associating an offset, which reduces the storage and processing overhead. Each consumer exclusively processes one group of segments. Once the segments of a group are filled (the number of segments per group is configurable), a new one is created and the old group is closed (i.e., no longer enables appends). A group can also be closed after a timeout if it was not appended in this time.

Favoring parallelism: consumer and producer protocols

Producers only need to know about streamlets when inter-acting with KerA. The input batch is always ingested to the ac-tive group computed deterministically on brokers based on the producer identifier and parameter Q of given streamlet (each producer request has a header with the producer identifier with each batch tagged with the streamlet id). Producers writing to the same streamlet synchronize using a lock on the streamlet in order to obtain the active group corresponding to the Qth entry based on their producer identifier. The lock is then released and a second-level lock is used to synchronize producers accessing the same active group. Thus, two producers appending to the same streamlet, but different groups, may proceed in parallel for data ingestion. In contrast, in Kafka producers writing to the same partition block each other, with no opportunity for parallelism.

Consumers issue RPCs to brokers in order to first discover streamlets' new groups and their segments. Only after the application offset is defined, consumers can issue RPCs to pull data from a group's segments. Initially each consumer is associated (non-exclusively) to one or many streamlets from which to pull data from. Consumers process groups of a streamlet in the order of their identifiers, pulling data from segments also in the order of their respective identifiers. Brokers maintain for each streamlet the last group given to consumers identified by their consumer group id (i.e., each consumer request header contains a unique application id). A group is configured with a fixed number of segments to allow fine-grained consumption with many consumers per streamlet in order to better load balance groups to consumers. As such, each consumer has a fair access chance since the group is limited in size by the segment size and the number of segments. This approach also favors parallelism. Indeed, in KerA a consumer pulls data from one group of a streamlet exclusively, which means that multiple consumers can read in parallel from different groups of the same streamlet. In Kafka, a consumer pulls data from one partition exclusively.

Architecture and implementation

KerA's architecture is similar to Kafka's (Figure 3): a single layer of brokers (nodes) serve producers and consumers. However, in KerA brokers are used to discover stream partitions. Kera builds atop RAMCloud [RC15] to leverage its network abstraction that enables the use of other network transports (e.g., UDP, DPDK, Infiniband), whereas Kafka only supports TCP. Moreover, it allows KerA to benefit from a set of design choices like polling and request dispatching [RC17] that help boost performance (kernel bypass and zero-copy networking are possible with DPDK and Infiniband).

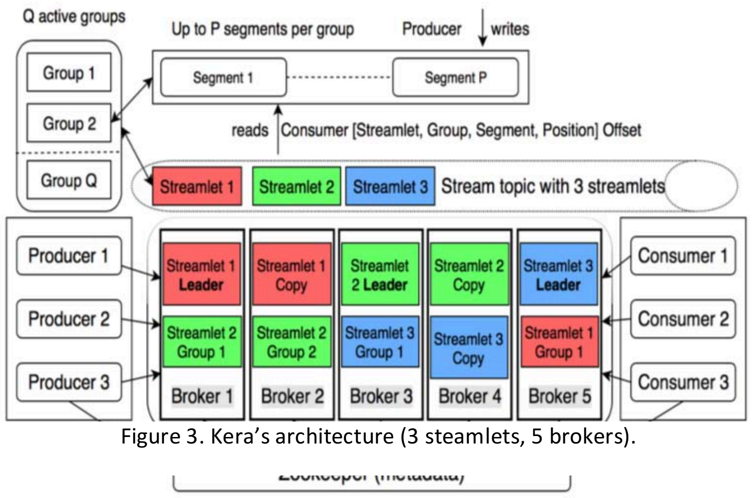

Figure 3. Kera's architecture (3 steamlets, 5 brokers).

Each broker has an ingestion component offering pub/sub interfaces to stream clients and an optional backup component that can store stream replicas. This allows for separation of nodes serving clients from nodes serving as backups. Another important difference compared to Kafka is that brokers directly manage stream data instead of leveraging the kernel virtual cache. KerA's segments are buffers of data controlled by the stream storage. Since each segment contains the stream, streamlet, group] metadata, a streamlet's groups can be durably stored independently on multiple disks, while in Kafka a partition's segments are stored on a single disk.

To support durability and replication, and implement fast crash recovery techniques, it is possible to rely on RAM-Cloud [RC11], by leveraging the aggregated disk bandwidth in order to recover the data of a lost node in seconds. KerA's fine-grained partitioning model favors this recovery technique. However it cannot be used as such: producers should continuously append records and not suffer from broker crashes, while consumers should not have to wait for all data to be recovered (thus incurring high latencies). Instead, recovery can be achieved by leveraging consumers' application offsets. We plan to enable such support as future work.

3.5 *Experimental evaluation*

We evaluate KerA compared to Kafka using a set of synthetic benchmarks to assess how partitioning and (application defined) offset based access models impact performance.

A. Setup and parameter configuration

We ran all our experiments on Grid5000 Grisou cluster [G5K]. Each node has 16 cores and 128 GB of memory. In each experiment the source thread of each producer creates 50 million non-keyed records of 100 bytes, and partitions them round-robin in batches of configurable size. The source waits no more than 1ms (parameter named linger.ms in Kafka) for a batch to be filled, after this timeout the batch is sent to the broker. Another producer thread groups batches in requests and sends them to the node responsible of the request's partitions (multi TCP synchronous requests). Similarly, each consumer pulls batches of records with one thread and simply iterates over records on another thread.

In the client's main thread we measure ingestion and pro-cessing throughput and log it after each second. Producers and consumers run on different nodes. We plot average ingestion throughput per client (producers are represented with KeraProd and KafkaProd, respectively consumers with KeraCons and KafkaCons), with 50 and 95 percentiles computed over all clients measurements taken when concurrently running all producers and consumers (without considering the first and last ten seconds measurements of each client).

Each broker is configured with 16 network threads that corresponds to the number of cores of a node and holds one copy of the streamlet's groups (we plan to study pull-based versus push-based replication impact in future work). In each experiment we run an equal number of producers and consumers. The number of partitions/streamlets is configured to be a multiple of the number of clients, at least one for each client. Unless specified, we configure in KerA the number of active groups to 1 and the number of segments to 16. A request is characterized by its size (i.e., request.size, in bytes) and contains a set of batches, one for each partition, each batch having a batch.size in bytes. We use Kafka 0.10.2.1 since it has a similar data model with KerA (newest release introduces batch entries for exactly once processing, a feature that could be efficiently enabled also in KerA [Lee15]). A Kafka segment is 512 MB, while in KerA it is 8MB. This means that rolling to a new segment happens more often and may impact performance (since KerA's clients need to discover new segments before pulling data from them).

B. Results

While Kafka provides a static offset-based access by maintaining and indexing record offsets, KerA proposes dynamic access through application defined offsets that leverage streamlet-group-segment metadata (thus, avoiding the over-head of offset indexing on brokers). In order to understand the application offset overhead in Kafka and KerA, we evaluate different scenarios, as follows.

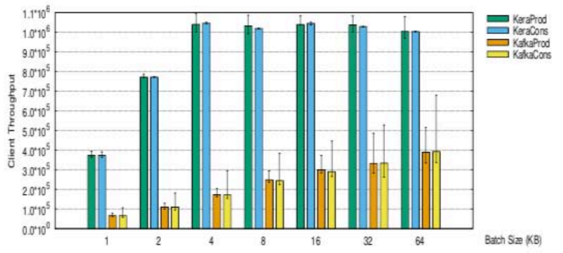

Fig. 4. Increasing the batch size (request size). Parameters: 4 brokers; 16 producers and 16 consumers; number of partitions/streamlets is 16; *request.size* equals *batch.size* multiplied by 4 (number of partitions per node). On X we have producer *batch.size* KB, for consumers we configure a value 16x higher.

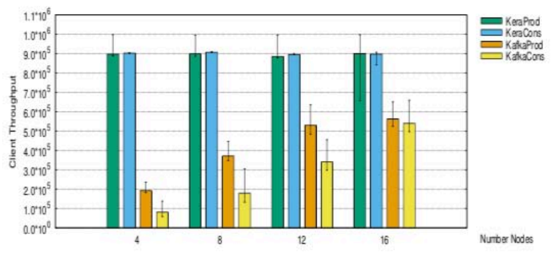

Fig. 6. Adding nodes (brokers). Parameters: 32 producers and 32 consumers; 256 partitions, 32 streamlets with 8 active groups per streamlet; *batch.size* = 16KB; *request.size* = *batch.size* multiplied by the number of partitions/active groups per node.

