In support of push-based streaming for the computing continuum

Ovidiu-Cristian Marcu, Pascal Bouvry

In support of push-based streaming for the computing continuum

Ovidiu-Cristian Marcu and Pascal Bouvry

University of Luxembourg, Luxembourg
{ovidiu-cristian.marcu,pascal.bouvry}@uni.lu

Abstract. Real-time data architectures are core tools for implementing the edge-to-cloud computing continuum since streams are a natural abstraction for representing and predicting the needs of such applications. Over the past decade, Big Data architectures evolved into specialized layers for handling real-time storage and stream processing. Open-source streaming architectures efficiently decouple fast storage and processing engines by implementing stream reads through a pull-based interface exposed by storage. However, how much data the stream source operators have to pull from storage continuously and how often to issue pull-based requests are configurations left to the application and can result in increased system resources and overall reduced application performance. To tackle these issues, this paper proposes a unified streaming architecture that integrates co-located fast storage and streaming engines through push-based source integrations, making the data available for processing as soon as storage has them. We empirically evaluate pull-based versus push-based design alternatives of the streaming source reader and discuss the advantages of both approaches.

Keywords: Streaming · Real-time storage · Push-based · Pull-based · Locality.

1 Introduction

The edge-to-cloud computing continuum [10] implements fast data storage and streaming *layered architectures* that are deployed intensively in both Cloud [9] and Fog architectures [22]. Fast data processing enables high-throughput data access to streams of logs, e.g., daily processing terabytes of logs from tens of billions of events at CERN accelerator logging service [1]. Moreover, implementing sensitive information detection with the NVIDIA Morpheus AI framework enables cybersecurity developers to create optimized AI pipelines for filtering and processing large volumes of real-time data [2].

Over the past decade, Big Data architectures evolved into specialized layers for handling real-time storage and stream processing. To efficiently decouple fast storage and streaming engines, architects design a streaming source interface that implements data stream reads through *pull-based* remote procedure calls (RPC) APIs exposed by storage. The streaming source operator continuously pulls data

from storage while its configuration (i.e., how much data to pull and how often to issue pull requests) is left to the application, being a source of bottlenecks and overall reduced application performance. In turn, decoupled streaming sources help manage backpressure [17] and simplifies fault-tolerance implementation for system crash management.

The pull-based integration approach is opposed to monolithic architectures [28] that have the opportunity to more efficiently and safely optimize data-related tasks. Another architectural choice is to closely integrate fast storage and streaming engines through *push-based* data sources, making the data available to the processing engine as soon as it is available. The push-based source integration approach should keep control of the data flow to ensure backpressure and should promote an easy integration through storage and processing non-intrusive extensions to further promote open-source real-time storage and streaming development.

Our challenge is then **how to design and implement a push-based streaming source strategy to efficiently and functionally integrate real-time storage and streaming engines** while keeping the advantages of a pull-based approach. Towards this goal, this paper introduces a push-based streaming design to integrate co-located real-time storage and processing engines. We implement pull-based and push-based stream sources as integration between open-source KerA [1], a real-time storage system, and Apache Flink [3], a stream processing engine. We empirically evaluate the KerA-Flink push-based and pull-based approaches and we show that the push-based approach can be competitive with a pull-based design while requiring reduced system resources. Furthermore, when storage resources are constrained, the push-based approach can be up to 2x more performant compared to a pull-based design.

2 Background and Related Work

Decoupling producers and consumers through message brokers (e.g., Apache Kafka [16]) can help applications through simplified real-time architectures. This locality-poor design is preferred over monolithic architectures by state-of-the-art open-source streaming architectures. Big Data frameworks that implement MapReduce [13] are known to implement data locality (pull-based) optimizations. General Big Data architectures can thus efficiently co-locate map and reduce tasks with input data, effectively reducing the network overhead and thus increasing application throughput. However, they are not optimized for low-latency streaming scenarios. User-level thread implementations such as Arachne [24] and core-aware scheduling techniques like Shenango, Caladan [23, 14] can further optimize co-located latency-sensitive stream storage and analytics systems, but are difficult to implement in practice.

