KerA: A Unified Ingestion and Storage System for Scalable Big Data Processing

Ovidiu-Cristian Marcu

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

INSA RENNES

THESE DE DOCTORAT DE

L'INSA RENNES

COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Ovidiu-Cristian MARCU »

« KerA : A Unified Ingestion and Storage System for Scalable Big Data Processing »

Thèse présentée et soutenue à « Inria Rennes Bretagne Atlantique », le « 18 Décembre 2018»
Unité de recherche : Inria Rennes Bretagne Atlantique

Rapporteurs avant soutenance :

1. Patrick VALDURIEZ, H, Directeur de Recherche (DR), Inria Sophia Antipolis - Méditerranée
2. Frédéric DESPREZ, H, Directeur de Recherche (DR), Inria Grenoble Rhône-Alpes

Composition du Jury :

1. Tilmann RABL, H, Directeur de Recherche (DR), Berlin Big Data Center, examinateur
2. Nicolae ȚĂPUȘ, H, Professeur, Politehnica University Politehnica of Bucharest, examinateur
3. Frédéric DESPREZ, H, Directeur de Recherche (DR), Inria Grenoble Rhône-Alpes, examinateur
4. María S. PÉREZ-HERNÁNDEZ, F, Professeure, Universidad Politécnica de Madrid, co-encadrant de thèse
5. Alexandru COSTAN, H, Maître de Conférences – Assistant Professor, INSA Rennes, co-encadrant de thèse
6. Gabriel ANTONIU, H, Directeur de Recherche (DR), Inria Rennes Bretagne Atlantique, directeur de thèse

Intitulé de la thèse :

KerA: Un Système Unifié d'Ingestion et de Stockage pour le Traitement Efficace du Big Data

Ovidiu-Cristian MARCU

En partenariat avec :

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Context

BIG Data is now the new natural resource. In the last few years we have witnessed an unprecedented growth of data that need to be processed at always increasing rates in order to extract valuable insights (e.g., Facebook, Amazon, LHC, etc.). As data volumes continue to rise at even higher velocity, current Big Data analytics architectures face significantly higher challenges in terms of scalability, fast ingestion, processing performance and storage efficiency.

In order to cope with these massive, exponentially increasing amounts of heterogeneous data that are generated faster and faster, Big Data analytics applications have seen a shift from batch processing to stream processing, which can reduce dramatically the time needed to obtain meaningful insights. At the same time, complex workflow applications forced users to demand unified programming abstractions needed to simplify existing workloads and to enable new applications. Processing architectures evolved and now support batch, interactive and streaming computations in the same runtime [18, 120, 2], therefore optimizing applications by removing work duplication and resource sharing between specialized computation engines.

Current state-of-the-art Big Data analytics architectures are built on top of a three layer stack: data streams are first acquired by the ***ingestion layer*** (e.g., Apache Kafka [46]) and then they flow through the ***processing layer*** (e.g., Apache Flink [26], Apache Spark [98], Twitter Heron [59]) which relies on the ***storage layer*** (e.g., HDFS [95]) for storing aggregated data, intermediate checkpoints or for archiving streams for later processing. Under these circumstances, data are often written twice to disk or sent twice over the network (e.g., as part of a fault-tolerance strategy of the ingestion layer and of the persistency requirement of the storage layer). Second, the lack of coordination between the layers can lead to I/O interference (e.g., the ingestion layer and the storage layer compete for the same I/O resources when collecting data streams and writing archival data simultaneously). Third, in order to efficiently handle the application state [107] (batch and streaming) during execution, the processing layer often implements custom advanced data management (e.g., operator state persistence, checkpoint-restart) on top of inappropriate basic ingestion and storage APIs, which results in significant performance overhead.

Unfortunately, in spite of potential benefits brought by specialized layers (e.g., simplified implementation), moving large quantities of data through specialized systems is not efficient: instead, data should be acquired, processed and stored while minimizing the number of copies. We argue that the aforementioned challenges are significant enough to offset the benefits of specializing each layer independently of the other layers.

Moreover, the difficulty to maintain such complex architectures (i.e., robust and fault-tolerant systems that are able to serve both online and offline data access patterns required by modern streaming use cases) suggests the need for ***an optimized (unified) solution for bounded (objects) and unbounded (streams) data ingestion and storage***. The design and implementation of such a dedicated solution is challenging: not only it should provide traditional storage functionality (i.e., support for objects), but it should also meet the real-time access requirements of modern stream-based applications (e.g., low-latency I/O access to data items or high throughput stream data ingestion and processing).

This thesis was carried out in the context of the BigStorage ETN with the goal of understanding the limitations of state-of-the-art Big Data analytics architectures and to design and implement data processing and streaming models to alleviate from current limitations and to optimize stream processing.

1.2 Objectives

This dissertation argues that a plausible path to follow to alleviate from previous limitations is the careful design and implemention of a ***unified architecture for stream ingestion and storage*** which can lead to the optimization of the processing of Big Data (stream-based) applications, while minimizing data movement within the analytics architecture, finally leading to better performance and to better utilized resources. Towards this general goal we propose a set of objectives:

1. Identify a set of requirements for a dedicated stream ingestion/storage engine.
2. Explain the impact of the different Big Data architectural choices on end-to-end performance and understand what are the current limitations faced by a streaming engine when interacting with a storage system for holding streaming state.

3. Propose a set of design principles for a scalable, unified architecture for data ingestion and storage.

4. Implement a prototype for a dedicated stream ingestion/storage engine with the goal of efficiently handling diverse access patterns: low-latency access to stream records and/or high throughput access to (unbounded) streams and/or objects.

1.3 Contributions

Towards the objectives previously mentioned, this thesis brings the following contributions:

Requirements for a dedicated stream ingestion and storage solution. We identify and discuss the application characteristics of a set of modern stream-based scenarios that further inspired a set of challenging requirements for an optimized ingestion and storage architecture. Then, we survey state-of-the-art ingestion and storage systems and discuss how they partially meet our requirements. These systems served as an excellent foundation and inspiration on which to build our proposed solution for a unified architecture for stream ingestion and storage with the goal of optimizing processing analytics.

Understanding performance in Big Data analytics frameworks

Stream-based versus batch-based execution runtime. Flink (stream-based) and Spark (batch-based) are two Apache-hosted data analytics frameworks that facilitate the development of multi-step data pipelines using directly acyclic graph patterns. Making the most out of these frameworks is challenging, because efficient executions strongly rely on complex parameter configurations and on an in-depth understanding of the underlying architectural choices. To this end, we develop a methodology for correlating the parameter settings and the operators execution plan with the resource usage, and we dissect the performance of Spark and Flink with several representative batch and iterative workloads on up to 100 nodes. Our key finding is that none of the two frameworks outperforms the other for all data types, sizes and job patterns: we highlight how results correlate to operators, to resource usage and to the specifics of the engine design and parameters.

Our experiments suggest that the state management function for Big Data processing should be enabled by a dedicated engine like the one we propose. Moreover, designing a unified analytics architecture through stream-based interfaces can be a compelling choice. Our intuition is the following: treating both input/output data and respectively the (intermediate) state of the application operators as streams can finally lead to a simplified, fault-tolerant and optimized analytics architecture.

Exploring shared state for window-based streaming analytics. Big Data streaming analytics tools enable real-time handling of live data sources by means of stateful aggregations through window-based operators: e.g., Apache Flink enables each operator to work in isolation by creating data copies. To minimize memory utilization, we explore the feasibility of deduplication techniques to address

the challenge of reducing memory footprint for window-based stream processing without significant impact on performance (typically measured as result latency). Our key finding is that more fine-grained interactions between streaming engines and (key-value) stores need to be designed (e.g., lazy deserialization or pushing processing functions to storage) in order to better respond to scenarios that have to overcome memory scarcity. These insights contributed to our proposed design principles.

Design principles for a scalable, unified architecture for data ingestion and storage. Based on our experiences with Big Data analytics frameworks and considering the identified requirements for a dedicated stream ingestion and storage solution, we propose a series of design principles for building a scalable, unified architecture for data ingestion and storage in support of more efficient processing analytics. First, towards our vision for a unified analytics architecture, we propose that processing engines should focus on the operator workflow (i.e., on the execution of stateful and/or stateless operators) and leave the state management function to a unified ingestion/storage engine that addresses high-level data management (e.g., caching, concurrency control). Furthermore, processing and ingestion/storage engines should interact through stream-based interfaces.

Next, for a scalable data ingestion we propose two core ideas: (1) dynamic partitioning based on semantic grouping and sub-partitioning, which enables more flexible and elastic management of stream partitions; (2) lightweight offset indexing (i.e., reduced stream offset management overhead) optimized for sequential record access. Then, we propose a set of principles for handling diverse access patterns: low-latency access to stream records and/or high throughput access to streams, records or objects. In order to efficiently handle multiple streams, we propose a new method for adaptive and fine-grained replication that allows durable ingestion of multiple streams. Finally, we discuss two design principles (i.e., data locality support and distributed metadata management) that can lead to an optimized integration of the ingestion/storage systems with processing engines.

Prototype implementation of a unified architecture for data ingestion and storage. Based on the design principles previously mentioned, we introduce the KerA architecture for an optimized ingestion/storage engine for efficient Big Data processing. We present our unified data model for unbounded streams, records and objects, and then describe the protocols of dynamic partitioning and lighweight offset indexing mechanisms for scalable ingestion. We further describe the KerA architecture and present the role of each component for stream ingestion and storage management, and then we characterize the client interfaces (APIs) exposed by the unified ingestion/storage engine. Finally, we comment on the design elements needed for a fault-tolerant architectural implementation. We further describe the implementation of the KerA techniques that we developed in a high-performance software prototype based on C++ and Java. We zoom on the adaptive and fine-grained replication implementation, which uses a zero-copy virtual log technique and we describe the architectural extension and implementation of the data locality support for streaming operations. In the end, we evaluate our implementation through synthetic workloads.

1.4 Publications

International conferences

[ICDCS2018] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez, Bogdan Nicolae, Radu Tudoran, Stefano Bortoli. ***KerA: Scalable Data Ingestion for Stream Processing***. In IEEE International Conference on Distributed Computing Systems, Jul 2018, Vienna, Austria, https://hal.inria.fr/hal-01773799.

[Cluster2016] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez. ***Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks***. In IEEE International Conference on Cluster Computing, Sep 2016, Taipei, Taiwan, https://hal.inria.fr/hal-01347638v2.

Workshops at international conferences

[BigData2017] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez, Radu Tudoran, Stefano Bortoli, Bogdan Nicolae. ***Towards a Unified Storage and Ingestion Architecture for Stream Processing***. In Second Workshop on Real-time & Stream Analytics in Big Data Colocated with the 2017 IEEE International Conference on Big Data, Dec 2017, Boston, United States, https://hal.inria.fr/hal-01649207.

[CCGrid2017] Ovidiu-Cristian Marcu, Radu Tudoran, Bogdan Nicolae, Alexandru Costan, Gabriel Antoniu, María Pérez. ***Exploring Shared State in Key-Value Store for Window-Based Multi-Pattern Streaming Analytics***. In Workshop on the Integration of Extreme Scale Computing and Big Data Management and Analytics in conjunction with IEEE/ACM CCGrid, May 2017, Madrid, Spain, https://hal.inria.fr/hal-01530744.

1.5 Dissertation plan

This thesis proceeds as follows. The first part (Chapter 2) describes the main requirements of a unified ingestion and storage architecture for efficient Big Data processing and presents limitations of state-of-the-art ingestion and storage systems for handling them. The second part (Chapters 3 and 4) is focused on how current Big Data processing engines perform when handling batch and streaming workloads. Chapter 3 provides a methodology for understanding the architectural differences between stream-based and batch-based execution runtimes. Chapter 4 explores the feasibility of deduplication techniques to address the challenge of reducing memory footprint for window-based streaming operations. The third part (Chapters 5, 6, 7 and 8) is focused on the design, implementation and evaluation of the KerA's unified architecture for optimized stream ingestion and storage. We design KerA with the goal of efficiently handling diverse access patterns: low-latency access to stream records and/or high throughput access to (unbounded) streams and/or objects. Based on identified requirements and our experiences with Big Data processing architectures, Chapter 5 describes a set of design principles for a scalable data ingestion and storage architecture. Chapter 6 presents a high-level description of the KerA architecture that implements these

principles. Further, the implementation of the most challenging parts of the KerA architecture is described in Chapter 7. Chapter 8 presents an evaluation of the KerA techniques. In the last part (Chapter 9) we conclude and discuss future research directions.

Part I

Background

Chapter 2

Data ingestion and storage support for Big Data processing

Contents

STREAM-based applications need to immediately ingest and analyze data and in many cases combine live (unbounded streams) and archived (objects, i.e., bounded streams) data in order to extract better insights. In this context, online and interactive Big Data processing runtimes (e.g., Apache Flink [26], Apache Spark [121]) designed for both batch and stream processing are rapidly adopted in order to eventually replace traditional, batch-oriented processing models (such as MapReduce [18] and its open-source implementation Hadoop [37]) that are insufficient to additionally meet the need for low-latency and high-update frequency of streams [69, 105, 106, 5, 14].

The broad question we address in this thesis is ***how to build a general Big Data streaming architecture that is able to efficiently handle very diverse stream applications***, while minimizing data movement for better resource utilization and improved performance? To

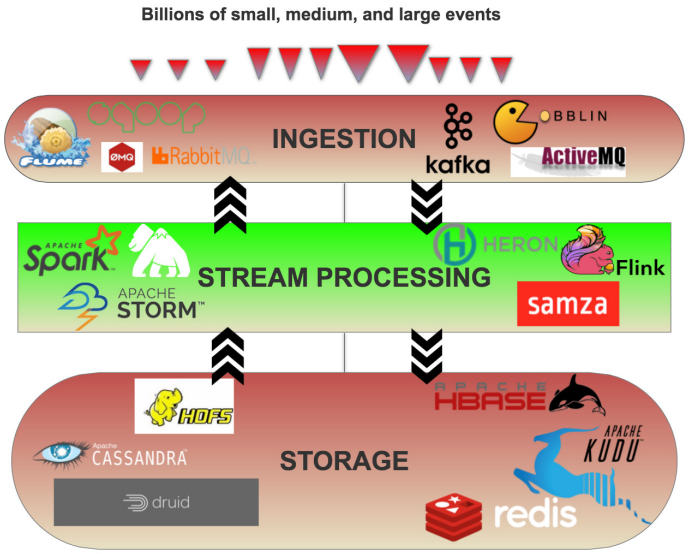

Figure 2.1 – The usual streaming architecture is composed of three layers: data streams are first acquired by the ***ingestion layer*** and then they flow through the ***processing layer*** which relies on the ***storage layer*** for storing aggregated data, intermediate results or for archiving streams for later processing.

this end, we first present the state-of-the-art stream processing pipeline and then discuss application requirements for a set of modern stream-based use cases.

2.1 The stream processing pipeline

A typical state-of-the-art online Big Data analytics runtime is built on top of a three layer stack (as illustrated in Figure 2.1):

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), having no support for object access.

Storage: unlike the ingestion layer, this layer is responsible for persistent storage of data (i.e., objects or files) and can additionally be used to simply archive processed streams (for later usage). This typically involves either the archival of the buffered data streams from the ingestion layer or the storage of the intermediate/final results of stream analytics, both of which are crucial to enable fault-tolerance or deeper, batch-oriented analytics that complement the online analytics; the persistent storage layer is typically based on HDFS [95], for interoperability reasons with previous developments in the area of Big Data analytics, although it was not designed for streaming patterns and, as such, it cannot provide the required sub-second performance (e.g., online access to data repositories).

Processing: this layer consumes the streaming data buffered by the ingestion layer and sends the intermediate/final results to the ingestion layer in order to enable other streaming applications in a real-time workflow scenario; additionally, the processing layer may send intermediate/final results to the storage layer for ensuring a fault-tolerant execution (enabling later recovery of stored checkpoints).

Current streaming architectures are designed with distinct components for ingestion and storage of data streams. The difficulty to maintain such complex architectures (i.e., robust and fault-tolerant systems that are able to serve a wide range of workloads, e.g., to support both online and offline data access patterns required by modern streaming use cases) suggests the need for ***an optimized solution for (un)bounded stream data ingestion and storage***. Such a solution should of course keep providing traditional storage functionality (i.e., support for persistent objects), but should also meet the real-time access requirements of stream-based processing (e.g., low-latency I/O access to data items and high throughput stream ingestion).

2.2 Motivating use cases: processing and data requirements

Stream processing can solve a large set of business and scientific problems, including network monitoring, real-time fraud detection, e-commerce, etc. In short, these applications require real-time processing of stream data in order to gather valuable insights that immediately contribute with results for final users: data streams are pushed to stream ingestion systems and queries are continuously executed over them [43]. We describe below examples of modern stream-based scenarios that exhibit challenging requirements for state-of-the-art ingestion and storage systems.

Monetizing streaming video content. Streaming video providers display video advertisements and are interested in efficiently billing their advertisers [1]. Both video providers and advertisers need statistics about their videos (e.g., how long a video is watched and by which demographic groups); they need this information as fast as possible (i.e., in real-time) in order to optimize their strategy (e.g., adjust advertisers budgets). We identify a set of requirements associated with these applications: (1) stream items are ***ingested*** as fast as possible and consumed by processing engines that are updating statistics in real-time; (2) stream items and aggregated results are ***durably stored*** for future usage (e.g., offline analysis); (3) users ***interrogate*** streams (SQL queries on streams, monitoring) to validate business quality agreements.

Distributed system/network monitoring. The log component is a first-class member of many business or scientific applications because it can deliver a concise description of the current status of a running application. There are many use cases [19, 16] related to real-time log processing: monitoring server metrics (CPU, memory, disk, network) in order to gain insights into the system health and performance; monitoring application usage to gain insights into user activity and get real-time alerts and notifications that help maintain the service level agreements. In most scenarios data is immediately dropped after it gets filtered, while some cases require data to be durably retained (archived) for a configurable amount of time. Real-time monitoring involves certain

steps: (1) stream logs are ***ingested***, potentially indexed and/or archived in real time; (2) monitoring systems run continuous queries on live and/or archived data and generate alerts if problems are identified; (3) long living stream analytics are deployed to gather more insights post-ingestion; (4) dynamic partitioning techniques are necessary to handle peak moments when data streams arrive with higher throughput.

Decision support for Smart Cities applications. Future cities will leverage smart devices and sensors installed in the public infrastructure to improve the citizens' life. In this context, several aspects are important: (1) streaming data from sensors (pre-processed at the edge) are initially ***ingested*** before delivering them to the streaming engines; (2) massive quantities of data can be received over short time intervals, consequently ingestion components have to support ***a high throughput of streaming data***; (3) stream-based applications need to efficiently scale up and down based on data input, which further introduces challenges for implementing dynamic data partitioning techniques.

Summary and discussion.

To sum up, stream-based applications strongly rely on the following features, not well supported by current streaming architectures:

1. ***Fast ingestion***, possibly doubled by ***simultaneous indexing*** (often, through a single pass on data) for real-time processing.
2. ***Low-latency storage*** with additional ***fine-grained query support*** for efficient filtering and aggregation of data records.
3. Storage coping with events accumulating in ***large volumes over a short period of time***.

A few general trends can be observed from the applications presented above. First, the data access model is complex and involves a diversity of data sizes and access patterns (i.e., records, streams, objects). Second, the stream processing architectures need to enable advanced features for the applications, such as dynamic partitioning, distributed metadata management, pre-preprocessing, flow control. In addition to these features, they need to also address non-functional aspects such as high availability, data durability and control of latency versus throughput. Based on these general trends, we synthetized the following requirements for an optimized stream storage architecture.

2.3 Requirements for stream ingestion and storage

Current Big Data streaming architectures (Figure 2.1) rely on message broker solutions that decouple data sources (i.e., how data streams are ingested) from applications (i.e., how streams are processed) that further store (intermediate) results and input streams to storage engines. Moving large quantities of data through specialized layers is not efficient and it also forces users to handle very complex architectures. We claim that a unified architecture for ingestion and storage would help overcome these limitations. Let us discuss ***the main requirements that need to be addressed by a unified solution for data ingestion and storage of stream-based applications***.

Data access patterns: records, streams and objects. Stream-based applications dictate the way data is accessed, with some of them (e.g., [36, 1]) requiring fine-grained record access. Stream operators may leverage multi-key-value interfaces in order to optimize the number of accesses to the data store or may access data items sequentially (batches of stream records). Moreover, when coupled with offline analytics (post-ingestion), the need for object (bounded streams) data access interfaces is crucial.

Dynamic stream partitioning. Partitioning for streaming [117, 31] is a recognized technique used in order to increase processing throughput and scalability. It consists of logically spliting a stream of records in multiple partitions (managed by distinct brokers) that can be accessed in parallel by multiple producers and consumers. The ability of the stream storage to handle increasing amounts of data, while remaining stable in front of peak moments when high rates of bursting data arrive, is crucial. The main key to a scalable stream storage is a dynamic stream partitioning model with advanced, fine-grained support for application-level message routing [75].

Temporal persistence versus stream durability. Temporal persistence refers to the ability of a system to temporarily store a stream of events in-memory or on disk (e.g., configurable data retention of a short period of time after which data is automatically discarded). Big Data analytics engines may benefit from such support, specifically when pushing processing to storage. Moreover, stream records may also need to be durably stored for high availability and later processing. It is of utmost importance to be able to configure a diverse set of retention policies in order to give applications the ability to replay historical stream events or to derive later insights by running continuous queries over both old and new stream data.

Latency versus throughput. Some applications need low-latency access to streams of records, while others can accommodate higher latencies, requiring high throughput access to objects or streams. Specialized systems implement this trade-off through a key-value data model for low-latency record access or by using a batching architecture on streaming clients with support from storage systems [67]. For a unified ingestion and storage system the implementation of this feature could be more challenging, since it must simultaneously satisfy both low-latency and high throughput application requests.

Data locality support. Most batch analytics engines are optimized by scheduling operators' tasks based on data locality (memory or disk). The locality feature can also be useful for streaming workloads which are very sensitive to latency. Moreover, Big Data analytics frameworks develop complex solutions for in-memory state management due to lack of data locality support from a dedicated low-latency stream storage solution. Even more, stream processing would greatly benefit from ***pushing user-defined aggregate functions to storage*** (a popular technique to avoid moving large amounts of data over the network and to reduce serialization and de-serialization overheads). The "in-storage" procesing feature requires more fine-grained control over acquired data, yet current ingestion/storage solutions do not offer native support for it.

2.4 Selection of distributed systems for data ingestion and storage

A Big Data stream processing architecture is tightly coupled with the data ingestion and storage components. We make an overview of a set of systems that provide ingestion and storage features for streaming and further discuss their limitations with respect to previous requirements.

2.4.1 Stream-based ingestion systems

We first review state-of-the-art open source ingestion/storage systems that provide stream-based access interfaces to producer and consumer clients.

Apache Kafka [49] is a distributed stream platform that provides durability and publish/-subscribe functionality for data streams (making stream data available to multiple consumers). It is the ***de-facto*** open-source solution used in end-to-end pipelines with streaming engines like Apache Spark or Apache Flink, which in turn handle data movement and computation. A Kafka cluster comprises a set of broker nodes that store streams of records in categories called topics [46]. Each stream can be statically split into multiple ***partitions***, allowing to logically split stream data and parallelize consumer access.

DistributedLog [96] 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, including support for transactions. A stream is statically partitioned into a fixed number of partitions, each partition being backed by a log; a log's segments are spread over multiple nodes managed by Bookkeeper [48], which is a scalable, fault-tolerant storage service optimized for real-time workloads. In DistributedLog there is only one active writer for a log at a given time. The reader starts reading records at a certain position (offset) until it reaches the tail of the log. At this point, the reader waits to be notified about new log segments or records.

Apache Pulsar [91] is a pub-sub messaging system developed on top of Bookkeeper, with a two-layer architecture composed of a stateless serving layer and a stateful persistence layer. Compared to DistributedLog, reads and writes cannot scale independently (first layer is shared by both readers and writers) and Pulsar clients do not interact with Bookkeeper directly. Pulsar unifies the queue and topic models, providing exclusive, shared and failover subscription models to its clients [73]. Pulsar keeps track of consumer cursor position, being able to remove records once acknowledged by consumers, simplifying memory management.

Pravega [89] is another open-source stream storage system built on top of Bookkeeper. Pravega 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.

2.4.2 Record-based storage systems

Next, we review two state-of-the-art key-value stores.

Redis [93] is an in-memory data structure 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 [112].

RAMCloud [86] 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 [50], achieving high availability by distributing indexes independently from their objects (independent partitioning).

These systems primarly optimize for low-latency fine-grained record access, with limited support for high throughput stream ingestion.

2.4.3 HDFS

Streaming architectures evolve in large and complex systems in order to accommodate many use cases. Ingestion tools are connected either directly or by streaming engines to the storage layer. The Hadoop Distributed File System (HDFS) [95] provides scalable and reliable data storage (sharing many concepts with GoogleFS [32]), and is recognized as the de-facto standard for Big Data analytics storage. Although HDFS was not designed with streams in mind, many streaming engines (e.g., Apache Spark, Apache Flink, etc.) depend on it (e.g., for persistence they define HDFS sinks or for storing checkpoints). Among HDFS limitations, we recall the metadata server (called NameNode) as a single point of failure and a source of limited scalability, although solutions exist [79] to overcome the metadata bottleneck. Also, there is no support for random writes as a consequence of the way the data written are available for readers (only after the file is closed, data can be appended to files). In fact, in [20] authors point out that HDFS does not perform well for managing a large number of small files, and discuss certain optimizations for improving storage and access efficiency of small files on HDFS. This is why streaming engines develop custom memory management solutions to overcome limitations of HDFS.

2.4.4 Specialized stores

A few other systems (such as Druid and Kudu) are designed as an alternative to executing queries over HDFS with columnar formats like Parquet.

Druid [116, 22] is an open-source, distributed, columnar-oriented data store designed for real-time exploratory analytics on Big Data sets. Its motivation is straightforward: although HDFS is a highly available system, its performance degrades under heavy concurrent load and is not optimized for ingesting data and making data immediately available for processing. A Druid cluster consists of specialized nodes: ***real-time nodes*** (that maintain an in-memory index buffer for all incoming events, regularly persisted to disk) provide functionality to ingest and query data streams; ***historical nodes*** give the functionality to load and serve immutable blocks of data which are created by real-time nodes; a set of ***broker nodes*** that act as query routers to historical and real-time nodes (with metadata published to ZooKeeper [42]). 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.

Apache Kudu [56] is a columnar data store that integrates with Apache Impala [99], HDFS and HBase [39]. It can be seen as an alternative to Avro [9]/Parquet [88] over HDFS (not suitable for updating individual records or for efficient random reads/writes) or an alternative to semi-structured stores like HBase or Cassandra [61] (not a fit for handling mutable data sets, allowing for low-latency record-level reads and writes, but not efficient for sequential read throughput needed by applications like machine learning or SQL). Kudu's cluster is a set of tables: each table has a well-defined schema and consists of a finite number of columns; each column is defined by a type and defines a primary key, but no secondary indexes. Kudu offers fast columnar scans (comparable to Parquet, ORC [84]) and low-latency random updates.

These systems offer specialized support for querying streams, but rely on static stream partitioning and do not offer support for objects.

2.4.5 Limitations of existing systems

Table 2.1 presents how the analyzed systems partially meet previous requirements. Since our goal is to provide efficient access interfaces for records, streams and objects, one potential approach would be to enhance one such system with the missing features. However, this is difficult to achieve since considering the initial design choices of their developers, the other remaining requirements would be difficult, if not impossible to support at the same time. For instance, Kafka's design is based on the OS cache, making it difficult to co-locate with a processing engine; Redis does not offer strong consistency (which some use cases may require); Druid's choice to differentiate real-time and historical nodes was similar to adopting both Kafka and HDFS, in contrast to our goal of minimizing data copies. Although RAMCloud was not designed for high throughput stream ingestion, its key-value data model was a good start for a stream record representation. However, building streaming interfaces on top of RAMCloud was not a choice since many workloads do not require fine-grained record-level access. Therefore, while extending an existing system is not a solution, building a new dedicated solution from scratch leveraging these systems or some of their core ideas as building blocks is a good option.

Requirement / System	Data access patterns	Stream partitioning	Temporal persistence vs. durability	Latency vs. throughput	Data locality support
Apache Kafka	streams	static	both	both	no
Apache Distributedlog	streams	static	durability	both	no
Apache Pulsar	streams	static	both	both	no
Pravega	streams	dynamic	durability	both	no
Redis	records, streams	static	both	both	limited
RAMCloud	records	static	durability	latency	no
Druid	records, streams	static	durability	both	no
Apache Kudu	records	static	durability	throughput	limited
HDFS	objects (files)	static (fixed block size)	durability	throughput	yes

Table 2.1 – How available ingestion and storage systems support the identified requirements.

2.5 Conclusion

As the need for more complex online/offline data manipulations arises, so does ***the need to enable better coupling between the ingestion, storage and processing layers***. Current (three layer) streaming architectures do not efficiently support our identified requirements. Moreover, emerging scenarios emphasize complex data access patterns, (optionally) requiring fault-tolerance and archival of streaming data for deeper analytics based on batch processing. Under these circumstances, data are often written twice to disk or sent twice over the network (e.g., as part of a fault-tolerance strategy of the ingestion layer and the persistency requirement of the storage layer). Second, the lack of coordination between the layers can lead to I/O interference (e.g., the ingestion layer and the storage layer compete for the same I/O resources, when collecting data streams and writing archival data simultaneously). Third, the processing layer often implements custom advanced data management (e.g. operator state persistence, checkpoint-restart) on top of inappropriate basic ingestion/storage API, which results in significant performance overhead. We argue that the aforementioned challenges are significant enough to offset the benefits of specializing each layer independently of the other layers.

Part II

Understanding performance in Big Data analytics frameworks

Chapter 3

Stream-based versus batch-based execution runtimes

Contents

In the last decade, MapReduce ("a programming model and an associated implementation for processing and generating large data sets" [18]) and its open-source implementation, Hadoop [37], were widely adopted by both industry and academia, thanks to a simple yet powerful programming model that hides the complexity of parallel task execution and fault-tolerance from the users. This very simple API comes with the important caveat that it forces applications to be expressed in terms of map and reduce functions ("users specify a map function that processes a key/value pair to generate a set of intermediate key/value

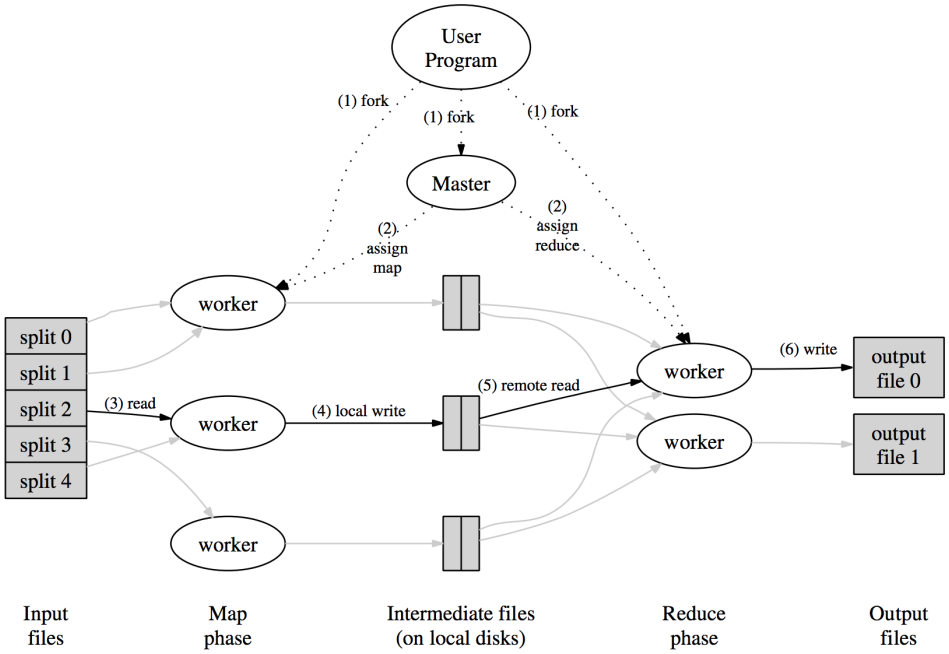

Figure 3.1 – The MapReduce programming model (taken from [18]). "The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function are specified by the user" [18].

pairs, and a reduce function that merges all intermediate values associated with the same intermediate key" [18], see Figure 3.1).

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. Moreover, rising data volumes and the online dimension of data processing require streaming models in order to enable real-time handling of live data sources.

To address these limitations, a second generation of analytics platforms emerged in an attempt to unify the landscape of Big Data processing. In this chapter we focus on two competitors, state-of-the-art Big Data processing engines: Spark [98] (batch-based) introduced Resilient Distributed Datasets (RDDs) [120], a set of in-memory data structures able to cache intermediate data across a set of nodes, in order to efficiently support ***iterative*** algorithms, and Flink [26] (stream-based), that with the same goal, proposed more recently native closed-loop iteration operators [24] and an automatic cost-based optimizer, that is able to reorder the operators and to better support ***low-latency streaming execution***.

Making the most out of these frameworks is challenging because efficient executions strongly rely on complex parameter configurations and on an in-depth understanding of the underlying architectural choices. In order to identify and explain the impact of the different

architectural choices and the parameter configurations on the perceived end-to-end performance, we develop a methodology for correlating the parameter settings and the operators execution plan with the resource usage. We use this methodology to dissect the performance of Spark and Flink with several representative batch and iterative workloads on up to 100 nodes.

In this chapter, our goal is to assess whether using a single engine for all data sources, workloads and environments [78] is efficient or not, and also to study how well frameworks that depend on smart optimizers work in real life [10]. Moreover, understanding the advantages and limitations of a stream-based execution engine (i.e., Flink) versus a batch-based execution engine (i.e., Spark), when running batch and iterative workloads, provides the needed insights for better defining the requirements of a dedicated stream ingestion/storage solution.

The remainder of this chapter is organized as follows. Section 3.1 describes Spark and Flink, highlighting their architectural differences. Section 3.2 presents our methodology, providing a description of the batch and iterative workloads (subsection 3.2.1), emphasizing the impact on performance of parameter configuration (subsection 3.2.2), and finally describing our experimental setup (subsection 3.2.3). Section 3.3 mentions the results alongside a detailed analysis and lists our insights. Finally, in Section 3.4 we survey the related work and we discuss final remarks regarding fault-tolerance trade-offs.

Listing 3.1 – Spark word count program (taken from Apache Spark)

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

Listing 3.2 – Flink word count program (taken from Apache Flink)

```
val counts = text.flatMap { _.toLowerCase.split("\\W+") filter {_.nonEmpty} }
  .map {(_, 1)}
  .groupBy(0)
  .sum(1)
```

3.1 Background

Spark and Flink extend the MapReduce model in order to facilitate the development of multi-step data pipelines using directly acyclic graph (DAG) patterns. At a higher level, both engines implement a driver program that describes the high-level control flow of the application, which relies on two main parallel programming abstractions: (1) structures to describe the data and (2) parallel operations on these data.

While the data representations differ, both Flink and Spark implement similar dataflow operators (e.g., ***map***, ***reduce***, ***filter***, ***distinct***, ***collect***, ***count***, ***save***), or expose an API that can be used to obtain the same result. For instance, Spark's ***reduceByKey*** operator (called on a dataset of key-value pairs to return a new dataset of key-value pairs where the value of each key is aggregated using the given reduce function - see Listing 3.1) is equivalent to Flink's ***groupBy*** followed by the aggregate operator ***sum*** or ***reduce*** (see Listing 3.2).

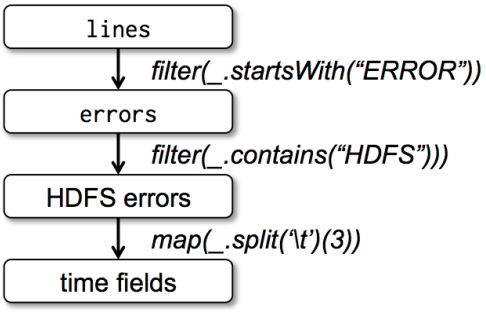

Figure 3.2 – Lineage graph for a Spark query (taken from [120]). "Boxes represent RDDs and arrows represent transformations" [120].

3.1.1 Apache Spark

Spark is built on top of RDDs (read-only, resilient collections of objects partitioned across multiple nodes) that hold provenance information (referred to as ***lineage***) and can be rebuilt in case of failures by partial recomputation from ancestor RDDs (Figure 3.2). Each RDD is by default lazy (i.e., computed only when needed) and ephemeral (i.e., once it actually gets materialized, it will be discarded from memory after its use). However, since RDDs might be repeatedly needed during computations, the user can explicitly mark them as persistent, which moves them in a dedicated cache for persistent objects.

The operations available on RDDs seem to emulate the expressivity of the MapReduce paradigm overall, however, RDDs can be cached for later use, which greatly reduces the need to interact with the underlying distributed file system in more complex workflows that involve multiple reduce operations.

3.1.2 Apache Flink

Flink is built on top of DataSets (collections of elements of a specific type on which operations with an implicit type parameter are defined), Job Graphs and Parallelisation Contracts (PACTs) [113]. Job Graphs represent parallel data flows with arbitrary tasks, that consume and produce data streams and are further translated into execution graphs (see Figure 3.3). PACTs (see Figure 3.4) are second-order functions that define properties on the input/output data of their associated user defined (first-order) functions (UDFs); these properties are further used to parallelize the execution of UDFs and to apply optimization rules [3]. Flink executes a batch program as a special case of streaming (considering bounded streams as input data to programs). For recovery, Flink batching relies on fully replaying the original streams.

3.1.3 Zoom on the differences between Flink and Spark

In contrast to Flink, Spark's users can control two very important aspects of the RDDs: the persistence (i.e., in-memory or disk-based) and the partition scheme across the nodes [120].

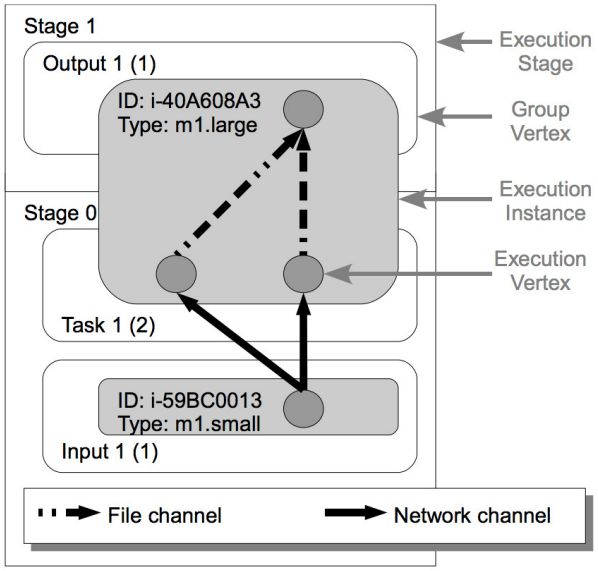

Figure 3.3 – Flink execution graph (taken from [113]). The execution graph defines the mapping of subtasks to instances and the communication channels between them.

