Disaggregated State Management in Apache Flink® 2.0

Yuan Mei [†] Zhaoqian Lan[†] Lei Huang[‡] Yanfei Lei[†] Han Yin[†]
Rui Xia [†] Kaitian Hu[†] Paris Carbone[§] Vasiliki Kalavri[‡] Feng Wang[†]
Alibaba Group[†] Boston University[‡] KTH Royal Institute of Technology[§]
{meiyuan.my, zhaoqian.lzq, leiyanfei.lyf, yinhan.yh, fufeng.xr, hukaitian.hkt, feng}@alibaba-inc.com[†]
{lei, vkalavri}@bu.edu[‡], parisc@kth.se[§]

ABSTRACT

We present Apache Flink 2.0, an evolution of the popular stream processing system's architecture that decouples computation from state management. Flink 2.0 relies on a remote distributed file system (DFS) for primary state storage and uses local disks as a secondary cache, with state updates streamed continuously and directly to the DFS. To address the latency implications of remote storage, Flink 2.0 incorporates an asynchronous runtime execution model. Furthermore, Flink 2.0 introduces ForSt, a novel state store featuring a unified file system that enables faster and lightweight checkpointing, recovery, and reconfiguration with minimal intrusion to the existing Flink runtime architecture. Using a comprehensive set of Nexmark benchmarks and a large-scale stateful production workload, we evaluate Flink 2.0's large-state processing, checkpointing, and recovery mechanisms. Our results show significant performance improvements and reduced resource utilization compared to the baseline Flink 1.20 implementation. Specifically, we observe up to 94% reduction in checkpoint duration, up to 49× faster recovery after failures or a rescaling operation, and up to 50% cost savings.

PVLDB Reference Format:
Yuan Mei, Zhaoqian Lan, Lei Huang, Yanfei Lei, Han Yin, Rui Xia, Kaitian Hu, Paris Carbone, Vasiliki Kalavri, and Feng Wang. Disaggregated State Management in Apache Flink 2.0. PVLDB, 18(12): 4846 - 4859, 2025.
doi:10.14778/3750601.3750609

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/apache/flink/tree/release-2.0.

1 INTRODUCTION

Alibaba leverages Apache Flink [19, 40] across all core business operations, demonstrating its versatility in handling high-volume, large-scale and real-time data processing. From powering dynamic e-commerce features, like personalized recommendations and real-time dashboards during peak sales events, to enabling critical financial risk control through fraud detection and credit scoring, Flink's ability to process data instantaneously is essential. It also optimizes logistics with real-time route adjustments and warehouse management, enhances advertising through dynamic ad placements and performance tracking, and ensures system stability with real-time logging and anomaly detection. Through extensive application and active contributions to the open-source community, Alibaba has solidified Flink's position as a leading stream processing engine [43]. Alibaba's Flink infrastructure supported an inbound data flow of over 4.4 billion TPS during the 2024 Double 11 shopping event.

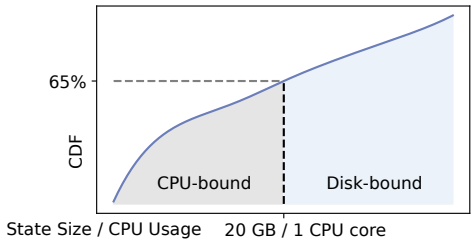

Figure 1: Distribution of CPU-bound vs. Disk-bound workloads in Alibaba's logistics business.

Apache Flink's success extends beyond Alibaba, with deployments across hundreds of companies [35], multiple cloud service offerings [7, 39, 44, 52], and over 1800 contributors and 100 committers contributing to its codebase [37, 43]. Flink was one of the first systems to provide features that are considered standard in modern stream processing engines [42, 56, 63]: support for larger-than-memory state, exactly-once state guarantees, and embedded state management with remote checkpointing.

Over the years, Flink's design has evolved alongside its use cases. Originally built for continuous analytics, it has gradually been adopted as a platform for distributed stateful applications with exactly-once processing and consistent out-of-order execution guarantees [17]. In the past decade, we have witnessed a dramatic shift in Flink's deployment mode and workload patterns, partly driven by hardware improvements and advances in cloud computing. We have transitioned from a Map-Reduce era, where computation is performed on distributed clusters of machines responsible for both computation and storage, to cloud-native settings, where containerized deployments on Kubernetes have become commonplace. Meanwhile, an increase in network bandwidth and the availability of cheap object storage unlock new disaggregated design options.