Finally, it is well known that message brokers, e.g., Apache Kafka [16, 4], Apache Pulsar [5], Distributedlog [25], Pravega [8], or KerA [19], can contribute

[1] KerA-Flink integration source code: https://gitlab.uni.lu/omarcu/zettastreams.

to higher latencies in streaming pipelines [15]. Indeed, none of these open-source storage systems implement locality and thus force streaming engines to rely on a pull-based implementation approach for consuming data streams. Consistent state management in stream processing engines is difficult [12] and depends on real-time storage brokers to provide indexed, durable dataflow sources. Therefore, stream source design is critical to the fault-tolerant streaming pipeline and potentially a performance issue.

A pull-based source reader works as follows: it waits no more than a specific timeout before issuing RPCs to pull (up to a particular batch size) more messages from stream storage. One crucial question is how much data these sources have to pull from storage brokers and how often these pull-based RPCs should be issued to respond to various application requirements. Consequently, a push-based approach can better solve these issues by pushing the following available messages to the streaming source as soon as more stream messages are available. However, a push-based source reader is more difficult to design since coupling storage brokers and processing engines can bring back issues solved by the pull-based approach (e.g., backpressure, scalability). Thus, we want to explore a non-monolithic design that integrates real-time storage and streaming engines through a push-based stream source approach and understand the performance advantages (e.g., throughput) of both approaches. Towards this goal let us introduce next a push-based streaming design that unifies real-time storage and processing engines and describe our implementation.

3 Unified Real-time Storage and Processing Architecture: Our Push-based Design and Implementation

Background. Fast storage (e.g., [19, 20]) implements a layer of *brokers* to serve producers and consumers of data streams. As illustrated in Figure 1, a multi-threaded broker is configured with one dispatcher thread polling the network and responsible for serving read/write RPC requests and multiple working threads that do the actual writes and reads. Streaming engines [27, 21, 3] implement a layer of workers. E.g., in Apache Flink, each worker implements a JVM process that can host multiple slots (a slot can have one core). Sources and other operators are deployed on worker slots ([3, 6]) and are configured to use in-memory buffers. Backpressure is ensured as follows: sources continuously issue pull-based read requests as long as buffers are not filled.

Design principles. To ensure backpressure, our push-based design takes source buffers outside the processing engine and shares buffer control with storage and processing through pointers and notifications. To avoid network overhead, we propose to co-locate real-time storage and streaming engines. To remove storage interference of read and write RPC requests, we separate reads and writes through dedicated push worker threads. The push-based mechanism completely removes the RPC and networking overheads of the pull-based approach.