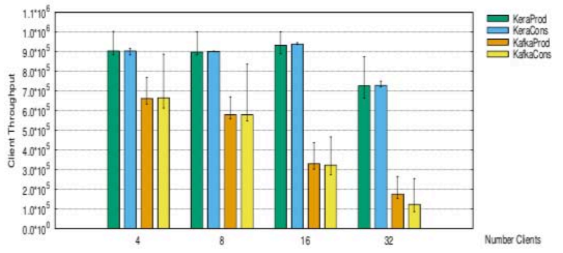

Fig. 5. Adding clients. Parameters: 4 brokers; 32 partitions/streamlets, 1 active group per streamlet; *batch.size* = 16KB; *request.size* = 128KB.

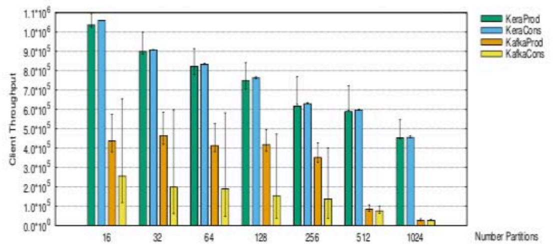

Fig. 7. Increasing the number of partitions and respectively streamlets. Parameters: 4 brokers; 16 producers and 16 consumers; *request.size* = 1MB; *batch.size* equals *request.size* divided by the number of partitions.

Impact of the batch/request size. By increasing the batch size we observe smaller gains in Kafka than KerA (Figure 4). KerA provides up to 5x higher throughput when increasing the batch size from 1KB to 4KB, after which throughput is limited by that of the producer's source. For each producer request, be-fore appending a batch to a partition, Kafka iterates at runtime over batch's records in order to update their offset, while Kera simply appends the batch to the group's segment. To build the application offset, KerA's consumers query brokers (issuing RPCs that compete with writes and reads) in order to discover new groups and their segments. This could be optimized by implementing a smarter read request that discovers new groups or segments automatically, reducing the number of RPCs.

Adding clients (vertical scalability). Having more concur-rent clients (producers and consumers) means possibly reduced throughput due to more competition on partitions and less worker threads available to process the requests. As presented in Figure 5, when running up to 64 clients on 4 brokers (full parallelism), KerA is more efficient in front of higher number of clients due to its more efficient application offset indexing.

Adding nodes (horizontal scalability). Since clients can leverage multi-TCP, distributing partitions on more nodes helps increasing throughput. As presented in Figure 6, even when Kafka uses 4 times more nodes, it only delivers half of the performance of KerA. Current KerA implementation prepares a set of requests from available batches (those that are filled or those with the timeout expired) and then submits them to brokers, polling them for answers. Only after all requests are executed, a new set of requests is built. This implementation can be further optimized and the network client can be asynchronously decoupled, like in Kafka, in order to allow for submissions of new requests when older ones are processed.

Increasing the number of partitions/streamlets. Finally, we seek to assess the impact of increasing the number of partitions on the ingestion throughput. When the number of partitions is increased we also reduce the batch.size while keeping the request.size fixed in order to maintain the target maximum latency an application needs. We configure KerA similarly to Kafka: the number of active groups is 1 so the number of streamlets gives a number of active groups equal to the number of partitions in Kafka (one active group for each streamlet to pull data from in each consumer request). We observe in Figure 7 that when increasing the number of partitions the average throughput per client decreases. We suspect Kafka's drop in performance (20x less than KerA for 1024 partitions) is due to its offset-based implementation, having to manage one index file for each partition.

With KerA one can leverage the streamlet-group abstractions in order to provide applications an unlimited number of partitions (fixed size groups of segments). To show this benefit, we run an additional experiment with KerA configured with 64 streamlets and 16 active groups. The achieved throughput is almost 850K records per second per client providing consumers 1024 active groups (fixed-size partitions) compared to less than 50K records per second with Kafka providing the same number of partitions. The streamlet configuration allows the user to reason about the maximum number of nodes on which to partition a stream, each streamlet providing an unbounded number of fixed-size groups (partitions) to process. KerA provides higher parallelism to producers resulting in higher ingestion/processing client throughput than Kafka.

3.6 *Summary*

We introduced KerA, a novel data ingestion system for Big Data stream processing specifically designed to deliver high throughput, low latency and to elastically scale to a large number of producers and consumers. The core ideas proposed by KerA revolve around: (1) dynamic partitioning based on semantic grouping and sub-partitioning, which enables more flexible and elastic management of partitions; (2) lightweight offset indexing optimized for sequential record access using streamlet metadata exposed by the broker. We illustrate how KerA implements these core ideas through a research proto-type. Based on extensive experimental evaluations, we show that KerA outperforms Kafka up to 4x for ingestion throughput and up to 5x for the overall stream processing throughput. Furthermore, we have shown KerA is capable of delivering data fast enough to saturate a Big Data stream processing engine acting as the consumer. Encouraged by these initial results, we plan to integrate KerA with streaming engines and to explore in future work several topics: data locality optimizations through shared buffers, durability as well as state management features for streaming applications.

4 Tailwind: Fast and Atomic RDMA-based Replication in Support of In-Memory Big Data Analytics

4.1 *Context and problem*

In-memory key-value stores are an essential building block for large-scale Big Data analytics applications [Dynamo,Nis13]. Recent research has led to in-memory key-value stores that can perform millions of operations per second per machine with a few microseconds remote access times. Harvesting CPU power and eliminating conventional network overheads has been key to these gains. How-ever, like many other systems, they must replicate data in order to survive failures.

As the core frequency scaling and multi-core architecture scaling are both slowing down, it becomes critical to reduce replication overheads to keep-up with shift-ing application workloads in key-value stores [Li17]. We show that replication can consume up to 80% of the CPU cycles for write-intensive workloads, in strongly-consistent in-memory key-value stores. Techniques like remote-direct memory access (RDMA) are promising to improve overall CPU efficiency of replication and keep predictable tail latencies.

Existing RDMA-based approaches use message-passing interfaces: a sender remotely places a message into a receiver's DRAM; a receiver must actively poll and handle new RDMA messages. This approach guar-antees the atomicity of RDMA transfers, since only fully received messages are applied by the receiver [Farm,Kal14,Su17]. However, this approach defeats RDMA efficiency goals since it forces receivers to use their CPU to handle in-coming RDMA messages and it incurs additional mem-ory copies.

The main challenge of efficiently using RDMA for replication is that failures could result in partially ap-plied writes. The reason is that receivers are not aware of data being written to their DRAM. Leaving receivers idle is challenging because there is no protocol to guarantee data consistency in the event of failures.

A second key limitation with RDMA is its low scal-ability. This limitation comes from the connection-oriented nature of RDMA transfers. Senders and re-ceivers have to setup queue pairs (QP) to perform RDMA. Lots of recent work has observed the high cost of NIC connection cache misses [Farm,Kal16,Tsa17]. Scalability is limited as it typically depends on the cluster size.

To address the above challenges, we developed Tailwind, a zero-copy primary-backup log replication proto-col that completely bypasses CPUs on all target backup servers. In Tailwind, log records are transferred directly from the source server's DRAM to I/O buffers at tar-get servers via RDMA writes. Backup servers are com-pletely passive during replication, saving their CPUs for other purposes; they flush these buffers to solid-state drives (SSD) periodically when the source triggers it via remote procedure call (RPC) or when power is inter-rupted. Even though backups are idle during replication, Tailwind is strongly consistent: it has a protocol that allows backups to detect incomplete RDMA transfers.

Tailwind uses RDMA write operations for all data movement, but all control operations such as buffer allocation and setup, server failure notifications, buffer flushing and freeing are all handled through conventional RPCs. This simplifies such complex operations without slowing down data movement. In our implementation, RPCs only account for 10 ^{5} of the replication requests. This also makes Tailwind easier to use in systems that use log replication over distributed blocks even if they were not designed to exploit RDMA.

Since Tailwind needs only to maintain connections between a primary server and its backups, the number of connections scales with the size of a replica group, not with the cluster size, making Tailwind a scalable approach.

We implemented and evaluated Tailwind on RAM- Cloud, a scale-out in-memory key-value store that exploits fast kernel-bypass networking. Tailwind is suited to RAMCloud's focus on strong consistency and low latency. Tailwind significantly improves RAMCloud's throughput since each PUT operation in the cluster re- sults in three remote replication operation that would oth- erwise consume server CPU resources.