These recent trends reveal shortcomings in Flink's current architecture, mainly due to its tight coupling of compute and state. While embedded state management provides a robust and high-throughput solution for applications with larger-than-memory state, it hinders the design of low-downtime auto-scaling and reconfiguration mechanisms, due to the requirement for checkpointed state migration. At the same time, it falls short of efficiently handling multi-terabyte states encountered in today's applications. As

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750609

an example, Figure 1 shows the cumulative distribution of state size alongside the corresponding CPU utilization for several hundreds of jobs in Alibaba's logistics business. We adopt a threshold of 20 GB state size per CPU core - equivalent to a single Compute Unit (CU) in Alibaba Cloud Service [30] - to distinguish between CPU-bound and Disk-bound jobs. We observe that 35% of jobs are disk-bound, requiring additional CUs only to address storage limitations.

In this paper, we present Flink 2.0, a disaggregated state architecture designed to efficiently support modern cloud infrastructure and emerging application requirements. Flink 2.0 designates the remote distributed file system (DFS) as the primary state storage and uses local disks as an optional, secondary cache. State updates are streamed to the DFS continuously, enabling faster checkpointing, recovery, and reconfiguration, as files are immediately accessible in the remote file system. Flink 2.0 avoids expensive state migration by sharing active state and checkpointed state in the same DFS.

Decoupling computation from state management in Flink poses several technical challenges. Maintaining state in a remote DFS unavoidably increases access latency and requires careful design to avoid performance overhead. Naively combining a remote state backend with Flink's synchronous execution model further leads to low throughput and resource underutilization. At the same time, redesigning the fault tolerance and rescaling mechanisms is essential to effectively leverage the new file storage layout, while maintaining compatibility. Finally, it is vital to ensure that the new design preserves Flink's out-of-order execution semantics and fault-tolerance guarantees and that the introduction of new features is seamless, with intuitive and non-intrusive APIs.

To address the above challenges, Flink 2.0 introduces two major innovations: (i) an asynchronous record execution framework that enables non-blocking state access and leverages out-of-order execution to improve performance, and (ii) a new disaggregated state store, ForSt (short for *For Streaming)*, that provides an LSM-tree abstraction to seamlessly unify local and remote state access. This paper makes the following contributions:

- We identify a set of limitations in Flink 1.x's embedded state management architecture that hinder support for very large states and efficient exploitation of modern cloud infrastructure. We then present Flink 2.0's disaggregated state model and explain how it addresses the shortcomings by discussing a real-world use case (§ 3).
- We introduce Flink 2.0's asynchronous execution model that enables out-of-order record processing and non-blocking state access, to achieve high bandwidth utilization and hide the remote access latency. We discuss how the asynchronous model preserves Flink 1.x's per-key processing order, watermark correctness, and fault tolerance guarantees (§ 4).
- We introduce ForSt (*For Streaming*), Flink 2.0's disaggregated state store, and a new unified filesystem layer that facilitates fast and lightweight checkpointing, instant recovery, and seamless reconfiguration (§ 5).
- We present a comprehensive experimental evaluation of Flink 2.0's features and a comparison with Flink 1.20, on two types of benchmarks: (i) a large-scale stateful production workload to demonstrate cost savings and performance of checkpointing, recovery, and rescaling (§ 6.1), and (ii) the widely-used Nexmark benchmark [60] (§. 6.2).

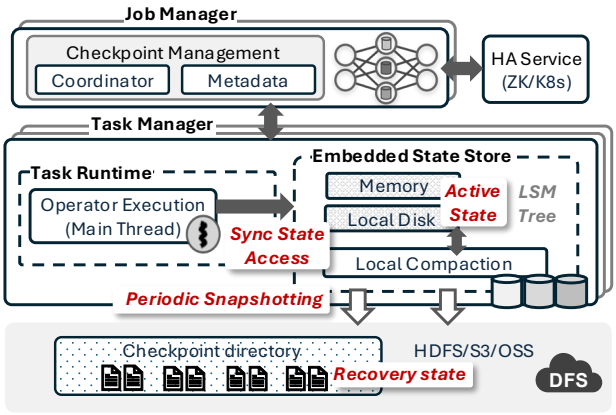

Figure 2: Overview of Flink 1.x architecture.