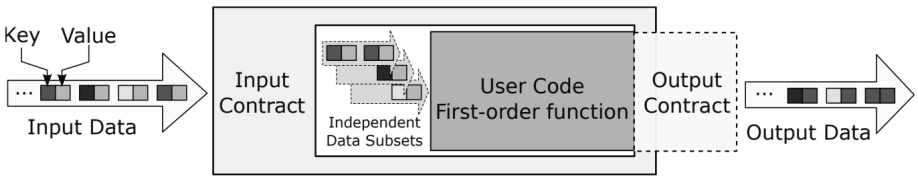

Figure 3.4 – Parallelization contract - PACT (taken from [3]). "PACT consists of exactly one second-order function which is called Input Contract and an optional Output Contract. An Input Contract takes a first-order function with task-specific user code and one or more data sets as input parameters. The Input Contract invokes its associated first-order function with independent subsets of its input data in a data-parallel fashion. Input Contracts can be seen as MapReduce map and reduce functions with additional constructs to complement them" [3].

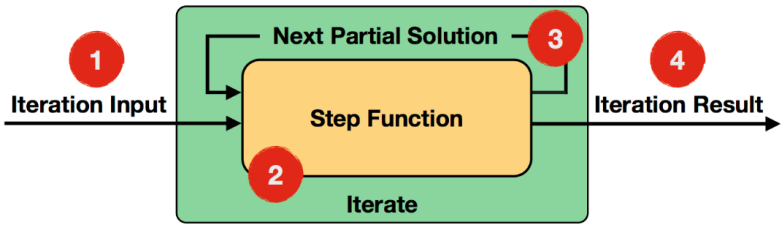

Figure 3.5 – Flink iterate operator (taken from Apache Flink). "It consists of the following steps: (1) Iteration Input: Initial input for the first iteration taken from a data source or from previous operators. (2) Step Function: The step function will be executed in each iteration. It is an arbitrary data flow that consists of operators like map, reduce, join, etc. and that depends on specific tasks. (3) Next Partial Solution: In each iteration, the output of the step function will be fed back into the next iteration. (4) Iteration Result: The output of the last iteration is written to a data sink or it is used as input to the next operators" (Apache Flink).

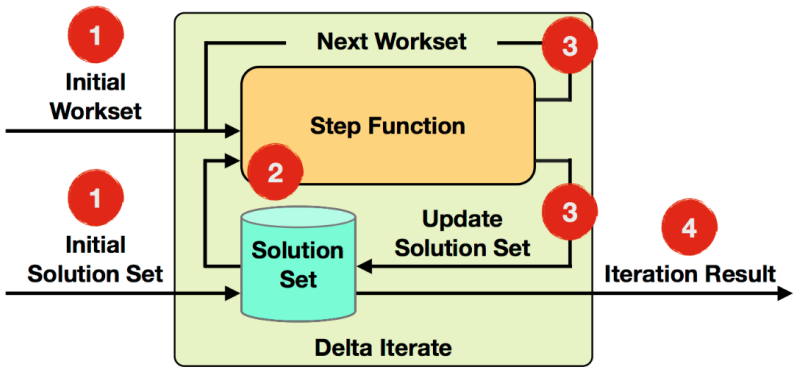

Figure 3.6 – Flink delta iterate operator (taken from Apache Flink). "It consists of the following steps: (1) Iteration Input: The initial workset and solution set are read from data sources or previous operators as input to the first iteration. (2) Step Function: The step function will be executed in each iteration. It is an arbitrary data flow consisting of operators like map, reduce, join, etc. and depends on specific tasks. (3) Next Workset/Update Solution Set: The next workset drives the iterative computation, and will be fed back into the next iteration. Furthermore, the solution set will be updated and implicitly forwarded (it is not required to be rebuilt). Both data sets can be updated by different operators of the step function. (4) Iteration Result: After the last iteration, the solution set is written to a data sink or used as input to the next operators" (Apache Flink).

For instance, this fine-grained control over the storage approach of intermediate data proves to be very useful for applications with varying I/O requirements.

Another important difference relates to iterations handling. 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. 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 are fed back from the tail of an iteration to its head. Basically, data are flowing in cycles around the operators within an iteration. Since operators are just scheduled once, they can maintain a state over all iterations. Flink's API offers two dedicated iteration operators to specify iterations: 1) ***bulk iterations***, which are conceptually similar to loop unrolling (see Figure 3.5), and 2) ***delta iterations***, a special case of incremental iterations in which the solution set is modified by the step function instead of a full recomputation (see Figure 3.6). Delta iterations can significantly speed up certain algorithms because the work in each iteration decreases as the number of iterations goes on.

Considering the above, the main question we aim to answer is: ***how do these different architectural choices impact performance and what are their limitations?***

3.2 Methodology

3.2.1 Workloads

Although they were initially developed to enhance the batch-oriented Hadoop with efficient iterative support, currently Spark and Flink are used conversely for both ***batch*** and ***iter-***

Operators	Batch (one pass)			Iterative (caching)		
	WC	G	TS	KM	PR	CC
map			✓	✓	✓	✓
flatMap	✓				✓	✓
mapToPair (S)	✓					
groupBy → sum (F)	✓					
reduceByKey (S)	✓			✓		
collectAsMap (S)				✓		
filter → count		✓				
distinct					✓	✓
repartitionAndSort - WithinPartitions (S)			✓			
partitionCustom → sortPartition (F)			✓			
Graph specific operators					✓	✓
coalesce, mapPartitionsWithIndex (S)					✓	✓
DeltaIteration, join, groupBy, aggregate (F)						✓
BulkIteration, groupBy, reduce, withBroadcastSet(F)				✓		
save	✓		✓	✓	✓	✓

Table 3.1 – Big Data operators for batch and iterative workloads. Selected operators used in each workload: Word Count (WC), Grep (G), Tera Sort (TS), K-Means (KM), Page Rank (PR), Connected Components (CC). Operators annotated with F or S are specific only to Flink or Spark respectively, the other ones are common for both frameworks.

ative processing. Recent extensions brought SQL [40, 7] and streaming [67, 121] support, providing general architectures for Big Data processing.

For the batch category we have selected three benchmarks implementing the one-pass processing: Word Count, Grep and Tera Sort. These are representative workloads used in several real-life applications, either scientific (e.g., indexing the monitoring data at the Large Hadron Collider [63]) or Internet-based (e.g., search algorithms at Google, Amazon [34, 51]). For the iterative category we have opted for three benchmarks that are mainly used to evaluate the effectiveness of the loop-caching: K-Means, Page Rank and Connected Components. These workloads are frequent in machine learning algorithms [76] and social graphs processing (e.g., at Facebook [25] or Twitter [109]). Table 3.1 lists the use of the most important operators by each workload, including basic core operators and specific ones implemented by the graph libraries of each framework.

Word Count is a simple metric for measuring article quality by counting the total number of occurences of each word. It is a good fit for evaluating the ***aggregation component*** in each framework, since both Spark and Flink use a map-side combiner to reduce the intermediate data. In Flink, the following sequence of operators is applied to the DataSets: ***flatMap*** (map phase) → ***groupBy*** → ***sum*** (reduce phase) → ***writeAsText***. In Spark, the following sequence is applied to RDDs: ***flatMap*** → ***mapToPair*** (map phase) → ***reduceByKey*** (reduce phase) → ***saveAsTextFile***.

Grep is a common command for searching plain-text data sets. Here, we use it to evaluate

the ***filter*** transformation and the ***count*** action. Both Flink and Spark implement the following sequence of operators applied on their specific datasets: ***filter*** → ***count***. For both Word Count and Grep, Spark's RDDs and Flink's DataSets are built by reading Wikipedia text files from HDFS.

Tera Sort is a sorting algorithm suitable for measuring the performance of the two engines core capabilities. We have chosen the implementation described in [104] on 100-byte records, with the first 10-bytes representing the sort key. The input data is generated using the ***TeraGen*** [103] program with Hadoop and the same range partitioner has been used in order to provide a fair comparison. A number of equally sized partitions is generated and a custom partitioner is used based on Hadoop's ***TotalOrderPartitioner***. Spark is creating two RDDs: the first one by reading from HDFS and performing a local sort (***newAPIHadoopFile***) and the second one by repartitioning the first RDD according to the custom partitioner (***repartitionAndSortWithinPartition***). Flink first creates a DataSet from the given HDFS input and then applies a ***map*** to create key-value tuples. These are stored using an ***OptimizedText*** binary format in order to avoid deserialization when comparing two keys. Next, Flink's algorithm partitions the tuple DataSet (***partitionCustom***) on the specified keys using a custom partitioner and applies a ***sortPartition*** to locally sort each partition of the dataset. Finally, the results are saved using the same Hadoop output format.

K-Means is an unsupervised method used in data mining to group data elements with a high similarity. The input is generated using the ***HiBench*** suite [41] (training records with 2 dimensions). In each iteration, a data point is assigned to its nearest cluster center, using a map function. Data points are grouped to their center to further obtain a new cluster center at the end of each iteration. This workload evaluates the effectiveness of the caching mechanism and the basic transformations: ***map***, ***reduceByKey*** (for Flink: ***groupBy*** → ***reduce***), and Flink's ***bulk iterate*** operator.

Page Rank is a graph algorithm which ranks a set of elements according to their references. For Flink we evaluated the vertex-centric iteration implementation from its ***Gelly*** [24] library (iteration operators: ***outDegrees***, ***joinWithEdgesOnSource***, ***withEdges***), while for Spark we evaluated the standalone implementation provided by its ***GraphX*** [33] library (iteration operators: ***outerJoinVertices***, ***mapTriplets***, ***mapVertices***, ***joinVertices***, ***foreachPartition***).

Connected Components gives an important topological invariant of a graph. For Flink we evaluated the vertex-centric iteration implementation (iteration operators: ***mapEdges***, ***withEdges***), while for Spark we evaluated the ***ConnectedComponents*** implementation (iteration operators: ***mapVertices***, ***mapReduceTriplets***, ***joinVertices***) as provided by their respective libraries. In Flink's case, we evaluated a second algorithm expressed using ***delta iterations*** in order to assess their speedup over classic ***bulk iterations***. Both Page Rank and Connected Components are useful to evaluate the caching and the data pipelining performance. For these two workloads, we have used three real datasets (small, medium and large graphs) to validate different cache sizes.

3.2.2 The importance of parameter configuration

Both frameworks expose various execution parameters, pre-configured with default values and allow a further customization. For every workload, we found that different parameter settings were necessary to provide an optimal performance. 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 and the data serialization.

Task parallelism. The meaning and the default values of the parallelism setting are different in the two frameworks. Nevertheless, this is a mandatory configuration for each dataflow operator in order to efficiently use all the available resources. Spark's default parallelism parameter (***spark.def.parallelism***) refers to the default number of partitions in the RDDs returned by various transformations. We set this parameter to a value proportional to the number of cores per number of nodes multiplied by a factor of 2 to 6 in order to experience with a various number of partitions in RDDs for distributed shuffle operations like ***reduceByKey*** and ***join***. Flink's default parallelism parameter (***flink.def.parallelism***) allows to use all the available execution resources (***Task Slots***). We set this parameter to a value proportional to the number of cores per number of nodes. Therefore, in Flink the partitioning of data is hidden from the user and the parallelism setting can be automatically initialized to the total number of available cores.

Shuffle tuning. One difference in configuring Flink and Spark lies in the mandatory settings for the network buffers, used to store records or incoming data before transmitting or respectively receiving over a network. In Flink these are the ***flink.nw.buffers*** and represent logical connections between mappers and reducers. In Spark, they are called ***shuffle.file.buffers***. We enabled Spark's shuffle file consolidation property in order to improve filesystem performance for shuffles with large numbers of reduce tasks. The default buffer size is pre-configured in both frameworks to 32 KB, but it can be increased on systems with more memory, leading to less spilling to disk and better results. In all our experiments we initialize the Spark shuffle manager implementation to ***tungsten-sort***, a memory efficient sort-based shuffle. This is to provide a fair comparison to Flink, which is using a sort-based aggregation component.

Memory management. In Spark, all the memory of an executor is allocated to the Java heap (***spark.executor.memory***). Flink allows a hybrid approach, combining on- and off-heap memory allocations. When the ***flink.off-heap*** parameter is set to true, this hybrid memory management is enabled, allowing the task manager to allocate memory for sorting, hash tables and caching of intermediate results outside the Java heap. The total allocated memory is controlled in Flink by the ***flink.taskmanager.memory*** parameter, while the ***flink.taskmanager.memory.fraction*** indicates the portion used by the JVM heap. In Spark, the fractions of the JVM heap used for storage and shuffle are statically initialized by setting the ***spark.storage.fraction*** and ***spark.shuffle.fraction*** parameters to different values so that the computed RDDs can fit into memory and ensure enough shuffling space.

Data serialization. Flink peeks into the user data types (by means of the ***TypeInformation*** base class for type descriptors) and exploits this information for better internal serialization; hence, no configuration is needed. In Spark, the serialization (***spark.serializer***) is done by default using the Java approach but this can be changed to the ***Kryo*** serialization library [55], which can be more efficient, trading speed for CPU cycles.

3.2.3 Experimental setup

In order to understand the impact of the previous configurations on performance and to quantify the differences in the design choices, we devised the following approach. For both Flink and Spark we plot the execution plan with different parameter settings and correlate it with the resource utilisation. As far as performance is concerned, we focus on the end-to-end execution time, which we collect using both timers added to the frameworks source code and by parsing the available logs, and analyze the performance in the context of strong and weak scalability.

We deploy Flink (version 0.10.2) and Spark (version 1.5.3) on Grid'5000 [11], a large-scale versatile testbed, in a cluster with up to 100 nodes. Each node has 2 CPUs Intel Xeon E5-2630 v3 with 8 cores per CPU and 128 GB RAM. All experiments use a single disk drive with a capacity of 558 GB. The nodes are connected using a 10 Gbps ethernet.

For every experiment we follow the same cycle. We install Hadoop (HDFS version 2.7) and we configure a standalone setup of Flink and Spark. We import the analyzed dataset and we execute on average 5 runs for each experiment. For each run we measure the time necessary to finish the execution excluding the time to start and stop the cluster and at the end of each experiment we collect the logs that describe the results. We make sure to clear the OS buffer cache and temporary generated data or logs before a new execution starts. We plot the mean and standard deviation for aggregated values of all nodes for multiple trials of each experiment. We dissect the resource usage metrics (CPU, memory, disk I/O, disk utilization, network) in the operators plan execution.

3.3 Evaluation

3.3.1 Results

In this section we describe our experience with both frameworks and interpret the results taking into account the configuration settings and the tracked resource usage. For the batch workloads, our goal was to validate strong and weak scalability. For the iterative workloads, we focused on scalability, caching and pipelining performance.

Word count: evaluating the aggregation component

We first run the benchmark with a fixed problem size per node (Figure 3.7) with the parameter configuration detailed in Table 3.2 and then with a fixed number of nodes and increased datasets (Figure 3.8).

Number of nodes	2	4	8	16	32
spark.def.parallelism	192	384	768	1536	1024
flink.def.parallelism	32	64	128	256	512
spark.executor.memory (GB)	22	22	22	22	22
flink.taskmanager.memory (GB)	4	4	4	4	11

Table 3.2 – Word Count and Grep configuration settings for the fixed problem size per node (24 GB). Other parameters: ***HDFS.block.size*** = 256 MB, ***flink.nw.buffers*** = $Nodes \times 2048$, ***buffer.size*** = 64 KB.

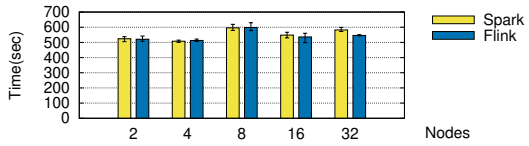

Figure 3.7 – Word Count - fixed problem size per node (24 GB).

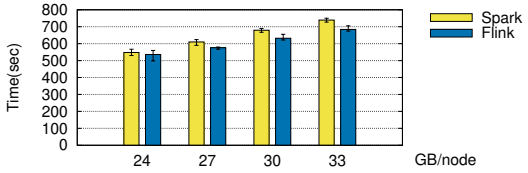

Figure 3.8 – Word Count - 16 nodes, different datasets.

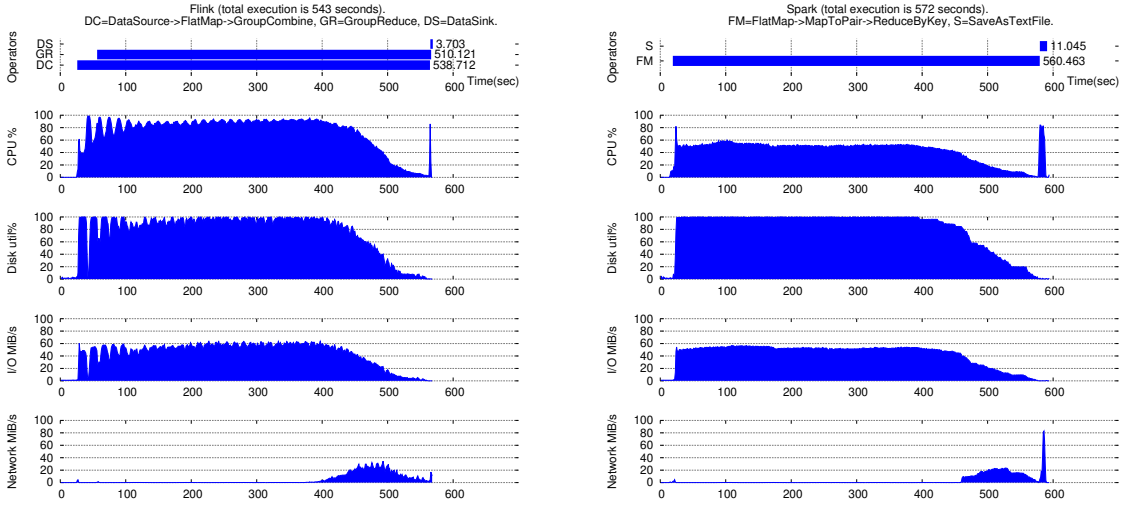

Figure 3.9 – Word Count operators and resource usage. Flink (left) versus Spark (right), 32 nodes and 768 GB dataset. Similar memory usage, growing linearly up to 30%.

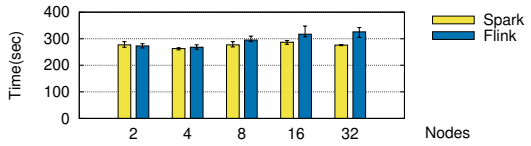

Figure 3.10 – Grep - fixed problem size per node (24 GB).

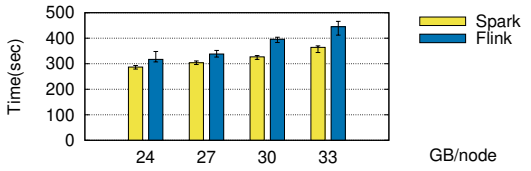

Figure 3.11 – Grep - 16 nodes, different datasets.

Weak Scalability. We observe in Figure 3.7 that both frameworks scale well when adding nodes, sharing a similar performance for a small number of nodes (2 to 8), but for a larger number (16, and 32), Flink performs slightly better. This happens even though, for fairness, we configured Spark with more memory because of its use of the Java serializer.

Strong Scalability. This observation is further confirmed for large datasets and a fixed number of nodes (Figure 3.8) with Flink constantly outperforming Spark by 10%. Spark's behaviour is influenced by the configured parallelism: the transformation ***reduceByKey*** that merges the values for each key, locally on each mapper before sending the results to the reducer, hash-partitions the output with the number of partitions (i.e., the default parallelism). In fact, for a similar cluster setup (8 nodes) we experimented with a decreased parallelism for Spark (double the number of cores) and obtained an execution time increased by 10%. Flink showed an improved execution when configured with 2 ***Task Slots*** for each available core. We further analyze the resource usage in order to understand this gap in performance.

Resource Usage. Figure 3.9 presents the correlation between the operators execution plan and the resource usage for 32 nodes. For this workload, both Flink and Spark are CPU and disk-bound. For Flink, we notice an anti-cyclic disk utilization (i.e., correlated to the CPU usage: the CPU increases to 100% while the disk goes down to 0%), which is explained by the use of a sort-based combiner for grouping, collecting records in a memory buffer and sorting the buffer when it is filled. CPU-wise, Flink seems more efficient than Spark and also takes less time to save the results with the corresponding action, contributing to the reduced end-to-end execution time. Flink is currently investigating the introduction of a hash-based strategy for the ***combine*** and ***reduce*** functions, that could yield further improvements (to overcome the anti-cycling effect).

Grep: evaluating text search

The next benchmark is evaluated in the same scenario: a fixed problem size per node (Figure 3.10) and a fixed number of nodes for increasing datasets (Figure 3.11), with the same parameters from Table 3.2.

Weak and Strong Scalability. When increasing the number of nodes, we notice an improved execution for Spark, with up to 20% smaller times for large datasets (16, and 32 nodes). Spark's advantage is preserved over larger datasets as well.

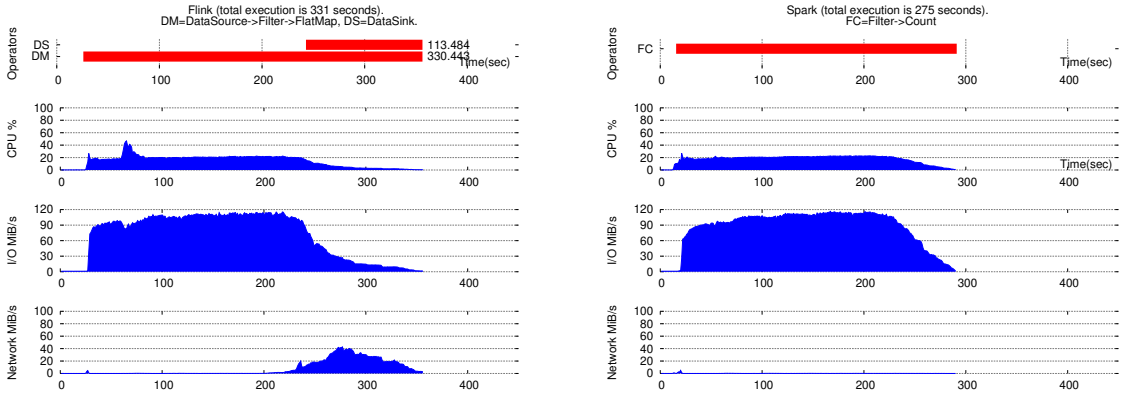

Figure 3.12 – Grep operators and resource usage. Flink (left) versus Spark (right), 32 nodes and 768 GB dataset. Similar memory usage, growing linearly up to 30%.

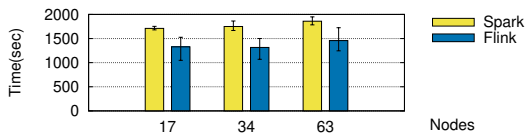

Figure 3.13 – Tera Sort - fixed problem size per node (32 GB).

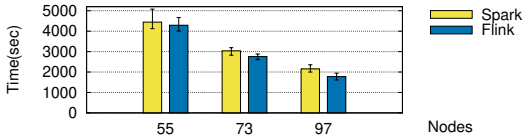

Figure 3.14 – Tera Sort: adding nodes, fixed dataset (3.5 TB).

Resource usage. In order to understand this performance gap, we zoom on the main differences in network and disk usage observed in Figure 3.12. Flink's current implementation of the ***filter*** $\rightarrow$ ***count*** operators is leading to inefficient use of the resources in the latter phase. While in the first stage, Spark's disk usage is almost similar compared to Flink's, the absence of network transfers during both phases (compared to the network usage during second stage in Flink) makes it compensate and further reduce the execution time.

Number of nodes	17	34	63	55	73	97
spark.def.parallelism	544	1088	1984	1760	2336	3104
flink.def.parallelism	134	270	500	475	580	750

Table 3.3 – Tera Sort configuration settings. Both Flink and Spark use 62 GB memory. The number of partitions is equal to the Flink parallelism number. Other parameters: ***HDFS.block.size*** = 1024 MB, ***flink.nw.buffers*** = *Nodes* $\times$ 1024, ***buffer.size*** = 128 KB.

Tera Sort: shuffle, caching, and the execution pipeline

We ran this benchmark with a fixed data size per node (32 GB) up to 64 nodes and then for a fixed dataset (3.5 TB) with up to 100 nodes (parameter settings in Table 3.3).

Weak Scalability. In Figure 3.13 we notice that although Flink is performing on average better than Spark, it also shows a high variance between each of the experiments results,

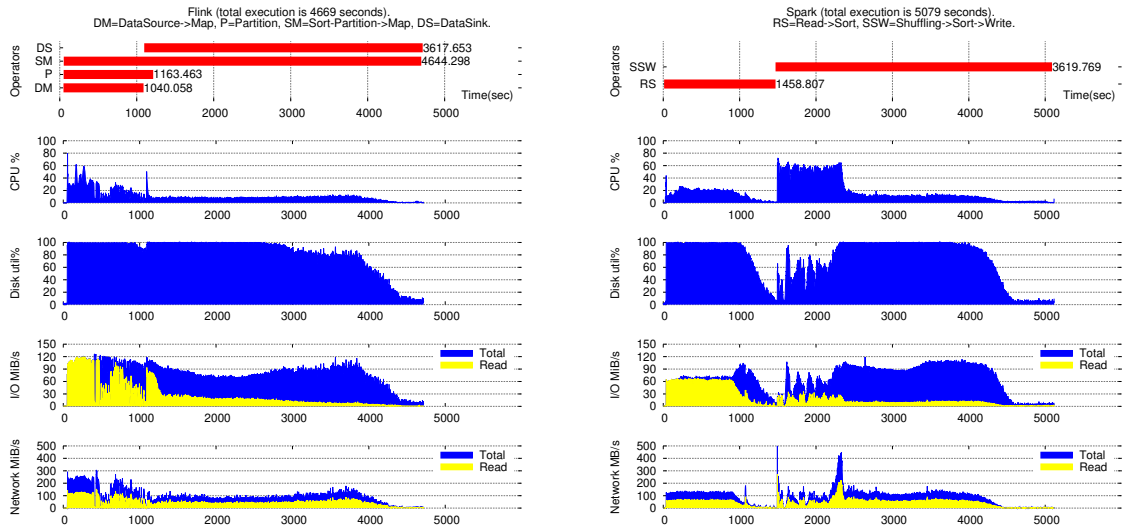

Figure 3.15 – Tera Sort operators and resource usage of Flink and Spark for 55 nodes and 3.5 TB dataset (similar memory usage).

when compared to Spark. This variance may be explained by the I/O interference in Flink's execution due to its pipeline nature. We wanted to check whether Flink's speedup is preserved for a larger dataset processed at each node and we experimented with sorting 75 GB per node using a cluster of 27 nodes (432 cores) and increasing the memory quota for both up to 102 GB). Again, Flink showed 15% smaller execution times.

Strong Scalability. As seen in Figure 3.14, Flink's advantage is increasing with larger clusters, which can be explained by less I/O interference caused by a reduced dataset to sort by each node.

Resource usage. Figure 3.15 presents the correlation between the operators execution plan and the resource usage for sorting 3.5 TB of data on a cluster with 55 nodes. Flink's and Spark's default parallelism settings were initialized to 1760 (twice the number of cores, following Spark's recommendation) and 475 respectively (half the number of cores in order to match the number of custom partitions, otherwise Flink fails due to insufficient task slots), and the number of custom partitions to 475 (i.e., Flink's parallelism, for a fair comparison). A few important observations differentiate Flink and Spark executions. First, Flink pipelines the execution, hence it is visualized in a single stage, while in Spark the separation between stages is very clear. Next, a virtual second stage is observable in both sides which is triggered by the separation of disk I/O read and write processes defined by one process domination along the time axis. Finally, Spark uses less network in this case due to the map output compression.

K-Means: bulk iterations

We evaluated the loop support in both platforms (Figure 3.16) for a fixed dataset of 51 GB (1.2 billion edges) needing 10 iterations to stabilise. While both Spark and Flink scale gracefully

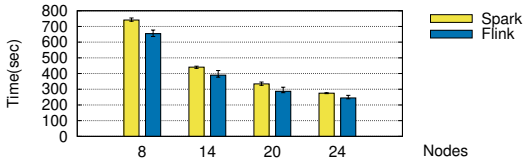

Figure 3.16 – K-Means: increasing cluster size, fixed dataset (1.2 billion edges).

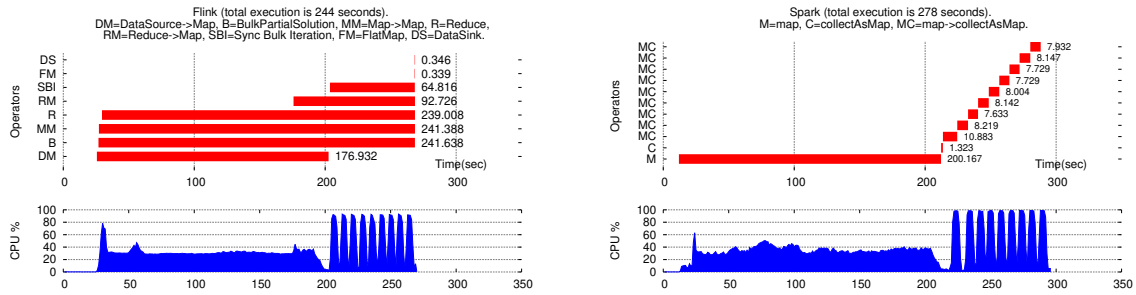

Figure 3.17 – K-Means operators and resource usage of Flink and Spark for 24 nodes, 10 iterations and 1.2 billion samples dataset. Resource usage elements not shown for similarity reasons: memory and disk utilization are less than 10%, total disk I/O (r+w) is less than 20 MB/s, and total network (r+w) is less than 30 MB/s.

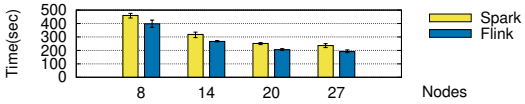

Figure 3.18 – Page Rank: Small Graph with increasing cluster size.

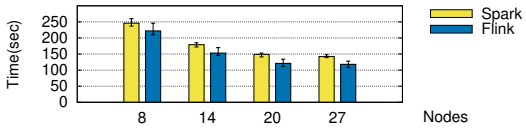

Figure 3.19 – Connected Components: Small Graph with increasing cluster size.

when adding nodes (up to 24), we notice that Flink's ***bulk iterate*** operator and its pipeline mechanism outperform by more than 10% the loop unrolling execution of iterations implemented in Spark. As seen in Figure 3.17, both frameworks have a similar resource usage, CPU-bound when loading the data points and processing the iterations.

Graph type	Small [97]	Medium [30]	Large [114]
Nodes / Edges	24.7 M / 0.8 B	65.6 M / 1.8 B	1.7 B / 64 B
Size	13.7 GB	30.1 GB	1.2 TB

Table 3.4 – Graph datasets characteristics.

Page Rank and Connected Components: graph processing

We have selected 3 representative graph datasets (Small [97] and Medium [30] social graphs from Twitter and Friendster respectively, and a Large [114] one, the largest hyperlink graph available to the public) as detailed in Table 3.4. We executed Page Rank and Connected Components with each graph type and specific parameter configuration, as discussed below.

Parameter	Formula
spark.def.parallelism	$[nodes \times cores \times 6]$
flink.def.parallelism	$[nodes \times cores]$
spark.edge.partition	$[nodes \times cores]$
flink.nw.buffers	$[cores \times cores \times nodes \times 16]$

Table 3.5 – Configuration settings for the Small Graph where nodes equals the total number of nodes and cores=16. Other parameters: HDFS.block.size = 128 MB, flink.taskmanager.memory.fraction = 0.8, spark.storage.fraction = 0.5, spark.shuffle.fraction = 0.3.

Nodes	24	27	34	55
spark.def.parallelism	1440	1620	1632	2640
flink.def.parallelism	288	297	442	715
spark.executor.memory(GB)	22	96	62	62
flink.taskmanager.memory(GB)	18	18	62	62
spark.edge.partition	1440	256	320	480

Table 3.6 – Configuration settings for the Medium Graph. Other parameters: HDFS.block.size = 128 MB, flink.nw.buffers = $Nodes \times 2048$, flink.taskmanager.memory.fraction = 0.8, buffer.size = 128 KB, spark.storage.fraction = 0.4, spark.shuffle.fraction = 0.4.

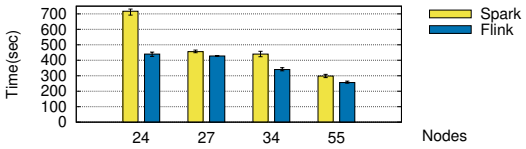

Figure 3.20 – Page Rank: Medium Graph with increasing cluster size.

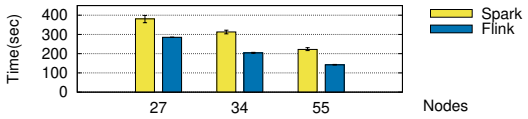

Figure 3.21 – Connected Components: Medium Graph with increasing cluster size.

Small Graph. With the settings detailed in Table 3.5, we notice a slightly better performance of Flink both for Page Rank (Figure 3.18) and Connected Components (Figure 3.19). For Page Rank this was rather surprising, considering that Flink's implementation will first execute a job to count the vertices, reading the dataset one more time in order to load the graph. We experimented with various values of the Spark's ***spark.edge.partition*** parameter and we obtained a drop in performance when this value is increased (more files to handle) or decreased (inefficient resource usage) for both algorithms. Flink's better performance is mainly backed by its bulk iteration operator and its pipeline

Nodes	27		44		97	
Large Graph	**Load**	**Iter.**	**Load**	**Iter.**	**Load**	**Iter.**
Flink PR	no	no	no	no	1096s	645s
Spark PR	3977s	no	667s	no	418s	596s
Flink CC	no	no	no	no	580s	1268s
Spark CC	3717s	3948s	798s	978s	357s	529s

Table 3.7 – Page Rank (PR) and Connected Components (CC) with 5 and 10 iterations (Iter.) respectively. Flink's load graph (Load) stage includes the vertices count.

nature.

Medium Graph. With the configuration in Table 3.6 and a larger graph we executed Page Rank (Figure 3.20) and Connected Components (Figure 3.21). For Flink we experimented with a decreased parallelism setting in order to test the pipeline execution implementation and we observed that during the iteration computation we can obtain a similar performance, but in the load graph phase (including vertices count for Page Rank) the performance drops due to inefficient resource usage. For Spark with 27 nodes and more we had to decrease the number of edge partitions because we experimented with larger values (proportional to the number of cores per number of nodes) for a configuration of 24 nodes and we found a large drop in performance (up to 50%). Flink's Connected Components outperforms Spark by a much larger factor than in the case of Small Graphs (up to 30%) mainly because of its efficient ***delta iteration*** operator.

Large Graph. We experimented with the large graph dataset on 3 clusters of 27, 44 and 97 nodes respectively, as shown in Table 3.7. Flink's execution with 27 and 44 nodes failed because of the ***CoGroup*** operator's internal implementation which computes the solution set in memory. For Spark's Page Rank and Connected Components with 27 and 44 nodes we were able to process correctly the graph load stage only when we doubled the number of edge partitions from a value equal to the total number of cores. The execution of Page Rank and Connected Components with 97 nodes was successful for both frameworks. Flink is less efficient because the parallelism is reduced. In Flink's case, we set the parallelism to three quarters of the total number of cores in order to allocate more memory to each ***CoGroup*** operator so that the execution does not crash. Setting the parallelism to the total number of cores causes a failure. We suspect two problems: one with the pipeline execution and the I/O interference and second with the inefficient management of a very large number of network buffers.

Resource Usage. As seen in Figure 3.22 we identified two processing ***stages*** for Page Rank: the first one is necessary to load the edges and prepare the graph representation, while the second one consists of the iterative processing. In the first stage, both Flink and Spark are CPU- and disk-bound, while in the second stage they are CPU- and network-bound. Spark is using disks during iterations in order to materialize intermediate ranks and we observe that the memory increases from one iteration to another, while for iterations in Flink there is no disk usage in case of Page Rank and some disk usage for the first iterations in the case of Connected Components, while the memory remains constant. Flink uses more network during the iterative process and, because of its pipeline mechanism and its bulk iterator operator, it is able to reduce the execution time. Overall, for Connected Components we observe a similar resource usage (Figure 3.23). However, although Flink's delta iterate operator makes a more efficient use of CPU, it is still forced to rely on disk for iterations on large graphs, hence Spark's better results.

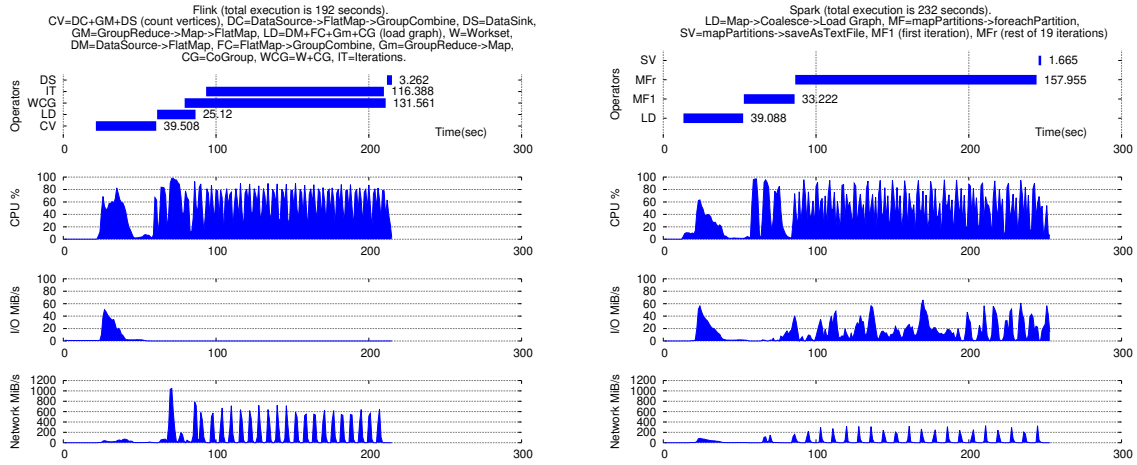

Figure 3.22 – Page Rank operators and resource usage of Flink and Spark for 27 nodes, 20 iterations, and small graph. Disk utilization is similar to disk I/O, memory is 40%.