Tailwind improves RAMCloud's throughput by 1.7× on the YCSB benchmark, and it reduces durable PUT median latency from 32 µs to 16 µs and 99th percentile latency from 78 µs to 28 µs. Theses results stem from the fact that Tailwind significantly reduces the CPU cycles used by the replication operations: Tailwind only needs 1/3 of the cores RAMCloud uses to achieve the same throughput.

This work makes four key contributions.

• it analyzes and quantifies CPU related limitations in modern in-memory key-value stores;

• it presents Tailwind's design, it describes its imple-mentation in the RAMCloud distributed in-memory key-value store, and it evaluates its impact on RAM-Cloud's normal-case and recovery performance;

• to our knowledge, Tailwind is the first log repli-cation protocol that eliminates all superfluous data copying between the primary replica and its back-ups, and it is the first log replication protocol that leaves servers CPU idle while serving as replication targets; this allows servers to focus more resources on normal-case request processing;

• Tailwind separates the replication data path and control path and optimizes them individually; it uses RDMA for heavy transfer, but it retains the simplicity of RPC for rare operations that must deal with complex semantics like failure handling and resource exhaustion.

4.2 *The Promise of RDMA and Challenges*

Replication and redundancy are fundamental to fault tolerance, but at the same time they are costly. Primary-backup replication (PBR) is popular in fault-tolerant storage systems like file systems and key-value stores, since it tolerates f stop-failures with f + 1 replicas. Note that, we refer to a primary replica server as primary, and secondary replica server as secondary or backup. In some systems, backup servers don't process user-facing requests, but in many systems each node acts as both a primary for some data items and as a backup for other data items. In some systems this is implicit: for example, a key-value store may store its state on HDFS [Hdfs] and a single physical machine might run both a key-value store frontend and an HDFS chunkserver.

Replication is expensive for three reasons. First, it is inherently redundant and, hence, brings overhead: the act of replication itself requires moving data over the network. Second, replication in strongly consistent systems is usually synchronous, so a primary must stall while holding resources while waiting for acknowledgements from backups (often spin-ning a CPU core in low-latency stores). Third, in systems, where servers (either explicitly or implicitly) serve both client-facing requests and replication operations, those operations contend.

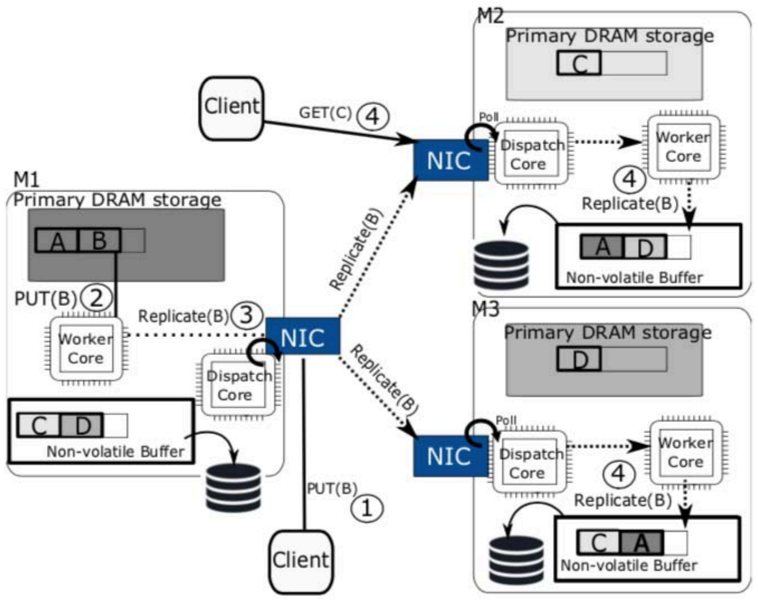

Figure 8. Flow of primary-backup replication

Figure 8 shows this in more detail. Low-latency, high-throughput stores use kernel-bypass to directly poll NIC control rings (with a dispatch core) to avoid kernel code paths and interrupt latency and throughput costs. Even so, a CPU on a primary node processing an update operation must receive the request, hand the request off to a core (worker core) to be processed, send remote messages, and then wait for multiple nodes acting as backup to process these requests. Batching can improve the number of backup request messages each server must receive, but at the cost of increased latency. Inherently, though, replication can double, triple, or quadruple the number of messages and the amount of data generated by client-issued write requests. It also causes expensive stalls at the primary while it waits for responses. In these systems, responses take a few microseconds which is too short a time for the primary to context switch to another thread, yet its long enough that the worker core spends a large fraction of its time waiting.

RDMA Opportunities

One-sided RDMA operations are attractive for replication; replication inherently requires expensive, redundant data movement. Backups are (mostly) passive; they often act as dumb storage, so they may not need CPU involvement. RAMCloud, an in-memory low-latency kernel-bypass-based key-value store, is often bottlenecked on CPU. For read-heavy workloads, the cost of polling net-work and dispatching requests to idle worker cores dominates. Because of that, worker cores cannot be fully utilized. One-sided operations for replicating PUT operations would reduce the number of requests each server handles in RAMCloud, which would indirectly but significantly improve read throughput. For workloads with a significant fraction of writes or where a large amount of data is transferred, write throughput can be improved, since remote CPUs needn't copy data between NIC receive buffers and I/O or non-volatile storage buffers.

Challenges

The key challenge in using one-sided RDMA operations is that they have simple semantics which offer little control on the remote side. This is by design; the remote NIC executes RDMA operations directly, so they lack the generality that a conventional CPU-based RPC handlers would have. A host can issue a remote read of a single, sequential region of the re-mote processes virtual address space (the region to read must be registered first, but a process could register its whole virtual address space). Or, a host can issue a remote write of a sin-gle, sequential region of the remote processes virtual address space (again, the region must be registered with the NIC). NICs support a few more complex operations (compare-and-swap, atomic add), but these operations are currently much slower than issuing an equiva-lent two-sided operation that is serviced by the remote CPU [Su17, Kal16]. These simple, restricted semantics make RDMA operations efficient, but they also make them hard to use safely and correctly. Some existing systems use one-sided RDMA operations for replication (and some also even use them for normal case operations [Farm]).

However, no existing primary-backup replication scheme reaps the full benefits of one-sided operations. In existing approaches, source nodes send replication operations using RDMA writes to push data into ring buffers. CPUs at backups poll for these operations and apply them to replicas. In practice, this is is effectively emulating two-sided operations [Farm]. RDMA reads don't work well for replication, because they would require backup CPUs to schedule operations and "pull" data, and primaries wouldn't immediately know when data was safely replicated.

Two key, interrelated issues make it hard to use RDMA writes for replication that fully avoids the remote CPUs at backups. First, a primary can crash when replicating data to a backup. Because RDMA writes (inherently) don't buffer all of the data to be written to remote memory, it is possible that an RDMA write could be partially applied when the pri-mary crashes. If a primary crashes while updating state on the backup, the backup's replica wouldn't be in the "before" or "after" state, which could result in a corrupted replica. Worse, since the primary was likely mutating all replicas concurrently, it is possible for all replicas to be corrupted. Interestingly, backup crashes during RDMA writes don't create new chal-lenges for replication, since protocols must deal with that case with conventional two-sided operations too. Well-known techniques like log-structured backups or shadow paging can be used to prevent update-in-place and loss of atomicity. Traditional log implementations enforce a total ordering of log entries. In database systems, for instance, the order is used to recreate a consistent state during recovery.

Unfortunately, a second key issue with RDMA operations makes this hard: each opera-tion can only affect a single, contiguous region of remote memory. To be efficient, one-sided writes must replicate data in its final, stable form, otherwise backup CPU must be involved, which defeats the purpose. For stable storage, this generally requires some metadata. For example, when a backup uses data found in memory or storage it must know which por-tions of memory contain valid objects, and it must be able to verify that the objects and the markers that delineate them haven't been corrupted. As a result, backups need some meta-data about the objects that they host in addition to the data items themselves. However, RDMA writes make this hard. Metadata must inherently be intermixed with data objects, since RDMA writes are contiguous. Otherwise, multiple round trips would be needed, again defeating the efficiency gains.