Deployed for over two years in Alibaba Cloud Realtime Computation Service, Flink's disaggregated architecture has delivered highly efficient services for large-state jobs, enabling fast cluster-level scaling and recovery, particularly during major sales events. We are pleased to contribute this proven architecture to the community.

2 PRELIMINARIES

Apache Flink adopts a distributed dataflow execution model, where stream computations are modeled as directed acyclic graphs (DAGs) of tasks. A Flink job consists of stateful operators, such as window aggregations and joins that process records and update local state continuously. These operators execute in parallel across a cluster of machines. At a high level, Flink represents computation as a logical dataflow graph, describing the sequence of transformations applied to records. This logical plan is then optimized and transformed into a physical graph, where operators map to stream task instances that run in parallel across compute nodes. Next, we provide a brief overview of Flink's distributed architecture and execution model [17, 19] while highlighting its core guarantees: 1) per-key sequential processing order, 2) exactly-once processing, and 3) low-watermark event-time order.

2.1 Distributed Dataflow Architecture

System overview. Figure 2 shows an overview of Flink 1.x. The system deployment entails two components: the Job Manager (JM) and a set of Task Managers (TMs) running on Kubernetes, YARN, or other cloud-based platforms. The JM acts as the main entry node to the system, compiling applications to dataflow graphs, scheduling tasks, coordinating job checkpoints, and instrumenting the lifecycle of applications. For high availability (HA), the JM node maintains its metadata in Zookeeper or etcd-Kubernetes for failover. Metadata includes checkpoint references to durable files in an external distributed file system (DFS), such as HDFS/S3/OSS. The TMs are responsible for the distributed execution of stateful dataflow programs. Each TM executes its part of the distributed dataflow computation on a set of dedicated stream tasks.

Synchronous task execution. Stream tasks are dedicated threads running one physical dataflow task each. They consume multiple

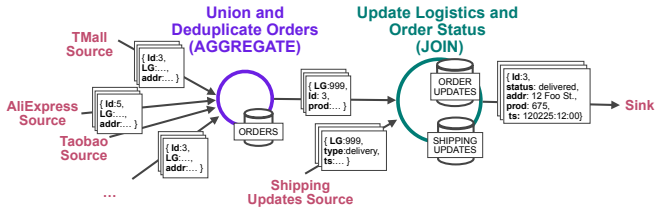

Figure 3: Logical dataflow graph for a real-time logistics management use case. The Flink job tracks the lifecycle of order packages as they move through the supply chain.

input streams that contain records and control events, such as *low-watermarks* and *checkpoint markers*. The network layer derives a single input stream per task, which maintains the partial FIFO orders of the inputs. Each stream task in Flink 1.x also follows a synchronous FIFO execution: pulls the next input element, executes task logic, and updates state and output as a single atomic operation.

Embedded state management. In Flink, state is pre-allocated in non-overlapping virtual partitions (key-groups). The partitioning mechanism pre-arranges state and input partitions using the `keyBy` operator. This ensures that all records in the same key partition are processed by the same task. TMs in Flink 1.x are configured with an embedded *state backend* module that manages all stateful read and write operations on their allocated partitions using single-writer semantics. In Flink 1.x, active state is exclusively handled by the local state backend, such as the out-of-core RocksDB-based backend or the in-memory Java-heap alternative. Whereas, recovery state from all local checkpoints is managed externally on DFS with checkpoint references managed by the JM. Embedded state backends are responsible for obtaining and transferring all active state changes to DFS during checkpointing for recovery.

2.2 Core System Guarantees

Flink's default stream task dataflow execution provides three basic guarantees, well-understood by its users.

Per-key FIFO order. While Flink does not guarantee total ordering across different input streams, it ensures that records with the same key are processed in FIFO order. This means that the order of records identified by a specific key is preserved during processing. This stems from Flink's FIFO channels in combination with the strict sequential execution of its stream tasks. Furthermore, its state partitioning mechanism guarantees that records belonging to the same key are always handled by the same stream task, ensuring exclusive access and enforcing single-writer semantics.

