From Big Data to Fast Data: Efficient Stream Data Management

Alexandru Costan

École doctorale MathSTIC

HABILITATION À DIRIGER DES RECHERCHES

Discipline: INFORMATIQUE

présentée devant l'École Normale Supérieure de Rennes sous le sceau de l'Université Bretagne Loire

par

Alexandru Costan

préparée à IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires

From Big Data to Fast Data: Efficient Stream Data Management

Soutenue à Bruz, le 14 mars 2019,
devant le jury composé de:

Rosa Badia / rapporteuse
Directrice de recherche, Barcelona Supercomputing Center, Espagne

Luc Bougé / examinateur
Professeur des universités, ENS Rennes, France

Valentin Cristea / examinateur
Professeur des universités, Université Politehnica de Bucarest, Roumanie

Christian Pérez / rapporteur
Directeur de recherche, Inria, France

Michael Schöttner / rapporteur
Professeur des universités, Université de Düsseldorf, Allemagne

Patrick Valduriez / examinateur
Directeur de recherche, Inria, France

Abstract

This manuscript provides a synthetic overview of my research journey since my PhD defense. The document does not claim to present my work in its entirety, but focuses on the contributions to data management in support of stream processing. These results address all stages of the stream processing pipeline: data collection and in-transit processing at the edge, transfer towards the cloud processing sites, ingestion and persistent storage.

I start by presenting the general context of stream data management in light of the recent transition from Big to Fast Data. After highlighting the challenges at the data level associated with batch and real-time analytics, I introduce a subjective overview of my proposals to address them. They bring solutions to the problems of in-transit stream storage and processing, fast data transfers, distributed metadata management, dynamic ingestion and transactional storage. The integration of these solutions into functional prototypes and the results of the large-scale experimental evaluations on clusters, clouds and supercomputers demonstrate their effectiveness for several real-life applications ranging from neuro-science to LHC nuclear physics. Finally, these contributions are put into the perspective of the High Performance Computing - Big Data convergence.

Keywords:

Big Data; stream processing; storage; data management; data analytics; transactions; data transfers; metadata management; in-transit processing; workflow management; HPC.

Contents

Foreword

THE LAST SEVEN YEARS, SINCE MY PH.D. DEFENSE, HAVE BEEN INCREDIBLY RICH and exciting. This manuscript does not claim to present all my research during this period. It focuses on the challenges associated with the evolution from traditional Big Data processing to the recent stream processing, from a data management perspective.

My interest for making sense out of huge sets of data was forged during my Ph.D., working on the *MonALISA* monitoring system for the LHC experiments at CERN. Then, my focus was naturally steered on ways of effectively storing and handling these data during my post-doctoral stay at Inria Rennes in 2011–2012. The work presented in this manuscript mainly covers the period 2012–2018, both at Inria and at IRISA, since my recruitment at INSA Rennes. However, it seems obvious that the evolution of my research themes is not exempt from the strong influence of the various collaborators with whom I had the pleasure of working and to whom I would like to thank. Since my arrival at Inria, my research has been organized along two major axes, presented briefly in Figure 1.

Big Data processing optimizations. My initial focus was on enhancing the MapReduce paradigm, the de-facto processing model for Big Data at that time. I studied the extension of this model with iterative support for reduce-intensive workloads, designed a blob storage system able to expose the locality needed by MapReduce and eventually generalized the MapReduce processing from single clusters to geographically distributed ones.

From Big Data to Fast Data. In parallel, I became more and more aware that processing performance is highly determined by the efficiency of the underlying data management. So, I started to focus on collecting data, processing them in-situ or in-transit, transferring them from their source to the cloud datacenters and eventually pushing them to the processing engines and storing them for persistence. Initially, this was done in the context of Big Data ("data at rest"), focusing on batch, *a posteriori* processing. But more recently, given the explosion of sensors and applications needing immediate actions, I shifted my interest towards Fast Data ("data in motion"), focusing on stream, *real-time* processing.

I summarize below the context and the chronological evolution of my research along these two axes.

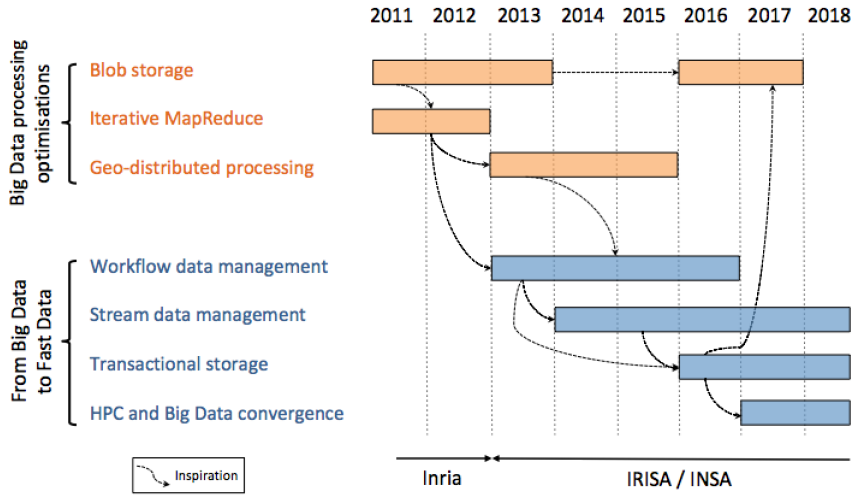


Figure 1 – Thematic organization of my research during the 2011–2018 period.

Big Data processing optimizations

During my Ph.D., I was mainly interested by visualization and presentation of (monitoring) data. In the following period, I became more attracted by the processing phase — making sense out of these huge volumes of data.

After my arrival at Inria, I studied the limitations of the MapReduce processing model, namely the lack of iterative support and the poor data management on clouds. Naturally, I joined the ANR MapReduce project [175], which aimed to address precisely these challenges on clouds and desktop grids. In this context, I co-supervised the thesis of **Radu Tudoran** and contributed to several optimisations for MapReduce (i.e., iterative support and geo-distributed processing) [19], and to a blob storage system, specifically designed to leverage the locality of MapReduce [1, 23, 22]. The project brought the opportunity to visit **Kate Keahey** at Argonne National Laboratory. We validated together these approaches on the Nimbus cloud and we also setup a new associated team between Inria and ANL. Moreover, in collaboration with **Gilles Fedak** (Inria, ENS Lyon) we were able to show that these optimisations can also benefit desktop grids [2].

Seeking validation of the storage and processing optimisations with a real-life application, I joined the A-Brain project [174] with the Microsoft Research Inria Joint Centre. The project analyzed large masses of neuroimaging data through cloud computing. An important breakthrough of this period was possible after meeting **Bertrand Thirion** (Inria Saclay). Thanks to the geo-distributed version of MapReduce on the Azure cloud, his team was able to prove for the first time the correlation between brain regions and genetic data [24]. This enables early diagnostic of psychiatric illnesses [13].

Although my work on Big Data processing optimizations has played an important role in shaping the results presented hereafter, I decided to leave it out of the scope of this document for the sake of clarity and brevity.

From Big Data to Fast Data

My recruitment as an Associate Professor at INSA Rennes in 2012 coincided with the creation of the KerData team. This was an opportunity for me to contribute to the team's vision and research program. My initial focus was on Big Data processing models more general than MapReduce. *Workflows* are such an example. They describe the relationship between individual computational tasks (standalone binaries) and their input and output data in a declarative way (e.g., directed acyclic graphs) and exchange data through files. Workflows are typically processed in batches. More recently, my interest steered towards the emerging trend of processing data in real-time, by means of streams. *Streams* are unbounded, unordered, global-scale datasets, flowing from their production site to the datacenters. They are often referred as Fast Data, due to their small item size and high processing rate. Workflows and streams come with specific data management challenges, which I addressed incrementally.

Workflows: Big Data management. A significant category of workflows typically handle a large number of small files (in the order of kilobytes). In order to understand the impact of this data access pattern on the processing performance, I joined the ZCloudFlow project [180], with Microsoft Research. The project investigated the problem of advanced data storage and processing for workflows on multi-site clouds. In this context, I supervised the thesis of **Luis Pineda** and I came to collaborate with **Patrick Valduriez** (Inria Sophia Antipolis) on geo-distributed metadata management for scientific workflows [15, 14]. Little research has been devoted to accelerate the data movements for workflows. In order to fill this gap, I started to work with **Bogdan Nicolae** (ANL) on several generalizations of the Multipath TCP protocol to speed up workflow data transfers [3, 18, 21]. All these contributions were delivered in a seamless way to scientists, through a general, easy to use *Workflow Data Management as a Service* [20, 26].

Streams: Fast Data management. From their production site (e.g., wireless sensors) to the processing nodes, streams typically pass through a three-stage pipeline: *collection*, *ingestion* and *storage*. In the absence of dedicated data management tools for each of these phases, I proposed in 2015 a uniform approach for stream data management. This stayed at the core of the ANR OverFlow project [176] that I was granted in 2015. In this context I co-advised with **Luc Bougé** (ENS Rennes) the thesis of **Paul Le Noac'h** and started the collaboration with **Pedro Silva**, postdoctoral researcher at KerData.

At the collection level, I contributed to the *JetStream* system [25, 28, 27] for fast delivery of streams to the clouds, by co-advising the thesis of **Radu Tudoran**. JetStream was integrated in the Microsoft Azure cloud for message transfers within the control plane.

At the ingestion level, state-of-the-art solutions partition streams statically which limits elasticity. To alleviate from this, I took the technical lead of the HIRP project [178] with Huawei Research, which focused on low-latency storage for streams. In this context, I co-supervised the thesis of **Ovidiu Marcu** and collaborated with the team of **Goetz Brasche** (Huawei) to design *KerA* [7, 5, 8, 16], a dynamic ingestion solution for streams.

At the storage level, neither distributed file-systems nor key-value stores serve well the needs of streams. The H2020 BigStorage project [177], focusing on HPC and cloud storage, was the occasion to study the pathway towards an efficient stream storage. This was the framework for a close collaboration with **Maria S. Pérez** (Universidad Politécnica de Madrid), which brought a fresh modelling and machine-learning perspective to my storage vision. Together with **Gabriel Antoniu** (Inria Rennes), we co-advised the thesis of **Pierre Matri**, which led to the first prototype of a transactional blob storage system — *Týr* [10, 12, 9]. Týr gained a huge interest at the SuperComputing conference where it was presented in 2016. I used this momentum to strengthen our team's collaboration with **Rob Ross** (ANL). We worked on TýrFS [11] — a file-system atop Týr able to increase small file access performance up to one order of magnitude compared to state-of-the-art file-systems. We also devised a set of design principles for converged storage for HPC and Big Data.

The remainder of this manuscript focuses on my second line of research: *From Big Data to Fast Data*. At a first glance, these contributions span on several distinct domains (i.e., workflow data management, file transfers, stream ingestion, blob storage etc.). However, an original aspect brought by this manuscript is their presentation in a new, uniform perspective — from the angle of *stream processing*. Essentially, they can be considered as multiple facets of a single more general problem: scaling-up in large distributed systems, both in terms of latency (accessing data faster) and throughput (getting the results sooner). These observations are both my inspiration and the foundation of my research contributions.

Chapter 1

Introduction

Contents

WE LIVE IN A DATA-DRIVEN WORLD. The amount of data generated in 2016 was 16.1 Zettabytes[1] [109], while forecasts estimate that the world will create 163 Zettabytes in 2025, a ten-fold increase in less than a decade. Interestingly, same studies anticipate that by that time the global volume of data subject to analysis will be of "only" 3 %, roughly 5.2 Zettabytes. Not only will this deluge of data raise issues in storing, managing and analysing the 3 %, but it will also challenge the processing paradigms in order to filter out on the fly the other 97 %.

To extract useful information from this huge amount of data, various frameworks and data processing models have been proposed. Among them, the most notable, MapReduce [79] and its open-source implementation Hadoop [42] quickly became an industry standard for Big Data processing, mainly due to their simplicity and scalability. More recently, new frameworks like Spark [192] were introduced in an attempt to abstract from the very strict, functional model based on Map and Reduce steps. They generalize the processing model to support iterative applications such as machine learning or graph processing. These frameworks are usually deployed on *clouds* [100], and execute complex analytics on *batches of historical data*.

[1] One Zettabyte is approximately equal to a thousand Exabytes or a billion Terabytes ($1ZB = 10^{21} bytes$).

1.1 The need for real-time processing

However, the most explosive proliferation in data generation today is taking place across the network from the cloud datacenters, at its *edge*. Such new data sources include major scientific experiments (e.g., LHC at CERN) and instruments (e.g., Square Kilometre Array telescope), and a deluge of distributed sensors from the Internet of Things (IoT).

Figure 1.1 illustrates this change. The edge environments in which these data originate are currently lacking the processing and storage resources to handle such volumes. The highest concentrations of computing power and storage remains at the "center", in commercial clouds or High Performance Computing (HPC) centers.

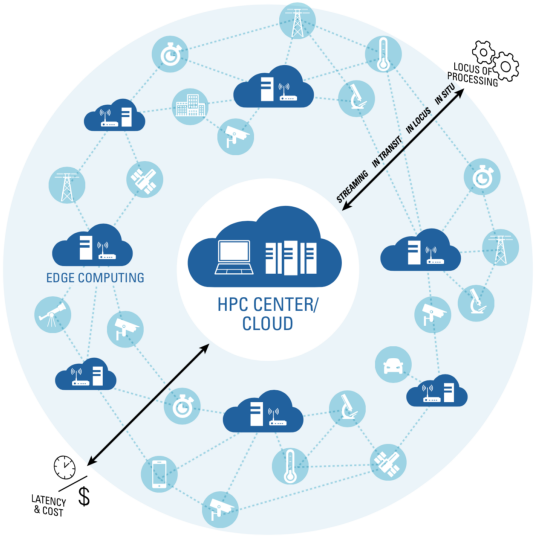


Figure 1.1 – Edge vs. cloud [54].

> ❝ *The traditional approach of shipping all data to the cloud for batch analytics is no longer a viable option.* ❞

The traditional approach of shipping all data to the cloud for batch analytics is no longer a viable option due to the high latency of the Wide Area Networks (WANs) connecting the edge and the datacenters. This disruptive change makes the advance in many different areas of research uncertain. At the core of this uncertainty lies the fact that the explosive growth of heterogeneous data torrents at the edge creates problems that are highly dimensional. Let us illustrate them with an example from sensor data processing.

1.1.1 Motivating use-case: autonomous cars

We are currently witnessing the biggest and most exciting change to driving in the history of the automobile: autonomous cars. The U.S. Department of Transport identified 5 levels of autonomy when it began looking into legislation for self-driving cars. The Level 5 (highest) stipulates that the vehicle will be able to drive itself with no human supervision or input whatsoever, and function just as effectively as any human driver would, on any road and in any condition. Full Level 5 autonomy is expected by 2021.

Sensors. Autonomous vehicles are outfitted with a myriad of sensors (radars, LIDARs, cameras) to monitor things like their position, proximity to obstacles, traffic guides and much more. At any given time, the cars are tirelessly analyzing their local surroundings, looking for telltale signs that the brakes need to be applied or that they need to accelerate.

Autonomous connected cars will make Big Data even bigger. Currently, the data estimated to be generated by an autonomous vehicle rises to 300TB per year [55]. This figure is expected to shift with at least one order of magnitude since currently autonomous cars comply (only) with Level 3 (L3) of autonomy. Drivers are still necessary in L3 cars, but are

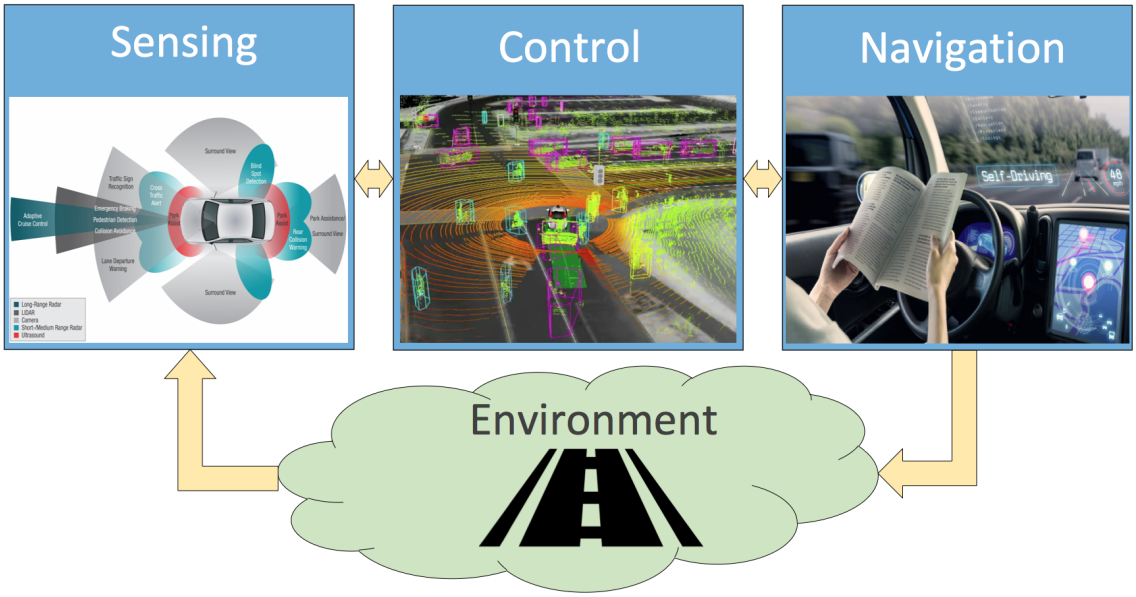


Figure 1.2 – Big Data hits the road: the autonomous loop fueling the self-driving cars with huge torrents of data. Data is first collected from the on-board sensors, analyzed for effective control using machine-learning algorithms, and then acted upon.

able to completely delegate safety-critical functions to the vehicle, under certain conditions. There is a popular view that the technology could move quickly beyond L3 autonomy, effectively leaping towards L5 (full autonomy). This L5 vehicle is thus likely to become the most complex and the most data-rich platform in the whole IoT, creating huge challenges for identifying critical risks and designing against them. By 2020, it is expected that more than 10 million self-driving cars will be on the road.

Real-time decisions. Most of the data from such sensors need to be measured, monitored and alerted upon in real-time, following the autonomous loop depicted in Figure 1.2. The key issue is how to process such massive amount of data very fast, detect anomalies or, even better, predict a fault before it happens: accidents, abnormal driver or car behavior.

In such a case, it is easy to see why the traditional Big Data processing approaches based on back hauling all data across WANs to the cloud for batch-based historical analytics are not sufficient. The value of sensor data decreases drastically over time, and the ability to make decisions based on that data is only useful if the analysis is done in near *real-time*.

1.1.2 Solution: stream computing in real-time

> ❝Streams *are unbounded, unordered, global-scale datasets, flowing from their production site to the datacenters. They are often referred as* Fast Data.❞

Under the pressure of increasing data velocity, Big Data analytics has shifted towards a new paradigm: *stream computing*. *Streams* are unbounded, unordered, global-scale datasets, flowing from their production site to the datacenters. They are often referred as *Fast Data*, due to their small item size

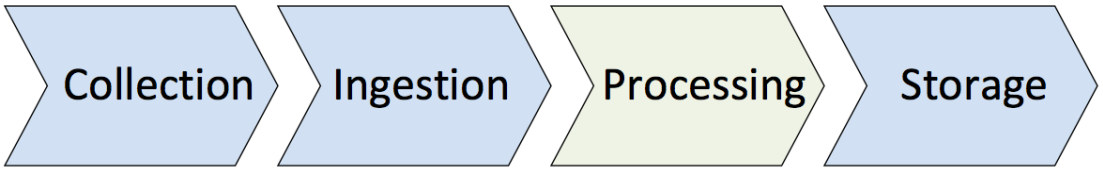


Figure 1.3 – The typical stream computing pipeline. In blue, the steps relevant for data management (discussed in this manuscript).

and high processing rate. *Stream computing* applies a series of operations to each element in the stream, typically in a pipeline fashion. Hence, stream computing is able to deliver insights and results as soon as possible, with minimal latency and high throughput that keeps up with the rate of new data. With streaming analytics, one can:

(i) **Aggregate or filter data.** Group variables or eliminate noise near the source and only transmit the results. For the connected cars, this allows data reduction to be applied locally before any data movement is attempted.

(ii) **Spot meaningful trends, patterns and correlations.** Get instant insights to react immediately with no need to transmit and store status quo data. In the case of autonomous cars, constantly analyzing events as they occur can detect patterns that are otherwise lost through information lag.

(iii) **Run complex analytical models** for predictive behavior and proactive decisions. For autonomous cars, one can forecast expected values just a few seconds into the future, and compare actual to forecasted values to identify meaningful deviations that may indicate a problem leading to abnormal behavior.

(iv) **Make smart, real-time decisions.** In the case of autonomous cars, this allows to avoid obstacles and constantly adapt to the driving environment.

1.2 The challenge of data management for streams

The typical stream computing pipeline consists of four stages, depicted in Figure 1.3.

- **Collection** acquires data from sensors or other distributed sources and transfers them to the cloud datacenters, using simple protocols like FTP or UDP.
- **Ingestion** serves to buffer, log and optionally pre-process data streams (e.g., filter) before they are consumed by the processing phase. Apache Kafka [44] is the most widely used ingestion system.
- **Processing** executes complex analytics on streams. This stage is served by a large family of Stream Processing Engines (SPEs). Examples include Flink [70], Spark Streaming [193], Storm [51], Samza [148] and others.
- **Storage** enables data persistence. This typically involves either the archival of the buffered data streams from the ingestion layer or the storage of the intermediate results of stream processing. Various backends are used today: from object stores (e.g.,

Amazon S3 [38]) and distributed file-systems (e.g., HDFS [63]) to key-value stores (e.g., Cassandra [121], RAMCloud [151]).

The processing phase is served by a rich eco-system of scalable SPEs leveraging *dedicated* abstractions for streams (i.e., *DataStreams* in Flink, *RDDs* in Spark). These allow to process streams on the fly with low latency.

> *❝ The data management phases (i.e., collection, ingestion and storage) are still supported by decade-old, general-purpose solutions. ❞*

However, the data management phases (i.e., collection, ingestion and storage) are still supported by decade-old, general-purpose solutions. For instance, streams are typically transferred using the very basic UDP protocol and stored in distributed file-systems like HDFS. These solutions do not leverage the specificities of streams: small size per item (in the order of bytes and kilobytes) and high arrival rate (in the order of millions items per second). They were actually designed with opposite principles in mind: for instance HDFS is optimized for storing huge binary objects (in the order of gigabytes) for batch-based processing.

The lack of adequate systems, tailored to the specificities of streams, for *collection*, *ingestion* and *storage* has a great impact on the processing performance. On the one hand, the SPEs are not served at their full processing capacity, resulting in wasted resources, and eventually delayed results. On the other hand, SPEs remain tributary to the old model of collecting the stream data to a centralized datacenter for processing. Instead, with a dedicated collection system for instance, one could do some parts of the processing *in-transit*, with faster results and less data movements.

1.3 Mission statement

We claim that ***dedicated* stream data management could lead to substantial performance improvements over the state-of-the-art**, that was designed for batch processing of Big Data.

1.4 Objectives

In order to prove this hypothesis, this manuscript fulfils a series of sub-objectives covering all three stages of stream data management — collection, ingestion and storage.

Evaluate existing software

We are interested in assessing whether the state-of-the-art solutions are ready to support the next-generation models and architectures for stream computing. This translates into the following sub-objective.

Investigate a series of existing data management and processing paradigms for distributed computing in order to *understand their limitations* with respect to stream processing (Chapters 2, 3 and 4).

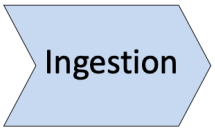
Improving collection

Our goal is to enable parts of the computation to be executed close to the data collection site, at the edge, in order to avoid routing large amounts of data over the network. Even when this is impossible, we aim to minimize the data transfer time from the edge to the cloud. This leads to the following sub-objectives.

Show that using dedicated nodes for *in-transit* stream data management at the edge or in the cloud is not only possible, but also brings substantial benefits to both cloud providers and users. We introduce DataSteward, an *in-transit storage and processing* system for streams (chapter 5).

Reduce stream collection time from source to the processing clouds by means of *fast data transfers across geographically distributed sites*. We propose JetStream, a high-performance batch-based streaming middleware for efficient transfers of streams between cloud datacenters over WANs, despite latency and bandwidth variations (chapter 6).

Improving ingestion



We aim to support most of the real-time processing applications with delays of no more than a few seconds. Hence, we need to be able to access and deliver high volumes of stream data with low latency. This translates into the following sub-objectives.

Improve stream access performance through *adaptive distributed metadata management*. Huge loads of metadata are required to keep track of each stream data unit and of the whole execution state. We introduce the notion of *hot metadata* (for frequently accessed metadata), and show that a distributed architecture for accessing it can speed up the stream processing, especially in multi-site clouds (chapter 7).

Speed-up stream delivery for processing. We propose KerA, a *data ingestion framework* that accelerates stream buffering and logging prior to processing, thanks to a dynamic partitioning scheme and to lightweight indexing (chapter 8).

Improving storage

We identify data consistency as a hard, unaddressed problem in the context of highly concurrent streams. Our goal is to propose a storage system natively offering data access coordination and ready to serve a broader spectrum of applications (i.e., not only stream-based, but also HPC ones, for instance). This motivates our final sub-objectives.

Introduce a transactional stream storage solution — Týr. In particular, we show that enforcing strong consistency at the storage level by means of transactions provides support for a variety of use-cases such as real-time data aggregation (chapter 9).

Define a series of design principles for converged storage for HPC and Big Data. We show that despite important divergences, storage-based convergence between HPC and Big Data is not only possible, but also leads to substantial performance improvements over the state-of-the-art (chapter 10).

Part I

Context: Stream Processing in the Clouds

Chapter 2

Big Data Processing: Batch-based Analytics of Historical Data

Contents

DATA IS THE NEW NATURAL RESOURCE. Its processing is nowadays transformative in all aspects of our world. However, unlike natural resources, whose value is proportional to the scarcity, the value of data grows larger the more of it is available. This trend is facilitated by *Big Data Analytics* (BDA): more data means more opportunities to discover new correlations and patterns, which lead to valuable insight.

Data science is thus emerging as the *fourth paradigm of science* [107]. It allows researchers to extract insights from (past) traces of both scientific instruments and computational simulations. This paradigm shift happened naturally. Centuries ago, science was mainly done through *empirical observations* (first paradigm of science). The next step was to synthesize those observations about the world in *theoretical models* (second paradigm). When those models became too complex to be solved and interpreted analytically and when technology

allowed it, science moved towards a third, *computational paradigm*. It used computers to analyze and simulate the theoretical models. However, this computation-driven science led to a continuous growth of the scientific data sets. The techniques and technologies used for "searching" for discoveries in these large data sets *"distinguish this data-intensive science from computational science as a new, fourth paradigm for scientific exploration"* [59].

> ❝ *Data-generation capabilities in most science domains are growing more rapidly than compute capabilities, causing these domains to become* data-intensive. ❞

Broadly speaking, data-generation capabilities in most science domains are growing more rapidly than compute capabilities, causing these domains to become *data-intensive*. Hence, a distinctive feature of BDA compared to High Performance Computing (HPC), which mostly focuses on large computational loads, is that it targets applications that handle very large and complex data sets (i.e., typically of the order of multiple terabytes or exabytes in size). BDA applications are thus very demanding in terms of storage while HPC is usually thought more in terms of sheer computational needs.

Cloud platforms are on par with these high needs. They democratize science, by facilitating the access to large-scale resources (e.g., storage, compute, network) by means of virtualization [53, 66, 111]. BDA on clouds (leveraging machine learning for instance) have already yielded new insights into health risks and the spread of disease via analysis of social networks, web-search queries, and hospital data [98]. It is also key to event identification and correlation in domains as diverse as high-energy physics and molecular biology [29].

As with successive generations of other large-scale scientific instruments, each new generation of advanced computing brings new capabilities, along with technical design challenges and economic trade-offs. BDA make no exception. The rising cost of ever-larger computing systems and the new challenges at massive scale are raising questions about the BDA systems design. This chapter examines these technical challenges, our proposed solutions to some of them (e.g., Map-IterativeReduce [19], GeoMapReduce [1], TomusBlobs [23]), as well as the global collaboration and competition for leadership in BDA. We begin with a primer on MapReduce, the *de facto* standard for BDA, and then follow the explosion of new programming models targeting diverse computing workloads.

2.1 Batch processing with MapReduce: the execution model for Big Data

At first, these programming models were relatively specialized, with new models developed for new workloads. MapReduce [78] was introduced by Google to support batch processing. It addressed the volume challenge by distributing partitions of input data across the cluster. This allowed to deploy and scale a simple data-crawling, shared-nothing strategy, processing locally, on each node, the data partitions. The solution was less powerful than the index mechanisms from databases [155] in terms of data access and processing time, but became a default BDA tool due to its massive scalability capabilities.

The MapReduce model is inspired from functional languages, as shown in Figure 2.1. It consists of 2 functions: *map* and *reduce*, executed in 2 successive phases by worker processes. The map function, written by the user, is applied in parallel, on all the partitions of the input

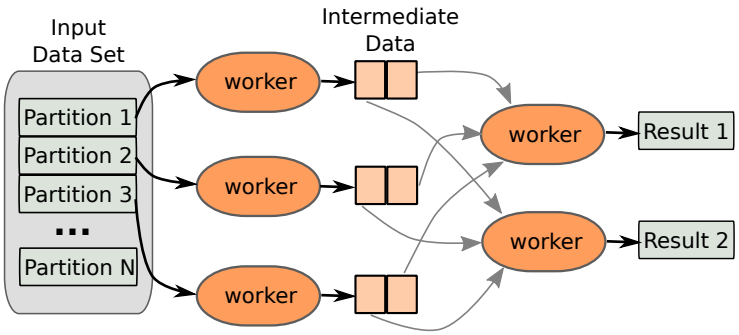


Figure 2.1 – The classical MapReduce scheme, as introduced in [78].

data and generates a set of key/value pairs. Example of Map functionality are: counting items occurrences, ranking records, searching for patterns etc. The reduce function, also provided by the user, merges all values associated with the same key to generate the final output. The assignment of the keys to be processed by each reducer is done based on a hashing function. Typical examples of Reduce operations are: sum, sort, filter, merge, etc.

The large success of this model is due to the fact that it effectively facilitates the execution of many parallel tasks by elastically scaling the computation on virtualized commodity hardware. Users only need to provide the 2 functions, which are then scaled with the number of resources and applied on the partitioned input set. Additionally, since the computation is performed on independent subsets, the model has a natural tolerance to failures.

2.1.1 MapReduce extensions

MapReduce enforces a clear hierarchy: the map and the reduce functions define the semantics of the processing. The simplicity of this model limits the workloads on which it can be applied. We developed several optimisations with the goal of extending this scheme to accommodate new types of processing, while preserving its initial design properties.

Map-IterativeReduce [19]. Reduce-intensive workloads that compute a unique output (e.g., min, max, select etc.) are not well served by MapReduce. The model lacks support for full reduction of the results, as each reducer provides a partial output. Map-IterativeReduce was our proposal to add support for *iterative reduce* processing to the initial MapReduce model. This extension enables to schedule efficiently the reduce jobs in parallel, based on a reduction tree, in order to compute a unique final result. Map-IterativeReduce further eliminates the synchronization barriers between the Map and the Reduce phases from the initial model. Instead, the reducers start the computation as soon as some data is available (Figure 2.2). Also, considering that results are accumulated down the reduction tree, there is no need for any (centralized) entity to control the iteration process, check the termination condition or collect data from reducers, as in vanilla MapReduce.

GeoMapReduce [1]. MapReduce was designed for single-cluster or single-site processing. However, BDA on the clouds requires many compute and data resources which are not always available in a single datacenter. To address this limitation and enable global scaling,

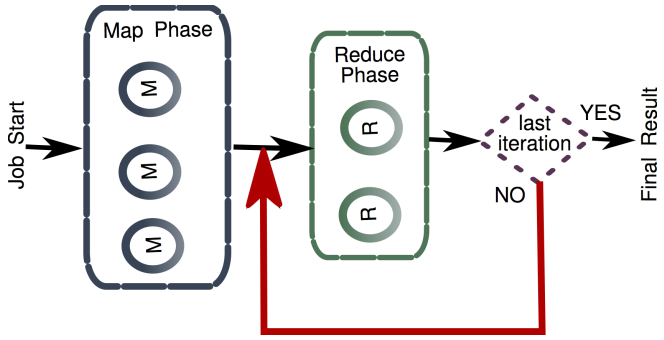


Figure 2.2 – The Map-IterativeReduce conceptual model.

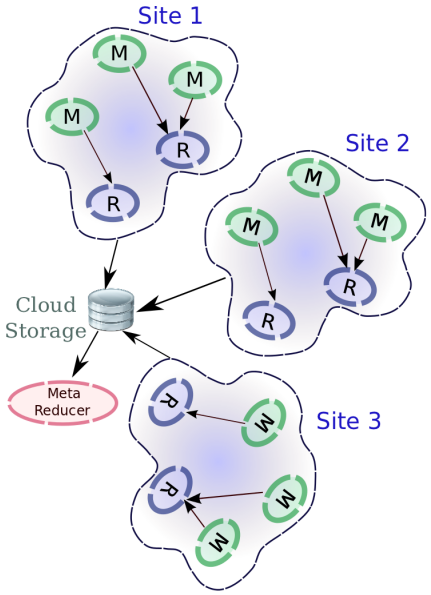


Figure 2.3 – GeoMapReduce computation across multiple sites.

we proposed GeoMapReduce, a *multi-site hierarchical* MapReduce scheme for geographically distributed computation. Data is partitioned, scattered and processed in parallel across several datacenters. Conceptually, GeoMapReduce relies on a layered data access model built on top of different storage systems in order to support both storage and transfers. The architecture is presented on Figure 2.3 and consists of two tiers:

- At the bottom level, distinct instances of Map-IterativeReduce are enabled within each datacenter in order to reduce locally the number of MapReduce outputs and in this way to minimize the inter-site data exchanges.
- In the top tier, the aggregation of the global result is computed by a new entity, called *MetaReducer*. Its role is to implement a final Reduce step to aggregate the results from all sites as soon as they become available.

Using these extensions, we were able to achieve high scalability for scientific applications in public clouds, leveraging the processing power of 1000 cores across 3 geographically distributed datacenters. Performance-wise, we were able to reduce the data management time by up to 60% compared with state-of-the-art solutions [1].

2.2 Big Data processing frameworks

A rich ecosystem of hardware and software environments has emerged for BDA. The common feature of these data analytics clusters is that they are typically based on commodity Ethernet networks and local storage, with cost and capacity as the primary optimization criteria. For instance, at Yahoo! the world's largest MapReduce cluster [104] uses more than 100,000 CPUs in over 40,000 commodity machines. To go further, industry is now turning to GPUs, FPGAs and improved network designs to optimize performance.

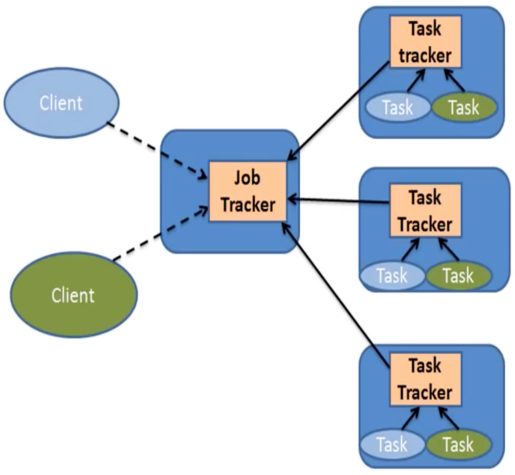


Figure 2.4 – The Hadoop architecture. The Job Tracker is the master node. The Task Trackers are the slave daemons running Map and Reduce tasks [189].

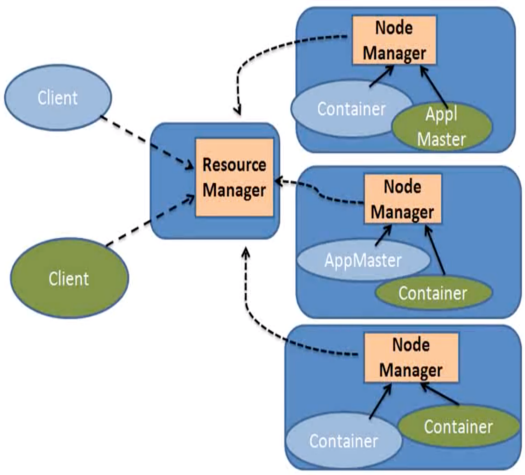


Figure 2.5 – The Yarn architecture. The Node Managers are generalized Task Trackers, handling any type of application — not only MapReduce [189].

2.2.1 From Hadoop to Yarn

Hadoop [42] is the most notorious and used implementation of MapReduce. Its core design principle is bringing the computation to the data. In a Hadoop cluster, each compute node has its direct attached storage, which has to be large enough to store the data of any assigned analytics job. One of the nodes in a Hadoop cluster, called the master node, is in charge of coordinating all other nodes (Figure 2.4). It also maintains a large amount of metadata so that it understands which data is where on the cluster and what node is the most available. The master node assigns the analysis of an analytics request to a node within its cluster that has the appropriate data on its internal storage. When the job is complete, the result is sent back to the master node and given to the user. As in the original model, the Map and Reduce phases are separated through a synchronization barrier.

Because of its success, the framework was adopted and used in diverse scenarios beyond its initial designed goals. For example, high-level frameworks such as Pig [48] or Hive [43], or full workflow descriptive languages such as Kepler [185], orchestrate complex and heterogeneous MapReduce execution plans. This extensive use, as well as a vast number of research studies, revealed several limitations of the architecture. Examples include the tight coupling of the processing model with the infrastructure or the scalability and the performance bottlenecks due to the centralized scheduling. To address these limitations, a new service-oriented architecture, called Yarn [183], was proposed.

Yarn splits the multiple responsibilities of the single master between several resource managers and application masters (Figure 2.5). The former administrate entire clusters while the latter schedule and coordinate the execution of the work per application. The compute infrastructure is divided into discrete quotas called containers, which are managed by entities called node managers. In this way, Yarn provides better resource utilization, higher fault tolerance and greater scalability.

In the open source Apache Hadoop stack, several systems like Drill [40] and Oozie [47] propose some specialized engines for interactive analysis and complex task scheduling, respectively. As the MapReduce model is not general enough, they move away from the "one-size-fits-all" trend. Instead, they propose some dedicated solutions for very specific processing needs (e.g., SQL queries, machine learning algorithms, columnar storage, etc.)

2.2.2 Workflow management systems

Unfortunately, most BDA applications cannot be implemented using only one such specific engine. Instead, they need to combine several different ones. The very nature of Big Data is that it is diverse and messy. A typical pipeline will need MapReduce-like code for data loading and filtering, SQL-like queries, and iterative machine learning. Specialized engines can thus create both complexity and inefficiency. Users must stitch together disparate systems, and some applications simply cannot be executed efficiently in any engine.

Luckily, a large class of BDA applications can be expressed in a more general fashion, by means of workflows. *Workflows* are complex directed acyclic graphs (DAGs) that describe the relationship between individual computational tasks and their input and output data in a declarative way. Typically, workflows feature many interrelated, loosely-coupled tasks (up to thousands or even millions) with data in the order of petabytes. Unlike tightly-coupled HPC applications (e.g., MPI-based) communicating directly via the network, workflow tasks exchange data through files. This means that the end-to-end performance of such workflows strongly depends on the data handling of large distributed datasets on limited resources.

Workflow management systems are software systems for setting up, executing, and monitoring of workflows. Their broad goals are reducing the workflow execution time and minimizing the resource usage. Several workflow management systems were developed in order to support efficient deployment of workflows on clouds. We briefly recall the most representative ones, form both industry and academia:

Pegasus [82] optimizes the mapping of the workflow tasks on the resources, with the goal of minimizing the data movements. However, when migrated to the cloud, its performance is highly sensitive to the data management solution used. To a large extent, this determines the scheduling optimizations that can be applied [30].

e-Science Central [83] was designed as a workflow platform accessible via web browsers. It enables collaborative work, as well as data and computation sharing. Workflow execution is performed as a sequence of service invocations. The scheduling and the control of the execution are performed by a centralized coordinator which limits the scalability and the performance.

Generic Worker [169] was specifically designed by Microsoft Research to run on clouds. It uses a local infrastructure when available, and scales-out to a public or private cloud when necessary. By default, Generic Worker relies on the cloud storage to perform all data management operations such as tasks description or data uploading and sharing. This choice translates into low I/O performance for data transfers between tasks.

Dryad [110] introduced embedded user-defined functions in general DAG-based workflows. It was enriched with a language and an SQL optimizer on top of it. Apache Tez [52] can be seen as an open-source implementation of the principles in Dryad.

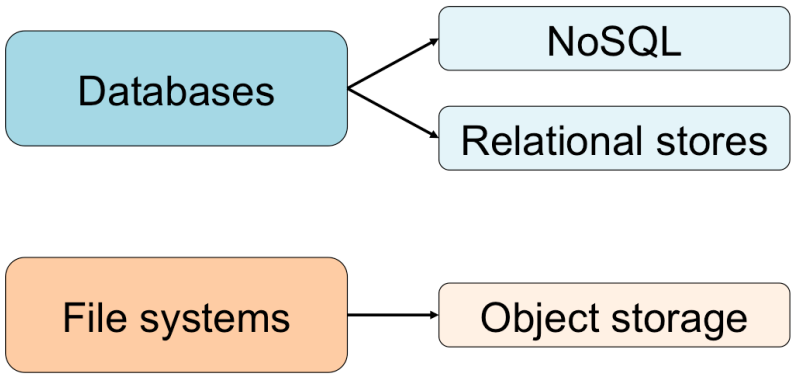


Figure 2.6 – Storage taxonomy according to the levels of storage abstraction.

2.3 Big Data management

As seen in the previous section, the BDA analytics architectures have been built from the bottom up to grind through enormous amounts of data with the highest possible performance. This is clearly dependent on the efficiency of the underlying data management (i.e., storage, buffering, transfer). The data management issue is typically solved by *bringing the computation to the storage nodes* rather than the opposite. This principle has driven the design of the underlying storage and transfer stack, that we briefly survey in this section.

2.3.1 Data storage

Big Data stores are generally classified according to the different levels of storage abstractions (Figure 2.6). These levels are as follows:

- **File-level** is the most common form of data abstraction due to its ease of management via simple APIs. It is typically provided by means of *object storage* that enable users to store large binary objects.
- **Database** mode offers storage in the forms of *relational data stores* and *NoSQL databases*. Relational stores exploiting the SQL standard do not scale easily to serve large applications but guarantee strong consistency (i.e., ACID properties: atomicity, consistency, isolation, durability). In contrast, NoSQL provides horizontal scalability by means of shared nothing, replicating, and partitioning data over many servers for simple operations. In fact, it preserves BASE (basically available, soft state, eventually consistent) properties instead of ACID ones to achieve higher performance and scalability. For these reasons, BDA applications typically use NoSQL.

According to how data is logically organized (i.e., structured or unstructured) and how it is retrieved (i.e., using keys or queries) these storage abstractions optimize in different dimensions. Intuitively, as the storage abstraction level increases, the applicability and the performance reduce. We now discuss the differences between file systems, object storage and NoSQL databases in order to understand their benefits and limitations for BDA.

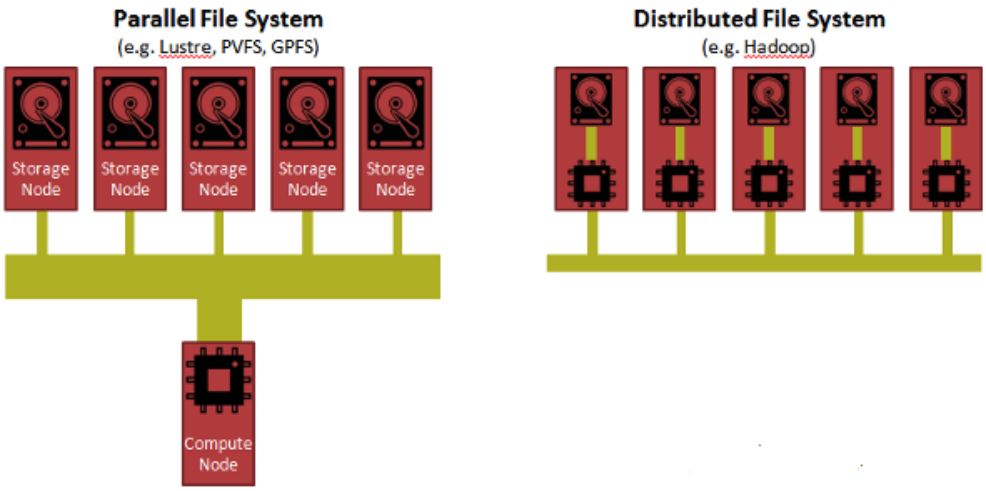


Figure 2.7 – Parallel and distributed file systems [134].