4.3 *Tailwind*

Tailwind is a strongly-consistent RDMA-based replication protocol. It was designed to meet four requirements:

Zero-copy, Zero-CPU on Backups for Data Path. In order to relieve backups CPUs from processing replication requests, Tailwind relies on one-sided RDMA writes for all data movement. In addition, it is zero-copy at primary and secondary replicas; the sender uses kernel-bypass and scatter/gather DMA for data transfer; on the backup side, data is directly placed to its final storage location via DMA transfer without CPU involvement.

Strong Consistency. For every object write Tailwind synchronously waits for its replication on all backups before notifying the client. Although RDMA writes are one-sided, reliable-connected QPs generate a work completion to notify the sender once a message has been correctly sent and acknowledged by the receiver NIC (i.e. written to remote memory). One-sided operation raise many issues, Tailwind is designed to cover all corner cases that may challenge correctness.

Overhead-free Fault-Tolerance. Backups are unaware of replication as it happens, which can be unsafe in case of failures. To address this, Tailwind appends a piece of metadata in the log after every object update. Backups use this metadata to check integrity and locate valid objects during recovery. Although a few backups have to do little extra work during crash recovery, that work has no impact on recovery performance.

Preserves Client-facing RPC Interface. Tailwind has no requirement on the client side; all logic is implemented between primaries and backups. Clients observe the same consistency guarantees. However, for write operations, Tailwind highly improves end-to-end latency and throughput from the client perspective.

The Metadata Challenge

Metadata is crucial for backups to be able to use replicated data. For instance, a backup needs to know which portions of the log contain valid data. In RPC-based systems, meta-data is usually piggybacked within a replication request [RC15, Kal16]. However, it is challenging to update both data and metadata with a single RDMA write since it can only affect a con-tiguous memory region. In this case, updating both data and metadata would require send-ing two messages which would nullify one-sided RDMA benefits. Moreover, this is risky: in case of failures a primary may start updating the metadata and fail before finishing, thereby invalidating all replicated objects.

For log-structured data, backups need two pieces of information: (1) the offset through which data in the buffer is valid. This is needed to guarantee the atomicity of each update. An outdated offset may lead the backup to use old and inconsistent data during crash recov-ery. (2) A checksum used to check the integrity of the length fields of each log record during recovery. Checksums are critical for ensuring log entry headers are not corrupted while in buffers or on storage. These checksums ensure iterating over the buffer is safe; that is, a corrupted length field does not “point” into the middle of another object, out of buffer, or indicate an early end to the buffer.

The protocol assumes that each object has a header next to it [Farm, Kafka]. Implementation-agnostic information in headers should include: (1) the size of the object next to it to allow log traversal; (2) an integrity check that ensures the integrity of the contents of the log entry.

Tailwind checksums are 32-bit CRCs computed over log entry headers. The last check-sum in the buffer covers all previous headers in the buffer. For maximum protection, check-sums are end-to-end: they should cover the data while it is in transit over the network and while it occupies storage.

To be able to perform atomic updates with one-sided RDMAs in backups, the last check-sum and the current offset in the buffer must be present and consistent in the backup after every update. A simple solution is to append the checksum and the offset before or after every object update. A single RDMA write would suffice then for both data and metadata. The checksum must necessarily be sent to the backup. Interestingly, this is not the case for the offset. The nature of log-structured data and the properties of one-sided RDMA make it possible, with careful design, for the backup to compute this value at recovery time without hurting consistency. This is possible because RDMA writes are performed (at the receiver side) in an increasing address order. In addition, reliable-connected QPs ensure that updates are applied in the order they were sent.

Based on these observations, Tailwind appends a checksum in the log after every object update; at any point of time a checksum guarantees, with high probability, the integrity of all previous headers preceding it in the buffer. During failure-free time, a backup is ensured to always have the latest checksum at the end of the log. On the other hand, backups have to compute the offset themselves during crash recovery.

4.4 *Evaluation*

We implemented Tailwind on RAMCloud a low-latency in-memory key-value store. Tail-wind’s design perfectly suits RAMCloud in many aspects:

Low latency. RAMCloud's main objective is to provide low-latency access to data. It relies on fast networking and kernel-bypass to provide a fast RPC layer. Tailwind can further improve RAMCloud (PUT) latency by employing one-sided RDMA without any additional complexity or resource usage.

Replication and Threading in RAMCloud. To achieve low latency, RAMCloud dedicates one core solely to poll network requests and dispatch them to worker cores (Figure 8). Worker cores execute all client and system tasks. They are never preempted to avoid con-text switches that may hurt latency. To provide strong consistency, RAMCloud always re-quests acknowledgements from all backups for every update. With the above threading-model, replication considerably slows down the overall performance of RAMCloud as we have showed in the last chapter. Hence Tailwind can greatly improve RAMCloud's CPU-efficiency and remove replication overheads.

Log-structured Memory. RAMCloud organizes its memory as an append-only log. Mem-ory is sliced into smaller chunks called segments that also act as the unit of replication, i.e., for every segment a primary has to choose a backup. Such an abstraction makes it easy to re-place RAMCloud's replication system with Tailwind. Tailwind checksums can be appended in the log-storage, with data, and replicated with minimal changes to the code. In addition, RAMCloud provides a log-cleaning mechanism which can efficiently clean old checksums and reclaim their storage space.

We compared Tailwind with RAMCloud replication protocol, focusing our analysis on three key questions :

Does Tailwind improve performance? Measurements show Tailwind reduces RAM-Cloud's median write latency by 2x and 99th percentile latency by 3x (Figure 10). Tailwind improves throughput by 70% for write-heavy workloads and by 27% for workloads that include just a small fraction of writes.

Why does Tailwind improve performance? Tailwind improves per-server throughput by eliminating backup request processing (Figure 11), which allows servers to focus effort on processing user-facing requests.

What is the overhead of Tailwind? We show that Tailwind's performance improvement comes at no cost. Specifically, we measure and find no overhead during crash recovery compared to RAMCloud.

Experimental Setup

Experiments were done on a 35 server Dell r320 cluster on the CloudLab [Clab] testbed.

We used three YCSB [ycsb] workloads to evaluate Tailwind: update-heavy (50% PUTs, 50% GETs), read-heavy (5% PUTs, 95% GETs), and update-only (100% PUTs). We intitially inserted 20 million objects of 100 B plus 30 B for the key. Afterwards, we ran up to 30 client machines. Clients generated requests according to a Zipfian distribution (q = 0.99). Objects were uniformly inserted in active servers. The replication factor was set to 3 and RDMA buffers size was set to 8 MB. Every data point in the experiments is averaged over 3 runs.

RAMCloud's RPC-based replication protocol served as a baseline for comparison. Note that, in the comparison with Tailwind, we refer to RAMCloud's replication protocol as RAM-Cloud for simplicity.

Performance Improvement

The primary goal of Tailwind is to accelerate basic operations' throughput and latency. To demonstrate how Tailwind improves performance we show Figure 9, i.e. throughput per server as we increase the number of clients. When client operations consist of 5% PUTs and 95% GETs, RAMCloud achieves 500 KOp/s per server while Tailwind reaches up to 635 KOp/s. Increasing the update load enables Tailwind to further improve the throughput. For instance with 50% PUTs Tailwind sustains 340 KOp/s against 200 KOp/s for RAM-Cloud, which is a 70% improvement. With update-only workload, improvement is not fur-ther increased: in this case Tailwind improves the throughput by 65%.

Tailwind can improve the number of read operations serviced by accelerating updates. CPU cycles saved allow servers (that are backups as well) to service more requests in general.

Figure 10 shows that update latency is also considerably improved by Tailwind. Under light load Tailwind reduces median and 99th percentile latency of an update from 16 µs to 11.8 µs and from 27 µs to 14 µs respectively. Under heavy load, i.e. 500 KOp/s Tailwind reduces median latency from 32 µs to 16 µs compared to RAMCloud. Under the same load tail latency is even further reduced from 78 µs to 28 µs.

