KerA: Scalable Data Ingestion for Stream Processing

Ovidiu-Cristian Marcu*, Alexandru Costan*, Gabriel Antoniu*, María S. Pérez-Hernández§
Bogdan Nicolae†, Radu Tudoran‡, Stefano Bortoli‡
*INRIA Rennes Bretagne Atlantique, {ovidiu-cristian.marcu, alexandru.costan, gabriel.antoniu}@inria.fr
†Argonne National Laboratory, bogdan.nicolae@acm.org
‡Huawei Germany Research Center, {radu.tudoran, stefano.bortoli}@huawei.com
§Universidad Politecnica de Madrid, mperez@fi.upm.es

***Abstract*—Big Data applications are increasingly moving from batch-oriented execution models to stream-based models that enable them to extract valuable insights close to real-time. To support this model, an essential part of the streaming processing pipeline is *data ingestion*, i.e., the collection of data from various sources (sensors, NoSQL stores, filesystems, etc.) and their delivery for processing. Data ingestion needs to support high throughput, low latency and must scale to a large number of both data producers and consumers. Since the overall performance of the whole stream processing pipeline is limited by that of the ingestion phase, it is critical to satisfy these performance goals. However, state-of-art data ingestion systems such as Apache Kafka build on static stream partitioning and offset-based record access, trading performance for design simplicity. In this paper we propose *KerA*, a data ingestion framework that alleviate the limitations of state-of-art thanks to a dynamic partitioning scheme and to lightweight indexing, thereby improving throughput, latency and scalability. Experimental evaluations show that KerA outperforms Kafka up to 4x for ingestion throughput and up to 5x for the overall stream processing throughput. Furthermore, they show that KerA is capable of delivering data fast enough to saturate the big data engine acting as the consumer.**

Keywords*—*Stream processing, dynamic partitioning, ingestion.

I. Introduction

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

Since users are interested in obtaining results as soon as possible, there is a need to minimize the end-to-end latency of the three stage pipeline. This is a non-trivial challenge when records arrive at a fast rate and create the need to support a high throughput at the same time. To this purpose, Big Data engines are typically designed to scale to a large number of simultaneous consumers, which enables processing for millions of records per second [1], [2]. 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.

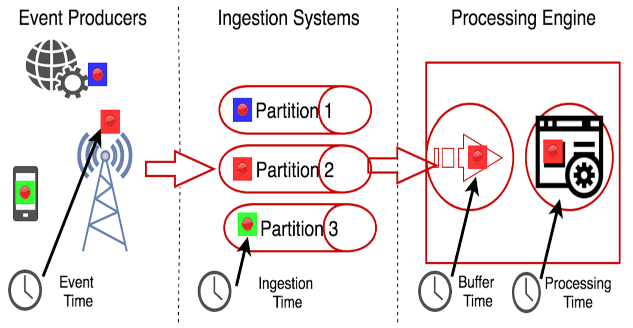

Fig. 1. Stream processing pipeline: records are collected at *event time* and made available to consumers earliest at *ingestion time*, after the events are acknowledged by producers; processing engines continuously pull these records and buffer them at *buffer time*, and then deliver them to the processing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

Achieving all these objectives simultaneously is challenging, which is why Big Data applications typically rely on specialized ingestion runtimes to implement the ingestion phase. One such popular runtime is Apache Kafka [3]. It quickly rose as the de-facto industry standard for record brokering in end-to-end streaming pipelines. It follows a simple design that allows users to manipulate streams of records similarly to a message queue. More recent ingestion systems (e.g., Apache Pulsar [4], DistributedLog [5]) provide additional features such as durability, geo-replication or strong consistency but leave little room to take advantage of trade-offs between strong consistency and high performance.

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

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

We summarize our contributions as follows: (1) we identify and study the key aspects that influence the performance and scalability of data processing in the ingestion phase (Section II); (2) we introduce a set of design principles that optimize the stream partitioning and record access (Section III); (3) we introduce the KerA prototype, which illustrates how to implement the design principles in a real-life solution (Section IV); (4) we demonstrate the benefits of KerA experimentally using state-of-art ingestion systems as a baseline (Section V).

II. Background: Stream Ingestion

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

A. Static partitioning

State-of-art stream ingestion systems (e.g., [3], [4], [5]) employ a static partitioning scheme where the stream is split among a fixed number of partitions, each of which is an unbounded, ordered, immutable sequence of records that are continuously appended. Each broker is responsible for one or multiple partitions. Producers accumulate records in fixed-sized batches, each of which is appended to one partition. To reduce communication overhead, the producers group together multiple batches that correspond to the partitions of a single broker in a single request. Each consumer is assigned to one or more partitions. Each partition assigned to a single consumer. This eliminates the need for complex synchronization mechanisms but has an important drawback: the application needs a priori knowledge about the optimal number of partitions.