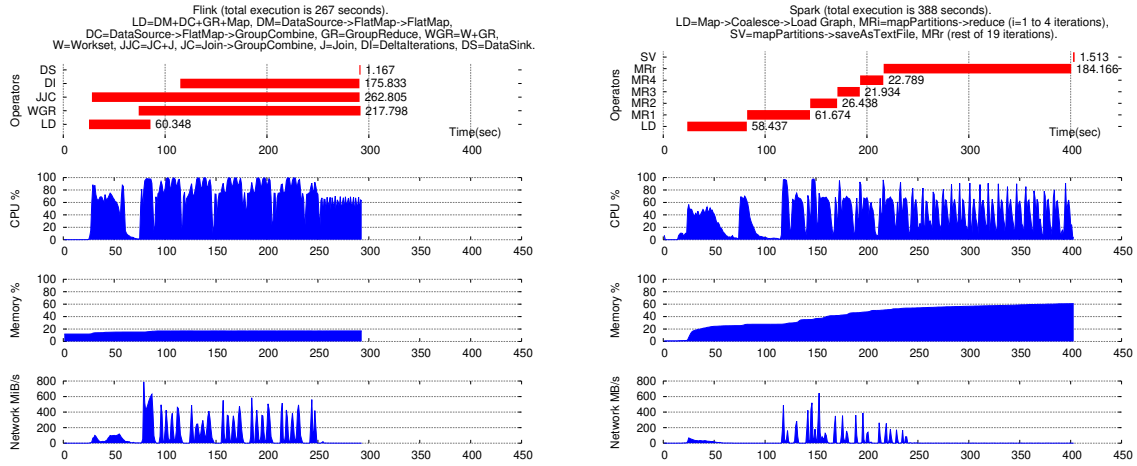

Figure 3.23 – Connected Components operators and resource usage of Flink and Spark for 27 nodes, 23 iterations, and medium graph.

3.3.2 Summary of insights

Our key finding shows that ***neither framework overperforms the other 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. This behaviour is explained by different design choices that we recall below.

Memory management plays a crucial role in the execution of a workload, particularly for datasets larger than available memory. For instance, Flink's aggregation component (sort-based combiner) appears more efficient than Spark's, building on its improved custom managed memory and its type oriented data serialization. With its ***DataSet*** API [44] for structured data, Spark aimed for a similar approach.

While common wisdom on processing lots of data in a JVM means storing them as objects on the heap, this approach has a few notable drawbacks, as also mentioned in [47]. First, as seen in the Large graph case from Section 3.3.1, memory overallocation will immediately destroy the JVM. Moreover, large sized JVMs (multiple GBs), overwhelmed with 1000s of new objects, can suffer from the overhead of garbage collection. Finally, on most JVM platforms, Java objects increase the space overhead. In order to avoid use-case specific tuning of the system parameters, necessary to optimize memory setups so that we avoid overallocation and garbage collection issues, analytics engines should architect with efficient memory as a first-class concern. During our experiments we noticed that, as opposed to Spark, Flink does not accumulate lots of objects on the heap but stores them in a dedicated off-heap memory region, to avoid memory issues. However, hybrid setups of memory (on- and off-heap) are difficult to tune and the memory fraction setting should (ideally) be automatically configured by the system and dynamically changed at runtime. In Flink, most of the operators are implemented so that they can survive with very little memory (by spilling to disk when necessary). We also observed that although Spark can serialize data to disk, it requires that (significant) parts of the data to be on the JVM's heap for several operations; if the size of the heap is not sufficient, the job dies. Recently, Spark has started to catch up on these memory issues with its Tungsten [90] project, highly inspired from the Flink model, for the explicit custom memory management aiming 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. For example, Flink's Tera Sort evaluation emphasizes the importance of the execution pipeline implemented by its smart optimizer (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 [27]. For instance, due to Flink's pipeline nature, we had to increase the number of buffers in order to avoid failed executions. In Flink there is a correlation between the number of network buffers, the parallelism and the workflow's operators. As such, users need to pay attention to the correct configuration (i.e., operators tasks configured with the required number of network buffers for communication between them).

Optimizations are automatically built in Flink for the whole application workflow, while Spark optimizes at the stage level (although for SQL jobs, SparkSQL [7] uses an optimizer that supports both rule-and cost-based optimizations). Spark batch and iterative jobs have to be manually optimized and adapted to specific datasets through fine-grained control of partitioning and caching. For grep-style workloads we have already seen Spark overperforming Flink due to poor implementation of the filter operator. For more complex workflows 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. This important feature is missing in the current implementation of Flink.

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 instance, for graph-based workloads Spark needs a careful parameter configuration (for parallelism, partitions, etc.), which is highly dependent on the dataset, in order to obtain an optimal performance. In Flink's case, one needs to make sure that enough memory is allocated so that its ***CoGroup*** operator that builds the solution set in memory could be successfully executed. Applications handling a solution set built with delta iterations should consider the development of a spillable hash table in order to avoid a crash, trading performance for fault tolerance. For such workloads, in which the execution consists 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. Backed by these experiments we can argue for the importance of the ***delta iterate*** operator feature and a pipeline execution which can bring improved performance.

3.4 Discussion

3.4.1 Related work

While extensive research efforts have been dedicated to optimize the execution of MapReduce based frameworks, there has been relatively little progress on identifying, analyzing and understanding the performance issues of more recent data analytics frameworks like Spark and Flink.

Execution optimization. Since the targeted applications are mostly data-intensive, a first approach to improving their performance is to make ***network optimizations***. In [118] the authors provide the best parameter combination (i.e., parallel stream, disk, and CPU numbers) in order to achieve the highest end-to-end throughput. ***Storage optimizations*** try either to better exploit disk locality [108] or simply to eliminate the costly disk accesses by complex in-memory caches [120, 64]. In both cases, the resulting aggregated uniform storage spaces will lag behind in widely distributed environments due to the huge access latencies. In [24] the authors analyze the changes needed by the optimizer and the execution engine of Flink in order to support bulk and incremental (delta) ***iterations***. Similarly to us, they consider graph processing algorithms like Page Rank when comparing to Spark, but the cluster size is small (hence no intuition about scalability) and they ignore recent improvements in Flink, like the memory management.

Performance evaluation. The vast majority of research in this field focuses on the Hadoop framework, since, for more than a decade, this has become the de-facto industry standard. The problem of how to predict completion time and optimal resource configuration for a MapReduce job was proposed in [15]. To this end, the work introduces a methodology that combines analytical modelling with micro-benchmarking to estimate the time-to-solution in a given configuration. The problem of disproportionately long-running tasks, also called stragglers, has received considerable attention, with many mitigation techniques being designed around ***speculative execution*** [4]. Other studies focus on the ***partitioning skew*** [60] which causes huge data transfers during

the shuffle phases, leading to significant unfairness between nodes. More recent performance studies specifically target Spark [94]. The authors analyze three major architectural components (shuffle, execution model and caching) in Hadoop and Spark. Similarly to us, they use a ***visual tool*** to correlate resource utilization with the task execution; however, they do not evaluate the operator parallelism and do not consider Flink with its own cost-based optimizer. ***Blocked time analysis*** has been introduced in [87] in order to understand the impact of disk and network and to identify the cause of stragglers. The authors show that in Spark SQL this is due to the Java Garbage Collector and the time to transfer data to and from the disk. This technique could be applied to Flink as well, where stragglers are caused by the I/O interference in the execution pipelines, as seen in the Tera Sort workload from our study.

Overall, most of the previous work typically focuses on some specific low-level issues of big data frameworks that are not necessarily well correlated with the higher level design. It is precisely this gap that we aim to address in this chapter by linking bottlenecks observed through parameter configuration and low level resource utilization with high-level behavior in order to better understand performance.

3.4.2 Fault tolerance trade-offs

MapReduce systems (e.g., Spark) adopt the bulk-synchronous model (BSP [110]) in which the computation consists of two phases that continuously repeat: a computation (map) phase during which all nodes of a distributed system perform some computation, followed by a blocking barrier that enables a communication (reduce) phase during which nodes communicate. To implement fault-tolerance, such systems implement a barrier snapshot: intermediate results are materialized (through checkpoints) or, more efficiently, the processing lineage is recorded (e.g., RDDs). However, since Big Data applications can be composed of many map and reduce phases, the user is forced to decide when is best to materialize intermediate results or simply to rely on lineage recovery techniques.

Dataflow systems (e.g., Flink) adopt the continuous (long running) operator model in which operators are scheduled once as long running tasks. While the execution model is more flexible compared to BSP, these systems rely on costly distributed checkpointing algorithms during normal execution; in order to handle node failures, all nodes are rolled back to the last available checkpoint and each continuous operator is recovered serially [111].

We argue that with proper support from a low-latency stream storage, continuous stream-based operators can additionally implement lineage techniques in order to optimize processing by, e.g., leveraging techniques such as parallel recovery [121]. In this case, each dataflow operator could asynchronously store a lineage stream of its task's computations along with offsets of input and output streams for each deployed task; the execution driver could then recover each operator task independently while back-pressure techniques could help maintain a consistent computation. With such a powerful feature, dataflow systems could become more attractive for both batch and streaming executions at very large scale where faults occur more often. For this to happen, a fundamental shift of approach is necessary: unbounded (stream-based) data processing engines should be designed to rely for state management on fine-grained, dynamic ingestion/storage systems as the one we propose in this thesis.

Chapter 4

Exploring shared state for window-based streaming analytics

Contents

While the previous chapter explores the architectural differences between batch-based and stream-based execution runtimes when running batch and iterative workloads characterized by offline data sources (i.e., previously stored in a distributed file system before computation), in this chapter we complement our previous study with understanding the performance of (window-based) streaming operators when the dataflow engine keeps the streaming state internally (on-heap) or within an external store (off-heap).

Stream-oriented engines [1, 67, 121] typically process live data sources (e.g., web services, social and news feeds, sensors, etc.) using stateful aggregations (called operators) defined by the application, which form a directed acyclic graph through which data flows. In this context, it is often the case that such stateful aggregations need to operate on the same data (e.g., top-K and bottom-K entries observed during the last hour in a stream of integers). Current state-of-the-art approaches create data copies that enable each operator to work in isolation, at the expense of increased memory utilization. However, with increasing number of cores and decreasing memory available per core [38], memory becomes a scarce resource and can potentially create efficiency bottlenecks (e.g., underutilized cores), extra cost (e.g., more expensive infrastructure) or even raise the question of feasibility (e.g., running out of memory). Thus, the problem of minimizing memory utilization without significant impact on performance (typically measured as result latency) is crucial.

In this chapter, we explore the feasibility of deduplication techniques in order to address this challenge. What makes this context particularly difficult is the complex interaction and concurrency introduced by the operators as they compete for the same data, which is not originally present in the case when operators work in isolation. We design a deduplication method specifically for window-based operators that relies on key-value stores to hold a shared state and we experiment with a synthetically generated workload while considering several deduplication scenarios. Our goal is to understand what are the current limitations faced by a streaming engine when interacting with a storage engine for holding streaming state. The main intuition is the following: if the shared state approach is generally feasible with current state-of-the-art streaming engines, our stream storage design will have to integrate the design principles lying at the core of this work. Otherwise, a new approach is needed to alleviate from identified limitations.

4.1 Background

This section discusses the general context of this work, targeted use cases and the main working assumptions. Based on this we introduce the problem statement.

4.1.1 Context

Stream processing window functions such as aggregates and UDF (user defined functions, i.e., ***patterns***) are more challenging than typical streaming patterns (e.g., filtering, projecting, data structure or event enhancements) as they pre-require buffering the data over some periods of times (i.e., these functions are typically applied over the window contents). The functions that are applied are quite generic and range from mathematical functions (e.g., computing statistics, histograms) to extracting data features for machine learning or for business intelligence (e.g., min, max, summations, metrics over partitions) to binary or multivariate functions (e.g., labeling items as relevant or irrelevant in a specific context). To exemplify, one can consider the example of gaming specific scenarios [28], which puts in evidence Terabytes of state generated by billions of events per day. The processing focuses on computing revenue streams in real time (e.g., summations – total revenue, metrics over partitions – computing average revenues per country) and on determining user activities (i.e., labeling functions – which levels make user quit; histograms – hourly activities for games), etc.

Listing 4.1 – Two patterns on a shared input data stream

```
DataStream<EventType> input = env.readParseSource(params);

DataStream<ResultType1> patternOne = input
.keyBy(<first key selector>)
.window(<first window assigner>)
.<window transformation>(<first window function>);

DataStream<ResultType2> patternTwo = input
.keyBy(<second key selector>)
.window(<second window assigner>)
.<window transformation>(<second window function>);
```

Multi-Patterns. We consider the general case of applying such aggregations and UDFs (two or more patterns) over partial or full common stream data, and without focusing on a particular domain. We consider how the underlying stream operator (i.e., the window) can better support these concurrent analysis and make resource usage more efficient (e.g., decrease memory footprint) without leveraging properties (e.g., associativity) of the patterns' functions that are applied. This raises additional challenges with the cases where no specific assumptions can be made, other than the ones that are generally considered by the stream paradigm; on the other hand the approaches considered need to be transparently encapsulated within the stream framework without altering the stream paradigm or the API semantics.

4.1.2 Problem statement

We define the working scenario as follows: we have a rate of new events (typically a few thousand events per second – half a billion events per day). This is a general assumption on the event workload that applies across the aforementioned domains: IoT, banks, gaming companies, e-commerce sites have events in the range of million to tens of millions per day (e.g., a large game company will have about 30 million events per day). We consider analysis history up to 12 months of historical events. This can cover analysis from instant metrics to complex machine learning algorithms that aim to learn user behavior, which require large time-spans. In terms of domain parallelism, we build millions of windows (each event can be associated to one or multiple windows) that we keep as state in memory in order to process ***multiple patterns*** (that correspond to window-based UDF or aggregations). The choice for this granularity is motivated by the fact that banking or ecommerce have millions of users. Furthermore, the specific analysis can require various partitions (e.g., computing averages per user, per country or per currency) which drive the need to associate each event with multiple windows to support the corresponding processing. Each event value size is ***significant*** (hundreds of bytes) and corresponds to multiple attributes that are possibly used in each pattern's computations. The arity of the tuples can range from tens (e.g., data specific to financial markets) to hundreds of attributes (data in e-commerce is large and augmented with metadata from various cookies).

The computation will thus contain multiple window processing operators (N) running concurrently within the stream engine, in order to process windows built from the same input of infinite events. In Listing 4.1 (Flink's API) we give an example of building a topology

with two patterns running window functions on the same data stream: after creating an input ***DataStream*** by parsing events from one source (***readParseSource***), we subsequently define two patterns as window operators. Current implementations are based on duplicating stream events in memory, leading to inefficient memory usage and potentially increased processing event latency. Consequently, the memory footprint is equal to the sum of the states of all processed windows. One can imagine that if the number of pattern analysis that run in parallel grows, we can end up with a several ten-folded multiplication factors over the entire data. e.g., consider the implementation of fraud detection, e-banking and other historical/analytics services over card transaction data that span over one year for hundreds of millions of users. If we consider on average 1000 transactions per second each of 1 KB, then in one year we have 29 TB of state to store in memory without considering additional copies.

Our goal is to explore the possibility to store the shared state in an external key-value store in order to efficiently deduplicate memory corresponding to events that are common to multiple (overlapping) window-based operators.

4.2 Memory deduplication with shared state backend

In this section, we briefly introduce the concept of stateful window-based stream processing and propose a deduplication approach specifically designed for this context.

4.2.1 Stateful window-based processing

At its basis, an infinite data stream is a set of events or tuples that grows indefinitely in time [69]. An infinite data stream is divided (based on event timestamp or other attributes) into finite slices called windows [29]. The properties of a window are determined by a window assigner: it specifies how the elements of the stream are divided into windows. The main categories are:

Global windows: each element is assigned to one single per-key global window;

Tumbling windows: elements are assigned to fixed length, non-overlapping windows of a specified window size;

Sliding windows: elements are assigned to overlapping windows of fixed length equal to the window size, the size of the overlap is defined by the window slide; and

Session windows: windows are defined by features of the data themselves and window boundaries are adjusting to incoming data.

Stateful operators implemented as (sliding) window-based aggregations are working over a state that defines the confines of the (sliding) window. The window state is a set of M recent tuples and is usually persisted as a list structure in heap memory or off-heap embedded key-value store. The implementation can also be hybrid, as we propose, with references (hash over tuple key/value) of tuples stored in heap memory and actual values stored in an external key-value store.

To build and modify a window state, the (evicting) window operator is using a ***ListState*** interface that gives access to various methods to add a tuple to the state, remove a

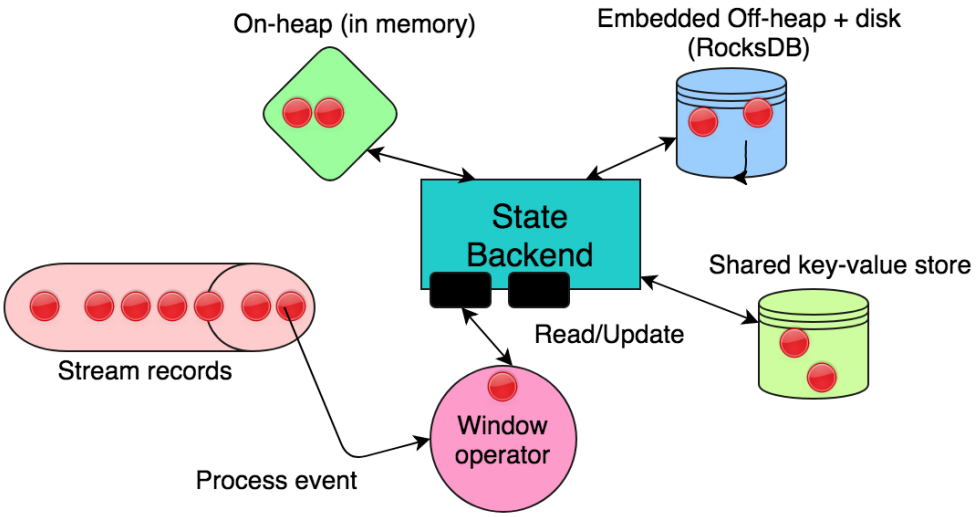

Figure 4.1 – State backend options for window-based streaming operations.

tuple from the state or retrieve all the tuples of the state. ***ListState*** methods can be defined for both generic tuples and serialized (byte array) ones, depending on the method used to persist state in memory. For instance, storing tuples in a serialized format helps reduce the memory footprint, at the cost of increased CPU usage. The window state backend abstraction is hidden from the developer, but can be parametrized in order to use different implementations.

State backend options for window-based processing. Apache Flink gives two ways of storing state for window-based operators (see Figure 4.1):

In-heap (memory or file) state backend: it stores its data in heap memory with/out capabilities to spill to disk (backed by a file system);

Off-heap embedded (key-value store RocksDB) state backend: it stores its data in RocksDB with capabilities to spill to disk.

We choose Apache Flink [67] to develop a proof of concept of our techniques, as it is today the most advanced open-source streaming engine. Flink adopts most of the Dataflow window model as described in [66], being the state-of-the-art for windowing semantics. We further discuss the buffering options we can consider for window state backends. While some of the options are currently implemented (heap and RocksDB), some other states are proposed by us and used in our evaluation (heap+Redis, RocksDB+Redis).

Object state in JVM heap memory. By default Flink stores data internally as objects on the Java heap in a memory state backend which has strong limitations: (1) the size of each individual state is limited to a few Megabytes (for in-heap memory state); (2) the aggregate state must fit into the configured job heap memory.

In our window-based scenarios (i.e., jobs with large state, many large windows) we are required to save each window operator's instance states on the local task manager

heap memory. For this situation we can configure Flink to a file state backend, which is characterized by holding data in the task manager heap memory and further checkpointing the state into a file system (e.g., HDFS) in order to ensure consistency guarantees. To configure this state we have to initialize two parameters: 1) ***state.backend*** to value ***filesystem*** and 2) ***state.backend.fs.checkpointdir*** to the HDFS path for checkpointing state. Each operator window's state is a list of Java objects and it is updated every time a new element arrives.

Serialized objects state in off-heap memory. Similar to the heap object state, Flink offers an option to configure an operator state to off-heap and it implements an embedded key-value store state interface (i.e., RocksDB). The main difference is that objects are serialized before they are persisted in the off-heap state and every time objects are accessed the cost of deserialization adds to the processing latency of corresponding operator's user defined function. Another difference consists in the fact that the RocksDB database is using local task manager data directories and, as such, the state size is limited by the amount of disk space available.

4.2.2 Deduplication proposal

Before we try to find an efficient way of reducing the pressure on memory for persisting window states, it is important to understand what properties of user-defined functions can enable a reduction of the state and thus reduced memory utilization.

As discussed in [102], if the aggregation function is associative (not necessary to be commutative or invertible), then a general ***incremental*** approach could possibly avoid buffering window states. It can help to achieve much better event latency for large windows, while the memory footprint for storing partial aggregates is much lower than in the case of storing entire windows. For small windows, it provides almost the same event latency. However, in some cases, there is a need to access the elements of a window after the aggregation was executed, so although incremental aggregation can be efficient, a window state may still be necessary. If we consider that not all the aggregation functions are associative, then we are forced to re-aggregate from scratch for each window update.

Existing approaches do not consider sharing a window's state elements. The analyzed framework (Apache Flink) is caching buffers in either JVM heap (leading to increased memory footprint because of Java representation overhead) or to an embedded key-value store (RocksDB), possibly wasting memory resources because of duplicated stream events. As such, our approach is worth being explored in order to respond to critical situations where memory usage needs to be reduced.

Next, we describe the implementation of the proposed shared-state backend and we detail the necessary enhancements added to Flink's interfaces.

Memory deduplication with shared key-value store (illustrated in Figure 4.2). Our approach for window states memory deduplication is based on the following: for each element (event value) of a stream we calculate and associate a key (event reference). Each window's buffer is defined as a list of references to the assigned events as follows: ***WindowKey*** $\rightarrow$ ***ListStruct***$<$***EventReference***$>$

Based on the properties of the windows (how elements are arriving, ordering, eviction policies), ***ListStruct*** may be implemented as a simple list or as a more complex structure.

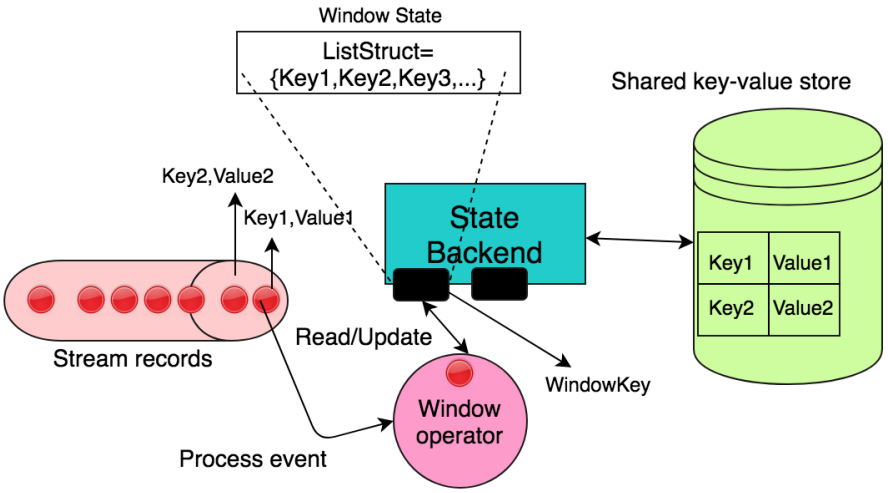

Figure 4.2 – Deduplication proposal for window-based streaming operators through shared key-value store.

Each event value with its associated reference (i.e., event key) will be stored once in a key-value store and accessed every time a window aggregation is activated.

For each new object that is assigned to an operator's window, we calculate a key by hashing the value of the event. We implement a new interface ***SharedListState*** that is configurable by setting the parameter ***state.backend*** to ***sharedfilesystem***. When we add a value to the shared state we make the following operations: 1) we append the reference key to a list and we store this list in JVM heap memory; 2) we store the <***key, serialized value***> pair in the external key-value store. When an operator's window execution is triggered because a new event arrived, we retrieve the list of keys from heap in order to make a call (multi-get) to the external key-value store in order to obtain all the serialized values. We subsequently deserialize each value and further trigger the user defined function that computes the window aggregation. Our approach is not only effective for memory deduplication, but will also be useful for moving computation to other nodes (separating state from the streaming execution) if we consider that our key-value store is configured to replicate its data.

4.3 Synthetic evaluation

This section describes the experimental setup, methodology and results.

4.3.1 Setup and Methodology

We implemented an event generator that is capable of streaming events through a socket. As a motivating scenario, the event generator is designed to emulate user transactions in a banking system, which are used in a fraud detection scenario. Specifically, the user transactions are strings (events) composed of relevant attributes (type of transaction, date, merchant name, value of transaction, type of card, name of customer). Their content is generated randomly, according to the following distribution (in order to draw one real scenario where

events arrive uniformly): an equal number of twelve events in a number of steps proportional to 1000 milliseconds (e.g., 60 events are streamed as 12 events every 200 milliseconds). The user transactions are consumed by a Flink application that operates on some of the parameters through a user-defined aggregation operator. The operators we implemented perform two low-CPU metrics (sum, min).

For every experiment we follow a similar cycle. We installed Redis 3.2.4 [93] and we configured a standalone Flink (version 1.1 modified with our shared state backend approach) to use it as the state backend, on a single node which has an Intel Xeon CPU E5-2630 v3 @ 2.40GHZ X 16, Ubuntu 16.04 LTS 64-bit, 31 GB RAM and 512 GB disk. We start the event generator as a Java socket program that listens to a configured port. First, it generates strings (events) until they fill the window state. Then, it generates strings for five iterations, each of one minute, keeping the same rate of new events. At the same time, the Flink application uses the ***socketTextStream*** method of the ***StreamExecutionEnvironment*** in order to create a new data stream that contains the strings received from the configured socket. We parse each event with a ***flatMap*** operation, assigning it a timestamp. After applying a user-defined function (as mentioned in the previous section), the timestamp is used to obtain the event latency, defined as the time seen at the end of the aggregation minus the initial time of the last event of a window. We collect each aggregated value and write it as text with the operator ***writeAsText***.

We measure only the event latencies corresponding to the five iterations and we compute the aggregated upper bound of latencies experienced by 99% of events. We make sure to clear the OS buffer cache and temporary data or logs before a new execution starts. After each execution ends we clear the Redis cache and we collect logs that hold the event latency percentiles.

For each experiment we fix the values for the following parameters (while keeping all other parameters at their default): (1) ***window_size*** – is the size of the window for which we execute an user defined function aggregation, window slide is fixed to 1 (in order to put pressure on window evaluation); (2) ***window_keys*** – gives the number of windows that are processed in parallel, equals the number of cores; (3) ***events_rate*** – is the rate of new events that are streamed by the socket program each second according to the distribution mentioned in the previous section. Other parameters: (4) the size of event reference (key) is 16 bytes (to avoid colisions for one year worth of data); (5) the size of each event value is 100 bytes (estimated size for a typical user transaction).

4.3.2 Results

Impact of heap size on event latency

Our first series of experiments aim at understanding the overhead of operating under low memory constraints, which leads to frequent invocation of the garbage collector and thus decreased performance. To this end, we use a variable heap memory size for the task manager, while keeping the other parameters constant. We choose to evaluate a low-CPU aggregation operator over a sliding window of size ***10240*** (slide equal one) with the rate of new events set at ***480*** events per second. The number of processed windows is two and we only use one task slot (parallelism one). The window state is configured to use the heap.

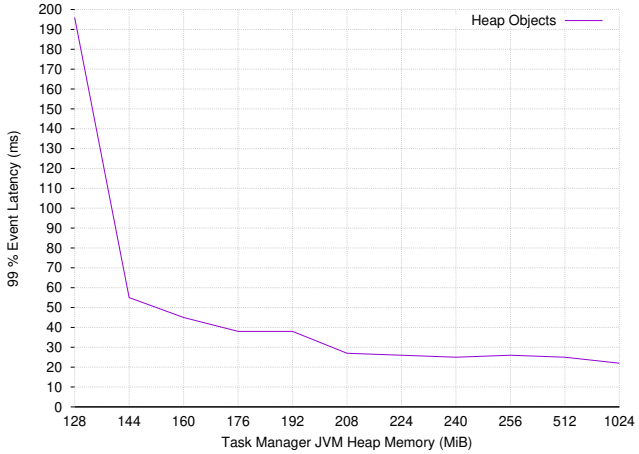

Figure 4.3 – Event processing latency (99% percentile) for fixed window size and event rate when varying the Task Manager Heap Size.

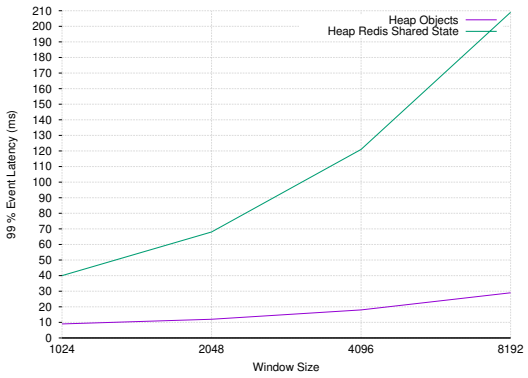

Figure 4.4 – Event processing latency (99% percentile) for fixed event rate when varying the window size (heap versus shared). Rate of new events is 60. Heap size is 1 GB. Parallelism is one: all events correspond to the same window.

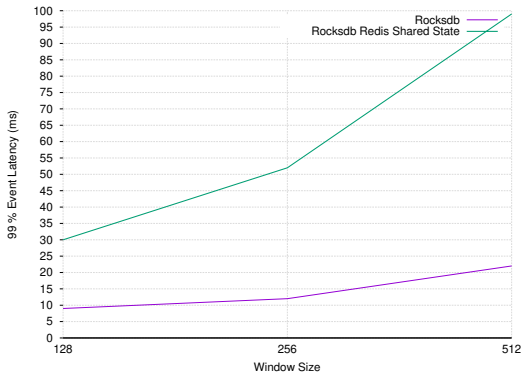

Figure 4.5 – Event processing latency (99% percentile) for fixed event rate when varying the window size (off-heap versus shared). Rate of new events is 36. Heap size is 1 GB. Parallelism is one: all events correspond to the same window. Heap event latency is 5 milliseconds.

We observe that the CPU usage decreases as a consequence of a reduced garbage collector overhead, which leads to a decrease in measured event latency of up to ten times (as observed in Figure 4.3). This emphasizes the importance of avoiding running stream processing operators under low memory constraints.

Impact of window size on event latency

Next, we evaluate the impact of the window size on the perceived event latency, which is the main indicator of performance in a stream processing application.

In Figure 4.4 we observe that with larger windows the effect of queueing on the event latency increases. Specifically, the event latency breaks down as follows: event queueing

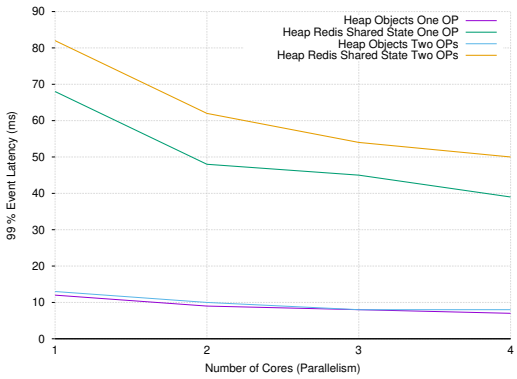

Figure 4.6 – Event processing latency (99% percentile) for fixed event rate when varying parallelism. Rate of new events is 60 per core every second. Heap size is 1GB. Window size is 2048.

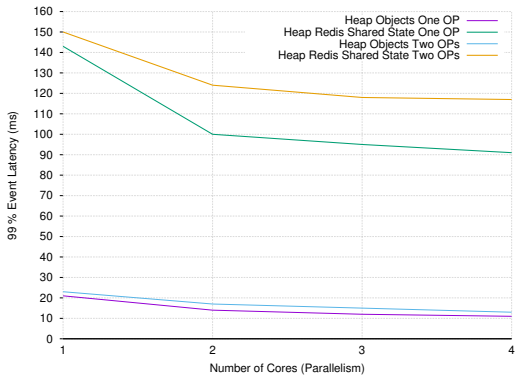

Figure 4.7 – Event processing latency (99% percentile) for fixed event rate when varying parallelism. Rate of new events is 120 every second. Heap size is 1GB. Window size is 4096.

(how much time an event is buffered before it got its chance to be processed), overhead to add an event to state, overhead to retrieve the whole window from the state, time to process aggregation and framework overhead.

We also evaluate the off-heap object serialization option using ***RocksDB*** as the state backend (assuming default configuration in Flink of the embedded key-value store) for the same rate of 60 new events per second. We do not plot these numbers as they show much higher latencies: 111 milliseconds for windows of size 1024, 310 milliseconds for windows of size 2048 and hundreds of seconds for a window of 4096 events.

To facilitate a feasible comparison between "Redis-dedup" using RocksDB and plain RocksDB, we decreased the rate of new events to 36 and we also reduced the window size. The results in Figure 4.5 show that although the size of the windows is much smaller compared with the previous experiment, the overhead of using RocksDB as a state backend is much higher. Nevertheless, the effect of event queueing follows a similar trend.

One versus two operators using shared events

The next experiment evaluates the event processing latency when increasing the number of operators sharing the same events from one to two. To this end, we fix the problem size per core as follows: for each core we generate 12 events every 200 milliseconds, considering that events are all part of the same window. We evaluate up to four windows corresponding to the same parallelism of each operator.

In Figure 4.6 we plot four benchmarks: two correspond to the application of one operator having state in heap (Heap Objects One OP) or sharing state in Redis (Heap Redis Shared State One OP) and the other two correspond to the application of two operators having state in heap (Heap Objects Two OPs) or sharing state in Redis (Heap Redis Shared State Two OPs). While for the heap state we observe similar latencies, for the shared state we observe an almost constant gap between each execution. This gap is related to the increased pressure on the key-value store, trying to access the same data concurrently, in addition to Java serialization costs.

As we can see in Figure 4.7, this experiment evaluates the same problem (events rate of 120 per second, window size of 4096 events) while increasing the parallelism from one to four (each core will process an equal number of elements) for the same four benchmarks like in the previous section. It is clear that with larger windows and increased throughput, relying on an external key-value store for memory deduplication of window-based processing may be unfeasible due to large overheads of Java serialization.

Scenario	Experimental Use Case	Large Use Case
Heap Only	800 KB	4.7 TB
Deduplicated	592 KB	1.3 TB

Table 4.1 – Estimating the memory utilization for the use case presented in the previous section (N is 2, M is 4096) and a potential large use case at scale (N is 10, M is one month worth of 2000 events per second) with and without deduplication.

4.3.3 Memory savings

Although there is significant performance overhead when using deduplication (as detailed in the previous section), such a technique may lead to large memory savings. While we did not measure the memory utilization directly (which is inherently difficult due to garbage collection), we estimate it as $M * N * EventSize$ for the heap-only case and, respectively $M * EventSize + (N + 1) * M * KeySize$ for the deduplicated case. In these formulas, $\boldsymbol{N}$ is the total number of concurrent operators, $\boldsymbol{M}$ is the total number of events, $EventSize$ is fixed at 100 bytes, $KeySize$ is fixed at 16 bytes. All operators are assumed to consume the same events (i.e., full window overlap).

The results of our estimation, both for the use case from our experiments as well as for a larger hypothetical use case at scale are summarized in Table 4.1. For the case where only two operators share common data, it can be observed that deduplication leads to a consistent memory saving of 26% compared with the heap-only case, while for a large scale use case with ten operations sharing data we estimate that deduplication leads to higher memory savings of up to 72% compared with the heap-only case. However, using a shared state approach can possibly lead to increased event processing latency, making difficult to rely on external storage for keeping the streaming state.

4.3.4 Summary of insights

Based on this study, we draw three conclusions. First, under low memory constraints, window-based operators tend to perform poorly due to frequent invocations of the garbage collector. In this case, the latency needed to process 99% of the events is up to 10x higher compared for the case when there is no memory pressure. Thus, deduplication has potential to improve latency. Second, deduplication leads to higher performance degradation for an increasing window size compared to the case when copies are used. Thus, careful selection of the window size is needed. Third, deduplication has a large potential to save memory: already for two operators that share the same state there is a 25% reduction, which continues to grow proportionally with the number of operators sharing the same state.

4.4 Discussion

4.4.1 Comparison with existing approaches

Deduplication is a common technique used in a variety of scenarios, both obvious (e.g., saving space in file systems [122, 23] or reducing the size of large scale memory dumps [82]) and less obvious (e.g., detection of natural replicas to reduce the cost of replication-based resilience [81]).

In the context of stream computing, recent research efforts have concentrated on the problem of sharing the state for overlapping sliding windows over event streams. However, as described below, they focus on very specific issues, which they alleviate in isolation, in most cases trading performance for expressivity.

Exploiting data redundancy. In [72], the authors introduce a buffer management algorithm that exploits the access pattern of sliding windows in order to efficiently handle memory shortages. The idea is that sliding-window operators are most of the time manipulating only a small fraction of their data set and are doing so in a very predictable pattern: once a tuple is stored on the window, it is not going to be accessed by the sliding-window operator until it is time to expire it. Hence, they implement a shared operator working over multiple windows, consisting of tuples in a shared tuple repository.

However, they only consider time-based windows and it is not clear how tuples are referenced from a window. Also, the total space of the shared repository is the largest window. This means that the algorithm only works for the particular case of windows that overlap in a deterministic way (with a coarse granularity of one hour), all included in a larger, containing window. In contrast, our approach also supports ***fragmented intersections of windows*** and that with a ***finer granularity*** (i.e., an event).

Tuples referencing. The idea of using pointers to the data tuples was introduced in the Continuous Query Language (CQL) [5], an SQL-based declarative language for registering continuous queries against streams and updatable relations. In CQL, windows are implemented by non-shared arrays of pointers to shared data items, so that a single data item might be pointed to from multiple windows. To minimize copying and proliferation of tuples, all tuple data is stored in synopses (i.e., in-memory hash tables) and is not replicated. Synopses are used at runtime to compile the query plans, which are merged whenever possible, in order to share computation and state.

Similarly to us, CQL uses the same idea of pointers to shared data items, yet the effects of serialization/deserialization as well as the heap buffer limitations are not assessed. Deduplication only works as a side-effect of merging various query plans. Queues contain references to tuple data within synopses, along with tags containing a timestamp and an insertion/deletion indicator. Our contribution is the ***generalization of synopses into shared state*** as key-value stores.

Incremental event processing. Sliding-window aggregation is a key operation in stream processing and incremental aggregations help to avoid re-aggregating from scratch after each window change. In order to exploit this incremental processing, the targeted functions need all to be associative (e.g., count, sum, max, min, mean, etc.). Without

the associativity, one can only handle insertions one element at a time at the end of the window. Hence, associativity enables breaking down the computation in flexible ways. Reactive Aggregator [102] is an example of such a framework for incremental sliding-window aggregation. It stays competitive (10% higher throughput than from-scratch re-computation) on small windows (1 to 100 events), but its true performance is shown for large windows (thousands of events), when it delivers at least one order of magnitude higher throughput compared to re-executions. Orthogonal to Reactive Aggregator is Cutty [13], a project that is considering more general non-periodic windows (punctuations and sessions, custom deterministic windows) which are expressed as user-defined operators. Cutty relies on a technique to discretize a stream into a minimal set of slices for efficient aggregate sharing of user-defined windows. Shared data fragments [54] focus on various techniques for aggregation sharing without the need of optimizing queries upfront. Our work is orthogonal to these solutions, considering the more general user defined functions that are not associative.