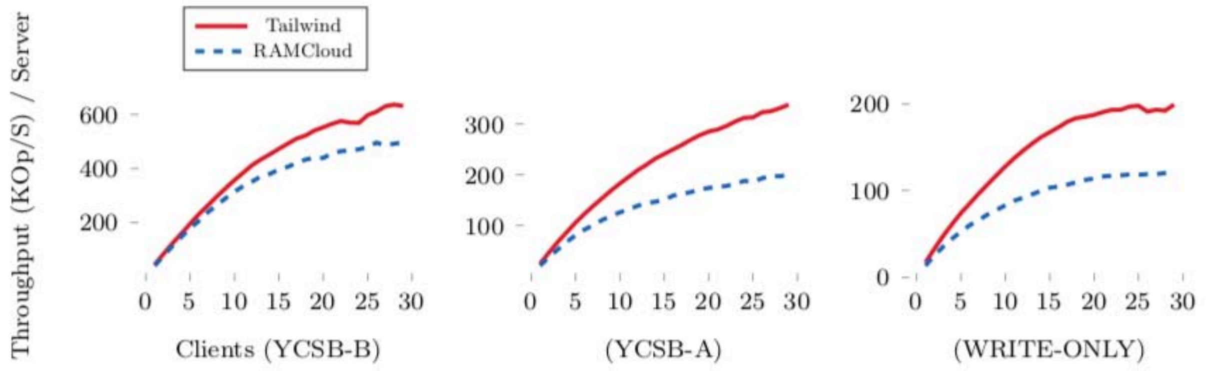

Figure 9. Throughput per server in a 4-server cluster

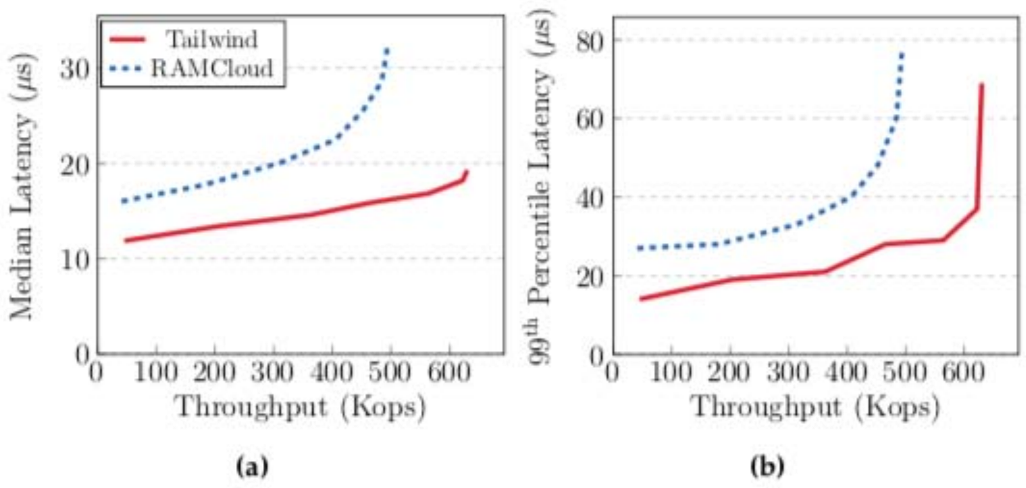

Figure 10. (a) Median latency and (b) 99th percentile latency of PUT operations when varying the load.

Tailwind can effectively reduce end-to-end client latency. With reduced acknowledge-ments waiting time, and more CPU power to process requests faster, servers can sustain a very low latency even under heavy concurrent accesses.

Gains as Backup Load Varies

Since all servers in RAMCloud act as both backups and primaries, Tailwind accelerates normal-case request processing indirectly by eliminating the need for servers to actively process replication operations. Figure 11 shows the impact of this effect. In each trial load is directed at a subset of four RAMCloud storage nodes; "Active Primary Servers" indicates the number of storage nodes that process client requests. Nodes do not replicate data to themselves, so when only one primary is active it is receiving no backup operations. All of the active primary's backup operations are directed to the other three otherwise idle nodes. Note that, in this figure, throughput is per-active-primaries. So, as more primaries are added, the aggregate cluster throughput increases.

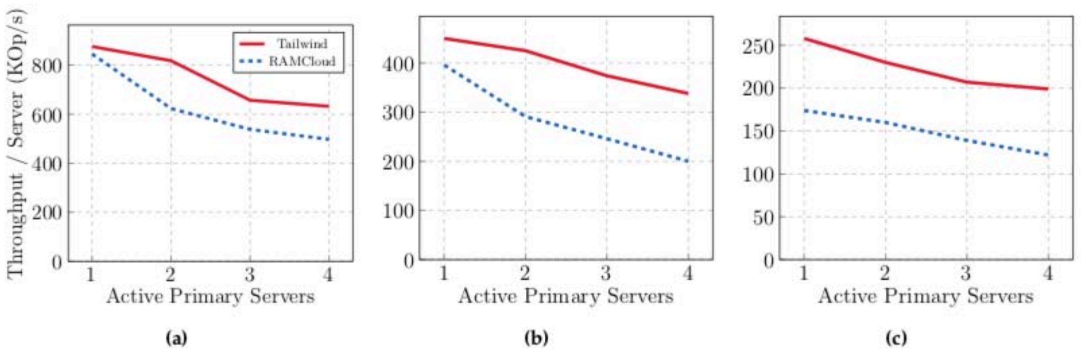

Figure 11. Throughput per active primary servers when running (a) YCSB-B (b) YCSB-A (c) WRITE-ONLY with 30 clients.

As client GET/PUT operations are directed to more nodes (more active primaries), each node slows down because it must serve a mix of client operations intermixed with an in-creasing number of incoming backup operations. Enough client load is offered (30 clients) so that storage nodes are the bottleneck at all points in the graphs. With four active pri-maries, every server node is saturated processing client requests and backup operations for all client-issued writes.

Even when only 5% of client-issued operations are writes (Figure 11a), Tailwind in-creasingly improves performance as backup load on nodes increases. When a primary doesn't perform backup operations Tailwind improves throughput 3%, but that increases to 27% when the primary services its fair share of backup operations. The situation is sim-ilar when client operations are a 50/50 mix of reads and writes (Figure 11b) and when clients only issue writes (Figure 11c).

As expected, Tailwind enables increasing gains over RAMCloud with increasing load, since RDMA eliminates three RPCs that each server must handle for each client-issued write, which, in turn, eliminates worker core stalls on the node handling the write.

In short, the ability of Tailwind to eliminate replication work on backups translates into more availability for normal request processing, and, hence, better GET/PUT performance.

Resource Utilization

The improvements above have shown how Tailwind can improve RAMCloud's baseline replication normal-case. The main reason is that operations contend with backup operations for worker cores to process them. Figure 11a illustrates this: we vary the offered load (updates-only) to a 4-server cluster and report aggregated active worker cores. For example, to service 450 KOp/s, Tailwind uses 5.7 worker cores while RAMCloud requires 17.6 active cores, that is 3x more resources. For the same scenario, we also show Figure 11b that shows the aggregate active dispatch cores. Interestingly, gains are higher for dispatch, e.g., to achieve 450 KOp/s, Tailwind needs only 1/4 of dispatch cores used by RAMCloud.

Both observations confirm that, for updates, most of the resources are spent in processing replication requests. To get a better view on the impact when GET/PUT operations are mixed, we show Figure 12.

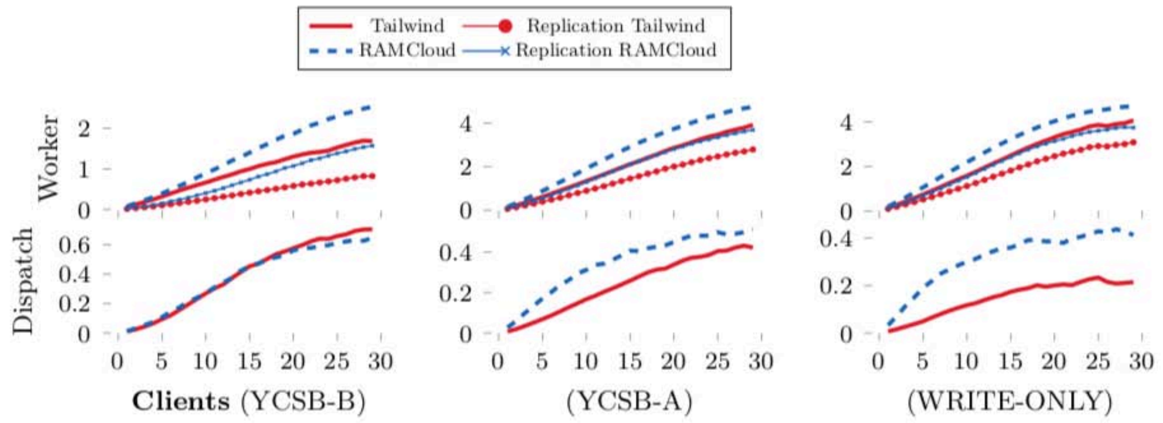

Figure 12. Total dispatch and worker cores utilization per server in a 4-server cluster. "Replication" in worker graphs represent the fraction of worker load spent on processing replication requests on primary servers

It represents active worker and dispatch cores, respectively, when varying clients. When requests consist of updates only, Tailwind reduces worker cores utilization by 15% and dispatch core utilization by 50%. This is stems from the fact that a large fraction of dispatch load is due to replication requests in this case. With 50/50 reads and writes, worker utilization is slightly improved to 20% while it reaches 50% when the workload consists of 5% writes only.