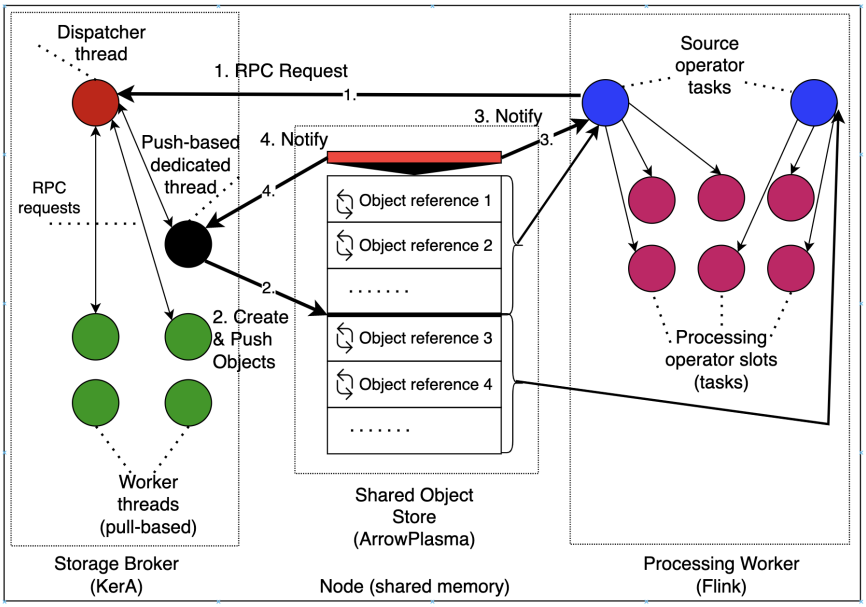

Fig. 1. Unified real-time storage and stream processing architecture. Storage broker and processing worker are co-located on a multi-core node while stream source operators implement push-based consumers through shared memory partitioned object store. At runtime, source tasks consume data from shared reusable memory objects filled by storage as data arrives.

Push-based architectural design. As illustrated in Figure 1, we co-locate a storage broker and one or multiple processing workers on a multi-core node. We propose to leverage a shared partitioned in-memory buffer between (push-based) streaming sources and storage brokers to provide backpressure support to streaming engines and to allow for transparent integration with various streaming storage and processing engines. Multiple sources coordinate to launch only one push-based RPC request (step 1). Storage creates (once) a dedicated worker thread (in black) that is responsible to continuously fill shared reusable objects with next stream data (step 2). Source tasks are notified when objects have new updates (step 3) and then process these data. The worker thread is notified (step 4) when an object was consumed, so it can reuse it an refill with new data. This flow (steps 2-4) executes continuously. Objects have a fixed-size and are shared through pointers by worker thread and source tasks. The source creates tuples and pushes them further to the stream processing operator tasks. Backpressure is ensured through shared store notifications.

Implementation . On the same node live three processes: the streaming broker, the processing worker and the shared partitioned object store. As illustrated in Figure 1, two push-based streaming source tasks are scheduled on one process-

ing worker. At execution time, only one of the two sources will issue (once!) the push-based RPC (e.g., based on the smallest of the source tasks' identifiers). This special RPC request (implemented by storage) contains initial partition offsets (partitioned streams) used by sources to consume the next stream records. The storage handles the push-based RPC request by assigning a worker thread responsible for creating and pushing the next chunks of data associated with consumers' partition offsets. We implement the shared-memory object store based on Apache Arrow Plasma, a framework that allows the creation of in-memory buffers (named objects) and their manipulation through shared pointers. Our push-based RPC is implemented on the KerA storage engine while we transparently integrate our KerA connector with Apache Flink for stream processing.

4 Evaluation

Our goal is to understand the performance advantages of push-based and pull-based streaming source integrations between real-time storage and streaming engines. While the pull-based approach simplifies implementation, the push-based approach requires a more tight integration. What performance benefits characterize each approach?

Experimental Setup and Parameter Configuration. We run our experiments on the Aion cluster [2] by deploying Singularity containers (for reproducibility) over Aion regular nodes through Slurm jobs. One Aion node has two AMD Epyc ROME 7H12 CPUs (128 cores), each with 256 GB of RAM, interconnected through Infiniband 100Gb/s network. Producers are deployed separately from the streaming architecture. The KerA broker is configured with up to 16 worker cores (for pull-based, while the push-based approach uses only one dedicated worker core) while the partition's segment size is fixed to 8 MiB. We use Apache Flink version 1.13.2. We configure several producers *Np* (respectively consumers *Nc*, values=1,2,4,8), similarly to [20], that send data chunks to a partitioned stream having *Ns* partitions. Each partition is consumed exclusively by its associated consumer. Producers concurrently push synchronous RPCs having one chunk of *CS* size (values=1,2,4,8,16,32,64,128 KiB) for each partition of a broker, having in total *ReqS* size. Each chunk can contain multiple records of configurable *RecS* size for the synthetic workloads. We configure producers to read and ingest Wikipedia files in chunks having records of 2 KiB. Flink workers correspond to the number of Flink slots *NFs* (values=8,16) and are installed on the same Singularity instance where the broker lives.

To understand previous parameters' impact on performance, we run each experiment for 60 to 180 seconds while we collect producer and consumer throughput metrics (records every second). We plot 50-percentile cluster throughput per second for each experiment (i.e., by aggregating the write/read throughput of each producer/consumer every second), and we compare various configurations.