File systems

File-based storage are arguably the *de facto* standard in the industry. Large-scale applications use file systems aggregating several nodes, as shown in Figure 2.7. *Parallel file systems* are often run on storage architectures that are physically separated from the compute systems. They are designed for HPC applications and leverage fast networks to optimize for high concurrency. *Distributed file systems* run on architectures where the storage is typically co-located with the compute nodes. They are geared for BDA applications and enforce fault-tolerance. Key differences between various file systems are in their caching, consistency semantics, locking policies and support for the POSIX I/O standard, which has defined the expected API and behavior of local file systems for decades.

Parallel file systems tend to interplay with consistency support according to their targeted workload needs. PVFS [161] does not support POSIX I/O atomicity semantics. Instead, it guarantees sequential consistency only when concurrent clients access non-overlapping regions of a file (referred to as non-conflicting writes semantics). On the other hand, Lustre [164] guarantees consistency in the presence of concurrent conflicting requests, at a higher cost.

Distributed file systems hide from users the complexity of their distributed nature (i.e., data distribution, replication, load balancing, etc.) This transparency is largely due to them implementing part or all of the POSIX I/O standard. While most distributed file systems strictly adhere to the POSIX I/O API, the compliance to the associated semantics greatly varies from system to system. We briefly recall some representative distributed file systems.

HDFS (Hadoop Distributed File System) [63] manages many large files, distributed (with block replication) across the local storage of the cluster. It is integral part of the Hadoop stack and was developed to serve as the storage backend for MapReduce compute platforms. As such, HDFS only offers a subset of the POSIX I/O API: it does not need random file update semantics. This significantly eases the design of the file system by eliminating the synchronization required for handling potential conflicting write operations. Indeed, its design is very simplistic, following a master-workers architecture.

A centralized control component manages all metadata within the system, and additionally splits and disseminates the data for storage. However, HDFS cannot sustain a high throughput for concurrent reads [146]. Also, concurrent writes or appends are not possible. HDFS can be seen as the BDA analog of Lustre for HPC, albeit optimized for different hardware and access patterns.

BSFS/BlobSeer [146] is a concurrency-optimized distributed storage system for large binary objects, which can be used via a file system interface. Data striping and replication is performed transparently for applications. A version-oriented metadata scheme for serialization and atomicity enables lock-free access to data, and thereby favors scalability under heavy concurrency. The system is capable of delivering high throughput performance, but requires meticulous configuration, which is not always straightforward for application designers.

While file systems exhibit strong benefits for managing and organizing data, these benefits are usually shaded by significant performance and scalability limitations as the data size grows. *These limits are the result of their design:*

- **POSIX I/O** is almost universally agreed to be one of the most significant limitations standing in the way of I/O performance for large systems. This is due to various constraints it poses relative to the behavior of the file system when shared by multiple concurrent users. This includes its *stateful nature*, *prescriptive metadata* or its often-unneeded *strong consistency* when facing concurrent, conflicting writes. While these features are not problematic by themselves, they imply a substantial performance cost that the users have to pay no matter their actual requirements. For that reason, several file systems choose to relax parts of the POSIX I/O standard to offer increased performance for specific use-cases where these features limit performance for no tangible benefit [161]. Also, various extensions to the POSIX I/O standard have been proposed in the literature to achieve similar results [35, 119, 184].
- **The hierarchical nature**, which is at the very root of their design, can be seen as the most significant limitation with respect to performance. Indeed, when accessing any piece of data, its associated metadata must also be fetched. In a hierarchical design, accessing metadata implies fetching multiple layers of metadata. In file systems with centralized metadata, accessing it tends to become a bottleneck at scale because of the concentration of requests to a small number of machines. In file systems with distributed metadata, this bottleneck effect is reduced at the cost of a higher access latency due to the metadata distribution across the whole system, which only increases with the size of the system and the hierarchical depth of the files that are accessed.

These factors inherently limit the scalability potential of distributed file systems. They make them either unsustainable for exascale applications or requiring significant changes to the very design of these systems. This ultimately negates the very reason they were initially built for: providing a simple and efficient way for users to manage data.

Object-based storage

Distributed file systems are typically built as thin layers (adding file semantics) atop object storage. These allow to store binary large objects (blobs). They combine flat namespaces and

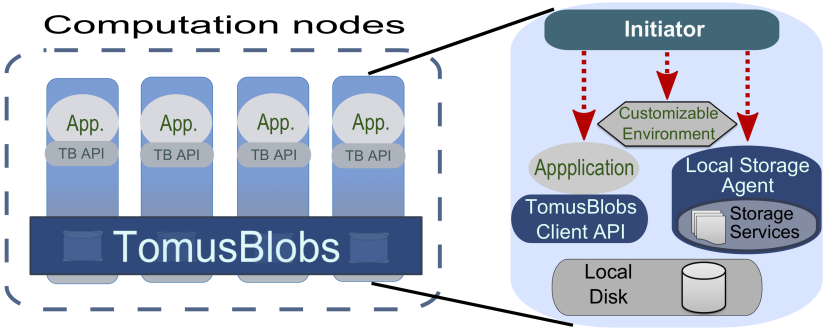


Figure 2.8 – The architecture of TomusBlobs.

small APIs which represent an interesting alternative to the hierarchical nature and POSIX semantics of the file systems. Hence, more and more BDA applications have started to use object storage directly, especially when file semantics are not needed.

Typical examples of such blob storage systems include RADOS [188], or hosted services available on cloud platforms such as Microsoft Azure Blobs [67] or Amazon S3 [38]. They are optimized for high availability and data durability, under the assumption that data are frequently read and only seldom updated. Therefore, achieving high throughput or enabling optimizations for Big Data applications (e.g., by exposing the data layout) are auxiliary goals. Moreover, there is a clear separation within the data center, between these systems and the computation infrastructure.

To address these issues for managing data in the clouds, we proposed an architecture for *concurrency-optimized, cloud object storage leveraging virtual disks*, called TomusBlobs.

TomusBlobs [23] federates the local disks of the application Virtual Machines (VMs) into a globally-shared, object-based data store. Hence, applications directly use the local disk of the VM instance to share input files and save the output files or intermediate data. In order to balance the load and thus to enable scalability, data is stored in a striped fashion, i.e. split into small chunks that are evenly distributed and replicated among the local disks of the storage (Figure 2.8). Read and write access performance under concurrency is greatly enhanced (up to 5x for writes and up to 3x for reads, compared to Azure Blobs), as the global I/O workload is evenly distributed among the local disks. This scheme reduces latency by enabling data locality and has a potential for high scalability, as a growing number of VMs automatically leads to a larger storage system. A file system atop TomusBlobs adds the hierarchical semantics need by some applications and makes the files from the virtual disk of each compute node available directly to all nodes within the deployment.

NoSQL databases

NoSQL (Not Only SQL) stores [123] were designed to cope with the scale and agility challenges that face modern applications. They address several issues that the relational model is not designed to address: large volumes of rapidly changing structured, semi-structured, and unstructured data; geographically distributed scale-out architecture atop of commodity

storage instead of expensive, monolithic architecture. According to the data access model and the stored data types, we distinguish two categories.

Key-value stores. They are the simplest NoSQL databases. At a high level, key-value stores propose an associative array as the fundamental data model, in which each key is associated with one and only one value in a collection. This relationship is referred to as a *key-value pair*. In each key-value pair the key is represented by an arbitrary string. The value can be any kind of data like an image or a document. The value is opaque to the key-value store requiring no upfront data modelling or schema definition. Key-value stores supports primitive operations and high scalability via keys distribution over servers. While the basic set of operations is roughly similar across systems, no fixed standard exists that would constrain the exact set of features and semantics offered by each system. Consequently, a wide variety of key-value stores exists, each exhibiting potentially a different set of features and semantics suited to different ranges of applications (e.g., variable ACID guarantees enable efficient storage of massive data collections).

Cassandra [121] was designed by Facebook for high scalability on commodity hardware within or across datacenters. It resembles a relational database as it stores data in tables with row index capabilities, but has no support for a full relational model. Cassandra implements customizable consistency levels, by allowing to define the minimum number of nodes that must acknowledge a read or write operation before the operation can be considered successful.

Dynamo [80] is the highly available internal storage for the Amazon web services, optimized for writes and for low latencies. It is organized as a distributed hash table across a ring of virtual nodes, which have a many-to-one mapping with the physical nodes. Key-value pairs are asynchronously replicated and eventually reconciled based on versions. Riak [115] implements the principles of Dynamo in an open-source key-value store, heavily influenced by the CAP theorem.

Azure Tables [96] is a structured key-value store offered by Microsoft as a cloud service. A table contains entities similar to rows in classical relational databases. Properties of the entities are used for load balancing across multiple storage nodes and to optimize consecutive accesses for performance.

The main limitation of key-value stores is the result of their very simple API. Specifically, modifying a key-value pair is only possible by replacing the key as a whole. In practice, this limits key-value stores to two usage scenarios: *small data* pairs or *static content*. Most key-value store are indeed designed to handle relatively small data objects, with a typical size ranging from a few bytes to a few megabytes. This makes key-value stores unfit for a wide range of use-cases, for instance those who frequently need to update potentially large output files from event generators.

Document databases. They pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents. This is a shift from an opaque value to a *structured value*, understandable by the storage system. The latter is hence able to use this structure to extract metadata,

perform optimizations or provide the user with advanced query functionality. For instance, this enables the organization of items in rows and columns, providing partial indexes with respect to their scheme.

BigTable [74] was designed by Google to scale to Petabytes of data and thousands of machines. As for Cassandra, it does not support a full relational model, offering instead dynamic control over layout and format of tuples and advanced locality management. It features compression, in-memory operation, and Bloom filters on a per-column basis. HBase [95] is its open-source implementation, introduced by Yahoo.

MongoDB [75] supports field, range query, and regular expression searches. Fields in a document can be indexed with primary and secondary indices. MongoDB provides strong consistency, with tunable consistency for reads.

The main limitation of document databases lies in the very reason they are popular for web application development: the data structure they expect makes them unfit for handling *large, unstructured data*. These databases trade versatility for expressiveness to various extents, which in practice limits their scope of application. For instance, optimizing for a certain consistency level or availability guarantee for the data, they do not always deliver the highest performance. Moreover, properties such as data locality are not considered or not applicable, specially considering that their initial goal is to scale over large (geographically) distributed infrastructures.

Interestingly, these NoSQL databases can be implemented as thin layers over a simpler, opaque storage layer. This is for example the case of HBase, which offers a wide column store over HDFS or Titan [141] which is implemented as a layer above a pluggable backend such as HBase or Cassandra.

2.3.2 Data transfer

Research on cloud-based BDA has focused so far on optimizing the performance and cost-effectiveness of the computation and storage. It largely neglected an important aspect: data transfers. These are needed to upload/download massive datasets to/from the cloud, to replicate data or to distribute intermediate results between the computing nodes.

The handiest option for transferring data, even across cloud datacenters, is to rely on the *cloud storage service*, which is shared by all application nodes. This approach allows to transfer data between arbitrary endpoints by uploading it from the source to the cloud storage and then downloading it at the destination. It is adopted by several workflow management systems in order to manage data movements over wide-area networks [117, 143]. However, this functionality arises only as a "side effect" of the initial storage goal. Therefore achieving high throughput or any other potential optimization, such as differentiated QoS or cost effectiveness, is difficult. Trying to fill this utility gap, several alternatives were proposed.

GlobusOnline [91] emerged from the team of Ian Foster. It was for long the *de-facto* standard for file transfers over WAN, targeting primary data sharing between different scientific infrastructures. GlobusOnline runs atop GridFTP [36], that was initially developed for grids. It remains unaware of the environment, therefore its transfer optimizations are mostly done statically. Several extensions brought to GridFTP allow users to

enhance transfer performance by tuning some key parameters: threading in [131] or overlays in [114]. Still, these extensions leave the burden of applying the most appropriate settings to scientists, which are often unexperienced users.

StorkCloud [116] is one of the most significant efforts to improve data transfers at scale. It integrates multi-protocols in order to optimize the end-to-end throughput based on a set of parameters and policies (e.g., adapting the parallel transfers based on the cluster link capacity, disk rate and CPU capacity). The communication between StorkCloud components is done using textual data representation, which can artificially increase the traffic for large transfers of unstructured data.

Multi-path TCP [158] is a recent standard designed to address the challenge of moving large volumes of data by enabling parallel data transfers. The idea is to leverage multiple independent routes to simultaneously transfer disjoint chunks of a file to its destination. The approach can incur additional costs such as higher per-packet latency due to timeouts under heavy loads and larger receiving buffers. Nevertheless, this remains an interesting solution for Big Data processing.

These efforts show that the need for efficient tools for transferring data (to/from and across cloud datacenters) is well understood in the community and stands as an important issue. Nevertheless, many aspects remain unaddressed, particularly finding solutions that would provide high-performance for transfers between the running instances of applications on the cloud.

2.4 Discussion: challenges

Porting BDA to the clouds brings forward many issues in exploiting the benefits of current and upcoming infrastructures. We recall here the main challenges identified in this chapter.

Basic API of MapReduce. The very simple model of MapReduce comes with the important caveat that it *forces applications to be expressed in terms of map and reduce functions*. However, most applications do not fit this model and require a more general data orchestration, independent of any programming model. For instance, iterative algorithms used in graph analytics and machine learning, which perform several rounds of computation on the same data, are not well served by the original MapReduce model.

Object storage, the *de-facto* backend for BDA storage. This chapter has revealed that both *NoSQL databases and distributed file systems can be implemented as thin layers over simpler, opaque storage layers, like object stores*. This is consistent with one key observation that can be made when examining the limitations of distributed file systems and key-value stores: these limitations are relative to different aspects of the systems. Indeed, distributed file systems are mostly limited by their hierarchical nature and semantics, that is, the way they manage metadata. In contrast, key-value stores limitations derive from the insufficient set of operations permitted on the data. In this context, object-based storage solutions provide a *middle-ground approach*, combining most of the advantages of both distributed file systems and key-value stores. They leverage the flat namespace and small API that make key-value stores efficient

and highly-scalable, and the update operations that make distributed file systems adaptable to a wide range of BDA use-cases.

Lack of data staging and efficient transfers. An important limitation of the original MapReduce model is the constraint of having *all data available before a job can be started*. This limitation can become a serious problem especially when the input data is massive and needs to be uploaded from an external source, which involves large data transfers over network links of limited capacity. Under such circumstances, even the best MapReduce implementation cannot stop the overall time-to-solution from growing to unacceptable levels. Regarding the support for managing data movements for the applications running on the clouds, things are worse. There is a clear need for developing solutions that would enable efficient data sharing across compute instances, in order to sustain the scalability required to accommodate the Big Data processing

Chapter 3

The World Beyond Batch: Streaming Real-Time Fast Data

Contents

THE DEVELOPMENT OF THE INTERNET OF THINGS AND SOCIAL NETWORKS increases the velocity of data generation. These new data sources are often referred as live sources due to their real-time nature. They produce *unbounded, unordered, global-scale* datasets, called *streams*. Items in a stream are referred to as events or tuples.

Streams require *fast processing* since they typically serve applications trying to solve *mission-critical problems*. Examples include fraud detection, national security, stock market management and customer service optimization. For instance, PayPal managed to save 710 million dollars in their first year thanks to the fast data processing for fraud detection [132].

Monitoring or simulating natural events has the purpose of predicting and minimising disasters (e.g., answering questions like *"where will the hurricane strike?"*). In all these contexts, the results of the processing are needed as fast as possible.

Live data sources and streams are increasingly playing a critical role in BDA for two reasons. First, they introduce an online dimension to data processing, improving the reactivity and "freshness" of the results, which can potentially lead to better insights. Second, processing live data sources can offer a potential solution to deal with the explosion of data sizes, as the data is filtered and aggregated before it gets a chance to accumulate.

Traditional batch-based BDA cannot cope with this online dimension of the processing. In order to keep up with the rate of new data, BDA has shifted towards a new paradigm — *stream computing*. This is capable to deliver insights and results as soon as possible with minimal latency and high throughput. An entire family of Stream Processing Engines (SPEs) was designed and developed to enable this paradigm. In this chapter, we first characterize stream computing and then survey the state-of-the-art SPEs and stream data management.

3.1 Stream computing

Stream computing applies a series of operations to each element in the stream, typically in a pipeline fashion. Even if the analysis can vary in scope across domains, streaming patterns of data processing share several features, that differentiate them from batch processing. Let us discuss them below.

> ❝ *Stream computing applies a series of operations to each element in the stream.* ❞

3.1.1 Unbounded streaming vs. bounded batch

At its basis, a data stream is an infinite set of events or tuples that grows indefinitely in time [133]. For years, the BDA community blindly associated the processing of unbounded datasets with streaming engines and the bounded datasets with batch engines. In reality, things are more subtle, since there is a constant inter-play between the two. Unbounded streams have been initially processed using repeated runs with batch systems (e.g, using micro-batches in Apache Spark [191]). At the same time, well-designed streaming systems are perfectly capable of processing bounded data (e.g., Apache Flink [70]).

One trend that emerges in this context is to avoid transforming unbounded streams into finite datasets that eventually become complete. Instead accept that *we will never know if or when we have seen all of our data, only that new data will arrive*. This principle is the very essence of stream computing and the main challenge making this paradigm non-trivial. The way to turn it tractable is via principled abstractions that allow to choose the appropriate tradeoffs along the axes of interest: correctness, latency, and cost [33].

3.1.2 Windowing

One such abstraction is windowing — dividing an infinite data stream into finite slices called windows [126]. The division is done using event timestamps or other attributes. Windows allow to process events as a group (e.g., aggregations, outer joins, time-bounded operations).

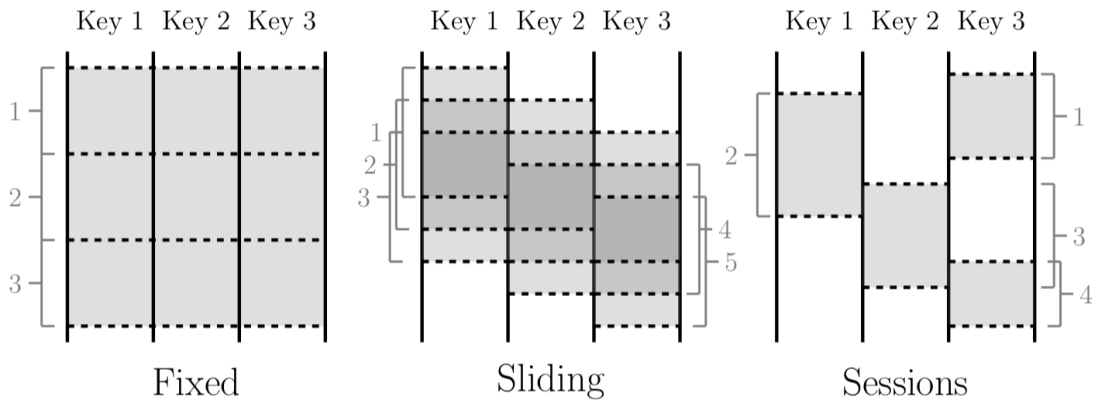


Figure 3.1 – Windows patterns for several streams (one per key). Session are unaligned: Window 2 applies to Key 1 only [33].

According to how the elements of a stream are divided into windows, the latter fall into 3 main categories, illustrated in Figure 3.1:

Fixed (tumbling) windows. Elements are assigned to fixed length, non-overlapping windows of a specified *size* (e.g., hourly windows or daily windows). They are generally *aligned*, i.e., every window applies across all of the data for the corresponding *period* of time. Any element belongs to only one window.

Sliding windows. Elements are assigned to overlapping windows of fixed size with the size of the overlap defined by the slide period (e.g., hourly windows starting every minute). Sliding windows are also typically *aligned*. Even though the diagram is drawn to give a sense of sliding motion, all five windows would be applied to all three keys in the diagram. An element can belong to more than one sliding window. Fixed windows are really a special case of sliding windows where *size* equals *period*.

Session windows. They are defined by features of the data themselves (e.g., per key) and window boundaries are adjusting to incoming data. By definition, session windows are *unaligned*, i.e., applied across only specific subsets of the data for a given timeout.

Let us note that windowing is not always necessary (i.e., for filtering, mapping, inner joins, etc. one does not need to operate on finite chunks). Yet when it is needed, it has a significant impact on performance as we will see in the following section.

3.1.3 State management

Any application that processes a stream of events and does not just perform trivial record-at-a-time transformations needs to be stateful, i.e., *have the ability to store and access input and intermediate data* for fault tolerance purposes. Not surprisingly, state and windowing are closely linked. The functions that are applied over the window contents are quite generic and range from mathematical functions (e.g., min, max, summations, aggregations, metrics

over partitions) to extracting data features for machine learning (e.g., statistics, histograms). They all require buffering the data (i.e., the state) over some periods of time. State is usually stored as a list structure in heap memory or off-heap in an embedded key-value store. The implementation can also be hybrid, with references (hash keys) of events stored in heap memory and actual values stored in an external key-value store.

The state management becomes problematic in the often case of operators that need to work in parallel on the same data. For instance, computing the top-K and bottom-K entries observed during the last hour in a stream of integers. Current state-of-art approaches create data copies that enable each operator to work in isolation. This leads to increased memory utilization. For instance, in several gaming specific scenarios [69] terabytes of state are generated by billions of events processed in parallel. Thus, the problem of minimizing memory utilization without significant impact on the performance (typically measured as result latency) is crucial.

Currently, SPEs that support state through an external storage system (e.g., Akka Streams [34]) are generally slower than SPEs that internally embed and manage state (e.g., Apache Flink). This is due to the fact that the addition of another database system for state management introduces significant overhead, since most operations in stream processing involve reading or writing persistent data on a regular basis. Clearly, when designing efficient storage for SPEs, state has to be taken into account.

3.1.4 Correctness

Correctness boils down to remaining consistent in light of machine failures. There are three correctness semantics in stream processing:

At-most-once: an event is processed zero or one time, meaning it can be lost. This is acceptable in many cases, especially in situations where the occasional lose of a message does not leave the system in an inconsistent state.

At-least-once: potentially multiple attempts are made at processing an event, such that at least one succeeds. This means that events may be duplicated but not lost.

Exactly-once: each incoming event affects the final results exactly once. Even in case of a machine or software failure, there is no duplicate data and no data that goes unprocessed.

At-most-one semantics is the cheapest — highest performance, least implementation overhead — because it can be done in a fire-and-forget fashion without keeping state. At-least-once correctness requires retries to counter transport losses, which means keeping state at the sending end and having an acknowledgement mechanism at the receiving end. Exactly-once semantics is most expensive because in addition to the second it requires state to be kept at the receiving end in order to filter out duplicate deliveries. This is the most desirable guarantee.

3.2 Fast Data processing frameworks

SPEs were specifically designed to operate on continuous, unbounded data streams. All engines implement a driver program that describes the high-level control flow of the application by means of a directed acyclic graph. This relies on two main abstractions: (i) parallel operations on the data (the nodes) and (ii) structures to describe the data flow (the directed edges).

The main differentiator between these SPEs arises from the observation that one can never fully optimize along all dimensions of correctness, latency, and cost. As a result, several systems emerged, reconciling in different ways the tensions between these seemingly competing axes. Among them, Apache Spark and Flink are the most notable examples.

3.2.1 Micro-batching with Apache Spark

Spark [191] is the first "post-MapReduce" framework for BDA, originally developed at the University of California, Berkeley's AMPLab, by the team of Ion Stoica. Central to Spark is the principle of *treating unbounded computation as micro-batches*. The idea is very simple: a continuous computation is broken down in a series of small, atomic batch jobs, called *micro-batches*. The goal is to overcome the complexity and overhead of record-level synchronization imposed by continuous operators. Each micro-batch may either succeed or fail. At a failure, the latest micro-batch can be simply recomputed.

This model generated a lot of debates in the community on the true streaming nature of the framework.

Critics argue that batching is artificially grouping records into static datasets (e.g., hourly, daily, or monthly chunks) that are processed in a time-agnostic fashion. Detecting missing data or data arriving out of time order becomes impossible or at best expensive. Also, the batch length restricts window-based analytics.

Supporters of the model show that in practice this is not problem, because the batches are as short as 0.5 seconds. In most applications, the latency to get the data in is much higher (e.g., sensors sending in data every 10 seconds) or the targeted analysis is over a longer window (e.g., track events over the past 10 minutes).

The true benefit of Spark's micro-batch model is full fault-tolerance and "exactly-once" processing for the entire computation, meaning it can recover all state and results even if a node crashes. This is not the case for other "true streaming" engines like Storm, which require application developers to worry about missing data or to treat the streaming results as potentially incorrect.

Spark introduces a new abstraction, Resilient Distributed Datasets (RDDs), which represent the input data partitions that are distributed across the cluster. They are read-only and can be rebuilt in case of failures by partial recomputation from ancestor RDDs. Two types of RDD operations are transformations and actions. Transformation produce new RDDs from the existing ones (e.g., map, filter) while actions return final results of RDD computations (e.g., reduce, collect) or persist data (e.g., save). Transformations are computed lazily, meaning that Spark adds them to a DAG of computation and only when the driver program

requests some data (i.e., an action is executed), does this DAG actually gets executed. This enables Spark to make many optimization decisions by looking at the DAG in entirety.

As Hadoop, Spark relies on a distributed storage system (e.g., HDFS) to store the input and output data of the jobs submitted by users. However, unlike Hadoop, Spark allows RDDs to be cached in the memory and therefore intermediate data between different iterations of a job can be reused efficiently. This reduces the number of costly disk I/O accesses to the distributed storage system. This memory-resident feature of Spark is particularly essential for some Big Data applications such as iterative machine learning algorithms which intensively reuse the results across multiple iterations.

3.2.2 True streaming with Apache Flink

Flink [70] is the European counterpart of Spark, originally developed at TU Berlin, and advocated as the first *true streaming* processing engine (i.e., treating streams as streams, not as batches). Every incoming record is processed as soon as it arrives, without waiting for others. There are some continuous running operators which run for ever and every record passes through them to get processed.

The state is managed consistently with "exactly-once" guarantees. This boils down to determining what state the streaming computation currently is in, drawing a consistent snapshot of that state, storing it in durable storage, and doing this frequently. Flink's snapshot algorithm is based on a technique introduced in 1985 by Chandy and Lamport. Similar to the micro-batching approach, all computations between two checkpoints either succeed or fail atomically as a whole. However, one great feature of Chandy Lamport is that one never has to press the "pause" button in stream processing to schedule the next micro batch (i.e., data processing keeps going, while checkpoints happen in the background).

Flink executes iterations as cyclic data flows. This means that a data flow program (and all its operators) is scheduled just once and the data is fed back from the tail of an iteration to its head. Basically, data is flowing in cycles around the operators within an iteration. Since operators are just scheduled once, they can maintain a state over all iterations. There are two types of iterations: (i) bulk iterations, which are conceptually similar to loop unrolling, and (ii) delta iterations, a special case of incremental iterations in which the solution set is modified by the step function instead of a full recomputation. In contrast, Spark implements iterations as regular for-loops and executes them by loop unrolling. This means that for each iteration a new set of tasks/operators is scheduled and executed. Each iteration operates on the result of the previous iteration which is held in memory.

3.2.3 Performance comparison of Spark and Flink

We ran a series of experiments [5] to investigate *the impact of these different architectural choices and the parameter configurations on the perceived end-to-end performance*. A bit surprisingly, at the time of this evaluation, most of the studies assessing the performance of Spark or Flink benchmarked them against Hadoop, as a baseline. This was a rather unfair comparison considering the fundamentally different design principles (i.e., in-memory vs. on-disk processing). The following evaluation aimed to bring some justice in this respect, by *directly comparing for the first time the performance of Spark and Flink*.

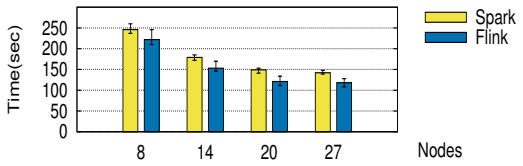


Figure 3.2 – Connected Components - Small Graph (increasing cluster size).

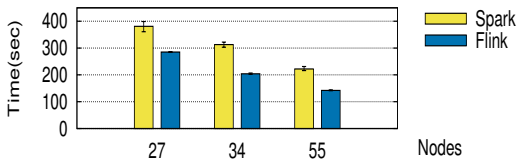


Figure 3.3 – Connected Components - Medium Graph (increasing cluster size).

We devised a methodology for performance analysis by means of *correlations between the operators execution plan and the resource utilisation*. We used this methodology to study the performance (i.e., end-to-end execution time) of several representative benchmarks (word count, grep, terasort, page rank, k-means, connected components) on up to 100 nodes. For the sake of brevity, we recall here just a summary of the insights. We illustrate them with the performance figures of the connected components benchmark (Figures 3.2, 3.3 and 3.4).

Our key finding is that *there is not a single framework best suited for all data types, sizes and job patterns.* Spark is about 1.7x faster than Flink for large graph processing while the latter outperforms Spark up to 1.5x for batch and small graph workloads using sensitively less resources and being less tedious to configure. There are significant differences in configuring Flink and Spark, in terms of ease of tuning and the control that is granted over the framework and the underlying resources. We have identified a set of 4 most important parameters having a major influence on the overall execution time, scalability and resource consumption. They manage the *task parallelism, the network behaviour during the shuffle phase, the memory management* and the *data serialization*.

Memory management plays a crucial role in the execution of any workload, particularly for huge datasets. We noticed that, as opposed to Spark, Flink does not accumulate lots of objects on the heap but stores them in a dedicated memory region, to avoid overallocation and the garbage collection issues. All operators are implemented in such a way that they can cope with very little memory and can spill to disk. Although Spark can serialize data to disk, it requires that (significant) parts of the data to be on the JVM heap for several operations. If the size of the heap is not sufficient, the job dies. Spark has started to catch up on these memory issues with its Tungsten [157] project. It uses explicit memory management to eliminate the overhead of the JVM object model and garbage collection.

The pipelined execution brings important benefits to Flink, compared to the staged one in Spark. Especially for the batch workloads, reordering the operators enables more efficient resource usage and drastically reduces the execution time. There are several issues related to the pipeline fault tolerance, but Flink is currently working in this direction [89]. For more complex batch workflows (e.g., with multiple filter layers applied on the same dataset), Spark can take more advantage of its persistence control over the RDDs (disk or memory) and further reduce the execution times.

Optimizations are automatically built-in Flink. Spark batch and iterative jobs have to be manually optimized and adapted to specific datasets through fine grain control of partitioning and caching. For SQL jobs however, SparkSQL [50] uses an optimizer that supports both rule- and cost-based optimizations.

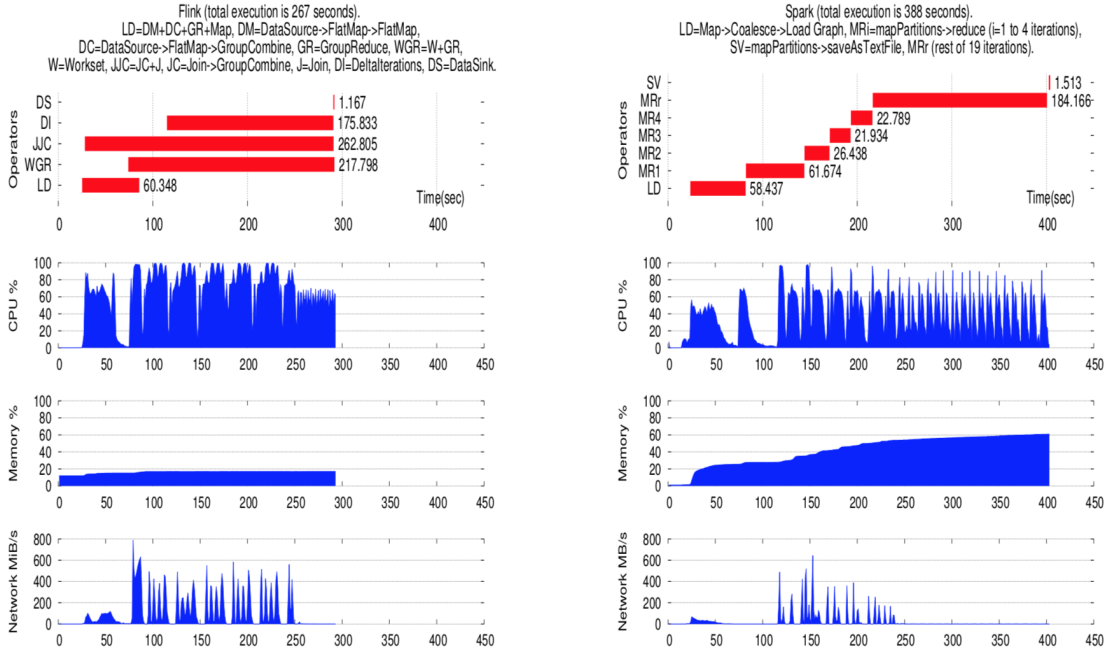


Figure 3.4 – Connected Components resource usage of Flink and Spark for 27 nodes, 23 iterations, Medium Graph.

Parameter configuration proves tedious in Spark, with various mandatory settings related to the management of the RDDs (e.g., partitioning, persistence). Flink requires less configuration for the memory thresholds, parallelism and network buffers, and none for its serialization (as it handles its own type extraction and data representation). For workloads consisting of one stage to prepare the graph (load edges) and another one to execute a number of iterations, an optimal performance can be obtained by configuring the parallelism setting of the operators separately for each stage. In this case, Flink's delta operator can drastically improve performance over Spark.

Contrary to our expectations, we find that Flink performs better for each analyzed workload, except for the large graphs. We emphasize that we do not claim that Flink is faster than Spark. Instead we show that for a number of reasonable workloads it is possible to achieve comparable or better performance without the pain of tuning RDDs in order to control the partitioning or persistence. However, we argue that general data analytics systems should provide also the flexibility of a shared memory abstraction like RDDs, offering fine-grained control over the data partitioning and persistence of intermediate data.

3.2.4 Other frameworks

Alternative approaches for stream processing tend to trade different performance dimensions for the sake of some application specific optimisations (i.e., enabling horizontal scalability with weaker state consistency guarantees). We discuss below the most notable examples and list in Table 3.1 a summary of their features.

	Storm	Spark	Dataflow	Flink
Guarantee	At least once	Exactly once	Exactly once	Exactly once
Latency	Very Low	High	Low (delay of transaction)	Very Low
Throughput	Low	High	Medium to High (Depends on throughput of distributed transactional store)	High
Computation model	Streaming	Micro-batch	Streaming	Streaming
Overhead of fault tolerance	High	Low	Depends on throughput of distributed transactional store	Low
Flow control	Problematic	Problematic	Natural	Natural
Separation of application logic from fault tolerance	Partially (timeouts matter)	No (micro batch size affects semantics)	Yes	Yes

Table 3.1 – Overview of Fasta Data processing engines [182].

Storm [51] from Twitter was perhaps the first widely used large-scale stream processing framework in the open source world. The system leverages the "at-least-once" or "at-most-once" processing semantic guarantees. It uses a mechanism of upstream backup and record acknowledgements to guarantee that messages are re-processed after a failure. This approach does not guarantee state consistency, any mutable state handling is delegated to the user. This may be acceptable for some applications, but is not for many others. Also, Storm faces low throughput and problems with flow control, as the acknowledgment mechanism often falsely classifies failures under backpressure (i.e., accumulation of data behind a buffer during slow processing).

Dataflow [33] from Google implements "exactly-once" low latency stream processing and "out-of-order" processing, being actually very influential to the evolution of Flink. To do so, Dataflow atomically logs record deliveries together with updates to the state. Upon failure, state and record deliveries are repeated from the log. Thus, it supports the unaligned, event-time-ordered windows modern applications require. Dataflow also abstracts away the distinction of batch vs. micro-batch vs. streaming, allowing pipeline builders a more fluid choice between them.

3.3 Fast Data management

In order to take advantage of the push-based nature of streaming, SPEs need dedicated solutions for stream transport and storage. In this context, a typical state-of-art online BDA stack is completed with two additional layers on top of the processing one:

Ingestion: this layer serves to acquire, buffer and optionally pre-process data streams (e.g., filter) before they are consumed by the analytics application. The ingestion layer has limited support for guaranteeing persistence. It buffers streams only temporarily (e.g., hours, days) and enables limited access semantics to them (e.g., it assumes a producer-consumer streaming pattern that is not optimized for random access).

Storage: this layer is responsible for persistent storage of data (i.e., objects or files). This typically involves either the archival of the buffered data streams from the ingestion layer or the storage of the intermediate results of stream analytics, both of which are crucial to enable fault tolerance or deeper, batch-oriented analytics that complement the online analytics.

3.3.1 Data ingestion

Ingestion systems collect data from distributed sources, queue it, order it, optionally clean it and route it to the processing sites as efficiently and correctly as possible. While time-series order is naturally maintained with respect to each data source, global ordering can be much trickier in the presence of thousands of streams.

This is typically achieved via *publish-subscribe messaging* frameworks that leverage brokering solutions to decouple data sources from applications.

Apache Kafka [44] makes stream data available to multiple consumers through a publish/-subscribe functionality. It is the de-facto standard for ingestion in pipelines with streaming engines like Spark or Flink. A Kafka cluster comprises a set of broker nodes that store streams of records in categories called topics. Each topic (stream) can be split into multiple partitions, which enable parallel access. However, the *static number of partitions* per stream (similar to the static number of mappers in Hadoop) and the fact that *each consumer is associated with one partition* leads to overprovisioning [6]. This prevents elasticity and scalability. Moreover, there is *no support to search* a record by its key or other attributes, only by its offset, meaning this logic has to be pushed into the application.

Apache Pulsar [49] is a publish/subscribe messaging system with a two-layer architecture composed of a stateless serving layer and a stateful persistence layer. Reads and writes cannot scale independently (first layer is shared by both readers and writers). Pulsar unifies the queue and topic models, providing exclusive, shared and failover subscriptions models to its clients.

Pravega [156] partitions a stream in a fixed number of partitions called segments with a single layer of brokers providing access to data. It provides support for auto-scaling the number of segments (partitions) in a stream and based on monitoring input load (size or number of events) it can merge two segments or create new ones. Producers can only partition a stream by a record's key.

These systems do not offer support for fine-grained record access and employ a static partitioning model with no support for data locality.

3.3.2 Data storage

Both data ingestion and processing layers need to maintain local storage for two main purposes:

(i) **temporary storage** for staging of new data and handling intermediate results. For example, a large number of streaming time-series from distributed sources may need to be buffered by the ingestion phase to ensure their correct temporal ordering and alignment.

(ii) **long-term storage** for persistent data. For instance, datasets that would be later used for further batch processing.

State-of-the-art stream storage systems typically leverage *log-based storage and databases* or a hybrid combination of these.

Redis [163] is an in-memory store that is used as a database, cache and message broker. Redis supports many data structures such as strings, lists, hashes, sets, sorted sets, bitmaps and geospatial indexes. Redis implements the pub-sub messaging paradigm and groups messages into channels with subscribers expressing interest into one or more channels. Redis implements persistence by taking snapshots of data on disk, but it does not offer strong consistency.

RAMCloud[151] is an in-memory key-value store that aims for low-latency reads and writes, by leveraging high performance Infiniband-like networks. Durability and availability are guaranteed by replicating data to remote disks on servers relying on batteries. Among its features we can name fast crash recovery, efficient memory usage and strong consistency. Recently it was enhanced with multiple secondary indexes, achieving high availability by distributing indexes independently from their objects (independent partitioning).

DistributedLog [103] is a strictly ordered, geo-replicated log service, designed with a two-layer architecture that allows reads and writes to be scaled independently. DistributedLog is used for building different messaging systems (e.g., Pulsar). A topic is split into a fixed number of partitions, and each partition is backed by a log. A read proxy mechanism optimizes the reader path by caching log records for the case multiple consumer groups are reading from the same stream.

Druid [190] is an open-source, distributed, columnar-oriented data store for real-time exploratory analytics. Druid's data model is based on data items with timestamps (e.g., network event logs). As such, Druid requires a timestamp column in order to partition data and supports low latency queries on particular ranges of time. Druid uses specialized nodes: real-time nodes (that maintain an in-memory index buffer for all incoming events, regularly persisted to disk) and historical nodes (that handle immutable blocks of data which are created by real-time nodes).

Apache Kudu [45] is a columnar data store with fast scans and low-latency random updates. It is a good alternative to systems like Avro/Parquet over HDFS that cannot handle mutable datasets, or to semi-structured stores like HBase or Cassandra, that are not efficient for sequential reads needed in machine learning or SQL.

S-Store [136] is a new streaming database system built explicitly to handle shared, mutable state. Unlike traditional streaming systems, S-Store models the processing graphs as a series of transactions, each of which ensure a consistent view of the modified state upon commit. It provides ACID transactions and "exactly-once" processing.

Redis and RAMCloud primarly optimize for low-latency fine-grained record access, with limited support for high throughput. Also, Redis does not offer strong consistency, which is often required for stream processing. Druid and Kudu offer specialized support for querying streams, but rely on static stream partitioning and do not offer support for objects (i.e., bounded streams). In particular, Druid's choice to differentiate real-time and historical nodes favors proliferation of data copies.

3.4 Discussion: challenges

The future of data processing is Fast (unbounded) Data. Big (bounded) Data will always have an important role especially for retrospective analysis, yet it is semantically subsumed by its unbounded counterpart. Eventually, processing engines will be able to analyse both seamlessly and extract insights in a timely fashion while avoiding excessive data accumulation by means of states. The models and systems that exist today serve as an excellent foundation in this direction. At the same time, several axes remain problematic.

Storage and ingestion separation. Current streaming architectures are designed with distinct components for ingestion and storage of stream data. Unfortunately, this separation prevents data locality and, worse, can become an overhead especially when data needs to be archived for later analysis. In such cases, *stream data has to be written twice* to disk and may pass twice over high latency networks: once persisted by the ingestion system and once by the processing engine. Also, the lack of coordination between these two layers can lead to *I/O interference*. The ingestion layer and the storage layer compete for the same I/O resources, when collecting data streams and writing archival data simultaneously.

Increased memory utilization. In heavily concurrent scenarios, SPEs favor operators working in isolation by creating data copies, at the expense of increased memory utilization. Given the scarcity of memory due to increasing application complexity and decreasing memory per core (a trend of modern multi-core architectures), the problem of optimizing memory utilization for stream processing becomes critical. One idea would be to study the feasibility of deduplicating the shared data across the operator states.

Fault tolerance at scale. As a general observation, SPEs that provide scalability and fault-tolerance fall short on expressiveness or correctness vectors. Many lack the ability to provide "exactly-once" semantics (e.g., Storm, Samza, Pulsar). Others simply lack the temporal primitives necessary for windowing, or provide windowing semantics that are limited to tuple-based windows (e.g., Spark).

Chapter 4

The Lambda Architecture: Unified Stream and Batch Processing

Contents

STREAM AND BATCH PROCESSING ARE TWO SEPARATE PARADIGMS. They are programmed using different models and APIs, executed by different systems and used by different applications. Batch processing is typically used for analyzing huge logs while stream processing enables monitoring and content delivery. The data that accumulates for batch processing from different sources is denoted as *historical (past) data* or *data at rest*. At the same time, an up-to-date vision of the actual status of those sources is the *real-time (present) data* or *data in motion*, collected for stream processing.

This separation that was historically perpetuated brings partial and isolated insights, that are not correlated. For instance, analyzing with stream processing real-time data from a temperature sensor one might notice (too late) a threshold exceeded. This could have been easily anticipated using batch processing on the recent history data from that sensor. One alternative is to *leverage past (historical) data to interpret present (real-time) data* in order to better and faster understand the status of the data source (i.e., a monitored system) and to eventually predict its future evolution.

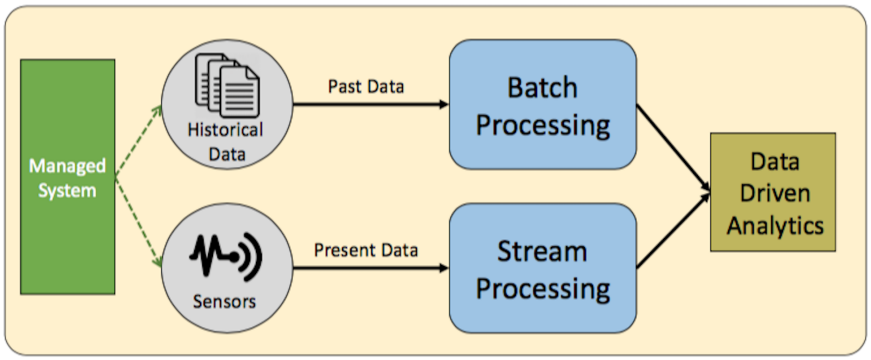


Figure 4.1 – The unified batch and stream computing paths in the Lambda architecture.