Aggregate computation optimizations. In [65] the authors introduce the panes technique to evaluate sliding-window queries by pre-/sub-aggregating and sharing computations. The problem of efficiently computing a large number of sliding-window aggregates over continuous data streams was introduced for the first time in [6]. The authors put forward three cost parameters that need to be considered: a) memory required to maintain state (space); b) the time to compute an answer (lookup time) is proportional to the window size; c) the time to update the window state when a new tuple arrives (update time) is composed of the time to evict old tuples as well. While their focus is on pre-aggregating values that are eventually shared, our approach maintains full window-state in memory. As opposed to the case of associative functions, for which incremental processing was a good option, this is no longer the case in the general (non-associative) scenario. In this context, our aggregation functions are able to reconsider all tuples of a window for each evaluation and to gracefully compute from scratch.

4.4.2 Pushing processing to storage

Streaming runtimes develop complex mechanisms to manage internally the windowing state. As can be seen from previous experiments, it can be difficult to rely on an external storage for keeping the streaming state. However, for fast crash recovery and fast adaptability this state should also be kept (and incrementally updated) inside a distributed storage engine (operations realized through incremental checkpoints). This complexity could be avoided at the processing level if the ***required support for keeping the windowing state was developed within a unified ingestion/storage layer***. With such support, pushing user-defined aggregate functions to storage (in order to avoid moving large amounts of data over the network, and also to avoid de/serialization overheads) could help to reduce the latency of window-based operations and further increase processing throughput (see also [58]).

One possible solution is to rely on highly specialized platforms for in-memory data like Apache Arrow [8]. Apache Arrow is a columnar memory format for efficient analytics operations on modern memory hardware, providing zero-copy streaming messaging and inter-process communication. Apache Arrow can be backed by Apache Parquet [88], a columnar

storage format built to support efficient compression and encoding schemes. Apache Arrow provides high-performance interfaces and could be natively integrated with our unified storage and ingestion architecture. To enable "in-storage" processing, a shift in the overall mindset is necessary: processing engines should focus on how to transform data and avoid complicated mechanisms to ***handle state*** at this level (as currently done). We argue it is more efficient to leave this function to a specialized stream storage engine such as the one we propose.

Part III

KerA: a unified architecture for stream ingestion and storage

Chapter 5

Design principles for scalable data ingestion and storage

Contents

Based on previous ingestion and storage requirements (Chapter 2) and considering our experience with current Big Data processing architectures (Chapters 3 and 4), we argue that a fundamental shift is necessary in Big Data analytics: the state management function (for input un/bounded streams and intermediate operators execution) should be handled

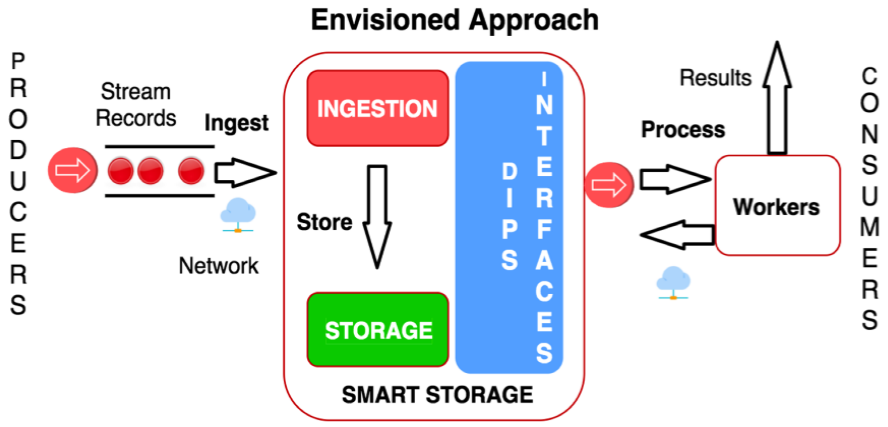

Figure 5.1 – Data first envisioned approach: unified ingestion/storage engine with data ingestion, processing, and storage (DIPS) stream-based interfaces.

by a dynamic ingestion/storage engine, while a dataflow system should focus on how to transform data (through stream-based interfaces that subsume the standard batch models) and avoid complicated mechanisms to handle state at its level. Therefore, we propose a set of design principles for a scalable, unified architecture for ingestion and storage that can optimize (un/bounded) Big Data processing.

5.1 Data first: towards a unified analytics architecture

We first discuss a set of design principles for streaming that we envision future Big Data analytics architectures will rely on (we present our approach in Figure 5.1).

5.1.1 Processing engines should focus on the operators workflow

Stream processing engines should model the computation. They should focus on how to transform data through a workflow composed of stateful and/or stateless stream-based operators. The main focus should be the computation: how to define the computation, when to trigger the computation and how to combine the computation with offline analytics. Processing engines should offer the necessary APIs and semantics for user-defined computation flows and optimizations of the execution flow. Processing engines could follow a dataflow graph-like execution, a natural choice for handling streams of records (subsuming batch execution) and should avoid complicated mechanisms to store (large) processing state at this level. We argue it is more efficient to leave the state management function to a specialized unified ingestion and storage system for un/bounded data (as previously discussed in Chapter 3 subsection 3.4.2 and Chapter 4 subsection 4.4.2).

5.1.2 Ingestion and storage should be unified and should focus on high-level data management

Both ingestion and storage are exposed through a common engine that is capable of leveraging synergies to avoid I/O redundancy and I/O interference arising when using independent solutions for the two aspects. This engine handles caching hierarchy, de-duplication, concurrency control, etc. Furthermore, all high-level data management currently implemented in the processing engine (fault-tolerance, persistence of operator states, etc.) should be handled natively by the unified layer. Storage systems should represent data state through an interface capable of fast ingestion of streams of records, storing data and providing efficient access for streams and objects data to the processing engines.

5.1.3 Processing engines and ingestion/storage systems should interact through stream-based abstractions

Ingestion/Storage systems and processing engines should understand native, stream-based interfaces for data ingestion, processing, and storage of streams of records that can subsume the standard batch models:

The data ingestion interface is leveraged by stream producers that write input streams but also by processing engine workers that store processing state to local storage instances.

The data storage interface is handled internally by the storage system, being used to permanently store input streams when needed: this action can be done asynchronously based on stream metadata and hints sent by the processing engine.

The data process interface is bidirectionally exposed: first, it can be leveraged by the processing engine to pull data from the stream storage; second, we envision future processing workflows sending process functions to storage whenever possible.

Moreover, to respond to previously identified requirements, we envision that on top of the (stream-based) unified ingestion/storage architecture are built key-value interfaces (e.g., put/get, multi-write/multi-read) necessary to provide fine-grained access to ingested streams. Enhancing storage solutions with ingestion capabilities will also help, on the one hand, developing complex ***stream-based workflows*** (i.e., by allowing to pass intermediate/final aggregated stream results to other streaming applications) and, on the other hand, better supporting ***stream checkpointing*** techniques (i.e., by efficiently storing temporary results).

5.2 Scalable data ingestion for stream processing

State-of-the-art stream ingestion systems (e.g., [49, 91, 96]) employ a ***static*** partitioning scheme where the stream is split among a fixed number of partitions, each of which is an unbounded, ordered, immutable sequence of records that are continuously appended. Each broker is responsible for one or multiple partitions. Producers accumulate records in fixed-sized batches, each of which is appended to one partition. To reduce communication overhead, the producers group together multiple batches that correspond to the partitions of a

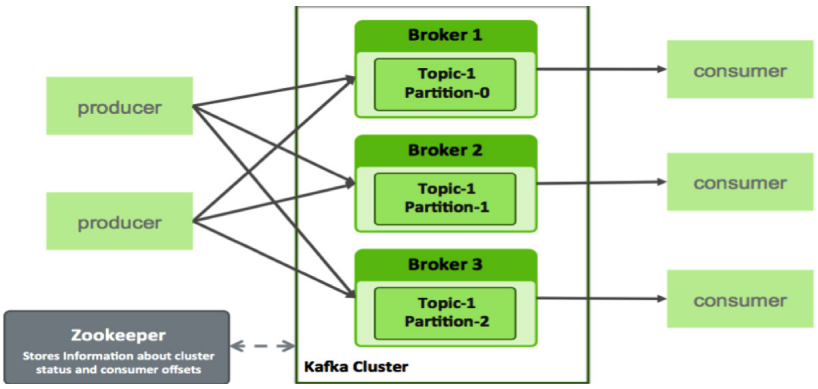

Figure 5.2 – Static partitioning in Kafka (taken from [49]). Fixed partitioning can be a source of imbalance. Each partition is exclusively processed by one consumer.

single broker in a single request. Each consumer is assigned to one or more partitions. Each partition is assigned to a single consumer. This eliminates the need for complex synchronization mechanisms but has an important drawback: the application needs a priori knowledge about the optimal number of partitions.

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

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

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

5.2.1 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-the-art approaches assign a single producer and consumer to each partition (see Figure 5.2), the users need to be aware of both semantics (e.g., logical partitioning) and performance issues when using static partitioning. Therefore, we propose a dynamic partitioning scheme where users fix the high level partitioning criteria from the semantic perspective, while the ingestion system is responsible to make each partition elastic by allowing multiple producers and consumers to access it simultaneously. To this end, we propose to dynamically split each logical partition into fixed-size sub-partitions, each of which is independently managed and attached to a potentially different producer and consumer. Therefore, each logical partition corresponds to multiple physical sub-partitions that are created and filled ***in order*** by producers and dynamically discovered and processed ***in order*** by consumers.

5.2.2 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.

For certain streaming applications that handle stream records with multiple versions (i.e., each record has a key and possibly multiple values), users may be interested in compacting the stream (i.e., preserving only the last value of a given record). This feature is called in Apache Kafka ***the log compaction*** and it is handled by a log cleaner (each log represents one partition of a stream). For these situations, the partition offset (associated with each record) helps to decouple application consumers from storage implementations. Since our lightweight offset is associated by consumers with the record's physical position, we need an efficient way to additionally handle the stream compaction. We propose to leverage the fixed sub-partitions of a stream partition as follows. Applications normally define checkpoints that are based on the lighweight offset that corresponds to one sub-partition. Therefore, we propose to enable stream compaction only for sub-partitions that correspond to old checkpoints that become obsolete (i.e., not anymore used by applications in the process of recovery). Since our goal is to provide a unified architecture for ingestion and storage, the unified engine also stores application checkpoints and therefore has enough knowledge to transparently handle stream compaction if needed.

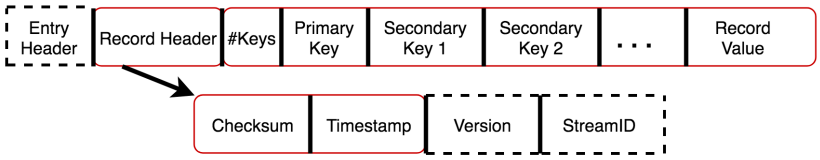

Figure 5.3 – Multi-key-value data format for stream records: Version, StreamID, and EntryHeader are optional, being necessary only for enabling key-value put/get interfaces for fine-grained access (based on [50]).

5.3 Handling diverse data access patterns

A unified ingestion/storage engine should efficiently provide interfaces for both online (records or streams) and offline (objects) access patterns. Moreover, the system should efficiently support access to a large number of streams and objects. Therefore, in order to efficiently tackle these diverse data access patterns, we propose the following set of design principles.

5.3.1 Model stream records with a multi-key-value data format

Streaming queries or SQL on streams have recently emerged as a convenient abstraction to process flows of incoming data, inspired by the deceptively simple decades-old SQL approach. However, they extend beyond time-based or tuple-based stream models [45]. Given the complexity of stream SQL semantics and the support they require for handling state for such operations, it is important to understand how a stream storage system can sustain and optimize such applications. For instance, computing ***aggregate functions*** on streams is possible using various ***windowing constructs*** (e.g., tumbling, sliding windows) which require fine-grained access to data. Moreover, storage for streaming needs to be flexible and ***detect and then dynamically adapt*** to the observed stream access patterns [12, 68]: ranging from fine-grained per record/tuple access to group queries (multi get/put) or scan-based.

Stream records are traditionally modelled with a simple key-value format, where value is an uninterpreted blob of data (e.g., in Kafka). In order to have secondary indexes, required to efficiently search for records by their attributes (including the primary key), clients and servers have to agree on where the secondary keys are located in the record. To efficiently index a stream's records, we want to avoid parsing the record's value. We propose to represent a record with a primary key, optionally multiple secondary keys, and a variable-length uninterpreted value blob (allowing for other attributes, e.g., as done in [50]), in order to give more flexibility to an enhanced ingestion/storage architecture (see Figure 5.3). Our goal is to build a data model flexible enough to allow the sequential efficiency access to streams/objects and a low-latency fine-grained access to records of a stream.

5.3.2 Leverage log-structured storage in memory and on disk

This is a decision that comes naturally, being emphasized by the structure of a stream: data arrives in a record-by-record fashion and it is processed and archived/replicated similarly (see Figure 5.4). Moreover, leveraging sequential access to streams in-memory or on disk can

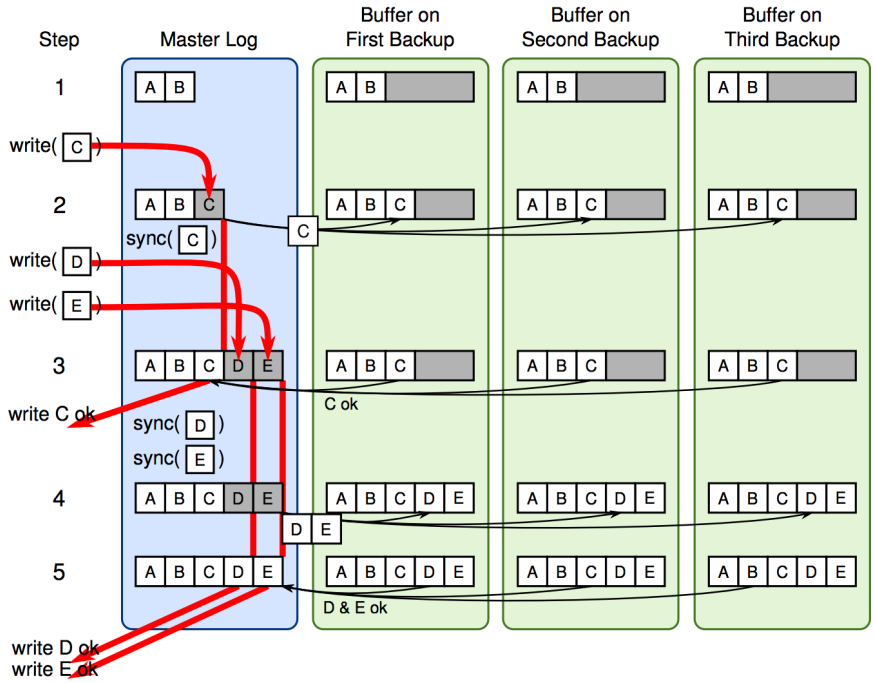

Figure 5.4 – Stream records ingestion and replication with logs. The Master (also called broker) is the process managing the stream ingestion while Backups are responsible to store stream replicas. Maintaining the order of ingested records in consistent, uniform segments part of a log (e.g., one log per stream). Step 1: Records A and B are replicated on backups in memory. Then, a producer asks to write record C. Step 2: During the time a Master is replicating C on its backups, new requests to write records D and E arrived. Step 3: C is acknowledged, while D and E are copied to the active uniform segment. Step 4: D and E are batched in a single replication operation. Step 5: Producers of D and E are acknowledged [figure taken from RAMCloud].

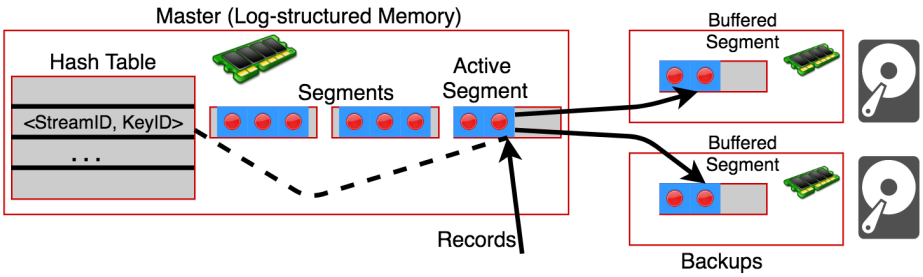

Figure 5.5 – Log-structured storage in memory and on disk. The Master server consists primarily of a hash table and possibly one or more in-memory logs for each stream. The hash table can optionally represent the indexed records as a requirement for ensuring the state management function of local processing. Each new record of a stream is appended to the stream log's active segment (i.e., buffer) associated to the record's stream and is a/synchronously replicated to volatile (i.e., DRAM) buffers on backups. Client writes are acknowledged once all backups have buffered the new addition to the active segment. Stream logs are replicated across several backup disks for durability.

maximize performance (i.e., ingestion/processing throughput and latency) for both online and offline access patterns. We argue that an ingestion/storage system should implement a (unified) log-structured approach for both data in memory and on disk that could serve well under changing access patterns for both real-time data serving and batch analytics (see Figure 5.5). This is very similar to the log-structured techniques developed in the RAMCloud project [86]. However, while in RAMCloud the access patterns are record-oriented thus have a random nature, we propose to primarily optimize for the sequential access pattern of dynamic streams and objects. As such, each broker (providing direct access to streams) can organize the sub-partitions of a stream with one or multiple logs, with its data similarly stored in memory and on disk. Likewise, backups leverage log-structured storage and maintain the replicated data in-memory and then flushes it (asynchronously) to disk to ease access at recovery time.

5.3.3 Adaptive and fine-grained replication for multiple streams

Replication [115] is the standard solution used for ensuring fault-tolerant stream data ingestion/storage. We propose that each stream (ingesting data into multiple sub-partitions) is further logically organized into one or multiple ***virtual logs*** that continuously aggregate producer requests containing multiple chunks, at least one for each partition of a stream, in order to durably replicate the chunks. Backup entities responsible to durably store replicated (virtual) logs can continue to see both in-memory and on disk data as structured logs.

A scalable ingestion and storage system has to efficiently accommodate multiple tenants each pushing multiple streams of data with different requirements for ingestion throughput and read/write access latency: for instance, it should efficiently support the ingestion of tens of very large streams (i.e., having tens of thousands of partitions) or the ingestion of millions of very small streams (i.e., having a few partitions). We propose to add support for customizing the replication throughput of a single stream by allowing the system/users to tune the ***replication capacity***, i.e., how many replicated virtual logs can be created for a single stream. Replication should be possible in both synchronous and asynchronous modes. e.g., for applications that require faster ingestion with weaker consistency requirements, data should be replicated asynchronously; for applications that prefer strong durability over relaxed consistency, data should be replicated durably and synchronously before producers are acknowledged and consumers pull data for processing.

5.4 Efficient integration with processing engines

Towards minimizing data movement for efficient data ingestion and processing, a unified architecture for ingestion and storage should efficiently handle applications with a diverse set of access patterns. Using an extensible, modular architecture could provide the flexibility needed to handle not only online applications requiring low-latency access to unbounded streams, but also offline/interactive applications needing scalable, high throughput access to multiple bounded streams (e.g., consider a large application with tens of thousands of operators, each one managing its data through a stream). To enable an efficient integration with processing engines we propose the following design principles.

5.4.1 Enable data locality support as a first class citizen

Offline/batch analytics engines developed a series of optimizations based on data locality support [119, 120], that also have to be enabled by a unified ingestion and storage architecture. Moreover, recent (near) real-time applications requiring low-latency data access may also benefit from data locality support (i.e., bypassing the network and reducing the communication interference between reads and writes, consuming data directly from memory whenever possible). To this end, we propose that the ingestion/storage system should have a fine-grained control over acquired streams and avoid leveraging third parties for persistence of data (e.g., as done by Kafka leveraging the operating system kernel cache). Then, the co-location of the processing engines with the ingestion/storage nodes should leverage a shared memory buffer approach while streaming consumers could be implemented with a push-based approach (versus a pull-based approach in state-of-the-art streaming), thus improving throughput and reducing considerably the processing latency. We argue that current hardware trends (i.e., multi-core nodes, up to tens or hundreds of cores per node with 100s of GB of memory) encourage the implementation of a unified model, allowing when possible the co-location of stream ingestion/storage and processing engines for better cooperation towards handling huge volumes of streams of data while minimizing data movement.

5.4.2 Distributed metadata management for un/bounded streams

We refer to stream metadata as the small data describing a stream and its partitions. A scalable ingestion/storage system should not rely on a single node for handling metadata (e.g., as done in HDFS with the NameNode or by Kafka's Zookeeper for metadata management). We propose to handle metadata on the brokers providing access to their corresponding data: in this way metadata are distributed naturally over the whole cluster of (broker and backups) nodes. Metadata should be allowed to be queried independently of data. In order to boost performance, stream partition metadata should be stored along the data it characterizes: in this way metadata can be easily and dynamically rebuilt when data is recovered or migrated on other brokers. Moreover, when needed to scale up and down the ingestion/storage engine, in order to efficiently handle different stream workloads, we should only migrate metadata describing their data (on current brokers) and avoid moving stream data.

5.4.3 Towards pushing processing to storage

We describe in the next chapters the prototype implementation of the KerA architecture that illustrates the majority of the previous principles, without considering major enhancements to Big Data processing frameworks, such as pushing processing functions to storage, and the required support from the ingestion/storage engine that we leave for future work. However, we do leverage the ***locality and metadata techniques*** in order to efficiently integrate our ingestion/storage engine with a state-of-the-art Big Data stream processing framework as a first step towards "in-storage" processing. Recent work [58] proves the importance (and performance impact) of pushing "code" to storage systems, while discussing the oportunity brought by low-latency networking and in-memory storage. One future problem is understanding ***which (batch/streaming) workloads can benefit most from pushing processing to storage and when it is best to leverage such technique?***.

Chapter 6

High level architecture overview

Contents

THis chapter presents a high level description of the KerA architecture, a large-scale unified ingestion and storage system that illustrates the majority of the design principles introduced in the previous chapter. We recall the reader three critical objectives that drove the current architecture of KerA:

1. To enable support for fine-grained access to ingested/intermediate streams managed by streaming applications (i.e., enable state management through the ingestion/storage engine);

2. To better respond to streaming applications that need faster responses than what current state-of-the-art ingestion systems (such as Apache Kafka) offer;

3. To reduce the storage and network utilization significantly, which can contribute to reduced times for stream processing and archival.

For the first objective, as opposed to current systems (e.g., Kafka) that rely on the operating system for handling streams, we propose to directly manage ingestion and persistence of acquired streams through a custom memory management (we explore the implementation of this topic in the next chapter). Stream records are represented with a simplified multi-key-value data model in order to enable both first and second level indexing mechanisms (e.g., as the ones developed by RAMCloud). The second objective can be achieved by relaxing jump forward/backward at random offsets while still enabling elimination of earlier values (certain streaming use cases may require deduplication by record key), if required, and by enabling data locality support whenever possible. The third objective can be achieved by unifying the ingestion and storage components and exposing a set of common online/offline interfaces for data ingestion (i.e., producers writing streams), data processing (i.e., consumers reading streams), and data storage (i.e., the archival/storage function is handled internally based on additional hints from applications – such as consumer offsets).

After introducing the data model used for record and stream representations, we describe the dynamic partitioning model and our lightweight offset indexing technique. Then, we describe the system architecture, specifying the roles of each component and their interactions towards ensuring the ingestion and storage functions. After we present the client interfaces exposed by KerA for managing streams and objects, we describe our proposal for efficient distributed metadata management. Finally, we discuss the necessary techniques towards a fault-tolerant architectural implementation. The next chapter discusses in more detail other technical aspects regarding data locality and adaptive replication support.

6.1 Unified data model for unbounded streams, records and objects

A ***stream*** is an unbounded sequence of records that are not necessarily correlated with each other. An ***object*** is simply represented as a bounded stream. Figure 6.1 illustrates the conceptual representation of records, chunks, segments, groups and streamlets that we describe below.

Stream records. 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 [50]. The record's entry header contains an attribute to optionally define a version and a timestamp field that are necessary to efficiently enable key-value interfaces.

Chunks. Record entries are further grouped by producers into ***chunks*** (a chunk has a configurable fixed size of up to 8-16 MB). A request contains multiple chunks. The chunk aggregation is useful for three 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

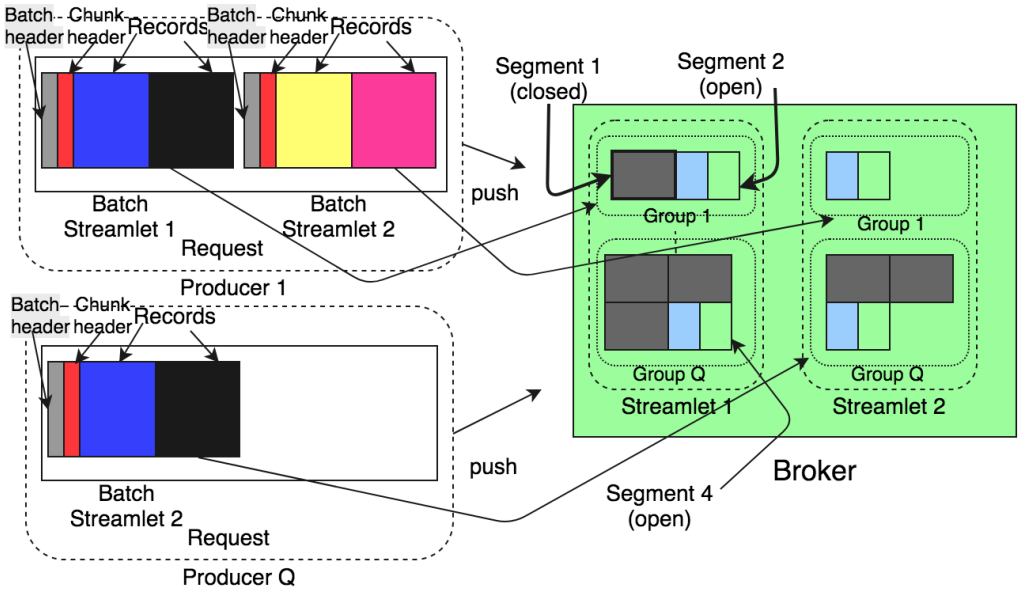

Figure 6.1 – Representation of records, chunks, segments, groups and streamlets: we illustrate how producers aggregate records into requests and push them to brokers. Each request (of configurable request size) contains one or multiple chunks (each chunk with a configurable size). We exemplify with a stream composed of two streamlets (a streamlet is a logical partition that contains multiple fixed-size sub-partitions called groups, see next section). Each streamlet holds up to Q active groups where chunks are appended. Each producer is responsible for one or multiple streamlets (based on the partitioning strategy each producer can be configured to push records to one or all streamlets). Assuming we have Q distinct producers identified by identifiers 1 to Q, each producer's request is appended to the active group corresponding to the entry calculated as producer identifier modulo Q. As such, producer 1 writes into group 1 of each streamlet, producer Q writes into group Q of each streamlet.

section), this helps ensuring exactly once semantics and ordering semantics necessary for consistent ingestion and processing. Third, as described in the next chapter, the replicated ***virtual log*** keeps references to a set of chunks, aggregating them in ***virtual segments*** in order to avoid another copy before replicating the chunks.

Segments. A producer client prepares and writes a request containing a set of chunks. Each chunk is acquired by the ingestion/storage system and appended into a ***physical segment*** representing a buffer managed by the broker. A segment has a customizable fixed size (8-16 MB) necessary for efficiently moving data from memory to disk and backwards (on brokers, processes handling data access, segments have the same structure on both disk and memory). A stream is composed of a set of uniform segments containing chunks of records of the same stream. Virtual segments aggregate chunks (through chunk references) from possibly various physical segments and replicate them on backups that store them on disk.

Groups of segments. In order to reduce the metadata necessary to describe the unbounded set of segments of a stream, we further logically assemble a fixed number of segments into a ***group***. In this way, each stream can be logically represented by a smaller,

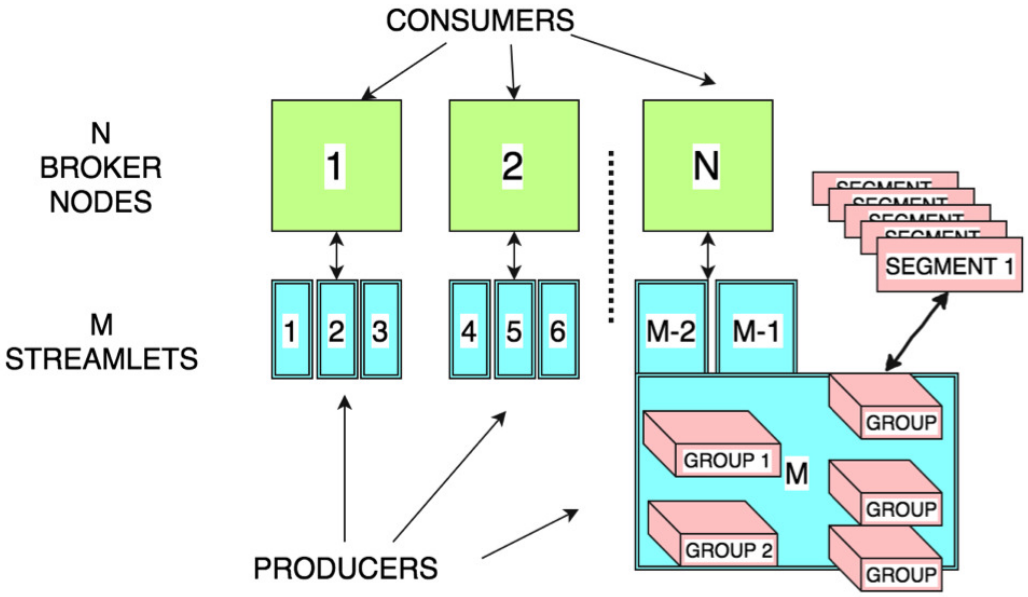

Figure 6.2 – Semantic partitioning of streams with streamlets. A stream (also called distributed topic) is logically composed of a set of streamlets. At its turn, each streamlet is composed of an unbounded set of fixed-size sub-partitions called groups of segments.

unbounded set of groups of segments. Each group is further assigned to one consumer/producer to better load balance data (higher parallelism) and increase processing/ingestion throughput. An object is composed of a fixed number of groups.

Streamlets. A streamlet is a container for fixed-size sub-partitions (groups of segments), with each group created dynamically (active groups are created as needed while closed groups suffer no appends). A stream has up to M number of streamlets that are initially created on a set of N, $N \leq M$, number of 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 can contain an unlimited number of groups of segments that can be processed in parallel by multiple consumers (ensuring vertical scalability) of which up to Q active groups correspond to physical sub-partitions that allow appends from multiple producers.

Objects. An object can be seen as a distributed, ***immutable*** bounded stream, that allows read/append operations (see [71] for handling distributed, ***mutable*** objects that additionally support update/modify operations). In KerA, the object is naturally represented by a fixed number of groups (e.g., we can think about the group as an equal representation of the HDFS file's block). In order to reduce the metadata overhead for ingesting large (unstructured) objects (i.e., only for read/append operations, objects do not benefit from the fine-grained record model), producers create chunks having a single record with a large value (8-16 MB). Since groups are split into small segments, this can help batch analytics to load balance the processing of the groups segments in order to solve the straggler issues common in Big Data analytics. We argue that

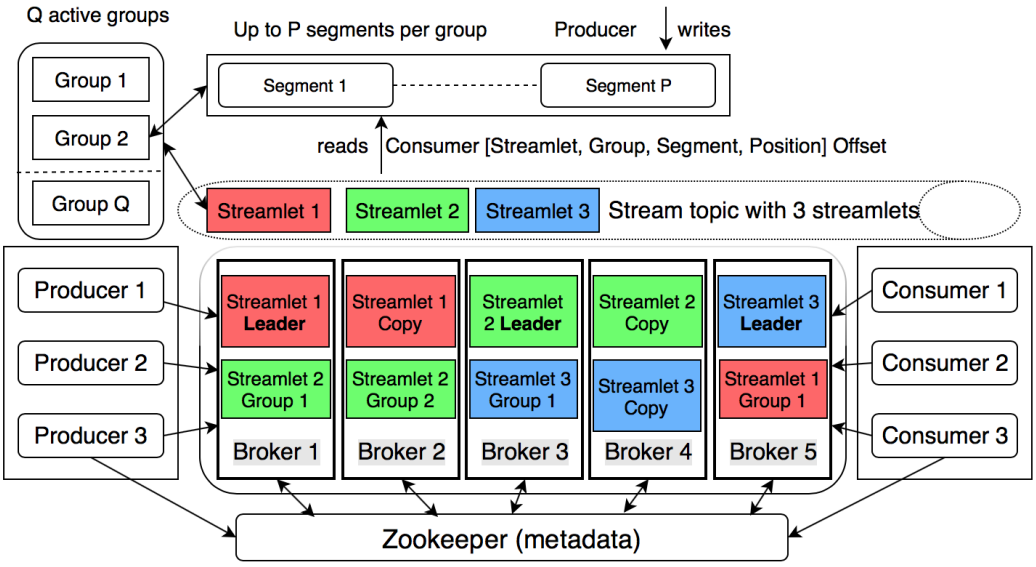

Figure 6.3 – Stream topic partitioning illustrated with 3 streamlets and 5 brokers. Zookeeper is responsible for providing clients the metadata of the association of streamlets with brokers. Streamlets' groups and their segments are dynamically discovered by consumers querying brokers for the next available groups of a streamlet and for new segments of a group. 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.

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 efficient management of streams and objects by the unified ingestion/storage system. However, objects in KerA could additionally support (in-place) update operations by leveraging the fine-grained record-based representation. This would require support for transactions and versioning [80], that we leave to future work.

6.2 Scalable data ingestion and processing

6.2.1 Dynamic stream partitioning model

KerA implements dynamic partitioning based on the concept of ***streamlet*** (Figure 6.2), which corresponds to the semantic high-level partition that groups records together. Each stream is therefore composed of a fixed number of streamlets. In turn, each streamlet is dynamically split into ***groups***, which correspond to the sub-partitions assigned to one producer and one 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 in an efficient fashion, each group is further split into fixed-sized ***segments*** (as we describe later in the lightweight offset indexing mechanism). Moreover, since segments are fixed-size in-memory buffers (e.g., 8-16 MB), the segment metadata management is crucial for ensuring consistent processing. The maximum size of a group is a

multiple of segment size $P \geq 1$. To control the level of parallelism allowed on each broker, for each streamlet only $Q \geq 1$ groups can be active at a given moment.

As can be seen in Figure 6.3, elasticity is achieved by assigning an initial number of brokers $N \geq 1$ to hold the streamlets M, $M \geq N$. 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 sub-partitions (groups) to process.

6.2.2 Lighweight offset indexing

In order to ensure ordering semantics, each streamlet dynamically creates groups (and their segments, initially one) that have unique, monotonically increasing identifiers; a streamlet can have groups uniquely identified starting from 1, e.g., 1 to 1000, with each group having up to a fixed number of logical segments with identifiers starting from 1, e.g., 1 to 16. Each logical segment is associated with a physical segment (custom memory management) that is uniquely identified on each broker by a monotonically increasing identifier.

Brokers expose metadata information through RPCs to consumers 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. Since segments are immutable, the position field does not change. The consumer initializes it to 0 (the broker understands to iterate to the first record available in that segment) and the broker responds with the last record position for each new request, so the consumer can update its latest offset to start a future request with. Using this dynamic approach (as opposed to the static approach used by explicit offsets per partition, in which clients have to query brokers to discover groups and segments), we implement lightweight offset indexing optimized for sequential record access.

Stream records (grouped in chunks 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/segment can (optionally) be closed (and its memory released) after a configurable timeout if it was not appended in this time: since segments are represented by managed in-memory buffers, releasing them when not used could help efficiently create more active streams. We plan as future work to study the efficiency of a garbage collector for such metadata.

6.2.3 Favoring parallelism: consumer and producer protocols

Producers only need to know about streamlet metadata when interacting with KerA. The input chunk is always ingested to the streamlet 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 chunk tagged with the streamlet id). Producers writing to the same streamlet synchronize using a lock on the streamlet (also required to ensure streamlet migration) in order to obtain the active group (or create one if needed) corresponding to the Q^{th} entry based on the producer identifier. The lock is then released and a second-level lock is used to synchronize producers accessing the same active

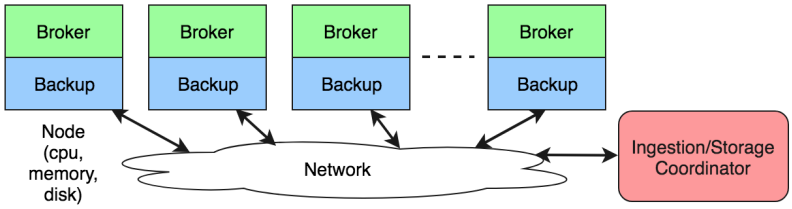

Figure 6.4 – The KerA architecture: similar to RAMCloud, the coordinator manages storage nodes on which live brokers and/or backups. Clients mainly interact with brokers while backups are simply used for storing stream's replicas.

group. Thus, two producers appending to the same streamlet, but to different groups, may proceed in parallel for data ingestion.

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. Consumers process groups of a streamlet in the order of their identifiers, pulling data from segments also in the order of their respective identifiers. Brokers maintain for each streamlet the last group given to streaming consumers identified by their consumer group id (i.e., each consumer request header contains a unique application id). A group is configured with a fixed number of segments to allow fine-grained consumption with many consumers per streamlet in order to better load balance groups to consumers. As such, each consumer has a fair access chance since the group is limited in size by the segment size and the number of segments. This approach also favors parallelism. Indeed, in KerA a consumer pulls data from one group of a streamlet exclusively, which means that ***multiple consumers can read in parallel from different groups of the same streamlet***.

6.3 Global architecture

As represented in Figure 6.4, KerA's architecture contains a single layer of brokers that serve producers and consumers. On each node it is possible to install a broker service (each broker has an ingestion component offering pub/sub interfaces to stream clients) and/or a backup service that is used for storing stream's replicas that are only read during crash recovery. The current architecture allows (if required) for separation of nodes serving clients from nodes serving as backups. Brokers also expose through RPCs the metadata describing the stream's partitions (i.e., the streamlet's groups of segments). The implementation of the coordinator, broker and backup services (e.g., process instantiation and their communication through the RPC system) is based on RAMCloud.

6.3.1 Stream management: the coordinator role

The ingestion/storage 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 management of brokers stream-streamlet 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 (e.g., the coordinator state is replicated through a log-based system such as Zookeeper). The coordinator should also be 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.