However, in real-life situations it is difficult to know the optimal number of partitions a priori, both because it depends on a large number of factors (number of brokers, number of consumers and producers, network size, estimated ingestion and processing throughput target, etc.). In addition, the producers and consumers can exhibit dynamic behavior that can generate large variance between the optimal number of partitions needed at different moments during the runtime. Therefore, users tend to over-provision the number of partitions to cover the worst case scenario where a large number of producers and consumers need to access the records simultaneously. However, if the worst case scenario is not a norm but

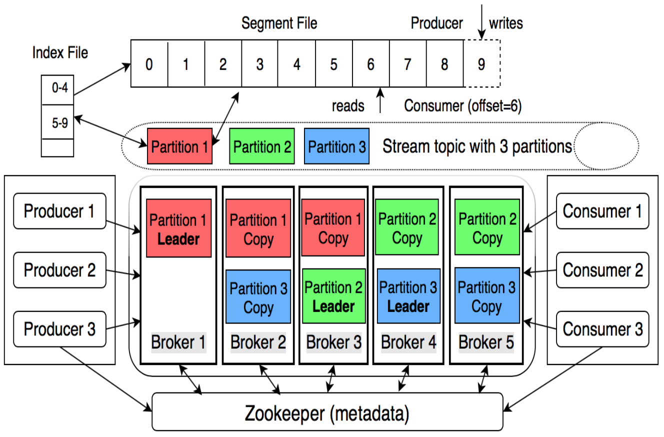

Fig. 2. Kafka's architecture (illustrated with 3 partitions, 3 replicas and 5 brokers.). Producers and consumers query Zookeeper for partition metadata (i.e., on which broker a stream partition leader is stored). Producers append to the partition's leader (e.g., broker 1 is assigned partition 1 leader),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

an exception, this can lead to significant unnecessary overhead. Furthermore, a fixed number of partitions can also become a source of imbalance: since each partition is assigned to a single consumer, it can happen that one partition accumulates or releases records faster than the other partitions if it is assigned to a consumer that is slower or faster than the other consumers.

For instance, in Kafka, a stream is created with a fixed number of partitions that are managed by Kafka's brokers, as depicted in Figure 2. Each partition is represented by an index file for offset positioning and a set of segment files, initially one, for holding stream records. Kafka leverages the operating system cache to serve partition's data to its clients. Due to this design it is not advised to collocate streaming applications on the same Kafka nodes, which does not allow to leverage data locality optimizations [6].

B. Offset-based record access

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

III. Design Principles for Stream Ingestion

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

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

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

IV. KerA: Overview and Architecture

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

A. Dynamic partitioning model and offset indexing

KerA implements dynamic partitioning based on the concept of *streamlet* (Figure 3), 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 a single producer and consumer. A streamlet can have an arbitrary number of groups created as needed, each of which can grow up to a maximum predefined size. To facilitate the management of groups and offsets in an efficient fashion, each group is further split into fixed-sized *segments*. The maximum size of a group is a multiple of segment size $P \geq 1$. To control the level of parallelism allowed on each broker, only $Q \geq 1$ groups can be active at a given moment. 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 groups (sub-partitions) to process.

In order to ensure ordering semantics, each streamlet dynamically creates groups (and their segments, initially one) that have unique, monotonically increasing identifiers. Brokers expose this information through RPCs (*getNextAvailableGroupsForStreamlets, getNextAvailableSegmentsForGroups*) 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. The consumer initializes it to 0 (broker understands to iterate to first record available in that segment) and the broker responds with the last record position for each new request, so the consumer can update its latest offset to start a future request with. Using this dynamic approach (as opposed to the static approach used by explicit offsets per partition, clients have to query brokers to discover groups), we implement lightweight offset indexing optimized for sequential record access.

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

B. Favoring parallelism: consumer and producer protocols

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

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

C. Architecture and implementation

KerA's architecture is similar to Kafka's (Figure 3): a single layer of brokers (nodes) serve producers and consumers. However, in KerA brokers are used to discover stream subpartitions. Kera builds atop RAMCloud's [7] framework in order to leverage its network abstraction that enables the use of other network transports (e.g., UDP, DPDK, Infiniband), whereas Kafka only supports TCP. Moreover, it allows KerA to benefit from a set of design choices like *polling and request dispatching* [8] that help boost performance (kernel bypass and zero-copy networking are possible with DPDK and Infiniband, that we leave for future evaluations). Kera consists of about 5K lines of C++ code for client and server side implementations.

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

To support durability and replication, and implement fast crash recovery techniques, it is possible to rely on RAMCloud [9], by leveraging the aggregated disk bandwidth in order to recover the data of a lost node in seconds. KerA's fine-grained partitioning model favors this recovery technique. However it cannot be used as such: producers should continuously append records and not suffer from broker crashes, while consumers should not have to wait for all data to be recovered (thus incurring high latencies). Instead, recovery can be achieved by leveraging consumers' application offsets. We plan to enable such support as future work.

