Toward a Push-based Stream Programming Model with AIMSS: An Active In-Memory Storage System Approach

Ovidiu-Cristian Marcu*, Grégoire Danoy†*, Pascal Bouvry†*
*SnT, University of Luxembourg
†FSTM/DCS, University of Luxembourg
{ovidiu-cristian.marcu, gregoire.danoy, pascal.bouvry}@uni.lu

***Abstract*—This paper introduces the vision for an Active In-Memory Storage System (AIMSS), a novel architecture that shifts data movement responsibilities, such as source handling, sink management, and data shuffling, from applications like training large language models (LLMs) and big data streaming engines, directly to AIMSS. AIMSS will operate on a log-structured in-memory storage framework, leveraging immutable data access patterns, facilitating efficient real-time data movement. The AIMSS architecture deploys on CPU and GPU nodes, harnessing their memories and ensures efficient and transparent communication with disk-based file storage systems. We propose a push-based stream programming execution model that allows AIMSS to cost-effectively harness application-specific data (such as consumer offsets and data access patterns including read, write, and shuffle) and thereby enable a set of data-based optimizations. These include scalable data movement partitioning algorithms, faster stream storage recovery (speeding up application restarts), easy identification of application stragglers, and mitigation of power fluctuation issues during large-scale LLM training (e.g., by efficiently leveraging idle GPU resources for other computing tasks). Furthermore, AIMSS will minimize I/O interference in multi-CPU-GPU setups for multiple applications sharing a high-performance computing infrastructure, including CPUs, GPUs, and advanced interconnects. AIMSS promises significant performance improvements by actively handling data movement for data-intensive applications and by combining in-memory processing with a novel push-based stream programming model suitable for exascale computing.**

***Index Terms*—in-memory systems, streaming, ml/ai applications, hpc infrastructure, unified storage and compute, push-based streaming model**

I. Introduction: Motivation, Vision and Objectives

In today's data-driven world, exemplified by recent advancements in large language models (LLMs, e.g., OpenAI's GPT 4, Google Gemini) and new Cloud-HPC services (e.g., HPC federated learning [1]), the rapid growth of machine learning/artificial intelligence (ML/AI) and big data applications has generated an unprecedented demand for scalable, energy-efficient and fault-tolerant, data-intensive, and Active In-Memory Storage Systems (AIMSS) in support of ML/AI over large-scale HPC infrastructure including many-core CPU-GPU nodes, large memory clusters (TBs/node) and advanced interconnects (e.g., NVIDIA Infiniband).

Scaling AIMSS (as depicted in Figure 1) across the memory hierarchy of CPUs and GPUs [2] at HPC exascale (encompassing thousands of compute nodes with millions of CPU/GPU cores and hundreds of TBs of memory) presents a significant challenge. For example, GPUs can spend up to 70% of their time idle, waiting for data [3]. This inefficiency highlights the need for a more effective data management approach. Our primary focus is to enable the efficient execution of data-intensive workloads by introducing a novel programming model that allows applications to **delegate data movement** responsibilities to AIMSS. This delegation, achieved through a **push-based streaming execution programming model**, allows AIMSS to leverage application-specific data access (such as consumer offsets and read/write patterns) for optimizing data movement and execution across the HPC storage and computing infrastructure. As we argue in the next sections, this includes having application workloads completely delegate data movement tasks such as ingestion, output writing, and data shuffling.

Fault tolerance, a critical challenge at exascale, can lead to significant wasted compute capacity (20% or more) due to failures and recovery, as highlighted by the European strategic research agenda for HPC [4] [pages 79-82]. Addressing this challenge is a core focus of our research. Our second goal is to develop a fault-tolerant in-memory storage system for HPC exascale. AIMSS will achieve this through an immutable log-structured design and its novel push-based streaming programming model. Our envisioned approach enables valuable application insights into data access patterns, facilitating faster recovery, as detailed in the following sections.

Extreme power jitter [5], arising from the synchronization of tasks like checkpointing [6], collective communication [7], and training computations during large-scale LLM training, presents a significant challenge. As highlighted in [5] [The Llama 3 Herd of Models, section 3.3], synchronized power fluctuations across tens of thousands of GPUs can strain data center power grids, potentially reaching tens of megawatts. To address this, our third goal is to mitigate, and potentially eliminate, this power jitter through a unified AIMSS and computing engine enabled by our proposed push-based streaming execution model. By leveraging AIMSS's awareness of remaining computations, derived from hints within data streams, we can efficiently utilize idle GPU resources during synchronization tasks such as checkpointing, which often takes tens of seconds.