6.3.2 Stream ingestion: the broker role

A broker manages the main memory of a server and handles multiple streams by ingesting stream chunks into the active segment of the streamlet's active group (if one exists, otherwise a new group/segment is created). The broker handles requests (for data and metadata) from both producers and consumers through RPCs.

To facilitate the in-memory management of stream metadata, a broker implements the ***stream manager*** component responsible to handle a map structure that associates to a stream identifier a ***Stream*** object that manages the access to its local streamlet partitions. The stream manager is also responsible to provide physical segments and manage their associated in-memory buffers.

The stream object maintains a list of ***Streamlets*** objects that manage the metadata structures regarding their groups of segments. Furthermore, it provides a stream lock to facilitate consistent ingestion, migration and processing. Broker APIs are defined through the Stream and Streamlet interfaces. Each streamlet provides a number of active (open) groups (up to Q) and their corresponding active logical segments associated to physical segments (i.e., pointers to memory buffers) into which the next writes are appended. Once a logical segment is closed (it suffers no more appends), the ***segment manager*** provides a new segment.

The broker also implements the ***persistence manager*** that is responsible to asynchronously move data from memory to disk and back to memory at the segment granularity. We associate a physical segment id with the logical segment id that is part of the streamlet group metadata. If a client sends a request with an offset corresponding to a logical segment for which its physical segment is not loaded into memory (with the offset valid), we send back a retry error and we trigger the persistent manager to load to memory that particular segment and its successive segments/groups in order of their identifiers (catch-up reads).

6.3.3 Stream replication: the backup role

The backup service is responsible to store a stream's replicas. A backup is configured with a limited number of in-memory segments (e.g., 256 segments of 8 MB each) in order to acknowledge as fast as possible the replication RPCs. A backup 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 disk device. The backup maintains in memory the association of replicated streams with local segments through virtual logs; this metadata is later useful for the recovery or migration of a stream's chunks.

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 a stream a set of ***virtual logs***, with each virtual log managing replicated virtual segments. Each virtual segment contains references 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.

6.4 Client APIs

We describe in Table 6.1 the main operations that KerA's streaming clients can leverage to manage streams. Batch analytics processing objects need additional metadata support. Since an object has its groups of segments already filled, batch clients can reduce the number of RPCs required to fill consumer offsets by leveraging the operations described in Table 6.2. Multiread/write operations issue multiple RPCs in parallel to a set of brokers.

For a production-ready system, other APIs are necessary to fulfill the user needs. Although we do not develop operations for deleting or migrating streams, the current implementation provides needed support (e.g., locking, metadata management). We discuss in the next chapter more details about our implementation prototype and the necessary integration techniques needed to employ efficient access to data for both online and offline analytics.

6.5 Distributed metadata management

As previously mentioned, stream partitions metadata are dynamically created at ingestion time and are made available through RPCs by each broker. If a broker crashes, its local stream's partitions are recovered on other brokers that recreate the corresponding metadata at recovery time. Another important issue is how migration impacts metadata management. Initially, a stream is created on a smaller number of brokers than the number of streamlets. When the ingestion throughput is higher than what current brokers can sustain, the coordinator can decide that some streamlets should be migrated to other brokers. However, we only want to migrate the streamlet's metadata (e.g., last groups identifiers) and leave the current data (groups of segments) on their current brokers.

Considering that stream processing sequentially pulls data in order to always process latest stream records, we consider the following design. For each streamlet, we associate current groups with their broker and save this metadata on the new broker (this information is also replicated on backups that associate the current broker identifiers with replicated data). If clients ask for any of these groups, we redirect the requests to their brokers. The same mechanism is used when scaling down, i.e., migrating streamlets towards a smaller

RPC method	Description
createStream (streamName, N, M) $\rightarrow$ ***streamId***	Creates a new stream named streamName if it does not already exist and returns a 64-bit stream identifier to be used by read and write operations. N is the initial number of brokers, on each broker a subset of the M streamlets is created and associated, a streamlet being exclusively managed by a single broker.
getStreamId (streamName) $\rightarrow$ ***streamId***	Resolve the stream named streamName and return its 64-bit idendifier.
multiWriteChunks (requests, countRequests, streamId, clientId) $\rightarrow$ ***status***	Used by producers to write a number of chunks. Each request contains a set of chunks; each chunk has a header and a set of stream records. Each chunk corresponds to a single streamlet. Based on the chunk's streamletId, we identify the broker node responsible for its groups. The streamId field is used to further identify the stream object that is responsible for the streamlet metadata. The producerId field is used to identify the corresponding active group into which the chunk is appended.
getNextGroups (streamId, streamletId, applicationId, q) $\rightarrow$ ***[groupId]***	Used by consumers to obtain the next available groups for given [streamId, streamletId]. The applicationId is used by multiple consumers from an application. Since a streamlet's groups are potentially shared by multiple consumers, the broker uses the applicationId in order to cache the last given groupId. The parameter q, $1 \leq q \leq Q$, can be used to tell the broker the maximum number of groups that can be returned in a single request.
getNextSegments (streamId, streamletId, groupId) $\rightarrow$ ***[segmentId]***	Used by consumers to obtain the next available segments for a given sub-partition represented by [streamId, streamletId, groupId]. A group has a configurable fixed number of segments, initially one. While producers append more data, new segments are created until the group gets filled and eventually a new group is also created. This RPC could be pipelined with a multiReadChunks request in order to reduce the number of RPCs.
multiReadChunks (requests, countRequests, buffer, params) $\rightarrow$ ***status***	Used by consumers to pull one or more chunks, given the offset specified by each request object. Each offset is specified as [streamId, streamletId, groupId, segmentId, position]. The params field is used to pass other attributes (e.g., chunk size, request size, applicationId). The buffer is filled with pulled chunks corresponding to each request and metadata to be used for the next request (e.g., next position to be used by the next request if the current segment is not fully processed, else next available groups and/or segments).

Table 6.1 – Available KerA operations for managing streams or objects.

RPC method	Description
getAvailableGroups (streamId, streamletId) → [firstGroupId, lastGroupId]	Used by batch consumers to obtain the available groups for given [streamId, streamletId]. Because a streamlet's groups are created in order and assigned a monotonically increasing identifier, the client can build offsets for groups with identifiers starting at firstGroupId and ending at lastGroupId.
getAvailableSegments (streamId, streamletId, groupId) → [firstSegmentId, lastSegmentId]	Used by batch consumers to obtain the available segments for given [streamId, streamletId, groupId]. Because a group's segments are created in order and assigned a monotonically increasing identifier, the client can build offsets for segments with identifiers starting at firstSegmentId and ending at lastSegmentId.

Table 6.2 – Additional KerA operations for managing streams or objects.

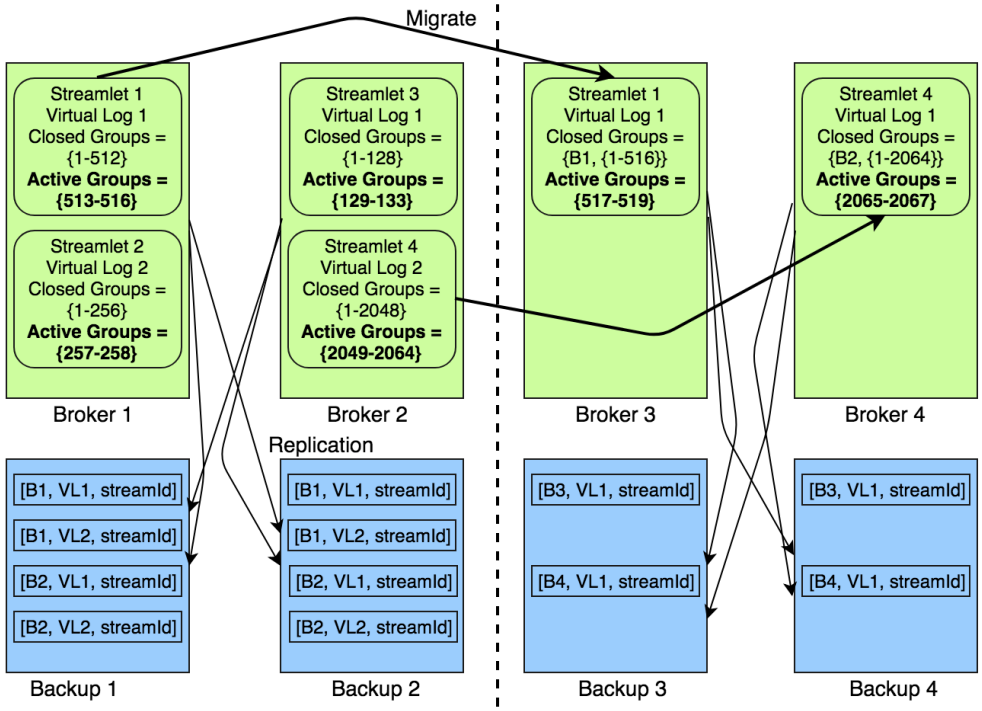

Figure 6.5 – Distributed metadata management and impact of streamlet migration. We illustrate the metadata management for a stream having 4 streamlets before and after migration.

number of brokers. Eventually, the coordinator builds an updated map of which broker keeps which interval of groups of a certain streamlet. This design in which groups of a streamlet can possibly be managed by different brokers allows for the minimization of data movement at the expense of the retry of the first client request for older groups. However, only one broker is actively ingesting data for associated streamlets, while other brokers could serve readers the data corresponding to old groups.

Figure 6.5 illustrates how we propose to manage metadata for a stream with 4 streamlets before and after migration. Initially the stream is created on 2 brokers, with each broker associated with 2 streamlets (e.g., Broker 1 is associated with streamlets 1 and 2). Brokers are uniquely identified with monotonically increasing identifiers that are never reused. Each streamlet is associated with a virtual log that is replicated on 2 backups. e.g., streamlet 1 is associated with virtual log 1 and is replicated on backups 1 and 2. Each streamlet has a set of closed groups (suffer no more appends) and a set of active groups for ingesting data. We then add 2 more brokers into the cluster and decide to migrate streamlets 1 and 4 since they have the most number of groups on each broker. When we migrate a streamlet, we close current active groups and retain the next active group identifier to be associated on the newly associated broker. We also keep metadata on migrated streamlets to identify older groups. e.g., streamlet 1 associates broker 1 with groups 1 to 516 and creates active groups starting with group id 517. The coordinator updates the association of streamlets 1 and 4 with brokers 3 and 4. New producer requests are redirected to the new brokers. Consumer requests for older groups (e.g., streamlet 4 groups 1–2064 are managed by broker 2) first reach the new brokers in order to find the brokers responsible for them and cache this information for next requests. As such, brokers are actively responsible for new groups of a streamlet while other brokers can be responsible for older groups of the same streamlet. For crash recovery, we consider the scenario in which broker 1 is crashed. The coordinator queries backups in order to find who is responsible for its streamlets virtual logs. Then, based on broker id, virtual log id and stream id, it recovers streamlet's groups on another broker. The brokers actively responsible for a stream's streamlets need also to update their metadata references to older groups.

We leave to future work other implementation implications that may be necessary to correctly recover data of a crashed broker that holds additional groups of a streamlet, although it is responsible for other streamlets.

6.6 Towards an efficient implementation of fault-tolerance mechanisms in KerA

Fault-tolerant (storage) systems are able to continuously perform their function in spite of errors. Users of stream-based applications require low-latency (milliseconds to seconds) answers in any conditions. In order to guarantee such strict requirements, we need to employ techniques that are recognized to recover data as fast as possible. To implement fast crash recovery for low latency continuous processing, we should rely on techniques similar to the ones developed by RAMCloud [83], and leverage 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.

Similar to the migration technique previously described, we propose that in case of broker crashes we immediately recover the streamlet's metadata (based on backup's metadata) and re-enable producers to push the next stream chunks. In parallel, we proceed with the recovery of data as follows. Based on the last consumer offsets (e.g., every second, for each consumer group, we store the last offsets used in read requests – the consumer hints), 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).

Chapter 7

Implementation details

Contents

In this chapter we dive into the details of the most challenging components of the KerA architectural implementation. First, we present the producer and consumer streaming architectures that leverage the client APIs previously described (Section 7.1). KerA builds atop RAMCloud's [86] RPC framework in order to leverage its network abstraction that enables the use of other network transports (e.g., UDP, DPDK, Infiniband), thus offering an efficient RPC framework on top of which we develop KerA's client RPCs. Moreover, this allows KerA to benefit from a set of design principles like ***polling and request dispatching*** [57] that help boost performance (e.g., kernel bypass and zero-copy networking are possible with DPDK and Infiniband). Second, we describe the implementation of the persistence manager component that mediates both online and offline accesses (Section 7.2) and we comment on the threading architecture implementation of brokers, differentiating in time (i.e., switching

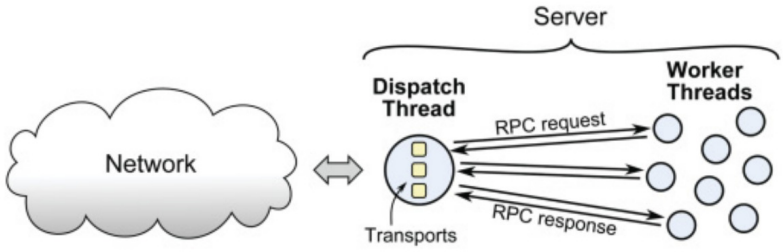

Figure 7.1 – The dispatching-workers threading architecture [taken from [86]].

between different broker process implementations) between real-time brokers (used for both stream/real-time and object/batch operations) and dedicated offline brokers (used for object operations needed for batch processing). Third, we describe the implementation of our virtual log replication mechanism necessary to efficiently handle multiple streams (Section 7.3). Finally, we describe the data locality support for streaming operations and the integration of KerA with Apache Flink as a first step towards pushing processing to storage in Big Data analytics.

7.1 Streaming clients: how reads and writes work

7.1.1 The RPC layer

An important challenge when building a system such as KerA, suitable for efficiently handling both online and offline access patterns, is the low-level management of read/write client interfaces. Simple remote procedure calls such as reading a set of small objects (few KBs) should finish in a few (tens) of microseconds, while bulk transfers (a few MBs) should be efficiently supported as well. Even more, KerA's clients should easily benefit from more recent networking equipments such as Infiniband, thus KerA should additionaly leverage multiple networking transports (e.g., TCP or DPDK).

Fortunately, this problem was intensively investigated by the RAMCloud project. RAMCloud [86] achieves very low end-to-end latencies when reading or writing small objects (e.g., 5-10 microseconds) by leveraging high-speed networking such as Infiniband. Moreover, it additionally implements support for UDP, TCP and DPDK through a low-level networking transport design. RAMCloud employs a dispatching architecture (Figure 7.1) with a single dispatch thread handling all network communication. As such, each incoming RPC request is passed by the dispatch thread to one of the worker threads for its handling; once a response is built, the worker thread passes it to the dispatch thread for transmission. RAMCloud built a general and efficient RPC framework that KerA leverages in order to implement the client RPCs detailed in the previous chapter. Each RPC can manage up to a configurable amount of data (e.g., up to 8-16 MB per request).

7.1.2 Streaming clients architecture

First, we introduce the management of chunks and physical segments by producers and brokers: this is necessary to further understand how each chunk is finally replicated on

backups.

Building producer requests: management of chunks

Figure 6.1 illustrates how producers organize stream records into requests that contain multiple chunks. Being able to send more chunks in a request helps mitigating latency and throughput tradeoffs. A chunk is appended to the open segment of corresponding active group (the group is computed at the broker side based on producer and streamlet identifiers that are found in the request header). The broker only uses the portion of the request starting with the chunk header (including subsequent records) and appends it as such to a group's open segment. Each append operation can lead to the creation of a new segment or even the creation of a new group. The chunk header (reference) contains attributes for the corresponding ***[group, segment]*** of a streamlet that are updated at append time: these attributes are important at recovery time in order to consistently reconstruct each group (additionally, each segment is tagged with the stream and streamlet identifiers).

Each producer implements two threads that communicate through shared memory (a set of empty chunks are reused for next requests). For each new stream record the first thread identifies an available chunk where the record is appended according to the partitioning strategy (round-robin or by record's key, which is hashed to identify a streamlet). For each streamlet, a set of chunks is dynamically created in the shared memory. The second thread creates the next requests: it gathers a set of chunks up to the request size, one for each streamlet, and manages the RPC invocations.

Replicating chunks after broker appends. Each producer request is characterized by the stream and producer identifiers and a set of chunks, each one characterized by its length and a streamlet identifier. The broker identifies the stream object corresponding to the stream identifier and then for each chunk identifies the streamlet active group based on the producer identifier and parameter Q (how many active groups a streamlet is configured with). The chunk buffer is appended to the active group (this operation internally creates a new segment and/or a new group if necessary) and then a chunk reference is appended to the replicated virtual log corresponding to the streamlet of the active group entry. Once all chunks of a request are appended, the corresponding replicated (virtual) logs are synchronized on backups in order to replicate physical chunks to backups.

Building consumer requests: management of streamlet offsets

Similar to the producer, the consumer implements two threads that communicate through shared memory (a set of empty, reusable in-memory chunks). The first thread (source) builds the streamlet offsets (based on ***getStreamId, getNextGroups, getNextSegments*** RPCs previously detailed) and invokes the ***multiReadChunks*** RPC in order to pull the next set of chunks (one or more chunks for each streamlet associated to the consumer up to the request size). The second thread implements the user defined function and processes chunks records one by one.

Broker reads. By default, the broker responds to consumer requests only with stream data that is durably replicated: each logical segment (part of a group of a streamlet) main-

tains a durable head attribute for reads and another head attribute for where the next writes should go.

7.2 Efficient management of online and offline operations

KerA should be able to efficiently serve both online operations (i.e., serve real-time clients with high throughput and/or low-latency data accesses to streams of records) and offline operations (i.e., serve batch-based clients with high throughput data accesses to objects). Therefore, data should be efficiently served from memory, avoiding whenever possible disk accesses. At the same time, KerA should acquire the next data as fast as possible (i.e., data not needed should be moved to disk, while the next required data should be loaded back to memory). KerA proposes a unique set of read/write APIs for both stream and object accesses. To mediate clients accesses to data, KerA implements a ***persistence manager*** component similar to the operating system page cache, but optimized for streaming workloads (sequential data access). Moreover, KerA proposes two different roles for the broker implementation: the ***real-time broker*** is dedicated to both online/offline operations, while the ***offline broker*** is dedicated only to offline operations.

7.2.1 Persistence manager: high-performance ingestion

Each broker implements a segment manager component that associates a physical segment with an in-memory buffer. The physical segment is uniquely identified by a monotonically increasing identifier, and maintains other important attributes:

The header points to the next available offset for new appends.

The durable header points to what is currently durably replicated.

A set of references to virtual segments that replicate the segment chunks.

The closed attribute which denotes if the segment is closed or not.

A physical segment can be safely released when it is durably closed, i.e., it can suffer no appends and the durable header corresponds to the header attribute, and virtual segments that replicate the segment chunks are also durably closed. The last validation step is necessary since we do not want to invalidate the reference to the physical segments kept by the virtual segment: in this way reads and replicas are implemented lock-free. Writes and reads are also lock-free since each active group keeps a reference to the open physical segment associated to the open segment of each group (open segments are always available).

We next describe the implementation of the persistence manager component that manages the movement of closed physical segments from memory to disk and back (internally it also manages the in/validation of the references of the physical segments that may be required by next client operations). Therefore, the persistence manager needs to:

- identify the physical segments that are durably closed and that can be safely persisted on disk and finally release them;

- load the required physical segments from disk back into memory and make them available to readers as needed.

The persistence manager keeps a hashmap with 1024 buckets, for each bucket another map keeps the association of the physical segment with its pointer to the in-memory buffer. For each bucket a set of physical segment references are associated. Since a streamlet keeps the association of the logical segment with the physical segment id, for each read from a logical segment we calculate the bucket corresponding to the associated physical segment (physical segment id modulo 1024) and then we take the bucket lock, and a copy from the in-memory buffer to the RPC buffer is made before responding to the client RPC. Since the virtual segment caches references to physical segments of respective aggregated chunks, the replication RPCs are implemented lock-free with reader RPCs.

The persistence manager takes into account the current reader offsets (i.e., last physical segment accessed by a consumer request) ordered by streams and streamlets. In this way the persistence manager flushes to disk physical segments that are durably closed and processed. The segment manager associates different statuses to the physical segment. When a new segment is created its status is HEAD (open, available for next writes), if the previous segment in that group exists then it is closed and its status is updated to GROUP (closed, it suffers no appends). Each read request that pulls the last record of a segment will then update the segment status to PROCESSED. The persistence manager queries the PROCESSED segments and writes them to disk starting with the oldest physical segments (in addition, it updates the segment status to CLOSED, i.e., stored on disk, not available for in-memory operations). Catch up reads are handled as follows: if the associated physical segment is in memory, we serve the read, otherwise we trigger the persistence manager to reload that physical segment back to memory from disk. The persistence manager obtains from the streamlet object the metadata about the next physical segments that should be loaded back to memory. We plan in future work to expose to users different persistent manager policies optimized for various use cases, taking additionally hints from applications (e.g., pre-fetch data for a stream with certain priority versus other streams).

7.2.2 Real-time versus offline brokers

In order to achieve low-latency responses, the dispatch thread employs the polling technique (busy waiting) to wait for events (in order to avoid interrupts which can add microseconds to each request). This means one core is always dedicated to polling the network for new requests. While this is comfortable for KerA's brokers serving low-latency streaming (online) workloads, it can also become a bottlenech for such read/write intensive workloads (certain RDMA-based technique manages to reduce the replication CPU overhead [101], that alleviate the dispatcher bottlenecks due to backup RPCs). However, with recent hardware advancements we can assume tens of cores could be assigned to worker threads in order to process higher volumes (increased throughput). This further increases the chance for the dispatcher-worker thread architecture to become a bottleneck.

One important issue concerns the management of offline/batch analytics that need to read archived streams at large granularity with higher throughput and having less constraints on latency. Having one dedicated core for dispatching such operations can be inefficient (recall that the broker exposes interfaces for read/write accesses to bounded streams

and handles all the stream partitioning metadata). For such offline operations the crash recovery operations do not additionally impact batch operations (e.g., pings to identify crashed brokers) since latency is not an issue.

To avoid the dispatcher inefficiency we propose to differentiate in time between real-time brokers that handle both online and offline operations (i.e., real time streaming and batch accesses) from offline brokers that eventually become dedicated only to offline operations (thus optimized for high throughput operations). The implementation of the broker process should leverage Arachne [92], a recent extension of RAMCloud: "Arachne is a user-level implementation of threads that provides both low latency and high throughput for applications with short-lived threads" [92]. Thus, the offline broker does not employ the busy waiting technique for polling the network, instead it leverages interrupts which can be easily absorbed by batch workloads. This transition can be done naturally observing the streaming evolution: some streams are both ingested and archived while other streams are created either temporary or configured with a quota (e.g., keep data acquired the previous day).

Moreover, a broker has a limited amount of disk storage, thus once this space is filled, we have to add new brokers to handle more data. Once a broker disk storage is filled up to a certain threshold, the broker informs the coordinator that it should be labeled for batch operations (thus avoiding this broker in future real-time streaming operations); then, the broker restarts in order to: (1) switch to a dispatcher implementation that avoids busy waiting for polling new requests (similar to HDFS and other storage systems) and (2) reduce the memory dedicated for ingesting streams to only what is needed by the persistence manager and leave the rest for offline analytics. Broker's computational resources (CPU, memory, local storage) can still be used for further offline analytics. Later, a broker's stream data can be moved to long term archival storage (e.g., manually or through robots replacing local disks to avoid pushing through the network tens of TBs of data) and be re-enabled for both online and/or offline operations.

To further alleviate from previous issues, we propose two techniques (we describe their implementation in the next sections):

- first, based on ***adaptive replication*** we aim to reduce the number of write RPCs used for replication (additionally, this technique is also useful in absence of high-end networking hardware that offers support for efficient remote direct memory access operations such as RDMA);
- second, we leverage ***locality support*** whenever possible in order to allow KerA to reduce the interference of read and write RPCs. This helps reduce or completely remove the read RPCs necessary to pull data for processing (i.e., for both data source operators and intermediate streaming operations that keep their state as streams inside KerA).

7.3 Durable ingestion of multiple streams: adaptive and fine-grained replication

This section describes the replication design and implementation in KerA, suitable for the efficient and durable ingestion of multiple streams. First, we motivate the need for simplified metadata management for stream replication operations by backups: the replication

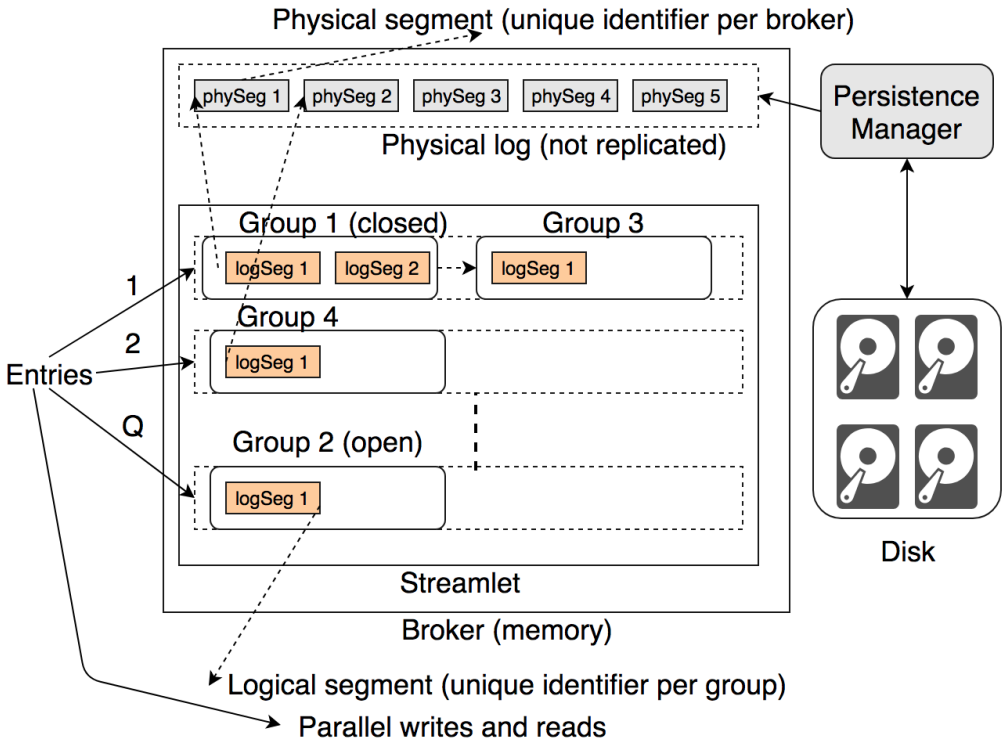

Figure 7.2 – Broker management of logical and physical segments. Each broker has a segment manager that provides a physical log with physical segments uniquely identified (monotonically assigned identifiers). The persistence manager role is to move the closed physical segments from memory to disk and back. The streamlet is composed of Q entries in order to ensure reads and writes parallelism. Streamlet groups are dynamically created and are uniquely identified (monotonically assigned identifiers). A group is composed of a configurable (fixed) number of logical segments that are associated with physical segments (distinct logical segment id – ***logSeg***, and physical segment id – ***phySeg***).

design highly impacts ordering consistency and how streamlet groups (and their segments) are identified by clients. Second, in order to efficiently manage multiple streams (durably replicated) we discuss the memory limitations of backups that lead to the design and implementation of our proposal: the virtual log technique.

7.3.1 Motivation

The need for simplified metadata management

As illustrated in Figure 7.2, streamlet groups are fixed-size sub-partitions that can be configured with a fixed number of (logical) segments. Each streamlet can possibly have an unlimited number of groups: most groups are closed, suffering no appends, and only up to Q open active groups per streamlet allowing appends. Each group of segments has to be replicated over a set of backups. Each broker should efficiently manage multiple replicated streams, thus the replication metadata impact (on brokers and backups) should be minimized. Moreover, some streams could be configured to have a higher ingestion priority

over other streams, thus, an efficient way to throttle replication per stream should also be implemented.

The main challenge is to answer the question: ***how to organize streamlet groups so that we can efficiently (i.e., achieve high throughput ingestion) and consistently (i.e., enable group order per streamlet and record order ingestion) replicate them?***

The first option is to independently replicate each streamlet group but this implies backups should maintain explicit metadata for each replicated group and their association with streamlet owner.

The second option is to consider log-structured storage similar to systems such as Apache Kafka that organize a stream into multi-logs (i.e., one replicated log for each partition). Since a streamlet has Q entries, groups being created in-order (a monotonically identifier being associated with each group), each of the Q entries can be naturally seen as a log. Thus, the streamlet concept can be backed by a multi-log (as illustrated in Figure 7.2), with ordered groups of an entry part of a log.

The implementation of the multi-log was carried out in the framework of the PhD thesis of Yacine Taleb [100]. However, although this option simplifies the implementation of crash recovery (relying on fault-tolerant techniques implemented in RAMCloud), for use cases where a large number of streams (and partitions) is needed, this technique needs higher in-memory storage support, being limited by hardware power settings (see next subsection). Moreover, the multi-log implementation did not consider the persistence requirements of brokers: each physical segment has to be uniquely identified in order to correctly store it on disk or in memory. Although Apache Kafka uses similar log-structured storage, because it depends on the operating system cache for persistence of data, it does not suffer from the previous limitations. Since KerA uses a customized memory management, it can offer better replication performance compared to Kafka. Moreover, KerA's streamlets offer higher parallelism compared to Kafka's partitions.

Constraint: limited backup memory for fast replication

Each backup holds a limited set of in-memory segments (e.g., 256 segments of 8 MB each) that can acquire data from replication RPCs. For instance, considering a setup with 100 brokers, 1000 streams replicated with factor 3, i.e., 2 backups replicating streams, the segment size 8 MB, each stream having 8 streamlets with Q configured to 4 (up to 4 producers/consumers per streamlet), backups would require $1000 \times 8 \times 4 \times 2 \times 8MB = 500GB$ to manage two open segments for each replicated log. Large streaming use cases (e.g., Twitter [96]) may require to ingest even more streams.

This limitation is imposed by the capacity of the servers batteries to safely flush this data to disk in case of power failures. While a replicated segment is filled, the backup writes it efficiently and asynchronously to disk, with minimal impact on the replication performance. Finally, the backup releases closed replicated segments. Each backup organizes replicated (physical) segments into logs, with one log file for each storage device. Backups maintain in memory the metadata necessary to identify the segments location in the backup logs. Each segment is equally striped over multiple disks in order to increase read/write IOs.

Replicating each group individually would dramatically reduce the number of streams that could actively, efficiently and durably ingest data. As such, we need a way to aggregate

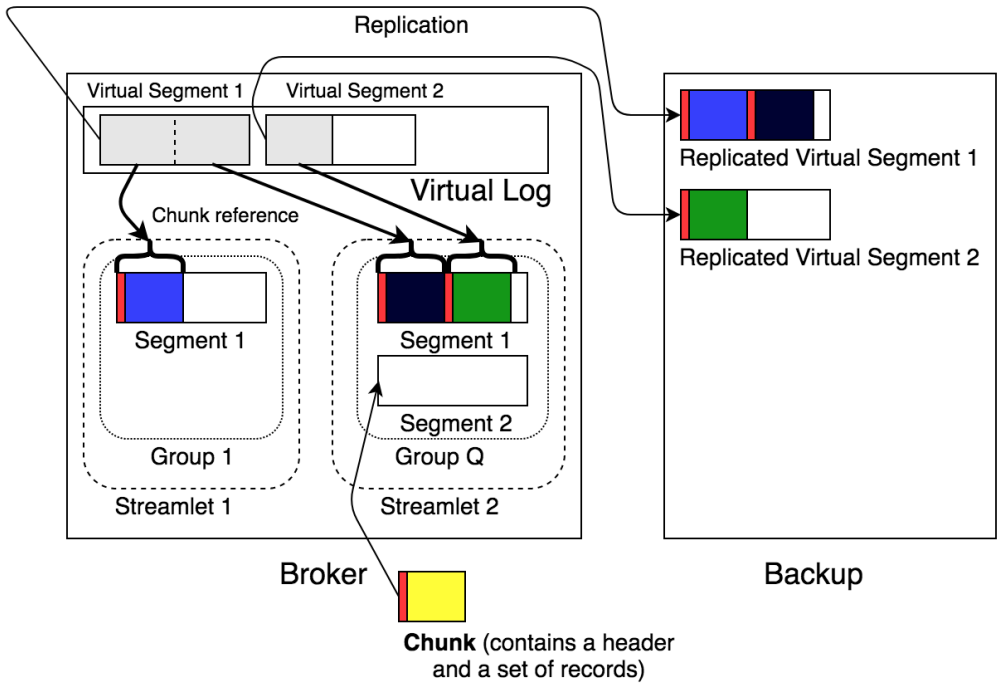

Figure 7.3 – The virtual log technique for adaptive and fine-grained replication. In this example a stream with 2 streamlets is created and managed by one broker, with one backup used to replicate the stream. We create and maintain in-memory a log-based data structure to represent our virtual log that aggregates chunks of different groups of streamlets of a single stream. Optionally, the stream can be configured with more virtual logs that can be associated to its streamlets.

the chunks of a stream's groups before replicating them in order to: (1) increase the number of active streams the system can efficiently support (reducing the number of utilized backup segments) and (2) increase the ingestion throughput for small streams by replacing small IOs with large ones on backups.

Our proposal is to aggregate acquired chunks at the stream level and organize them into one or multiple virtual logs per stream. Therefore, in order to simplify metadata management we leverage the chunk metadata that already is present when appending the chunk to the group's open segment (such metadata is also necessary for handling producer crashes and avoiding appending the same chunk twice or ordering inconsistencies). Each replication RPC should also replicate chunks atomically, not necessarily in the order of their creation. As such, backups should not be aware of the streamlet groups that are replicated in order to reduce the metadata overhead: instead, the backup is provided with a virtual log identifier along the broker and stream identifiers. To this end, we next introduce the virtual log technique.

7.3.2 Our proposal: virtual logs

In order to alleviate from previous limitations we introduce the concept of virtual log. Each stream (composed of a number of streamlets) can be associated with one or multiple virtual logs that are responsible to organize stream's chunks for replication. Therefore, each virtual

log is composed of an increasing number of virtual segments. The virtual segment keeps references to chunks that are physically part of different streamlets. The virtual log replicates its virtual segments to possible different backups.

An example is illustrated in Figure 7.3: a virtual log with 2 virtual segments is used to replicate one stream. The virtual log is composed of virtual segment 1 that is closed and fully replicated on the backup (see replicated virtual segment 1) and a virtual segment 2 that is open and has a reference to a chunk (see pointer to segment 1 of the group Q under streamlet 2) with its corresponding data already replicated on the backup (see replicated virtual segment 2). Assume we have a new chunk of data to be appended to the group Q of the streamlet 2. Since the segment 1 has no remaining space for this chunk, a new segment 2 is created and the chunk is physically appended to it. Then a reference to this chunk is appended to the virtual log: since the virtual segment 2 is open and has enough space (virtually) to hold this chunk, the chunk reference is saved. Perhaps, in parallel there are other chunk references saved by segment 2; one of the write requests will trigger the replication of the available chunks and will store them on backup's replicated segment 2. The virtual segment only stores chunk metadata and calculates its remaining virtual space based on the accumulated chunk lengths (the virtual log could allocate a buffer similar to the physical segment for the storage of a large set of virtual segments).

On each broker we associate each stream with one or multiple virtual logs (this configuration is exposed to users). Since each stream is logically partitioned into multiple streamlets, in case we want to maximize the replication throughput of very large (high throughput) streams, each streamlet can be associated with one or more virtual logs, up to Q logs. Therefore, the virtual log technique generalizes the multi-log approach with one important distinction: streamlets and virtual logs are distinct entities being represented by different data structures for two reasons. First, the streamlet can have one or more virtual logs or even share a virtual log with other streamlets. Second, in order to efficiently ensure elasticity we may decide to move streamlet metadata to other brokers and reference old stream data to initial brokers (see previous chapter).

For small streams (low ingestion throughput) users can configure a single virtual log per stream (per broker) that is shared by all streamlets. In contrast to the multi-log approach, we provide users a customized approach which can reduce backup storage requirements, enabling the support of more streams, at the cost of little metadata (chunk references). Moreover, the virtual log technique can be leveraged to efficiently implement throttling mechanisms for certain streams. For instance, in the context of enabling state management for Big Data analytics, operator sub-tasks seen as a stream can benefit from such aggregation under the virtual log implementation.

Each virtual log is composed of a set of virtual segments to be replicated, always a single open virtual segment (the replication of the virtual log resembles the RAMCloud log implementation). Each virtual segment contains references to the chunks appended to the group's physical segments. We only store the chunk header with a reference to the physical segment containing the chunk's records. The virtual segment has a header with the stream id and a checksum that covers the chunk's checksums. This information is used by backups for recovery and data integrity. The virtual segment also keeps two attributes to denote the next available/free offset (called header) and an attribute that points to what was already durably replicated (called durable header) – similar attributes are kept for each physical segment. The virtual segment does not hold any data but keeps a list of references to chunks

to be replicated; for each chunk a reference to the physical segment is kept along with the chunk's offset and length. Since the reference to the physical segment can be updated (by the persistence manager), we store it once in an array and keep the position associated with its chunk. The replication implementation ensures that each chunk is atomically replicated thus the durable header points to the next chunk to be replicated. After a chunk is replicated, the virtual log implementation updates the durable head of the physical segment so that consumers can pull records up to the durable head.

The broker associates physical segments with logical segments part of a group of a streamlet. Each closed physical segment can be possibly written to disk and released by the persistence manager in order to reuse its space. However, before removing a physical segment from memory, we must ensure its chunks are already replicated in order to avoid an invalid reference to this segment stored by the virtual segment handling at least one of its chunks. Thus, after replicating a chunk through a virtual segment, we also update the durable head offset of the physical segment. We use this field to decide when it is safe to push this physical segment to disk and release its space. We always keep in memory the virtual log metadata (i.e., chunk references maintained by virtual segments): if the broker crashes, another broker recreates lost groups, including the virtual log metadata, based on the chunk's metadata stored on backup's segments.

Since backups may crash, brokers may need to re-replicate virtual segments. A virtual segment points to multiple physical segments that may not be available in memory. As such, we need a way to handle invalid references to physical segments. For each physical segment we maintain a list of references to the virtual segments that replicate their chunks. Recall that a virtual segment may replicate chunks from multiple physical segments corresponding to different groups of a stream. When we write a physical segment to disk and release its memory, we invalidate its reference kept by each associated virtual segment (however, the physical segment id does not change). Later, if a virtual segment needs to be re-replicated, we read back corresponding physical segments and update their references. Each streamlet maintains the metadata necessary to identify its groups. For each logical segment a physical segment identifier is kept. However, streamlets do not keep references to physical segments (except for managing the head segments of the active groups).