V. Experimental Evaluation

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

A. Setup and parameter configuration

We ran all our experiments on Grid5000 *Grisou* cluster [10]. Each node has 16 cores and 128 GB of memory. In each experiment the source thread of each producer creates 50 million non-keyed records of 100 bytes, and partitions them

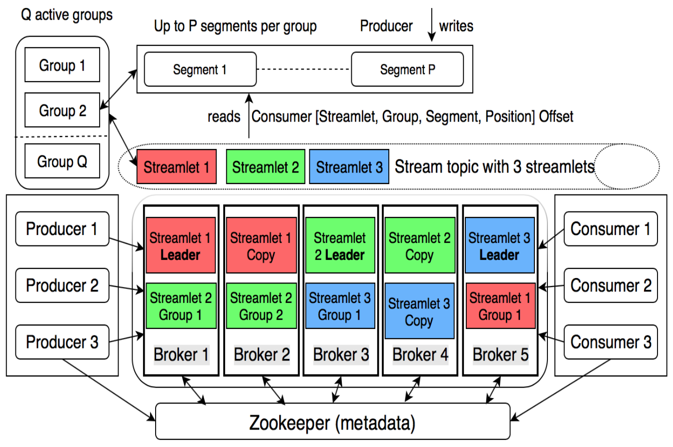

Fig. 3. KerA's architecture (illustrated with 3 streamlets and 5 brokers). Zookeeper is responsible for providing clients the metadata of the association of streamlets with brokers. Streamlets' groups and their segments are dynamically discovered by consumers querying brokers for the next available groups of a streamlet and for new segments of a group. Replication in Kera can leverage its fine-grained partitioning model (streamlet-groups) by replicating each group on distinct brokers or by fully replicating a streamlet's groups on another broker like in Kafka.

round-robin in batches of configurable size. The source waits no more than 1ms (parameter named *linger.ms* in Kafka) for a batch to be filled, after this timeout the batch is sent to the broker. Another producer thread groups batches in requests and sends them to the node responsible of the request's partitions (multi TCP synchronous requests). Similarly, each consumer pulls batches of records with one thread and simply iterates over records on another thread.

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

Each broker is configured with 16 network threads that corresponds to the number of cores of a node and holds one copy of the streamlet's groups (we plan to study pull-based versus push-based replication impact in future work). In each experiment we run an equal number of producers and consumers. The number of partitions/streamlets is configured to be a multiple of the number of clients, at least one for each client. Unless specified, we configure in KerA the number of active groups to 1 and the number of segments to 16. A request is characterized by its size (i.e., *request.size*, in bytes) and contains a set of batches, one for each partition/streamlet, each batch having a *batch.size* in bytes. We use Kafka 0.10.2.1 since it has a similar data model with KerA (newest release introduces batch entries for exactly once processing, a feature that could be efficiently enabled also in KerA [11]). A Kafka segment is 512 MB, while in KerA it is 8MB. This means that rolling to a new segment happens more often and may impact performance (since KerA's clients need to discover new segments before pulling data from them).

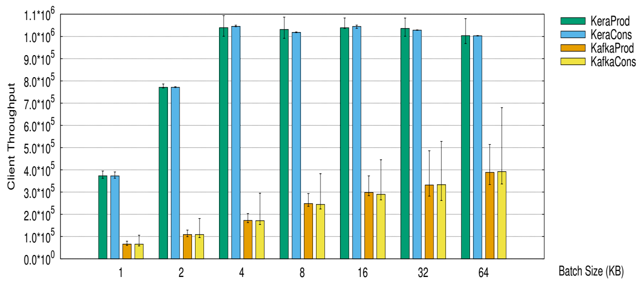

Fig. 4. Increasing the batch size (request size). Parameters: 4 brokers; 16 producers and 16 consumers; number of partitions/streamlets is 16; *request.size* equals *batch.size* multiplied by 4 (number of partitions per node). On X we have producer *batch.size* KB, for consumers we configure a value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

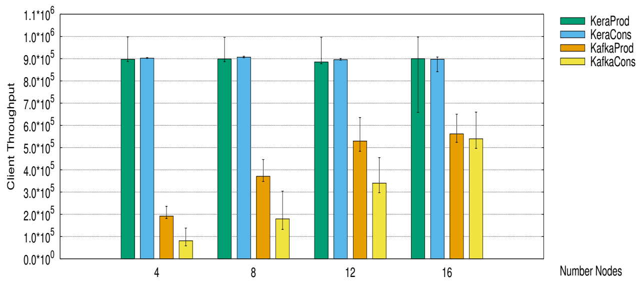

Fig. 6. Adding nodes (brokers). Parameters: 32 producers and 32 consumers; 256 partitions, 32 streamlets with 8 active groups per streamlet; *batch.size* = 16KB; *request.size* = *batch.size* multiplied by the number of partitions/active groups per node.