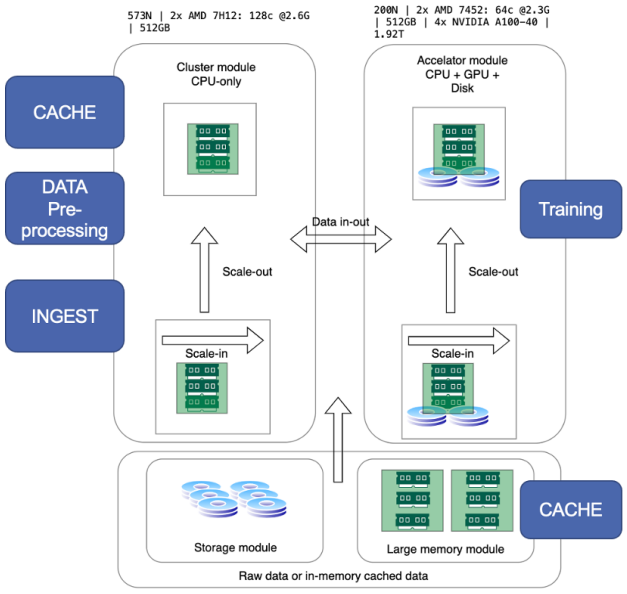

Fig. 1. The scalability challenge and hardware overview of main HPC MeluXina modules and how they fit various ML pipeline components.

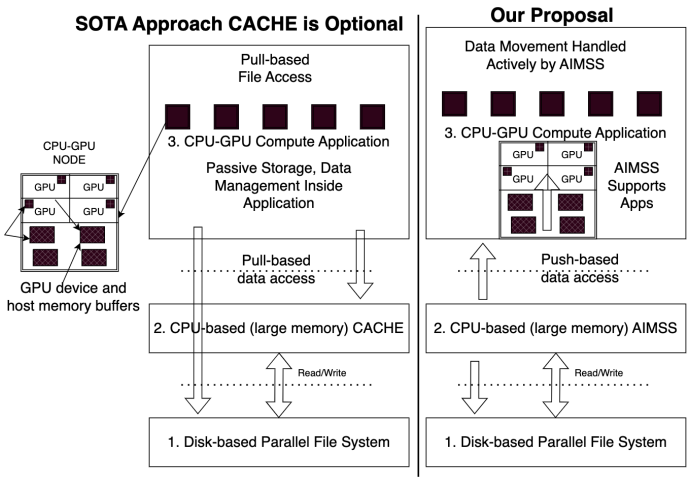

Fig. 2. The AIMSS active push-based storage approach versus passive pull-based state of the art (SOTA) storage approach. AIMSS manages resources depicted in blue in Figure 1 and integrates with LLM-engines for training through the push-based streaming execution model.

This active resource management, facilitated by AIMSS's in-memory storage capabilities, promises to reduce power jitter by dynamically scheduling other tasks on otherwise idle GPUs.

Therefore, the central research question driving our envisioned AIMSS is: **How can we efficiently (in terms of energy, performance, and developer transparency to exascale deployments) scale in and scale out large-scale ML/AI pipelines on HPC infrastructure in a fault-tolerant manner?** This paper is structured as follows: Section II introduces the architecture and design principles of AIMSS. Section III presents a novel push-based stream-based programming model for AIMSS. Section IV explores the architectural optimizations enabled by AIMSS. Section V discusses related work and highlights AIMSS's vision for optimization contributions.

II. The AIMSS Approach

Our key insight is that closely integrating (in-memory) storage and processing for ML by delegating data movement control from the application to the storage layer, as our AIMSS proposes, will lead to various optimizations (explained later). Our global vision for AIMSS is a unified storage and compute architecture for ML/AI processing on HPC infrastructure, powered by a **push-based streaming execution model** with the following key benefits:

- **Unified CPU-GPU Deployment and Optimized Performance:** AIMSS will be deployed across CPU-GPU HPC infrastructure, leveraging their combined memory resources to support a push-based stream-based programming model (as depicted in Figure 2). This unified approach will facilitate efficient data movement and processing at HPC exascale.
- **Transparent Scalability and Resiliency:** AIMSS will provide users with transparency and resiliency while automatically scaling ML pipelines on HPC infrastructure. This will empower users to focus on their core research and development tasks without being burdened by the complexities of manual scaling and fault tolerance storage management.