7.4 Pushing processing to storage: enabling locality support for streaming

In this section we describe the implementation of an architectural extension based on shared memory techniques that allow streaming source operators to leverage data (in-memory) locality support and access stream data at lower latency and potentially higher throughput. We first present the current architecture for managing streaming source operators and describe its potential performance limitations that can further lead to higher latency and lower throughput.

7.4.1 State-of-the-art architecture: pull-based consumers

Big Data real-time stream processing typically relies on message broker solutions that decouple data sources from applications. This translates into a three-stage pipeline described

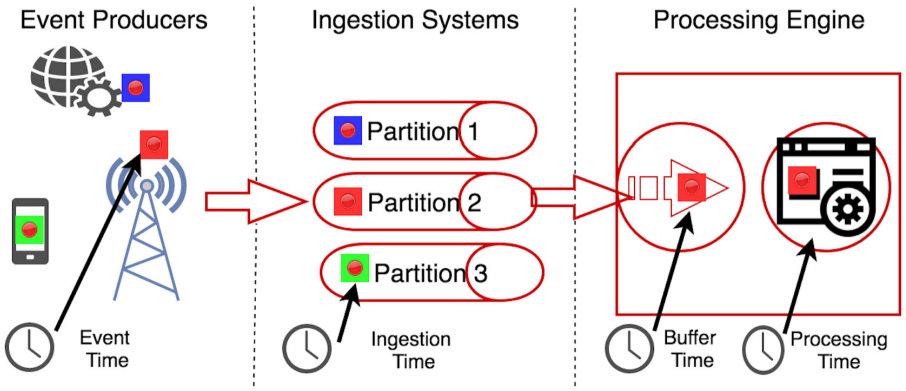

Figure 7.4 – Current streaming architecture with pull-based consumers: records are collected at ***event time*** and made available to consumers earliest at ***ingestion time***, after the events are acknowledged by producers; processing consumers continuously pull these records and buffer them at ***buffer time***, and then deliver them to the processing operators, so results are available at ***processing time***.

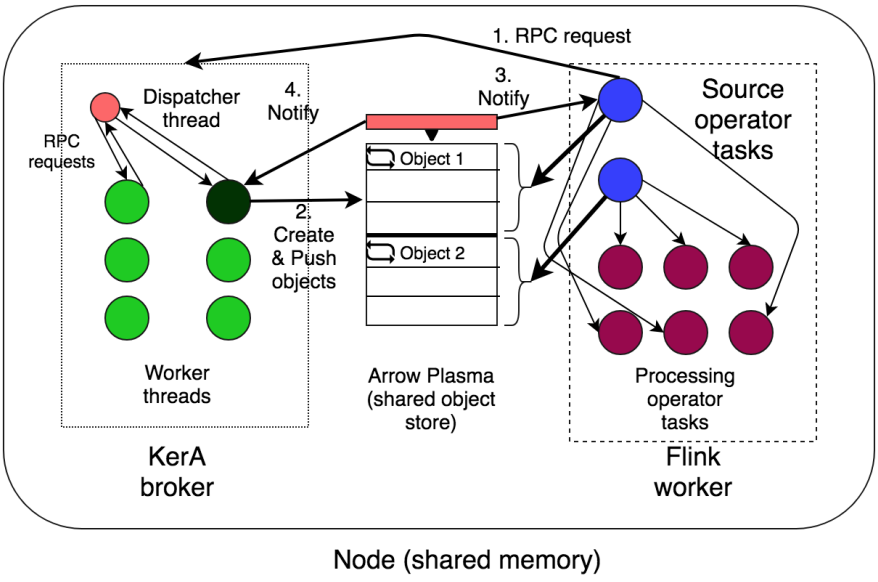

Figure 7.5 – Data locality architecture through shared in-memory object store. On the same node live three processes: the streaming broker (KerA), the processing worker (Flink) and the shared object store (Arrow Plasma). Source tasks coordinate to launch one RPC request (step 1). The worker thread is responsible to fill shared objects with next stream data (step 2). Source tasks are notified for object updates (step 3) and process new stream data. The worker thread is notified (step 4) after each source processed all objects, so a new 'iteration' for that source can be started. This flow executes continuously.

in Figure 7.4. First, in the ***production*** phase, event sources (e.g., smart devices, sensors, etc.) continuously generate streams of records. Second, in the ***ingestion*** phase, these records are acquired, partitioned and pre-processed to facilitate consumption. Finally, in the ***processing*** phase, Big Data engines consume the stream records using a ***pull-based*** model. Since users are interested in obtaining results as soon as possible, there is a need to minimize the end-to-end latency of the three stage pipeline. This is a non-trivial challenge when records arrive at a fast rate (from producers and to consumers) and create the need to support a high throughput at the same time.

To this purpose, Big Data streaming engines are typically designed to scale to a large number of simultaneous pull-based consumers, which enables processing for millions of records per second [111, 74]. Thus, the weak link of the three stage pipeline is the ingestion phase: it needs to acquire records with a high throughput from the producers, serve the consumers with a high throughput, scale to a large number of producers and consumers, and minimize the write latency of the producers and, respectively, the read latency of the consumers to facilitate low end-to-end latency. Since producers and consumers communicate with message brokers through RPCs, there is inevitably interference between these operations which can lead to increased processing times. Moreover, since consumers (i.e., source operators) depend on the networking infrastructure, its characteristics can limit the read throughput and/or increase the end-to-end read latency. One simple idea is to co-locate processing workers (source and other operators) with brokers managing stream partitions. We further describe the implementation of our approach.

7.4.2 Leveraging locality: push-based consumers

We illustrate in Figure 7.5 our proposal for adding locality support for streaming operations. A Big Data source operator can be scheduled on a set of worker nodes and configured with a parallelism that allows one or more source tasks to run on each worker. Other Big Data operators (including sinks) are similarly scheduled and connected with sources. Each source task is implemented with one thread responsible to build the first request according to its parameter settings (that include initial stream offsets) and process next chunks of data. We implement a shared-memory object-based store based on Apache Arrow Plasma [77], a framework that allows the creation of in-memory buffers and their manipulation through shared pointers.

Our sharded object store is leveraged by all local source tasks of a worker as follows. We partition the object-based store into objects that give access through a pointer to their memory. The object pointer is used by both broker and sources based on notifications. Local sources synchronize so that only one RPC request is sent to the broker and one worker thread implements the request as a normal consumer source (managing offsets internally). The broker then pushes chunks through shared objects and notifies sources when each object is updated (object buffers are reused). Once a local source processes its objects, it notifies the broker to push more chunks.

Integration of KerA and Flink. Kera consists of about 10K lines of C++ code for client and server side implementations and 3K lines of Java code for the integration with Apache Flink. Although the current KerA prototype does not implement crash recovery nor migration techniques to support elasticity, its design and current implementation efficiently allow the implementation of such features (as described in previous chapter) that are left for future

work.

Chapter 8

Synthetic evaluation

Contents

In this chapter we focus on the evaluation of our KerA prototype: we evaluate the impact of dynamic ingestion, replication and locality techniques on the performance of producers and consumers running concurrently over multiple brokers. We first execute a set of synthetic micro-benchmarks by tuning a set of important parameters in order to assess the baseline performance (measured as ingestion and processing throughput) of a single producer and single consumer. Then, in a distributed setup with multiple brokers and concurrent producers and consumers, we compare KerA and Kafka while focusing on the ingestion

component (with replication disabled) in order to show the effectiveness of KerA's dynamic stream partitioning model and the lightweight offset indexing techniques. Further, we explore the efficiency of the virtual log replication implementation and evaluate the impact of various configurations on ingestion and processing throughput. Finally, we evaluate how the locality architecture impacts performance of streaming reads and durable writes due to reduced interference between read, write and replication RPCs.

8.1 Setup and parameter configuration

We ran all our experiments on Grid5000 ***Paravance*** cluster [35]. Each node has 16 cores and 128 GB of memory. We recall the reader the KerA architecture: as illustrated in Figure 6.4, a single layer of brokers serve producers and consumers. In our experiments we install on each node a broker service (each broker has an ingestion component offering RPC interfaces to stream clients) and/or a backup service that is used for storing stream's replicas. Since the current architecture allows the separation of nodes serving clients from nodes serving as backups, we leverage this feature to isolate backups from brokers when studying the replication impact. Brokers also expose through RPCs the metadata describing stream's partitions (i.e., streamlet's groups of segments).

Each broker is configured with 16 threads that correspond to the number of cores of a node and holds one copy of the streamlet groups. In each experiment we run an equal number of concurrent producers and consumers. The number of streamlets is configured to be a multiple of the number of clients, at least one for each client. When replication is enabled, we configure one or more virtual logs that manage the streamlets replication while consumers only pull durably replicated data.

Unless specified, we configure in KerA the number of active groups to 1 (i.e., streamlets are configured similar to partitions in systems such as Apache Kafka) and the number of segments to 16 (the segment size is configured to 8 MB, the total size of a closed group being 128MB, similar to the default block size in HDFS; this parameter is configurable and can be leveraged in production systems to allow more consumer parallelism). A request is characterized by its size (i.e.., ***request.size***, in bytes) and contains a set of chunks, one for each streamlet, each chunk having a ***chunk.size*** in kilobytes. We use Kafka 0.10.2.1 since it has a similar data model with KerA (the newest release introduces batch entries for exactly-once processing, a feature that we plan to enable and evaluate also in KerA based on RAMCloud linearizability technique [62]). A Kafka segment has 512 MB, while in KerA it is configured to 8 MB (i.e., record size is bounded by this configuration), a configuration that supports a large number of streams. This means that the creation of a new segment (needed by streamlet groups /sub-partitions in order to ingest more data) happens more often and may impact performance (also because KerA's clients need to discover new segments before pulling data from them). Each experiment further details additional parameters necessary to understand their implications on performance.

In each experiment the source thread of each producer (as illustrated in Figure 8.1) creates up to 100 million non-keyed records of 100 bytes, and partitions them in a round-robin way in chunks of a configurable size. The source waits no more than 1ms (parameter named ***linger.ms*** in Kafka) for a chunk to be filled, after this timeout the chunk is sent to the broker. Another producer thread aggregates chunks in requests (one request for each broker)

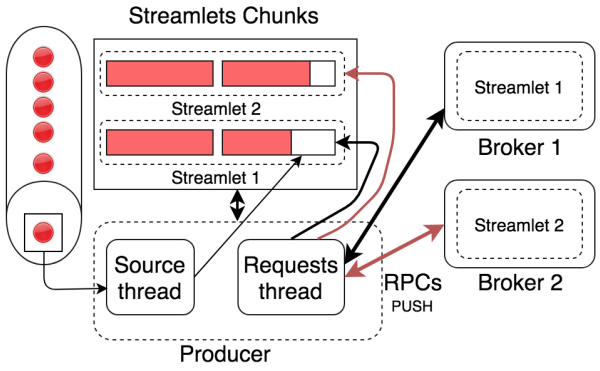

Figure 8.1 – Producer architecture. The Source thread appends records to in-memory chunk buffers. The Requests thread batches one chunk for each streamlet (if the chunk is filled or a configurable timeout per chunk passed) in requests and pushes them to brokers.

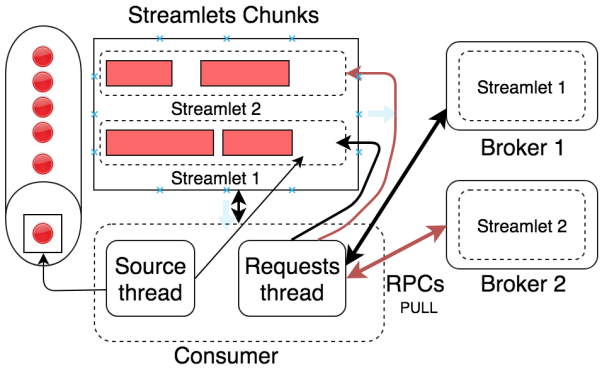

Figure 8.2 – Consumer architecture. The Requests thread builds one request for each broker and pulls one chunk for each streamlet associated to the consumer. The Source thread consumes in-order one chunk per streamlet: it iterates the chunk and creates records. The source could push these records to other streaming operators for further processing.

and sends them to the broker (one or multiple synchronous TCP requests). Similarly (as illustrated in Figure 8.2), each consumer pulls chunks of records with one thread and simply iterates over serialized records on another thread. Each client has a cache of up to 1000 chunks. In the client's main thread we measure ingestion/processing throughput (e.g., million records per second) and log it after each second. Producers and consumers run on different nodes. We calculate the average ingestion/processing throughput per producer/-consumer for 60 seconds worth of experiments, with 50 and 95 percentiles computed over all clients measurements taken when concurrently running all producers and consumers (without considering the first and last five seconds measurements of each client).

Producers are configured as ***proxy clients*** and share the streamlets: they compete when producing stream records (since each request contains a chunk for each streamlet). When the number of active groups is higher than 1, appends on brokers can happen in parallel. Proxy-based producers represent the worst-case scenario and most streaming use cases are implemented similarly (each RPC request contains one chunk for each partition in order to target similar record latencies). Another option is to configure one producer for each streamlet:

this brings no competition between producers and can be used by certain HPC simulations where each simulation core is configured to produce data partitioned by its number.

We choose the first option since it represents real-world scenarios (e.g., streaming sink operators are similar to proxy-based producers) while it brings more pressure on broker implementation (e.g., due to competition between RPCs, concurrency between appends). Moreover, producers aggregate chunks in a single request in order to reduce the overhead due to TCP requests, while reducing latency processing of records part of different streamlets living on the same broker. One alternative for the producer is to issue one request with one chunk for each streamlet. Now imagine the producer is configured to manage multiple streams each with tens of streamlets. Obviously, sending hundreds more RPCs could increase the latency of certain records that arrive at the producer at the same time with records already pushed to brokers (worst case most RPCs will suffer a latency penalty equal to the previous RPCs due to reduced broker resources). Thus, we batch more chunks in a request, up to a certain size that we evaluate in different experiments. Consumers are configured to pull data exclusively from their associated streamlets. Consumers also pull more chunks in a request if there are more streamlets on a broker.

8.2 Baseline performance of a single client: peak throughput

We first run a set of micro-benchmarks to assess the performance of a single producer and single consumer running (standalone or concurrently) over a single broker while we vary a set of important parameters that can impact ingestion and processing throughput:

The number of streamlets associated to one producer/consumer impacts the ingestion/processing throughput due to higher request size and increased concurrency.

The chunk size is important to understand the ingestion throughput limitations, being an important bottleneck of the producer source (it is also used by clients to trade-off throughput versus latency).

The checksum computation over records values can largely impact the producer source throughput.

The number of in-flight RPCs (TCP requests) is characteristic to high-performance pipelined producers.

While in this section we experiment with all four parameters to show the performance peak of our implementation, in the next sections we only consider the first three parameters. Increasing the number of in-flight RPCs can bring better ingestion throughput at the cost of losing record ordering. We leave the evaluation of pipelined producers to future work as it depends on specific use cases.

8.2.1 Producers: how parameters impact ingestion throughput

As illustrated in Figure 8.3, we experiment with a single producer running TCP requests over one broker, considering the parameters previously introduced. We observe that increasing

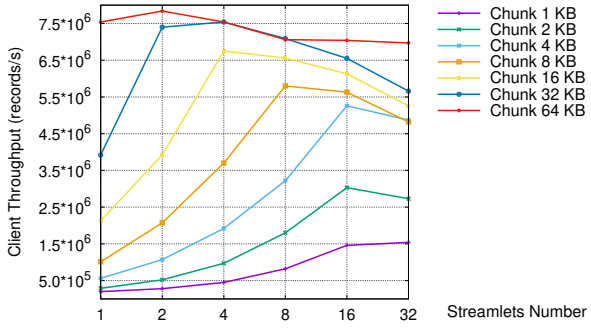

Figure 8.3 – Single producer throughput (measured in million records per second) when increasing the chunk size and respectively the number of streamlets. The request size equals the chunk size multiplied by the number of streamlets. The pipelined producer (5 TCP requests) does not compute the record checksum.

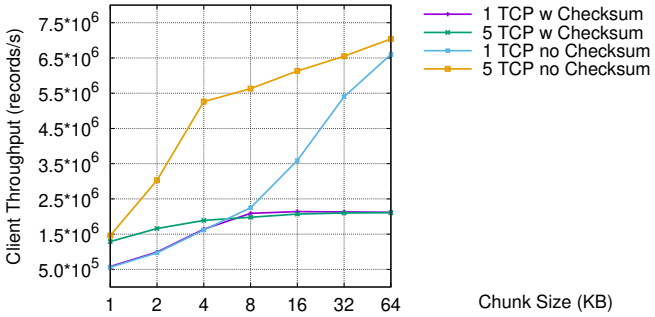

Figure 8.4 – Impact of source record checksum computation on producer throughput (stream configured with 16 streamlets, thus, the request size is 16x the chunk size).

the request size (either by increasing the chunk size for a single streamlet or by increasing the number of streamlets) brings substantially higher throughput. With more streamlets we also observe a slight decrease in throughput since the source is not able to fill chunks with more records due to the 1ms timeout in which chunks can be filled before sending them to brokers. Additionally the chunk headers contribute to less batched records. These experiments also show that the source can be highly limited when enabling the record checksum computations (see Figure 8.4): the source generates in real-time the stream of records and appends records one by one to the streamlet chunks; then, once chunks are filled or after a 1ms timeout (for lower latency), the producer aggregates chunks in requests and sends them to the broker. Another important aspect is pipelining the producer (i.e., allowing more requests over the same broker that may execute in parallel) since it allows continuously appending to the broker. However, when checksuming is enabled, the source limits the producer throughput by a factor of 3 and pipelining is less important with a single source thread.

8.2.2 Consumers: how parameters impact processing throughput

As illustrated in Figure 8.5, we experiment with a single consumer running in parallel with one producer and sending TCP requests over the same broker, considering the parameters previously introduced. Each consumer (not pipelined) issues one synchronous TCP request

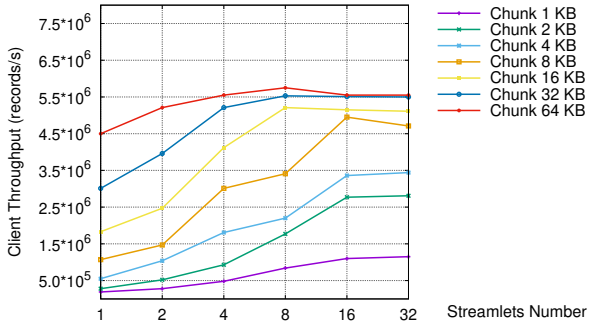

Figure 8.5 – Single consumer throughput (measured in million records per second) when increasing the chunk size and respectively the number of streamlets. The request size equals the chunk size multiplied by the number of streamlets. The pipelined producer (5 TCP requests) does not compute the record checksum.

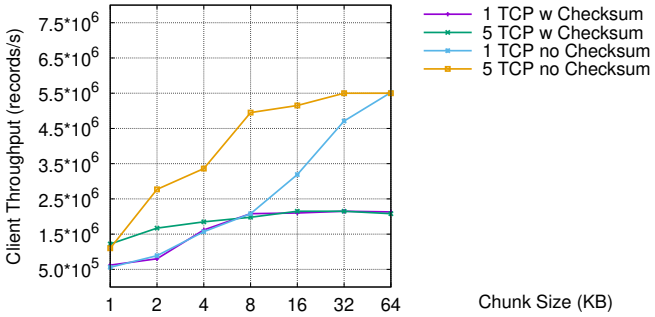

Figure 8.6 – Impact of source record checksum computation on consumer throughput (one producer with one consumer running in parallel).

in order to pull one chunk for each streamlet (the consumer chunk size is five times the producer chunk size to better manage the pipelined producer sending more data). The consumer creates a set of records by parsing and iterating the chunk records. The producer throughput results are similar to measurements previously presented. Although the consumer is not pipelined, it can pull up to 5.9 million records per second compared to maximum 7.8 million records per second obtained by a pipelined producer. When records are checksumed, the consumer is able to keep up with the producer (see Figure 8.6).

8.3 Impact of dynamic ingestion on performance: KerA versus Kafka

While Kafka provides a static offset-based consumer access by maintaining and indexing record offsets (at append time), KerA proposes dynamic access through application defined offsets that leverage the ***[streamlet,group,segment]*** metadata, therefore, avoiding the overhead of offset indexing on brokers. In Kafka each partition is composed of a set of segments. Kafka maintains for each segment an index file (memory mapped) that maps offsets to their record position in the segment log file. In Kafka, the index file is updated for each partition ***append***. Kafka's consumers also have to go through the index file in order to pull data at

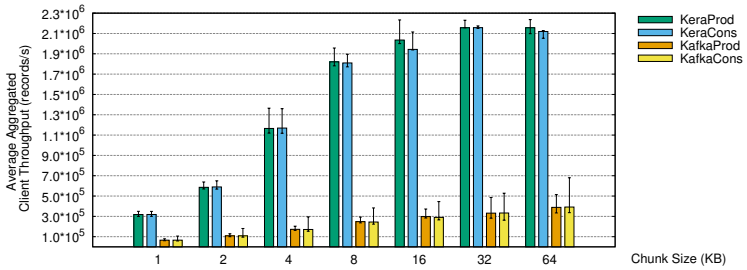

Figure 8.7 – Evaluating the ingestion component when increasing the chunk/request size. Parameters: 4 brokers; 16 producers running concurrently with 16 consumers; number of partitions/streamlets is 16, 1 active group per streamlet; ***request.size*** equals ***chunk.size*** multiplied by 4 (number of partitions/streamlets per broker). On X we represent producer ***chunk.size*** (KB), for consumers we configure a value 16x higher.

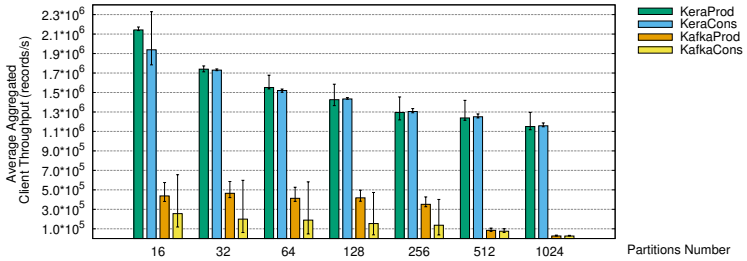

Figure 8.8 – Evaluating the ingestion component when increasing the number of partitions/streamlets. Parameters: 4 brokers; 16 producers and 16 consumers; ***request.size*** = 1 MB; ***chunk.size*** equals ***request.size*** divided by the number of partitions per broker.

a specified offset. In KerA there is no offset overhead like in Kafka. Streamlet groups and segments are created in order and their identifiers are part of the consumer logical offsets. Consumers remember the last record position (in a given logical segment) and specify it to the next RPC request. Therefore, consumers and producers in KerA do not have to interact through any additional offset indexing mechanism like in Kafka. In order to understand the application offset overhead in Kafka and KerA, we evaluate different scenarios, as follows.

8.3.1 Impact of the chunk size: a throughput versus latency trade-off

Batching more data is a well known technique for increasing ingestion/processing throughput. At the same time it represents a good trade-off between throughput and latency (since producers may have to wait for more records to fill-up chunks before pushing the next requests). By increasing the chunk (batch) size we observe smaller gains in Kafka than in KerA (Figure 8.7). KerA provides up to 7x higher throughput when increasing the batch size from 1KB to 32KB, after which the throughput is limited by that of the producer's source. For each producer request, before appending a chunk to a partition, Kafka iterates at runtime over chunk's records in order to update their offset, while KerA simply appends the chunk to the group's active 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. These operations could further be pipelined with read RPCs (eliminating interference with normal requests) or simply eliminated as we later discuss in the locality

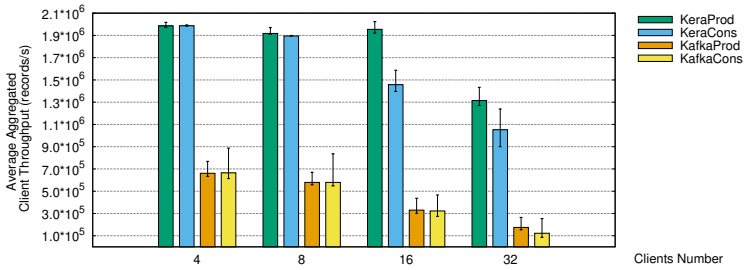

Figure 8.9 – Vertical scalability: increasing the number of clients. Parameters: 4 brokers; 32 partitions/streamlets, 1 active group per streamlet; ***chunk.size*** = 16KB; ***request.size*** = 128KB.

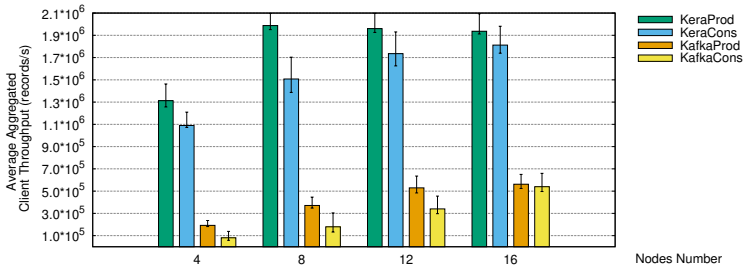

Figure 8.10 – Horizontal scalability: increasing the number of nodes (brokers). Parameters: 32 producers running concurrently with 32 consumers; 256 partitions, 32 streamlets with 8 active groups per streamlet; ***chunk.size*** = 16KB; ***request.size*** = ***chunk.size*** multiplied by the number of partitions/active groups per node.

evaluation section.

8.3.2 Validating horizontal and vertical scalability

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 illustrated in Figure 8.9, when running up to 64 clients on 4 brokers (full parallelism), KerA is more efficient in front of higher number of clients, up to one order of magnitude better throughput due to KerA's more efficient application offset indexing. Producers appends execute faster (no need to maintain offsets) allowing for more ingestion requests. Consumers pull data faster due to reduced indexing overhead. Overall, we observe more efficient broker resource utilization resulting in higher ingestion and processing throughput.

Although the number of brokers is limited to 64 clients, the aggregated throughput obtained with KerA's clients is about 30 million records/s concurrently acquired and processed. This is more than enough to respond to most critical Big Data applications. Since we are limited by the number of nodes in our cluster we could not test with larger configurations (e.g., 16/32 brokers with 256/512 clients) as clients are deployed separately. However, if we consider that producers are configured similarly (sharing all streamlets), then, in a larger distributed setup, although the number of RPCs would increase, the request size would be smaller. In this situation KerA would benefit even more from its architecture designed for serving fast a large number of small requests. More consumers would equally share stream-

lets, with similar impact on performance (as number of RPCs per broker). We leave this evaluation to future work as it depends on real use cases datasets that would benefit from such large setups.

Adding nodes: horizontal scalability. Since clients can leverage multi-TCP (over a set of brokers), distributing partitions/streamlets on more nodes (1 broker process per node) helps increasing throughput. As presented in Figure 8.10, even when Kafka uses 4x more nodes, it only delivers half of the KerA performance. Similarly to Kafka, KerA producers prepare a set of requests from available chunks (those that are filled or those with the timeout expired) and then submit them to brokers, polling them for answers. Only after all requests are executed, a new set of requests is built. As seen in the baseline experiments, pipelining the producer helps to further optimize throughput and allows the submissions of new requests while older ones are processed. However, the pipelined implementation makes it difficult to ensure record ordering (since streamlet chunks may be acquired by different worker threads of the same broker which may execute in different order). We leave this setup exploration for future work.

8.3.3 Impact of the number of partitions/streamlets

Finally, we seek to assess the impact of increasing the number of partitions/streamlets on the ingestion throughput. When the number of partitions is increased we also reduce the ***chunk.size*** while keeping the ***request.size*** fixed (1MB) 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 8.8 that when increasing the number of partitions the average throughput per client decreases (as expected). Kafka's drop in performance (up to 30x for 512 and 1024 partitions) is mainly due to its offset-based implementation, having to manage one offset index file for each partition segment. Producer appends additionally result in offset operations (for each partition segment) that have to synchronize, reducing considerably the ingestion throughput. Offset-based (lookup) operations are also needed by consumers before serving each pull request. In fact, Kafka's users over-configure the number of partitions by choosing a large number of brokers to host them upfront. However, each Kafka broker can optimally host a number of partitions per stream proportional to the number of cores (2x-3x).

With KerA users can leverage the streamlet concept (which semantically subsumes the Kafka partition) in order to provide applications an unlimited number of fixed-size subpartitions (groups of segments) for better performance. Therefore, KerA provides higher parallelism to producers and consumers resulting in higher ingestion/processing throughput than in systems such as Apache Kafka.

8.3.4 Discussion

Apache Kafka and other similar ingestion systems (e.g., Amazon Kinesis [52], MapR Streams [70]) provide publish/subscribe functionality for data streams by statically partitioning a stream with a fixed number of partitions. To facilitate future higher workloads and better consumer scalability, streams are over-provisioned with a high enough number of partitions.

In contrast, KerA's ingestion component enables resource elasticity by means of streamlets, which enables storing an unbounded number of fixed-size sub-partitions. Furthermore, to alleviate from unnecessary offset indexing, KerA's clients dynamically build an application offset based on streamlet-group metadata exposed through RPCs by brokers. None of the state-of-the-art ingestion systems is designed to leverage data locality optimizations as considered by KerA. Moreover, thanks to its network agnostic implementation [86], KerA can benefit from emerging fast networks and RDMA, providing more efficient reads and writes than using TCP/IP.

Lower-level measurements. KerA's implementation (we mostly refer to the dynamic ingestion component) benefits not only from its high-level design that proposes dynamic partitioning and lighweight offset indexing, but also from its high-performance C++ implementation. Although KerA relies on a dispatcher threading architecture optimized for low-latency requests [86], our experiments prove that throughput-oriented workloads can also benefit from this design principle: in comparison with Kafka, KerA brings up to one order of magnitude better performance under high-load (due to increased clients parallelism). However, Kafka's implementation is based on Java/Scala, relying on expensive offset-based operations, while its underlying architecture leverages the operating system kernel cache. Moreover, Kafka's clients are implemented in Java, therefore, Java serialization can impact performance, resulting in reduced ingestion throughput (although Java's garbage collector can help better manage the creation and destruction of millions of objects). In order to better understand previous (possible) bottlenecks, in future work we will consider in KerA the implementation of Kafka-like offset management (broker and client) interfaces. In parallel, we aim to add low-level instrumentation to both KerA and Kafka implementations in order to better understand the impact of different components (e.g., network transport, record serialization). Since KerA implements a customized memory management, we hope to learn from these one level deeper measurements [85] and then improve our KerA prototype implementation, while better emphasizing different design principles that should be considered by system designers in their work.

8.4 Understanding the impact of the virtual log replication

In this section we explore the efficiency of the virtual log replication under different configurations.

8.4.1 Baseline

For the next experiments the producer source does not compute the record checksum before pushing chunks to brokers in order to put more pressure on replication. Producers are not pipelined in order to observe the replication impact. An equal number of consumers run in parallel with producers in order to pull durably replicated data. For each configuration we plot the average throughput per client. A stream is configured with 16 streamlets on 4 brokers backed by 8 backups each (the backup is configured with 1 HDD disk on cluster paravance). We leave backups separated from brokers to effectively isolate the interference between normal client requests and replication RPCs, so we can better observe the replication impact.

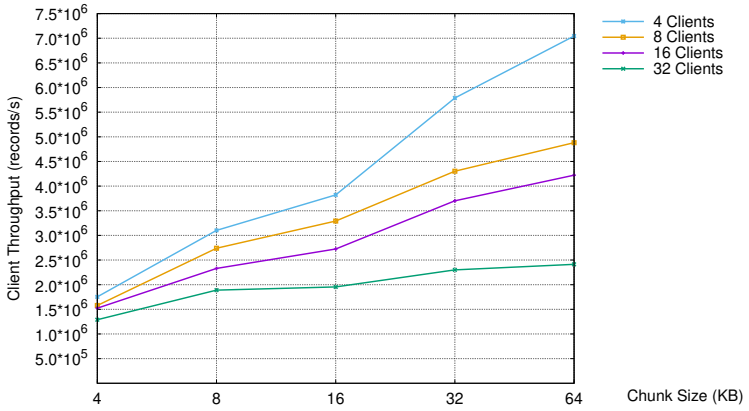

Figure 8.11 – Baseline: average throughput per client with replication disabled. The configuration with 16 clients (8 producers running in parallel with 8 consumers) corresponds to maximum broker requests parallelism. Increasing the number of clients results in increased competition between producers thus reduced ingestion and processing throughput per client.

We vary the chunk size, the number of active groups of a streamlet and the number of virtual logs shared by all streamlets. Our goal is to understand the impact of these configurations while increasing the replication factor (up to 3 copies including the broker). Producers share all streamlets, building requests that contain one chunk for each streamlet. Therefore, each producer submits a number of requests equal to the number of brokers over multi-TCP connections. In parallel, the same number of TCP requests is executed by each consumer, competing on broker resources.

We first run a set of experiments with replication disabled. Experiments configured with 16 clients (8 producers and 8 consumers) correspond to the maximum parallelism. We double the number of clients to 32 in order to put more pressure on brokers, trying to understand how the system behaves for more critical workloads. These situations normally trigger streamlet migration and it is important for brokers to efficiently respond to higher workloads before scaling the system. As illustrated in Figure 8.11, doubling the number of clients results in less-than-half decreased throughput, while consumers (not shown in figure) manage to keep up with producers. The aggregated throughput is about 40 million records/s acquired and concurrently processed by 16 and respectively 32 clients running requests over TCP with 4 brokers (64 cores). Considering that producers compete when appending data, these results correspond to a worst case scenario (we configure 1 active group per streamlet which corresponds to maximum concurrency on producer appends).

For the next experiments we increase the streamlet active groups and the number of virtual logs, while increasing the replication factor, and then we compare results with the baseline for 16 and 32 clients. While we expect to obtain reduced performance (i.e., lower ingestion and processing throughput), it is important to understand the impact of these parameters on performance. First, we expect the streamlet active groups configuration would bring better throughput due to higher parallelism (faster broker appends). Second, we seek to understand how replication performance can be improved with more virtual logs (and how many we actually need for peak performance).

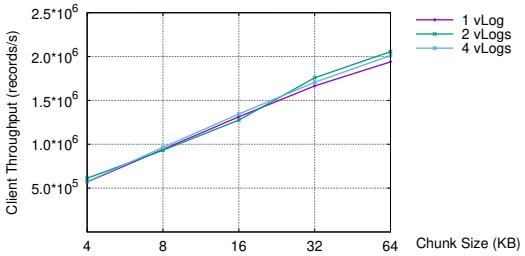

Figure 8.12 – Configuring more virtual logs (shared by streamlets) brings no advantage when 16 clients compete on streamlets with single active group (with replication factor 1).

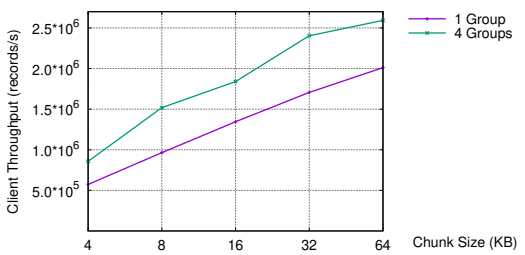

Figure 8.13 – Increasing the number of active groups brings increased throughput (replication factor 1). 16 clients compete on streamlets configured with 1 and 4 active groups, while 4 virtual logs are used for replication.

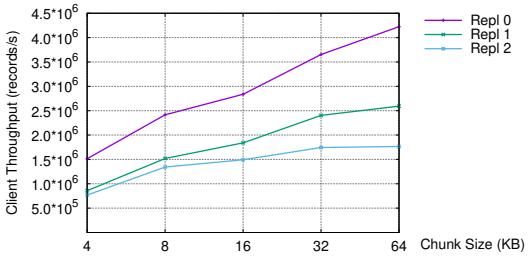

Figure 8.14 – Increasing the replication factor (16 clients).

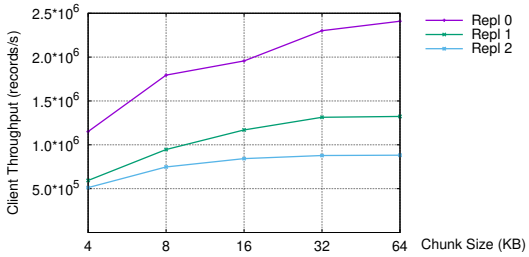

Figure 8.15 – Increasing the replication factor (32 clients).

8.4.2 Impact of the configuration of the streamlet active groups

In order to understand the impact of the configuration of the streamlet active groups on performance, we experiment with a single active group per streamlet while increasing the number of virtual logs for replication factor 1. As illustrated in Figure 8.12, when streamlets are configured similar to Kafka (no parallelism) there is no performance advantage in configuring an increased number of replicated logs (since producers are bottlenecked due to append synchronization). To further show the benefit of streamlets higher parallelism, we increase the number of active groups to correspond to the number of virtual logs used for replication. As can be seen in Figure 8.13, aggregated clients throughput increases from 16 to 20 million records per second due to reduced broker appends contention.

8.4.3 Impact of the replication factor

Further, we seek to understand ***how does the replication factor impact performance?*** We configure each streamlet with 4 active groups while using 4 virtual logs for replicating them. As illustrated in Figure 8.14, for smaller chunks we obtain similar results although we increase the replication factor from 1 to 2. Since producers are not pipelined, each client request has to wait for the replication requests to finish, reducing ingestion throughput. Overall, increasing the chunk size brings better throughput as expected. Increasing the replication factor brings lower throughput due to replication synchronization on logs. These effects are kept even when doubling the number of clients (Figure 8.15).

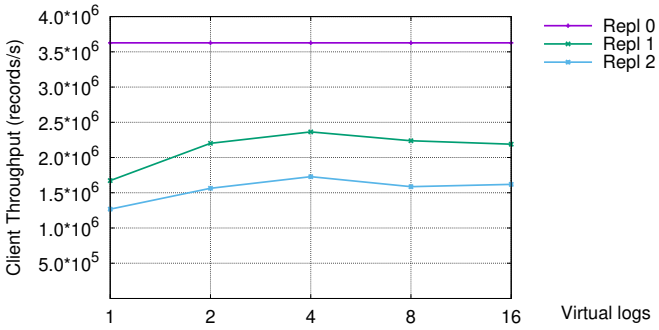

Figure 8.16 – Increasing the number of virtual logs (16 clients, 8 streamlets, 4 brokers). 16 virtual logs correspond to the total number of active groups on a broker (2 streamlets per broker multiplied by 8 active groups)

8.4.4 Increasing the number of virtual logs

Impact of backups configuration. The virtual log keeps (for each additional replica) a randomly selected open segment on one of the backups. Backups continuously and asynchronously write on disk data acquired by open segments. Since our experiments leverage HDDs, the disk may impact results (although they are scheduled with low priority, disk IOs may compete with in-memory appends). To understand the disk impact we run similar experiments for a stream configured with 8 streamlets each having 8 active groups, while increasing the number of virtual logs from 1 to 16 (we show the configuration of chunk size being 64 KB, the other configurations from 1 KB to 32 KB look similar). As illustrated in Figure 8.16, although more virtual logs may boost replication throughput, the backup disk can also be a bottleneck (configurations with 2, 8 and 16 virtual logs are similar, being slightly slower than the peak one with 4 virtual logs).