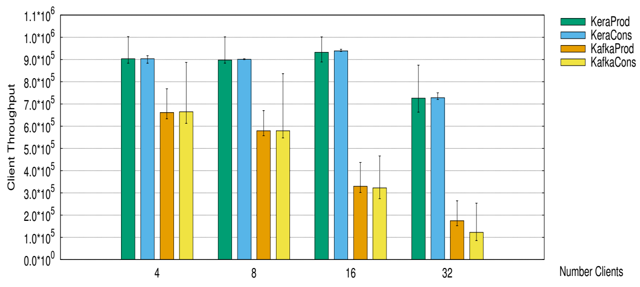

Fig. 5. Adding clients. Parameters: 4 brokers; 32 partitions/streamlets, 1 active group per streamlet; *batch.size* = 16KB; *request.size* = 128KB.

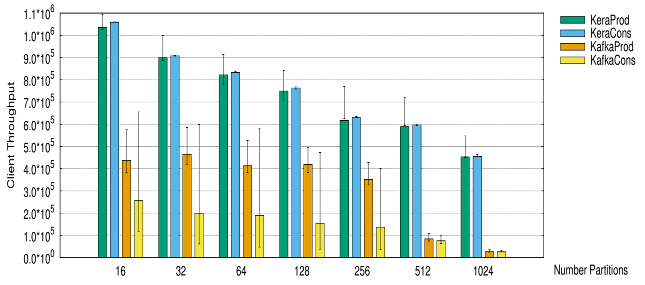

Fig. 7. Increasing the number of partitions and respectively streamlets. Parameters: 4 brokers; 16 producers and 16 consumers; *request.size* = 1MB; *batch.size* equals *request.size* divided by the number of partitions.

B. Results

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

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

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

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

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

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

VI. Related Work

Apache Kafka and other similar ingestion systems (e.g., Amazon Kinesis [12], MapR Streams [13]) 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-partitioned with a higher number of partitions. In contrast, KerA enables resource elasticity by means of streamlets, which enables storing an unbounded number of fixed-size 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.

DistributedLog [5] 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 [14]. A topic is partitioned into a fixed number of partitions, and each partition is backed by a log. Log segments are spread over multiple nodes (based on Bookkeeper [15]). 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. While KerA favors parallelism for writers appending to a streamlet (collection of groups), in DistributedLog there is only one active writer for a log at a given time.

Apache Pulsar [4] is a pub-sub messaging system developed on top of Bookkeeper, on a two-layer architecture composed of stateless serving layer and 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 subscriptions models to its clients [16]. Pulsar keeps track of consumer cursor position, being able to remove records once acknowledged by consumers. Similar to Pulsar/DistributedLog, KerA could leverage a second layer of brokers to cache streamlet's groups when needed to provide large fan-out bandwidth to multiple consumers of the same stream.

Pravega [17] 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.

None of the state-of-art ingestion systems are designed to leverage data locality optimizations as envisioned with KerA in a unified storage and ingestion architecture [18]. Moreover, thanks to its network agnostic implementation [7], KerA can benefit from emerging fast networks and RDMA, providing more efficient reads and writes than using TCP/IP.

VII. Conclusions

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

Acknowledgment

This research was funded by Huawei HIRP OPEN and the BigStorage project (H2020-MSCA-ITN-2014-642963, under the Marie Sklodowska-Curie action).

References

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

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

[3] "Apache Kafka." https://kafka.apache.org/.

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

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

[6] G. Németh, D. Géhberger, and P. Mátray, "DAL: A Locality-Optimizing Distributed Shared Memory System," in *9th USENIX HotCloud*, 2017.

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

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

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

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

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

[12] "Amazon Kinesis," https://aws.amazon.com/kinesis/data-streams/.

[13] "Mapr Streams," https://mapr.com/products/mapr-streams/.

[14] S. Guo, R. Dhamankar, and L. Stewart, "Distributedlog: A High Performance Replicated Log Service," in *33rd ICDE*, April 2017, pp. 1183–1194.

[15] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 1, pp. 9–15, Jan. 2013.

[16] "Messaging, storage, or both?" https://streaml.io/blog/messaging-storage-or-both/.

[17] "Pravega," http://pravega.io/.

[18] O.-C. Marcu, A. Costan, G. Antoniu, M. Perez, R. Tudoran, S. Bortoli, and B. Nicolae, "Towards a Unified Storage and Ingestion Architecture for Stream Processing," in *Second Workshop on Real-time and Stream Analytics in Big Data Collocated with the 2017 IEEE Big Data*, Dec.