> ❝ *The* Lambda architecture *is a recent paradigm that enables this metaphor by* combining batch and stream processing . ❞

The *Lambda architecture* is a recent paradigm that enables this metaphor by *combining batch and stream processing* to implement multiple paths of computations [135]. A streaming fast path is used for timely, approximate results. Batch processing is used to complement this online dimension with a machine/deep learning dimension and gain more insights based on historical data (e.g., discover new correlations and patterns). Ultimately, this approach enables the fast streaming path to detect *what* is happening with a monitored object, while the batching path helps to understand *why* this is happening.

4.1 Unified processing model

The model proposed by the Lambda architectures attempts to balance latency, throughput, and fault-tolerance of the two worlds in order to obtain new views of the data. Essentially, the architecture relies on two computing layers, as depicted in Figure 4.1.

The stream (speed) layer deals with Fast (recent) data only. It uses real-time stream processing to provide views of online data and timely results. These may be approximate since they are computed on subsets of the whole data (i.e., the sensors from a subregion). Its secondary role is to compensate for the high latency of the adjacent batch layer.

The batch (slow) layer manages the Big (historical) data, typically an immutable, append-only set of raw data. It computes comprehensive, accurate but late results. Naturally, batch processes occur on some interval and are long-lived. The scope of data is anywhere from hours to years. The goal of this layer is to complement the real-time results with some data-driven analytics that allow to understand the system's behaviour and its rationale from past data, using machine learning/deep learning techniques. In some deployments, the batch layer is supported by a *serving layer*, which indexes batch views so that they can be queried in a low-latency, ad-hoc way.

With this model, data analytics may get a complete picture by retrieving data from both

historical and real-time views - the best of both worlds. As time goes on, real-time data are moved to the historical dataset. This slow path is also useful to replay the same data and produce new views when analytics are updated. On long term, the batch layer will only be used for off-line back-end processing, mainly dealing with archival and further processing that is not time-critical and fault tolerance.

Speaking of fault tolerance, such architectures require *support for persistence* in order to ensure correctness and consistency. In general, one (or more) stream store(s) is (are) used to capture a picture of the targeted system that is as accurate as possible. If correctness criteria are not met, the contents of the analytics system can drift arbitrarily far from the true state of the targeted system.

Currently, both Flink and Spark can implement Lambda architectures seamlessly. In Spark, the unified API for batch and streaming (using the mini-batch model) allows to easily translate batch jobs to streaming jobs or to join streaming data with historical data from batch. In Flink, stream processing is the unifying model for real-time and batch analysis both in the programming model and in the execution engine. Basically, there is no distinction between processing the latest events in real-time, continuously aggregating data periodically in large windows, or processing terabytes of historical data.

4.1.1 The case for batch-processing

With the advent of Lambda architectures, many rushed to call the end of the age of offline only Big Data analytics. Still, the interplay between offline and online analytics put into light by these very architecture proves there is, and will be, a need for dedicated batch processing (dealing with static data sets).

Complex queries over static data are still a good match for a batch processing abstraction. Furthermore, batch processing is still needed both for legacy implementations of streaming use cases, and for analysis applications where no efficient algorithms are yet known to perform this kind of processing on streaming data. Batch programs can be seen as special cases of streaming programs, where the stream is finite, and the order and time of records does not matter (all records implicitly belong to one all-encompassing window).

Flink, for instance, supports batch use cases with competitive ease and performance through a specialized API. It leverages static data sets, specialized data structures and algorithms for the batch versions of operators like join or grouping, and dedicated scheduling strategies.

4.2 Limitations of the Lambda architecture

Lambda architecture systems can achieve many of the desired requirements for Fast Data processing (i.e., correctness, low-latency), but suffer from several limitations that we sum up in this section.

4.2.1 High complexity of two separate computing paths

The biggest detraction to this architecture has been the need to maintain two distinct, complex systems for the batch and speed layers. The model fails on the simplicity axis on account

of connecting and orchestrating several systems, and implementing processing logic twice.

Luckily with Spark and Flink one can abstract the underlying batch and streaming systems behind a single interface (as seen in the previous section), although the operational burden still exists. Even so, at infrastructure level, one of the main challenges is to integrate the different kinds of data (past and present), considering the differences in velocity, volume and format as well as the heterogeneity.

Separate storage stacks for stream and batch. Although a single interface may be used for processing purposes, currently, in the Lambda architecture, storage layers are still distinct for stream and batch. This design has several drawbacks: data is often written or sent twice to disk or over the network; there is a lack of coordination between the stream and batch layers, which can lead to I/O interference; increased overhead of the custom data management tools leveraging the two storage stacks at the processing layer.

4.2.2 Lack of support for global transactions

The processes executed within Lambda systems are fundamentally concerned with the creation of *state*. Incoming tuples are potentially duplicated along the two layers. Thus, engines process multiple streams at once or multiple copies of the same stream. Each processing instance may try to make modifications to the same state simultaneously (i.e., *state can be seen as metadata for stream processing*). In all of these cases, data isolation is necessary to ensure that any changes do not conflict with one another. Transactions are crucial for maintaining correctness of the state.

Absence of global synchronisation. Although transaction support exists within each dimension (i.e., for streaming and batching), there is no global synchronisation between the two layers. That is because the mechanisms implemented in SPEs for the two models are distinct: distributed snapshotting for streaming and micro-batch replay for batching.

4.3 Research agenda

The challenges of Big and Fast data management identified in the chapters 2, 3 and 4 provide a roadmap for what is expected from next-generation streaming architectures. We firmly believe they will shift the overall mindset of reasoning about unbounded data. Conjointly, these challenges also set the agenda for our contributions. The rest of this manuscript aims to address some of them by combining ideas from existing approaches as well as proposing novel data management techniques designed from scratch.

In-transit stream processing: bringing intelligence to storage. Historically, storage devices have been performing the same basic functions: read and write. This means that stored stream data (e.g., collected by the ingestion layer) needs to be further shipped to some processing nodes. This greatly increases the time to results. We believe it is time to move beyond this prevailing mindset and generalize to stream processing the locality principle first introduced in HDFS. Executing computations near the data minimizes network traffic and increases the throughput. One way of achieving this is by executing some (simple) parts of the computation in the storage nodes, anywhere in-transit from the stream collection sites to

the processing nodes. Such "smart" storage just needs to leverage some dedicated nodes in the deployment to dial in precisely the amount of latency and locality for a large spectrum of computations, as we will show in chapter 5.

Efficient stream data transfer. We have noticed that a major shortcoming of all the models and systems previously mentioned is that they focus only on optimizing the performance and cost-effectiveness of the storage and computations. They largely neglect data transfers to/from/between geographically distributed processing sites. Existing solutions remain basic, far behind the advance in processing, mainly tributary to the decade old multi-hop path splitting, or, worse, direct TCP connections. JetStream is our proposal to alleviate from these issues. It is the first system targeting fast transfer of unbounded data coping with the timeliness demanded by new BDA systems (chapter 6).

Distributed metadata management. Since streaming deals with large volumes of small data, the associated metadata can quickly reach huge sizes. Many storage systems presented in the previous chapters are in the same situation as HDFS, relying on centralized metadata management (e.g., Lustre, GoogleFS). An important challenge that arises in this context is to find scalable ways of managing the metadata in largely distributed environements, as the ones in which streams operate. Decentralization seems to promise a lot of potential in this direction, but introduces metadata consistency issues, as we will see in chapter 7.

Unification of stream storage and ingestion enables data locality. Our proposal, KerA, unifies ingestion and storage, in a stream ingestion solution that offers interfaces for both bounded and unbounded data. Designed with data locality support in mind, it avoids redundant data copies. KerA supports massively parallel stream processing, by "freeing" the processing engine from the data housekeeping tasks and by allowing it to focus on computations only. KerA is described in chapter 8

Transactional access to stream storage. We have seen that in a concurrent streaming setting, there is a need to support shared in-memory storage. When multiple clients access this storage, transactions are needed to ensure consistent views of the most recent data for real-time analytics. This transactional support is currently missing or comes at the cost of performance. We have addressed this issue by introducing Týr, the first blob storage system to provide built-in, multi-blob transactions, while retaining sequential consistency and high throughput under heavy access concurrency (chapter 9).

Storage support for HPC and Big Data convergence. Our vision of future analytics combines processing of past and real-time data from BDA with future (simulated) data from HPC. The convergence of HPC and Big Data enables this vision. While the convergence has already started at the infrastructure level (e.g., HPC clouds), little progress was achieved to converge the processing stacks. We believe that a first step in this direction is by means of a unified storage layer, based on Týr, as explained in chapter 10.

Part II

From Sensors to the Cloud: Stream Data Collection and Pre-processing

Chapter 5

DataSteward: Using Dedicated Nodes for In-Transit Storage and Processing

Contents

THE HPC COMMUNITY INTRODUCED IN-SITU AND IN-TRANSIT PROCESSING [60] to allow data to be visualized and processed asynchronously in real-time as it is produced, while the HPC application (e.g., a simulation) is running. Their difference lies in how and where the computation is performed. In-situ processing typically shares the primary simulation compute resources. In contrast, when (more complex) analytics are performed in-transit, some or all of the data are transferred to different, dedicated machines, either on the same infrastructure (e.g., a supercomputer) or on different computing resources all together. This brings the possibility to build early knowledge on the simulation results and even react back and steer the simulation accordingly.

We think that a similar approach would greatly benefit to stream processing. In the traditional streaming pipeline, data collected from thousands of sources is shipped to the

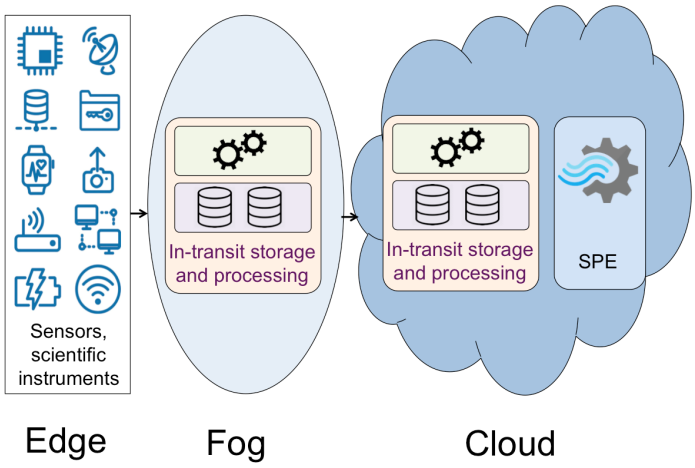


Figure 5.1 – Stream in-transit processing takes place either in the fog or in the cloud.

processing platforms (i.e., cloud datacenters) for analysis. This transfer is very costly, and no output is available till the end of the processing. *In-transit* processing would mean to dedicate a set of nodes from the allocated deployment where to:

- *compute* fast some early results, although maybe inaccurate, by means of user-defined functions. This computation would take place before or during the transfer of the data (from the sensors, scientific applications etc.) to the processing site.
- *store* these results and incoming data for persistence and flow control.

> ❝ *In-transit stream processing avoids routing large amounts of data over the network (but only processed results) and minimizes the time to solution.* ❞

This approach builds on the observation that the highest concentrations of computing power and storage are in the "center" of the networks (i.e., in clouds or HPC centers). However, most of the streams originate in edge environments (see Figure 1.1). In this context, in-transit stream processing avoids routing large amounts of data over the network (but only processed results) and minimizes the time to solution. As a side effect, it ensures all components along the data path are utilized in a balanced way, by overlapping data processing with computation. In this chapter we introduce DataSteward [18, 17], our approach for providing such *in-transit* storage and processing on dedicated compute nodes.

5.1 A storage service on dedicated compute nodes

In the case of streams, there are two options for hosting *in-transit* storage and processing, illustrated in Figure 5.1. The first is to exploit the computing infrastructure available close to the edge of the network, where the data is produced. This is referred in the literature as "fog computing". Examples include the computing and storage resources in the antennas, routers or edge devices of the Internet providers. The second option is to use some nodes in the cloud, close to where the SPEs are deployed.

In both cases, in order to enable *in-transit* processing, we first need a storage service to buffer data and control the flow In a second phase, we will enhance this storage service with data analytics capabilities by allowing users to push in-storage processing functions.

5.1.1 Design principles

The design of the fog and cloud storage service is guided by the following principles.

Dedicated nodes for storing data. We plan to isolate the storage from the computation in order to avoid intrusiveness. Hence, we opt to use dedicated nodes, separated from the ones used for computations (in the case of clouds) or for data collection (in the case of fog). This approach preserves the data proximity within the deployment and increases the application reliability through isolation. The goal is to have high data access performance while freeing application nodes from managing data. Moreover, keeping the management of data within the same compute infrastructures (i.e., same racks, switches) optimizes the utilization of the cluster bandwidth by minimizing the intermediate routing layers to the application nodes.

Topology estimation. Cloud and fog applications operate in a virtualized space which hides the infrastructure details. However, our proposed solution needs some mechanism to ensure that the dedicated storage servers are located as "close" as possible to the rest of the computing nodes (in the case of clouds) or the sensors (in the case of fog). This "closeness" is then defined in terms of bandwidth and latency and determines the communication efficiency. As information about the topology is not available, our solution estimates it, based on set of performance metrics, in order to perform an environment-aware selection of nodes.

5.1.2 Architectural overview

DataSteward is designed as a multi-layered architecture, shown in Figure 5.2.

The Cloud Tracker selects the nodes to be dedicated for data management. The selection process is done once, in 4 steps, at the starting of the deployment. First, a leader election algorithm is run, based on the VM IDs. Second, the trackers within each VM collaboratively evaluate the network links between all VMs and report the results back to the leader. Third, the leader runs the clustering algorithm described in section 5.1.3 to select the most fitted, throughput-wise, nodes for storage. Finally, the selection of the nodes is broadcast to all compute nodes within the deployment. The Cloud Trackers evaluate the network capacities of the links, by measuring their throughput, using the `iperf` tool [181].

The Distributed Storage is the data management system deployed on the dedicated nodes, that federates their local disks. Users can select the distributed storage system of their choice, for instance any distributed file system that can be deployed in a virtualized environment. The local memory of the dedicated nodes is aggregated into an *in-memory*

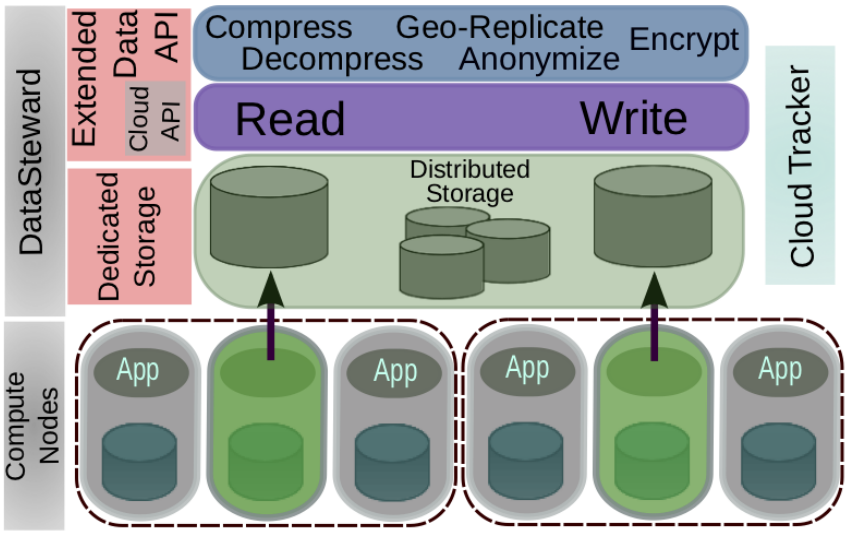


Figure 5.2 – DataSteward overview. Compute nodes are dedicated for data storage, on top of which a set of data processing services are provided.

storage, used for storing, caching and buffering data. The Distributed Storage can dynamically scale up and down, dedicating new nodes when faced with bursts or releasing some of the existing ones.

The Data Processing Services are a set of advanced data handling operations, provided by DataSteward and targeting scientific applications. They enable the *processing* capabilities of DataSteward. The goal is to capitalize the computation power now available for data management, in order to provide high-level data functions. These processing services are further detailed in section 5.2.

5.1.3 Zoom on the dedicated nodes selection in the cloud

One of the key aspects that determines the performance of DataSteward is the selection method for the dedicated nodes. The problem of choosing optimal cloud application deployments is not new. The major approaches fall into three categories. *Random* methods simply select nodes randomly and have been extensively employed in works like Boinc [39]. *Ranking* strategies rate the cloud nodes based on their QoS and select the best ones [37, 65]. *Clustering* methods like [88, 87], consider the communication between the selected nodes to cluster the ones with good communication together.

We use a similar approach to cluster all the nodes within a deployment and then choose "leaders" from each cluster. The leaders are the best connected with all nodes within the cluster and will make up the global dedicated storage nodes. In contrast to existing clustering solutions, we first discover the communication topology and the potential virtual network bottlenecks by pre-executing a series of measurements. Next, we consider both the resource and the topology properties of a node in a unified way, in order to select the most suited ones. This is a key difference from existing works, which only take into consideration the CPU and/or the bandwidth of a node.

Algorithm 1 Initialization of the clustering algorithm.

Input:
Nodes $= \{\text{node}_1..\text{node}_N\}$ ▷ the set of compute nodes
$\text{Client}_{\text{Clusters}}[] = \{List_1..List_{\text{NrOfDataServers}}\}$ ▷ the set clients grouped in clusters
Output:
$\text{Servers} = \{\text{server}_1..\text{server}_{\text{NrOfDataServers}}\}$ ▷ the set of data servers - the cluster centroids
clients_per_server $= N/\text{NrOfDataServers}$

for $i \leftarrow 0, \text{NrOfDataServers}$ **do**
    Servers $\leftarrow$ node $\in$ Nodes (random selected)
end for

Cluster creation. To get an intuition of the cloud topology and the physical placement of the VMs, we rely on the fact that the cloud providers distribute the compute nodes in different fault domains (i.e., behind multiple rack switches). We aim to discover these clusters based on the proximity between the nodes within a fault domain. Hence, we fitted the clustering algorithm with adequate hypotheses for centroid selection and assignment of nodes to clusters. Finally, the selection of the dedicated nodes is done based on the discovered clusters, minimizing the overall data exchanges across switches and long-wires. The criteria that we want to maximize is the aggregated throughput that the application nodes will get to the dedicated storage nodes. Hence, the node assignment to a cluster is given in Equation 5.1:

$$\text{cluster} = \underset{i \in Servers}{\arg\max}\, \text{Max throughput}[\underbrace{i, j}_{|Client[i]|<clients\_per\_server}] \tag{5.1}$$

Centroids update. We select as a centroid the node towards which all other nodes in the cluster have the highest aggregated throughput. This maximizes the overall throughput of the application VMs within the cluster to the storage node:

$$\text{maxserver} = \underset{j \in Client[i]}{\arg\max} \sum_{k \in Client[i]} \text{throughput}[j, k] \tag{5.2}$$

Next, we present the cluster-based algorithm for selecting the dedicated nodes. Algorithm 1 introduces the data structures used to represent the problem and the random initialization of the centroids. The advantage of starting with a random setup is that no extra information is required. Algorithm 2 describes the 2 phases of the clustering algorithm. The first phase corresponds to Equation 5.1. The compute nodes are assigned to the clusters based on the throughput towards the dedicated nodes and by considering an upper-bound for concurrency (i.e., the maximum number of clients allowed per server). This upper limit for the load per storage allows the use of DataSteward even for cloud platforms which do not guarantee a distribution of nodes across racks. The second step consists in updating the centroids. We select the nodes which provide the highest aggregated throughput within each cluster, according to Equation 5.2. At the same time, we filter the nodes with poor QoS (i.e., low throughput or high I/O variability).

Algorithm 2 Clustering-based dedicated nodes selection

```
1: procedure DEDICATENODES(NrOfDataServers, N)
2:     repeat
3:         changed ← false
4:
5:                 ▷ Phase 1: Assign nodes to cluster based on proximity within clients limit
6:         for i ← 0, N do
7:             if i ∉ Servers then
8:                 max ← 0
9:                 maxserver ← 0
10:                for j ← 0, NrOfDataServers do
11:                    if throughput[Servers[j], i] > max && Client[j].Count < clients_per_server then
12:                        max = throughput[Servers[j],i]
13:                        maxserver = j
14:                    end if
15:                end for
16:                Client[maxserver].Add(i)
17:            end if
18:        end for
19:
20:        ▷ Phase 2: Centroid Selection — reselect the data servers based the assignment of nodes to clusters
21:        for i ← 0, NrOfDataServers do
22:            maxserver ← 0
23:            max ← 0
24:            for all j ∈ Client[i] do
25:                if j.std < ADMITTED_STD and j.thr > ADMITTED_THR then
26:                    server_thr ← 0
27:                    for all k ∈ Client[i] do
28:                        server_thr + = throughput[j,k]
29:                    end for
30:                    if server_thr > max then
31:                        max ← server_thr
32:                        maxserver ← j
33:                    end if
34:                end if
35:            end for
36:            if Servers[i] ≠ maxserver then
37:                Servers[i] ← maxserver
38:                changed ← true
39:            end if
40:        end for
41:    until changed == true
42: end procedure
```

5.2 In-transit data processing

To capitalize on the separation between storage and computation, we introduce a set of data processing services on top of the storage layer, that can *overlap with the executing applications*, in order to obtain some *early results*.

Generally, users have to implement such high level functionality for handling data. This translates in stalling the computation for executing these operations. Our approach offloads this overhead to the dedicated nodes and provides a data processing toolkit with an easily extensible API. At the same time, such a functionality exposed directly at the storage level allows to efficiently leverage some low-level data abstractions (e.g., layouts, partitions).

The Data Processing Services are exposed to applications through an API, currently available in C#, independent of the distributed storage chosen by the user. The API keeps the handling of data transparent to applications. In this way, the system remains generic and is able to accommodate future requirements. The processing services are loaded dynamically, from the default modules or from libraries provided by the users.

5.2.1 Data services for scientific applications

We provide three implementations of such services, targeting scientific applications.

Geographical replication. This is useful in the context of fog and federated clouds to enable fault tolerance, interoperability with other services and to disseminate the applications results. As the involved data movements are time and resource consuming, it is inefficient for applications to stall their executions in order to perform such transfers. DataSteward provides an alternative, as the applications can simply check-out their results to the dedicated nodes, which act as brokers. This operation is fast, consisting in a low-latency data transfer within the deployment. Then, DataSteward performs asynchronosulsy the time consuming geographical replication, while the application continues the main computation.

Data compression. Typically, the separation of scientific applications in multiple tasks leads to multiple results. Before storing them persistently, one can decrease the costs of persistent (long-term) storage through compression. By grouping together these results on the dedicated nodes, we are able to achieve higher compression rates, than if the results were compressed independently on their source node. In fact, many scientific applications have been shown to have high spatial or time correlation between the outputs of the computing sub-processes [99, 147]. DataSteward exploits these data similarities and minimizes the compression overhead of multiple files/objects, reaching compression rates that could not be achieved otherwise at the process or node level.

Scientific toolkit. Scientific applications typically require additional processing of their input/output data, in order to make the results exploitable. For large data sets, these manipulations are time and resource consuming. Moreover, an aspect which typically tends to be neglected, but impacts the overall data handling time as well, is the number of objects in the data set (e.g., I/O files, log files). Due to the simplicity of the default cloud storage API, these objects need to be managed independently as there are no operations which can be applied on multiple files. Additionally, they need to

be downloaded at the client side most of the time, even for simple operations such as searching or checking the file exists. By using the dedicated nodes, such processing on multiple files can be provided directly at the data management layer and can be overlapped with the main computation. Therefore, DataSteward provides an integrated set of tools applicable on groups of files. These operations support file set transformations such as filter, grep, select, search or property check.

5.3 Evaluation and perspectives

We perform an evaluation of DataSteward both in synthetic settings and in the context of scientific applications. The experiments are carried out on the Azure cloud in the North Europe and West US datacenters. The experimental setup consists of up to 100 Medium Size VMs, each having 2 virtual CPU which are mapped to physical CPUs, 3.5 GB of memory and a local storage of 340 GB.

5.3.1 Data storage evaluation

First, we compare DataSteward performance with TomusBlobs. The goal is to evaluate the benefits of dedicating compute nodes for storage against collocating storage with the computation. To ensure a fair comparison, each system is deployed on 50 compute nodes and both use the BlobSeer [146] backend. Next, we compare against the local cloud storage service (Local AzureBlobs) and a geographically remote storage instance from another data center (Geo AzureBlobs). The goal of this second comparison is to demonstrate that, although DataSteward is not collocating data with computation, the delivered I/O throughput is superior to a typical cloud remote storage.

Multiple reads / writes. We consider 50 concurrent clients that read and write, from memory, increasing amounts of data, ranging between 16 to 512 MB. When using TomusBlobs, the clients are collocated on the compute nodes. For DataSteward the clients run on distinct machines. We report the cumulative throughput of the storage system for the read and write operations in Figures 5.3 and 5.4, respectively. Unsurprisingly, both approaches considerably outperform (between 3x and 4x) the cloud storage, whether located on-site or on a geographically distant site. This comes as a result of keeping data within the deployment. This minimizes the number of infrastructure hops (e.g., switches, racks) between application nodes and storage nodes, thanks to our topology-aware strategy for allocating resources.

Let's focus on the performance of DataSteward and TomusBlobs. One might expect that, due to data locality, the collocated storage option delivers better performance than managing data in dedicated nodes. However, the measurements reported in Figures 5.3 and 5.4 show differently. While for small data volumes they perform similarly, for larger data sizes DataSteward outperforms the collocated storage with more than 15 %. This has 2 reasons. First, for both approaches, the underlying BlobSeer backend splits data into chunks scattered across the federated virtual disks. So even with the collocated storage not all data accessed by a client is always entirely present on the local VM. Moreover, the throughput is determined both by the network bandwidth, which is the same in both setups, and by the CPU's capability to handle the incoming data. The latter is better leveraged by DataSteward which

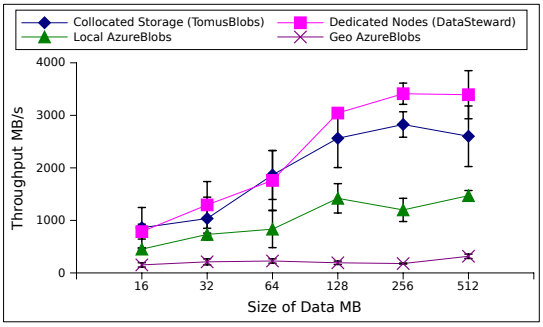


Figure 5.3 – The cumulative read throughput with 50 concurrent clients.

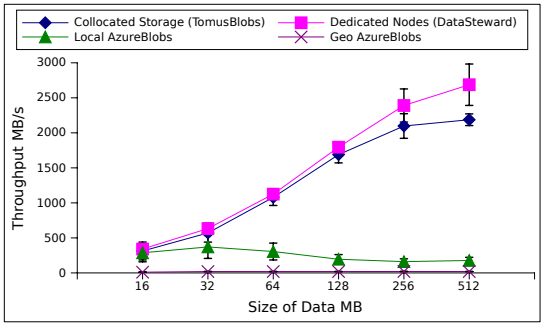


Figure 5.4 – The cumulative write throughput with 50 concurrent clients.

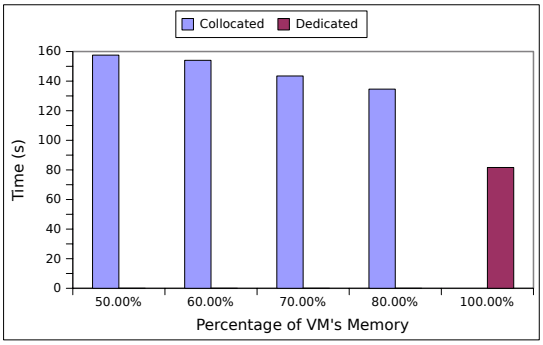


Figure 5.5 – The execution time of the Kabsch-based application. The percentage of the VM memory used by the application is increased, when using the collocated storage. The final bar represents the application execution time when all VM memory is used for the computation and the storage is moved to dedicated nodes

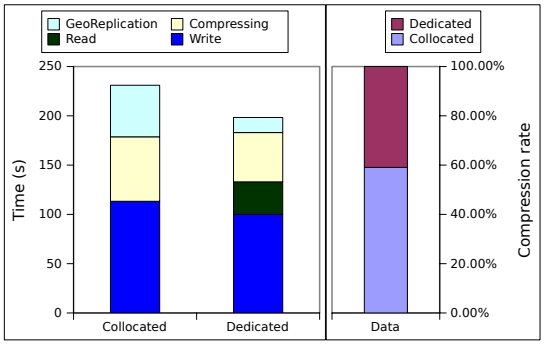


Figure 5.6 – Left, the total time spent to compress and geographically replicate 100 MB of data from the Kabsch-based application, when these operations are performed on the application nodes and with DataSteward. On right, we show the gains in compression rate obtained when data is aggregated first from multiple nodes, before compressing it.

separates computation from communication. Second, with increasing sizes of data operated by a client, a poor management of the network links between the VMs leads to a faster saturation of the network. This can be observed for sizes beyond 200 MB, for which TomusBlobs reaches its upper bound of performance faster. It is not the case for DataSteward. Thanks to the topology-aware distribution strategy of data nodes, it manages the network better, resulting in a higher upper bound for I/O throughput.

5.3.2 Gains of in-transit processing for scientific applications

Impact of the processing collocation vs. in-transit processing. We first compare the execution times when the memory is split between the computation and the collocated storage handling against dedicating the full memory to computation (as with DataSteward). To perform this analysis, we used a bio-informatics application which enables a configurable execution in terms of memory usage. The application performs the Kabsch algorithm [112],

which computes the optional rotation matrix that minimizes the root mean squared deviation between two sets of data. This algorithm is used in many scientific computations from fields like statistical analysis for molecular structures comparison. The experimental methodology consisted in running the application with increasing amounts of memory used for the actual computation. The remaining memory is assigned to the collocated storage, for which we used the TomusBlobs approach as before. The execution times of this scenario were compared with the case in which all the memory of the VMs is used for computation and the storage is handled by DataSteward, located on dedicated nodes. The results in Figure 5.5 show that DataSteward reduces the execution timespan (computation and data handling time) for such scenarios to half compared to a collocation strategy.

Next, we evaluate the data processing services provided by the DataSteward. For this analysis we consider two sets of data. First we use the data computed by the Kabsch-based application, having a size of approximately 100 MB, which is compressed and geographically replicated from the European to the United States data center. The second data set is the 28,000 log files, each file having a size less than 10 KB, used by A-Brain [174], a bioinformatics application.

Data compression. We compare the DataSteward *in-transit* approach with the default option in which each users implement the compression on the application nodes (i.e., collocation). The evaluation considers the total times, shown on the left in Figure 5.6, to write the data to the storage system, compress it and transfer it to the remote datacenters. Additionally, we present on the right side the levels of compression achieved when the operation is applied independently on each file, as in the case where the functionality is provided on each node (i.e., collocated), or collectively on the entire data set (i.e., in-transit). The price paid by DataSteward to execute these operations non-intrusively is an extra transfer from the application nodes to the dedicates storage nodes. Such a transfer is not required when each node handles its own data locally, therefore the missing "read" label for the collocated option in Figure 5.6. Nevertheless, the overall execution time is reduced by 15 % with DataSteward. This is because it is more efficient to transfer locally and then compress all the aggregated data at one place than to do it independently on small chunks in the application nodes. Building on this grouping of data, DataSteward is also able to obtain up to 20 % higher compression rates.

Handling large number of files. Figure 5.7 presents the execution times for several operations commonly observed during long-running experiments on large groups of files. We compare DataSteward to their implementation on top of the cloud storage. We use 28,000 monitoring files from the A-Brain experiment. The time to manage the files, regardless of the operation performed, is reduced with one order of magnitude.

5.3.3 Going further

The idea of *in-transit* processing on dedicated nodes opens numerous options for the data services to be executed on such deployments, besides the 3 already implemented in DataSteward. We briefly discuss here a few interesting (yet not exhaustive) directions.

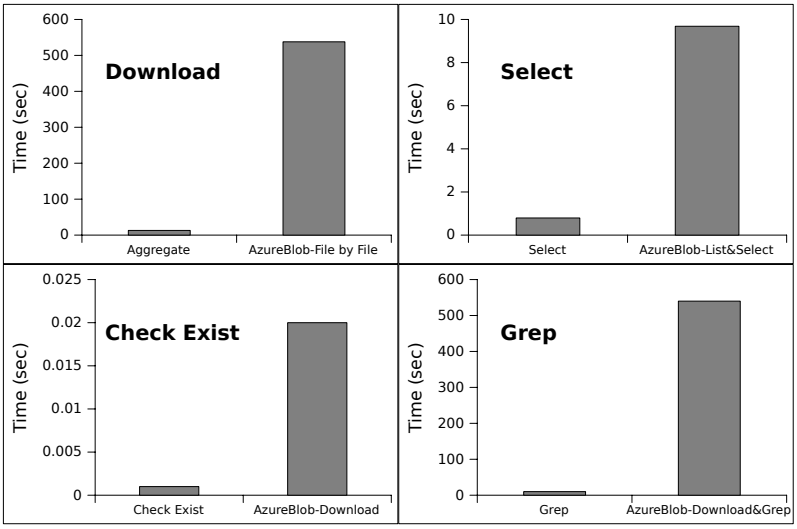


Figure 5.7 – The execution time of recurrent operations from scientific experiments on 28,000 monitoring files, when the operations are supported and executed by the in-transit service or implemented at client side.

Cache for the persistent storage. Its role would be to periodically backup the data from the dedicated storage nodes into the cloud persistent storage. As a result, DataSteward would be enhanced with persistence, following closely the structure of the physical storage hierarchy: machine memory, local and network disks, persistent storage. For critical systems, this service could be coupled with the (already provided) geographically replication one. As a result, data could be backed-up across geographical-distinct cloud storage instances, to guarantee availability against disasters or outages. The client applications would access any storage only through the dedicated nodes. If the data is not available within the deployment (e.g., in case of a crash that affects all replicas of a data object), then the object is copied from the persistent storage, cached locally and made available to applications.

Edge analytics. With edge devices becoming more and more powerful and energy-efficient, they could be used to host DataSteward and perform an important part of the analysis at the collection site, *in-situ*. This allows to filter and aggregate data locally, before it gets a chance to accumulate, and to further take local decisions improving the reactivity of the analytics.

Cloud introspection as a service. The cloud model, as it is defined today, hides from applications all infrastructure aspects: load, topology, connection routes, performance metrics, etc. On the one hand, this simplifies the task of building and deploying applications, but on the other hand it prevents them to optimize the usage of the leased resources. Building on the clustering scheme presented in section 5.1, one could design an introspection service that could reveal information about the cloud internals. In a geographically distributed setting, this would allow applications to gain an intuition of the number of datacenters, their location (i.e., latency) or interconnecting links. Within a datacenter, applications could learn the type of topology used, the available bandwidth or the number of physical machines and racks.

Chapter 6

JetStream: Fast Stream Transfer

Contents

STREAM PROCESSING REQUIRES FREQUENT DATA MOVEMENTS across widely distributed sites. These occur either from the data production places (i.e., the edge) to the data processing ones (i.e., the cloud sites), or between the cloud sites when the processing is geographically distributed. Typically, these sites are connected through high-latency, low-throughput wide-area networks. They make inter-site data transfers up to an order of magnitude slower than intra-site data transfers. However, despite the growth in volumes of stream data, the research in this domain has neglected the support for efficient stream transfers.

In fact, this functionality tends to be delegated to the event source [64, 194]. As a result, the typical way to transfer events is individually (i.e., *event by event*), as they are produced by the data source. This is highly inefficient, especially in geographically distributed scenarios, due to the latencies and overheads at various levels (e.g., application, technology encoding tools, virtualization, network).

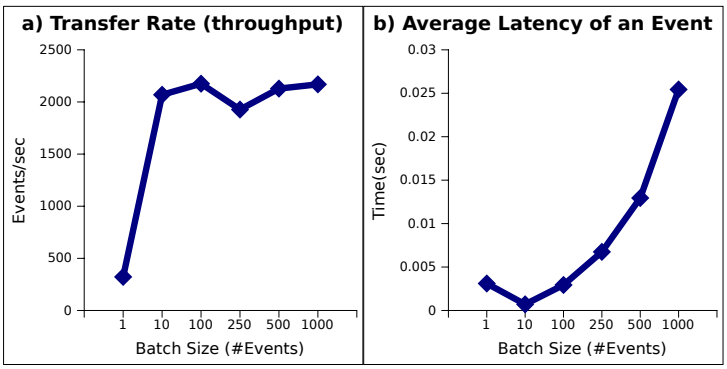


Figure 6.1 – a) Transfer rate and b) average event latency for transferring events in batches between North Europe and North US Azure datacenters.

> *❝ The key challenge is the selection of an optimal batch size and the decision on when to trigger the batch sending. ❞*

A better option is to transfer events in *batches*. While this improves the transfer rate, it also introduces a new problem, related to the selection of the proper batch size (i.e., *how many events to batch?*). Figure 6.1 presents the impact of the batch size on the transfer rate, and transfer latency per event, respectively. We notice that the key challenge is the selection of an optimal batch size and the decision on when to trigger the batch sending. This choice strongly relies on the streaming scenario, the resource usage and on the context (i.e., the cloud environment).

To address these issues, we propose a set of strategies for efficient transfers of events between cloud datacenters. We implement them in JetStream [28, 25, 27], a high-performance, batch-based streaming middleware. JetStream is able to self-adapt to the streaming conditions by modelling and monitoring a set of context parameters. The size of the batches and the decision on when to stream the events are controlled dynamically, based on the context model. JetStream leverages multi-route streaming across cloud nodes, aggregating inter-site bandwidth. The approach was validated on the Microsoft Azure cloud using synthetic benchmarks and a real-life scenario based on the MonALISA [4] monitoring system of the CERN LHC experiment [29].

6.1 Modelling the stream transfer in the context of clouds

The main objectives of an efficient transfer system are: to dynamically adapt to the environment and to minimize the latency. In our case, this translates to *correlating the transfer parameters* (i.e., batch size, triggering moment) with *the cloud context* (i.e., variability of resources, fluctuating event generation rates, nodes and datacenters routes). Ideally, this correlation would sustain a high transfer rate while delivering a small average latency per event. Finding the right correlation requires an appropriate model for streaming in the cloud.

To achieve this, we argue for *decoupling the event transfer from the processing*. Designing the transfer module as a stand-alone component allows seamless integration with any SPE running in the cloud. At the same time, it provides sustainable performance independent on the usage setup or specific architectures.

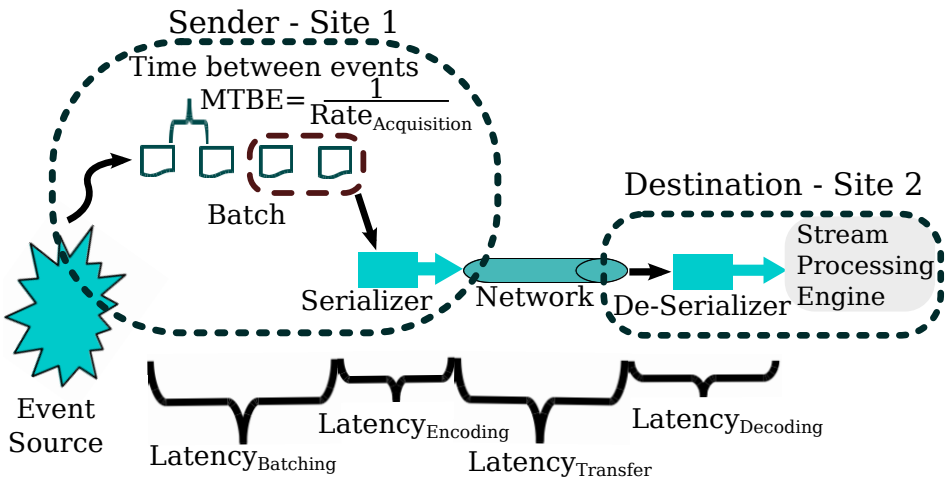


Figure 6.2 – Breaking down the latency to deliver an event from source to stream processing engine across cloud nodes.

6.1.1 Zoom on the event delivery latency

The model we propose expresses the latency of the events based on a set of cotext parameters which can be monitored. This allows to correlate the batch size corresponding to the minimal event latency both with the stream context and the environment information. The set of parameters describing the stream context are:

- the average acquisition rate ($\text{Rate}_{\text{Acquisition}}$) or mean time between events (MTBE);
- the event size ($\text{Event}_{\text{SizeMB}}$);
- the serialization/de-serialization technique;
- the throughput (thr);
- the batch size ($\text{batch}_{\text{Size}}$).

The goal is to dynamically determine the batch size that minimizes the latency defined using these parameters. The latency between the source and the destination of an event has four components, depicted on Figure 6.2. These are: batch creation, encoding (e.g., serializing, compression, etc.), transfer and decoding.

The batching latency corresponds to the delay added when an event is waiting in the batch for other events to arrive, before they are all sent together. The parameters which describe this latency are the average acquisition rate of the events and the number of events in the batch. As the delay depends on the position of the event in the batch (i.e., position $\times \frac{1}{\text{Rate}_{\text{Acquisition}}}$, we choose to express it as the average latency of an event. This can be computed by averaging the sum of the delays of all events in the batch:

$$\text{Latency}_{\text{batching}} = \frac{\text{batch}_{\text{Size}}}{2} \times \frac{1}{\text{Rate}_{\text{Acquisition}}}$$

Intuitively, this corresponds to the latency of the event in the middle of the sequence.

The transformation latency gathers the times to encode and to decode the batch. This applies to any serialization library/technology. The latency depends on the used format

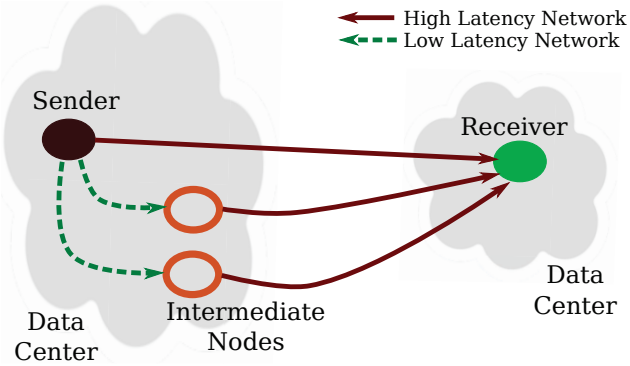


Figure 6.3 – The proposed schema for multi-route streaming across cloud nodes.

(e.g., binary, JSON, etc.), the number of bytes to convert and the number of events in the batch. To express this, we represent the transformation operation as an affine function (i.e., $f(x) = ax + b$) where a corresponds to the conversion rate (i.e., amount of bytes converted per second - time for data encoding tDe), while the b constant gives the time to write the metadata (time for header encoding tHe). The latency per event can be expressed as:

$$\text{Latency}_{\text{transformation}} = \frac{tHe + tDe \times \text{batch}_{\text{SizeMB}}}{\text{batch}_{\text{Size}}}$$

which holds both for the encoding and decoding operations.