8.4.5 Discussion

Backed by these experiments, we emphasize the importance of streamlet configuration under replication settings: more active groups per streamlet bring higher parallelism leading to increased throughput. Moreover, being able to tune the replication settings (e.g., by adding more virtual logs for stream replication) allows users to increase the performance at the cost of additional in-memory storage on backups. Our experiments are configured so that they describe worst-case scenarios that are usually encountered in production systems (e.g., client and replication requests are built over TCP with backups configured with HDDs; non-pipelined producers that compete on all partitions when producing data).

In parallel with our work, an evaluation of KerA in a high-performance scenario was done by our colleague Yacine Taleb [100]. To this end, KerA was integrated with Tailwind [101], a high-performance replication protocol that uses one-sided RDMA instead of RPCs in order to leave more CPU resources to request processing. Tailwind was initially implemented and tested on top of RAMCloud, a high-performance low-latency key-value store; KerA leverages RAMCloud's RPC framework and its support for multiple networking transports (e.g., TCP, DPDK, Infiniband). Moreover, the replication was implemented with streamlets backed by multi-logs (i.e., one replicated log for each entry corresponding to one

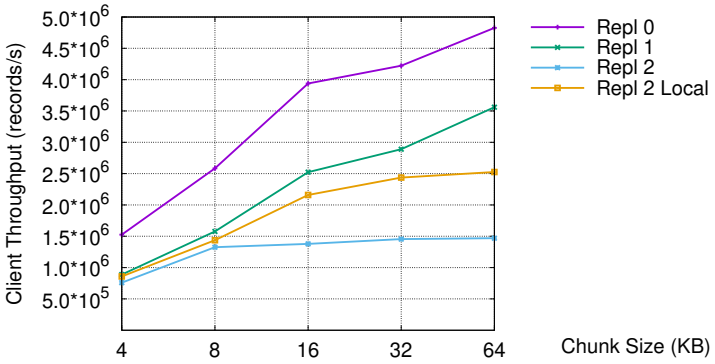

Figure 8.17 – Baseline for locality: increased replication factor brings more interference. Producers and consumers are running on separated nodes.

of the active groups, up to Q logs) in order to simplify fault-tolerance based on RAMCloud crash recovery techniques. Therefore, each streamlet (replicated) log is composed of a set of physical segments that also represent the logical segments of groups. The log replication is similar to that used by RAMCloud, except the fact that in this KerA setup there are multiple logs per broker. Experiments carried over Infiniband (leveraging RDMA-based replication), focused only on data writes, prove KerA can efficiently leverage high-performance networking hardware. Therefore, if data arrives very fast, the stream ingestion can be possibly bottlenecked by disk bandwidth, unless network bandwidth dominates. In future work, in order to better understand the impact of disk and network efficiency in KerA, we will analyze data- and network-intensive applications for workloads with concurrent reads and writes.

The virtual log technique generalizes the multi-log approach in order to provide users flexibility in tuning the ingestion/replication setup, trading backups in-memory storage for support for a larger number of streams, at the cost of little metadata (chunk references managed by virtual logs) and possibly similar performance (as previously seen, although increasing the number of virtual logs can bring higher throughput it can also lead to reduced or similar performance). Therefore, users can configure the streamlet to hold one virtual log for each entry, similar to the multi-log technique, at the cost of managing chunk references through the virtual log segments. In a future work we aim to recognize this configuration and rely on the multi-log simplicity by keeping in the virtual log segment a single reference to the logical segment, reducing chunk references metadata overhead.

We think that the virtual log support is extremely important for Big Data analytics: many large use cases (e.g., Netflix, Twitter, Facebook) require the management of a large number of streams, while Volume and Velocity remain important streaming attributes to consider. Our next steps are the integration of Tailwind with the virtual log technique and the evaluation of pipelined producers and consumers for various HPC and Big Data real-time use cases.

8.5 Going further: why locality is important for streaming

Next experiments evaluate the impact of locality on streaming performance. We have two configurations with local and remote consumers. The first configuration is similar to the one

used in the previous experiments: consumers pull data through normal RPCs (consumers are separated from producers and brokers/backups). The second configuration brings data locality support: consumers and brokers share the same nodes and communicate through shared memory (as explained in previous chapter). Backups and brokers also live on the same nodes sharing resources between normal client requests and replication RPCs.

As illustrated in Figure 8.17, when running 16 concurrent clients through normal RPCs, increasing the replication factor brings reduced throughput due to more competition between normal requests and replication RPCs. When deploying local consumers we observe the average client throughput improves up to 60%. This is explained by the fact that consumer RPCs are replaced by a single RPC at the beginning of the consumer creation, leaving broker resources to normal producer requests and backups replication. Replacing consumer RPCs with one dedicated worker thread that continuously pushes records to consumers through shared memory helps reducing the interference with producer requests. These improvements can be thought as similar to optimizations brought by techniques such as Tailwind: less interference on dispatcher and worker threads leaves more CPU space for executing producer and backup requests which translates into more processing throughput.

After integrating Tailwind and Arachne (a user-level thread implementation, see previous chapter), our future goal will be to understand how locality additionally impacts performance. Our intuition is that with the implementation of the three techniques the remaining CPU is enough to manage processing analytics without impacting normal ingestion, making more effective the integration of analytics in a unified architecture for ingestion/storage towards efficient processing. We leave this study for future work.

Part IV

Conclusion and Future Work

Chapter 9

Final words

Contents

Large web applications (e.g., Facebook, Twitter, Amazon, Alibaba, etc.) have to deal with an unprecedented growth of data that need to be acquired, stored and processed at always increasing rates in order to extract valuable insights. In this context, modern applications (such as IoT) demand more efficient support for real-time data ingestion and persistent storage.

In order to cope with these trends, Big Data processing architectures rapidly evolved to support batch, interactive and streaming computations in the same runtime, optimizing applications execution by removing work duplication and resource sharing between specialized engines. Processing engines depend on specialized ingestion and storage systems that provide the required data access interfaces. Basically, the current trend in Big Data analytics is to focus on ***stateful computations***, implementing custom advanced state management on top of inappropriate basic ingestion and storage APIs, leaving input/output data management to other specialized systems. Therefore, application workflows are designed to pull data from ingestion/storage systems and then transform and process the acquired data on local computing nodes and finally write results (or even input data) to dedicated storage systems. We argue that moving large quantities of data through specialized layers is not efficient. We propose that future Big Data analytics architectures should instead focus on ***data first***: data should be acquired, transformed, processed and stored while minimizing the number of copies.

This thesis makes a first step towards a data first approach by proposing and implementing KerA: a unified data ingestion and storage system with support for online and offline data accesses, by implementing dynamic partitioning, lightweight stream offset indexing,

adaptive and fine-grained replication, and data ("in-memory") locality for streaming operations, techniques that are required for efficient and fault-tolerant "in-storage" stream processing. Current hardware trends with increasing number of CPU cores per node and larger memory (RAM) capacities can efficiently sustain this approach in which, e.g., stream processing engines are co-located with ingestion/storage in order to further reduce data movement.

9.1 Achievements

This thesis was carried out in the context of the BigStorage ETN and brings the following contributions:

Requirements for dedicated stream ingestion and storage. After introducing a set of challenging requirements for an optimized ingestion and storage architecture, and then, after surveying state-of-the-art ingestion and storage systems, we discuss how they partially meet our requirements. These systems served as an excellent foundation and inspiration on which to build KerA with the goal of optimizing processing analytics. While Apache Kafka paved the way towards efficient (static) stream ingestion, recent workloads need to dynamically handle higher volumes at increased velocity, an important requirement that we considered in KerA's design and implementation. In order to provide low-latency accesses to high data volumes, we get inspired by RAMCloud, which is a distributed key-value system with a simple multi-key-value data model.

A methodology for understanding performance in Big Data analytics frameworks. We choose Flink (stream-based) and Spark (batch-based), two Apache-hosted data analytics frameworks, in order to identify and explain the impact of the different architectural choices and the parameter configurations on the perceived end-to-end performance. To this end, we develop a methodology for correlating the parameter settings and the operators execution plan with the resource usage, and we dissect the performance of Spark and Flink with several representative batch and iterative workloads on up to 100 nodes (1600 cores). Our key finding is that none of the two frameworks outperforms the other for all data types, sizes and job patterns: we highlight how results correlate to operators, to resource usage and to the specifics of the engines design and parameters.

Then, we focus on window-based streaming operators: Flink enables each stateful operator to work in isolation by creating data copies. To minimize memory utilization, we explored the feasibility of deduplication techniques to address the challenge of reducing memory footprint for window-based stream processing without significant impact on performance (typically measured as result latency). Our key finding is that more fine-grained interactions between streaming engines and (key-value) stores need to be designed (e.g., lazy deserialization or push processing to storage) in order to better respond to scenarios that have to overcome memory scarcity. Our experiments suggest that the ***state management*** function for Big Data processing can be enabled through stream-based interfaces exposed by a dedicated ingestion/storage engine like KerA. Our intuition is the following: treating both input/output data and respectively the (intermediate) state of the application operators as streams can finally lead to a sim-

plified, fault-tolerant and optimized processing analytics architecture. These insights contributed to our proposed design principles.

Design, implementation and evaluation of KerA: a unified data ingestion and storage system. Towards our vision of a unified analytics architecture, we propose that processing engines should focus on the operator workflow and leave the state management function to a unified ingestion/storage engine like KerA that addresses high-level data management (e.g., caching, concurrency control). Furthermore, processing and ingestion/storage engines should interact through stream-based interfaces. Therefore, processing engines expose the workflow APIs necessary to define the application semantics and orchestrate the scheduling and execution of stateless and stateful operators in cooperation with the ingestion/storage engine.

We describe KerA's unified data model for unbounded streams, records and objects, and then present the KerA architectural components. KerA was built on top of RAMCloud's framework in order to leverage its RPC framework and its support for multiple network transports (e.g., TCP, DPDK, Infiniband). KerA is integrated with Flink (can be co-located) through a shared-memory approach based on an extension of the Apache Arrow Plasma store. We evaluate the locality support and prove its importance for streaming analytics. We propose and implement in KerA two core ideas: (1) dynamic partitioning based on semantic grouping and sub-partitioning, which enables more flexible and elastic management of stream partitions; (2) lightweight offset indexing optimized for sequential record access. Our evaluation of KerA versus Kafka proves the importance and efficiency of these techniques.

Furthermore, in order to efficiently handle the durable ingestion of multiple streams, we propose a new method for adaptive and fine-grained replication: the virtual log technique provides a zero-copy replication framework which allows users/system to tune the configuration of the number of replicated logs per stream. We think that the virtual log support is extremely important for Big Data analytics: many large use cases (e.g., Netflix, Twitter, Facebook) require the management of a large number of streams, in addition to Volume and Velocity that remain important streaming attributes handled by our previous techniques for dynamic ingestion.

9.2 Future directions

Stream migration and recovery. More work is needed to add support in KerA for stream migration and recovery. This is extremelly challenging (research-wise) in the context of real-time data processing. We hope to leverage our proposed ideas and finish soon a KerA production-ready implementation. Then, we can work on integrating KerA with Apache Spark and Apache Flink in order to be able to execute more workloads (e.g., batch-oriented, interactive and iterative processing) through KerA's data stream interfaces. Finally, we can move to enabling real-time support for "in-storage" processing. There are two challenges that we describe below.

Dynamic and fine-grained stream access. The first challenge is to dynamically enable and evaluate fine-grained record access interfaces (e.g., multi-/put/get, as implemented by RAMCloud) on top of KerA, as needed by different streaming applications. This

is challenging since brokers are designed out-of-core, being able to move data to disk and back to memory as needed. Therefore, another research problem is understanding the performance impact of these fine-grained extensions enabled dynamically on KerA according to the applications needs. Other approaches leverage transient storage for serverless computing [53], techniques we plan to compare with.

A real-time Big Data processing engine. The second challenge is to design and implement a high-performance streaming engine (name it KerASP, stream processing on top of KerA) that can benefit from existing techniques (e.g., data lineage for recovery) and leverage KerA for handling state during processing. A challenge of KerASP is designing a workflow processing model that treats (fault-tolerant) operators as streams. KerASP should avoid multiple interfaces/APIs for the execution of different workflows (e.g., batch, iterative, streaming) in order to ease users work. KerASP coordinates closely with KerA in order to efficiently schedule (stateless and stateful) operators. Therefore, an important research problem is understanding ***which (batch/streaming) workloads can benefit most from pushing processing to storage and when is best to leverage such technique?*** These problems further exacerbate in the context of data distributed across multiple clusters or cloud data centers.

HPC-Big Data convergence at processing level by bridging in situ/in transit processing with Big Data analytics. A future objective is to propose an approach enabling HPC-Big Data convergence at the data processing level, by exploring alternative solutions to build a unified framework for extreme-scale data processing. The architecture of such a framework will leverage the extreme scalability demonstrated by in situ/in transit data processing approaches originated in the HPC area, in conjunction with Big Data processing approaches emerged in the BDA area (batch-based, streaming-based and hybrid). The high-level goal of this framework is to enable the usage of a large spectrum of Big Data analytics techniques at extreme scales, to support precise predictions in real-time and fast decision making [17].

In the process of designing the unified data processing framework, we will leverage the KerA architecture for high-performance ingestion and storage for stream processing and the Damaris [21] framework for scalable, asynchronous I/O and in situ and in transit visualization and processing.

Bibliography

[1] 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. ISSN: 2150-8097. DOI: 10.14778/2824032.2824076. URL: http://dx.doi.org/10.14778/2824032.2824076.

[2] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. "The Stratosphere Platform for Big Data Analytics". In: *The VLDB Journal* 23.6 (Dec. 2014), pp. 939–964. ISSN: 1066-8888. DOI: 10.1007/s00778-014-0357-y.

[3] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. "MapReduce and PACT - Comparing Data Parallel Programming Models". In: *Proceedings of the 14th Conference on Database Systems for BTW*. Kaiserslautern, Germany: GI, 2011, pp. 25–44. ISBN: 978-3-88579-274-1.

[4] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. "GRASS: Trimming Stragglers in Approximation Analytics". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 289–302. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616475.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL Continuous Query Language: Semantic Foundations and Query Execution". In: *The VLDB Journal* 15.2 (June 2006), pp. 121–142. ISSN: 1066-8888. DOI: 10.1007/s00778-004-0147-z. URL: http://dx.doi.org/10.1007/s00778-004-0147-z.

[6] Arvind Arasu and Jennifer Widom. "Resource Sharing in Continuous Sliding-window Aggregates". In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 336–347. ISBN: 0-12-088469-0. URL: http://dl.acm.org/citation.cfm?id=1316689.1316720.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797. URL: http://doi.acm.org/10.1145/2723372.2742797.

[8] *Apache Arrow*. 2018. URL: https://arrow.apache.org/ (visited on 2018-02-01).

[9] *Apache Avro*. 2018. URL: https://avro.apache.org/ (visited on 2018-10-01).

[10] *Big Data Digest: How many Hadoops do we really need?* 2018. URL: http://www.computerworld.com/article/2871760/big-data-digest-how-many-hadoops-do-we-really-need.html (visited on 2018-10-01).

[11] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Dayde, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quétier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed". In: *International Journal of High Performance Computing Applications* 20.4 (2006), pp. 481–494. DOI: 10.1177/1094342006070078.

[12] 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. ISBN: 978-1-60558-422-5. DOI: 10.1145/1516360.1516467. URL: http://doi.acm.org/10.1145/1516360.1516467.

[13] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. "Cutty: Aggregate Sharing for User-Defined Windows". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM '16. Indianapolis, Indiana, USA: ACM, 2016, pp. 1201–1210. ISBN: 978-1-4503-4073-1. DOI: 10.1145/2983323.2983807. URL: http://doi.acm.org/10.1145/2983323.2983807.

[14] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. "Realtime Data Processing at Facebook". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 1087–1098. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2904441. URL: http://doi.acm.org/10.1145/2882903.2904441.

[15] Francisco J. Clemente-Castelló, Bogdan Nicolae, Kostas Katrinis, M. Mustafa Rafique, Rafael Mayo, Juan Carlos Fernández, and Daniela Loreti. "Enabling Big Data Analytics in the Hybrid Cloud Using Iterative Mapreduce". In: *Proceedings of the 8th International Conference on Utility and Cloud Computing*. UCC '15. Limassol, Cyprus: IEEE Press, 2015, pp. 290–299. ISBN: 978-0-7695-5697-0. DOI: 10.1109/UCC.2015.47. URL: https://doi.org/10.1109/UCC.2015.47.

[16] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. "Gigascope: A Stream Database for Network Applications". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 647–651. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872838. URL: http://doi.acm.org/10.1145/872757.872838.

[17] *HPC-Big Data convergence at processing level by bridging in situ/in transit processing with Big Data analytics*. 2018. URL: https://team.inria.fr/kerdata/phd-position-hpc-bigdata-convergence/ (visited on 2018-10-01).

[18] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[19] Peter J. Desnoyers and Prashant Shenoy. "Hyperion: High Volume Stream Archival for Retrospective Querying". In: *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. ATC'07. Santa Clara, CA: USENIX Association, 2007, 4:1–4:14. URL: http://dl.acm.org/citation.cfm?id=1364385.1364389.

[20] Bo Dong, Qinghua Zheng, Feng Tian, Kuo-Ming Chao, Rui Ma, and Rachid Anane. "An Optimized Approach for Storing and Accessing Small Files on Cloud Storage". In: *J. Netw. Comput. Appl.* 35.6 (Nov. 2012), pp. 1847–1862. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2012.07.009. URL: http://dx.doi.org/10.1016/j.jnca.2012.07.009.

[21] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf. "Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations". In: *ACM Trans. Parallel Comput.* 3.3 (Oct. 2016), 15:1–15:43. ISSN: 2329-4949. DOI: 10.1145/2987371. URL: http://doi.acm.org/10.1145/2987371.

[22] *Apache Druid*. 2018. URL: https://druid.apache.org/ (visited on 2018-10-01).

[23] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. "HYDRAstor: a Scalable Secondary Storage". In: *FAST '09: Proccedings of the 7th conference on File and storage technologies*. San Francisco, USA: USENIX Association, 2009, pp. 197–210.

[24] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. "Spinning Fast Iterative Data Flows". In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1268–1279. ISSN: 2150-8097. DOI: 10.14778/2350229.2350245.

[25] *Facebook*. 2018. URL: https://www.facebook.com/ (visited on 2018-10-01).

[26] *Apache Flink*. 2018. URL: https://flink.apache.org/ (visited on 2018-10-01).

[27] *FLINK-2250*. 2018. URL: https://issues.apache.org/jira/browse/FLINK-2250 (visited on 2018-10-01).

[28] *Flink Large State Use Case*. 2018. URL: https://www.slideshare.net/GyulaFra/rbea-scalable-realtime-analytics-at-king (visited on 2018-10-01).

[29] *FlinkWindows*. 2018. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/windows.html (visited on 2018-10-01).

[30] *Medium Graph*. 2018. URL: http://snap.stanford.edu/data/com-Friendster.html (visited on 2018-10-01).

[31] Buğra Gedik. "Partitioning Functions for Stateful Data Parallelism in Stream Processing". In: *The VLDB Journal* 23.4 (Aug. 2014), pp. 517–539. ISSN: 1066-8888. DOI: 10.1007/s00778-013-0335-9. URL: http://dx.doi.org/10.1007/s00778-013-0335-9.

[32] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 29–43. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945450. URL: http://doi.acm.org/10.1145/945445.945450.

[33] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 599–613. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685096.

[34] *Google Algorithms and Theory*. http://research.google.com/pubs/AlgorithmsandTheory.html.

[35] *Grid5000*. 2018. URL: https://www.grid5000.fr/ (visited on 2018-10-01).

[36] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions". In: *Future Gener. Comput. Syst.* 29.7 (Sept. 2013), pp. 1645–1660. ISSN: 0167-739X. DOI: 10.1016/j.future.2013.01.010. URL: http://dx.doi.org/10.1016/j.future.2013.01.010.

[37] *Apache Hadoop*. 2018. URL: https://hadoop.apache.org/ (visited on 2018-10-01).

[38] *Hardware Trends in Keynote*. http://www.pdsw.org/keynote.shtml.

[39] *Apache Kudu*. 2018. URL: https://hbase.apache.org/ (visited on 2018-10-01).

[40] Arvid Heise et al. "Meteor/Sopremo: An Extensible Query Language and Operator Model". In: *Proceedings of the Int. Workshop on End-to-End Management of Big Data (BigData) in conjunction with VLDB*. 2012.

[41] *HiBench Suite*. 2018. URL: https://github.com/intel-hadoop/HiBench (visited on 2018-10-01).

[42] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851.

[43] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. "High-Availability Algorithms for Distributed Stream Processing". In: *Proceedings of the 21st International Conference on Data Engineering*. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 779–790. ISBN: 0-7695-2285-8. DOI: 10.1109/ICDE.2005.72. URL: http://dx.doi.org/10.1109/ICDE.2005.72.

[44] *Introducing Spark Datasets.* https://databricks.com/blog/2016/01/04/introducing-spark-datasets.html.

[45] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. "Towards a Streaming SQL Standard". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1379–1390. ISSN: 2150-8097. DOI: 10.14778/1454159.1454179. URL: http://dx.doi.org/10.14778/1454159.1454179.

[46] Kreps Jay, Narkhede Neha, and Rao Jun. "Kafka: A distributed messaging system for log processing". In: *Proceedings of 6th International Workshop on Networking Meets Databases*. NetDB'11. Athens, Greece, 2011.

[47] *Juggling with Bits and Bytes*. https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html.

[48] Flavio P. Junqueira, Ivan Kelly, and Benjamin Reed. "Durability with BookKeeper". In: *SIGOPS Oper. Syst. Rev.* 47.1 (Jan. 2013), pp. 9–15. ISSN: 0163-5980. DOI: 10.1145/2433140.2433144.

[49] *Apache Kafka*. 2018. URL: https://kafka.apache.org/ (visited on 2018-10-01).

[50] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. "SLIK: Scalable Low-latency Indexes for a Key-value Store". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 57–70. ISBN: 978-1-931971-30-0. URL: http://dl.acm.org/citation.cfm?id=3026959.3026966.

[51] *Keys to Understanding Amazon's Algorithms*. http://www.thebookdesigner.com/2013/07/amazon-algorithms/.

[52] *Amazon Kinesis*. 2018. URL: https://aws.amazon.com/kinesis/data-streams/ (visited on 2018-10-01).

[53] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 427–444. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[54] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. "On-the-fly Sharing for Streamed Aggregation". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 623–634. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142543. URL: http://doi.acm.org/10.1145/1142473.1142543.

[55] *Kryo*. 2018. URL: https://github.com/EsotericSoftware/kryo (visited on 2018-10-01).

[56] *Apache Kudu*. 2018. URL: https://kudu.apache.org/ (visited on 2018-10-01).

[57] Chinmay Kulkarni, Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. "Beyond Simple Request Processing with RAMCloud". In: *IEEE Data Eng.* (2017).

[58] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. "Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 627–643. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/kulkarni.

[59] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 239–250. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742788. URL: http://doi.acm.org/10.1145/2723372.2742788.

[60] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. "SkewTune: Mitigating Skew in Mapreduce Applications". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 25–36. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213840. URL: http://doi.acm.org/10.1145/2213836.2213840.

[61] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[62] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. "Implementing Linearizability at Large Scale and Low Latency". In: *25th SOSP*. Monterey, California: ACM, 2015, pp. 71–86. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815416.

[63] *Large Hadron Holider*. 2018. URL: http://home.cern/topics/large-hadron-collider (visited on 2018-10-01).

[64] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC. Seattle, WA, USA: ACM, 2014, 6:1–6:15. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670985.

[65] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. "No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams". In: *SIGMOD Rec.* 34.1 (Mar. 2005), pp. 39–44. ISSN: 0163-5808. DOI: 10.1145/1058150.1058158. URL: http://doi.acm.org/10.1145/1058150.1058158.

[66] 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. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066193. URL: http://doi.acm.org/10.1145/1066157.1066193.

[67] Björn Lohrmann, Daniel Warneke, and Odej Kao. "Nephele Streaming: Stream Processing Under QoS Constraints at Scale". In: *Cluster Computing* 17.1 (Mar. 2014), pp. 61–78. ISSN: 1386-7857. DOI: 10.1007/s10586-013-0281-8. URL: http://dx.doi.org/10.1007/s10586-013-0281-8.

[68] Danelutto M., Kilpatrick P., Mencagli G., and Torquati M. "State Access Patterns in Stream Parallel Computations". In: *International Journal of High Performance Computing Applications (IJHPCA)*. 2017. DOI: 10.1177/1094342017694134. URL: http://pages.di.unipi.it/mencagli/publications/preprint-ijhpca-2017.pdf.

[69] 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. DOI: 10.1007/978-3-540-30570-5_3. URL: http://dx.doi.org/10.1007/978-3-540-30570-5_3.

[70] *MapR Streams*. 2018. URL: https://mapr.com/products/mapr-streams (visited on 2018-10-01).

[71] Pierre Matri. "Týr: Storage-Based HPC and Big Data Convergence Using Transactional Blobs". June 2018. URL: http://oa.upm.es/51431/.

[72] Marcelo R.N. Mendes, Pedro Bizarro, and Paulo Marques. "Overcoming Memory Limitations in High-throughput Event-based Applications". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ICPE '13. Prague, Czech Republic: ACM, 2013, pp. 399–410. ISBN: 978-1-4503-1636-1. DOI: 10.1145/2479871.2479933. URL: http://doi.acm.org/10.1145/2479871.2479933.

[73] *Messaging, storage, or both?* 2018. URL: https://streaml.io/blog/messaging-storage-or-both (visited on 2018-10-01).

[74] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. "StreamBox: Modern Stream Processing on a Multicore Machine". In: *USENIX ATC*. Santa Clara, CA, USA: USENIX Association, 2017, pp. 617–629. ISBN: 978-1-931971-38-6.

[75] Cherniack Mitch, Balakrishnan Hari, Balazinska Magdalena, Carney Donald, Cetintemel Ugur, Xing Ying, and Zdonik Stan. "Scalable Distributed Stream Processing". In: *First Biennial Conference on Innovative Data Systems Research*. 2003. URL: http://cs.brown.edu/research/aurora/cidr03.pdf.

[76] Tom M. Mitchell. "Machine Learning and Data Mining". In: *Commun. ACM* 42.11 (Nov. 1999), pp. 30–36. ISSN: 0001-0782. DOI: 10.1145/319382.319388. URL: http://doi.acm.org/10.1145/319382.319388.

[77] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging AI Applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/nishihara.

[78] *New directions for Apache Spark in 2015*. http://www.slideshare.net/databricks/new-directions-for-apache-spark-in-2015.

[79] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. "HopsFS: Scaling Hierarchical File System Metadata Using newSQL Databases". In: *Proceedings of the 15th Usenix Conference on File and Storage Technologies*. FAST'17. Santa clara, CA, USA: USENIX Association, 2017, pp. 89–103. ISBN: 978-1-931971-36-2. URL: http://dl.acm.org/citation.cfm?id=3129633.3129642.

[80] Bogdan Nicolae. "BlobSeer: Towards efficient data storage management for large-scale, distributed systems". Theses. Université Rennes 1, Nov. 2010. URL: https://tel.archives-ouvertes.fr/tel-00552271.

[81] Bogdan Nicolae. "Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead". In: *IPDPS '15: 29th IEEE International Parallel and Distributed Processing Symposium*. Hyderabad, India, 2015, pp. 1023–1032.

[82] Bogdan Nicolae. "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal". In: *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*. Boston, USA, 2013, pp. 19–28. DOI: 10.1109/IPDPS.2013.14.

[83] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. "Fast Crash Recovery in RAMCloud". In: *23rd SOSP*. Cascais, Portugal: ACM, 2011, pp. 29–41. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043560.

[84] *Apache Orc*. 2018. URL: https://orc.apache.org/ (visited on 2018-10-01).

[85] John Ousterhout. "Always Measure One Level Deeper". In: *Commun. ACM* 61.7 (June 2018), pp. 74–83. ISSN: 0001-0782. DOI: 10.1145/3213770. URL: http://doi.acm.org/10.1145/3213770.

[86] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. "The RAMCloud Storage System". In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015), 7:1–7:55. ISSN: 0734-2071. DOI: 10.1145/2806887. URL: http://doi.acm.org/10.1145/2806887.

[87] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. "Making Sense of Performance in Data Analytics Frameworks". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 293–307. ISBN: 978-1-931971-218. URL: http://dl.acm.org/citation.cfm?id=2789770.2789791.

[88] *Apache Parquet*. 2018. URL: https://parquet.apache.org/ (visited on 2018-10-01).

[89] *Pravega*. http://pravega.io/.

[90] *Project Tungsten.* https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[91] *Apache Pulsar*. 2018. URL: https://pulsar.apache.org/ (visited on 2018-10-01).

[92] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. "Arachne: Core-Aware Thread Management". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/qin.

[93] *Redis*. 2018. URL: https://redis.io/ (visited on 2018-10-01).

[94] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. "Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics". In: *Proc. VLDB Endow.* 8.13 (Sept. 2015), pp. 2110–2121. ISSN: 2150-8097. DOI: 10.14778/2831360.2831365. URL: https://doi.org/10.14778/2831360.2831365.

[95] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://dx.doi.org/10.1109/MSST.2010.5496972.

[96] Guo Sijie, Dhamankar Robin, and Stewart Leigh. "DistributedLog: A High Performance Replicated Log Service". In: *IEEE 33rd International Conference on Data Engineering*. ICDE'17. San Diego, CA, USA: IEEE, 2017. URL: http://ieeexplore.ieee.org/document/7930058/.

[97] *Small Graph.* http://an.kaist.ac.kr/~haewoon/release/twitter_social.

[98] *Apache Spark*. 2018. URL: https://spark.apache.org/ (visited on 2018-10-01).

[99] *Apache Impala*. 2018. URL: https://impala.apache.org/ (visited on 2018-10-01).

[100] Yacine Taleb. "Optimizing Distributed In-memory Storage Systems: Fault-tolerance, Performance, Energy Efficiency". Theses. ENS Rennes, Oct. 2018. URL: https://hal.inria.fr/tel-01891897.

[101] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. "Tailwind: Fast and Atomic RDMA-based Replication". In: *ATC '18 - USENIX Annual Technical Conference*. Boston, United States, July 2018, pp. 1–13. URL: https://hal.inria.fr/hal-01676502.

[102] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. "General Incremental Sliding-window Aggregation". In: *Proc. VLDB Endow.* 8.7 (Feb. 2015), pp. 702–713. ISSN: 2150-8097. DOI: 10.14778/2752939.2752940. URL: http://dx.doi.org/10.14778/2752939.2752940.

[103] *Hadoop TeraGen for TeraSort*. 2018. URL: https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html (visited on 2018-10-01).

[104] *Tera Sort*. 2018. URL: http://eastcirclek.blogspot.fr/2015/06/terasort-for-spark-and-flink-with-range.html (visited on 2018-10-01).

[105] *The world beyond batch: Streaming 101*. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101.

[106] *The world beyond batch: Streaming 102*. https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102.

[107] Quoc-Cuong To, Juan Soto, and Volker Markl. "A Survey of State Management in Big Data Processing Systems". In: *CoRR* abs/1702.01596 (2017). arXiv: 1702.01596. URL: http://arxiv.org/abs/1702.01596.

[108] Radu Tudoran, Alexandru Costan, Gabriel Antoniu, and Hakan Soncu. "TomusBlobs: Towards Communication-Efficient Storage for MapReduce Applications in Azure". In: *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 427–434. ISBN: 978-0-7695-4691-9. DOI: 10.1109/CCGrid.2012.104. URL: https://doi.org/10.1109/CCGrid.2012.104.

[109] *Twitter*. https://twitter.com/.

[110] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: 10.1145/79173.79181. URL: http://doi.acm.org/10.1145/79173.79181.

[111] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. "Drizzle: Fast and Adaptable Stream Processing at Scale". In: *26th SOSP*. Shanghai, China: ACM, 2017, pp. 374–389. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132750.

[112] Paolo Viotti and Marko Vukolić. "Consistency in Non-Transactional Distributed Storage Systems". In: *ACM Comput. Surv.* 49.1 (June 2016), 19:1–19:34. ISSN: 0360-0300. DOI: 10.1145/2926965. URL: http://doi.acm.org/10.1145/2926965.

[113] Daniel Warneke and Odej Kao. "Nephele: Efficient Parallel Data Processing in the Cloud". In: *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*. Portland, Oregon: ACM, 2009. ISBN: 978-1-60558-714-1. DOI: http://doi.acm.org/10.1145/1646468.1646476.

[114] *Large Graph*. 2018. URL: http://webdatacommons.org/hyperlinkgraph (visited on 2018-10-01).

[115] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. "Understanding Replication in Databases and Distributed Systems". In: *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*. ICDCS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 464–. ISBN: 0-7695-0601-1. URL: http://dl.acm.org/citation.cfm?id=850927.851782.

[116] 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. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2595631. URL: http://doi.acm.org/10.1145/2588555.2595631.

[117] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. "A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing". In: *SIGMETRICS Perform. Eval. Rev.* 40.4 (Apr. 2013), pp. 23–32. ISSN: 0163-5999. DOI: 10.1145/2479942.2479946. URL: http://doi.acm.org/10.1145/2479942.2479946.

[118] Esma Yildirim and Tevfik Kosar. "Network-aware end-to-end data throughput optimization". In: *Proceedings of the first international workshop on Network-aware data management*. Seattle, Washington, USA, 2011, pp. 21–30. ISBN: 978-1-4503-1132-8.

[119] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 265–278. ISBN: 978-1-60558-577-2. DOI: 10.1145/1755913.1755940. URL: http://doi.acm.org/10.1145/1755913.1755940.

[120] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301.

[121] 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. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737.

[122] Benjamin Zhu, Kai Li, and Hugo Patterson. "Avoiding the disk bottleneck in the data domain deduplication file system". In: *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. San Jose, USA: USENIX Association, 2008, 18:1–18:14.

Part V

Appendix

Contexte

Le BIG DATA est devenu une nouvelle ressource naturelle. Les dernières années ont connu une croissance sans précédent des données qui doivent être traitées avec une vitesse toujours croissante afin d'en extraire des renseignements précieux (e.g., Facebook, Amazon, LHC, etc.). Comme le volume des données augmente sans cesse, les architectures actuelles d'analyse de données massives doivent relever des défis de plus en plus élevés en termes d'extensibilité, d'ingestion rapide, de rendement de traitement et d'efficacité de stockage.

Cette masse de données hétérogènes augmentent de manière exponentielle et surtout sont produites de plus en plus vite. Les applications d'analyse des données massives sont passées du traitement par lot (*batch processing*) au traitement par flux (*stream processing*), ce qui peut réduire considérablement le temps nécessaire à l'extraction des renseignements précieux. En même temps, les applications complexes de gestion du flux de travail (*workflow*) ont conduit les usagers à demander des modèles de programmation unifiés pour simplifier les charges de travail existantes et pour favoriser l'émergence de nouvelles applications. Les architectures de traitement ont évolué à tel point qu'elles sont capable de nos jours de supporter des traitements par lot, des traitements interactifs et des traitements par flux dans le même système [18, 120, 2]. Il est ainsi possible d'optimiser les applications en supprimant le dédoublement de travail et le partage des ressources entre les moteurs de calcul spécialisés.

Les architectures actuelles (*state-of-the-art*) pour l'analyse des données massives sont construites sur une pile à trois couches : premièrement les flux de données sont acquis par la couche d'ingestion (e.g., Apache Kafka [46]) pour ensuite circuler à travers la couche de traitement (e.g., Apache Flink [26], Apache Spark [98]) qui s'appuie sur la couche de stockage (e.g., HDFS [95]) pour stocker des données agrégées, pour permettre des contrôles intermédiaires ou pour archiver des flux pour un traitement ultérieur. Dans ces circonstances, les données sont souvent écrites deux fois sur le disque ou envoyées deux fois sur le réseau, par exemple dans le cadre d'une stratégie de tolérance aux pannes de la couche d'ingestion ou des exigences de persistance de la couche de stockage. Deuxièmement, le manque de coordination entre les couches peut créer des interférences entre les différentes entrées-sorties ; par exemple, la couche d'ingestion et la couche de stockage pourraient être en concurrence pour les mêmes ressources d'entrées-sorties au moment de la collecte des flux de données et de l'écriture simultanée des données des archives. Troisièmement, afin de gérer efficacement l'état des données (*state*) par lot et en flux pendant l'exécution, la couche de traitement met souvent en œuvre une gestion des données avancée et personnalisée (e.g., *operator state persistence*, *checkpoint-restart*) sur des API d'ingestion et de stockage de base inappropriées, ce qui a un impact significatif sur les performances.

Malheureusement, malgré les bénéfices potentiels apportés par les couches spécialisées (e.g., une mise en œuvre simplifiée), déplacer des quantités importantes de données à travers des couches spécialisées s'avère peu efficace. Au lieu de cela, les données devraient être acquises, traitées et stockées en minimisant le nombre de copies. Nous considérons que les défis mentionnés ci-dessus sont assez significatifs pour compenser les bénéfices de la spécialisation de chaque couche indépendamment des autres couches.

En outre, ces architectures sont de plus en plus complexes à maintenir. Ce sont des systèmes robustes et tolérants aux pannes qui sont aptes à servir des modèles d'accès aux données à la fois en ligne et hors ligne, demandées par des cas d'utilisation moderne des flux. Il

est donc nécessaire d'avoir ***une solution optimisée pour l'ingestion et le stockage des données, qu'elles soient limitées (objets) ou illimitées (flux)***. La conception et la mise en œuvre d'une telle solution dédiée est un grand défi : elle devrait assurer non seulement la fonctionnalité traditionnelle de stockage (i.e., support pour des objets), mais aussi le respect des critères d'accès en temps réel des applications modernes basées sur les flux, par exemple des latences d'accès aussi faibles que possibles pour les éléments des données, débit aussi haut que possible pour l'ingestion et le traitement des flux de données, etc.

Cette thèse a été réalisée dans le cadre du projet européen *BigStorage*, avec comme but de caractériser et comprendre les limites des architectures d'analyse Big Data les plus récentes et pour concevoir et implémenter des modèles de traitement et de gestion en flux des données pour dépasser les limitations actuelles et optimiser le traitement des flux.

Objectifs

Pour dépasser les limites mentionnées ci-dessus, cette thèse propose la conception et la mise en œuvre de manière minutieuse ***d'une architecture unifiée pour la gestion des flux* (streaming)*, l'ingestion et le stockage***, apte à optimiser le traitement des applications Big Data de gestion des données en flux tout en minimisant le déplacement des données à travers l'architecture d'analyse, conduisant ainsi à une amélioration de la performance et de l'utilisation des ressources. Cette proposition de décline dans les objectifs suivants.