AIMSS manages various data movement operations in support of processing engines that typically deploy pipelines of operators, including source and sink operators. Source operators fetch input data from a distributed storage system (e.g., disk-based file or caching systems) using a pull-based approach. Sink operators, on the other hand, write data to the storage system using a push-based approach. In addition, shuffle operators are responsible for redistributing data based on partitioning methods, such as key-based partitioning. Traditionally, the storage system plays a passive role, responding to read/write requests, while the shuffle mechanism is implemented at the application level. Managing source, sink and shuffle data movement, AIMSS will have a global I/O overview advantage over current approaches.

Recognizing that data-intensive applications (e.g., real-time streaming, LLM training) require continuous data movement (input, output, and shuffling), we propose the AIMSS strategy to delegate these operations to the AIMSS itself. Our strategy works as follows: Source operators register their input stream requirements (including any filtering functions) with AIMSS, which then proactively fills input buffers using a push-based approach. Sink operators operate on pre-registered stream buffers and notify AIMSS when data is ready to be written, triggering asynchronous persistence to disk and buffer reuse. Shuffle operations function similarly, allowing AIMSS

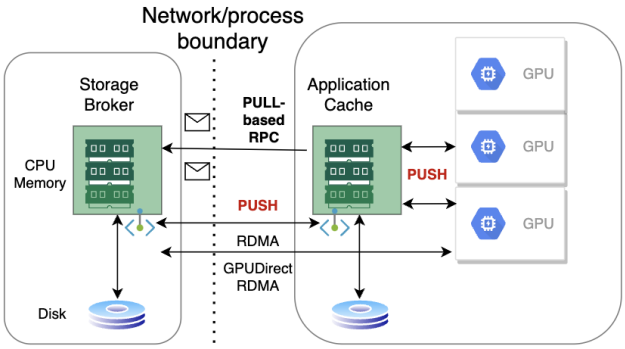

Fig. 3. The novel AIMSS architecture for scalable, unified storage and ML processing comprises coordinators (managing metadata and system components), CPU-based Storage Brokers (e.g., [8] interacting with file/cache systems to fill the CPU-GPU-based Application Cache components that are managing host and kernel memory for the application through push-based stream buffers (e.g., [9].

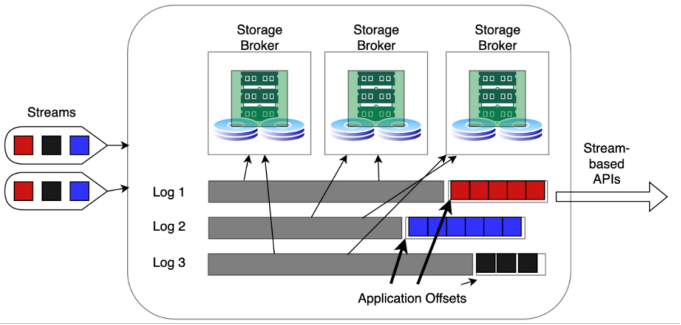

Fig. 4. Our novel approach to fast crash recovery for unified in-memory log-structured storage (e.g., [10]) and ML processing simplifies fault tolerance by enabling recovery from the last recorded application consumer/producer offsets already available to AIMSS.

to reorganize input stream buffers asynchronously.

Our vision for separating data movement operations from processing operators is realized through the AIMSS middleware layer that sits between the disk-based file storage system and application engines deployed on, for example, CPU-GPU nodes. Furthermore, AIMSS manages both CPU and GPU host memory and integrates with GPU device memory through native code (e.g., CUDA streams), using a push-based approach. Garbage memory collection is managed at the AIMSS level. All stream metadata is registered with AIMSS before and during deployment when Source, Sink, and Shuffle operations are delegated. When an application crashes or shuts down, AIMSS automatically cleans up its associated active streams.

Optimizing data movement for enhanced scalability, better performance, faster data and application recovery, and reduced power jitter at exascale is significantly more efficient when managed at the AIMSS data system level. This contrasts with today's approach, where this burden often falls on application developers and their engines, leading to sub-optimal performance and increased complexity.