The transfer latency models the time required to transfer an event between cloud nodes across datacenters. To express it, we consider both the amount of data in the batch as well as the overheads introduced by the transfer protocol (e.g., HTTP, TCP) — size overhead for transport sOt and the encoding technique — size overhead for encoding sOe. Due to the potentially small size of data transferred at a given time, the throughput between geographically distant nodes cannot be expressed as a constant value. It is rather a function of the total batch size ($\text{Size}_{\text{Total}} = \text{batch}_{\text{SizeMB}} \times \text{batch}_{\text{Size}}$), since the impact of the high latency between datacenters depends on the batch size. The cloud inter-site throughput - $thr(\text{Size})$ is discussed in more detail in the following section. The average latency for transferring an event can then be expressed as:

$$\text{Latency}_{\text{transfer}} = \frac{sOt + sOe + \text{batch}_{\text{SizeMB}}}{thr(\text{Size}_{\text{Total}})}$$

6.1.2 Multi-route streaming

In order to address the issue of low inter-data center throughput, we leverage DataSteward to dedicate a set of compute nodes within the source datacenter to coordinate the transfer as follows. The idea is to aggregate additional bandwidth by sending batches of events from the sender nodes to intermediate nodes within the same deployment. They will then forward them towards the destination. The dedicated nodes are used for fast local replication with the purpose of transferring data in parallel streams. Multiple parallel paths are then used

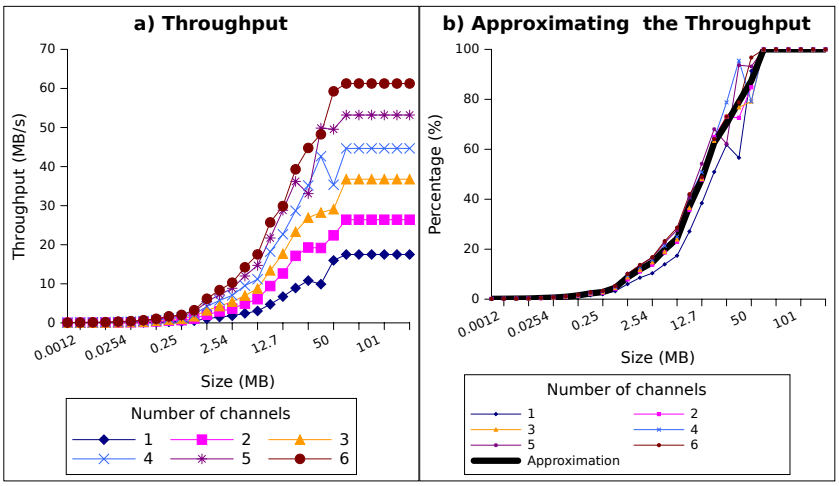


Figure 6.4 – a) The throughput with respect to the number of routes across cloud Small VMs for increasing size of the data. b) Approximating the cloud throughput, independent of the number of routes, based on the averages of the normalized throughput functions for each number of routes.

originated from these points, leveraging the fact that the cloud routes packages through different switches, racks and network links. In this way, the wide-area transfers will use multiple routes aggregating the inter-site throughput. This scheme is shown in Figure 6.3.

To integrate this approach within our model, we extend the set of parameters that are used to characterize the stream context with the number of *channels*. This parameter gives the parallelism degree of the multi-route schema deployed for streaming, i.e., the number of nodes used on the sender site for streaming.

Multi-route throughput. Independent of the number of routes used for streaming, the value of the throughput function needs to be known to model the latency. Considering the potentially small sizes of the data to be transferred, one needs to study the function of the throughput with respect to the size, not just to measure its peak stable value. In order to limit the number of network samples that need to be performed by a monitoring service, we approximate the throughput function. In Figure 6.4 a) we present measurements of the throughput for a number of routes between North-Central US and North EU Azure data centers. In Figure 6.4 b) we normalize these values (i.e., % of the corresponding stable throughput) and approximate them using a polynomial function. This was determined empirically to give a good approximation, with an error introduced due to the cloud variability of less than 15 % with respect to the approximation based on measuring the stable value. Using this approximation, the entire function can be extrapolated by measuring only the asymptotic stable throughput. This will be used as the amplitude, which multiplied with the normalized estimation, will give the throughput for any size.

Batch reordering. The downside of using multiple routes for sending batches is that the ordering guarantees offered by the communication protocol for one route do not hold any-

more. This translates into batches arriving out of order due to changing conditions on the physical communication routes (e.g., packet drops, congestion, etc.). Hence, batches need to be reordered at the destination and the corresponding delay (i.e., *latency for reordering*) needs to be accounted for within the model. The reordering is done by buffering the batches at the destination until their turn to be delivered to the streaming engine arrives. We model this additional latency by using the Poisson distribution ($\text{Poisson}(k, \lambda) = \frac{\lambda^k \times e^{-\lambda}}{k!}$) to estimate the probability of having k number of batches arriving before the expected batch. As we take as reference the transfer of the next expected batch, λ parameter becomes 1. This probability can then be correlated with a penalty assigned to each unordered batch. We use as penalty the latency (i.e., $\text{Latency}_{\times batch}$) incurred by having a number of batches (j) arriving out of order. This gives $\text{Poisson}(j, 1) \times j \times \text{Latency}_{\times batch}$, over which we sum in order to account for all potential number of batches arriving out of order. We denote L the maximum number of batches (e.g., 10) potentially arriving before the reference one through a channel, giving the upper limit for the summation. Finally, we sum these penalties over the number of channels, as each channel can incur its own number of unordered batches, and normalizing based on the events, as our model expresses everything as latency per event. The final equation that models the unordered batches arriving through all channels is:

$$\text{Latency}_{\text{reordering}} = \frac{\sum_{i=2}^{\text{channels}} \sum_{j}^{L} \text{Poisson}(j, 1) \times j \times \text{Latency}_{\times batch}}{\text{batch}_{\text{size}} \times L}$$

6.2 The JetStream transfer middleware

In this section we show how the previous model is applied to select the optimal number of routes and events to batch, as well as the architecture of the JetStream middleware which implements this approach.

6.2.1 Adaptive batching for stream transfers

In Algorithm 3, we present the decision mechanism for selecting the number of events to batch and the parallelism degree (i.e., channels/routes) to use. The algorithm successively estimates the average latency per event, using the formulas presented in section 6.1, for a range of batch sizes and channels, retaining the best one. Instead of exhaustively searching in the whole space we apply a simulating annealing technique, by probing the space with large steps and performing exhaustive searches only around local optima. Depending on the value of the optimal batch size, the maximal end-to-end event latency introduced by batching can be unsatisfactory for some applications, even if the system operates at optimal transfer rates. Hence, the users can set a maximum acceptable delay for an event, which will be converted in a maximum size for the batch (Line 3).

The selection of the number of channels is done by estimating how many batches can be formed while one is being transferred (Lines 6-8). Beyond this point, adding new channels leads to idle resources and therefore decreases the cost efficiency. The condition on Line 10 prevents the system from creating such idle channels. Finally, the CPU usage needs to be accounted in the decision process: sending frequent small batches will increase the CPU consumption and artificially decrease the overall performance of the cloud node. We therefore assign a penalty for the CPU usage, based on the ratio between the time to form a

Algorithm 3 The selection of the optimal batch size and the number of channels to be used

```
1:  procedure BATCHANDCHANNELSSELECTION
2:      getMonitoredContextParameters()
3:      ESTIMATE MaxBatched from [MaxTimeConstraint]
4:      while channels < MaxNodesConsidered do
5:          while batch_size < MaxBatched do
6:              ESTIMATE latency_batching from [Rate_Acquisition, batch_size]
7:              ESTIMATE latency_encoding from [overheads, batch_sizeMB]
                                          ▷ Estimate the transfer latency for 1 channel
8:              ESTIMATE latency_transfer1 from [batch_sizeMB, thrRef, 1channel]
9:              COMPUTE Ratio_CPU from [latency_encoding, latency_batching, VM_Cores]
                                                          ▷ Prevents idle channels
10:             if Ratio_CPU * latency_batching × channels < latency_transfer1 + latency_encoding then
11:                 ESTIMATE latency_decoding from [overheads, batch_sizeMB]
12:                 ESTIMATE latency_transfer from [batch_sizeMB, thrRef, channels]
13:                 ESTIMATE latency_reordering from [channels, latency_transfer]
14:                 latency_perEvent = ∑ latency_*
15:                 if latency_perEvent < bestLatency then
16:                     UPDATE [bestLatency, Obatch, Ochannels]
17:                 end if
18:             end if
19:         end while
20:     end while
21: end procedure
```

batch (a period with a low CPU footprint) and the time used by the CPU to encode it (a CPU intensive operation), according to the formula:

$$\text{Ratio}_{CPU} = \frac{\text{latency}_{\text{encoding}}}{(\text{latency}_{\text{batching}} + \text{latency}_{\text{encoding}}) \times \text{VM\_Cores}}$$

When computing the ratio of intense CPU usage, we account also for the number of cores available per VM. Having a higher number of cores prevents CPU interference from overlapping computation and I/O and therefore does not require to prevent small batches.

6.2.2 Architecture overview

We implemented this approach into a high-performance cloud streaming middleware, called JetStream, as a proof of concept. The system does not require changes in application semantics nor modifications or elevated privileges to the cloud hypervisor or stream processing engines. Its conceptual scheme is presented in Figure 7.2. The events are fed into the system at the sender side, as soon as they are produced, and they are then delivered to the stream processing engine at the destination. Hence, the adaptive-batch approach remains transparent to the system and users. The implementation of this architecture is done in C# using the .NET 4.5 framework and consists of several modules described below.

The Buffer is used as an event input and as an output endpoint. The source simply adds the events to be transferred, as they are produced, while the receiver application (i.e.,

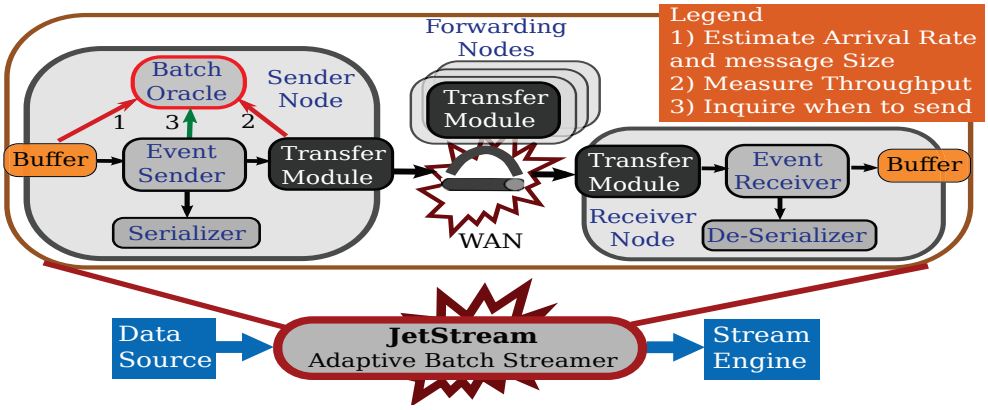


Figure 6.5 – The architecture and the usage setup of the adaptive batch streamer.

the stream processing engine) *pops* (synchronously) the events or is notified (asynchronously) when they are available. The *Buffer* entity at the sender side is also in charge of monitoring the input stream in order to assess the *acquisition rate* of the events and their *sizes* in real-time.

The Batch Oracle enforces the environment-aware decision mechanism for adaptively selecting the batch size and the amount of channels to use. It implements Algorithm 3 and collects the monitoring information from the *Buffer* and the *Transfer Module*. It further controls the monitoring intrusiveness by adjusting the frequency of samples.

The Transfer Module performs the multi-route streaming. Batches are sent in a round-robin manner through the channels assigned for the transfer. On the intermediate nodes, the role of this component is to forward the batches towards the destination. Currently, the system offers several implementations on top of TCP: synchronous and asynchronous, single- or multi-threaded.

The Event Sender coordinates the event transfers by managing the interaction between modules. It queries the *Batch Oracle* about when to start the transfer of the batch. Next, it setups the batch by getting the events from the *Buffer* and adding the metadata (e.g., batch ID, streams IDs and the acknowledgments needed for multi-route transfers). The batch is then serialized by the *Serialization* module and the data are transferred across datacenters by the *Transfer Module*.

The Event Receiver de-serializes, buffers, reorders, and delivers the arriving batches to the application as a stream of events, in a transparent fashion for the stream processing engine. The module issues acknowledgments to the sender or makes requests for re-sending lost or delayed batches. Alternatively, based on users policies, it can decide to drop late batches, supporting the progress of the stream processing despite potential cloud-related failures.

Serialization/De-Serialization has the role of converting the batch to raw data, which are afterwards sent over the network. We integrate in our prototype several libraries: Binary (native), JSON (scientific) or Avro (Microsoft HDInsight), but others can be easily integrated. Moreover, this module can be extended to host additional functionality: data compression, deduplication, etc.

6.3 Experimental evaluation

We validate JetStream in a real cloud setup and discuss the main aspects that impact its performance. The experiments were run in the Microsoft's Azure cloud in the North-Central US and the North EU data centers, using Small Web Role VMs (1 CPU, 1.75 GB of memory, 225 GB local storage). For multi-route streaming, up to 5 additional nodes were used within the sender deployment. Each experiment sends between 100,000 and 3.5 million events, which, depending on the size of the event to be used, translates into a total amount of data ranging from tens of MBs to 3.5 GB. Each sample is computed as the average of at least ten independent runs of the experiment performed at various moments of the day (morning, afternoon and night).

The performance metrics considered are the *transfer rate* and the *average latency of an event*. The transfer rate is computed as the ratio between a number of events and the time it takes to transfer them. More specifically, we measured at the destination the time to transfer a fixed set of events. For the average latency of an event, we measured the number of events in the sender buffer, the transfer time, and reported the normalized average per event based on the latency formulas described in section 6.1. The generation rates are varied between hundreds of events per second and tens of thousands of events per second, as indicated by the scale of the transfer rates.

6.3.1 Individual vs. batch-based event transfers

The goal of this set of experiments is to analyze the performance of individual event streaming compared to batch-based streaming between cloud datacenters. We consider both static batch sizes as well as the adaptive batch selection of JetStream. The experiments were repeated for 1, 3 and 5 parallel routes for streaming. We measured the transfer rates (top) and average latency per event (bottom).

The experiments presented on Figure 6.6 use an event of size 224 bytes and evaluate the transfer strategies considering low (left) and high (right) event generation rates. The first observation is that batch-based transfers clearly outperform individual event transfers for all the configurations considered. The results confirm the impact of the streaming overheads on small chunk sizes and the resulting low throughput achieved for inter-site transfers. Grouping the events increases the transfer rate tens to hundreds of times (up to 250 times for JetStream) while decreasing the average latency per event.

Two aspects determine the performance: the size of the batch with respect to the stream context and the performance variations of the cloud resources (e.g., nodes, network links). Static batches cannot provide the solution, as certain batch sizes are good in one context and bad in others. For example batches of size 10 deliver poor performance for 1 route and high event acquisition rate and good performance for 5 routes and low acquisition rates. Selecting the correct size at runtime brings an additional gain between 25 % and 50 % for the event transfer rate over static batch configurations (for good batch sizes not for values far off the optimal). To confirm the results, we repeated the same set of experiments for larger event sizes. Figure 6.7 illustrates the measurements obtained for event sizes of 800 bytes, showing that JetStream is able to increase the performance with up to 2 orders of magnitude over current streaming strategies.

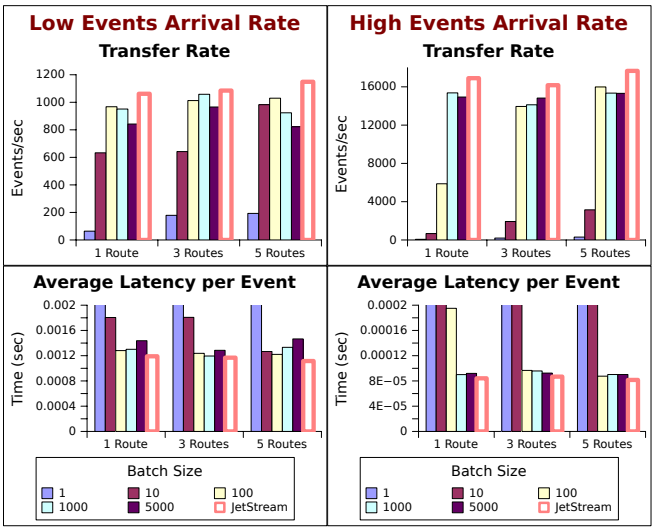


Figure 6.6 – Comparing the performance (transfer rate - top and average latency per event - bottom) of individual event streaming and static batches with JetStream for different acquisition rates, while keeping the size of an event fixed (224 bytes). The performance of individual event streaming (i.e., batch of size 1) are between 50 and 250 times worse than the ones of JetStream.

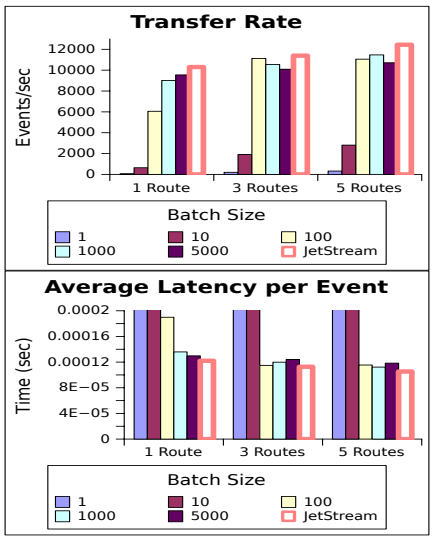


Figure 6.7 – The performance (transfer rate - top and average latency per event - bottom) of individual event streaming, static batches and JetStream for events of size 800 bytes. The performance of individual event streaming (i.e., batch of size 1) are from 40 to 150 times worse than JetStream.

6.3.2 Adapting to context changes

The event acquisition process in streaming scenarios is not necessarily uniform. Fluctuations in the event rates of an application running in the cloud can appear. They are due to the nature of the data source, the virtualized infrastructure or the cloud performance variability [90]. To analyze the behavior of JetStream in such scenarios, we performed an experiment in which the event generation rate randomly changes in time. For the sake of understanding, we present in Figure 6.8 a snapshot of the evolution of the transfer rate in which we use fine grain intervals (10 seconds) containing substantial rate changes.

JetStream is able to handle these fluctuations by appropriately adapting the batch size and number of routes. In contrast, static batch transfers are introducing huge latencies from waiting for too many events, especially when the event acquisition rate is low (e.g., batches of size 1000 or 5000 at time moment 9). They are also falling behind the acquisition rate which leads to increasing the amount of memory used to buffer the events not transferred yet (e.g., batch of size 100 at moment 5). Reacting fast to such changes is crucial for delivering high-performance.

6.3.3 Benefits of multi-route streaming

Figure 6.9 shows the gains obtained in transfer rate with respect to the number of routes used for streaming, for JetStream and for a static batch of a relatively small size (i.e., 100 events).

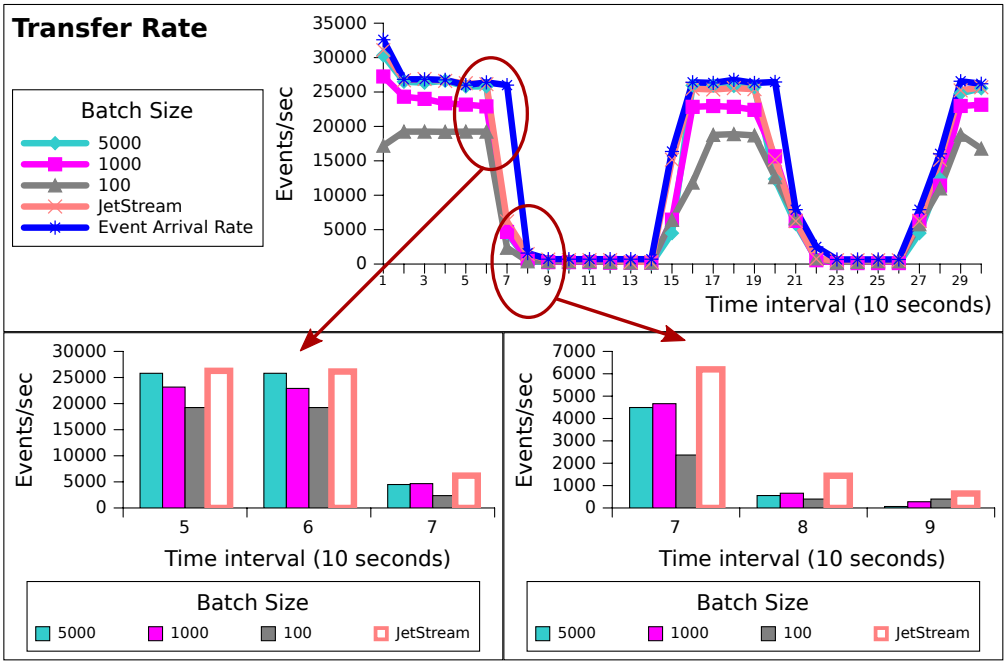


Figure 6.8 – The evolution of the transfer rate in time for variable event rates with JetStream and static batches transfer strategies

When increasing the amount of data to be sent, multi-route streaming pays off for both strategies. By aggregating extra bandwidth from the intermediate nodes, we are able to decrease the impact of the overhead on smaller batches: batch metadata, communication and serialization headers. More precisely, a larger bandwidth allows to send more data, and implicitly, the additional data carried with each batch does not throttle the inter-site network anymore. This brings the transfer rate of smaller, and consequently more frequent, batches closer to the maximum potential event throughput. This can be observed for the static batch of size 100 on Figure 6.9, which delivers a throughput close to JetStream for a high number of routes.

With higher throughput and a lower overhead impact, the optimal batch size can be decreased. In fact this is leveraged by JetStream, which is able to decrease the end-to-end latency by selecting lower batch sizes. Hence, we conclude that sustaining high transfer rates under fixed time constraints is possible by imposing upper bounds for the batch sizes and compensating with additional streaming routes. This enables JetStream to integrate users constraints for the maximal delay, which are integrated in the streaming decision shown in Algorithm 3 by considering a limit on the batch size.

6.3.4 JetStream in support of a real-life LHC application

ALICE (A Large Ion Collider Experiment) [29] is one of the four LHC (Large Hadron Collider) experiments at CERN, with a scale, volume and geographical distribution of data requiring appropriate tools for efficient processing. Indeed, the ALICE collaboration, consists of more than 1,000 members from 29 countries, 86 institutes and more than 80 computing

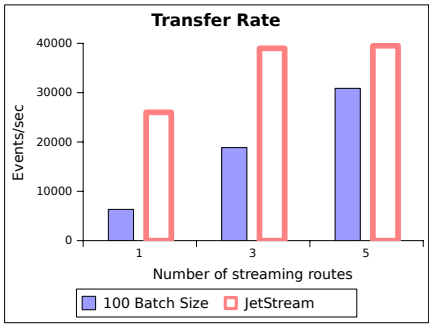


Figure 6.9 – The transfer rate for an increasing number of routes used for streaming, when the batch size is fixed or adaptively chosen using JetStream.

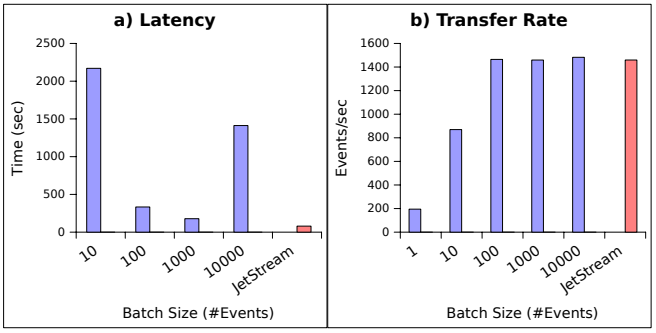


Figure 6.10 – The total latency (a) and the transfer rate (b) measured when transferring 1.1 million events from MonALISA. The latency for independent event transfer (batch of size 1) in a) is not represented because it would modify the scale, having a value of 30900 seconds as opposed to 80 seconds for JetStream.

centers worldwide. It is strongly dependent on a distributed computing environment to perform its physics program. The experiment collects data at a rate of up to four petabytes per year. Our focus, in these series of experiments, is on the monitoring information collected in real-time about all ALICE resources. We used the MonALISA [4] service to instrument and replay the huge amount of monitoring data issued from this experiment. More than 350 MonALISA services are running at sites around the world, collecting information about ALICE computing facilities, local and wide-area network traffic, and many thousands of concurrently running jobs. This yields more than 2 million parameters published in MonALISA, each with an update frequency of one minute. The MonALISA framework and its high-frequency updates for large volumes of monitoring data match closely with JetStream's purposes.

Based on the monitoring data collected by MonALISA as of December 2013, we have replayed a sequence of 1.1 million events considering their creation times at the rate they were generated by ALICE. The measurements were performed using 2 intermediate nodes located at the sender side (i.e., resulting in 3 streaming routes).

Figure 6.10 a) shows the total latency of the events at the sender and Figure 6.10 b) presents the transfer rate, when comparing JetStream with static configurations for various batch sizes. The transfer performance of static batches with more than 100 events is similar JetStream. Considering that the generation rate of the events varies from low to high, these sub-optimal batch sizes will in fact lead to an accumulation of the events in the sender queue during the peak rates. These buffered events will artificially increase the performance, at the expense of extra memory, during the periods when the acquisition rate of events is low. All in all, this behavior will produce a stable transfer performance over a wide range of static batch sizes, as it can be observed in Figure 6.10 b). But on the other hand, it will increase the latency of the events as depicted in Figure 6.10 a). As our approach selects the appropriate batch size at each moment, it consequently reduces the amount of events waiting in the sender queue and decreases the overall latency of the events. Compared to the static batch strategies, the latency obtained with JetStream is reduced between 2.2 (100-event batches) down to 17 times (10,000-event batches).

6.3.5 Towards stream transfer "as a Service"

This evaluation highlights the inefficiency of today's streaming strategies and the need for new tools able to deliver fast and efficient data transfers. JetStream aims to be such a tool, deployed in the *user space* and allowing them to optimize transfers by monitoring the cloud environment for insights on the underlying infrastructure.

An interesting direction to investigate is how such a tool can be "democratized" and transparently offered by the *cloud provider*, using a transfer "as a Service" paradigm. This shift of perspective arises naturally. Instead of letting users optimize their transfers by making deductions about the underlying network topology and performance through intrusive monitoring, the idea would be to delegate this task to the cloud provider. Indeed, the cloud owner has extensive knowledge about the network resources, which can be leveraged within the proposed system to optimize (e.g., by grouping) user transfers.

Our working hypothesis is that such a service will offer slightly lower performance than a highly-optimized dedicated user-based setup (e.g., based on multi-routing through extensive use of network parallelism as we proposed with JetStream), but substantial higher performance than today's state-of- the-art transfer solutions (e.g., using the cloud-provided storage service or GridFTP). Moreover, this approach has the advantage of freeing users from setting own systems, while providing the same availability guarantees as for any cloud managed service.

Chapter 7

Small Files Metadata Support for Geo-Distributed Clouds

Contents

STREAMING GENERATES METADATA ACROSS ALL THE PHASES of the pipeline: at the event generation to annotate with provenance information, at transfer to order batches, at storage to identify replicas and partitions, and finally at processing to keep track of the whole execution state. Hence, the size of the metadata can be proportionally large for streaming data. Worse, most files storing such metadata are typically *small, with median file sizes in the orders of kilo- or megabytes*, yet generated in large numbers. This means that metadata access has a high impact (sometimes being dominant) on the overall processing I/O. Such metadata overload can easily saturate state-of-the-art file systems, in which metadata are typically managed by a centralized server (see section 2.3.1).

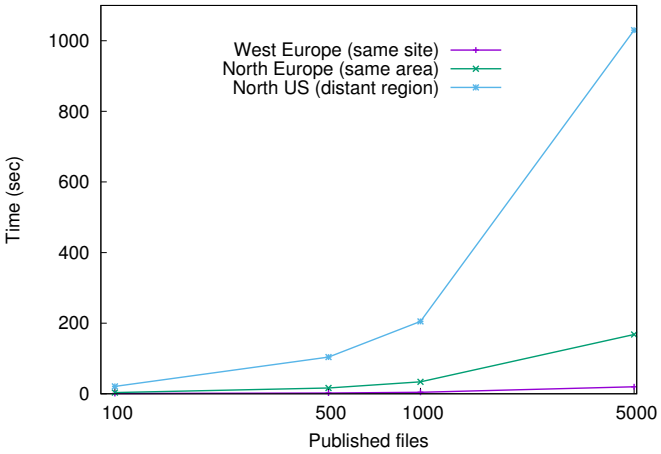


Figure 7.1 – Average time for file-posting metadata operations performed from a node the West Europe datacenter. The metadata server is respectively located within the same datacenter, the same geographical area and a remote region (log scale).

The scenario gets more complex considering that stream processing often leverages multi-site distribution. This is enabled either through deployment across several cloud datacenters or by combining edge and cloud processing as seen in section 1.1.2. Such multi-site deployments allow to aggregate resources beyond the limit of a single datacenter, replicate data on multiple geographically distributed locations or simply to deliver faster results, close to where the data is generated. The major issue with these deployments is the high latency between sites.

Problem: Huge latency for remote metadata access. The numerous metadata requests have to traverse the slow WANs connecting the datacenters (which in most cases are the property of the ISPs, so out of the control of the cloud providers). This limits drastically the achieved throughput of the streaming pipeline. A simple experiment conducted on the Azure cloud and isolating the metadata access times for up to 5,000 files (Figure 7.1) confirms that remote metadata operations take orders of magnitude longer than local ones. Clearly, multi-site stream processing calls for appropriate metadata management tools, that build on a consistent, global view of the entire distributed datacenter environment.

Solution: Adaptive decentralized metadata handling. Metadata management is not a new problem. It was actively studied in the context of parallel file systems and workflow management. In particular, streams and workflows share the same execution model (i.e., directed acyclic graphs) and the underlying infrastructures (i.e., multi-site clouds) where these graphs are deployed. It is then natural to take inspiration from the workflow metadata management and adapt it to the streaming context. In geographically-distributed environments, decentralized metadata management has proven a good option to keep up with the processing [14].

In this chapter we introduce strategies that leverage stream and workflow semantics in a 2-level metadata partitioning hierarchy that combines distribution and replication. We analyze stream metadata by their frequency of access and denote *hot metadata* to the most frequently required (conversely, *cold metadata*) [15]. We develop an approach that enables timely propagation of hot metadata while delaying cold metadata operations. This action

reduces network congestion by limiting the number of operations sent through high latency networks, thereby improving the overall execution time. We are able to obtain as much as 30 % gain in execution time for a parallel, geo-distributed real-world application across 4 cloud datacenters and up to 50 % for a metadata-intensive synthetic benchmark, compared to a baseline centralized configuration.

7.1 Strategies for multi-site metadata management

> ❝ *We propose to evaluate different metadata handling strategies, that result from exercising distribution and replication techniques across the available datacenters.* ❞

Stream processing contains a variety of relations, that can be as simple as a sequence of data dependencies between operators, or as complex as a mixture of multiple inputs, parallel operators and multiple outputs. However, some common data access patterns can be identified from these graphs: pipeline, gather, scatter, and parallel; the stream graphs deployed by SPEs are typically a combination of them, as seen in section 3.2.

Given the unlimited possible combinations of data access patterns, a single approach to mitigate the potential metadata bottleneck in multisite environments will certainly not fit all. Therefore, we propose to evaluate different metadata handling strategies, that result from exercising distribution and replication techniques across the available datacenters. We opt to keep all *metadata in memory* (in a uniform *DHT based cache*), which we make practical by reducing the per file metadata. That is we only store the information necessary to locate files and we don't keep additional POSIX type metadata, like permissions, since they are normally not used by the SPEs. Metadata are *distributed across cloud datacenters* following a *2-level hierarchy*: they are first partitioned to the datacenters where they are likely to be used (leveraging information from the SPE) and then replicated to other datacenters. With this approach, updates can be applied by only updating shares in one datacenter and propagating them to other datacenters.

Eventual consistency for geo-distributed metadata. Unfortunately, in a multisite cloud, this propagation and even simple metadata operations might take long time, particularly when the datacenters are geographically distant. In order to maintain a fully consistent state of the system, all nodes would have to wait until the newest operations are acknowledged by every member of the network. This is evidently inefficient considering the potentially long (physical and logical) distance between instances and the large number of metadata operations generally performed. Therefore we argue for a system where every metadata update is guaranteed to be *eventually* successful, but clearly not in real-time. That is, rather than using file-level eager metadata updates across datacenters, we favor the creation of batches of updates for multiple files. We denote this approach *lazy metadata updates*: it achieves low user-perceived response latency and high scalability in a widely distributed environment by asynchronously propagating metadata updates to all replicas after the updates are performed on one replica.

This lazy approach only guarantees *eventual consistency*. However, this is perfectly in line with the the typical data access patterns in multi-site stream processing: write once/read

many times, with readings occurring in two situations. For intermediate results, data are used as input for the next operator(s) in the stream graph, but in these cases the engine scheduler takes care to schedule the task close to the data production nodes (i.e., in the same datacenter) so the metadata updates are instantly visible here. For final results, data might be accessed from remote locations, but typically this a posteriori analysis takes places long after the execution has finished, leaving enough time for the lazy updates of the metadata to propagate. So, in both cases, the eventual consistency is not affecting the application performance or coherence.

In the remainder of this chapter we identify as *metadata registry* the instance or set of distributed instances in charge of managing metadata entries (shown as red diamonds in Figure 7.2). Every metadata registry instance is reachable by every node in the network. For the sake of practicality, we denote as a *read* the action of querying the metadata registry for an entry, and as a *write* the publishing of a new entry. Note that since a metadata entry can be created by one node and subsequently updated by others, a *write* operation actually consists of a look-up read operation to verify whether the entry already exists, followed by the actual write. File metadata entries are stored following a *key-value* approach, where the key is determined by a unique name of a file and the value indicates the node(s) where it is located.

We analyze the impact of high latencies as a consequence of the physical distance between an execution node and the corresponding metadata registry instance to/from where it will write/read an entry. The choice of an instance will depend on the management approach that we select; such approaches are described later in this section. Independently of the approach, in the remainder of this chapter we use the following terms to qualify physical distance between a node and a metadata registry:

Local: the node and the metadata registry are in the same datacenter;

Same area: the node and the registry are in different datacenters of the same geographic area (i.e., both datacenters are located in Europe);

Geo-distant: the datacenters are in different geographic areas (i.e., one in Europe, the other one in the US).

The last two situations can also be referred to as *remote*. Our design accounts for several datacenters in various geographic regions in order to cover all these scenarios. We have focused on four metadata management strategies, detailed below and depicted in Figure 7.2. In all cases, each datacenter is represented by a gray box, which contains a number of execution nodes (orange circles) and may contain an instance of the metadata registry as well (red diamonds). Solid lines connecting nodes and metadata registries denote metadata operations (reads or writes). The dashed lines represent a very large physical distance between datacenters; the ones on the "same side" of the line fit the *same area* scenario, whereas datacenters on "different sides" are *geo-distant*.

7.1.1 Centralized Metadata (Baseline)

Following the HDFS architecture, we first consider a single-site, single-instance metadata registry, independent of the execution nodes, arbitrarily placed in any of the datacenters

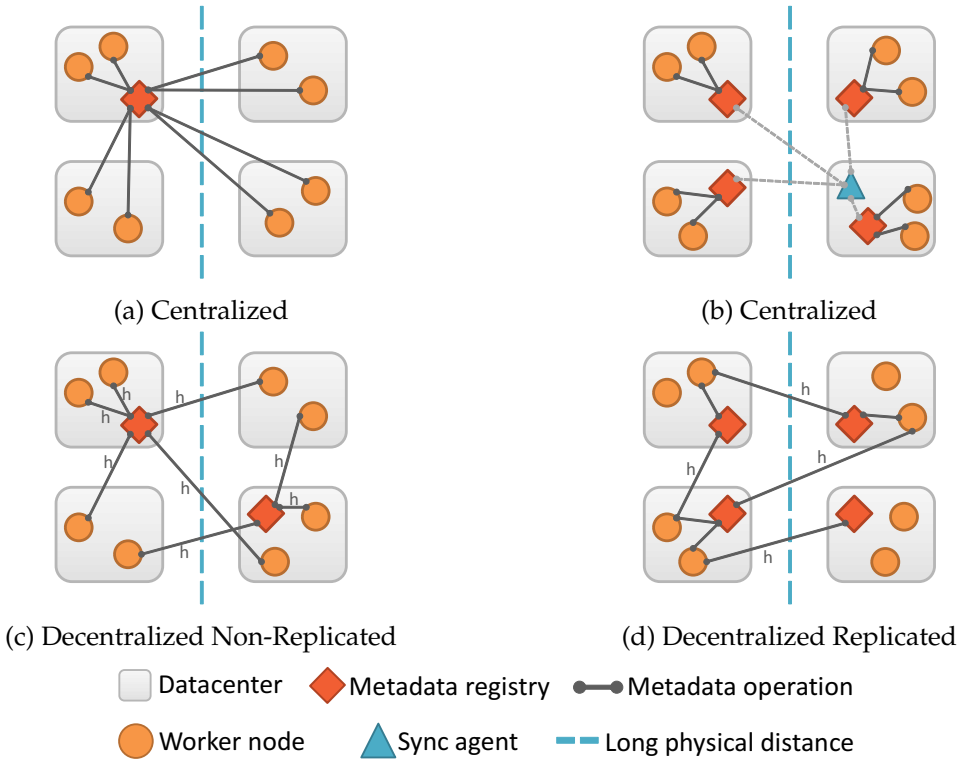


Figure 7.2 – Strategies for geographically distributed metadata management.

(Figure 7.2a), which will serve as a *state-of-the-art baseline*. In this setup, the application processes are run on nodes which are distributed both locally and remotely with respect to the site of the metadata registry. In the case of non-local accesses to the centralized metadata, high-latency operations may occur. While a centralized server guarantees a higher level of metadata consistency, it can quickly turn into a bottleneck as the workload increases. The purpose of this approach is to establish the (low) threshold at which these bottlenecks develop.

7.1.2 Replicated Metadata (on Each Site)

Our second strategy builds on the observation that local metadata operations are naturally faster than remote ones. Given a set of distributed datacenters, we place a local metadata registry instance in each of them, so that every node can locally perform its metadata operations. At this point, metadata information is processed quickly, but it would only be known at local level. Then, we propose to use a *synchronization agent*, a worker node whose sole task is to replicate across the network the content of the local metadata registries.

The synchronization agent systematically queries all registry instances for updates, and leverages non-metadata-intensive periods to perform batch synchronizations in all metadata registries. In this way, neither the execution nodes nor individual metadata registries are concerned with remote operations. The strategy is depicted in Figure 7.2b: the synchronization agent is presented as a blue triangle and can be placed in any site. The dotted lines between the agent and the registry instances represent the synchronization communication.

7.1.3 Decentralized, Non-Replicated Metadata

In our previous *replicated* approach, we took advantage of non-metadata-intensive computation time to maintain a synchronized distributed metadata registry. Even if a metadata registry instance is locally deployed in each datacenter, this strategy is still centralized in that it relies on a single synchronization agent, which can become a potential bottleneck, particularly in the case of a metadata-intensive workload. Even a multi-threaded synchronization agent might not provide a sufficiently fast synchronization to keep resource-waiting idle time at its lowest. Taking this into consideration, our third strategy favors *decentralization*, based on a distributed hash table (DHT). We maintain an instance of the metadata registry in each of the active sites. Every time a new entry is written to the metadata registry, we apply a hash function to a distinctive attribute of the entry (e.g., the file name). This determines the site where the entry should be stored by computing a simple modulo operation between the obtained hash value and the number of available sites; the hashing is indicated by an h in Figure 7.2c. A similar procedure applies for read operations to identify the metadata registry instance in charge of a given entry. Note that in this case the metadata are partitioned across the registry instances, so the contents of these instances are no longer identical in this strategy: each instance stores a share of the global set of metadata entries.

This approach involves remote operations: on average only $1/n$ of the operations would be local, where n is the number of sites. However, we notice two main improvements with respect to a centralized approach. First, as the registry is now distributed, metadata management is done in parallel by several instances, dividing the number of operations per instance and reducing the metadata-related idle time per node. Second, hash functions guarantee that identical input strings will always produce identical hash values. Hence, we can consistently determine the location of an entry from the hash value of its distinctive attribute. In this way, read operations require a single, direct lookup in a specific site, thus metadata operations remain of linear order, even when the registry is distributed.

7.1.4 Decentralized Metadata with Local Replication

Our last proposal aims at further leveraging the distributed setting of the metadata registry described above. As observed in Figure 7.1, local metadata operations take negligible time in comparison with remote ones, especially when the total number of operations becomes large, which is the case of data-intensive applications. Keeping this in mind, we propose to enhance the DHT-like approach with a local replica for each entry (Figure 7.2d).

Every time a new metadata entry is created, it is first stored in the local registry instance. Then, its hash value h is computed and the entry is stored in its corresponding remote site. When h corresponds to the local site, the metadata are not further replicated to another site but will only stay at that single location. For read operations we propose a two-step hierarchical procedure. When a node performs a read, we first look for the entry in the local metadata registry instance. With local replication, assuming uniform metadata creation across the sites, we have twice the probability to find it locally than with the *non-replicated* approach. Only if the entry is not available locally, it is then searched for in its remote location, determined by its hash value. We expect that the gain (in terms of latency and bandwidth) due to an enhanced probability to successfully look up metadata locally will be higher than the extra overhead added by local replication to the previous scheme.

(a) Without local replication (b) With local replication

Figure 7.3 – Decentralized metadata: local replicas avoid costly remote read operations.

To illustrate the benefits of local replication, we take the following interaction example involving two nodes n_1 and n_2 running in the same site s_1: n_1 writes an entry to the metadata registry, read by n_2, the location of the entry being determined by a hash function. Assume that the hash value places the entry in a geo-distant site s_2. Two situations may occur:

- In the *non-replicated* approach, both read and write operations would be remote and take up to 50x longer than a local operation (Figure 7.3a).
- With *local replication*, the write operation keeps a local copy and the subsequent read is performed locally, saving one costly remote operation and making reads up to 50x faster (Figure 7.3b).

7.1.5 Matching strategies to processing patterns

In this section, we address a question that arises from applying our strategies to different multisite stream processing: *which of the proposed strategies works better for each type of application?*

Following the previous design considerations and the various existing stream processing patterns, it is expected that no single hybrid strategy will outperform the rest. For instance, we have witnessed from our preliminary tests (as shown in Figure 7.1) that a centralized approach performs just as well as decentralization when a stream processing operates at small scale (i.e., using few nodes, managing at most 500 files each, running in a single site). Low latencies of intra-datacenter transfers coupled with the proximity of metadata servers enable a high throughput and reduce the access time. In such a scenario, the effort of putting in place a distributed metadata handling mechanism is likely not worth it. Therefore, we pose the question of *what strategy would best match what type of stream processing graph?* To answer it, we reason about the common stream processing characteristics and hypothesize the following situations, that are experimentally validated in section 7.3.

First, we believe that the *replicated* metadata registry with a *centralized synchronization* would perform at its best in a scenario where metadata operations are not so frequent. For instance, a stream processing which deals with few, very large files. In this case, the agent would have sufficient time to synchronize the registry instances and to provide consistency guarantees that enable easy reasoning on concurrency at application level.

Then, the decentralized strategies are expected to perform at their best with workloads managing a large number of small files. These strategies are of our particular interest, since this kind of workloads occur more frequently in large-scale applications. The *non-replicated approach* is foreseen to target applications with high degree of parallelism (i.e., following a scatter/gather pattern), where operators and data are distributed across datacenters. As we

mentioned, access to metadata remains linear across sites. Thus, we anticipate that the scalability and the throughput of the application will be preserved even for increased workloads.

Finally, we envisage that the *locally replicated* will fit better for stream graphs with a larger proportion of sequential jobs (e.g., with pipeline patterns). SPEs schedule sequential jobs with tight data dependencies in the same site as to prevent unnecessary data movements [129]. With this approach, we ensure that when two consecutive tasks are scheduled in the same datacenter the metadata are available locally. Even when a task is scheduled in a remote site, it will still be able to access metadata in linear time via the hash value.

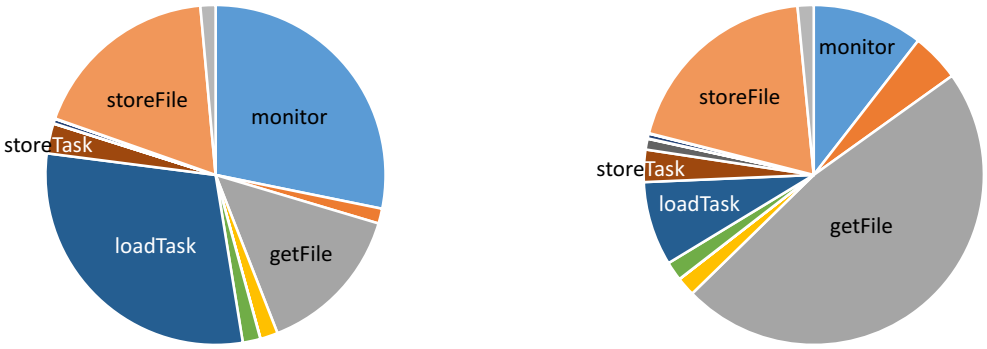
7.2 One step further: managing workflow hot metadata

As a next step we want to build on top of our previous ideas and not only distribute the metadata managing effort, but also give priority to relevant metadata in order to process it earlier. To this end, our focus is to explore the metadata access frequency, and identify fractions of metadata that do not require multiple updates. The goal is to enable a more efficient decentralized metadata management, reducing the number of (particularly inter-site) metadata operations by favoring the operations on frequently accessed metadata, which we call *hot metadata*.