Exactly-once processing. Flink employs an asynchronous 2-phase-commit (2PC) mechanism to commit all state changes that occur between two consecutive checkpoint markers [17]. During the first phase, the checkpoint coordinator (JM) inserts checkpoint markers into all input streams. Flink's network layer aligns markers leading to a single aligned checkpoint marker at the task's input. Upon its consumption, an asynchronous checkpoint is initiated at the backend module of the Task Manager. This process involves taking a durable physical copy to the external DFS while sending an acknowledgment to the coordinator. Upon gathering all acknowledgments, the coordinator enters the "commit" phase of the protocol, notifying all stream tasks that the checkpoint is complete. This allows special tasks, such as transactional sinks, to commit pending changes from a write-ahead-log to their output files or consumers.

Low-watermark event-time order. Watermarks are used as a consistent mechanism to establish progress when stream records arrive out-of-order from their respective substreams [3, 6]. Stream tasks use this mechanism to advance their current event-time "clock" and release timed operations. Then, tasks emit the same watermark in their output, letting downstream operators also advance their time. Flink's network layer automates part of the watermark computation through a monotonic `max-min` reduction: given the sets of watermarks per input stream $W_1, \ldots W_k$, the derived watermark is computed as such $w' = \min_{1 \le i \le k} \max_{W_i}$. In addition, synchronous task execution in Flink 1.x guarantees completeness i.e., all computations timed before a watermark w are guaranteed to be complete at the reception of w within each task.

3 DISAGGREGATED STATE MANAGEMENT

In this section, we present a real-world use case to illustrate the limitations of Flink's embedded state design and motivate the need for decoupling compute from storage. Then, we provide an overview of the disaggregated state management architecture in Flink 2.0 and discuss correctness and performance challenges.

3.1 Real-World Motivating Use Case

Figure 3 illustrates a representative use case of real-time logistics management within Alibaba's stream processing ecosystem. The use case is implemented by a Flink job that tracks the lifecycle of order packages as they move through the supply chain and consists of two stateful transformation stages. The first stage is a deduplication operator that ingests *Order* events originating from multiple e-commerce platforms (e.g., TMall, Taobao, AliExpress). Upon order placement, a new logistic order event is generated, containing initial data like order ID, address, and product information. At this stage, fields such as delivery time and status are left empty. Deduplication is performed by an aggregation operator that retains the most recent order entry as state. The second stage is a streaming join operator that matches *Order Update* events with *Shipping Update* events to produce an enriched, up-to-date order record. A *Shipping Update* event is generated every time a package goes through transit nodes and contains progress information, such as delivery, consignment, or customs clearance. The join maintains two states internally: (i) aggregated order updates (left stream) and (ii) real-time shipping updates (right stream). Although most packages are delivered and updated within a week, exceptions due to customs, weather, or unforeseen delays require longer update cycles. Consequently, the deduplication and join operators need to maintain 60 days' worth of data, resulting in terabytes of job states. Naturally, the state size is proportional to the cardinality of orders and events. During holidays or Double 11 shopping events, the number of orders exhibits bursts, and so do the shipping update events. The state size ranges from a few hundred gigabytes to terabytes.

3.2 Limitations of Embedded State Management

After nearly a decade of operating large-scale Flink deployments, we have identified several limitations of the embedded state management architecture (cf.§ 2.1). These challenges are particularly

prominent in jobs with large state, like the logistics use case above, and containerized deployments, as we explain below.

State size constraints imposed by local disk storage. Flink 1.x workers use embedded RocksDB instances to support streaming state beyond the limits of main memory. However, for jobs that demand significantly larger storage capacities, like the logistics use case, the local disk can quickly become the bottleneck. Even so, dynamically adjusting the capacity of local disks may be difficult, especially in containerized deployments. Cloud service providers, like Alibaba Cloud and AWS, commonly provide compute units with *predefined* and *static* resource allocations, with a fixed number of CPU cores and disk capacities. For example, Alibaba Cloud Realtime Computation Service offers compute units with 1 CPU core and 20 GB of disk space [30], while AWS Kinesis Processing Units offer 1 CPU core along with a 50 GB disk [44]. This inherent inflexibility limits the scalability and efficiency of applications with fluctuating resource needs. To accommodate such workloads, it is essential to extend the storage hierarchy to integrate remote elastic storage.