[2] more details at https://hpc.uni.lu/infrastructure/supercomputers the Aion section

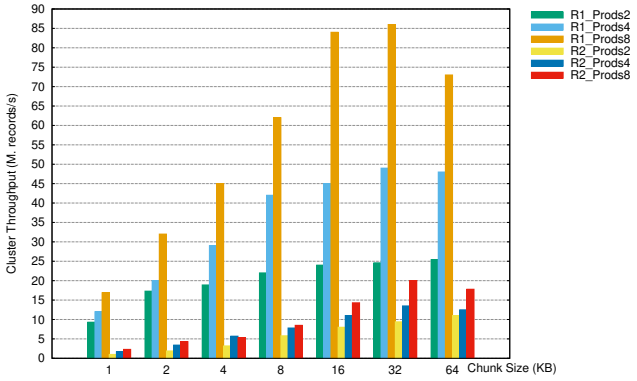

Fig. 2. Impact of replication and chunk size on ingestion cluster throughput (only producers). Parameters: Np = 2,4,8, RecS = 100 Bytes, one stream with Ns = 8 partitions. R1_Prods2 represents replication factor one and two concurrent producers writing to one stream. R2_Prods8 represents replication factor two and eight producers. Results are similar for a stream with Ns = 16.

Our second goal is to understand *who of the push-based and respectively pull-based streaming strategies is more performant, in what conditions, and what the trade-offs are in terms of configurations.*

Benchmarks. The first benchmark (relevant to use cases that transfer or duplicate partitioned datasets) implements a simple pass-over data, iterating over each record of partitions' chunks while counting the number of records. The second benchmark implements a filter function over each record, being a representative workload used in several real-life applications (*e.g.* indexing the monitoring data at the LHC [7]). The next benchmarks (CPU intensive) implement word count over Wikipedia datasets. For reproducibility, our benchmark code is open-source and a technical report [18] further describes these applications.

4.1 Results and Discussion

Synthetic benchmarks: the count operator. In our first evaluation, we want to understand how our chosen parameters can impact the aggregated throughput while ingesting through several concurrent producers. As illustrated in Figure 2, we experiment with two, four, and eight concurrent producers. Increasing the chunk size *CS*, the request size *ReqS* increases proportionally, for a fixed record size *RecS* of fixed value of 100 Bytes. While increasing the chunk size, we observe (as expected) that the cluster throughput increases; having more producers helps, although they compete at append time. We also observe that replication considerably impacts cluster throughput (as expected) since each producer has to wait for an additional replication RPC done at the broker side. Producers wait up to one millisecond before sealing chunks ready to be pushed to

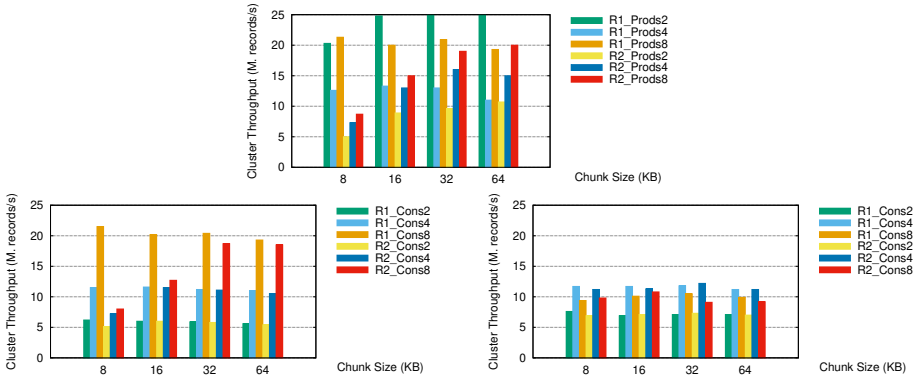

Fig. 3. Only producers (top). Pull-based (left) versus push-based (right) consumers for iterate and count benchmark for a stream with Ns = 8. Parameters: Np = Nc = 2,4,8, replication R = 1,2, consumer CS = 128 KiB, we plot producer CS = 8,16,32,64.