Interestingly, dispatch utilization is not reduced when reducing the proportion of writes. With 5% writes, Tailwind utilizes even more dispatch than RAMCloud. This is actually a good sign, since read workloads are dispatch-bound. Therefore, Tailwind allows RAMCloud to process even more reads by accelerating write operations. This is implicitly shown in Figure 12 with "Replication" graphs that represent worker utilization due to waiting for replication requests. For update-only workloads, RAMCloud spends 80% of the worker cycles in replication. With 5% writes, RAMCloud spends 62% of worker cycles waiting for replication requests to complete against 49% with Tailwind. The worker load difference is spent on servicing read requests.

Scaling with Available Resources

We also investigated how Tailwind improves internal server parallelism (i.e. more cores). Figure 13 shows throughput and worker utilization with respect to available worker cores. Clients (fixed to 30) issue 50/50 reads and writes to 4 servers. We do not count the dispatch core with available cores, since it is always available. With a single worker core per machine, RAMCloud serves 430 KOp/s compared to 660 KOp/s for Tailwind with respectively 4.5 and 3.5 worker cores utilization. RAMCloud can over-allocate resources to avoid deadlocks, explaining why it can go above the core limit. Tailwind scales better when increasing the available worker cores. RAMCloud does not achieve more throughput with more than 5 available cores. Tailwind improves throughput up to all 7 available cores per machine.

Even though both RAMCloud and Tailwind exhibit a plateau, this is actually due to the dispatch thread limit that cannot take more requests in. This suggests that Tailwind allows RAMCloud to better take advantage of per-machine parallelism. In fact, by eliminating the replication requests from dispatch, Tailwind allows more client-issued requests in the system.

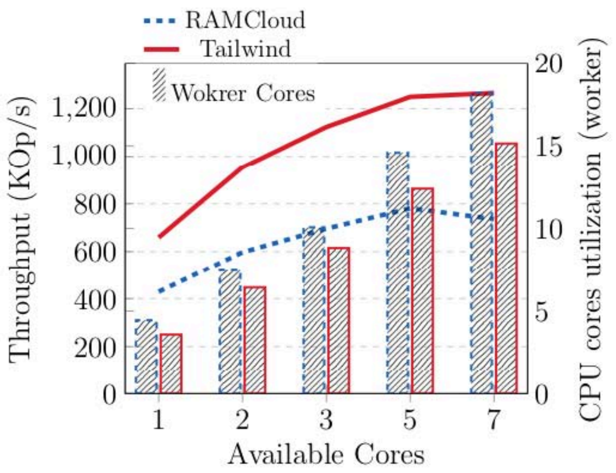

Figure 13. Throughput (lines) and total worker cores (bars) as a function of available cores per machine. Values are aggregated over 4 servers.

Impact on Crash Recovery

Tailwind aims to accelerate replication while keeping strong consistency guarantees and without impacting recovery performance. Figure 14 shows Tailwind's recovery perfor-mance against RAMCloud. In this setup data is inserted into a primary replica with possi-bility to replicate to 10 other backups. RAMCloud's random backup selection makes it so that all backups will end up with approximately equal share of backup data. After inserting all data, the primary kills itself, triggering crash recovery.

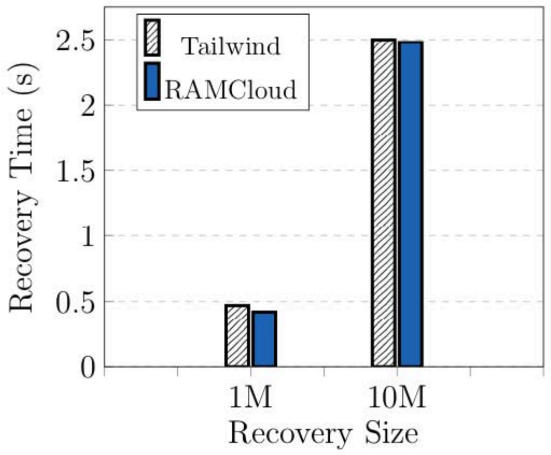

Figure 14. Time to recover 1 and 10 million objects with 10 backups

As expected, Tailwind almost introduces no overhead. For instance, to recover 1 million 100 B objects, it takes half a second for Tailwind and 0.48 s for RAMCloud. To recover 10 million 100 B objects, Both Tailwind and RAMCloud take roughly 2.5 s.

Tailwind must reconstruct metadata during recovery, but this only accounts for a small fraction of the total work of recovery. Moreover, reconstructing metadata is only necessary for open buffers, i.e. still in memory. This can be orders of magnitude faster than loading a buffer previously flushed on SSD, for example.

4.5 *Discussion*

Scalability

A current limitation of one-sided RDMAs is due to the requirement of reliable-connected QPs. Since information on QPs is cached at the NIC level, increasing the number of con-nections (e.g. scaling number of servers) over NIC cache size causes one-sided RDMA per-formance to collapse [Kal16]. By targeting only replication requests, Tailwind is not subject to such a limitation; in most common storage systems data is partitioned into chunks that are replicated several times. At any moment, only a small set of backups is replicated to.

Metadata Space Overhead

In its current implementation, Tailwind appends metadata after every write to guarantee RDMA writes atomicity. Although this approach appears to introduce space over-head, RAMCloud's log-cleaning mechanism efficiently removes old checksums without per-formance impact [RC15].

A different implementation could consist in appending the checksum with log-backup data instead. This would completely remove space Tailwind space overheads. However, this would raise a challenge on backups: how to decide backup buffers size. In the current imple-mentation primary DRAM log-storage and buffers on backups are the same. If checksums were appended to backup-data only, then buffers on backups will not have the same size as segments on primaries. A fixed-size buffer with only small objects will tend to have more checksums than a buffer which contains large objects, therefore more backup data could be stored on the latter.

In general, Tailwind adds only 4 bytes per object which is much smaller than, for exam-ple, RAMCloud headers (30 bytes).

Applicability

RDMA networks. Tailwind relies on RDMA-based networks to efficiently remove replication-CPU overheads. Historically, RDMA has been supported by Infiniband-based networks only. Recently, RDMA is supported in Ethernet in the form of RoCE or iWARP. More importantly, recent studies suggest that RDMA-based networks are becoming common in modern datacenters.

Note that Tailwind improves clients end-to-end latency but does not require any change on their side. Tailwind only requires RDMA-based networks on the storage system side to operate.

Distributed logging. Distributed logging is widely used approach to redundancy [Kafka]. Many recent in-memory key-value stores achieve durability and availability through dis-tributed logging as well [RC15, Farm]. Tailwind leverages log-backup data-layout in order to ensure the atomicity of one-sided RDMA writes.

Limitations

Tailwind is designed to keep the same system-level consistency guarantees. It synchronously waits for the acknowledgement from the remote backups NICs in order to return to the client. Although it can do work in parallel, worker cores have to stall waiting until the QP generates a work completion in order to move to process another write request. For instance, multiple updates can be queued and processed together. A better approach would be to pipeline replication requests. Worker cores would be able to perform some additional work and be notified when a work completion pops from the QP. RAMCloud's current RPC-based replication protocol is not pipelined neither [RC15] since its goal was to provide correctness and be able to reason about fault-handling.

4.6 *Summary*

Tailwind is the first replication protocol that fully exploits one-sided RDMA; it improves per-formance without sacrificing durability, availability, or consistency. Tailwind leaves backups unaware of RDMA writes as they happen, but it provides them with a protocol to rebuild metadata in case of failures. When implemented in RAMCloud, Tailwind substantially im-proves throughput and latency with only a small fraction of resources originally needed by RAMCloud.

With Tailwind, we make a step forward towards providing better efficiency in in-memory storage systems. As we show in Chapter 2, replication is expensive in in-memory storage systems, and it negatively impacts the energy efficiency and performance. Tailwind is a general design that can be applied to replicate data for any in-memory storage system.