1. Identifier un ensemble de critères pour un moteur dédié d'ingestion des flux (*stream ingestion*) et stockage.
2. Expliquer l'impact des différents choix architecturaux Big Data sur la performance et comprendre quelles sont les limitations actuelles d'un moteur de gestion des flux lors de son interaction avec un système de stockage pour préserver l'état des flux de données.
3. Proposer un ensemble de principes de conception d'une architecture unifiée et évolutive pour l'ingestion et le stockage des données.
4. Mettre en œuvre un prototype pour un moteur dédié de gestion de flux pour l'ingestion et le stockage dans le but de gérer efficacement divers modèles d'accès : accès à latence faible aux enregistrements des streams (*stream records*) et/ou accès à débit élevé aux flux (illimités) et/ou objets.

Contributions

Au vu des objectifs mentionnés précédemment, nous résumons les principales contributions de cette thèse comme suit.

Critères pour une solution dédiée d'ingestion et de stockage de flux

Les applications basées sur les flux doivent ingérer et analyser immédiatement les données et dans des nombreux cas combiner des données en direct (flux non liés) et archivées (objets,

i.e., les flux liés) afin d'extraire de meilleures informations. Dans ce contexte, les systèmes de traitement Big Data en ligne et interactif (e.g., Apache Flink [26], Apache Spark [121]) conçus à la fois pour le traitement par lots et le traitement par flux sont rapidement adoptés en vue de remplacer des modèles de traitement traditionnels orientés sur les lots seulement (tels que MapReduce [18] et son implémentation à code source libre Hadoop [37]) qui ne suffisent pas pour répondre aux besoins de latence faible et de fréquence élevée des flux [69, 105, 106, 5, 14]. La question centrale que nous abordons dans cette thèse est de ***construire une architecture générale de traitement des flux Big Data capable de gérer efficacement des applications de flux très diverses***, tout en minimisant le mouvement de données pour une meilleure utilisation des ressources et de meilleures performances.

Nous identifions et discutons les caractéristiques applicatives d'un ensemble de scénarios basés sur les flux qui ont inspiré un ensemble de critères intéressants pour une architecture optimisée d'ingestion et de stockage.

En résumé, les applications basées sur les flux reposent fortement sur les fonctionnalités suivantes, mal supportées par les architectures de streaming actuelles.

1. ***Ingestion rapide***, éventuellement doublée par ***indexation simultanée*** (souvent, par un seul passage sur les données) pour le traitement en temps réel.

2. ***Stockage à latence faible*** avec un ***support de requête à granularité fine*** supplémentaire pour le filtrage efficace et l'agrégation des enregistrements de données.

3. Stockage permettant de gérer des événements qui s'accumulent ***en grands volumes sur une courte durée***.

Quelques tendances générales peuvent être observées à partir des applications présentées ci-dessus. Premièrement le modèle d'accès aux données est complexe et implique une diversité de tailles de données et de modèles d'accès (i.e., enregistrements, flux, objets). Deuxièmement, les architectures de traitement des flux doivent permettre des fonctionnalités avancées pour les applications, telles que le partitionnement personnalisé, la gestion des métadonnées distribuées, le pré-prétraitement, le contrôle des flux. En plus de ces fonctionnalités, elles doivent également tenir compte des aspects non fonctionnels tels que la haute disponibilité, la durabilité des données et le contrôle de latence par rapport au rendement. Sur la base de ces tendances générales, nous pouvons déduire les exigences suivantes pour une architecture de stockage de flux optimisée.

Par la suite, nous étudions les systèmes d'ingestion et de stockage modernes afin d'évaluer la manière dont elles remplissent nos critères. Ce sont ces systèmes qui ont constitué une excellente source d'inspiration et une base solide sur lesquelles nous avons pu construire la solution proposée pour une architecture unifiée pour l'ingestion et stockage des flux dans le but d'optimiser les systèmes d'analyse de traitement des données.

Notre objectif est de fournir des interfaces d'accès efficaces pour les enregistrements, les flux et les objets. Une approche possible serait d'améliorer un tel système avec les fonctionnalités manquantes. Cependant, cela est difficile à réaliser car, compte tenu des choix de conception initiaux de leurs développeurs, les autres exigences restantes seraient difficiles, voire impossibles à satisfaire en même temps. Par exemple, la conception de Kafka est basée sur le *cache OS*, ce qui rend difficile la co-localisation avec un moteur de traitement ; Redis n'offre pas la cohérence forte que certains cas d'utilisation peuvent nécessiter ; le choix de

Druid de différencier les nœuds historiques et temps réel conduit à l'adoption à la fois de Kafka et HDFS, contrairement à notre objectif de minimiser le nombre de copies de données. Bien que RAMCloud n'ait pas été conçu pour une ingestion de flux à haut débit, son modèle de données *key-value* constituait un bon point de départ pour la représentation d'un enregistrement de flux. Toutefois, la création d'interfaces de diffusion en continu sur RAMCloud n'était pas possible, car de nombreuses situations n'exigent pas un accès au niveau des enregistrement à une granularité fine. Par conséquent, bien que l'extension d'un système existant ne soit pas une solution, la création d'une nouvelle solution dédiée en s'appuyant sur ces systèmes ou certaines de leurs idées fondamentales comme briques de base est une bonne option.

Comprendre la performance dans les systèmes d'analyse Big Data

Temps d'execution basé sur les flux (*stream-based*) versus temps d'execution basé sur les lots (*batch-based*)

Au cours de la dernière décennie, MapReduce (*un modèle de programmation et une implémentation associée pour le traitement et la génération de grands ensembles de données* [18]) et son implémentation libre Hadoop [37] ont été largement adoptés tant par l'industrie que par le monde universitaire. Ils fournissent en effet un modèle de programmation simple mais puissant qui cache aux utilisateurs la complexité de l'exécution des tâches en parallèle et de la gestion de la tolérance aux pannes. Cette API très simple est accompagnée d'une restriction importante liée au fait qu'elle force les applications à être exprimées en termes de fonctions d'application (*map*) et de réduction (*reduce*). Comme l'expliquent les concepteurs,

> ***les utilisateurs spécifient une fonction d'application qui traite une paire clé/valeur pour générer un ensemble de paires clé/valeur intermédiaires et une fonction de réduction qui fusionne toutes les valeurs intermédiaires associées à la même clé intermédiaire*** [18].

Cependant, la plupart des applications ne correspondent pas à ce modèle et nécessitent une orchestration de données plus générale, indépendante de tout modèle de programmation. Par exemple, les algorithmes itératifs utilisés dans l'analyse des graphes et l'apprentissage automatique effectuent plusieurs cycles de calcul sur les mêmes données, ce qui ne correspond pas au modèle MapReduce d'origine. En outre, l'augmentation des volumes de données et la dimension en ligne du traitement des données nécessitent des modèles de diffusion en continu afin de permettre le traitement en temps réel des sources de données.

Face à ces limitations, une deuxième génération de plates-formes d'analyse a vu le jour dans le but d'unifier le paysage du traitement Big Data. Flink (*stream-based*) et Spark (*batch-based*) sont deux environnement d'exécution d'applications d'analyse des données hébergés par Apache. Ils facilitent le développement des pipelines de données multi-étapes en utilisant de manière directe des modèles de graphes acycliques.

Spark [98] considère le traitement de données par lots. Il introduit la notion d'ensemble de données distribuées résilientes (*Resilient Distributed Datasets* ou RDD [120]). Un RDD est un ensemble de structures de données en mémoire capables de mettre en mémoire-cache des données intermédiaires sur un ensemble de nœuds, afin de prendre en charge efficacement des algorithmes itératifs. Flink [26] considère le traitement de données par flux. Dans le

même but, a proposé plus récemment des opérateurs d'itération en boucle fermée natifs [24] et un optimiseur automatique basé sur les coûts, capable de réorganiser les opérateurs et de mieux prendre en charge l'exécution en continu par la réduction de la latence.

Tirer le meilleur parti possible de ces structures constitue un défi considérable car l'efficacité des exécutions dépend fortement de l'ajustement des configurations complexes des paramètres par une compréhension fine des choix architecturaux sous-jacents. Pour cela, nous proposons une méthodologie qui permet de corréler le réglage des paramètres et le plan d'exécution des opérateurs avec l'usage des ressources. Nous analysons les performances de Spark et Flink avec plusieurs charges de travail qui sont représentatives à la fois du traitement par lot (batch) itératif sur des plateformes jusqu'à 100 nœuds. La principale conclusion de cette analyse est qu'aucune des deux structures ne surpasse l'autre pour tous les types de données, les dimensions et les modèles d'emploi. Nous approfondissons la manière dont les résultats sont corrélés avec les opérateurs, l'usage des ressources et les spécificités de la conception du moteur et des paramètres.

La gestion de la mémoire joue un rôle crucial dans l'exécution d'une charge de travail, en particulier pour les ensembles de données plus volumineux que la mémoire disponible. Par exemple, le composant d'agrégation de Flink (combinateur basé sur le tri) semble plus efficace que celui de Spark car il s'appuie sur une gestion personnalisée de la mémoire et la sérialisation différenciée des données selon leur type (*type oriented*). Avec son API DataSet [44] pour des données structurées, Spark visait une approche similaire.

Le sens commun veut que le traitement de beaucoup de données dans une machine virtuelle Java (JVM) conduise à les stocker en tant qu'objets sur le tas. Cette approche présente des inconvénients notables, comme mentionné dans [47]. Tout d'abord, comme le montre le graphique du cas de la grande taille de la section 3.3.1, une sur-allocation de mémoire détruira immédiatement la machine virtuelle Java. En outre, les machines virtuelles de grande taille (plusieurs Go), submergées par des milliers de nouveaux objets, peuvent souffrir de la surcharge de la récupération des zones mémoire inutilisées (*garbage collection*). Enfin, sur la plupart des plates-formes JVM, les objets Java augmentent la surcharge d'espace. Ceci pourrait conduire à devoir régler les paramètres du système de manière spécifique à chaque cas d'utilisation pour optimiser l'organisation de la mémoire, d'éviter la surallocation et les problèmes de récupération de la mémoire inutilisée. Les moteurs d'analyse doivent donc disposer d'une mémoire efficace. Au cours de nos expériences, nous avons remarqué que, contrairement à Spark, Flink n'accumule pas beaucoup d'objets sur le tas, mais les stocke dans une région de la mémoire dédiée en dehors du tas pour éviter les problèmes de mémoire. Ceci conduit à une configuration de mémoire hybride, dans le tas et en dehors du tas, qui est difficile à régler. Le réglage des fraction de mémoire devrait (idéalement) être fait automatiquement par le système et modifié dynamiquement à l'exécution. Dans Flink, la plupart des opérateurs sont implémentés pour qu'ils puissent survivre avec très peu de mémoire en utilisant le disque si nécessaire. Nous avons également observé que, bien que Spark puisse sérialiser des données sur disque, il faut que des parties (significatives) de données soient placées dans le tas de la machine virtuelle Java pour plusieurs opérations ; si la taille du tas n'est pas suffisant, le travail s'arrête. Récemment, Spark a commencé à corriger ces problèmes de mémoire avec son projet Tungsten [90], fortement inspiré du modèle Flink. Il permet la gestion personnalisée de la mémoire pour à éliminer la surcharge du modèle d'objet JVM et la récupération des zones mémoire inutilisées.

Nos expériences suggèrent que la gestion des configurations pour le traitement Big Data

devrait être favorisée par un assistant dédié comme celui que nous proposons dans cette thèse. En outre, la conception d'une architecture d'analyse unifiée par des interfaces orientées flux apparaît comme un choix incontournable ; traiter les données d'entrée-sortie ainsi que l'état (intermédiaire) des opérateurs d'application comme des flux permet obtenir une architecture d'analyse simplifiée et optimisée.

Les systèmes MapReduce (e.g., Spark) adoptent le modèle *synchrone par bloc* (BSP [110]) dans lequel le calcul consiste en deux phases qui se répètent en continu : une phase de calcul (application) au cours de laquelle tous les nœuds d'un système distribué effectuent des calculs, suivie par une barrière de synchronisation qui permet une phase de communication (réduction) au cours de laquelle les noeuds communiquent. Pour mettre en œuvre la tolérance aux pannes, ces systèmes implémentent un cliché de l'état à la barrière : les résultats intermédiaires sont sauvegardés (via des points de contrôle) ou, plus efficacement, l'ensemble de l'étape traitement est enregistré (e.g., les RDD). Cependant, étant donné que les applications Big Data peuvent être composées de nombreuses phases d'application et de réduction, l'utilisateur est obligé de décider du meilleur moment pour sauvegarder les résultats intermédiaires ou tout simplement pour activer les techniques de récupération de l'étape de traitement.

Les systèmes de flux de données (e.g., Flink) adoptent le modèle d'opérateur continu (long terme) dans lequel les opérateurs sont activés une seule fois pour s'exécuter comme des tâches à long terme. Bien que ce modèle d'exécution soit plus flexible que BSP, ces systèmes reposent sur des algorithmes de point de contrôle distribués coûteux lors d'une exécution normale. Pour gérer la défaillance d'un nœud, tous les noeuds sont restaurés au dernier point de contrôle disponible et chaque opérateur continu est rejoué séquentiellement [111].

Nous soutenons qu'avec le soutien approprié d'un stockage de flux à latence faible, les opérateurs basés sur les flux continus peuvent en outre mettre en œuvre des techniques de lignée afin d'optimiser le traitement, par exemple, en utilisant des techniques telles que la récupération parallèle [121]. Dans ce cas, chaque opérateur de flux de données peut stocker de manière asynchrone un flux de lignée des calculs de sa tâche ainsi que des compensations de flux d'entrée et de sortie pour chaque tâche déployée ; le pilote d'exécution pourrait alors récupérer chaque tâche d'opérateur indépendamment tandis que les techniques de contre-pression pourraient aider à maintenir un calcul cohérent. Avec une telle fonctionnalité puissante, les systèmes de flux de données pourraient devenir plus attrayants pour les exécutions par lots et par flux à la fois. Pour que cela se produise, un changement d'approche fondamental est nécessaire : les moteurs de traitement de données sans limites doivent être conçus pour s'appuyer, pour la gestion des états, sur des systèmes d'ingestion/stockage dynamiques à granularité fine, comme celui que nous proposons dans cette thèse.

L'exploration de l'état partagé (*shared state*) pour la diffusion en continu des analyses à base de fenêtres (*window-based streaming analytics*)

Les moteurs conçus pour la gestion de flux [1, 67, 121] traitent généralement des sources de données en direct (e.g., services Web, flux de nouvelles et de réseaux sociaux, capteurs, etc.) à l'aide d'agrégateurs à état (appelées *opérateurs*) définis par l'application qui forment un graphe acyclique dirigé à travers lequel les données circulent. Dans ce contexte, il arrive souvent que ces agrégateurs à états doivent analyser les mêmes données, par exemple les K valeurs supérieures et des K valeurs inférieures observées au cours de la dernière heure dans

un flux d'entiers). Les méthodes les plus récentes créent des copies des données permettant à chaque opérateur de travailler de manière isolée, au détriment d'une utilisation plus importante de la mémoire (e.g., Apache Flink permet à chaque opérateur de travailler isolément en créant des copies des données). Cependant, avec l'augmentation du nombre de cœurs et la diminution de la mémoire disponible par cœur [38], la mémoire devient une ressource rare et peut potentiellement créer des goulots d'étranglement d'efficacité (e.g., des cœurs sous-utilisés), des coûts supplémentaires (e.g., des infrastructures plus coûteuses) ou encore poser des problèmes de faisabilité (e.g., manque de mémoire). Par conséquent, le problème de la minimisation de l'utilisation de la mémoire sans impact significatif sur les performances (généralement mesuré en tant que latence des résultats) est crucial. Pour minimiser l'utilisation de la mémoire, nous explorons la faisabilité des techniques de déduplication pour réduire l'empreinte mémoire pour le traitement des flux à base de fenêtres, sans pour autant altérer de manière signicative la performance (typiquement mesurée par la latence des résultats).

Sur la base de cette étude, nous tirons trois conclusions. Premièrement, en cas de contrainte de faible mémoire, les opérateurs basés sur les fenêtres ont tendance à avoir des performances médiocres en raison d'appels fréquents au ramassage miette (*garbage collector*). Dans ce cas, la latence nécessaire pour traiter 99 % des événements est jusqu'à 10 fois supérieure au cas où il n'y a pas de contraintes liées à la mémoire. La déduplication peut donc améliorer la latence. Deuxièmement, la déduplication entraîne une dégradation accrue des performances pour une taille de fenêtre croissante par rapport au cas où des copies sont utilisées. Ainsi, une sélection minutieuse de la taille de la fenêtre est nécessaire. Troisièmement, la déduplication peut considérablement réduire l'usage de la mémoire : dans le simple cas où deux opérateurs partageant le même état, il y a une réduction de 25 %, qui continue de croître proportionnellement avec le nombre d'opérateurs partageant le même état.

L'une des principales constatations est la nécessité de concevoir plusieurs interactions à granularité fine entre les moteurs de traitement de flux (*streaming*) et les systèmes de stockage de données clé/valeur (*key-value stores*) (e.g., *lazy deserialization* ou *push processing to storage*) afin de réagir au mieux lorsque la mémoire devient insuffisante.

Les moteurs de gestion de flux continus développent des mécanismes complexes pour gérer en interne l'état de la fenêtre de traitement *windowing*. Comme le montrent les expériences (chapitre 4), il peut être difficile de s'appuyer sur un stockage externe pour conserver l'état de diffusion. Cependant, pour une récupération rapide après incident et une adaptabilité rapide, cet état doit également être conservé (et mis à jour de manière incrémentale) dans un stockage distribué ; ces opérations réalisées grâce à des points de contrôle incrémentaux. Cette complexité pourrait être évitée pour le traitement si le support requis pour conserver l'état de fenêtrage était développé au sein d'une couche d'ingestion et de stockage unifiée. Avec une telle prise en charge, il serait possible de placer des fonctions agrégées définies par l'utilisateur en mémoire afin d'éviter de déplacer de grandes quantités de données sur le réseau et d'éviter les surcoûts de sérialisation et de désérialisation. Ceci pourrait aider à réduire la latence des opérations basées sur les fenêtres et à augmenter le débit de traitement.

Pour permettre le traitement natif des requêtes, un changement d'approche est nécessaire : les moteurs de traitement doivent se concentrer sur la manière de transformer les données et éviter les mécanismes complexes de gestion d'état (*operators state*) à ce niveau qui sont souvent utilisés actuellement. Nous pensons qu'il est plus efficace de laisser cette fonction à un moteur spécialisé de stockage de flux tel que celui que nous proposons.

Ce sont toutes ces observations qui ont fondatementalement contribué à la définition de

l'ensemble de principes de conception.

Principes de conception d'une architecture unifiée et évolutive pour l'ingestion de données et le stockage

En nous fondant sur nos expériences avec les environnements d'analyse Big Data et compte tenu des critères identifiés pour une solution dédiée d'ingestion de flux et de stockage, nous proposons un ensemble de principes de conception pour la construction d'une architecture unifiée et évolutive pour l'ingestion des données et leur stockage pour rendre plus efficaces les systèmes d'analyse Big Data. Dans un premier temps, en accord avec notre vision d'une architecture d'analyse unifiée, nous proposons que les moteurs de traitement ne s'occupent que du flux de travail des opérateurs afin qu'un moteur unifiée d'ingestion et de stockage s'occupe de la fonction de gestion de l'état (*operators state*), tout en se concentrant sur la gestion haut niveau des données (e.g., *caching*, gestion des accès concurrents où *concurrency control*). De plus, les moteurs de traitement et ingestion/stockage devraient inteéragir par des interfaces basées sur le stream.

Les systèmes d'ingestion et de stockage et les moteurs de traitement doivent comprendre les interfaces natives à base de flux pour l'ingestion des données, leur traitement et le stockage de flux d'enregistrements d'une manière compatible avec les modèles standard de traitement par lots.

1. ***L'interface d'ingestion de données*** est exploitée par les producteurs de flux qui écrivent des flux d'entrée, mais également par les serveurs du moteur de traitement qui stockent l'état de traitement (flux liés) dans des instances de stockage locales.

2. ***L'interface de stockage de données*** est gérée en interne par le système de stockage. Elle est utilisée pour stocker en permanence les flux d'entrée en cas de besoin ; cette action peut être effectuée de manière asynchrone en fonction des métadonnées de flux et des indications envoyées par le moteur de traitement.

3. ***L'interface de traitement de données*** est exposée de manière bidirectionnelle : premièrement, le moteur de traitement peut l'utiliser pour extraire des données du stockage de flux ; deuxièmement, nous envisageons la possibilité future d'avoir des flux de travail envoyant autant que possible des fonctions de traitement au stockage.

De plus, pour répondre aux exigences identifiées précédemment, nous envisageons la création, en plus de l'architecture unifiée d'ingestion et de stockage, d'interfaces de type clé/valeur (*key-value*) (e.g., *put/get*, multi-écriture/multi-lecture) nécessaires pour fournir un accès à granularité fine aux flux ingérés. L'amélioration des solutions de stockage avec des capacités d'ingestion aidera également à développer des flux de travail complexes basés sur les flux en permettant de transmettre les résultats de flux agrégés intermédiaires et finaux à d'autres applications de gestion de flux. Elle permettra aussi de mieux prendre en charge les techniques de point de contrôle de flux en stockant efficacement les résultats temporaires.

Les systèmes d'ingestion de flux les plus récentes [49, 91, 96] utilisent un schéma de partitionnement statique dans lequel le flux est divisé en un nombre fixe de partitions, chacune étant une séquence illimitée, ordonnée et immuable d'enregistrements qui sont continuellement ajoutés. Chaque courtier (*broker*) est responsable d'une ou de plusieurs partitions.

Les producteurs accumulent des enregistrements par lots de taille fixe, chacun étant ajouté à une partition. Pour réduire les coûts de communication, les producteurs regroupent plusieurs lots correspondant aux partitions d'un seul courtier dans une seule demande. Chaque consommateur est affecté à une ou plusieurs partitions. Chaque partition est transmise à un seul consommateur. Ceci élimine le besoin de mécanismes de synchronisation complexes, mais présente un inconvénient important : l'application doit connaître a priori le nombre optimal de partitions.

Cependant, dans les situations réelles, il est difficile de connaître a priori le nombre optimal de partitions, car celui-ci dépend de nombreux facteurs : nombre de courtiers, nombre de consommateurs et de producteurs, taille du réseau, estimation de l'objectif de débit de traitement et d'ingestion, etc. De plus, les producteurs et les consommateurs peuvent avoir un comportement dynamique qui conduit à un écart important entre le nombre optimal de partitions nécessaires à différents moments de l'exécution. Par conséquent, les utilisateurs ont tendance à surestimer le nombre de partitions pour couvrir le pire des scénarios dans lequel un grand nombre de producteurs et de consommateurs doivent accéder aux enregistrements simultanément, ce qui peut entraîner des dépenses inutiles. De plus, un nombre fixe de partitions peut également devenir une source de déséquilibre : chaque partition étant assignée à un seul consommateur, il est possible qu'une partition accumule ou libère des enregistrements plus rapidement que les autres partitions si elle est assignée à un consommateur plus rapide que les autres consommateurs.

En outre, les courtiers streaming attribuent à chaque enregistrement d'une partition un identifiant croissant de façon monotone, appelé décalage de partition (*partition offset*). Il permet aux applications d'accéder de manière arbitraire au contenu d'une partition en spécifiant le décalage. Les applications de diffusion en continu accèdent normalement aux enregistrements dans un ordre séquentiel, mais l'accès arbitraire permet la récupération en cas d'erreur. Plus précisément, un consommateur qui a échoué peut revenir à un point de contrôle précédent et revoir les enregistrements à partir du dernier décalage auquel son état a été contrôlé. De plus, l'utilisation des décalages lors de l'accès à des enregistrements permet au courtier de rester sans état vis-à-vis des consommateurs. Toutefois, la prise en charge d'un accès aléatoire efficace n'est pas gratuite : l'attribution d'un décalage à chaque enregistrement avec une granularité aussi fine dégrade les performances d'accès et occupe plus de mémoire. De plus, étant donné que les enregistrements sont demandés par lots, chaque lot sera plus volumineux en raison des décalages, ce qui produit une surcharge du réseau.

Nous proposons principalement deux idées en vue d'une ingestion des données évolutive : (1) un partitionnement dynamique fondé sur un regroupement sémantique et sur le sous-partitionnement, ce qui favorise une gestion plus flexible et élastique des partitions de flux (*stream partitions*) ; (2) une indexation de compensation légère, optimisée pour un accès aux séquentiel aux enregistrements. Par la suite, nous décrivons un ensemble de principes pour la prise en charge des divers modèles d'accès : accès à faible latence aux enregistrements de flux (*stream records*) et/ou accès à haut débit aux flux ou aux objets. Pour gérer efficacement des flux multiples, nous proposons une nouvelle méthode pour une réplication à granularité fine ayant une grande capacité d'adaptation et permettant l'ingestion durable des flux multiples.

La réplication est la solution standard utilisée pour garantir une ingestion de données de flux tolérante aux pannes. Chaque flux ingère des données dans plusieurs sous-partitions. Nous proposons l'organisation logique de chaque flux en un ou plusieurs journaux virtuels

qui agrègent en permanence les demandes contenant plusieurs fragments des producteurs. Il faut au moins un journal pour chaque partition d'un flux afin de les répliquer durablement. Les entités de sauvegarde chargées de stocker durablement les journaux virtuels répliqués peuvent continuer à voir les données en mémoire et sur disque sous forme de journaux structurés.

Un système d'ingestion et de stockage évolutif doit pouvoir accueillir efficacement plusieurs hôtes, chacun poussant plusieurs flux de données avec des exigences différentes en termes de débit d'ingestion et de latence d'accès en lecture-écriture. Il doit pouvoir par exemple prendre en charge l'ingestion de dizaines de flux très importants, c'est-à-dire ayant des dizaines de milliers de partitions, mais aussi l'ingestion de millions de très petits flux, c'est-à-dire ayant quelques partitions. Nous proposons de prendre en charge additionnellement la personnalisation du débit de réplication d'un flux unique en permettant au système et aux utilisateurs d'ajuster la capacité de réplication, c'est-à-dire le nombre de journaux virtuels répliqués pouvant être créés pour un flux unique. La réplication devrait être possible en mode synchrone et asynchrone à la fois. Par exemple, pour les applications nécessitant une ingestion plus rapide avec des exigences de cohérence plus faibles, les données doivent être répliquées de manière asynchrone. Au contraire, pour les applications qui préfèrent une grande durabilité à une consistance souple, les données doivent être répliquées de manière durable et synchrone avant que les producteurs ne reçoivent confirmation de la bonne ingestion et que les consommateurs les récupèrent pour les traiter.

Enfin, nous discutons deux principes de conception : soutien local pour les données et gestion distribuée des métadonnées. Ces principes peuvent mener à une intégration optimisée du système d'ingestion et de stockage avec les moteurs de traitement.

À cette fin, nous proposons que le système d'ingestion et de stockage ait un contrôle à granularité fine des flux acquis et évite de faire appel à des tiers pour la persistance des données comme le fait Kafka en exploitant le cache du cœur du système d'exploitation. Ensuite, la co-localisation des moteurs de traitement avec les nœuds d'ingestion et de stockage devrait utiliser des tampons mémoire partagés, tandis que les consommateurs de flux pourraient être implémentés avec une approche *push*, par opposition à une approche *pull*, améliorant ainsi le débit et réduisant considérablement la latence de traitement.

Nous proposons de gérer les métadonnées sur le courtier fournissant accès à ses données correspondantes : nous distribuons ainsi les métadonnées de manière naturelle sur l'ensemble du *cluster* de nœuds. Les métadonnées doivent pouvoir être interrogées indépendamment des données. Afin d'améliorer les performances, les métadonnées de partition de flux doivent être stockées avec les données qu'elles caractérisent : de cette manière, les métadonnées peuvent être reconstituées facilement et de manière dynamique lorsque les données sont récupérées ou migrées sur d'autres courtiers. En outre, lorsque cela est nécessaire pour augmenter et réduire le moteur d'ingestion et de stockage afin de gérer efficacement des différentes charges de travail, nous ne devrions que migrer les métadonnées décrivant les données actuelles et éviter de déplacer des données de flux.

La réalisation d'un prototype pour une architecture unifiée pour l'ingestion des données et le stockage

Sur la base des principes de conception mentionnés précédemment, nous présentons l'architecture *KerA*, un moteur optimisé d'ingestion et de stockage pour le traitement Big Data.

Nous rappelons au lecteur trois objectifs critiques qui ont été au cœur de l'architecture actuelle de *KerA*.

1. Permettre un accès à granularité fine aux flux ingérés et intermédiaires gérés par des applications de diffusion en continu ; c'est-à-dire, permettre la gestion de l'état via le moteur d'ingestion et de stockage.

2. Améliorer le support des applications de diffusion en continu nécessitant des réponses plus rapides que celles proposées par les systèmes actuels d'ingestion de pointe tels qu'Apache Kafka.

3. Réduire considérablement le stockage des données et l'utilisation du réseau, ce qui peut contribuer à réduire les délais de traitement et d'archivage des flux.

Nous exposons notre modèle de données unifié pour des flux illimités, enregistrements et objets, puis nous décrivons les protocoles de partitionnement dynamique et les mécanismes légers d'indexation des décalages (*offsets*) pour l'ingestion évolutive. Nous détaillons l'architecture *KerA* en décrivant le rôle de chaque composant de la gestion de l'ingestion des flux et du stockage, puis nous décrivons les interfaces client exposées par le moteur unifié d'ingestion et de stockage. Finalement, nous discutons les éléments de conception requis pour une mise en œuvre architecturale tolérante aux pannes.

Les systèmes (de stockage) tolérants aux pannes sont en mesure de s'acquitter de leur fonction en permanence, malgré les erreurs. Les utilisateurs d'applications basées sur des flux requièrent des réponses à latence faible (quelques secondes) quelles que soient les conditions. Afin de garantir le respect de ces exigences strictes, nous devons utiliser des techniques reconnues pour récupérer les données aussi rapidement que possible. Pour mettre en œuvre une récupération rapide après incident dans un traitement continu à latence faible, nous devons nous appuyer sur des techniques similaires à celles développées par RAMCloud [83]. Elle permettent en effet de tirer parti de la bande passante agrégée du disque pour récupérer en quelques secondes les données d'un nœud perdu. Le modèle de partitionnement à granularité fine de *KerA* favorise cette technique de récupération. Cependant, il ne peut pas être utilisé en tant que tel ; les producteurs doivent ajouter en permanence des enregistrements et ne pas subir les conséquences de la perte de courtiers, tandis que les consommateurs ne doivent pas attendre que toutes les données soient récupérées car cela entraînerait des latences élevées. Au lieu de cela, la récupération doit être obtenue en exploitant le décalage des applications des consommateurs.

Nous nous inspirons de la technique de migration décrite précédemment. En cas erreur d'un courtier, nous proposons de récupérer immédiatement les métadonnées du *streamlet* à partir des métadonnées de la sauvegarde et de permettre à nouveau aux producteurs de pousser les segments suivants du flux. En parallèle, nous procédons à la récupération des données de la manière suivante. Toutes les secondes, pour chaque groupe de consommateurs, nous stockons les derniers décalages utilisés dans les demandes de lecture et d'indication des consommateurs. Sur la base des dernièrs décalages de consommation, nous récupérons d'abord les groupes non traités. Ensuite, nous procédons à la récupération des autres groupes traités. De cette manière, les lecteurs continuent d'extraire des données à traiter, même si ceci est limité par la vitesse de récupération.

Ensuite nous décrivons la mise en œuvre des techniques *KerA* que nous avons développées afin d'élaborer un prototype logiciel de haute performance basé sur C++ et Java. Nous

examinons plus en détail la mise en œuvre de la réplication qui utilise une technique sans copie pour les bases de données en mémoire (*zero-copy virtual log technique*) et nous décrivons l'extension architecturale et la mise en œuvre du support local pour les données des opérations de gestion de flux. Nous évaluons notre mise en œuvre par des charges de travail synthétiques.

Nous évaluons l'impact des techniques d'ingestion dynamique, de réplication et de localisation sur les performances des producteurs et des consommateurs s'exécutant simultanément sur plusieurs courtiers. Nous exécutons d'abord un ensemble de micro-tests synthétiques en balayant un ensemble important de paramètres afin d'évaluer les performances de base d'un producteur et d'un consommateur, mesurées en termes de débit d'ingestion et de traitement. Ensuite, dans une configuration distribuée avec plusieurs courtiers et des producteurs et consommateurs concurrents, nous comparons *KerA* et Kafka en nous concentrant sur le composant d'ingestion avec la réplication désactivée. L'objectif est de démontrer l'efficacité du modèle de partitionnement de flux dynamique de *KerA* et des techniques allégées d'indexation des décalages. En outre, nous explorons l'efficacité de la mise en œuvre de la réplication de journaux virtuels et nous évaluons l'impact de diverses configurations sur le débit d'ingestion et de traitement. Enfin, nous évaluons l'impact de la localité de l'architecture sur les performances des lectures et écritures en flux, grâce à la réduction des interférences entre les lectures RPC d'un côté et les écritures et la réplication RPC de l'autre.

Apache Kafka et d'autres systèmes d'ingestion similaires (e.g., Amazon Kinesis [52], MapR Streams [70]) fournissent une fonctionnalité de publication-souscription pour les flux de données en partitionnant de manière statique un flux avec un nombre fixe de partitions. Pour faciliter la prise en charge de futures charges de travail plus élevées et pour une meilleure évolutivité du consommateur, ces systèmes surestiment le nombre de partitions nécessaires. Au contraire, le composant d'ingestion de *KerA* permet une gestion élastique des ressources au moyen de ruisseaux (*streamlets*) ce qui permet de stocker un nombre illimité de sous-partitions de taille fixe. De plus, pour pallier le problème de l'indexation inutile des décalages, les clients de *KerA* construisent de manière dynamique un décalage au niveau de l'application en fonction des métadonnées du groupe de ruisseaux diffusés par les courtiers via des RPC. Aucun des systèmes d'ingestion modernes n'est conçu pour tirer parti des optimisations de la localité des données prises en compte par *KerA*. De plus, grâce à sa mise en œuvre indépendante du réseau [86], *KerA* peut profiter des réseaux rapides émergents et du RDMA, offrant des lectures et des écritures plus efficaces que l'utilisation de TCP/IP.

Publications

Publications dans des conférences internationales

[ICDCS2018] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez, Bogdan Nicolae, Radu Tudoran, Stefano Bortoli. ***KerA: Scalable Data Ingestion for Stream Processing***. In IEEE International Conference on Distributed Computing Systems, Jul 2018, Vienna, Austria, https://hal.inria.fr/hal-01773799.w

[Cluster2016] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez. ***Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks***. In IEEE International Conference on Cluster Computing, Sep 2016, Taipei, Taiwan, https://hal.inria.fr/hal-01347638v2.

Publications dans des workshops internationaux

[BigData2017] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez, Radu Tudoran, Stefano Bortoli, Bogdan Nicolae. ***Towards a Unified Storage and Ingestion Architecture for Stream Processing***. In Second Workshop on Real-time & Stream Analytics in Big Data Colocated with the 2017 IEEE International Conference on Big Data, Dec 2017, Boston, United States, https://hal.inria.fr/hal-01649207.

[CCGrid2017] Ovidiu-Cristian Marcu, Radu Tudoran, Bogdan Nicolae, Alexandru Costan, Gabriel Antoniu, María Pérez. ***Exploring Shared State in Key-Value Store for Window-Based Multi-Pattern Streaming Analytics***. In Workshop on the Integration of Extreme Scale Computing and Big Data Management and Analytics in conjunction with IEEE/ACM CCGrid, May 2017, Madrid, Spain, https://hal.inria.fr/hal-01530744.

Titre : KerA: Un Système Unifié d'Ingestion et de Stockage pour le Traitement Efficace du Big Data

Mots clés : Big Data, Streaming, Ingestion, Stockage, Partitionnement dynamique, Données en premier

Résumé : Le Big Data est maintenant la nouvelle ressource naturelle. Les architectures actuelles des environnements d'analyse des données massives sont constituées de trois couches: les flux de données sont acquis par la couche d'ingestion (e.g., Kafka) pour ensuite circuler à travers la couche de traitement (e.g., Flink) qui s'appuie sur la couche de stockage (e.g., HDFS) pour stocker des données agrégées ou pour archiver les flux pour un traitement ultérieur. Malheureusement, malgré les bénéfices potentiels apportés par les couches spécialisées (e.g., une mise en oeuvre simplifiée), déplacer des quantités importantes de données à travers ces couches spécialisées s'avère peu efficace: les données devraient être acquises, traitées et stockées en minimisant le nombre de copies. Cette thèse propose la conception et la mise en oeuvre d'une architecture unifiée pour l'ingestion et le stockage de flux de données, capable d'améliorer le traitement des applications Big Data. Cette approche minimise le déplacement des données à travers l'architecture d'analyse, menant ainsi à une amélioration de l'utilisation des ressources. Nous identifions un ensemble de critères de qualité pour un moteur dédié d'ingestion des flux et stockage. Nous expliquons l'impact des différents choix architecturaux Big Data sur la performance de bout en bout. Nous proposons un ensemble de principes de conception d'une architecture unifiée et efficace pour l'ingestion et le stockage des données. Nous mettons en oeuvre et évaluons le prototype KerA dans le but de gérer efficacement divers modèles d'accès: accès à latence faible aux flux et/ou accès à débit élevé aux flux et/ou objets.

Title : KerA : A Unified Ingestion and Storage System for Scalable Big Data Processing

Keywords : Big Data, Streaming, Ingestion, Storage, Dynamic partitioning, Data first

Abstract : Big Data is now the new natural resource. Current state-of-the-art Big Data analytics architectures are built on top of a three layer stack: data streams are first acquired by the ingestion layer (e.g., Kafka) and then they flow through the processing layer (e.g., Flink) which relies on the storage layer (e.g., HDFS) for storing aggregated data or for archiving streams for later processing. Unfortunately, in spite of potential benefits brought by specialized layers (e.g., simplified implementation), moving large quantities of data through specialized layers is not efficient: instead, data should be acquired, processed and stored while minimizing the number of copies. This dissertation argues that a plausible path to follow to alleviate from previous limitations is the careful design and implementation of a unified architecture for stream ingestion and storage, which can lead to the optimization of the processing of Big Data applications. This approach minimizes data movement within the analytics architecture, finally leading to better utilized resources. We identify a set of requirements for a dedicated stream ingestion/storage engine. We explain the impact of the different Big Data architectural choices on end-to-end performance. We propose a set of design principles for a scalable, unified architecture for data ingestion and storage. We implement and evaluate the KerA prototype with the goal of efficiently handling diverse access patterns: low-latency access to streams and/or high throughput access to streams and/or objects.