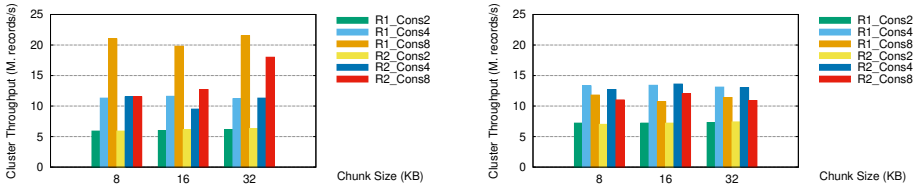

Fig. 4. Pull-based (left) versus push-based (right) consumers for iterate, count and filter benchmark. Parameters: Np = Nc = 2,4,8, replication R = 1,2, consumer CS = 128 KiB, we plot producer CS = 8,16,32,64, stream Ns = 8.

the broker (or the chunk gets filled and sealed) - this configuration can help trade-off throughput with latency. With only two producers, we can obtain a cluster throughput of ten million records per second, while we need eight producers to double this throughput. Next experiments introduce concurrent consumers in parallel with concurrent producers.

The subsequent evaluation looks at concurrently running producers and consumers and compares pull-based versus push-based consumers. The broker is configured with 16 working cores to accommodate up to eight producers and eight consumers concurrently writing and reading chunks of data. Since consumers compete with producers, we expect the producers' cluster throughput to drop compared to the previous evaluation that runs only concurrent producers. This is shown in Figure 3: due to higher competition to broker resources by consumers, producers obtain a reduced cluster throughput compared to the previous experiment. We observe that consumers fail to keep up with the producers' rate.

These experiments illustrates the impact on performance of the interference between reads and writes. Increasing the chunk size, we observe that pull-based consumers obtain better performance than push-based. Increasing the number of

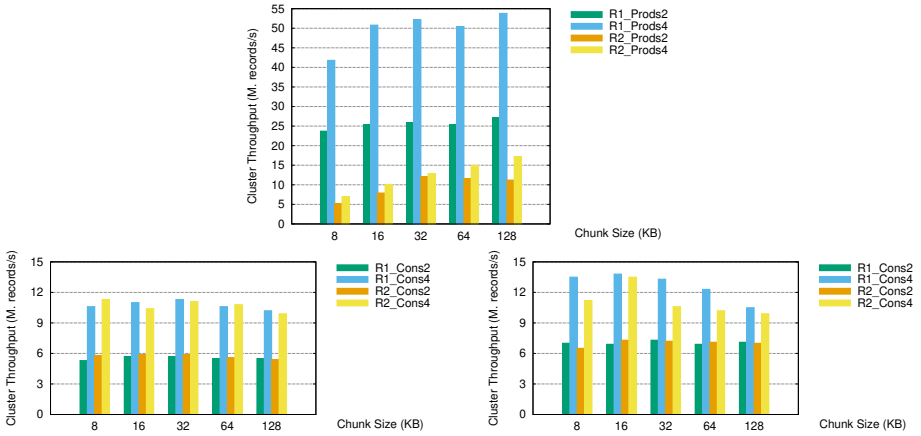

Fig. 5. Only producers (top). Pull-based (left) versus push-based (right) consumers for iterate, count and filter benchmark for a stream with 4 partitions.

consumers, we observe that the dedicated thread does not keep up with more than four consumers; otherwise, push-based is competitive with pull-based or slightly better. However, (although with eight consumers the pull-based strategy can obtain a better cluster throughput,) for up to four consumers the push-based strategy not only can obtain slightly better cluster throughput but the number of resources dedicated to consumers reduces considerably (two threads versus eight threads for the configuration with four consumers). While pull-based consumers double the cluster throughput when using 16 threads for the source operators, push-based consumers only use two threads for the source operator.