What is "hot" metadata? The term *hot data* refers to data that need to be frequently accessed. They are usually critical for the application and must be placed in a fast and easy-to-query storage [97]. A typical approach is to keep these data in main memory while moving infrequently accessed (*cold*) records to a secondary storage, like flash [125]. In an analogous way, we transfer this concept to the context of workflow metadata, and we define *hot metadata* as the metadata that are frequently accessed during the execution of a workflow, which should be promptly available to the metadata server(s) of the system. Conversely, less frequently accessed metadata will be denoted *cold metadata* and will be given a lower priority over the network as we explain later in this section. The term "hot metadata" has been previously used with similar semantics for flash file systems [153], but it had not been applied to scientific workflows. In a multisite stream processing environment, we distinguish two types of metadata as *hot*: *task* and *file* metadata.

> ❝ *We define* hot metadata *as the metadata that are frequently accessed during the execution of a workflow, which should be promptly available to the metadata server(s) of the system.* ❞

Task Metadata are the metadata corresponding to the execution of tasks, these metadata include the specific operator for each task and its possible arguments, start time, end time, current status and execution location (site/node). Task hot metadata enable the workflow engine to search and generate executable tasks. During the execution, the status and site of the tasks are queried often in order to search for new tasks ready to execute and to determine if a job is finished. Accordingly, a task's status has to be updated several times along the execution. Therefore, it is important to propagate these metadata quickly to each site.



(a) Centralized (b) Distributed

Figure 7.4 – Relative frequency of metadata operations in Montage.

File Metadata that we consider as "hot" are those relative to the size, location and possible replicas of a given piece of data, which can be a file, or a block of a file, depending on the data size and the workflow engine data partitioning mechanism. Knowledge of file hot metadata allows the engine to place the data close to the corresponding execution task, or vice-versa. This is especially relevant in multisite settings: timely availability of file metadata would permit to move data before it is needed, hence reducing the impact of low-speed inter-site networks. In general, other metadata such as file ownership or permissions are not critical for the execution and thus regarded as cold metadata.

Separate management of hot and cold metadata. We studied sample executions of a typical scientific workflow, i.e., Montage [81], running 820 jobs and 57,000 metadata operations. They reveal that in a centralized execution, 32.6 % of them are *file* metadata operations (storeFile, getFile) and 32.4 % are *task* metadata operations (loadTask, storeTask), as shown in Figure 7.4a. In contrast, in a distributed run, up to 67 % are file operations, and task operations represent 11% (Figure 7.4b). These simple runs make evident that a significant percentage correspond to metadata that will not be needed right away or that is used for statistical purposes (mostly monitoring and node/site related operations); yet, in modern engines all metadata are handled in the same way. We therefore require a model that ensures that:

a) hot metadata operations are managed with high priority over the network, and
b) cold metadata updates are propagated only during periods of low network congestion.

For this purpose, the metadata servers should include a component which discriminates operations as cold or hot *before* propagating them through the network.

Adaptive storage for hot metadata. Job dependencies in a workflow graph form common structures, e.g., pipeline, data distribution and data aggregation [61]. Workflow engines usually take into account these dependencies to schedule the job execution in a convenient way to minimize data movements (e.g., job co-location). Consequently, different workflows will yield different scheduling patterns. In order to take advantage of these scheduling optimizations, we must also dynamically adapt the workflow's metadata storage scheme. However, maintaining an updated version of all metadata across a multisite environment consumes

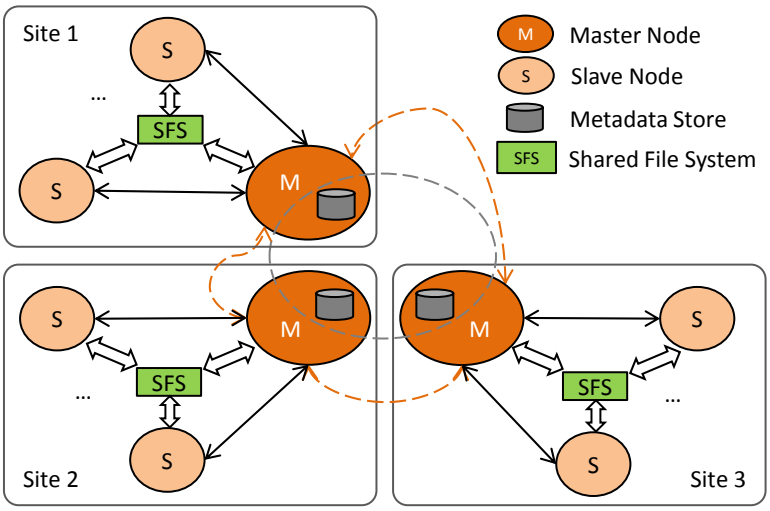


Figure 7.5 – Multisite execution architecture with decentralized metadata management. Dotted lines represent inter-site interactions.

a significant amount of communication time, incurring also monetary costs. In order to reduce this impact, we will apply adaptive storage strategies to our hot metadata handlers during the workflow processing, while keeping cold metadata stored locally and synchronizing such cold metadata only during the execution of the job. These strategies are based on those presented in section 7.1.

7.2.1 Architecture

The basis for our approach is a 2-level multisite architecture, depicted in Figure 7.5. Each level follows a master/slave model, as described below.

At the inter-site level all communication, data transfer and synchronization is handled through a set of master nodes (M), one per site. One site acts as a global coordinator (master site) and is in charge of scheduling jobs/tasks to each site. Every master node holds a *metadata store* which is part of the global metadata storage (shown in a grey dotted circle) that implements one of our distributed strategies and is directly accessible to all other master nodes.

At the intra-site level our system preserves the typical master/slave scheme widely-used today on single-site workflow management systems: the master node schedules and coordinates a group of slave nodes which execute the workflow tasks. All nodes within a site are connected to a shared file system to access data resources. At this level, all *metadata updates* are propagated to other sites through the master node, which classifies hot and cold metadata as we explain next.

Cold metadata operations must be identified *before* they are propagated to other sites through potentially slower networks. Therefore, we propose to add a filtering component, located in the master node of each site (Figure 7.6). When a master node receives a new metadata operation from a slave, the filter separates hot from cold metadata according to the

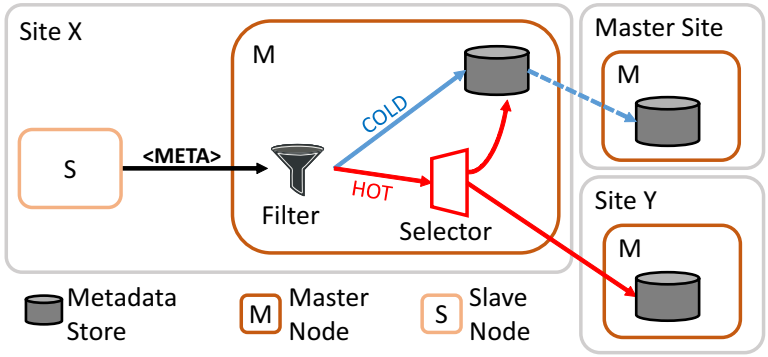


Figure 7.6 – The hot metadata filtering component.

criteria defined before, favoring the propagation of hot metadata and thus alleviates congestion during metadata-intensive periods. The cold metadata are kept locally in the meantime and transferred later to the master site (dotted line), which holds monitoring and statistical metadata. The storage location of the hot metadata is selected based on one metadata management strategy, as we develop in the coming section.

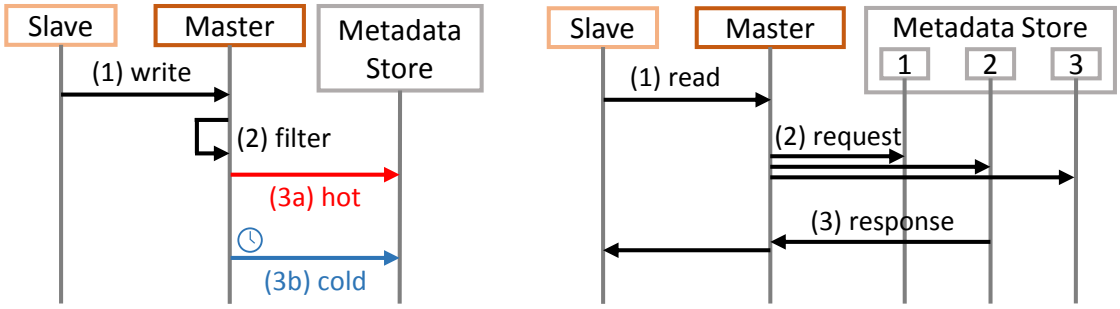
7.2.2 Protocols for hot metadata

We consider the three alternatives for decentralized metadata management explored in the previous section. In the following lines, we address their application to *hot* metadata. As exposed in Figure 7.2, all three scenarios include a metadata server in each of the datacenters where execution nodes are deployed. Unlike the former design, where the metadata registries where isolated entities, in this new 2-level architecture such metadata stores are now located in the master node of each site (Figure 7.5). The strategies differ in the way hot metadata are stored and replicated. We shortly recall their specificities below and explain how hot metadata entries are processed.

Local without replication (LOC). Every new hot metadata entry is stored at the site where it has been created. For read operations, metadata are queried at each site and the site that stores the data will give the response. If no reply is received within a time threshold, the request is resent. This strategy will typically benefit pipeline-like workflow structures, where consecutive tasks are usually co-located at the same site.

Hashed without replication (DHT). Hot metadata are queried and updated following the principle of a distributed hash table (DHT). The site location of a metadata entry will be determined by a simple hash function applied to its key attribute, *file-name* in case of file metadata, and *task-id* for task metadata. We expect that the impact of inter-site updates will be compensated by the linear complexity of read operations.

Hashed with local replication (REP). We combine the two previous strategies by keeping both a local record of the hot metadata and a hashed copy. Intuitively, this would reduce the number of inter-site reading requests. We expect this hybrid approach to highlight the trade-offs between metadata locality and DHT linear operations.



(a) Write (b) Read

Figure 7.7 – Metadata Protocols.

The following protocols illustrate our system's *metadata operations*. As we mentioned, metadata operations are triggered by the slave nodes at each site, which are the actual executors of the tasks.

Metadata Write. The process is depicted in Figure 7.7a: a metadata record is passed on from the slave to the master node at each site *(1)*. Upon reception, the master node will filter the record to determine if it corresponds to hot or cold metadata *(2)*. Hot metadata are assigned by the master node to the metadata storage pool at the corresponding site(s) according to different metadata strategies presented above *(3a)*. On the other hand, cold metadata are kept locally and propagated asynchronously to the coordinator site during the execution of the job *(3b)*.

Metadata Read. Each master node has access to the entire pool of metadata stores so it can retrieve hot metadata from any site. The *read* process (shown in Figure 7.7b) is as follows: First, a slave issues a read request *(1)*. When a read operation takes place in a remote location, a master node sends a request to each metadata store *(2)* and it processes the response that comes first *(3)*, provided such response is not an empty set (which would mean that such store does not keep a copy of the record). This mechanism ensures that the master node gets the required metadata in the shortest time.

7.2.3 Towards dynamic hot metadata

Applications evolve over their execution. This means that at a given point, some data might no longer be as relevant as they were initially; in other words, hot data become cold, or vice-versa. In the case of hot to cold data, file systems might remove them from the fast-access storage or even delete them; conversely, data that become relevant can be promoted to fast storage. Some frameworks assess the data "temperature" *offline*, i.e., they perform a later analysis on a frequency-of-access log to avoid overhead during the operation [125], however, this method is only useful when there are subsequent runs. More interestingly for us, *online* approaches maintain a rank on the frequency of access to the data alongside the execution, for example in adaptive replacement cache [137]. This phenomenon certainly occurs also at the metadata level; so, how could we handle these "temperature" changes in a multisite workflow engine? We look into the two situations.

Promoting cold to hot metadata. The idea would be to integrate an online ranking: given a workflow graph, a list of metadata attributes annotating the graph could be passed to the workflow engine in the same file. Then, the engine would monitor the access frequency of each of such attributes and periodically produce a ranking to verify whether an attribute is required more often, and thus *promote* it to hot metadata. The maximum number of attributes allowed as *hot* metadata could be also dynamically calculated according to the aggregated size of the metadata stores.

Downgrading hot to cold metadata. Less frequently accessed metadata could also be identified using the same attribute ranking approach as above. Upon identification, degrading some metadata to *cold* would also require a collection mechanism that ensures that metadata previously considered hot are removed from fast-access storage. Moreover, this action should take place during not-congested periods, or at the end of the execution so that it does not incur overhead. Taking one step further, we can consider an algorithm that determines the probability that metadata could become hot again later in the execution based on historical data; such metadata could be left in the storage, preventing I/O congestion.

7.3 Implementation and results

Our metadata management strategies are designed as a general purpose multisite metadata handling paradigm and not aimed to a specific cloud platform. The execution nodes can be mapped to regular virtual machines, and the metadata registries require in memory key-value store, which can be a generic, open solution such as Redis [163]. For validation purposes, in this implementation we use the Microsoft Azure Cloud [67] as a concrete example to demonstrate how to implement them in practice at Platform-as-a-Service (PaaS) level. We rely on the Azure SDK v2.5 for .NET which provides the necessary libraries for accessing and manipulating Azure features.

The Metadata Registry stays at the core of our implementation, as it serves as communication channel and distributed synchronization manager between all nodes in the network. We opted to implement the registry on top of the Azure Managed Cache[1] service [138]. Azure Cache provides a secure dedicated *key-value* cache that can be accessed remotely by means of a URI. Azure Cache allows to store any serializable object in the *value* field of an item. In order to guarantee the durability of our records, least-recently-used eviction and object expiration time properties were disabled. To allow concurrent access to the registry we chose the *optimistic concurrency* model of Azure Cache, which does not pose locks on the registry object during a metadata operation, leveraging the workflow's characteristic that data are written only once. The metadata registry has been implemented in C# and is composed of dedicated modules for caching, entries management, controlling and switching between strategies, and synchronization.

We used the Azure Cloud for our experiments. A significantly large amount of resources was made available to us thanks to our partnership with Microsoft Research through the

[1] By the time this manuscript was produced, Azure Managed Cache has been replaced by Azure Redis Cache as the recommended Azure Cache service.

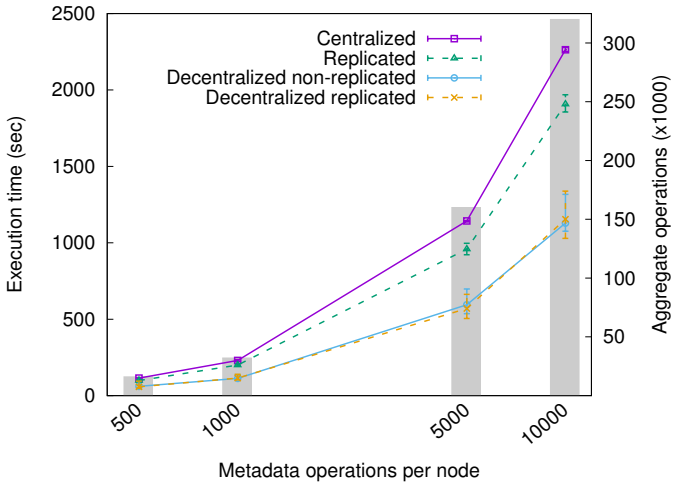


Figure 7.8 – Average execution time for a node performing write metadata operations.

ZCloudFlow project [180] We used small to mid-sized virtual machines, with a maximum of 8 cores per node. Azure Table key-value storage [96] was chosen for logging, since it handles concurrent writes from several nodes in a transparent way. Additionally, we leveraged Azure's PaaS solutions for several purposes, as we will detail later: Azure Managed Cache [138] for the metadata registries, Azure Cloud Services [140] for workflow management, and Azure Bus [139] for control messages.

7.3.1 Benefits of decentralized metadata management

This first set of experiments compares our three strategies: replicated, decentralized non-replicated, and decentralized replicated (section 7.1), to the centralized solution. Our testbed consisted of nodes distributed in four Azure datacenters: two in Europe and two in the US, using up to 128 *Standard_A1* virtual machines, each with 1 core and 1.75 GB of memory. For the Metadata Registry we deployed one *Basic* 512 MB instance of Azure Managed Cache per datacenter. To hinder other factors such as caching effects and disk contention, the metadata entries posted to the registry (e.g., create, update or remove) correspond to empty files.

Impact of metadata decentralization on makespan. We claim that the efficiency of our decentralized strategies becomes more evident in large-scale settings. The goal of the first experiment is to compare the performance of our implementation to the baseline centralized data management as the number of files to be processed increases. For this purpose, we keep a constant number of 32 nodes evenly distributed in our datacenters (i.e., 8 nodes per datacenter), while varying the number of entries to be written/read to/from the registry. To simulate concurrent operations on the metadata registry, half of the nodes act as writers and half as readers (i.e., 4 readers and 4 writers per datacenter). Writers post a set of consecutive entries to the registry (e.g., *file1, file2,* etc.) whereas readers get a random set of files (e.g., *file13, file201,* etc.) from it.

We measure the time required for a node to complete its execution, and obtain the average time for completion of all the nodes for each strategy. Figure 7.8 shows the results. We

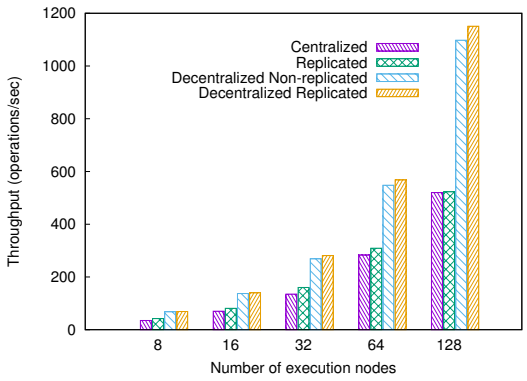


Figure 7.9 – Metadata throughput as the number of nodes grows.

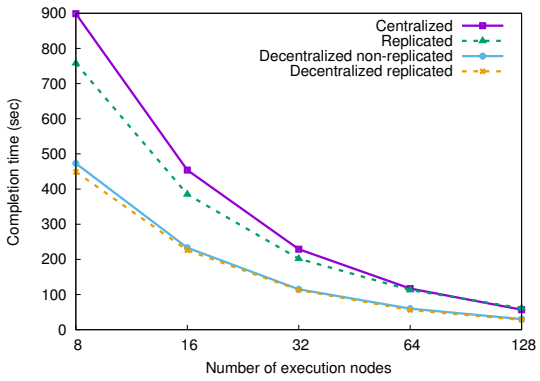


Figure 7.10 – Completion of 32,000 operations as the set size grows.

observe that for a rather small number of processed entries, none of our strategies significantly outperforms the centralized baseline in terms of overall execution time. They represent a gain of slightly more than 1 minute in the best case, which is rather low in our context. We infer that for small settings — up to 500 operations per node — a centralized approach remains an acceptable choice, since the effort of implementing a distributed solution would not be compensated by a meaningful gain.

However, as the number of operations grows, the improvement achieved by our strategies becomes more evident. Full metadata replication brings an average gain of 15 %; we attribute this simply to the fact that the metadata management duty is now distributed. In particular, the decentralized strategies (with and without replication) yield up to 50 % time gain compared to the centralized version. In the figure, the grey bars (linked to the right y-axis) indicate the aggregated number of operations in one execution. At the largest scale, a 50 % gain represents 18.5 minutes in a test with 320,000 metadata operations.

Scalability and concurrency sensitivity. In our next experiment, we evaluate the performance of our strategies when the number of nodes increases. Note that since each node acts also as a metadata client, this scaling translates into an increased concurrency as well. First, we measure the metadata throughput when increasing the number of nodes from 8 up to 128, with a constant workload of 5,000 operations per node. In Figure 7.9 we observe that the decentralized implementations clearly win: they yield a linearly growing throughput (given in operations per second), proportional to the number of active nodes. We only notice a performance degradation in the replicated scenario, intensified beyond 32 nodes. We assert that as the number of nodes grows, the single replication agent becomes a performance bottleneck; however, in smaller settings of up to 32 nodes, it still behaves efficiently.

To get a clearer perspective on the concurrency performance, we measured the time taken by each approach to complete a constant number of 32,000 metadata operations. Our results (Figure 7.10) were consistent with the previous experiment, showing a linear time gain for the centralized and decentralized approaches and only a degradation at larger scale for the replicated strategy.

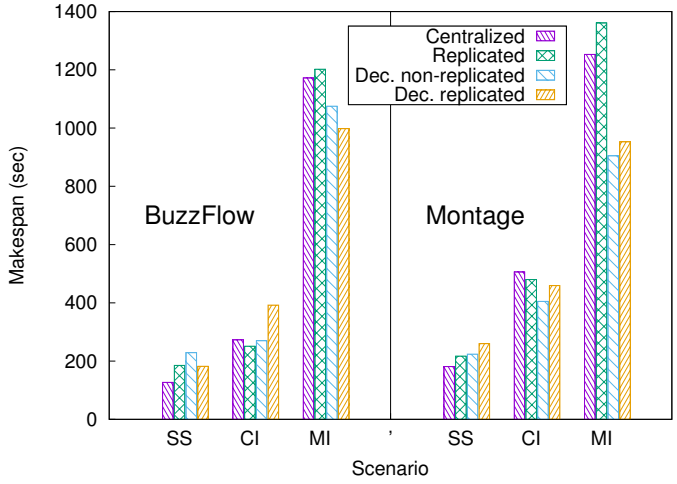


Figure 7.11 – Makespan for two real-life workflows. SS — Small Scale, CI — Computation Intensive, MI — Metadata Intensive.

7.3.2 Separate handling of hot and cold metadata

The second set of experiments relates to the implementation of *hot* metadata filtering, which we implemented within a the DMM-Chiron multi-site workflow engine [149]. DMM-Chiron was deployed on Azure cloud, using a total of 27 nodes of *standard_A4* virtual machines (8 cores, 14 GB memory). The VMs were evenly distributed among three datacenters: West Europe (WEU), North Europe (NEU), and Central US (CUS). Control messages between master nodes are delivered through the Azure Bus.

Hot metadata for different workflow structures. Our hypothesis is that no single decentralized strategy can fit all workflow structures: a highly parallel task would exhibit different metadata access patterns than a concurrent data gathering task. Thus, the improvements brought to one type of workflow by either of the strategies might turn to be detrimental for another. To evaluate this hypothesis, we ran several combinations of our strategies (local without replication - LOC, hashed without replication - DHT, hashed with local replication - REP, and a centralized baseline - CEN) with the featured workflows. Figure 7.12 shows the average execution time for the Montage workflow generating 0.5-, 1-, and 2-degree mosaics of the sky, using in all the cases a 5.5 GB image database distributed across the three datacenters. With a larger number of degrees, a larger volume of intermediate data is handled and a mosaic of higher resolution is produced.

In the chart we note in the first place a clear time gain of up to 28 % by using a local distribution strategy instead of a centralized one, for all the degrees. This result was expected since the hot metadata is now managed in parallel by three instances instead of one, and it is only the cold metadata that is forwarded to the coordinator site for scheduling purposes (and used at most one time).

We observe that for 1-degree mosaics and smaller ones, the use of distributed hashed storage also outperforms the centralized version. However, we note a performance degradation in the hashed strategies, starting at 1 degree and getting more evident at 2 degrees.

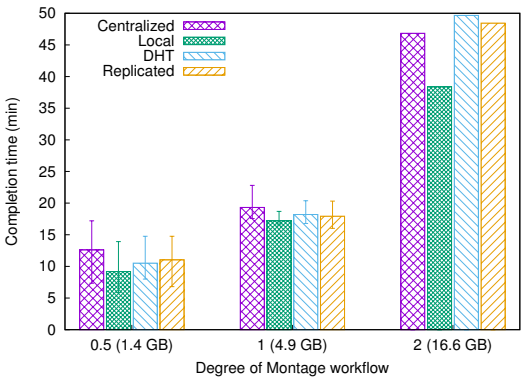


Figure 7.12 – Montage workflow execution time for the different strategies and number of degrees. Average intermediate data shown in parenthesis.

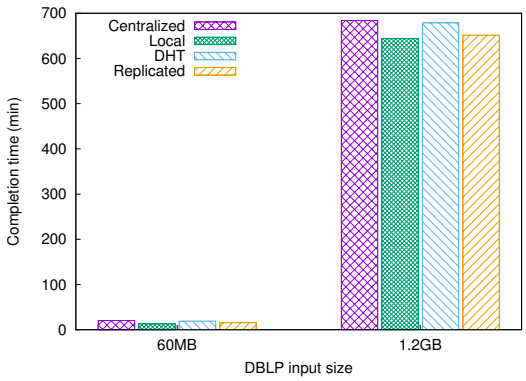


Figure 7.13 – Buzz workflow execution time. Left Y-axis scale corresponds to 60 MB execution, right Y-axis to 1.2 GB.

We attribute this to the fact that there is a larger number of long-distance hot metadata operations compared to the centralized approach: with hashed strategies, 1 out of 3 operations are carried out on average between CUS and NEU. In the centralized approach, NEU only performs operations in the WEU site, thus such long latency operations are reduced. We also associate this performance drop with the size of intermediate data being handled by the system. We try to minimize inter-site data transfers; however, with larger volumes of data such transfers affect the execution time up to a certain degree and independently of the metadata management scheme. We conclude that while the DHT method might seem efficient due to linear read and write operations, it is not well suited for geo-distributed executions, which favor locality and penalize remote operations.

In a similar experiment, we validated DMM-Chiron using the Buzz workflow, which is rather data intensive, with two DBLP database dumps of 60 MB and 1.2 GB. The results are shown in Figure 7.13; note that the left and right Y-axes differ by one order of magnitude. We observe again that DMM-Chiron brings a general improvement in the completion time with respect to the centralized implementation: 10 % for LOC in the 60 MB dataset and 6 % for 1.2 GB, while for DHT and REP the time improvement was less than 5 %.

Albeit the time gains perceived in the experiments might not seem significant at first glance, two important aspects must be taken into consideration.

Optimization at no cost. Our proposed solutions are implemented using exactly the same number of resources as their counterpart centralized approaches: the decentralized metadata stores are deployed within the master nodes of each site and the control messages are sent through the same existing channels. This means that such gains (if small) come at no additional cost for the user.

Actual monetary savings. Our longest experiment (Buzz 1.2 GB) runs in the order of hundreds of minutes. With today's scientific experiments running at this scale and beyond, a gain of 10 % actually implies savings of hours of cloud computing resources.

Our focus in this chapter was on handling metadata in a smart distributed way so that this improves job/task execution time when processing a large number of data pieces. While

our techniques show an improvement with respect to a centralized management, we also notice that when the scale of the processing and the size of data become larger, there is a performance degradation (see Figure 7.12) due to the increase of intermediate data transfers. To mitigate this degradation and allow for larger datasets, a data location awareness module should be added to the interface between the engine and our metadata manager.

Overall, we proved that a hybrid distributed/replicated solution can reduce, almost by half, the time to process inter-site metadata operations. We also showed that these solutions scale to hundreds of nodes. Then, we found a best-match between our decentralized strategies and metadata-intensive, large-scale workflows and streams. At the same time, we acknowledged the prevailing effectiveness of centralized metadata servers for single-site or smaller-scale workflows and streams.

Part III

Scalable Stream Ingestion and Storage

Chapter 8

KerA: Scalable Data Ingestion for Stream Processing

Contents

DATA INGESTION IS AN ESSENTIAL PART OF THE STREAM COMPUTING PIPELINE. Its role is to collect data from various sources (sensors, NoSQL stores, filesystems, etc.) and to deliver them for processing or storage. Ingestion acts as a broker between those numerous distributed data sources (called producers) and the SPEs (called consumers). Hence, the overall performance of the whole stream computing pipeline is limited by that of the ingestion phase However, as seen in section 3.3.1, state-of-art data ingestion systems trade performance for design simplicity. For instance, Apache Kafka uses static stream partitioning, which prevents elasticity and high throughput, and enables only offset-based record access, which limits usability.

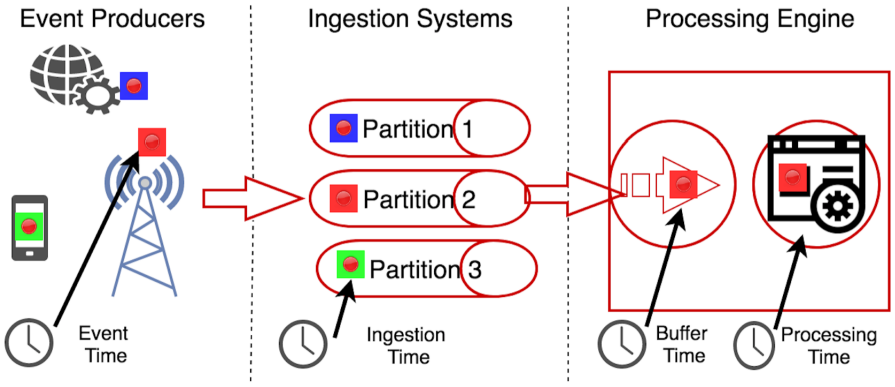


Figure 8.1 – Time domains on the stream processing pipeline: records are collected at *event time* and made available to consumers earliest at *ingestion time*, after the events are acknowledged by producers. Processing engines continuously pull these records and buffer them at *buffer time*, and then deliver them to the processing operators at *processing time*.

In this chapter we introduce KerA [6, 7, 8], a data ingestion framework that alleviates these limitations by means of dynamic partitioning and lightweight indexing. Experimental evaluations show that KerA outperforms Kafka up to 4x for ingestion throughput and up to 5x for the overall stream processing throughput. Furthermore, they show that KerA is capable to scale up and to deliver data fast enough to saturate the SPE acting as the consumer.

8.1 Impact of ingestion on stream processing

> *❝ The weak link of the three stage pipeline (collection, ingestion, processing) is the ingestion phase.❞*

SPEs rely on message brokers to decouple data sources from applications in order to hide the stream variability. For instance, events may arrive late, out of order, or simply too fast with respect to the processing capabilities of the engine. Thus, the weak link of the three stage pipeline (collection, ingestion, processing) 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.

Achieving all these objectives simultaneously is challenging, which is why applications rely on specialized ingestion runtimes to implement this phase. Several design dimensions of these frameworks are worth a closer look in order to identify the potential bottlenecks.

8.1.1 Time domains

Let us start with a vocabulary disambiguation. When processing streams of unbounded events, the notion of time plays an important role as it is associated with all the phases of the

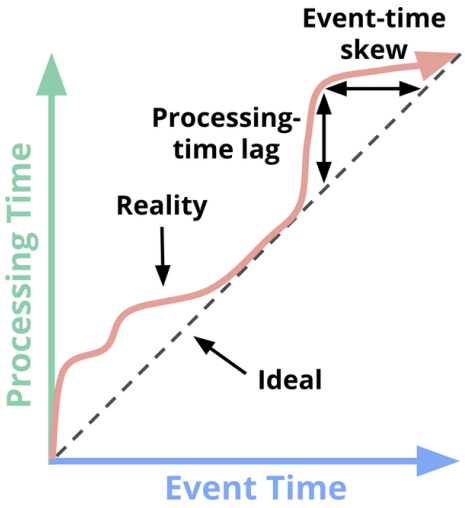


Figure 8.2 – Time-domain mapping. In reality, processing-time lag and event-time skew at any given point in time are identical [33].

events lifecycle, with distinct semantics for each. More particularly, there are four inherent domains of time to consider, illustrated in Figure 8.1 and explained below:

Event time is the time at which the event itself actually occurred, i.e., a record of system clock time (for whatever system generated the event). It is typically encoded by a timestamp attached to the data record that the source emits.

Ingestion time is the moment when the ingestion framework finished handling the collection process and makes the event available to registered consumers.

Buffer time is the time at which the SPEs pull the records from the brokers (i.e., ingestion frameworks) and are ready to process them.

Processing time is the time when the event arrives in the message queue for processing, measured by the clock of the node that processes it.

Problem: Variable lag between the event time and the processing time. In most setups, the event time for a given event essentially never changes. However, the processing time changes constantly for each event as it flows through the pipeline and time marches ever forward. The reasons for that are manyfold: data production spikes, volatile event producers, communication delays of different network paths, scheduling algorithms, pipeline serialization, queuing and backpressure effects from the stream consumers, etc. The end effect is a dynamically changing difference between the two time domains, which is often referred to as *event time skew* and is defined as *processing time - event time*.

In an ideal world, this skew would always be zero, with events being processed immediately as they occur. Reality is however much more mundane, and often what we end up with is more like Figure 8.2. In this example, the system lags a bit at the beginning of processing time, veers closer toward the ideal in the middle, and then lags again a bit toward the end. So, the skew between event time and processing time is not only nonzero, but often a highly variable function of the characteristics of the input sources, SPEs, and hardware.

Solution: Ingestion awareness of the correlation between the two domains. This dynamic variance in skew is very common in distributed data processing systems, and plays a big role in defining the functionalities and the objectives of an efficient ingestion layer. In particular, it tells that one cannot analyze data solely within the context of when they are observed by the processing pipeline but also considering when the events actually occurred. Unfortunately, the notion of windowing introduced by many historical systems did exactly that: they chopped the stream into finite datasets along temporal boundaries defined only by the processing time (i.e., processing-time windowing). This leads to out-of-order data and events ending up in the wrong processing-time windows. Instead, what we aim for is to maintain a *consistent correlation between processing time and event time* all along the ingestion phase in order to guarantee correctness.

8.1.2 Static vs. dynamic partitioning

Problem: Static partitioning leads to over-provisioning. State-of-art stream ingestion systems (e.g., [118, 49, 103]) employ a *static partitioning* scheme where the stream is split among a fixed number of partitions. Each partition is an unbounded, ordered, immutable sequence of records that are continuously appended. Producers accumulate records in fixed-sized batches, each of which is appended to one partition. Each consumer is assigned to one or more partitions and each partition is assigned to a single consumer. This eliminates the need for complex synchronization mechanisms but has an important drawback: the application needs apriori knowledge about the optimal number of partitions.

However, in real-life situations it is difficult to predict the optimal number of partitions, because this 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 the consumers 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 8.3. Each partition is represented by an index file for offset positioning and a set of segment files, initially one, for holding stream records. 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. 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 [145].

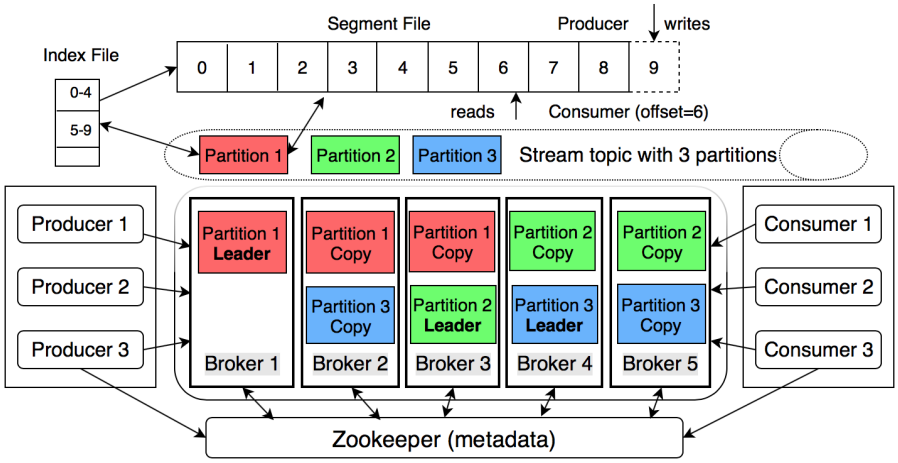


Figure 8.3 – Kafka's architecture illustrated with 3 partitions, 3 replicas and 5 brokers.

Solution: 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 define 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 makes each partition 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.

8.1.3 Record access

Problem: Offset-based record access creates unnecessary overhead. In Kafka, the brokers assign to each record of a partition a monotonically increasing identifier called *the partition offset*. This allows applications to get random access within partitions. However, *streaming applications normally access the records in sequential order*, so they don't actually need offset support. The rationale of providing random access is to enable 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.

Solution: 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.

8.2 KerA overview and architecture

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

8.2.1 Models

We start by presenting the data model used for record and stream representations, followed by the description of the dynamic partitioning model introduced by KerA.

Unified data model for streams

Stream records. A *stream* is an unbounded sequence of records that are not necessary correlated with each other. Each *record* of a stream is represented by an entry header which has a checksum covering everything but this field; the record is defined by a number of keys (possibly none) and its value, similar to the multi-key-value data format used in RAMCloud [151]. The record's entry header contains an attribute to optionally define a version and a timestamp necessary to efficiently enable key-value interfaces.

Chunks. Record entries are further grouped by producers into *chunks* (a chunk has a fixed size of up to 8-16 MB). The chunk aggregation is useful for two reasons. First, it gives clients the chance to efficiently (for metadata purposes) batch more records in a request in order to trade-off latency and throughput. Second, since each chunk is tagged with the producer identifier and with a dynamically assigned partition offset identifier (we explore offset details in the next section), this helps ensuring exactly once semantics and ordering semantics necessary for consistent ingestion and processing.

Segments. A producer client prepares and writes a request containing a set of chunks. Each chunk is acquired by the ingestion system and appended into a *segment* representing a pointer to a buffer managed by the system. A segment has a customizable fixed size (8-16 MB) necessary for efficiently moving data from memory to disk and backwards (segments have the same structure on both disk/flash and memory). A stream is composed of a set of uniform segments containing chunks of records of the same stream.

Groups of segments. In order to reduce the metadata necessary to describe the set of segments of a stream, we logically assemble a fixed number of segments into a group. Each stream is thus represented by a smaller, unbounded set of groups of segments.

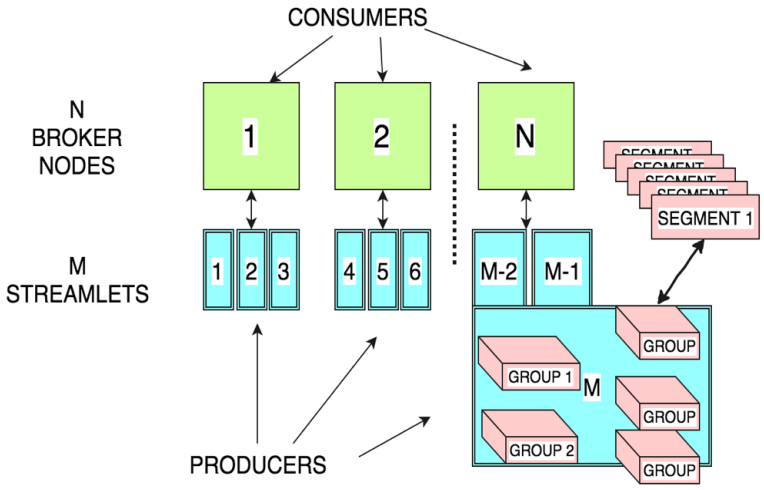


Figure 8.4 – A stream is logically composed of a set of streamlets.

Objects. KerA will also provide a storage layer for persistence. We have seen in section 2.3.1 that object-based storage is a versatile and efficient backend for both distributed file systems and databases. To this end, we model objects as *bounded streams*. They are represented by a fixed number of groups. For instance, we can think about the group as a similar concept to HDFS file's block. In order to reduce the metadata overhead for large unstructured objects (i.e., they do not benefit from the fine-grained record model), clients could create chunks having a single record with a large value (8-16MB). Since groups are split into small segments, this can help batch analytics load balance the processing of the group's segments in order to solve the straggler issues common in Big Data analytics. We think that our fine-grained representation of an object is a fair trade-off (due to small metadata overhead represented by chunks and segments) required to enable the management of streams and object by the same ingestion/storage system.

Dynamic partitioning model and offset indexing

KerA implements dynamic partitioning based on the concept of *streamlet* (Figure 8.4), which corresponds to the semantic high-level partition that groups records together. A streamlet is a container for fixed-size sub-partitions (groups of segments), with each group created dynamically. A stream has up to M streamlets that are initially created on a set of $N, N \leq M$, brokers (a broker is the entity offering pub/sub interfaces for handling streams). M represents the maximum number of nodes that can ingest and store a stream's records (ensuring horizontal scalability through migration of streamlets to new brokers).

Each streamlet is dynamically 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 created as needed, each of which can grow up to a maximum predefined size. To facilitate the management of groups and offsets, each group is further split into fixed-sized *segments*. The maximum size of a group is a multiple of segment size $P \geq 1$. To control the level of parallelism on each broker, only $Q \geq 1$ groups can be active at a given moment.

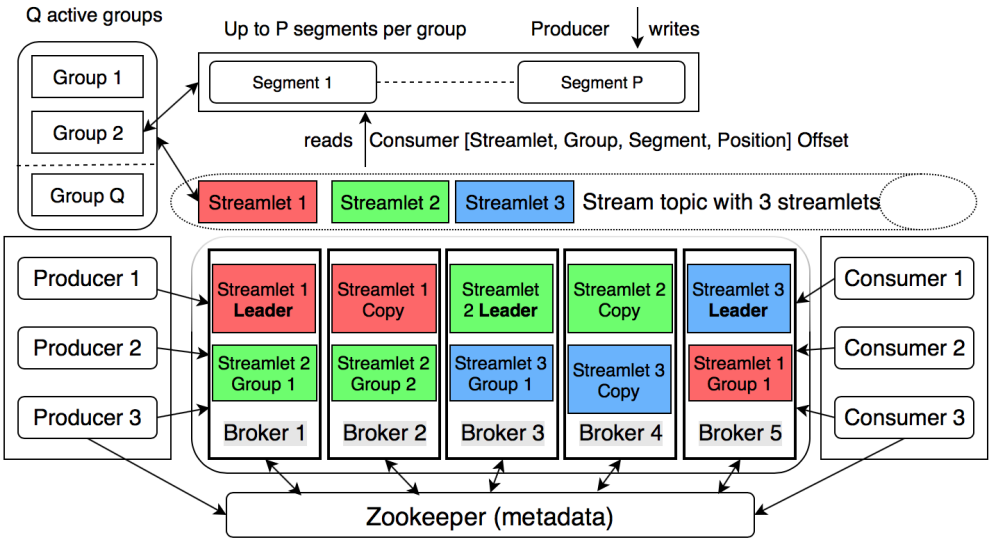


Figure 8.5 – Stream creation illustrated with 3 streamlets and 5 brokers.

Elasticity is achieved by assigning an initial number of brokers $N \geq 1$ to hold the streamlets M, $M \geq N$, as illustrated in Figure 8.5. Streamlet groups and their segments are dynamically discovered by consumers querying brokers for the next available groups of a streamlet and for new segments of a group. As more producers and consumers access the streamlets, more brokers can be added up to M. 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 (sub-partitions) to process. Replication in KerA can leverage its fine-grained partitioning model (streamlet-groups- segments) by replicating each group (i.e., its segments) on distinct brokers or by fully replicating a streamlet's groups on another broker.

In order to ensure ordering semantics, each streamlet dynamically creates groups (and their segments, initially one) that have unique, monotonically increasing identifiers. Brokers expose this information to consumers through RPCs that dynamically 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 we enable lightweight offset indexing optimized for sequential record access. In contrast, the static approach with explicit offsets per partition forces clients to query brokers to discover groups.

Stream records (grouped in batches at the client side) 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.

8.2.2 Favoring parallelism: consumer and producer protocols

Producers only need to know about streamlets when interacting with KerA. The input batch is always ingested to the active 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 (or create one if needed) corresponding to the Q^{th} 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. 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. 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.

8.2.3 Architecture and implementation

KerA's architecture (presented in Figure 8.6) is similar to Kafka's: a single layer of brokers (nodes) serve producers and consumers. However, in KerA brokers are used to discover stream sub-partitions. Kera builds atop RAMCloud's [151] framework in order to leverage its network abstraction that enables the use of other network transports (e.g., UDP, DPDK, Infiniband), whereas Kafka only supports TCP. This allows KerA to benefit from a set of design choices like *polling and request dispatching* [120] that help boost performance (kernel bypass and zero-copy networking are possible with DPDK and Infiniband). Kera consists of about 5K lines of C++ code for client and server side implementations.

Brokers manage the main memory of a server and handle multiple streams by ingesting stream batches into the active segment of the streamlet's active group (if one exists, otherwise a new group/segments is created). The Broker handles requests (for data and metadata) from both producers and consumers through RPCs. Each streamlet provides a number of active (open) groups (up to Q) and their corresponding active segments (i.e., pointers to memory buffers) into which the next writes are appended. Once a segment is closed (it suffers no more appends), a *segment manager* is responsible for providing a new segment.