5 Using Artificial Intelligence to Optimize Big Data Architectures

5.1 *Context and problem*

The overwhelming amount of data that needs to be processed nowadays has driven the design of increasingly complex distributed systems. This complexity is further exacerbated by new decentralised approaches, which process the data near where it is generated, such as Fog or Edge computing. Having full control of these infrastructures becomes a challenge even for the most experienced administrators, as there are many heterogeneous technologies and factors involved. Usually, administrators follow a process that involves using monitoring tools and browsing through logs in order to find insights that explain events happening in the system. Instead, this cumbersome process can be partially or totally automatised through the use of artificial intelligence techniques (AI) that extract these insights from all the incoming monitored information following a typical data science workflow. In this work, we propose a series of AI models that are able to solve some of the common problems that administrators find in these kind of systems. Namely, we focus on optimising the task parallelisation of Big Data jobs and performing root cause analysis for microservice architectures.

5.2 *Optimisation of task parallelisation of Big Data jobs*

Planning big data processes effectively on cloud and HPC platforms can become problematic. They involve complex ecosystems where developers need to discover the main causes of performance degradation in terms of time, cost or energy. However, processing collected logs and metrics can be a tedious and difficult task. In addition, there are several parameters that can be adjusted and have an important impact on application performance. One of the most important challenges is finding the best parallelization strategy for a particular application running on a parallel computing framework. Big data platforms like Spark or Flink use JVM's distributed along the cluster to perform computations. Parallelization policies can be controlled programmatically through APIs or by tuning parameters. These policies are normally left as their default values by the users. However, they control the number of tasks running concurrently on each machine, which constitutes the foundation of big data platforms. This factor can affect both correct execution and application execution time. Nonetheless, we lack concrete methods to optimize the parallelism of an application before its execution. This is specially challenging considering that the concurrent access to disk by these parallel tasks can create a bottleneck if it is not optimised.

We propose a method to recommend optimal parallelization settings to users depending on the type of application. We solve this optimization problem through machine learning, based on system and application metrics collected from previous executions. This way, we can detect and explain the correlation between an application, its level of parallelism and the observed performance. The model keeps learning from the executions in the cluster, becoming more accurate and providing several benefits to the user, without any considerable overhead. In addition, we also consider executions that failed and provide new configurations to avoid these kinds of errors. Finally, bottlenecks related to concurrent access to disk from various tasks are eliminated thanks to the optimal parallelism.

The benefits of an optimal task parallelisation were proven on a series of experiments in the Grid'5000 testbed. We build a benchmark that is a combination of Spark-bench, Bigdatabench and some workloads that we implemented on our own. We run all the applications of this unified benchmark with three different file sizes as we want to check how the model reacts when executing the same application with different amounts of data. These executions generate a series of metrics that we use to train and evaluate a machine learning model that has as inputs the metrics, the parallelism configuration and the duration of the workload. The final goal is to be able to predict the effect of different parallelism configurations on the duration of the workload depending on its metrics (the profile of the application). The overhead of applying the model is minimum, and the improvement in the workloads runtime goes up to 50% for graph processing processes. This is a first step towards optimising the access to data in distributed computing platforms.

5.3 *Root Cause Analysis for Microservice Architectures*

As many industries increasingly rely on information systems to operate efficiently, software development and architectures have evolved in different directions. Virtualisation has gained momentum in the last decade thanks to the ability of cloud providers to offer Infrastructure as a Service (IaaS) to their clients. The latest step in virtualization has been containerisation. Industry has acknowledged all the containers benefits, specially thanks to the adoption of technologies like Docke, with DevOps and microservices architectures becoming growing trends and common practices. The philosophy behind microservices consists of breaking the system logic into different, small units where each one has a single task or responsibility. The different units or microservices communicate and cooperate with each other to provide the system functionality as a whole. Any of the aforementioned units of logic is executed inside a container. Coordinating and scaling these containers require an orchestrating unit, specially if we want to deploy them in a distributed infrastructure with several machines. Tools like Kubernetes or DC/OS have filled this gap. These platforms have self-healing features, where unhealthy or faulty containers can be relaunched, or containers migrated to a different machine in case one of their hosts dies. But establishing the root cause of these failures can be really complex, specially when we have a network of different microservices that depend on each other. The troubleshooting process normally involves a tedious search through logs across the different containers, trying to find the faulty piece in the chain.

In a second step of our work, we present a graph-based method to perform root cause analysis (RCA) in such a microservice architecture. We argue that a graph representation of the system is able to capture important information for RCA, like the topology of the architecture or the different connections, both logical and physical, between elements in the system. Based on this graph representation, we propose a RCA system that is able to match an anomalous region in the system, represented as a subgraph A, with a similar anomalous graph B from the past that has already been troubleshooted by an expert. This similarity calculation relies on an inexact matching function that finds an optimal mapping between the nodes and edges of graph A and graph B. To solve this optimisation problem we apply AI techniques such as A* or Hill Climbing.

To evaluate the accuracy of this RCA technique, we create an experimental environment where we monitor the execution of three different microservice architectures commonly seen in industry and we inject six types of anomalies. All the monitored information is processed in order to build the graphs that will be passed onto our RCA framework to classify these anomalies into their root cause. Overall, our system achieves a 83% classification accuracy. We also compare these results with other methods that do not consider the topology of the system and the elements around the anomaly. To achieve that, we train a boosted classification trees model with a dataset that contains the metrics for the anomalous nodes as features and the anomaly that was induced in that node as the classification label. The results show how our graph-based approach improves the results of a machine learning method by 16 points. In addition, the system is easily interpretable and it can accept feedback from the user. Many monitoring tools adopt a graph approach to show the monitored information gathered and we believe that such an RCA approach could be a valuable addition to facilitate the performance and mantainance of microservice architectures.

6 References

[Aga13] Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., & Stoica, I. (2013). BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. *Eurosys'13*, 29–42. http://doi.org/10.1145/2465351.2465355

[Aki15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernandez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing, Proc. VLDB Endow., August 2015

[Aki16a] Tyler Akidau, The Evolution of Massive-Scale Data Processing https://docs.google.com/presentation/d/13YZy2trPugC8Zr9M8_TfSApSCZBGUDZdzi-WUz95JJw/present?slide=id.g63ca2a7cd_0_527

[Ale11] A. Alexandrov et al., "MapReduce and PACT - comparing data parallel programming models" in Proceedings of the 14th Conference on Database Systems for BTW 2011. Kaiserslautern, Germany, pp. 25–44.

[Apex] http://apex.apache.org/

[Arm15] Armbrust, M., Ghodsi, A., Zaharia, M., Xin, R. S., Lian, C., Huai, Y., … Franklin, M. J. (2015). Spark SQL. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, 1383–1394. http://doi.org/10.1145/2723372.2742797

[Ash09] Thusoo, A., Sarma, J. Sen, Jain, N., Shao, Z., Chakka, P., Anthony, S., … Murthy, R. (2009). Hive - A Warehousing Solution Over a Map-Reduce Framework. Sort, 2, 1626–1629. http://doi.org/10.1109/ICDE.2010.5447738

[Bdb] Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '99). USENIX Association, Berkeley, CA, USA, 43-43.

[Beam] http://beam.apache.org/

[Beam2] http://beam.apache.org/documentation/runners/capability-matrix/

[Bey16a] https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

[Bey16b] https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102

[Bigt] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages. DOI: http://dx.doi.org/10.1145/1365815.1365816

[Bra18] Álvaro Brandón Hernández, María S. Perez, Smrati Gupta, Victor Muntés-Mulero. Using machine learning to optimize parallelism in big data applications, Future Generation Computer Systems, Volume 86, 2018, pages 1076-1092, ISSN 0167-739X, https://doi.org/10.1016/j.future.2017.07.003.

[Car15] Paris Carbone, Gyula Fora, Stephan Ewen, Seif Haridi, Kostas Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows, 2015, https://arxiv.org/pdf/1506.08603.pdf

[Cass] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35-40. DOI: http://dx.doi.org/10.1145/1773912.1773922

[Ceph] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: a scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, Berkeley, CA, USA, 307-320.

[Clab] Ricci Robert and Eide Eric. "Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications". In: ;login: 39.6 (2014), pp. 36–38. ISSN: 1045-9219.