Synthetic Benchmarks: The Filter Operator. We further compare pull-based versus push-based consumers when implementing the filter operator, in addition, to counting for a stream with eight partitions. Similar to previous experiments, the push-based consumers are slower when scaled to eight for larger chunks, as illustrated in Figure 4. As illustrated in Figure 5, when experimenting with up to four producers and four consumers over a stream with four partitions, *the push-based strategy provides a cluster throughput slightly higher with smaller chunks, being able to process two million tuples per second additionally over the pull-based approach. With larger chunks, the throughput reduces: architects have to carefully tune the chunk size in order to get the best performance.* When experimenting with smaller chunks, more work needs to be done by pull-based consumers since they have to issue more frequently RPCs (see Figure 7). Moreover, the push-based strategy provides higher or similar cluster throughput than the pull-based strategy while using fewer resources.

Next, we design an experiment with constrained resources for the storage and backup brokers configured with four cores. We ingest data from four producers into a replicated stream (factor two) with eight partitions. We concurrently run

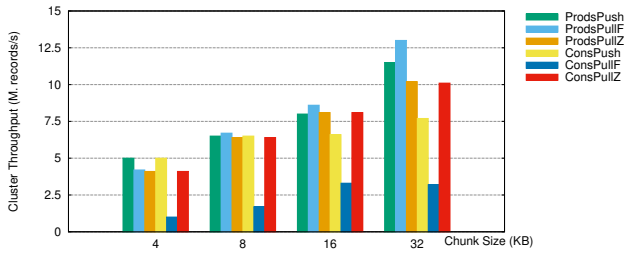

Fig. 6. Iterate, count and filter benchmark constrained broker resources. Comparing C++ pull-based consumers with Flink pull-based and push-based consumers. ProdsPush corresponds to producers running concurrently with push-based Flink consumers i.e. ConsPush. ProdsPullF corresponds to producers running concurrently with pull-based Flink consumers i.e. ConsPullF. ProdsPullZ corresponds to producers running concurrently with C++ pull-based consumers. Four producers and four consumers ingest and process a replicated stream (factor two) with eight partitions over one broker storage with four working cores. Consumer chunk size equals the producer chunk size.

four consumers configured to use Flink-based push and pull strategies and native C++ pull-based consumers. Consumers iterate, filter and count tuples that are reported every second by eight Flink mappers. We report our results in Figure 6 where we compare the cluster throughput of both producers and consumers. Producers compete directly with pull-based consumers, and we expect the cluster throughput to be higher when concurrent consumers use a push-based strategy. However, producers' results are similar except for the 32 KB chunk size when producers manage to push more data since pull-based consumers are slower. We observe that the C++ pull-based consumers can better keep up with producers while push-based consumers can keep up with producers when configured to use smaller chunks. *The push-based strategy for Flink is up to 2x better than the pull-based strategy of Flink consumers. Consequently, the push-based approach can be more performant for resource-constrained scenarios.*

Wikipedia Benchmarks: (Windowed) Word Count Streaming. For the following experiments, the producers are configured to read Wikipedia files in chunks with records of 2 KiB. Therefore, producers can push about 2 GiB of text in a few seconds. Consumers run for tens of seconds and do not compete with producers. As illustrated in Figure 8, pull-based and push-based consumers demonstrate similar performance. We plot word count tuples per second aggregated for eight mappers while scaling consumers from one to four. Results are similar when we experiment with smaller chunks or streams with more partitions since this benchmark is CPU-bound. *To avoid network bottlenecks when processing large datasets like this one (e.g., tens of GBs) on commodity clusters, the push-based approach can be more competitive when pushing pre-processing and local aggregations at the storage.*