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

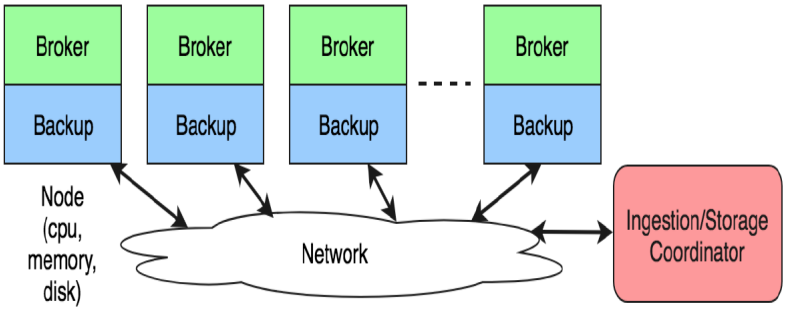


Figure 8.6 – KerA architecture: the Coordinator manages storage nodes on which live Brokers and Backups. Clients mainly interact with Brokers while Backups are simply used for storing stream's replicas.

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.

The Coordinator is a single service that handles the configuration of the cluster (e.g., adding or removing nodes, management of live or crashed brokers/backups) and the stream-streamlet-broker metadata (i.e., which broker is responsible for each streamlet of a stream). Clients first query the Coordinator in order to obtain and cache the association of Brokers and streamlets for a given stream. To avoid being a single point of failure, the Coordinator can be built using a fault-tolerant distributed consensus protocol similar to the implementation of the master processes of Hadoop, Spark, Flink or RAMCloud. The Coordinator is also responsible for the recovery of failed broker/backup services and for the migration of streamlets to other Brokers when necessary to respond to higher or lower ingestion load.

Backups store stream replicas. A Backup is configured with a limited number of in-memory segments (e.g., 256 segments of 8MB each) in order to acknowledge as fast as possible the replication RPCs. It is installed on servers backed by batteries in order to survive power failures. A Backup manages the storage provided by multiple disks in order to store segments (in a log-structured fashion) in multiple log files, one log for each device (disk/flash). The Backup maintains in memory the association of replicated streams with local segments; this metadata is useful for the recovery or migration of a stream's segments.

Since the number of in-memory segments managed by a Backup is limited, this setting pushes a restriction on the number of streams that can be efficiently and durably created and replicated. Moreover, the dynamic partitioning of a stream (up to the number of active groups multiplied by the number of streamlets) puts further pressure on this limitation. To maximize the number of active streams that can be created at a given time, we associate with each stream a set of *virtual logs* managing replicated *virtual segments*. Each virtual segment

contains pointers to chunks of a stream's partitions (that were acquired consecutively) and is replicated into a Backup's in-memory segment. The Backup eventually writes the segment on storage to ensure durability. As such, the Backup's segments contain chunks from possibly various groups of different streamlets of a stream. At recovery time, Backups read segments from disk and issue writes to the new brokers responsible to recover the lost data of a crashed broker. Each of these requests is handled as a normal producer request (i.e., chunks are ingested into their respective groups) while metadata is safely reconstructed.

8.2.4 Fast crash recovery for low-latency continuous processing

To support durability, replication, and fast crash recovery we rely on the techniques [150] introduced in RAMCloud, 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.

In case of Broker crashes we immediately recover the streamlet metadata (based on Backup's metadata) and re-enable producers to push the next stream batches. In parallel we proceed with the recovery of data as follows. Based on the last consumer offsets (e.g., every minute, for each consumer group, we store the last offsets used in read requests), we first recover the unprocessed groups. Then we proceed with the recovery of the other processed groups. In this way, readers continue to pull data for processing (although limited by recovery speed).

8.3 Experimental evaluation

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

8.3.1 Setup and methodology

We ran all our experiments on Grid5000 *Grisou* cluster [101]. 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 processing 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,

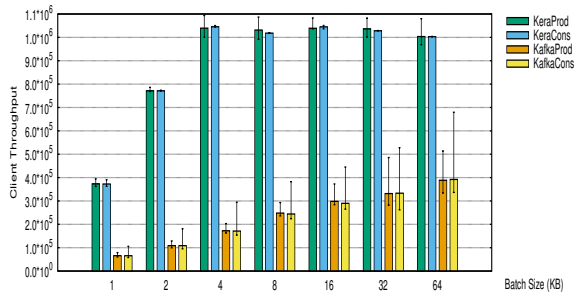


Figure 8.7 – Increasing the batch size (request size). Parameters: 4 brokers; 16 producers and 16 consumers; number of partitions/streamlets is 16; *request.size* = *batch.size* × 4 (number of partitions per node). On X we plot producer batch.size in KB, for consumers we configure a value 16x higher.

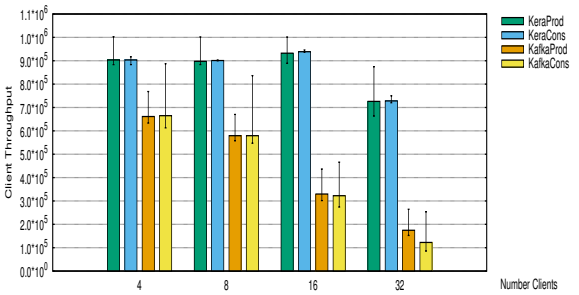


Figure 8.8 – Adding clients. Parameters: 4 brokers; 32 partitions/streamlets, 1 active group per streamlet; *batch.size* = 16KB; *request.size* = 128KB.

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. 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 [7]). 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).

8.3.2 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 overhead of offset indexing on brokers). In order to understand the application offset overhead in Kafka and KerA, we evaluate different scenarios, as follows.

Impact of the batch/request size. By increasing the batch size we observe smaller gains in Kafka than KerA (Figure 8.7). 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, before appending a batch to a partition, Kafka iterates

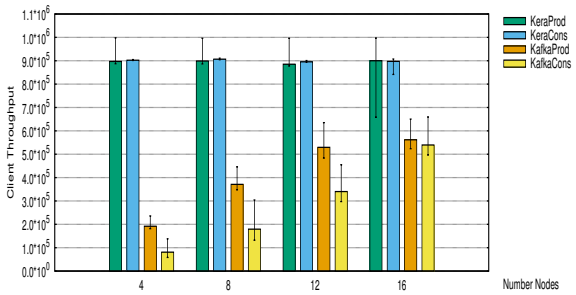


Figure 8.9 – Adding nodes (brokers): 32 producers, 32 consumers, 256 partitions, 32 streamlets, 8 active groups per streamlet; *batch.size* = 16KB; *request.size* = *batch.size*× partitions number/active groups per node.

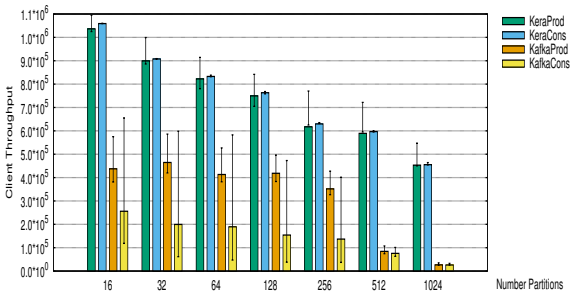


Figure 8.10 – Increasing the number of partitions and respectively streamlets. Parameters: 4 brokers; 16 producers and 16 consumers; *request.size* = 1MB; *batch.size* = *request.size*/*partitions number*.

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 concurrent 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 8.8, 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 8.9, 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 8.10 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 1,024 partitions) is due to its offset-based implementation, having to manage one index file for each partition.

8.3.3 Discussion

With KerA one can leverage the streamlet-group abstractions in order to provide applications an unlimited number of sub-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 1,024 active groups (fixed-size sub-partitions) compared to less than 50K records per second with Kafka providing the same number of partitions. KerA provides higher parallelism to producers and consumers resulting in higher ingestion/processing client throughput than Kafka.

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. 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.

Chapter 9

Týr: Transactional, Scalable Storage for Streams

Contents

STREAM STORAGE LAGS BEHIND PROCESSING in terms of number of solutions and their delivered throughput and I/O access performance. We have seen in Part 1 of this manuscript that:

a) state-of-the-art stream storage systems that leverage *distributed file systems, log-based storage or databases introduce processing skews* and significant memory, disk and network overheads (sections 2.3.1 and 3.3.2);

b) transactional support is required by real-time analytics but it is currently missing from stream storage solutions or comes at the cost of performance (section 4.2).

Concerning the first observation, we have noticed that the vast majority of the studied distributed file systems, log-based storage and databases are built as thin layers atop object storage. In order to reduce their overheads, one idea would be to use the object storage directly, bypassing the file, log or key-value semantics.

As for the second remark, consistency is a common requirement for all storage systems, not only for those dedicated to streams. Providing consistency has always posed hard challenges to system developers because of the wide range of conditions that can potentially result in data corruption. Besides system and network failures common to all systems, concurrent, conflicting storage requests can also yield incorrect results. This is the case when the storage system purposely relaxes these guarantees, such as Ceph [187] for writes operations crossing a chunk boundary [73]. Solutions to ensure consistency either rely on *applicative* locks or leverage synchronization primitives provided by a layer down the *storage* stack. Defining at which layer to implement consistency management is a hard question. However, in case of streams, the only viable option is the *storage-based consistency*. Implementing it at the application level would mean to burden the online processing pipeline with heavy synchronisation constructs that would render obsolete the real-time promise of the results.

In this chapter, we propose the design of Týr [10, 11], a blob storage system designed from the ground up to support multi-object synchronization with transactional semantics. Týr notably combines predictable data distribution that is central for low-latency with multi-version concurrency control that is key to high-performance under high write contention scenarios.

9.1 Blobs for stream storage

Problem: Stream storage is not well served by distributed file systems and databases. Most SPEs today still rely on distributed file systems (e.g., HDFS in Spark and Flink), which used to provide a good balance between performance and versatility. However, the fast growth of the data volumes and throughput requirements of data-instensive applications running on a new generation of extreme scale platforms pushes this storage paradigm to its limits. Also, looking at the *stream data model, its unbounded nature better fits an unstructured storage* (able to store any byte sequence of data). As per the data access patterns, the *vast majority of the storage calls are simple reads and writes*. Finally, most streaming applications simply *do not require the complex directory hierarchies and the heavyweight POSIX functionalities*. As such, there is a clear trend in the streaming community to move away from distributed file systems altogether.

While key-value or other NoSQL data stores provide a highly efficient data model able to cope with more than a billion I/O operations per second on a single server [128], they do lack support for the *large, mutable data objects storing huge streams*, that are required by a wide range of extreme scale stream processing applications.

> ❝ *Support for large objects with a flat namespace and a simple API is precisely the main target of blob storage systems.* ❞

Solution: Blob storage. Such support for large objects with a flat namespace and a simple API is precisely the main target of blob storage systems. They provide a middle-ground approach combining the horizontal scalability on par with that of key-value stores and a virtually infinite capacity as distributed file systems. More interestingly, they fit to the data access model required by streaming applications: reads and writes are easily covered by object storage systems, while the rest of the storage calls can be mapped atop them. Overall, their flat namespaces are friendly to applications by offering low data-management overhead. Furthermore, the deployment of blob storage does not conflict with legacy streaming applications due to their capability to serve as a building block for highly efficient file system interfaces. As such, *blob storage systems are a strong candidate for scalable stream storage*.

Limitations of state-of-the-art blob storage call for dedicated solutions for streams. However, despite being a very promising alternative to distributed file systems, blob storage systems are not exempt for limitations. Specifically, significant challenges lie in their consistency guarantees when faced with concurrent, conflicting writes. While some systems choose not to provide any such guarantees, some do so at the expense of a substantial storage operation latency increase compared to key-value stores. Rados performance is excellent under low write contention, but its lock-based concurrency control limits its throughput in highly-concurrent use-cases. On the contrary, the multi-version concurrency control of BlobSeer supports its performance under high concurrency, but the distributed metadata tree that is core to its design induces a significant read latency that hinders application performance. For blob storage systems to be proposed as an alternative to file systems for streaming, we believe that a middle-ground approach is here also required: *providing configurable consistency guarantees while only exhibiting minimal overhead when no such guarantees are required*.

Strong consistency required by real-life use-cases. In addition, while these blob systems answer the needs of legacy applications, a range of advanced use-cases require stronger consistency semantics, ensuring the correctness of multi-objects operations. This is notably the case of *the concurrent updates of the shared state in Lambda architectures* (section 4.2.2) or *data indexing and aggregation* (section 9.4). These use-cases are illustrated by the design of the Warp Transactional Filesystem (WTF) [84] relying on transactions for metadata operations, or the reliance of HBase [95] on ZooKeeper [108] for synchronization purposes.

9.2 Design principles and architecture

In this section, we summarize the key design principles that lay the foundation of Týr.

9.2.1 Predictable data distribution

Similarly with Rados [188] as well as a number of key-value stores [80, 170, 115], Týr enables clients to access any piece of data without any prior communication with any metadata node. The data are distributed in the cluster using a combination of *consistent hashing* and *data striping* techniques.

Consistent hashing [113] is a well known technique to distribute data in a cluster. The unique key associated with each object is hashed with a function that is common to all nodes in the cluster. The hashed value is used to determine the server which is responsible for the data with this key. Each node in the cluster is responsible for a part of the whole hash value range. The benefit of this technique is twofold. First, it distributes the loads across the whole cluster and helps alleviate potential hot spots when the number of objects is high enough. Second, it obviates the need for a centralized metadata server. Indeed, any node in the cluster is able to locate any piece of data with only knowledge of the mapping between nodes and hash value ranges. Furthermore, when this mapping is shared with the clients, they are able to send requests directly to the nodes holding the value they seek to read or write. This consequently eliminates unnecessary hops for pinpointing the location of data in the cluster. Such feature is key to the low-latency of key-value stores such as Chord [171], Dynamo [80] or Riak [115].

Data striping complements this data distribution. Each blob in the system is split into *chunks* of fixed size, that are distributed over all nodes of the system using consistent hashing. Thereby, the I/O is distributed over a larger number of nodes, supporting the scalability and performance of the system. This principle also enables storing objects with a size greater than the capacity of a single machine. With a chunk size s, the first chunk c_1 of a blob will contain the bytes in the range $[0, s)$, the second chunk c_2, possibly stored on another node, will contain the bytes in the range $[s, 2s)$, and the chunk c_n will contain the bytes in the range $[(n-1) * s, n * s)$. The chunks being distributed in the cluster using consistent hashing, their size must be fixed.

Chunks are distributed in the cluster as follows. Given a hash function $h(x)$, the output range $[h_{min}, h_{max}]$ of the function is treated as a circular space, or *ring* (h_{min} sticking around to h_{max}). Each node is assigned a different random value within this range, which represents its position on the ring. For any given chunk n of a blob k, a position on the ring is calculated by hashing the concatenation of k and n using $h(k : n)$. The *primary node* holding the data for a chunk is the first one encountered while walking the ring passed this position. Additional replicas are stored on servers determined by continuing walking the ring until the proper number of nodes is found.

9.2.2 Transparent multi-version concurrency control

Versioning is the core principle we use for data management; a new data version is generated each time a blob is written to. Multi-version concurrency control isolates readers from concurrent writers. Essentially, it enables writers to modify a copy of the original data while enabling concurrent readers to access the most recent version. Týr does not expose multiple versions to the reader, keeping this complexity hidden from the clients. This principle yields the following advantages:

High throughput under heavy access concurrency. Compared to systems where locks are used to synchronize concurrent access to a given piece of data, versioning offers significant performance advantages. Once written, a data version is considered *immutable*. As such, the data is never overwritten; instead, writes operate on a *copy* of the current version instead of overwriting it. The main benefit of such approach is that readers are

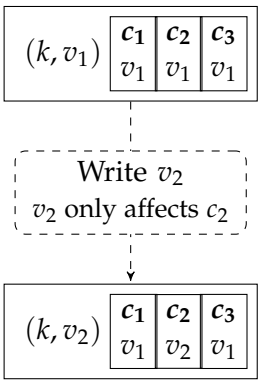


Figure 9.1 – Týr versioning model. When a version v_2 of the blob is written, which only affects chunk c_2, only the version of both the blob and c_2 is changed. The version id for both c_1 and c_3 remains unchanged. Sequential version numbers are used for simplicity.

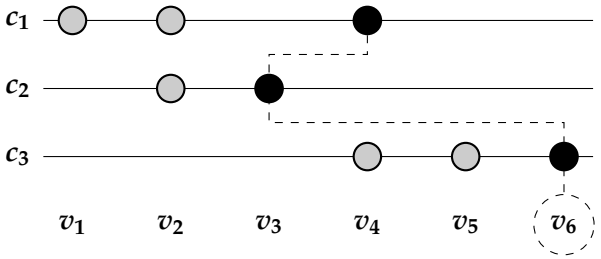


Figure 9.2 – Version management example. The version v_1 of this blob only affected the chunk c_1, v_2 affected both c_1 and c_2. In this example, v_6 is composed of the chunk versions (v_4, v_3, v_6). This versioning information is stored on the blob's metadata nodes. Sequential version numbers are used for simplicity.

not impacted by concurrent writers, and do not need to wait for conflicting write operations to finish. This contrasts with lock-based storage systems such as Rados, which typically offer degraded read throughput under heavy access concurrency because of lock contention.

Enhanced fault tolerance. A significant challenge when data is not versioned is the data recovery in the presence of failures. Indeed, should a fault occur during a write, the data could be left in an unconsistent state, hence rendering data recovery difficult. In contrast, since versioning ensures that writes only operate on a copy of the data, the previous version can be restored and the incomplete write discarded.

We propose to handle versioning at the chunk level, in contrast with traditional implementations of multi-version concurrency control. This obviates the need for centrally generating sequential version numbers, which are normally used to determine the ordering of successive versions. Generating such successive version numbers in a distributed context is a difficult problem. The common solutions either rely on a heavyweight consensus algorithm such as Paxos [122] or come at the cost of fault tolerance when a single sequencer is used as in BlobSeer.

In contrast, we propose to use non-sequential numbers for versioning. The ordering of successive versions of any given chunk is retained at each node storing a replica of this chunk. The blob version identifier is the same as the most recent version identifier of its chunks, as illustrated in Figure 9.1.

For any given write operation, only the nodes holding affected data chunks will receive information regarding this new version. Consequently, the latest version of a blob is composed of a set of chunks with possibly different version identifiers, as illustrated by Figure 9.2. To be able to read a consistent version of any given blob, information regarding

every successive versions of all chunks composing a blob is stored on the same nodes holding replicas of the first data chunk of the blob. These nodes are called *version managers* for the blob. Co-locating the first chunk of a blob and its version managers enables faster writes to the beginning of blobs. It also avoids using any metadata registry for version management.

9.2.3 ACID transactional semantics

ACID transaction support is the key feature driving the design of Týr. Transactions provide the user with an expressive way to indicate the relation between a set of queries. They ensure the correctness of the resulting multi-object operation that is either fully applied, either rejected as a complete unit. The lightweight transactional protocol is core to the internals of Týr. It notably ensures Týr linearizability, and supports its replication-based fault-tolerance as well as the internal version garbage collection. We argue that co-designing a storage system with its transaction processing provides the following features without requiring additional protocols.

Consistent data replication. Replicating an object for fault-tolerance is a significant challenge for distributed systems not relying on a centralized metadata management service. Indeed, all the replicas must agree on an ordering between the successive writes. A common method for doing so is to rely on logical clocks or on arbitrary ordering based, for example, on the local time the write has been applied on the server receiving the operation from the client. Transactions inherently enable the nodes holding the replicas of any given chunk to agree on a serializable order. From the point of view of the storage system, multiple replicas of a chunk are considered as different objects to which the same operation is applied in the context of a transaction.

Version bookkeeping. A significant challenge in versioned systems is to determine when it is safe to prune old versions of a chunk that are not used anymore. We propose to rely on the transaction algorithm so that all metadata nodes for a blob are able to agree on the older version of the blob to keep, and piggyback that information to the nodes storing chunk replicas of this blob.

Yet, designing an efficient transaction commit protocol in a distributed environment is hard. Traditional approaches used by relational database systems typically rely on centralized transaction managers aimed at coordinating transaction execution in a cluster. However, this approach has three main drawbacks. First, it creates a single point of failure in the system as well as a potential hotspot, hence defeating the distributed metadata managenent benefits for write scalability. Second, the cost of such transaction ordering is to be paid even when transactions are not conflicting with each other, hence hindering the write performance of the cluster. Finally, some form of complex, heavyweight static analysis is usually required to ensure that no long-running transaction is prioritized over and hence blocking small ones from committing.

The Warp [86] transaction chain protocol introduced in the HyperDex [85] key-value store completely decentralizes transaction processing, so that no transaction manager or defined consensus groups are required. It enables transactions to be ordered on-the-fly, delaying the synchronization until a conflict is detected. This significantly reduces the transaction

processing costs in cases where applications are specifically designed to prevent such conflicts from happening. Furthermore, this protocol allows for lock-free reads which, combined with the predictable data distribution, ensure a very low latency for read operations in most cases.

Besides their low overhead, transaction chains also ensure the scalability of the system. Indeed, Warp enables transactions to only involve the servers that are impacted by the transaction (that is, holding chunks that are read-from or written-to in the context of that transaction). This eliminates hot spots in the system, in addition to reducing communication costs.

9.2.4 Atomic transform operations

Transaction based coordination provides the user with a simple way to express interdependency between related storage operations. However, for simple common patterns, their cost can be further reduced. This is notably the case for read-modify-write operations that require two round-trips between the client and the server: one for reading the current value of the data, and a second to optimistically apply the modifications to the storage. There are two main issues with this. First, the two round-trips intuitively hinder the operation latency. Second, the transaction would abort should a conflicting transaction commit between the read and write operations, further increasing latency in addition to leaving up to the developer to handle retries when required.

This problem is well known and was studied extensively in the database and shared memory literature [160, 56, 173]. Two common alternatives to transaction processing for simple read-modify-write use-cases exist.

Conditional primitives such as compare-and-swap (CAS) provide a solid building block for higher-level operations. Yet, while they effectively decouple the read and write operation from the point of view of the storage system, they yield little to no performance gains when compared to transactions. This is because they still require a query to read the current value of the data, and because a failing CAS operation also requires the developer to manually handle failure scenarios. Nevertheless, they proved to be a solid alternative to lock-based synchronization in HPC systems [72].

Atomic transform primitives provide a way for developers to transfer the responsibility of the data modifications to the storage system. This is particularly interesting when the transform operation is simple, such as binary additions, multiplications or bitwise operations. These operations are key to efficient data aggregation (section 9.4). With this scheme, the client does not communicate to the storage cluster the new data to be written, but instead the modification to be applied to the current data. This scheme significantly cuts communication costs by enabling such operations to be performed with a single round-trip.

We advocate that the transactional foundation of the storage system we propose provides an ideal base for these operations to be supported. Indeed, transaction protocols already include such conditional checks. They can be extended to support both CAS and atomic transform primitives without additional overhead, ensuring the consistency of such operations during the commit phase.

9.3 Protocols and implementation

Designing the protocols enabling Týr users to read and write data is a difficult task. Indeed, the core design principles of data chunking and multi-version concurrency control raise additional challenges compared to other transactional storage systems. A key reason explaining this difficulty lies in the size of the objects being stored. Transactional key-value stores typically only support values of up to a few megabytes, that can be entirely stored on a single node. In transactional file systems such as the Warp Transactional Filesystem, which do not propose direct read access to the data, the scope of transactions is limited to *metadata* operations in which the objects are small.

A naive approach consists of considering the multiple chunks that compose a single blob as independent objects, and to apply existing transactional protocols to coordinate write operations. Yet, such implementation would cause significant problems for read operations spanning multiple chunks, which would contradict the following two requirements:

Consistent multi-chunk reads: When serving reads spanning multiple chunks, it is important to ensure that the versions of all chunks that are returned together form a consistent version of the blob. This must be guaranteed even while processing concurrent, conflicting write operations. As such, the versions of each chunk cannot be considered independently from one another.

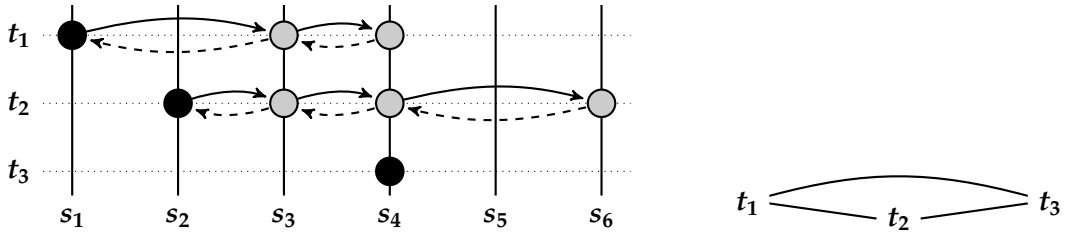
Repeatable reads: This isolation level is particularly desirable for large objects, whose potentially large size prevents them from being read entirely with a single read operation. Essentially, repeatable reads guarantees that, in the context of a transaction, all read operations to the same object will return data from the same consistent blob version. This is particularly challenging in presence of conflicting write operations, as the latest version of the blob can change between two read operations. While multi-version concurrency control solves part of the issue by ensuring that older versions are never overwritten, it does not bring a solution to all problems. In particular, the storage system should ensure that an older chunk version is only deleted when it is safe to do so, *i.e.* when no running transaction could potentially require that version of the chunk.

We solve these challenges by carefully designing the read and write protocols of Týr, which ensure that these two requirements are met.

9.3.1 Lightweight transaction protocol

Write coordination in Týr leverages the Warp optimistic transaction protocol, whose correctness and performance has been proven in [86]. This section provides an overview of Warp, upon which Týr is built.

In order to commit a transaction, the client constructs a chain of servers which will be affected by it. These nodes are all the ones storing the written data chunks, and one node holding the data for each chunk read during the transaction (if any). This set of servers is sorted in a predictable order, such as a bitwise ordering on the IP / port pair. The ordering ensures that conflicting transactions pass through their shared set of servers in the exact same order.



(a) Propagation graph (b) Dependency graph

Figure 9.3 – Warp lightweight transaction protocol overview. In this example, the directionality of the edge (t_1, t_2) will be decided by s_4, last common server in the transaction chains, during the backwards pass. Similarly, the directionality of (t_2, t_3) will be decided by s_4.

The client addresses the request to the first node of the chain, designated as the *coordinator* for that request. This node will validate the chain and ensure that it is up-to-date according to the latest ring status. If not, that node will construct a new chain and forward the request to the coordinator of the new chain.

Commit protocol. A linear transactions commit protocol guarantees that all transactions are either successful and serializable, or abort with no effect. This protocol consists of one forward pass to optimistically validate the values read by the client and ensure that they remained unchanged by concurrent transactions, followed by a backward pass to propagate the result of the transaction (either success or failure) and actually commit the changes to memory. Dependency information is embedded by the nodes in the chain during both forward and backward passes to enforce a serializable order across all transactions. A background garbage collection process limits this number of dependencies by removing those that have completed both passes.

Transaction validation. The coordinator node does not necessarily own a copy of all the chunks being read by every transaction, which are distributed across the cluster. As such, one node responsible for a chunk being read in any given transaction must validate it by ensuring that this transaction does not conflict nor invalidates previously validated transactions, for which the backward pass is not complete. Every node in the commit chain ensures that the transactions do not read values written by, or write values read by previously validated transactions. Nodes also check each value against the latest one stored in their local memory to verify that the data was not changed by a previously committed transaction. The validation step fails if transactions fail either of these tests. A transaction is aborted by sending an abort message backwards through the chain members that previously validated the transaction. As soon as the forward pass is completed, the transaction may commit on all servers. The last server of the chain commits the transaction immediately after validating it, and sends the commit message backwards to the chain.

Transaction serialization. Enforcing a serializable order across all transactions requires that the transaction commit order does not create any dependency cycles. To this end, a local dependency graph across transactions is maintained at each node, with the vertices being transactions and each directed edge specifying a conflicting pair of transactions. A conflicting pair is a pair of transactions where one transaction writes at least one data chunk read or written by the other. Whenever a transaction validates or commits after another one

at a node, this information is added to the transaction message sent through the chain: the second transaction will be recorded as a dependency of the first. This determines the directionality of the edges in the dependency graph. A transaction is only persisted in memory after all of its dependencies have committed, and is delayed at the node until this condition is met.

Figure 9.3 illustrates this protocol with an example set of conflicting transaction chains and the associated dependency graph. This example shows three transaction chains executing. Figure 9.3a shows the individual chains with the server on which they execute. The black dot represents the coordinating server for each transaction, the plain lines the forward pass, and the dashed lines the backwards pass. Because they overlap at some servers, they form conflicting pairs as shown on the dependency graph in Figure 9.3b. The directionality of the edges will be decided by the protocol, as the chain is executed.

9.3.2 Handling reads: direct, multi-chunk and transactional protocols

As per our requirements, the read protocol of Týr needs to ensure the correctness of consistent reads spanning multiple chunks while providing repeated reads isolation in the context of the transaction or direct reads whenever possible. Three cases need to be considered.

Direct reads

Direct reads is a desirable feature that stays at the core of the performance of key-value stores. It enables the client to address a read request directly to a node of the storage system that holds this information, without any prior communication with any metadata server. The client leverages the information it holds about the distribution of the data in the cluster to first locate a node holding the desired data, by hashing its key and chunk number (calculated from the data offset). The read request is sent directly to that node. The node responds with the data from the latest committed version of the chunk as per its internal dependency graph.

This protocol is the most efficient Týr implements, only involving a single node of the storage system. However, it is not applicable to multi-chunk reads as none of the servers involved do have information about the set of chunk versions that form a consistent view of the blob. For the same reason, inside a transaction, it cannot offer repeatable reads. As such, the Týr client only uses this protocol for reads spanning exactly one chunk performed outside the context of a transaction.

Multi-chunk reads

When reading a portion of a blob which overlaps multiple chunks, it is necessary to obtain the version identifiers of each of those chunks forming a consistent version of the blob. Only the version manager nodes for the blob to be read holds complete information about the chunk versions that form a consistent blob version. As such, the protocol used for served multi-chunk reads leverages these nodes to determine which versions of each accessed chunk to read from.

To perform a multi-chunk read, the client directs its request to any of the version manager nodes for the blob. Similarily to the direct read protocol, these nodes are determined using

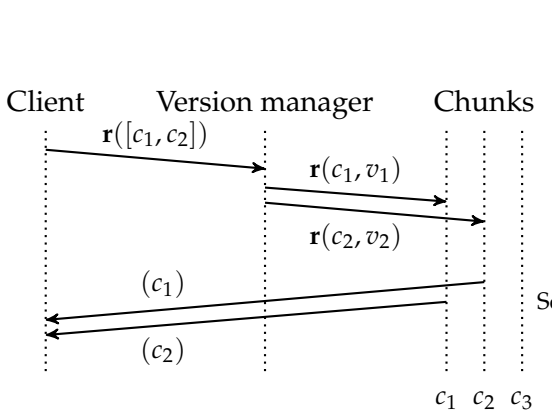


Figure 9.4 – Týr read protocol for a read spanning multiple chunks. The client sends a read query for chunks c_1 and c_2 to the version manager, which relays the query to the servers holding the chunks with the correct version information.

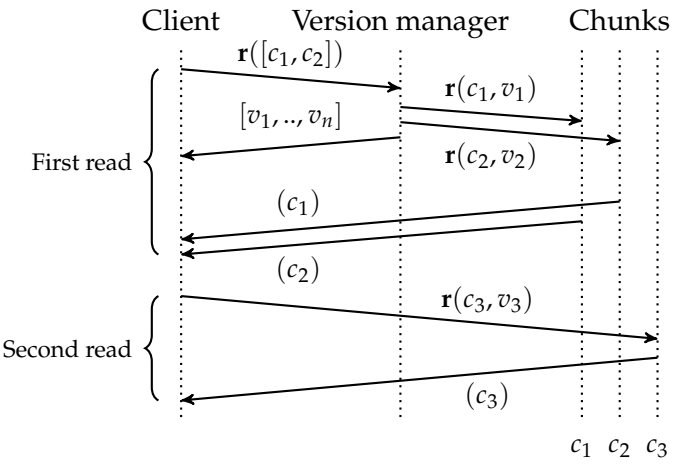


Figure 9.5 – Týr read protocol inside a transaction. The client sends a read query for chunks c_1 and c_2 to the version manager, which relays the query to the servers holding the chunks with the correct version information, and responds to the client with a snapshot of the latest chunk versions. Subsequent read on c_3 is addressed directly to the server holding the chunk data.

the client knowledge about the token ring, the version managers being the same servers holding data for the first chunk of the blob. From the version information it holds about the blob, the version manager determines, for the most recent committed version of the blob, the versions of each of the chunks accessed by the read operation. It also uses its own knowledge of the token ring to locate the servers holding the data for these chunks, selecting one at random. The read request is forwarded in parallel to each of these servers, including in the request the exact version identifier to read from. Each server replies directly to the client with the requested data. The client reconstructs the data in memory as it receives responses for each chunk. The protocol is illustrated by Figure 9.4.

Transactional, repeatable reads. In the context of a transaction, a further modification of the multi-chunk read protocol is required to provide repeatable reads. The transactional read algorithm is essentially a combination of the multi-chunk and direct read protocols. It combines the correctness of the former for the first read request with the simplicity and velocity of the latter for subsequent requests. A key difference enabling for repeatable reads is that the version manager node returns to the client the whole set of version identifiers for each chunk that together form the last committed blob version. This version information is cached by the client in the context of the transaction. Upon executing subsequent read requests for the same blob, the client extracts from that cached version information the version identifiers for each chunk to be accessed. The data is requested directly from the servers holding the chunk data as in the direct read protocol. It includes in the request message the version identifier to read from. The whole process is illustrated by Figure 9.5.

9.3.3 Handling writes: transactional protocol, atomic transforms

The complexity of version management is taken care of by the read protocol. As such, the basic write protocol of Týr is largely based on the unmodified Warp protocol. We introduce two modifications that are necessary for keeping the metadata managers up-to-date, and supporting in-place, atomic transform operations.

Atomic transforms

Atomic transforms provide a convenient way to express simple binary operations, such as arithmetic or bitwise operations. They operate on a binary value of fixed size located at a specific offset of the blob. The size of the binary value depends on the operation applied, but typically does not exceed 64 bits. We modify the write protocol to support atomic operations. Two cases are possible:

The binary value to be modified is contained in a single chunk. Handling of this case leverages the consistent ordering natively provided by Týr transaction protocol. During the forward pass, an atomic update operation is handled as any write operation. During the backwards pass, all servers holding a replica of the binary value to be modified atomically apply the atomic operation.

The binary value to be modified spans multiple chunks. Applying the operation in this context is challenging, as the value to be modified is not fully contained on a single machine. When such case is detected, Týr internally transforms the operation into a two-step read-update-write operation that is performed by the storage server on user's behalf.

Bookkeeping: purging expired versions

Týr uses multiversion concurrency control as part of its base architecture in order to handle lock-free read / write concurrency. Týr also uses versioning in support of the read protocol, specifically to achieve write isolation and ensure that a consistent version of any blob can be read even in the presence of concurrent writes. A background process called *version garbage collector* is responsible for continuously removing unused chunk versions on every node of the cluster. A chunk version is defined as unused if it is not part of the latest blob version, and if no version of the blob it belongs to is currently being read as part of a transaction.

How to determine the unused chunk versions? The transaction protocol defines a serializable order between transactions. It is then trivial for every node to know which is the last version of any given chunk it holds, by keeping ordering information between versions. Determining whether a chunk version is part of a blob version being read inside a transaction is however not trivial. Intuitively, one way to address this challenge is to make the version managers of the blob responsible for ordering chunk version deletion.

9.3.4 Implementation details

We implement the design principles of Týr in a software prototype. This includes the Týr server, an asynchronous C client library, as well as partial C++ and Java bindings. The server itself is approximately 25,000 lines of Rust and GNU C code.

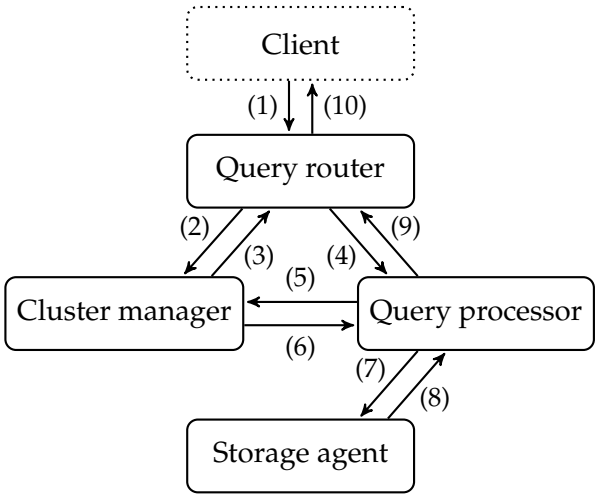


Figure 9.6 – High-level internal architecture of Týr server.

Týr is designed following a distributed, modular and loosely coupled architecture. Each node runs a set of different modules, executed on different threads. Figure 9.6 shows the interaction between these modules:

The Cluster Manager maintains the ring state between cluster nodes using a weakly consistent gossip protocol [77] to propagate information around the cluster (i.e. ring position allocations) and a ϕ accrual failure detector [105] to detect and confirm node failures.

The Query Processor coordinates the requests and handles the transaction protocol using cluster information from the cluster manager *(5, 6)*. It acts as the interface to the storage agent *(7, 8)*.

The Query Router is the main communication interface between a server and both the rest of cluster and the clients. It receives incoming client requests *(1)*, parses, validates and, if necessary, forwards them to the appropriate server according to cluster state information *(2, 3)*. It then forwards them to the query processor *(4, 9)*, and responds to the client *(10)*.

The Storage Agent is responsible for the persistent storage and retrieval of both data and version information.

9.4 Real-time, transactional data aggregation in support of system monitoring

In this section, we seek to prove the relevance of transactional blobs as a storage model for purpose-built application. In particular, we demonstrate the performance benefits of the two key features of Týr: transactions and atomic operations. We base these experiments on the ALICE (A Large Ion Collider Experiment) [29] real-world application from CERN (European Organization for Nuclear Research) [124]. In particular, we consider the needs of

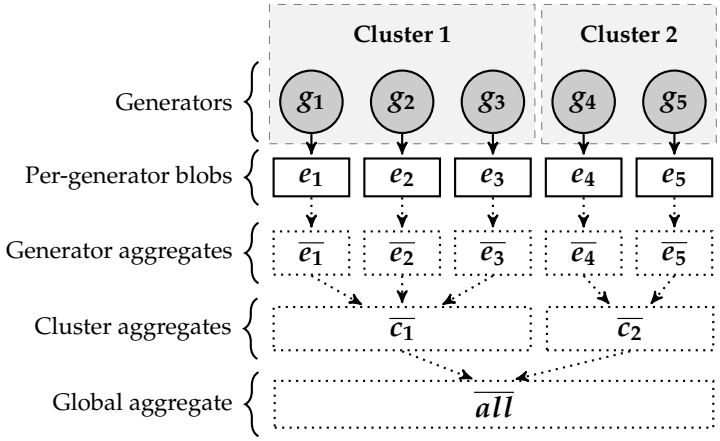


Figure 9.7 – Simplified MonALISA data storage layout, showing five generators on two different clusters, and three levels of aggregation. Solid arrows indicate events written, dotted arrows represent event aggregation. Each rectangle indicates a different blob. Dotted rectangles denotes aggregate blobs.

the monitoring system collecting and aggregating in real-time telemetry events from more than 80 datacenters around the world: MonALISA [4], that we introduced in section 6.3

We use Týr to process and aggregate real data from MonALISA, and compare its performance with other state-of-the-art systems on the Microsoft Azure cloud platform. We prove that Týr throughput outperforms its competitors by up to 75 % when faced with transactional operations while providing significantly higher consistency guarantees. We scale the storage system to up to 256 nodes to demonstrate the scalability of such transactional operations.

Managing monitoring data: what could be improved. The current implementation of ALICE is based on a PostgreSQL database [142]. Aggregation is performed by a background worker task at regular intervals. With the constant increase in volume of the collected metrics, this storage architecture becomes inefficient. Time-series databases such as OpenTSDB [168] or KairosDB [144] were considered to replace the current architecture. However, storing each event individually, along with the related metadata such as tags, leads to a significant overhead. In the context of MonALISA, the queries are known at the time measurements and stored by the system. This opens the way to a highly-customized storage backend using a data layout that would at the same time dramatically increase throughput, reduce metadata overhead, and ultimately lower both the computing and storage requirements for the cluster. We use Týr to develop such a backend.

The need for transactions. The blob-based storage layout for the MonALISA system is as follows. All measurements $(timestamp, measurement)$ are appended to a per-generator blob. Measurements are then averaged over a one-minute window with different granularity levels (machine, cluster, site, region, and job). This layout is explained in Figure 9.7. Updating an aggregate is a three-step operation: read old value, update it with the new data, and write the new value (*read-update-write*). In order to guarantee the correctness of such operations, all writes must be atomic. This atomicity also enables hot snapshotting of the data. As an optimization for aggregate computation, it is desirable for the read-update-write operations to be performed in-place, using Týr's atomic transform operations.

Experimental setup. We deploy Týr on the Microsoft Azure Compute [76] platform on up to 256 nodes. For all experiments, we used `D2 v2` general-purpose instances, located in the `East US` region (Virginia). Each virtual machine has access to 2 CPU cores, 7 GB RAM and 60 GB SSD storage. The host server is based on 2.4 GHz Intel Xeon E5-2673 v3 processors and is equipped with 10 Gigabit Ethernet connectivity. We compare Týr with Rados [188] and BlobSeer [146]. We also plot the results obtained with Microsoft Azure Storage Blobs [68], a managed blob storage system available on the Microsoft Azure platform. It comes in three flavors: *append blobs, block blobs and page blobs*. Append blobs are optimized for append operations, block blobs are optimized for large uploads, and page blobs are optimized for random reads and writes. We use a dump of real data obtained from MonALISA. This data set is composed of $\sim$ 4.5 million individual measurement events, each one being associated to a specific monitored site.

Ensuring write consistency on non-transactional systems. Because of the lack of native transaction support in Týr competitors, we use ZooKeeper 3.4.8 [108] (ZK), an industry-standard, high-performance distributed synchronization service, which is part of the Hadoop [167] stack. Zookeeper allows us to synchronize writes to the data stores with a set of distributed locks. ZooKeeper locks are handled at the lowest-possible granularity: one lock is used for each aggregate offset (8-byte granularity), except for Azure in which we have to use coarse-grained locks (512-byte granularity). Throughout our experiments, we measure the relative impact of the choice of ZooKeeper as a distributed lock provider. Overall, the results show that ZooKeeper accounts for less than 5 % of the total request latency for write storage operations. Although faster, more optimized distributed locks such as Redis [71] may be available, this would not significantly impact the results.

9.4.1 Transactional read/write performance

We first provide a baseline performance under a transactional workload. We measure the transactional write performance of Týr, Rados, BlobSeer and Microsoft Azure Blobs with the MonALISA workload. We perform data aggregation in Týr using atomic transforms as well as with transactional read-update-write operations. This is to provide a fair baseline for comparison with other systems. Týr transactions are used to synchronize the storage of the events and their indexing in the context of a concurrent setup. All systems are deployed on a 32-node cluster, except for Azure Storage which does not offer the possibility to tune the number of machines in the cluster.

Write performance. We first evaluate writes comparing with lock-based transactions. The results, depicted in Figure 9.8, show that the Týr peak throughput outperforms its competitors by 78 % while supporting higher concurrency levels. Atomic updates allowed Týr to further increase performance by saving the cost of read operations for simple updates. The significant drop of performance in the case of Rados, Azure Storage and BlobSeer at higher concurrency levels is due to the increasing lock contention. This issue appears most frequently on the global aggregate blob, which is written to for each event indexed. In contrast, our measurements show that Týr's performance drop is due to CPU exhaustion, mainly because of the additional resources required for handling incoming network requests. Under lower concurrency, however, we can see that the transaction protocol incurs a slight processing overhead, resulting in a comparable performance for Týr and Rados when the update

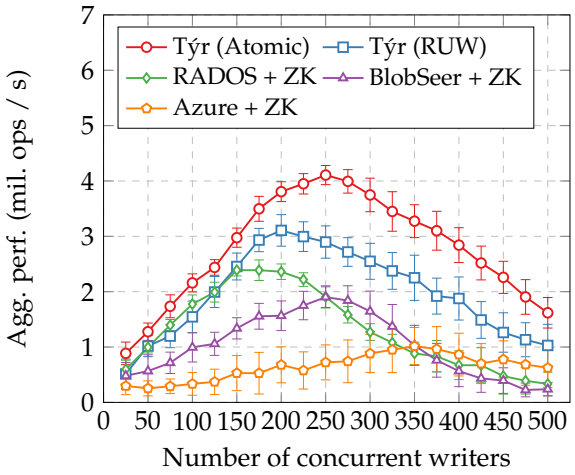


Figure 9.8 – Synchronized write performance of Týr, Rados, BlobSeer and Azure Storage, varying the number of clients, with 95 % confidence intervals.