III. Our Proposal: A Push-Based Streaming Programming Model for Enabling AIMSS

The core concept of our vision is integrating stream phases for both read and write I/O-intensive operations, including shuffling. This integration provides valuable insights into application data access behavior, enabling the stream storage layer (i.e, AIMSS) to optimize I/O performance, minimize I/O interference, and enhance both checkpointing efficiency and recovery speed. By replacing traditional passive storage approaches with the AIMSS framework, application developers can unlock significant performance gains and benefit from transparent data management workflows (e.g., avoid tuning efforts). Storage system techniques can more efficiently develop better data movement optimizations compared to letting this effort on developers.

Execution Model APIs. The following listing outlines the essential APIs for the AIMSS push-based streaming execution model. Compute engine's consumers and producers operators (e.g., GPU kernel tasks) create streams to interact with shared in-memory buffers managed by AIMSS through these APIs. Source, sink and shuffle operators delegate their read and write IO actions to AIMSS that is responsible to manage these push-based shared stream buffers.

```
1 CreateStream(ParentStreamId, InputSource,
      PartitionId, KernelReadWriteAttributes): return
      streamId # Initialize a new stream on AIMSS.
      Usage for read or write specified. Returns a
      unique streamId.
2 ReadFrom(StreamId) # Provides a set of shared
      StreamBuffers to iterate over
3 WriteTo(StreamId, StreamBuffer) # Write streamBuffer
      's content to a specified stream.
4 ShuffleStream(StreamId) # Signal shuffle ready.
5 ShuffleStreams(ParentStreamId) # Shuffle starts when
       all streamIds of ParentStreamId are ready
6 DestroyStream(StreamId)
```

Passive Storage Consumption Model APIs. The following listing offers a simplified overview of the typical file-based storage model, where the application compute engine, unlike with AIMSS, handles memory buffers and data movement.

```
1 CreateFile(FileName)
2 ReadFrom(FileName)
3 WriteTo(FileName, Data)
4 Shuffle(Files, PartitionFunction)
5 DeleteFile(Filename)
```

IV. Data-based Architectural Optimizations Enabled By AIMSS

The rationale behind a push-based streaming model, in addition to its low-latency processing advantages (see [9]), stems from the continuous and bursty [5] data processing requirements of use cases like LLM training, which often involve petabytes of input data and checkpointing data with tens of TB/s peak throughput. By proactively managing data

movement (input, output, and shuffling), a push-based stream storage system minimizes GPU idle time, reduces costs, and enables optimizations not easily achievable with a pull-based model, such as reduced I/O interference, faster recovery, straggler mitigation, and higher ingestion/checkpointing throughput. Moreover, the application knowledge insights provided by the push-based protocol's offsets eliminate the need for complex ML models and monitoring infrastructure to predict access patterns, simplifying the optimization process.

Traditionally, research engineers manually tune data partitioning, chunking, shuffling, checkpointing, and recovery during LLM training iterations. AIMSS, through the push-based model interactions, transparently manages these data movement operations on both consumer (input processing and recovery) and producer (shuffling, checkpointing) processes. Shifting data movement control from the application to the storage layer enables better optimization of ingestion and recovery due to its inherent application knowledge provided by stream access patterns available now to AIMSS.

Given the cost-effectiveness of CPU memory compared to expensive high-end GPUs, we advocate for aggressively optimizing data movement into and out of GPUs, leveraging AIMSS as a smart cache that provides needed storage features like availability and durability. Moreover, hardware trends (better interconnects [11], faster memory) will support our radical data movement approach. Beyond fast, scalable and dynamic data access, AIMSS provides two additional benefits: simplified fault tolerance implementation and the ability to detect stragglers easier (Application Cache nodes provide metadata of application compute tasks that may exhibit slower progress during training iterations).

A key technical implementation challenge involves optimizing the pipeline of computation with data feeding for GPUs. As exemplified in Figure 3, we plan to evaluate push-based RPC methods over RDMA technology [12]. Leveraging dynamic stream partitioning and push-based data movement for processing, AIMSS provides a foundation for simplifying application scalability. GPU kernels create and manage streams (see the previous section APIs) by interacting with the colocated Application Cache.

Critical questions include determining suitable data dynamic partitioning and program parallelism mechanisms for efficiently feeding multiple GPUs, and how to cache these datasets to ensure applications are not delayed. Another aspect is designing and developing a push-based approach for CPU to CPU-GPU nodes integration and examining its trade-offs in terms of availability, partitioning, performance, and fault tolerance.