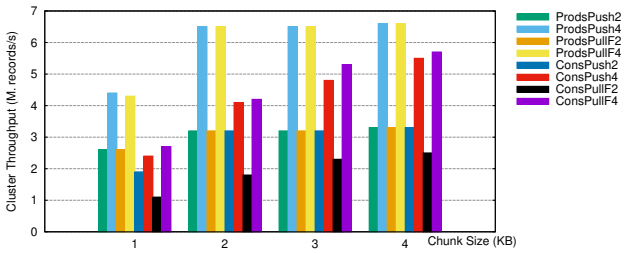

Fig. 7. Iterate and count benchmark stream with 8 partitions broker with 8 cores. Evaluation of pull-based and push-based Flink consumers. ProdsPush corresponds to producers running concurrently with push-based Flink consumers i.e. ConsPush. ProdsPullF corresponds to producers running concurrently with pull-based Flink consumers i.e. ConsPullF. Consumer chunk size equals the producer chunk size multiplied by 8. We plot producer chunk size.

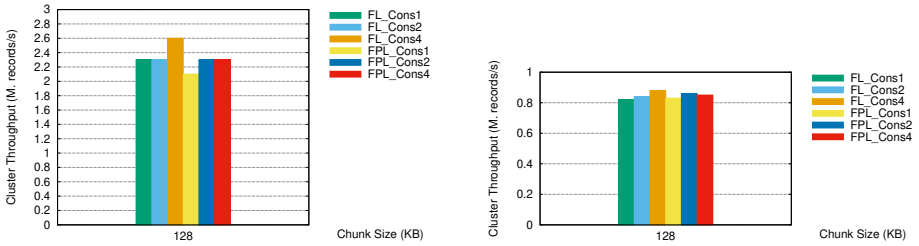

Fig. 8. Pull-based consumers versus push-based consumers for the word count benchmarks with 4 partitions. The left figure presents the word count benchmarks, the right figure corresponds to the windowed word count benchmark. FLCons2 represents two push-based consumers while FPLCons4 represents four pull-based consumers.

Discussion and Future Implementation Optimizations. Regarding our prototype implementation, we believe there is room for further improvements. One future step is integrating the shared object store and notifications mechanism inside the broker storage implementation. This choice will bring up two potential optimizations. Firstly, it would avoid another copy of data by leveraging existing in-memory segments that store partition data (necessary for high-throughput use cases). Secondly, we could optimize latency by implementing the notification mechanism through the asynchronous RPCs available in KerA. Furthermore, implementing pre-processing functions in-storage (e.g., as done in [11]) can further improve performance by reducing data movement.

5 Conclusion

We have proposed a unified real-time storage and processing architecture that leverages a push-based strategy for streaming consumers. Experimental evalua-

tions show that when storage resources are enough for concurrent producers and consumers, the push-based approach is performance competitive with the pull-based one (as currently implemented in state-of-the-art real-time architectures) while consuming fewer resources. However, when the competition of concurrent producers and consumers intensifies and the storage resources (i.e., number of cores) are more constrained, the push-based strategy can enable a better throughput by a factor of up to 2x while reducing processing latency.

Acknowledgment

The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [26] – see hpc.uni.lu. This work is partially funded by the SnT-LuxProvide partnership on bridging clouds and supercomputers and by the Fonds National de la Recherche Luxembourg (FNR) POLLUX program under the SERENITY Project (ref. C22/IS/17395419).

References

1. Next CERN Accelerator Logging Service Architecture., https://www.slideshare.net/SparkSummit/next-cern-accelerator-logging-service-with-jakub-wozniak
2. Sensitive information detection using the NVIDIA Morpheus AI framework (2021), https://developers.redhat.com/articles/2021/10/18/sensitive-information-detection-using-nvidia-morpheus-ai-framework
3. Apache Flink (2022), https://flink.apache.org/
4. Apache Kafka (2022), https://kafka.apache.org/
5. Apache Pulsar (2022), https://pulsar.apache.org/
6. Apache Spark (2022), https://spark.apache.org/
7. Large Hadron Holider. (2022), http://home.cern/topics/large-hadron-collider
8. Pravega (2022), http://pravega.io/
9. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., Whittle, S.: The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proc. VLDB Endow. **8**(12), 1792–1803 (Aug 2015). https://doi.org/10.14778/2824032.2824076
10. Antoniu, G., Valduriez, P., Hoppe, H.C., Krüger, J.: Towards Integrated Hardware/Software Ecosystems for the Edge-Cloud-HPC Continuum (Sep 2021). https://doi.org/10.5281/zenodo.5534464
11. Bhardwaj, A., Kulkarni, C., Stutsman, R.: Adaptive placement for in-memory storage functions. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). pp. 127–141. USENIX Association (Jul 2020)
12. Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., Tzoumas, K.: State management in apache flink®: Consistent stateful distributed stream processing. Proc. VLDB Endow. **10**(12), 1718–1729 (Aug 2017). https://doi.org/10.14778/3137765.3137777
13. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (Jan 2008). https://doi.org/10.1145/1327452.1327492