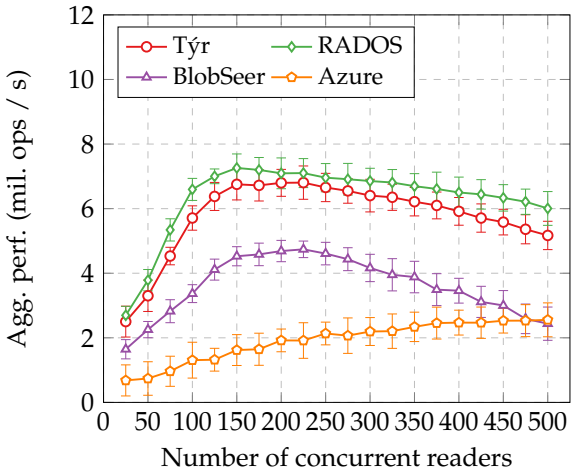


Figure 9.9 – Read performance of Týr, Rados, BlobSeer and Azure Storage, varying the number of clients, with 95 % confidence intervals.

concurrency is low. BlobSeer is penalized by its tree-based metadata management which incurs a non-negligible overhead compared to Týr and Rados. Overall, Azure shows a lower performance and higher variability than all systems. At higher concurrency levels however, Azure performs better than both Rados and BlobSeer. This could be explained by a higher number of nodes in the Azure cluster, although the lack of visibility into its internals doesn't allow us to draw any conclusive explanation. We note the added value of atomic transform operations, which enable Týr to increase its performance by 33 % by cutting the cost of read operations for simple aggregate updates and reducing the transaction failure rate.

Read performance. We evaluate the read performance of a 32-node Týr cluster and compare it with the results obtained with Rados and BlobSeer on a similar setup. As a baseline, we measure the performance of the same workload on the Azure Storage platform. We preload in each of these systems the whole MonALISA dataset, for a total of $\sim$ 100 Gigabytes of uncompressed data. We then performed random reads of 800 byte each from both the raw data and the aggregates, following a power-law distribution to increase read concurrency. This read size corresponds to a 100-minute average of aggregated data. We throttle the number of concurrent requests in the system to a maximum of 1,000.

We plot the results in Figure 9.9. The lightweight read protocol of both Týr and Rados allows them to process reads at near-wire speed and to outperform both BlobSeer and Azure Storage peak throughput by 44 %. On the other hand, BlobSeer requires multiple hops to fetch the data in the distributed metadata tree. This incurs an additional networking cost that limits the total performance of the cluster. Under higher concurrency, we observe a slow drop in throughput for all the compared systems except for Azure Storage due to the involved CPU in the cluster getting overloaded. Once again, linear scalability properties of Azure could be explained by the higher number of nodes in the cluster, although this can't be verified because of the lack of visibility into Azure internals. Týr and Rados show a similar

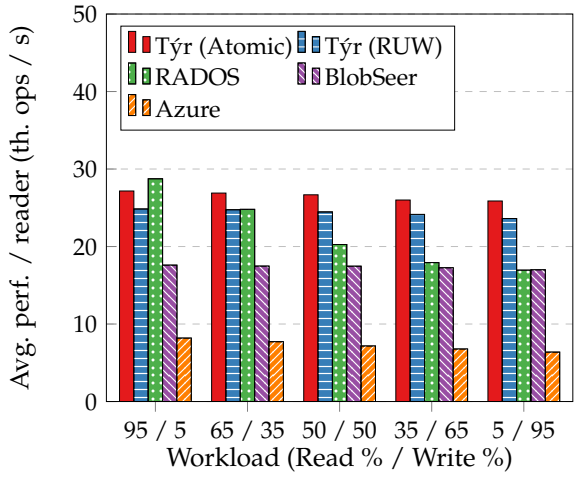


Figure 9.10 – Read throughput of Týr, Rados, BlobSeer and Azure Storage for workloads with varying read to write ratio. Each bar represents the average read throughput of 200 concurrent clients averaged over one minute.

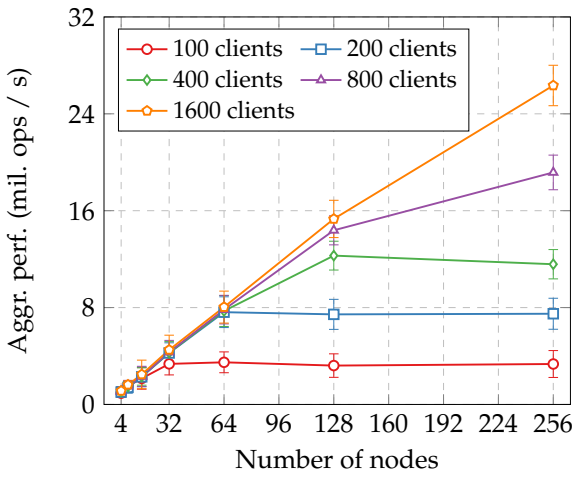


Figure 9.11 – Týr horizontal scalability. Each point shows the average throughput of the cluster over a one-minute window with a 65 % read / 35 % write workload, and 95 % confidence intervals.

performance pattern. Measurements show Rados outperforming Týr by a margin of approximately 7 %. This performance penalty can partly be explained by the slight overhead of the multi-version concurrency control in Týr, enabling it to support transactional operations.

Reader / writer isolation. A key, known advantage of multi-version concurrency control is the isolation it provides between readers and writers. We demonstrate this in Týr by simultaneously performing reads and writes in a 32-node cluster, using the same setup and methodology as with the two previous experiments. To that end, we preload half of the MonALISA dataset in the cluster and measure read performance while concurrently writing the remaining half of the data. We run the experiments using 200 concurrent clients. With this configuration, all three systems proved to perform above 85 % of their peak performance for both reads and writes, thus giving comparable results and a fair comparison between the systems. Among these clients, we vary the ratio of readers to writers in order to measure the performance impact of different usage scenarios. For each of these experiments, we monitor the average throughput per reader.

The results, depicted in Figure 9.10, clearly illustrate the added value of multi-version concurrency control on which both Týr and BlobSeer are based. For these two systems, we observe a near-stable average read performance per client despite the varying number of concurrent writers. In contrast, Rados, which outperforms Týr for a 95/5 read-to-write ratio, shows a clear performance drop as this ratio decreases. Azure performance decreases in a similar fashion as the number of writers increases.

9.4.2 Horizontal scalability

We test the performance of Týr when increasing the cluster size up to 256 nodes. This results in an increased throughput as the load is distributed over a larger number of nodes. We use the same setup as for the previous experiment, varying the number of nodes and the number of clients, and plotting the achieved aggregated throughput among all clients over a one-minute time window. We use the same 35 % write / 65 % read workload (with atomic transforms) as in the previous experiment. Figure 9.11 shows the impact of the number of nodes in the cluster on system performance. We see that the maximum average throughput of the system scales near-linearly as new servers are added to the cluster. With this setup, a 256-node Týr cluster peaks at 26.4 million operations per second.

Using Týr as a storage backend for MonALISA leverages write coordination, enabling us to prove the performance and scalability of both transactional operations and atomic transforms. In particular, we highlight the performance of Týr's lightweight transaction protocol which provides very high velocity compared to lock-based solutions while incurring only a small overhead compared to non-transactional systems. We finally note the relevance of the multi-version concurrency control, which enables the readers not to be impacted by a concurrent, conflicting write workload.

Part IV

Perspectives

Chapter 10

Stream Storage for HPC and Big Data Convergence

Contents

BIG AND FAST DATA APPLICATIONS ARE EXPECTED TO MOVE TOWARDS more compute intensive algorithms to get deeper insights for descriptive, predictive and prescriptive analytics [58, 152]. This fuels a recent trend towards the convergence of HPC and Big Data, which is currently greatly influencing the two worlds. Both communities have diverged significantly over the past in terms of proposed solutions and research orientation. As a result, *HPC and BDA stacks remain mostly separated today.* Interestingly however, their challenges at the data management layer are similar: trading versatility for performance.

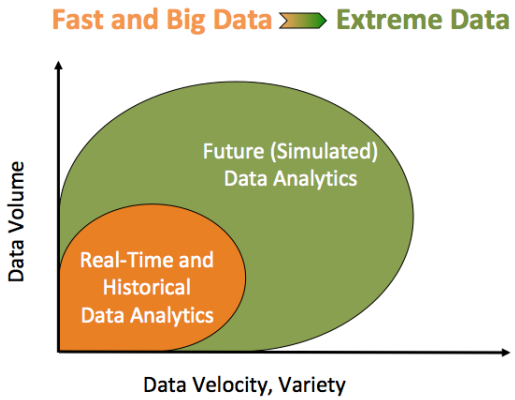


Figure 10.1 – From Fast and Big Data to Extreme Data by integrating the simulation data (about the predicted behavior of the targeted systems) into the analytics.

As such, we advocate that storage offers a high potential for convergence despite current divergences.

HPC and BDA convergence fuels new data processing models. As a result of this expected convergence, new data processing paradigms emerge, leveraging the best practices of the two worlds. For instance, *hybrid analytics* [106] are expected to combine HPC-inspired data processing techniques (i.e., in-situ, in-transit) with more traditional Big and Fast Data processing techniques (i.e., stream-based, batch-based). The goal is to enable joint analysis of past (historical), present (real-time) and future (hypothetical, simulated) data, allowing to predict with better accuracy how the targeted system or phenomenon would behave in critical situations.

Problem: *Hybrid analytics* call for adequate storage backends. To fully design and implement such processing paradigms, it is necessary to fill several technical gaps beyond the current status of the state-of-the-art. In particular, combining HPC and BDA techniques leads to high challenges related to the extreme scale of data management in terms of velocity, variety and volume, as shown in Figure 10.1. First, the number of hypothetical scenarios and the possibility to simulate them with a virtually unlimited combination of parameters *increases the data variety and velocity*. Moreover, the possibility to run joint analysis correlating such hypothetical data with past and real-time data may produce immense amounts of data to process (hence, the *extreme volume*). This requires a scalable data storage backend, able to cope with such huge volumes under operational constraints.

Solution: Blobs in general (and Týr in particular) address the needs of *hybrid analytics*. In this context, we witness in the last years a trend to apply strikingly similar techniques in support of the divergent HPC and BDA orientations. In particular, object storage has proven to be a well-suited storage model for both. In the former, it serves as support for Lustre [164] or DeltaFS [195] while in the latter it is the base building block for the Ceph [187] file system, or is exposed directly to the user in the form of key-value or document stores [80, 121, 170, 75].

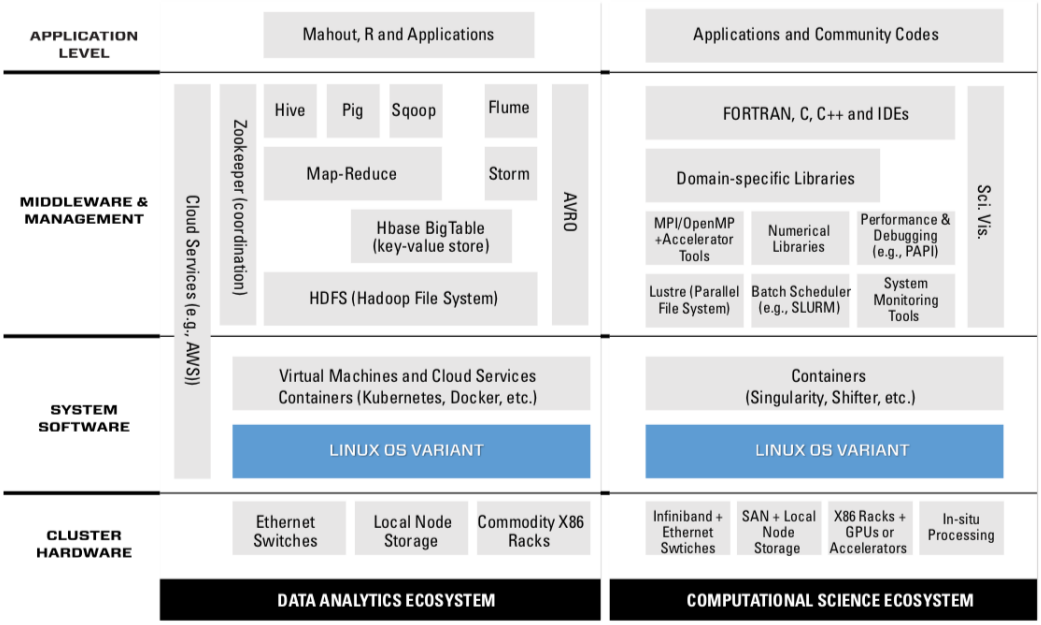


Figure 10.2 – Typical BDA (on the left) and HPC (on the right) software stacks [54].

> ❝ *Despite important divergences, storage-based convergence between HPC and BDA is not only possible, but also leads to substantial performance improvements.* ❞

In this chapter, we hypothesize that despite important divergences, storage-based convergence between HPC and BDA is not only possible, but also leads to substantial performance improvements over the state-of-the-art. Considering the common trends in both HPC and BDA communities, we assert through an empirical proof that at the data management layer such convergence can be achieved through object-based storage, and that Týr is the first such system able to cope with the challenges of the new hybrid analytics.

10.1 HPC and BDA: divergent stacks, convergent storage needs

The key difference between BDA and HPC is the type of data they are dealing with. While the focus of HPC is mainly on structured data, BDA often needs to handle new types of data originating from a wide variety of sources, with very diverse structures, if any. Hence the challenges associated with the *processing* of data is mainly a software challenge.

10.1.1 Comparative overview of the HPC and BDA stacks

Significant differences exist between HPC and BDA platforms. While HPC mostly focuses on large computational loads, BDA targets applications that need to handle very large and complex data sets. These data sets are typically of the order of multiple petabytes or exabytes in size. BDA applications are thus very demanding in terms of storage, to accommodate such a massive amount of data. HPC is usually thought more in terms of sheer computational needs. Figure 10.2 highlights the typical stacks for BDA and HPC applications.

The key differences between those two paradigms result in very different sets of key requirements. A traditional cloud-based BDA platform offers features that are attractive to the general public. These services comprise single, loosely coupled instances (an instance of an OS running in a virtual environment) and storage systems backed by service-level agreements (SLAs) that provide the end user with guaranteed levels of service. These clouds are generally designed to offer the following features.

Instant availability: Cloud offers almost instant availability of resources.
Large capacity: Users can instantly scale the number of applications within the cloud.
Software choice: Users can design instances to suit their needs from the OS up.
Virtualized environments: Instances can be easily moved to and from similar clouds.
Service-level performance: Users are guaranteed a minimal level of performance.

Although these features serve much of the market, HPC users generally have a different set of requirements.

Close to the hardware: HPC libraries and applications are often designed to work closely with the hardware, requiring specific OS drivers and hardware support.
Tuned hardware: HPC hardware is often selected on the basis of communication, memory, and processor speed for a given application set.
Tuned storage: Storage is often designed for a specific application set and user base.
Userspace communication: HPC user applications often need to bypass the OS kernel and communicate directly with remote user processes.
Batch scheduling: All HPC systems use a batch scheduler to share limited resources.

These different sets of requirements have a significant impact on the data-storage layer, which is largely divergent both in terms of architecture and design principles. Many reasons explain storage divergence between HPC and BDA. These include the different software development models, virtualization, scheduling, resource allocation, stateful networks vs. stateful services. Another key difference is the specifics of the data, which has a significant impact in the underlying software and hardware infrastructure, or stack. A stack can be seen as a set of different components, including operating system, execution framework, provisioning, remote console or power management, cluster monitoring, parallel file system and scheduling, development and performance monitoring tools enabling the final users to interact with the cluster, as well as the underlying hardware parts such as CPU, memory and networking.

Let us zoom on the key differences between the HPC and BDA storage stacks.

10.1.2 HPC and BDA storage

HPC: Centralized, file-based sorage. The need to process large volumes of data quickly has huge repercussions for HPC storage, given that storage I/O capabilities are typically much lower than those of processors. An HPC storage system needs large capacity accessible at high speed and to be highly expandable, while offering a single global namespace accessible to all users involved in the project. State-of-the-art storage systems for HPC are mainly the *parallel file systems* deployed on remote storage servers. This architecture is explained

by two main factors. First, many legacy scientific applications are compute-intensive, and hardly interact with the persistent storage except for initial input, occasional checkpointing, and final output. Second, a parallel file system proves to be highly effective for the concurrent I/O workload commonly seen in scientific computing.

While large-scale applications are becoming more data-intensive, moving very large amounts of data in and out of the remote file system is becoming a significant performance bottleneck, greatly hindering time to results. Hence researchers spend significant efforts on improving the I/O throughput of the aforementioned architecture by means of various techniques, such as burst buffers [130].

BDA: Modular, application-purpose storage. On the other hand, BDA architectures bring the computation to the storage nodes, rather than the opposite, as in HPC. This principle has driven the design of the underlying storage stack. Hadoop is the most notable example (presented in section 2.2). It abstracts the storage layer access by proposing an extendable set of data access strategies that essentially separate the computation from the storage. This brings various benefits such as transparent application portability across platforms and independence of the underlying storage system.

File-based storage are arguably the *de facto* standard in the industry. The Hadoop Distributed File System (HDFS) [167], for example, is an integral part of the Hadoop stack. However, most data warehousing solutions today tend to also offer connectors enabling users to analyze vast amounts of data originating from a wide range of storage systems and paradigms. These systems include key-value and columnar stores (Riak [115], Cassandra [121], Aerospike [170]), object stores (Amazon S3 [38], Ceph [187]) or time series databases (OpenTSDB [31]).

10.1.3 Challenges of storage convergence between HPC and BDA

While designing a common storage solution for HPC and BDA could improve portability of application between platforms, the challenges of fitting the requirements of both communities are hard [159]. Indeed, the broad variety of storage systems available for HPC and BDA exhibit different characteristics or deployment models targeted at a specific range of applications. Defining a storage solution able to combine the requirements of both HPC and BDA applications is challenging. For example, it is unclear which storage paradigm is a natural fit to handle data in a broad range of scientific application codes, nor the extent of code modification required to do so. In particular, there is a tension between the colocation of data and computation that is embraced by cloud platforms, and the contradictory dynamic task scheduling and execution that is the core to HPC storage architectures [94]. Also, while file-based storage is common between HPC and BDA stacks, many differences exist in their design principles and implementation. Additionally, BDA embraces a wider variety of storage options such as key-value stores or object stores.

Consequently, in order to achieve storage-based convergence between HPC and BDA, it is critical to evaluate which storage paradigm is better suited to answer the precise requirements of both communities and applications.

10.2 Blobs as a storage model for convergence

State-of-the-art storage systems for both HPC and BDA show a common trend towards relaxing many of the concurrent file access semantics, trading such strong guarantees for increased performance. Nevertheless, some differences remain. Specifically, while the BDA community increasingly relies on systems dropping the POSIX I/O API altogether, the HPC community tends to provide this relaxed set of semantics behind the same API. Although this choice increases backwards compatibility with legacy applications, it also has significant performance implications due to the design constraints imposed by this standard, i.e., hierarchical namespace or permission management.

In this section, we propose and discuss blobs as an alternative to traditional distributed file systems. Inspired by the API similarity with distributed file systems, allowing for random reads and writes in binary objects, we analyze the benefits and limitations of blobs as a storage model for HPC and BDA convergence.

10.2.1 General overview, intuition and methodology

Our goal is *to provide experimental evidence* that blobs can indeed substitute traditional file systems for both HPC and BDA workloads. We demonstrate that the hierarchical nature of distributed file systems is only provided for convenience to the end-user, and is only seldom exploited by the applications, if ever. The consequence would be the majority of storage operations to be file operations (open, close, read, write, create, delete). These operations are very close to the ones proposed by blob storage systems. Directory-level operations (opedir, mkdir, rmdir) do not have their blob counterpart because of the flat nature of the blob namespace. Should applications really need them (e.g., legacy applications), such operations can be emulated using a full database scan. This implementation would intuitively not be optimized compared to its counterpart on current file systems. Yet, since we expect these calls to be vastly outnumbered by blob-level operations, this performance penalty is likely to be compensated by the gains permitted by using a flat namespace and simpler semantics.

Applications. One of the emerging ideas from the discussions on convergence [62] is that the applications cannot be considered separately from the underlying software stack. The fuel for convergence could be a wide variety of HPC and BDA applications leveraging converged services and underlying infrastructure. We therefore base our discussion on a number of I/O-intensive applications extracted from the literature [92, 186, 127] that cover the diversity of I/O workloads commonly encountered on both HPC and BDA platforms. The *HPC applications* we use are based on MPI. They all leverage large input or output datasets associated with large-scale computations atop centralized storage usually provided by a distributed, POSIX-IO-compliant file system such as Lustre [165]. The *BDA applications* are extracted from SparkBench [127, 32], a widely-recognized, industry-standard benchmarking suite for Spark. It includes a representative set of workloads belonging to four application types: machine learning, graph processing, streaming, and SQL queries. In Table 10.1 we make a summary of these applications.

Table 10.1 – HPC and BDA application summary

	Application	Usage	Reads	Writes	Profile
HPC	mpiBLAST (BLAST)	Protein docking	27.7 GB	12.8 MB	Read-intensive
	MOM	Oceanic model	19.5 GB	3.2 GB	Read-intensive
	ECOHAM (EH)	Oceanic model	0.4 GB	9.7 GB	Write-intensive
	Ray Tracing (RT)	Video processing	67.4 GB	71.2 GB	Balanced
BDA	Sort	Text Processing	5.8 GB	5.8 GB	Balanced
	Connected Component (CC)	Graph Processing	13.1 GB	71.2 MB	Read-intensive
	Grep	Text Processing	55.8 GB	863.8 MB	Read-intensive
	Decision Tree (DT)	Machine Learning	59.1 GB	4.7 GB	Read-intensive
	Tokenizer	Text Processing	55.8 GB	235.7 GB	Write-intensive

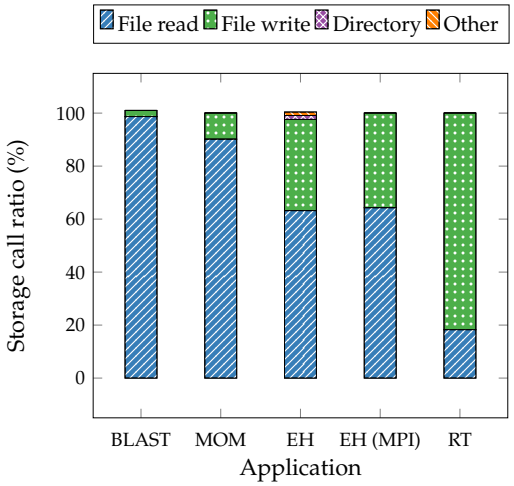


Figure 10.3 – Measured relative amount of different storage calls to the persistent file system for HPC applications.

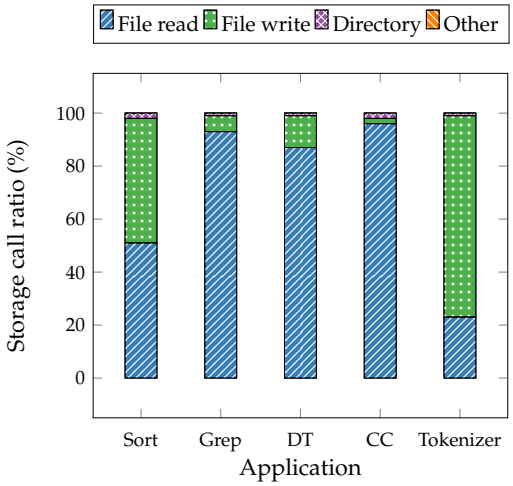


Figure 10.4 – Measured relative amount of different storage calls to the persistent file system for BDA applications.

Experimental platform. We run experiments on the Grid'5000 [57] experimental testbed. Each node embeds 2 x 12-core 1.7 Ghz, 48 GB of RAM, and 250 GB HDD. Network connectivity is supported either with Gigabit Ethernet connectivity (MTU = 1500 B) or by 4 x 20G DDR InfiniBand. We use the former for BDA and the latter for HPC applications in order to fit with the usual configuration of each domain. HPC applications run atop Lustre 2.9.0 and MPICH 3.2 [102], on a 32-node cluster configured with multiple ratios of storage-to-compute nodes. BDA application run atop Spark 2.1.0, Hadoop / HDFS 2.7.3 and Ceph Kraken on the same 32 nodes.

10.2.2 Storage call distribution for HPC and BDA applications

First, we demonstrate that the actual I/O calls made by both HPC and BDA applications are not incompatible with the set of features provided by state-of-the-art blob storage systems. Our intuition is that read and write calls are vastly predominant in the workloads of those applications and that other features of distributed file systems such as directory listings are rarely used, if at all.

Operation	Action	Count
`mkdir`	Create directory	43
`rmdir`	Remove directory	43
`opendir` *(Input directory)*	Open / List directory	5
`opendir` *(Other directories)*	Open / List directory	0

Table 10.2 – Spark directory operation breakdown. For comparison, all applications performed over 10 million total individual storage operations during this benchmark.

Tracing HPC applications. Figure 10.3 summarizes the relative count of storage calls performed by our set of HPC applications. The most important observation for all four applications is the predominance of reads and writes. Except for EH, no application performed any other call to the storage system than reads or writes, confirming our first intuition. This was expected because the MPI-IO standard does not permit any other operation. The few storage calls other than read and write (mainly extended attributes reads and directory listings) are due to the script necessary to prepare the computation and collect the results. These steps can be offlined from the I/O-heavy MPI part of the application, i.e., EH(MPI). This results in only reads and writes being performed in EH(MPI). We conclude that the only operations performed by our set of HPC applications, namely, file I/O, can be mapped to blob I/O on a blob storage system. Consequently, these applications appear to be suited to run unmodified atop blob storage.

Tracing BDA applications. Figure 10.4 shows the relative count of storage calls performed by our set of BDA applications to HDFS. Similar to what we observed with HPC applications, the storage calls are vastly dominated by reads and writes to files. In contrast with HPC, however, all applications also cause Spark to perform a handful of directory operations (91 in total across all our applications out of over 10 million individual storage operations in total). These directory operations are not related to the data processing because input/output files are accessed directly by using read and write calls.

Analyzing these directory operations, we note that they are related solely to (*i*) creating the directories necessary to maintain the logs of the application execution, (*ii*) listing the input files before each application runs if the input data is set as a directory, and (*iii*) maintaining the `.sparkStaging` directory. This directory is internally used by Spark to share information related to the application between nodes and it is filled during the application submission. It contains application files such as the Spark and application archives, as well as distributed cache files [162].

We analyze in detail the directory operations performed by BDA applications. Table 10.2 shows the breakdown of all such directory operations across all applications by storage call. We note that only the input data directories are listed, meaning that Spark accesses directly all the other files it needs with their path. Consequently, a flat namespace such as the one provided by blob storage systems could probably be used.

POSIX Call	Translated Call
`create(/foo/bar)`	`create(/foo__bar)`
`open(/foo/bar)`	`open(/foo__bar)`
`read(fd)`	`read(bd)`
`write(fd)`	`write(bd)`
`mkdir(/foo)`	*Unsupported operation*
`opendir(/foo)`	*Unsupported operation*
`rmdir(/foo)`	*Unsupported operation*

Table 10.3 – HPC storage call translation rules on a flat namespace.

10.2.3 Replacing file-based by blob-based storage

We now demonstrate the potential of blob-based storage to suit the storage needs of both HPC and Big Data applications. To do so, we deploy each aforementioned application atop two state-of-the-art blob storage systems: BlobSeer [146] and Rados [188]. We show that the performance of these applications running atop converged blob-based storage matches or exceeds that of the same applications running atop Lustre [165] for HPC, as well as Ceph [187] and HDFS [167] for Big Data. We assess the performance impact of replacing file-based with blob-based storage by observing three metrics: the *job completion time* is the total execution time of the application, from submission to completion; the *read bandwidth* and *write bandwidth* respectively represent the average data transferred per unit of time for read and write requests.

Replacing Lustre with blob-based storage on HPC

We first show how blob-based storage can be used to transparently support HPC applications while matching or exceeding Lustre I/O performance by replacing the latter with both BlobSeer and Rados. We experiment using three storage-to-compute node configurations in order to ensure that our results are independent of the cluster configuration. We run the same experiments respectively with 28 compute/4 storage nodes, 24/8 and 20/12.

On each node, we deploy a small interceptor to redirect POSIX storage calls to the blob storage system. It is based on FUSE [172], which is supported on most Linux kernels today. This adapter translates file operations to blob operations according to Table 10.3. Directory operations are not supported as we showed previously that they are unnecessary for HPC applications. The APIs of the blob storage systems we consider allow for a direct mapping between file-based and blob-based storage operations.

In Figure 10.5 we plot the average aggregate read and write bandwidth for all applications while varying the compute-to-storage node ratio. We note that for our configuration the 24 compute node/8 storage node setup results in the highest bandwidth for all storage systems. Hence, the following experiments are performed with that configuration. This ratio is much lower than on common HPC platforms (3:1 vs. $\sim$ 70:1 at ORNL, for instance [166]) mainly because the jobs we run are significantly more data-intensive than compute-intensive. We note from these results that blob storage systems constantly outperform Lustre in all configurations for both reads and writes. We will detail these results in the following experiments. For read-intensive applications such as BLAST and MOM, this performance increase allows blob storage systems with 4 storage nodes to achieve a bandwidth

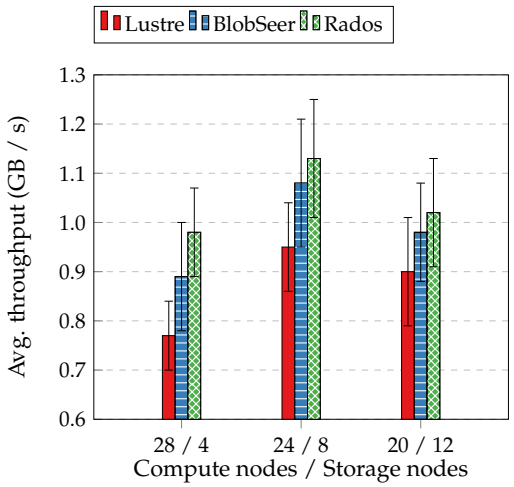


Figure 10.5 – Average agreggate throughput across all HPC applications varying the compute-to-storage ratio, with 95 % confidence intervals.

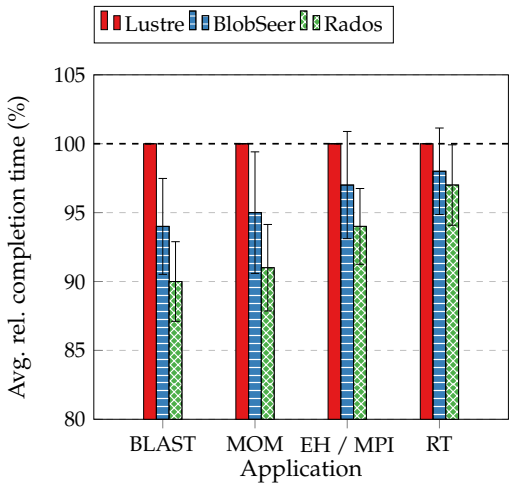


Figure 10.6 – Average performance improvement relative to Lustre for HPC applications using blob-based storage, with 95 % confidence.

comparable to Lustre's with 8 storage nodes.

In Figure 10.6 we plot the average application completion time improvement. The I/O performance gains are here diluted in compute operations. As expected considering the previous results, read-intensive applications exhibit the greatest decrease. BLAST and MOM show a completion time reduction of nearly 8 % with both blob storage systems. In contrast, write-intensive applications such as Ray Tracing show a lower 3 % completion time decrease with BlobSeer or Rados as the underlying storage when compared with Lustre.

Running BDA applications atop blob storage

We now run the same set of experiments for the set of Big Data applications. We demonstrate that BlobSeer and Rados significantly outperform HDFS for all applications. In order to provide an additional baseline of the performance of file systems, we also run these applications atop the Ceph [187] file system, itself based on Rados.

We integrate the storage adapter for blob storage directly inside HDFS. The Hadoop installation has been modified to redirect storage calls to blob storage systems. The translation between POSIX-like calls and flat-namespace blob operations is done using the translation rules defined in Table 10.4. We map all file operations directly to their blob counterpart. Directory operations are simulated using namespace scans. Despite being very costly, the low number of such calls relative to file operations should not significantly impact the application performance when the number of files is low.

In Figure 10.7 we plot the average read bandwidth achieved for SparkBench. While Rados significantly outperforms all systems for all applications, we note an overall performance tie between BlobSeer and Ceph. This is because of the heavy metadata management in all cases, which greatly impacts access latency and consequently read bandwidth. The performance cost of file-based storage is highlighted by the performance difference between

Original operation	Rewritten operation
`create(/foo/bar)`	`create(/foo__bar)`
`open(/foo/bar)`	`open(/foo__bar)`
`read(fd)`	`read(bd)`
`write(fd)`	`write(bd)`
`mkdir(/foo)`	*Dropped operation*
`opendir(/foo)`	`scan()`, *return all like* `/foo__*`
`rmdir(/foo)`	`scan()`, *remove all like* `/foo__*`

Table 10.4 – BDA storage call translation rules on a flat namespace.

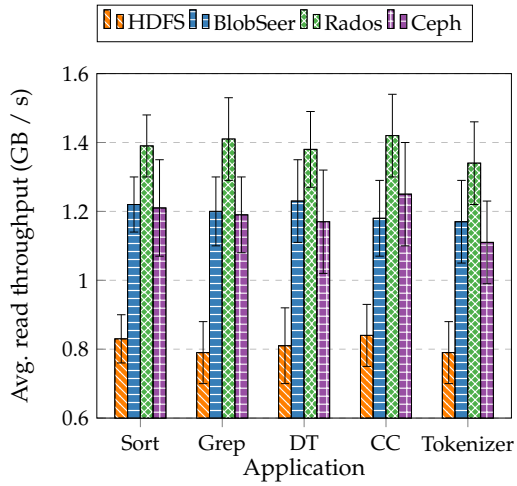


Figure 10.7 – Comparison of read throughput for each Big Data application with HDFS, BlobSeer, Rados and CephFS, with 95% confidence intervals.

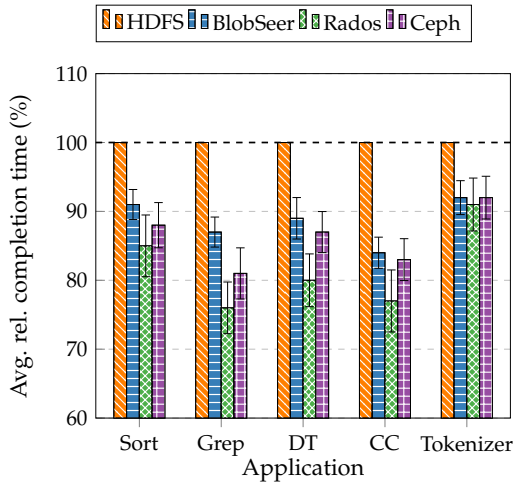


Figure 10.8 – Average performance improvement relative to HDFS for Big Data applications using blob-based storage, with 95% confidence intervals.

Ceph and Rados, upon which it is based. As with HDFS, this is mostly due to the additional communication with metadata servers in the critical path for read requests.

In Figure 10.8 we plot the relative improvement in the total application completion time, diluted in computation. Running Big Data applications atop blobs improves application completion time, up to 22 % compared to HDFS and 7 % compared to Ceph. For Big Data, the highest gains are obtained with read-intensive applications such as Grep and Decision Tree. In comparison, write-intensive applications such as Tokenizer also benefit from improved performance, although relatively smaller due to the globally greater complexity of the write protocols for each storage system.

10.2.4 Which consistency model for converged storage?

As in any storage stack, ensuring data consistency in a convergent HPC and BDA storage system is critical. Integrating consistency management however is not an easy task. HPC and BDA rely on opposite principles with respect to data consistency. The former will typically leverage a storage system offering POSIX I/O strong consistency while leaving up to

the user high-level synchronization tools to be defined directly inside the application when required. The latter usually relies on a wide variety of storage systems offering low-level consistency depending on the application requirements.

We can observe that the storage systems used for HPC tend to offer strong consistency. In contrast, BDA relies on a variety of systems targeted at different use-cases, each offering different consistency models ranging from BASE to full ACID. While both application-specific and transactional concurrency management each have advantages of their own, it is clear that an application that does not require transactions can be executed on a transactional system, while the opposite is not possible or would require significant amount of work. In the end, transactional systems offer a versatile and generic solution to handling consistency in a wide range of application. Indeed, it can at the same time meet the needs of legacy HPC and BDA applications expecting strong consistency and those of BDA applications accepting either strong consistency level or lesser guarantees.

10.2.5 Conclusion: blobs are the right candidate for storage convergence

Results confirm the applicability of blobs for HPC and BDA convergence. First, the set of applications from both HPC and BDA communities almost exclusively perform file-level operations. These operations are very close to those provided by blob storage. Second, replacing file-based with blob-based storage does not hinder the performance of the applications. On the contrary, a performance gain of up to 32 % is observed compared to file systems, which is mostly explained by the optimized read path enabled by the flat namespace. In particular, the *direct read capability* and simple, *decentralized metadata management scheme* that Rados provides excels for read performance, while the *multi-version concurrency control* of BlobSeer supports high write velocity for highly concurrent workloads.

These results demonstrate that blobs are able to support legacy applications at high velocity for both HPC and BDA use-cases. This make them ideal candidates for storage-based convergence between HPC and BDA.

10.3 Týr for HPC and BDA convergence

We now demonstrate the effectiveness of Týr's key design principles in support of legacy HPC and BDA applications. This evaluation leverages the same experimental platform and configuration as in section 10.2, adding Týr as a competitor for BlobSeer, Rados and CephFS. We focus on *non-transactional* performance, and seek to demonstrate that the performance of Týr is competitive with, or even exceeds that of other state-of-the-art storage systems, even when such transactional semantics are not needed. This is the case for the set of legacy applications we introduced in section 10.2.

In particular, using exactly the same experiments as in section 10.2, we seek to understand how Týr behaves compared with its blob-based storage competitors. For HPC applications, we also provide an overview of the horizontal scalability of Týr. We replicate the above experiments on up to 2,048 storage cores and 6,144 application ranks, on a high-end, leadership-class supercomputer hosted at Argonne National Laboratory.

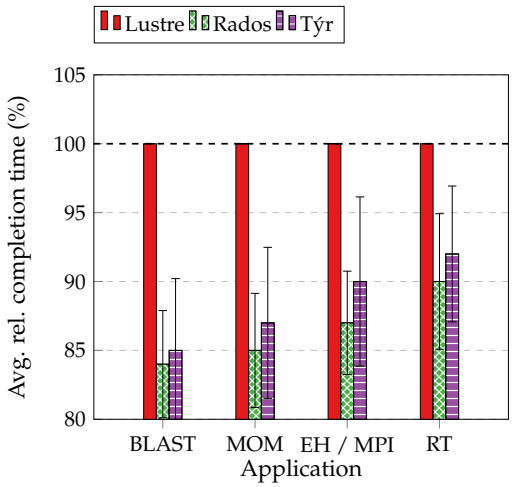


Figure 10.9 – Average performance improvement relative to Lustre for HPC applications using blob storage, with 95 % confidence on Theta.

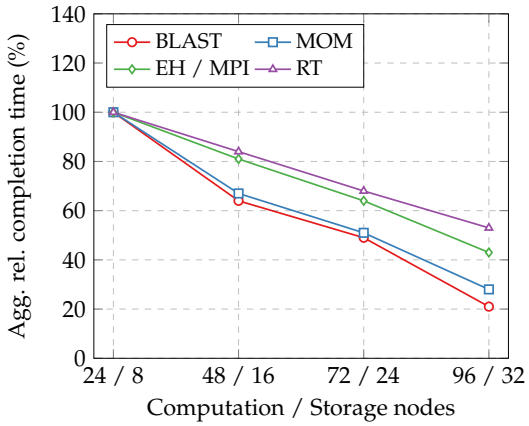


Figure 10.10 – Average performance improvement at scale relative to 32 nodes setup for HPC applications using blob-based storage on Theta.

10.3.1 Týr as a storage backend for HPC applications

In this section we add Týr as a backend storage system for BLAST, MOM, EH(MPI) and Ray Tracing applications, and compare its performance with that of Lustre, BlobSeer and Rados on a high-end supercomputer. To do so, we leverage the Theta supercomputer hosted at Argonne National Laboratory, and run the same experiments as in the previous section. We deploy the applications on 32 nodes (totalling 2,048 cores), using 24 nodes for computation and 8 nodes for storage, and measure the completion time for each application. Experiments with Lustre use the file system available to the computer, totalling 170 storage nodes shared across all users.

We plot the results on Figure 10.9. It is seen that the performance improvement is significantly higher than on our testbed. The reason for this performance increase is to be found in the technical characteristics of Theta, which offers significantly more RAM than SSD space. As such, for the most part, the storage systems deployed on these nodes behave as in-memory storage systems. We acknowledge that the setup of this platform is particular in this respect, and by consequence that the results are not representative of those of another platform with a different setup. Yet, we claim that the results demonstrate that deploying blob-based storage systems on high-end HPC platforms is possible without requiring any application modification. We observe a higher variance in the results compared to Grid'5000, which we attribute to the shared nature of the centralized storage.

In Figure 10.10 we scale all four applications on up to 128 nodes of Theta, or 8,192 cores. We notice a near-linear decrease of computation time as the size of the cluster increases. With applications such as EH(MPI) or Ray Tracing, the performance improvement is slightly lower than with purely read-intensive applications, because of the significantly higher cost of write operations compared to read operations.

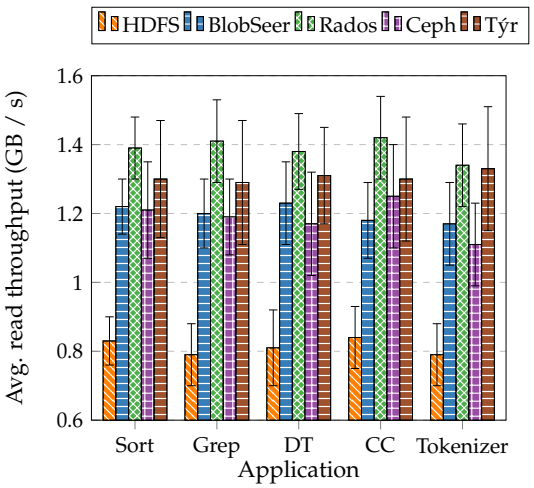


Figure 10.11 – Read throughput for each Big Data application with 95 % confidence intervals.

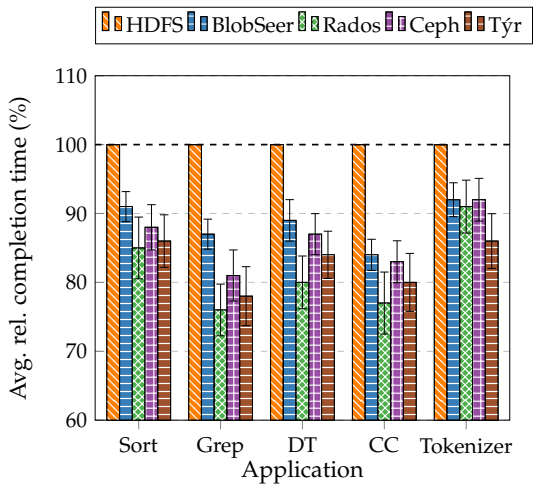


Figure 10.12 – Average performance improvement relative to HDFS using blob storage, with 95 % confidence intervals.

10.3.2 Týr as a storage backend for BDA applications

Figure 10.11 shows the average read bandwidth for these applications. Týr constantly outperforms both Ceph and BlobSeer in all cases, because of the direct read capability it shares with Rados. The experiments also demonstrate that over 98 % of the read operations performed by Spark could be served by the direct read protocol. The remaining 2 % were served by the multi-chunk read protocol. Yet, similarily to what we could observe with HPC, Týr performance is slightly lower than Rados performance due to the cost of the additional consistency guarantees it provides.

In Figure 10.12 we plot the relative improvement in the total application completion time, diluted in computation. As a result of the previous observations with read and write performance, Týr globally enables significantly lower application completion time compared to Ceph and BlobSeer for all applications. Due to its globally higher write performance with highly concurrent workloads, Týr brings substantial application completion time reduction for write-intensive applications such as Tokenizer, while causing a very slight completion time increase when reads are frequent.

10.3.3 Discussion

In this chapter, we have demonstrated Týr's performance using legacy applications, initially designed for file-based storage. Overall, we show that the design of Týr enables it to significantly outperform both BlobSeer and traditional file systems in all cases, for both HPC and BDA applicative use-cases. This is due to its design, leveraging non-blocking writes thanks to multi-version concurrency control and direct writes using consistent hashing techniques.