[Couch] Jan Lehnardt J. Chris Anderson and Noah Slater. CouchDB: The Definitive Guide. O'Reilly, first edition edition, 2010

[Data] https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

[Datb] https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

[Dea04] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI 2004, San Francisco, http://dl.acm.org/citation.cfm?id=1251254.1251264.

[DEC] http://sites.computer.org/debull/A15dec/A15DEC-CD.pdf

[DLog] "Apache DistributedLog," http://bookkeeper.apache.org/distributedlog/.

[Dynamo] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12). ACM, New York, NY, USA, 729-730. DOI: http://dx.doi.org/10.1145/2213836.2213945

[Farm] Dagojevic, Castro TRO, M. Farm: Fast remote memory. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 401–414.

[Flia] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/table_api.html

[Flib] https://cwiki.apache.org/confluence/display/FLINK/FlinkML%3A+Vision+and+Roadmap

[Flink] http://flink.apache.org/

[Flink2] https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/index.html

[FlinkS] https://flink.apache.org/news/2016/05/24/stream-sql.html

[G5K] "Grid5000," https://www.grid5000.fr.

[Gelly] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/gelly/index.html

[Gon14] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., Gonzalez, J. E., … Stoica, I. (2014). GraphX: Graph Processing in a Distributed Dataflow Framework. 11th USENIX Symposium on Operating Systems Design and Implementation, 599–613. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[GraphX] http://spark.apache.org/graphx/

[Hada] http://hadoop.apache.org/

[Hadb] https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[Hao14] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, Ion Stoica, Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks, SoCC 2014, Seattle USA, pp. 6:1--6:15, http://doi.acm.org/10.1145/2670979.2670985

[Hbase] Hbase, https://hbase.apache.org/

[Hdfs] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10). IEEE Computer Society, Washington, DC, USA, 1-10. DOI: http://dx.doi.org/10.1109/MSST.2010.5496972

[Heron] https://blog.twitter.com/2015/flying-faster-with-twitter-heron

[Igraph] Infinite Graph, http://www.objectivity.com/products/infinitegraph/

[Jai08] Namit Jain et al. Towards a Streaming SQL Standard, Proc. VLDB Endow., August 2008, pages 1379--1390, http://dx.doi.org/10.14778/1454159.1454179.

[Kafka] https://kafka.apache.org/

[Kafka1] http://kafka.apache.org/intro

[Kafka2] http://data-artisans.com/kafka-flink-a-practical-how-to/

[Kafka3] https://spark.apache.org/docs/1.6.1/streaming-kafka-integration.html

[Kal14] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In Proceedings of the 2014 ACM Conference on SIGCOMM (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.

[Kal16] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 185–201.

[Kul15] Sanjeev Kulkarni et al. Twitter Heron: Stream Processing at Scale, SIGMOD 2015, Melbourne, Victoria, Australia, pages 239--250, http://doi.acm.org/10.1145/2723372.2742788

[Lee15] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, "Implementing Linearizability at Large Scale and Low Latency," in 25th SOSP. ACM, 2015, pp. 71–86.

[Lev13] http://blogs.cisco.com/security/big-data-in-security-part-ii-the-amplab-stack

[Li17] LI, B., RUAN, Z., XIAO, W., LU, Y., XIONG, Y., PUTNAM, A., CHEN, E., AND ZHANG, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In Proceedings of the 26th Symposium on Operating Systems Principles (New York, NY, USA, 2017), SOSP '17, ACM, pp. 137–152.

[Mar16] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, Maria S. Perez-Hernandez, Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks, CLUSTER, Sept. 2016, Taiwan, Taipei, https://hal.inria.fr/hal-01347638v2.

[Mar18] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez-Hernández, Bogdan Nicolae, Radu Tudoran, Stefano Bortoli . KerA: Scalable Data Ingestion for Stream Processing. In ICDCS 2018 – 38th IEEE International Conference on Distributed Computing Systems, Jul 2018, Vienna, Austria. IEEE, pp.1480-1485, 2018.

[Men16] Meng, X., Bradley, J., Street, S., Francisco, S., Sparks, E., Berkeley, U. C., … Hall, S. (2016). MLlib : Machine Learning in Apache Spark, *17*, 1–7.

[Mia17] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "Streambox: Modern Stream Processing on a Multicore Machine," in USENIX ATC. USENIX Association, 2017, pp. 617–629.

[MLB] http://web.cs.ucla.edu/~ameet/mlbase_website/mlbase_website/index.html

[MLlib] http://spark.apache.org/mllib/

[Mongo] https://www.mongo.com

[Nat05] http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html

[Neo4j] https://neo4j.com/

[Nis13] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In Presented as part of the 10th USENIX Symposium on Networked Systems Design and Im-plementation (NSDI 13) (Lombard, IL, 2013), USENIX, pp. 385– 398.

[Pulsar] "Apache Pulsar." https://pulsar.incubator.apache.org/.

[RC11] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosen-blum, "Fast Crash Recovery in RAMCloud," in 23rd SOSP. ACM, 2011, pp. 29–41.

[RC15] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud Storage System," ACM Trans. Comput. Syst., vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015.

[RC17] C. Kulkarni, A. Kesavan, R. Ricci, and R. Stutsman, "Beyond Simple Request Processing with RAMCloud," IEEE Data Eng., 2017.

[Rey14] https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html

[Samza] http://samza.apache.org/

[Shi15] J. Shi et al., "Clash of the titans: Mapreduce vs. spark for large scale data analytics," Proc. VLDB Endow., vol. 8, pp. 2110–2121, Sep. 2015, http://dx.doi.org/10.14778/2831360.2831365

[Spark] http://spark.apache.org/

[SparqS] http://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[Storm] http://storm.apache.org/

[Su17] SU, M., ZHANG, M., CHEN, K., GUO, Z., AND WU, Y. Rfp: When rpc is faster than server-bypass with rdma. In Proceedings of the Twelfth European Conference on Computer Systems (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 1–15.

[Tal17] TALEB, Y., IBRAHIM, S., ANTONIU, G., AND CORTES, T. Characterizing performance and energy-efficiency of the ram-cloud storage system. In 2017 IEEE 37th International Confer-ence on Distributed Computing Systems (ICDCS) (June 2017), 1488–1498.

[Tal18] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, Toni Cortes. Tailwind: Fast and Atomic RDMA-based Replication. In ATC '18 – USENIX Annual Technical Conference, Jul 2018, Boston, United States. pp.850-863, 2018.

[Tan09] Stewart Tansley, Kristin Michele Tolle. The Fourth Paradigm: Data-intensive Scientific Discovery, Microsoft Research, 2009

[Tos14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel*, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy. Storm @Twitter, SIGMOD 2014, Snowbird, Utah, USA, pages 147--156, http://doi.acm.org/10.1145/2588555.2595641

[Tsa17] TSAI, S.-Y., AND ZHANG, Y. Lite kernel rdma support for datacenter applications. In Proceedings of the 26th Symposium on Operating Systems Principles (New York, NY, USA, 2017), SOSP '17, ACM, pp. 306–324

[Tud14] Radu Tudoran, Alexandru Costan, Olivier Nano, Ivo Santos, Hakan Soncu, Gabriel Antoniu. JetStream: Enabling high throughput live event streaming on multi-site clouds. Future Generation Computer Systems, Elsevier, 54: 274-291, 2016. DOI : 10.1016/j.future.2015.01.016

[Ven17] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica, "Drizzle: Fast and Adaptable Stream Processing at Scale," in 26th SOSP. ACM, 2017, pp. 374–389.

[War09] D. Warneke and O. Kao, "Nephele: Efficient parallel data processing in the cloud," in Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers 2009. NY, USA Portland, Oregon, 8:1--8:10, http://doi.acm.org/10.1145/1646468.1646476

[Xin13] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., Stoica, I., … Xin, R. S. (2013). Shark: SQL and rich analytics at scale. *Proceedings of the 2013 International Conference on Management of Data - SIGMOD '13*, 13. http://doi.org/10.1145/2463676.2465288

[ycsb] Brian F. Cooper, Adam Silberstein, Erwin Tam, et al. "Benchmarking Cloud Serving Systems with YCSB". In: Proceedings of the 1st ACM Symposium on Cloud Computing. SoCC '10. Indianapolis, Indiana, USA, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0.

[Zah12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. HotCloud June 2012, Boston, MA, http://dl.acm.org/citation.cfm?id=2342763.2342773

[Zah12a] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica, Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, NSDI 2012, San Jose, http://dl.acm.org/citation.cfm?id=2228298.2228301