Given that current state-of-the-art storage and processing systems handle recovery and fault tolerance independently, a significant challenge is enabling prioritization of data recovery without pipeline insights. The technical challenge involves providing the storage system with insights to prioritize the recovery of storage partitions essential to the application when a storage node crashes. This concept is illustrated in Figure 4. Assuming a log-structured storage [10] design with application consumer offsets corresponding to the next records to be processed in the stream, a rapid crash recovery mechanism should prioritize recovering logs starting at these application offsets. This is in contrast to current systems that begin expensive log recovery from the first log record without application-specific knowledge. The AIMSS stream-based recovery strategy allows access to data immediately.

V. Related Work

The following functional components are necessary to be modified for enabling the unified AIMSS architecture:

- Data Ingestion [13] acquires, buffers, and temporarily stores in-memory fast data streams and raw file data, achieving potentially low latency and high throughput (as implemented by KerA [14]).
- Data (Persistent and Caching) Storage [15] ensures durability, availability, and fault tolerance through multiple data copies stored over multiple nodes.
- Big data processing [16] and ML analytics [17], [18] enable ML applications to efficiently consume data streams.

In practice, each component is typically implemented as a dedicated solution independent of other layers, under the assumption that specialization enables better optimization opportunities. In contrast to monolithic architectures, e.g., [19], that can optimize data-related tasks more efficiently, these decoupled layered architectures do not easily benefit from data-related optimizations. While each specialized component benefits from an open-source community, coupling these components in complex architectures often results in a trade-off between productivity and performance/cost efficiency.

While scalable, stream storage systems such as Apache Kafka [20], [21] often require time-consuming and costly manual data re-partitioning to achieve high throughput. Kafka targets full stream log recovery, lacking support for partial recovery. Our in-memory storage system, KerA [8]–[10], [14], introduces dynamic partitioning to eliminate the need for manual data re-partitioning. AIMSS can build upon KerA, colocated with ML processing nodes, minimizing data migration during storage auto-scaling. While fault-tolerant storage systems typically fully recover crashed nodes [22], they often do so without considering application-specific needs [23] or knowledge. This full recovery becomes inefficient as compute node memory/storage increases; recovering terabytes of data can waste minutes on large CPU/GPU clusters, forcing applications to stop and restart from checkpoints inefficiently. AIMSS addresses this by leveraging log-structured in-memory storage [24] and push-based data movement, potentially over RDMA technology [12].

As we previously argued in [9] for a push-based streaming model across the computing continuum, more recent research on data flow in modern hardware [25] also supports the concept of stream processing [26] across the entire architecture. However, while their focus is primarily on reducing data movement, and thus orthogonal to ours, AIMSS takes a distinct data movement approach to seamlessly integrate with and enhance existing processing engines. By adopting a

holistic approach to AIMSS, and employing TLA+ [27], [28] (a formal verification language for concurrent and distributed systems) for addressing consistency issues (e.g., [29], we plan to research and develop a unified model design specification.

VI. Conclusion

In conclusion, the envisioned AIMSS push-based streaming execution model approach offers a compelling paradigm shift in managing data for large-scale, data-intensive applications. The key intuitions supporting this approach include leveraging: data immutability, the storage knowledge of application access patterns, the ability to more efficiently mitigate stragglers, and the potential to minimize costly GPU wait times. By adopting a system-level approach to data movement, AIMSS alleviates the burden on application developers for data movement tuning and enables optimizations not easily achievable with traditional in-application-based methods. The design, implementation and evaluation of AIMSS coupled with LLM-engines through streaming integration will be the focus of our future work.

VII. Acknowledgment

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] J. Hoffmann, P. Bauer, I. Sandu, N. Wedi, T. Geenen, and D. Thiemert, "Destination earth – a digital twin in support of climate services," *Climate Services*, vol. 30, p. 100394, 2023.

[2] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," vol. 47, no. 4, jul 2015. [Online]. Available: https://doi.org/10.1145/2788396

[3] S. Ben David, "Why a data plane architecture is critical for optimizing next-generation workloads," in *Proceedings of the 2022 Workshop on Emerging Open Storage Systems and Solutions for Data Intensive Computing*, ser. EMOSS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 9.