14. Fried, J., Ruan, Z., Ousterhout, A., Belay, A.: Caladan: Mitigating Interference at Microsecond Timescales. USENIX Association, USA (2020)
15. Javed, M.H., Lu, X., Panda, D.K.D.: Characterization of big data stream processing pipeline: A case study using flink and kafka. In: Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies. p. 1–10. BDCAT '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3148055.3148068
16. Jay, K., Neha, N., Jun, R.: Kafka: A distributed messaging system for log processing. In: Proceedings of 6th International Workshop on Networking Meets Databases. NetDB'11 (2011)
17. Kalavri, V., Liagouris, J., Hoffmann, M., Dimitrova, D., Forshaw, M., Roscoe, T.: Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. p. 783–798. OSDI'18, USENIX Association, USA (2018)
18. Marcu, O.C., Bouvry, P.: Colocating real-time storage and processing: An analysis of pull-based versus push-based streaming (2022)
19. Marcu, O.C., Costan, A., Antoniu, G., Pérez-Hernández, M., Nicolae, B., Tudoran, R., Bortoli, S.: Kera: Scalable data ingestion for stream processing. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). pp. 1480–1485 (2018). https://doi.org/10.1109/ICDCS.2018.00152
20. Marcu, O.C., Costan, A., Nicolae, B., Antonin, G.: Virtual log-structured storage for high-performance streaming. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER). pp. 135–145 (2021). https://doi.org/10.1109/Cluster48925.2021.00046
21. Miao, H., Park, H., Jeon, M., Pekhimenko, G., McKinley, K.S., Lin, F.X.: Streambox: Modern Stream Processing on a Multicore Machine. In: USENIX ATC. pp. 617–629. USENIX Association (2017)
22. Nguyen, S., Salcic, Z., Zhang, X., Bisht, A.: A low-cost two-tier fog computing testbed for streaming iot-based applications. IEEE Internet of Things Journal **8**(8), 6928–6939 (2021). https://doi.org/10.1109/JIOT.2020.3036352
23. Ousterhout, A., Fried, J., Behrens, J., Belay, A., Balakrishnan, H.: Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In: Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation. p. 361–377. NSDI'19, USENIX Association, USA (2019)
24. Qin, H., Li, Q., Speiser, J., Kraft, P., Ousterhout, J.: Arachne: Core-aware thread management. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). USENIX Association, Carlsbad, CA (2018)
25. Sijie, G., Robin, D., Leigh, S.: Distributedlog: A high performance replicated log service. In: IEEE 33rd International Conference on Data Engineering. ICDE'17
26. Varrette, S., Bouvry, P., Cartiaux, H., Georgatos, F.: Management of an academic hpc cluster: The ul experience. In: 2014 International Conference on High Performance Computing Simulation (HPCS). pp. 959–967 (2014). https://doi.org/10.1109/HPCSim.2014.6903792
27. Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M.J., Recht, B., Stoica, I.: Drizzle: Fast and Adaptable Stream Processing at Scale. In: 26th SOSP. pp. 374–389. ACM (2017). https://doi.org/10.1145/3132747.3132750
28. Zou, J., Iyengar, A., Jermaine, C.: Pangea: Monolithic distributed storage for data analytics. Proc. VLDB Endow. **12**(6), 681–694 (Feb 2019). https://doi.org/10.14778/3311880.3311885