Týr performance shows a small penalty compared to that of Rados, except for write-intensive applications for which multi-version concurrency control is particularly efficient. This lower performance is the result of the higher consistency guarantees of Týr, enabled by its fully transactional API. Legacy applications such as the ones used in this test are not designed to take advantage of these features.

Chapter 11

A Look Forward: Generalizing HPC and BDA Convergence

Contents

THIS MANUSCRIPT SUMMARIZES A SIGNIFICANT PART OF MY RESEARCH activities over the last years. Clearly, during this period, the domains of Big Data Analytics (BDA) and stream processing have evolved. The pace of change in the data analytics ecosystem is extraordinary, already rendering obsolete some of the elements in Figure 10.2.

Arguably, as noted by the BDEC [1] community [54], the main shift that is currently taking place is to adapt or replace the legacy BDA paradigm (i.e., the software stack in Figure 10.2, typically executed on clouds) with a new type of *distributed services platform*. This would *combine computing, communication, and buffer/storage resources in a data processing network that is far more integrated* than anything available. At the same time, this new platform should be *capable to execute on edge, fog and cloud environments*. An example of a recent system enabling this metaphor at the processing level is Apache NiFi [46]. It automates the flow of data between software systems, allowing for instance to control in a unified fashion a workflow combing edge analytics with Apache Edgent [41] and cloud analytics with Apache Flink [70].

The direction of this recent change is in line with the HPC and BDA convergence. More precisely, this emerging *distributed services platform* stays at the core of future processing combining both models. In this context, the choices we made during the development of this

[1] Big Data and Extreme-scale Computing, https://www.exascale.org/bdec/ .

work favored some research lines over others. For instance, KerA was designed from the beginning with the goal of unifying storage and ingestion, as advocated now by the BDEC community; Týr already enables convergence at the storage level. At the same time, there are certainly many directions that were left unexplored. In this chapter, we discuss the potential research areas that address new challenges related to HPC and BDA convergence.

11.1 Going beyond storage convergence

The challenges of achieving convergence between HPC and BDA platforms are staggering. While the literature tackling those hard problems for data processing is rich, very few works have studied this convergence from the perspective of storage. In the last part of this manuscript we focused on that layer and experimentally demonstrated the usefulness of Týr in a variety of applicative contexts.The future work associated with HPC and BDA convergence spans far beyond storage.

Emerging convergence in large-scale scientific applications. Many science domains are already combining HPC and BDA methods in large-scale *workflows*. They orchestrate simulations or incorporate them into the stages of large-scale analysis pipelines for data generated by simulations, experiments, or observations. Examples of such domains include: astronomy and cosmology, aerospace, autonomous vehicles, weather and climate prediction, smart cities and biomedicine [54]. Hence, application-level convergence is already achieved by means of wide-area, multistage workflows. Their DAG-based model is general enough to describe complex (and changing) configurations of software, hardware, and data flows from both HPC and BDA. Application-level convergence was recently formalized by the team of Geoffrey Fox [93]. They noted that the comparison of simulation and Big Data problems can be made more precise by distinguishing data and models for each use-case. To this end, they introduced a common set of properties used to characterize and compare applications from HPC and BDA.

Processing-level convergence. There are two conflicting views on how to achieve convergence at this level. On the one hand, Big Data processing systems could be adapted to the specifics of HPC platforms. For instance, stream processing in cloud computing was not designed with HPC in mind, and there is a need to examine the high-performance aspects of the runtimes used in this environment. On the other hand, one could imagine HPC-like programming tools and techniques made to fit BDA. There seem to be several non-exclusive alternatives for interfacing HPC with BDA: data streaming, in-transit processing, processing at the edge of the distributed system (i.e., as close as possible to the data sources), and logically centered, cloud-like processing [62].

We advocate that a convergent storage platform such as the one *Týr provides is a solid base for supporting both application-level and processing-level convergence*. Týr enables portability and reduces the scope of the challenges of data processing by proposing a single storage model for a wide variety of upcoming applications and use-cases designed from the ground up with convergence in mind.

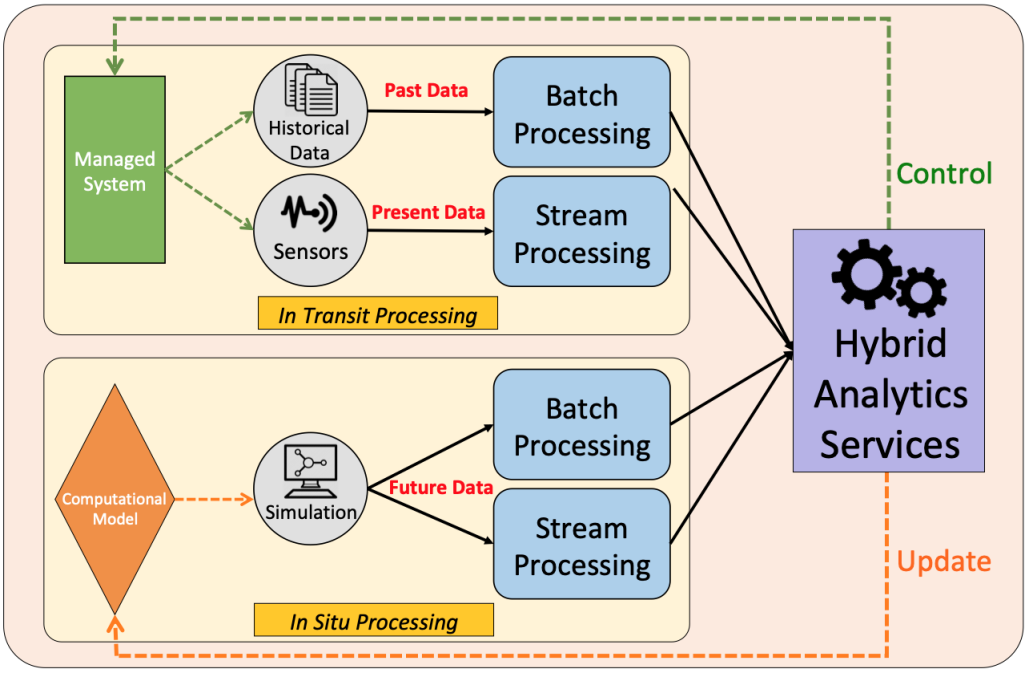


Figure 11.1 – An architecture enabling hybrid analytics.

11.2 Converging on an architecture for hybrid analytics

Hybrid analytics (see chapter 10) combine past (historical) and present (real-time) data jointly with future (hypothetical, simulated) data. The past and present data account for the *data-driven model* of the targeted system or phenomenon, while the future data is the result of the *simulation-based model*. Hybrid analytics are challenging since the two perspectives (data-driven and simulation-based) model the real-world system from two different viewpoints, ignorant of each other. For the simulation model, it is typically a physics perspective, while for the data-driven model it refers to a purely behavioral viewpoint. For example, in the case of autonomous cars (see section 1.1.1), the data-driven model is the ensemble of past and real-time data collected from all the sensors, while the simulation model is the physical model of the vehicle used by a simulation to anticipate its behavior. In the industry, these two models are often referred as "digital twins" [154].

Hybrid analytics correlate the two models in order to provide a deeper interpretation of measured data, enabling more reliable, transparent and innovative decision making. However, leveraging simulations as generators of future data pushes the challenges related to data processing to extreme scales in terms of both volume and velocity. Defining the underlying architectures for such hybrid analytics is thus a complex problem.

An HPC-inspired extension of the Lambda architecture. We propose to combine batch-based and stream-based Big Data processing techniques (i.e., the Lambda architecture introduced in chapter 4) with in-situ/in-transit data processing techniques inspired by the HPC. We illustrate this new, unified architecture for hybrid analytics in Figure 11.1. It enables collecting, managing and processing of extreme volumes of past, real-time and simulated data.

With this architecture, hybrid analytics would enable two things: *(i)* to *update the simulation model* dynamically using the past and real-time data through a continuous learning loop; *(ii)* to *proactively control in real-time* the targeted systems. This reduces uncertainty in prediction and thereby improves decision making.

Extending DataSteward and KerA to enable hybrid analytics. Towards this goal we plan to leverage:

- The scalability of the **DataSteward** framework (already demonstrated on hundreds of cores, see section 5.3.1) for extreme-scale in-situ/in-transit data processing.
- The **KerA** approach to low-latency, high-throughput stream ingestion (see section 8.3).

Since KerA currently handles only the ingestion phase of stream processing, we plan to extend it to make it able to support all the data management needs (i.e., storage and transfer). As such, we will add support for low-latency stream storage (in-memory) and persistence (on disk). For batch and stream processing, we will rely on an existing framework — Apache Flink, which we will extend with elasticity and fault-tolerance support in the presence of very large state. On the simulation side, we will extend the DataSteward in-situ processing framework with programmatic support for Big Data analytics (i.e., adding plugins for batch-based, stream-based and hybrid batch-stream processing).

11.3 Wider transactional semantics

In this manuscript we demonstrated how the semantics and the set of guarantees offered by Týr's transactional architecture could prove beneficial for the design of high-level storage abstractions. In this context, we believe that two directions deserve further investigation. The first one is to extend the benefits of storage-level synchronisation from BDA to HPC environments. This is non-trivial since, historically, HPC leverages application-level synchronisation. The second direction aims to build a richer eco-system of storage systems atop our transactional blob model.

Leveraging transactional operations in HPC contexts. Indeed, series of potential use-cases exist for explicit, storage-level coordination on HPC systems. We are convinced that transactions have a case in HPC applications to collaboratively maintain complex data structures. However, the lack of such semantics on current platforms makes it very hard to formally demonstrate the effectiveness of transactions when leveraged directly at the storage level. Thus, a promising research direction is to study the impact of transactional operations and optimistic write coordination in the design of large-scale applications for next-generation HPC platforms. Such progress would obviate application-level write coordination in most cases, while at the same time greatly simplifying the design of various I/O framework operations, among which MPI Collective I/O.

Exploring a wider variety of transactional storage abstractions over transactional blobs. Transactional blobs have the capacity to serve as low-level distributed storage layer for a variety of storage abstractions, besides distributed file systems [11]. These include transactional key-value stores, transactional wide-column databases or graph databases. A great

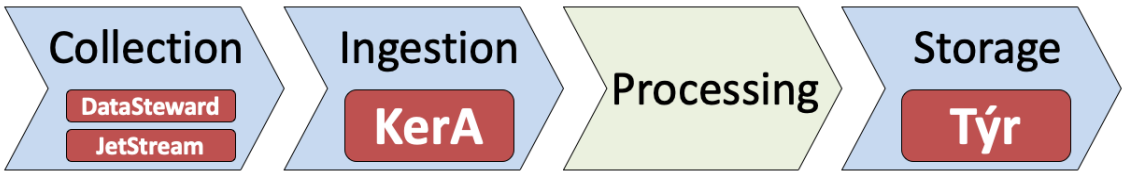


Figure 11.2 – Summary of software contributions.

deal of work certainly exists in the domain, but we believe the persrpective of convergence is yet to be considered. Indeed, such abstractions could be deployed in a wide variety of applicative contexts, from HPC and BDA communities. One could imagine proposing Týr as a base storage layer on both cloud and HPC platforms, letting the users choose the exact interface and semantics they need without investing time and effort on low-level data management or portability across platforms.

11.4 Concluding remarks

I have presented in this manuscript the state-of-the-art of my research domains, introduced several original results and discussed different points of view with respect to them. The contributions address data-management challenges at *all stages of the stream processing pipeline*, as illustrated in Figure 11.2: data collection and in-transit processing at the edge (*DataSteward*), transfer towards the processing sites (*JetStream*), ingestion (*KerA*) and storage (*Týr*).

Chapters 2, 3 and 4 introduce the challenges of the Big and Fast Data processing today, paving the pathway towards the contributions presented in this manuscript. In chapter 5 we present DataSteward, an approach to perform analysis of stream data in-transit on dedicated computed nodes, in order to build early knowledge on the results. Stream data is sent fast and reliably towards the processing sites with JetStream, presented in chapter 6. In chapter 7 we address the issue of metadata centralization, which has a significant impact for streams processed across geographically distributed sites. Chapter 8 is dedicated to KerA, our approach for efficient and scalable stream ingestion. Chapter 9 introduces Týr, a transactional blob storage system able to gracefully support streams with high velocity and horizontal scalability. In chapter 10, by carefully studying the state of the practice in the field of large-scale storage for HPC and BDA platforms, we propose blobs as a relevant storage model for convergence and show how Týr successfully proves relevant in this context with several applicative scenarios.

Acknowledgements

This work was carried out in the context of several collaborative projects, funded by the **Agence Nationale pour la Recherche** (i.e., *ANR MapReduce* [175], *ANR OverFlow* [176]), the **European Commission** (i.e., *H2020 BigStorage* [177], *EIT Digital Big Data Analytics* [179]) or by the joint **Inria - Microsoft Research Centre** (i.e., *ABrain* [174], *ZCloudFlow* [180]) as well as in the context of industrial collaborations like the *HIRP project* [178], funded by **Huawei Research**.

Personal Publications

[1] **Alexandru Costan**, Radu Tudoran, Gabriel Antoniu, and Goetz Brasche. "TomusBlobs: Scalable Data-intensive Processing on Azure Clouds". In: *Concurrency and Computation: Practice and Experience* 28.4 (2016), pp. 950–976.

[2] Gabriel Antoniu, Julien Bigot, Luc Bougé, François Briant, Franck Cappello, **Alexandru Costan**, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, and Christian Pérez et al. "Scalable Data Management for Map-Reduce-based Data-Intensive Applications: a View for Cloud and Hybrid Infrastructures". In: *International Journal of Cloud Computing* 2.2 (2013), pp. 150–170.

[3] Stefan Ene, Bogdan Nicolae, **Alexandru Costan**, and Gabriel Antoniu. "To Overlap or Not to Overlap: Optimizing Incremental MapReduce Computations for On-Demand Data Upload". In: *5th International Workshop on Data-Intensive Computing in the Clouds (in conjunction with IEEE/ACM SC 2014)*. New Orleans, LA, USA, 2014, pp. 9–16.

[4] Iosif Legrand, Ramiro Voicu, Catalin Cirstoiu, Costin Grigoras, Latchezar Betev, and **Alexandru Costan**. "Monitoring and Control of Large Systems with MonALISA". In: *Communications of the ACM* 52.9 (2009), pp. 49–55.

[5] Ovidiu-Cristian Marcu, **Alexandru Costan**, Gabriel Antoniu, and Maria S. Pérez. "Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks". In: *IEEE International Conference on Cluster Computing (CLUSTER 2016)*. Taipei, Taiwan, 2016, pp. 433–442.

[6] Ovidiu-Cristian Marcu, **Alexandru Costan**, Gabriel Antoniu, Maria S. Pérez, Bogdan Nicolae, Radu Tudoran, and Stefano Bortoli. "KerA: Scalable Data Ingestion for Stream Processing". In: *38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018)*. Vienna, Austria, 2018, pp. 1480–1485.

[7] Ovidiu-Cristian Marcu, **Alexandru Costan**, Gabriel Antoniu, Maria S. Pérez, Radu Tudoran, Stefano Bortoli, and Bogdan Nicolae. "Towards a Unified Storage and Ingestion Architecture for Stream Processing". In: *IEEE International Conference on Big Data (BIGDATA 2017)*. Boston, MA, USA, 2017, pp. 2402–2407.

[8] Ovidiu-Cristian Marcu, Radu Tudoran, Bogdan Nicolae, **Alexandru Costan**, Gabriel Antoniu, and Maria S. Pérez. "Exploring Shared State in Key-Value Store for Window-Based Multi-pattern Streaming Analytics". In: *1st Workshop on the Integration of Extreme Scale Computing and Big Data Management and Analytics (in conjunction with IEEE/ACM CCGRID 2017)*. Madrid, Spain, 2017, pp. 1044–1052.

[9] Pierre Matri, **Alexandru Costan**, Gabriel Antoniu, Jésus Montes, and Maria S. Pérez. "Towards Efficient Location and Placement of Dynamic Replicas for Geo-Distributed Data Stores". In: *7th Workshop on Scientific Cloud Computing - ScienceCloud (in conjunction with ACM HPDC 2016)*. Kyoto, Japan, 2016, pp. 3–9.

[10] Pierre Matri, **Alexandru Costan**, Gabriel Antoniu, Jésus Montes, and Maria S. Pérez. "Týr: Blob Storage Meets Built-In Transactions". In: *IEEE/ACM: International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016)*. Salt Lake City, UT, USA, 2016, pp. 573–584.

[11] Pierre Matri, Maria S. Pérez, **Alexandru Costan**, and Gabriel Antoniu. "TýrFS : Increasing Small Files Access Performance with Dynamic Metadata Replication". In: *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2018)*. Washington, DC, USA, 2018, pp. 452–461.

[12] Pierre Matri, Maria S. Pérez, **Alexandru Costan**, Luc Bougé, and Gabriel Antoniu. "Keeping Up With Storage: Decentralized, Write-enabled Dynamic Geo-Replication". In: *Future Generation Computer Systems* 86 (2018), pp. 1093–1105.

[13] Benoît Da Mota, Radu Tudoran, **Alexandru Costan**, Goetz Brasche, Gabriel Antoniu, and Betrand Thirion. "Machine Learning Patterns for Neuroimaging-Genetic Studies in the Cloud". In: *Frontiers in Neuroinformatics* 8.31 (2014), pp. 1–28.

[14] Luis Pineda-Morales, **Alexandru Costan**, and Gabriel Antoniu. "Towards Multi-site Metadata Management for Geographically Distributed Cloud Workflows". In: *IEEE International Conference on Cluster Computing (CLUSTER 2015)*. Chicago, IL, USA, 2015, pp. 294–303.

[15] Luis Pineda-Morales, Ji Liu, **Alexandru Costan**, Esther Pacitti, Gabriel Antoniu, Patrick Valduriez, and Marta Mattoso. "Managing Hot Metadata for Scientific Workflows on Multisite Clouds". In: *IEEE International Conference on Big Data (BIGDATA 2016)*. Washington, DC, USA, 2016, pp. 390–397.

[16] Roxana-Ioana Roman, Bogdan Nicolae, **Alexandru Costan**, and Gabriel Antoniu. "Understanding Spark Performance in Hybrid and Multi-Site Clouds". In: *6th International Workshop on Big Data Analytics: Challenges and Opportunities (in conjunction with IEEE/ACM SC15)*. Austin, TX, USA, 2015, pp. 10–16.

[17] Radu Tudoran, **Alexandru Costan**, and Gabriel Antoniu. "Big Data Storage and Processing on Azure Clouds: Experiments at Scale and Lessons Learned". In: *Cloud Computing for Data-Intensive Applications*. Springer, 2015, pp. 331–355.

[18] Radu Tudoran, **Alexandru Costan**, and Gabriel Antoniu. "DataSteward: Using Dedicated Compute Nodes for Scalable Data Management on Public Clouds". In: *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM 2013)*. Melbourne, Australia, 2013, pp. 1057–1064.

[19] Radu Tudoran, **Alexandru Costan**, and Gabriel Antoniu. "MapIterativeReduce: A Framework for Reduction-Intensive Data Processing on Azure Clouds". In: *3rd International Workshop on MapReduce and its Applications (in conjunction with ACM HPDC 2012)*. Delft, Netherlands, June 2012, pp. 9–16.

[20] Radu Tudoran, **Alexandru Costan**, and Gabriel Antoniu. "OverFlow: Multi-Site Aware Big Data Management for Scientific Workflows on Clouds". In: *IEEE Transactions on Cloud Computing* 4.1 (2016), pp. 76–89.

[21] Radu Tudoran, **Alexandru Costan**, and Gabriel Antoniu. "Transfer as a Service: Towards a Cost-Effective Model for Multi-site Cloud Data Management". In: *33rd IEEE Symposium on Reliable Distributed Systems (SRDS 2014)*. Nara, Japan, 2014, pp. 51–56.

[22] Radu Tudoran, **Alexandru Costan**, Gabriel Antoniu, and Luc Bougé. "A Performance Evaluation of Azure and Nimbus Clouds for Scientific Applications". In: *2nd International Workshop on Cloud Computing Platforms - CloudCP (in conjunction with ACM SIGOPS EuroSys 2012)*. Bern, Switzerland, 2012, pp. 10–16.

[23] Radu Tudoran, **Alexandru Costan**, Gabriel Antoniu, and Hakan Soncu. "Tomusblobs: Towards Communication-Efficient Storage for MapReduce Applications in Azure". In: *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012)*. Ottawa, Canada, 2012, pp. 427–434.

[24] Radu Tudoran, **Alexandru Costan**, Benoît Da Mota, Gabriel Antoniu, and Bertrand Thirion. "A-Brain: Using the Cloud to Understand the Impact of Genetic Variability on the Brain". In: *International Workshop on CloudFutures*. Berkley, CA, USA, 2012.

[25] Radu Tudoran, **Alexandru Costan**, Olivier Nano, Ivo Santos, Hakan Soncu, and Gabriel Antoniu. "JetStream: Enabling High Throughput Live Event Streaming on Multi-site Clouds". In: *Future Generation Computer Systems* 54 (2016), pp. 274–291.

[26] Radu Tudoran, **Alexandru Costan**, Ramin Rezai Rad, Goetz Brasche, and Gabriel Antoniu. "Adaptive File Management for Scientific Workflows on the Azure Cloud". In: *IEEE International Conference on Big Data (BIGDATA 2013)*. Silicon Valley, CA, USA, 2013, pp. 273–281.

[27] Radu Tudoran, **Alexandru Costan**, Rui Wang, Luc Bougé, and Gabriel Antoniu. "Bridging Data in the Clouds: An Environment-Aware System for Geographically Distributed Data Transfers." In: *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2014)*. Chicago, IL, USA, 2014, pp. 92–101.

[28] Radu Tudoran, Olivier Nano, Ivo Santos, **Alexandru Costan**, Hakan Soncu, Luc Bougé, and Gabriel Antoniu. "JetStream: Enabling High Performance Event Streaming Across Cloud Data-Centers." In: *8th ACM International Conference on Distributed Event-Based Systems (DEBS 2014)*. Mumbai, India, 2014, pp. 23–34.

Other References

[29] Kenneth Aamodt, Abrahantes Quintana, Achenbach Acounis, and Adamová Adler. "The ALICE experiment at the CERN LHC". In: *Journal of Instrumentation* 3.8 (2008), pp. 8–20.

[30] Rohit Agarwal, Gideon Juve, and Ewa Deelman. "Peer-to-Peer Data Sharing for Scientific Workflows on Amazon EC2". In: *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 82–89.

[31] Bikash Agrawal, Antorweep Chakravorty, Chunming Rong, and Tomasz Wiktor Wlodarczyk. "R2Time: a framework to analyse open TSDB time-series data in HBase". In: *6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2014)*. IEEE. 2014, pp. 970–975.

[32] Dakshi Agrawal, Ali Butt, Kshitij Doshi, Josep-L Larriba-Pey, Min Li, Frederick R Reiss, Francois Raab, Berni Schiefer, Toyotaro Suzumura, and Yinglong Xia. "SparkBench–a spark performance testing suite". In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2015, pp. 26–44.

[33] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1792–1803.

[34] *Akka Streams*. https://akka.io/. (Visited on 12/05/2018).

[35] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and Ponnuswamy Sadayappan. "Scalable I/O forwarding framework for high-performance computing systems". In: *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE. 2009, pp. 1–10.

[36] William Allcock. "GridFTP: Protocol Extensions to FTP for the Grid." In: *Global Grid ForumGFD-RP, 20* (2003).

[37] Beulah Kurian Alunkal. "Grid Eigen Trust a Framework for Computing Reputation in Grids". PhD thesis. Illinois Institute of Technology, 2003.

[38] *Amazon S3*. https://aws.amazon.com/s3/. (Visited on 12/05/2018).

[39] David. P. Anderson and Gilles Fedak. "The Computational and Storage Potential of Volunteer Computing". In: *6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. Vol. 1. May 2006, pp. 73–80.

[40] *Apache Drill*. https://drill.apache.org. (Visited on 12/05/2018).

[41] *Apache Edgent*. http://edgent.apache.org. (Visited on 12/05/2018).

[42] *Apache Hadoop*. http://hadoop.apache.org/. (Visited on 06/26/2017).

[43] *Apache Hive*. http://hive.apache.org/. (Visited on 12/05/2018).

[44] *Apache Kafka*. https://kafka.apache.org/. (Visited on 08/08/2017).

[45] *Apache Kudu*. https://kudu.apache.org. (Visited on 12/05/2018).

[46] *Apache Nifi*. https://nifi.apache.org. (Visited on 12/05/2018).

[47] *Apache Oozie*. http://oozie.apache.org. (Visited on 12/05/2018).

[48] *Apache Pig*. http://pig.apache.org/. (Visited on 12/05/2018).

[49] *Apache Pulsar*. https://pulsar.incubator.apache.org. (Visited on 12/05/2018).

[50] *Apache Spark SQL*. https://spark.apache.org/sql/. (Visited on 12/05/2018).

[51] *Apache Storm*. https://storm.apache.org/. (Visited on 06/27/2017).

[52] *Apache Tez*. http://tez.apache.org. (Visited on 12/05/2018).

[53] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[54] Mark Asch and Terry Moore. *Pathways to Convergence*. Tech. rep. BDEC, 2017.

[55] *Automotive Council UK*. https : / / www . automotivecouncil . co . uk. 2018. (Visited on 01/01/2018).

[56] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. "Coordination avoidance in database systems". In: *Proceedings of the VLDB Endowment* 8.3 (2014), pp. 185–196.

[57] Daniel Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20.

[58] BDVA. *European Big Data Value Strategic Research and Innovation Agenda*. Tech. rep. BDVA, 2017.

[59] Gordon Bell, Tony Hey, and Alex Szalay. "Beyond the Data Deluge". In: *Science* 323.5919 (Mar. 2009), pp. 1297–1298.

[60] Janie Bennett et al. "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–9.

[61] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. "Characterization of scientific workflows". In: *3rd Workshop on Workflows in Support of Large-Scale Science*. 2008, pp. 1–10.

[62] *Big Data and Extreme-scale Computing Workshop*. http://www.exascale.org/bdec/. 2013. (Visited on 07/30/2017).

[63] Dhruba Borthakur. "HDFS architecture guide". In: *Hadoop Apache Project* 53 (2008).

[64] Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. "Flexible and Scalable Storage Management for Data-intensive Stream Processing". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. Saint Petersburg, Russia: ACM, 2009, pp. 934–945.

[65] Krishnaveni Budati, Jason Sonnek, Abhishek Chandra, and Jon Weissman. "Ridge: Combining Reliability and Performance in Open Grid Platforms". In: *Proceedings of the 16th International Symposium on High Performance Distributed Computing*. HPDC '07. Monterey, California, USA: ACM, 2007, pp. 55–64. ISBN: 978-1-59593-673-8.

[66] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation Computer Systems* 25.6 (2009), pp. 599–616.

[67] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 143–157.

[68] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 143–157.

[69] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. "State Management in Apache Flink: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729.

[70] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. "Apache Flink: Stream and Batch Processing in a Single Engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38.4 (Dec. 2015), pp. 1–11.

[71] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[72] Philip Carns, Kevin Harms, Dries Kimpe, Robert Ross, Justin Wozniak, Lee Ward, Matthew Curry, Ruth Klundt, Geoff Danielson, Cengiz Karakoyunlu, et al. "A case for optimistic coordination in hpc storage systems". In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* IEEE. 2012, pp. 48–53.

[73] *Ceph: differences from POSIX*. http://docs.ceph.com/docs/kraken/cephfs/posix/. 2018. (Visited on 12/05/2018).

[74] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 4–30.

[75] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc., 2013.

[76] Marshall Copeland, Julian Soh, Anthony Puca, Mike Manning, and David Gollob. "Microsoft azure and cloud computing". In: *Microsoft Azure*. Springer, 2015, pp. 3–26.

[77] Abhinandan Das, Indranil Gupta, and Ashish Motivala. "Swim: Scalable weakly-consistent infection-style process group membership protocol". In: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE. 2002, pp. 303–312.

[78] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113.

[79] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[80] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[81] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. "The Cost of Doing Science on the Cloud: The Montage Example". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, pp. 501–512.

[82] Ewa Deelman et al. "Pegasus: A Framework for Mapping Complex Scientific Workflows Onto Distributed Systems". In: *Sci. Program.* 13.3 (July 2005), pp. 219–237.

[83] "Developing Cloud Applications using the e-Science Central Platform." In: *Proceedings of Royal Society A*. Vol. 371. 1983. Dec. 2012.

[84] Robert Escriva and Emin Gün Sirer. "The design and implementation of the warp transactional filesystem". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association. 2016.

[85] Robert Escriva, Bernard Wong, and Emin Gün Sirer. "HyperDex: A distributed, searchable key-value store". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM. 2012, pp. 25–36.

[86] Robert Escriva, Bernard Wong, and Emin Gün Sirer. "Warp: Lightweight multi-key transactions for key-value stores". In: *arXiv preprint arXiv:1509.07815* (2015).

[87] Pei Fan, Zhenbang Chen, Ji Wang, Zibin Zheng, and Michael R. Lyu. "Topology-Aware Deployment of Scientific Applications in Cloud Computing". In: *IEEE CLOUD*. 2012.

[88] Pei Fan, Zhenbang Chen, Ji Wang1, Zibin Zheng, and Michael R. Lyu. "Scientific application deployment on Cloud: A Topology-Aware Method". In: *Concurrency and Computattion: Practice and Experience* (2012).

[89] *Flink Fault Tolerance*. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/. 2018. (Visited on 12/05/2018).

[90] Ian Foster, Ann Chervenak, Dan Gunter, Kate Keahey, Ravi Madduri, and Raj Kettimuthu. "Enabling PETASCALE Data Movement and Analysis". In: *Scidac Review* (Winter 2009). (Visited on 12/05/2018).

[91] Ian Foster, Rajkumar Kettimuthu, Stuart Martin, Steve Tuecke, Daniel Milroy, Brock Palen, Thomas Hauser, and Jazcek Braden. "Campus Bridging Made Easy via Globus Services". In: *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*. XSEDE '12. Chicago, Illinois: ACM, 2012, 50:1–50:8.

[92] Geoffrey C. Fox, Jha Shantenu, Qiu Judy, Ekanayake Saliya, and Luckow Andre. "Towards a Comprehensive Set of Big Data Benchmarks". In: *Advances in Parallel Computing* 26 (2015), pp. 47–66.

[93] Geoffrey Fox, Judy Qiu, Shantenu Jha, Saliya Ekanayake, and Supun Kamburugamuve. *Big Data, Simulations and HPC Convergence*. Tech. rep. Indiana University, 2016.

[94] Geoffrey Fox, Judy Qiu, Shantenu Jha, Saliya Ekanayake, and Supun Kamburugamuve. "Big data, simulations and hpc convergence". In: *Big Data Benchmarking*. Springer, 2015, pp. 3–17.

[95] Lars George. *HBase: the definitive guide: random access to your planet-size data*. O'Reilly Media, Inc., 2011.

[96] John Giardino, Jim Haridas, and Ben Calder. *How to get most out of Windows Azure Tables*. 2013.

[97] Dan Gibson. *Is Your Big Data Hot, Warm, or Cold?* 2012. (Visited on 03/11/2017).

[98] Jeremy Ginsberg, Matthew H. Mohebbi, Rajan S. Patel, Lynnette Brammer, Mark S. Smolinski, and Larry Brilliant. "Detecting influenza epidemics using search engine query data". In: *Nature* 457 (2009), pp. 1012–1014.

[99] Leonardo B. Gomez and Frack Cappello. "Improving floating point compression through binary masks". In: *2013 IEEE International Conference on Big Data*. Oct. 2013, pp. 326–331.

[100] *Google Cloud Platform*. https://cloud.google.com/compute/. Google. (Visited on 06/26/2017).

[101] *Grid'5000 Nancy hardware*. https://www.grid5000.fr/mediawiki/index.php/Nancy:Hardware. 2017. (Visited on 12/05/2018).

[102] William Gropp and Ewing Lusk. *Users guide for MPICH, a portable Implementation of MPI*. Tech. rep. Argonne National Lab., IL (United States), 1996.

[103] Sijie Guo, Robin Dhamankar, and Leigh Stewart. "DistributedLog: A High Performance Replicated Log Service". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. Apr. 2017, pp. 1183–1194.

[104] Yahoo. *Powered By Hadoop*. http://wiki.apache.org/hadoop/PoweredBy/. 2017. (Visited on 03/24/2017).

[105] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. "The ϕ accrual failure detector". In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, pp. 66–78.

[106] Nicole Hemsoth. "HPC and Big Data: A "Best of Both Worlds" Approach". In: *HPC Wire* 1 (2014).

[107] Tony Hey, Stewart Tansley, and Kristin M. Tolle, eds. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[108] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. Boston, MA, USA. 2010.

[109] IDC. *IDC's Data Age 2025 study*. http://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf. International Data Corporation, 2017. (Visited on 06/25/2017).

[110] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72.

[111] Hai Jin, Shadi Ibrahim, Tim Bell, Wei Gao, Dachuan Huang, and Song Wu. "Cloud types and services". In: *Handbook of Cloud Computing*. Springer, 2010, pp. 335–355.

[112] William Kabsch. "A solution for the best rotation to relate two sets of vectors". In: *Acta Crystallographica Section A* 32.5 (), pp. 922–923.

[113] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM. 1997, pp. 654–663.

[114] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, Rajkumar Kettimuthu, P. Sadayappan, Ian Foster, and Joel Saltz. "Using overlays for efficient data transfer over shared wide-area networks". In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, 47:1–47:12.

[115] Rusty Klophaus. "Riak core: building distributed applications without shared state". In: *ACM SIGPLAN Commercial Users of Functional Programming*. ACM. 2010, p. 14.

[116] Tevfik Kosar, Engin Arslan, Brandon Ross, and Bing Zhang. "StorkCloud: Data Transfer Scheduling and Optimization As a Service". In: *Proceedings of the 4th ACM Workshop on Scientific Cloud Computing*. Science Cloud '13. New York, New York, USA: ACM, 2013, pp. 29–36.

[117] Tevfik Kosar and Miron Livny. "A Framework for Reliable and Efficient Data Placement in Distributed Computing Systems". In: *J. Parallel Distrib. Comput.* 65.10 (Oct. 2005), pp. 1146–1157.

[118] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka : a Distributed Messaging System for Log Processing". In: *NetDB Conference*. Athens, Greece, 2011.

[119] Michael Kuhn. "A semantics-aware I/O interface for high performance computing". In: *International Supercomputing Conference*. Springer. 2013, pp. 408–421.

[120] Chinmay Kulkarni, Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. "Beyond Simple Request Processing with RAMCloud". In: *IEEE Data Eng. Bull.* 40 (2017), pp. 62–69.

[121] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[122] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[123] Neal Leavitt. "Will NoSQL databases live up to their promise?" In: *Computer* 43.2 (2010).

[124] Christiane Lefevre. *The CERN accelerator complex*. Tech. rep. 2008.

[125] Justin J Levandoski, Per-Ake Larson, and Radu Stoica. "Identifying hot and cold data in main-memory databases". In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. IEEE. 2013, pp. 26–37.

[126] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. "Semantics and Evaluation Techniques for Window Aggregates in Data Streams". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 311–322.

[127] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark". In: *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM. 2015, pp. 53–63.

[128] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, and David Andersen. "Architecting to achieve a billion requests per second throughput on a single key-value store server platform". In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 476–488.

[129] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. "Scientific workflow scheduling with provenance support in multisite cloud". In: *12th International Meeting on High-Performance Computing for Computational Science (VECPAR)*. 2016, pp. 1–8.

[130] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. "On the role of burst buffers in leadership-class storage systems". In: *International Symposium on Mass Storage Systems and Technologies*. IEEE. 2012, pp. 1–11.

[131] Wantao Liu, Brian Tieman, Rajkumar Kettimuthu, and Ian Foster. "A data transfer framework for large-scale science experiments". In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC '10. Chicago, Illinois: ACM, 2010, pp. 717–724. ISBN: 978-1-60558-942-8.

[132] Isaac Lopez. *IDC Talks Convergence in High Performance Data Analysis*. https://www.datanami.com/2013/06/19/idc_talks_convergence_in_high_performance_data_analysis/. 2013. (Visited on 06/15/2017).

[133] David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. "Semantics of Data Streams and Operators". In: *Proceedings of the 10th International Conference on Database Theory*. ICDT'05. Edinburgh, UK: Springer-Verlag, 2005, pp. 37–52.

[134] Michael Malak. *Parallel vs. Distributed file systems: Time for RAID on Hadoop?* Tech. rep. Data Science Association, 2014.

[135] Nathan Marz and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Greenwich, CT, USA: Manning Publications Co., 2015.

[136] John Meehan et al. "S-Store: Streaming Meets Transaction Processing". In: *Proc. VLDB Endow.* 8.13 (Sept. 2015), pp. 2134–2145.

[137] Nimrod Megiddo and Dharmendra S Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." In: *FAST*. Vol. 3. 2003, pp. 115–130.

[138] *Microsoft Azure Managed Cache Service*. (Visited on 03/11/2017).

[139] *Microsoft Azure Service Bus - Cloud Messaging Service*. (Visited on 03/11/2017).

[140] *Microsoft Cloud Services - Deploy web apps and APIs*. (Visited on 03/14/2017).

[141] Vivek Mishra. "Titan graph databases with cassandra". In: *Beginning Apache Cassandra Development*. Springer, 2014, pp. 123–151.

[142] Bruce Momjian. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York, 2001.

[143] Henry M. Monti, Ali R. Butt, and Sudharshan S. Vazhkudai. "CATCH: A Cloud-Based Adaptive Data Transfer Service for HPC". In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1242–1253.

[144] Dmitry Namiot. "Time Series Databases." In: *DAMDID/RCDL*. 2015, pp. 132–137.

[145] Gábor Németh, Dániel Géhberger, and Péter Mátray. "DAL: A Locality-optimizing Distributed Shared Memory System". In: *Proceedings of the 9th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'17. Santa Clara, CA: USENIX Association, 2017, pp. 12–12.

[146] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. "BlobSeer: Next-generation data management for large scale infrastructures". In: *Journal of Parallel and Distributed Computing* 71.2 (2011), pp. 169–184.

[147] Bogdan Nicolae, Pierre Riteau, and Kate Keahey. "Bursting the Cloud Data Bubble: Towards Transparent Storage Elasticity in IaaS Clouds". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. May 2014, pp. 135–144.

[148] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. "Samza: Stateful Scalable Stream Processing at LinkedIn". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1634–1645.

[149] Eduardo Ogasawara, Jonas Dias, Fabio Porto, Patrick Valduriez, and Marta Mattoso. "An algebraic approach for data-centric scientific workflows". In: *Proceedings of VLDB Endowment* 4.12 (2011), pp. 1328–1339.

[150] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. "Fast Crash Recovery in RAMCloud". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 29–41.

[151] John Ousterhout et al. "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM". In: *SIGOPS Oper. Syst. Rev.* 43.4 (Jan. 2010), pp. 92–105.

[152] Jean-Pierre Panziera. *ETP4HPC Strategic Research Agenda*. Tech. rep. ETP4HPC, 2017.

[153] Youngwoo Park, Seung-Ho Lim, Chul Lee, and Kyu Ho Park. "PFFS: a scalable flash memory file system for the hybrid architecture of phase-change RAM and NAND flash". In: *Proceedings of the 2008 ACM symposium on Applied computing*. ACM. 2008, pp. 1498–1503.

[154] Aaron Parrott and Lane Warshaw. *Industry 4.0 and the digital twin*. Tech. rep. Deloitte University Press, 2018.

[155] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. "A Comparison of Approaches to Large-scale Data Analysis". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 165–178.

[156] *Pravega*. http://www.pravega.io. (Visited on 12/05/2018).

[157] *Project Tungsten*. https://databricks.com/blog/2015/04/28/. 2018. (Visited on 12/05/2018).

[158] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 29–29.

[159] Daniel A Reed and Jack Dongarra. "Exascale computing and big data". In: *Communications of the ACM* 58.7 (2015), pp. 56–68.

[160] David P Reed. "Implementing atomic actions on decentralized data". In: *ACM Transactions on Computer Systems (TOCS)* 1.1 (1983), pp. 3–23.

[161] Robert B Ross, Rajeev Thakur, et al. "PVFS: A parallel file system for Linux clusters". In: *Proceedings of the 4th annual Linux showcase and conference*. 2000, pp. 391–430.

[162] *Running Spark on YARN*. https://spark.apache.org/docs/latest/running-on-yarn.html. 2017. (Visited on 12/05/2018).

[163] Salvatore Sanfilippo. *Redis Cache*. (Visited on 03/11/2017).

[164] Philip Schwan et al. "Lustre: Building a file system for 1000-node clusters". In: *Proceedings of the 2003 Linux symposium*. Vol. 2003. 2003, pp. 380–386.

[165] Philip Schwan et al. "Lustre: Building a file system for 1000-node clusters". In: *Annual Linux Symposium*. Vol. 2003. 2003, pp. 380–386.

[166] Galen M Shipman, David A Dillow, H Sarp Oral, and Feiyi Wang. *The Spider center wide file system; from concept to reality*. Tech. rep. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States); Center for Computational Sciences, 2009.

[167] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The hadoop distributed file system". In: *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee. 2010, pp. 1–10.

[168] Benoit Sigoure. "OpenTSDB: The distributed, scalable time series database". In: *Proc. OSCON* 11 (2010).

[169] Yogesh Simmhan, Catharine van Ingen, Girish Subramanian, and Jie Li. "Bridging the Gap between Desktop and the Cloud for eScience Applications". In: *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 474–481.

[170] Victor Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. "Aerospike: architecture of a real-time operational DBMS". In: *Proceedings of the VLDB Endowment* 9.13 (2016), pp. 1389–1400.

[171] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Transactions on Networking (TON)* 11.1 (2003), pp. 17–32.

[172] Miklos Szeredi. *FUSE: Filesystem in userspace*. http://fuse.sourceforge.net/. 2010. (Visited on 12/05/2018).

[173] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. *Managing update conflicts in Bayou, a weakly connected replicated storage system*. Vol. 29. 5. ACM, 1995.

[174] *The ABrain Project*. https://www.msr-inria.fr/projects/a-brain/. (Visited on 01/01/2018).

[175] *The ANR MapReduce Project*. http://mapreduce.inria.fr/. (Visited on 01/01/2018).

[176] *The ANR OverFlow project*. https://sites.google.com/view/anroverflow/home. (Visited on 01/01/2018).

[177] *The BigStorage Project*. http://bigstorage-project.eu. 2018. (Visited on 01/01/2018).

[178] *The HIRP Project on Low Latency for Stream Storage*. http://innovationresearch.huawei.com/IPD/hirp/portal/. (Visited on 01/01/2018).

[179] *The Stratosphere Project*. http://stratosphere.eu. (Visited on 01/01/2018).

[180] *The ZCloudFlow Project*. https://www.msr-inria.fr/projects/z-cloudflow-data-workflows-in-the-cloud/. (Visited on 01/01/2018).

[181] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. "iPerf: TCP/UDP bandwidth measurement tool". In: *Networks* (Jan. 2005).

[182] Kostas Tzoumas. *High-throughput, low-latency, and exactly-once stream processing with Apache Flink*. Tech. rep. Data Artisans, 2015.

[183] Vinod Kumar Vavilapalli et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 5:1–5:16.

[184] Murali Vilayannur, Samuel Lang, Robert Ross, Ruth Klundt, Lee Ward, et al. *Extending the POSIX I/O interface: a parallel file system perspective*. Tech. rep. Argonne National Laboratory (ANL), 2008.

[185] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. "Kepler + Hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems". In: *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*. WORKS'09. Portland, Oregon, 2009, pp. 1–8.

[186] Lei Wang et al. "BigDataBench: A big data benchmark suite from Internet services". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2014.

[187] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. "Ceph: A scalable, high-performance distributed file system". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320.

[188] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn. "Rados: a scalable, reliable storage service for petabyte-scale storage clusters". In: *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*. ACM. 2007, pp. 35–44.

[189] *What is Hadoop Yarn*. Tech. rep. COSO IT, 2017.

[190] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. "Druid: A Real-time Analytical Data Store". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 157–168.

[191] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10.

[192] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *International Workshop on Hot Topics in Cloud Computing*. Vol. 10. USENIX. 2010, pp. 1–7.

[193] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. ACM, 2013, pp. 423–438.

[194] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438.

[195] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. "DeltaFS: Exascale file systems scale better without dedicated servers". In: *Proceedings of the 10th Parallel Data Storage Workshop*. ACM. 2015, pp. 1–6.