[4] T. E. T. P. for High Performance Computing (ETP4HPC), "Strategic research agenda for hpc in europe," 2022. [Online]. Available: https://www.etp4hpc.eu/pujades/files/ETP4HPC-SRA5_2022_web.pdf

[5] A. D. et al., "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[6] A. Maurya, R. Underwood, M. M. Rafique, F. Cappello, and B. Nicolae, "DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models," in *HPDC'24: 33nd International Symposium on High-Performance Parallel and Distributed Computing*, Pisa (IT), Italy, Jun. 2024. [Online]. Available: https://hal.science/hal-04614247

[7] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen, "Aluminum: An asynchronous, gpu-aware communication library optimized for large-scale training of deep neural networks on hpc systems," in *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, 2018, pp. 1–13.

[8] O.-C. Marcu, A. Costan, G. Antoniu, M. Pérez-Hernández, B. Nicolae, R. Tudoran, and S. Bortoli, "Kera: Scalable data ingestion for stream processing," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1480–1485.

[9] O.-C. Marcu and P. Bouvry, "In support of push-based streaming for the computing continuum," in *15th Asian Conference on Intelligent Information and Database Systems*, Phuket, Thailand, Jul. 2023.

[10] O.-C. Marcu, A. Costan, B. Nicolae, and G. Antoniu, "Virtual Log-Structured Storage for High-Performance Streaming," in *Cluster 2021 - IEEE International Conference on Cluster Computing*, Portland / Virtual, United States, Sep. 2021, pp. 1–11.

[11] M. Vuppalapati, S. Agarwal, H. Schuh, B. Kasikci, A. Krishnamurthy, and R. Agarwal, "Understanding the host network," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 581–594. [Online]. Available: https://doi.org/10.1145/3651890.3672271

[12] S. Ma, T. Ma, K. Chen, and Y. Wu, "A survey of storage systems in the rdma era," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, p. 4395–4409, dec 2022.

[13] O.-C. Marcu, A. Costan, G. Antoniu, M. S. Pérez-Hernández, R. Tudoran, S. Bortoli, and B. Nicolae, "Storage and Ingestion Systems in Support of Stream Processing: A Survey," INRIA Rennes - Bretagne Atlantique and University of Rennes 1, France, Technical Report RT-0501, Nov. 2018.

[14] O.-C. Marcu, "KerA: A Unified Ingestion and Storage System for Scalable Big Data Processing," Theses, INSA Rennes, Dec. 2018. [Online]. Available: https://theses.hal.science/tel-01972280

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[16] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, jan 2008.

[17] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 423–438. [Online]. Available: https://doi.org/10.1145/2517349.2522737

[18] M. e. a. Abadi, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[19] J. Zou, A. Iyengar, and C. Jermaine, "Pangea: Monolithic distributed storage for data analytics," *Proc. VLDB Endow.*, vol. 12, no. 6, p. 681–694, feb 2019. [Online]. Available: https://doi.org/10.14778/3311880.3311885

[20] G. e. a. Wang, "Consistency and completeness: Rethinking distributed stream processing in apache kafka," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2602–2613.

[21] A. e. a. Povzner, "Kora: A cloud-native event streaming platform for kafka," *Proc. VLDB Endow.*, vol. 16, no. 12, p. 3822–3834, aug 2023. [Online]. Available: https://doi.org/10.14778/3611540.3611567

[22] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 29–41. [Online]. Available: https://doi.org/10.1145/2043556.2043560

[23] A. Magalhaes, J. M. Monteiro, and A. Brayner, "Main memory database recovery: A survey," vol. 54, no. 2, mar 2021.

[24] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for dram-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14. USA: USENIX Association, 2014, p. 1–16.

[25] A. Lerner and G. Alonso, "Data flow architectures for data processing on modern hardware," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, May 2024, presented at the Data Engineering Future Technologies Track. [Online]. Available: https://exascale.info/assets/pdf/lerner2024icde.pdf

[26] O.-C. MARCU and P. BOUVRY, "Big data stream processing," University of Luxembourg, Tech. Rep., 30 August 2024. [Online]. Available: https://orbilu.uni.lu/handle/10993/61905

[27] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, p. 872–923, may 1994. [Online]. Available: https://doi.org/10.1145/177492.177726

[28] B. Batson and L. Lamport, "High-level specifications: Lessons from industry," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 242–261.

[29] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463–492, jul 1990. [Online]. Available: https://doi.org/10.1145/78969.78972