Resource interference from periodic state backend operations. Another problem with managing large states in embedded RocksDB instances is that background backend operations can interfere with the primary processing tasks. Specifically, we found that periodic operations, like compaction and checkpointing, can cause spikes in CPU utilization and disk and network I/O consumption. As we will show in Section 6 (cf. Figure 11), the problem persists in Flink 1.20, even when backend operations are executed asynchronously. The resource consumption spikes can adversely impact query performance, unless additional resources are reserved in advance to accommodate them.

Prolonged checkpointing and reconfiguration durations. To ensure fault tolerance in the logistics use case, Flink's checkpointing mechanism periodically transfers job state to durable storage. In Flink 1.x, checkpointing involves two sequential phases: (i) a synchronous phase, during which local state tables are locked and copied to temporary local storage, and (ii) an asynchronous phase, wherein the local copies are transferred to a distributed file system. The same process is followed for on-demand checkpoints, which are necessary to ensure consistent reconfiguration, when resource demands change. During recovery or rescaling, the stored state must be retrieved from remote storage and reloaded into the local state backends. As state size increases, both the resource consumption and the duration of these operations scale proportionally, impeding Flink's ability to adapt swiftly to workload variations and provide efficient end-to-end exactly-once processing guarantees. As we show in Section 6.1, an average incremental checkpoint of 1.89 GB for the logistics use case can take up to one minute to complete. Rescaling with a 290 GB total state size exceeds five minutes.

3.3 Overview of Flink 2.0

To address the above shortcomings, Flink 2.0 adopts a new disaggregated state architecture, shown in Figure 4, where the newly introduced components are highlighted in red. The core design principle is to decouple resource-intensive state operations from compute, both at the runtime execution layer and at the state management layer. Flink 2.0 relies on a Distributed File System (DFS)

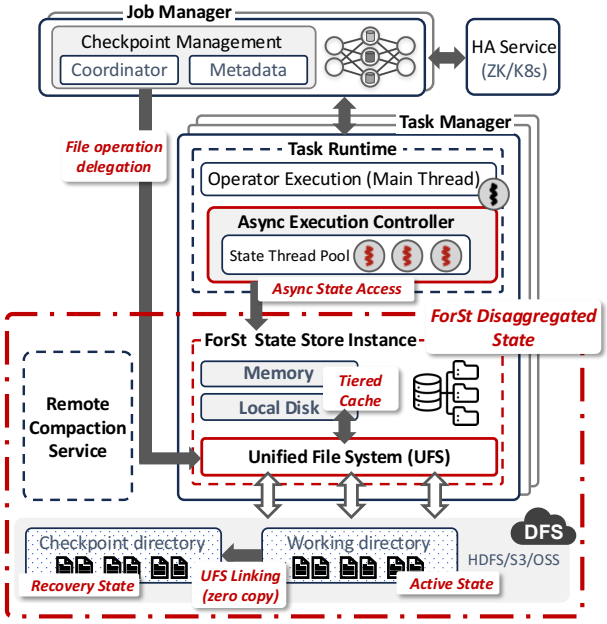

Figure 4: Disaggregated state management in Flink 2.0.

as its primary state storage, allowing faster reconfiguration, minimizing query performance disruption, and scaling to very large states that could exceed the storage capacity of a single node. In addition, we define three key objectives in Flink 2.0: (1) preserving the core processing semantics of Flink 1.x, (2) ensuring compatibility with its established execution framework while enabling new functionalities, and (3) facilitating a seamless migration experience with minimal modifications.

Asynchronous execution. At the runtime layer, we introduce the *Asynchronous Execution Controller* (AEC) to enable asynchronous execution. AEC decouples record processing (applying user-defined transformations to input events) from state access, mitigating latency associated with DFS access for state storage. Additionally, the AEC ensures correct scheduling of the execution to maintain core system guarantees (cf § 2.2). Flink 2.0's execution model is fully compatible with Flink 1.x. The AEC can be entirely bypassed if asynchronous execution is disabled or state access is not needed. We describe the asynchronous execution model in detail, in Section 4.

Disaggregated state backend. At the state management layer, we introduce ForSt, a disaggregated state store using a DFS as its primary state storage. In Figure 4, active working state is stored in the *Working Directory* on DFS, and is directly accessible by the ForSt instance to eliminate local disk constraints. Notably, ForSt instances operate within the Task Managers as before, ensuring compatibility with Flink 1.x's deployment paradigm. Further, we incorporate a Unified File System (UFS) abstraction in ForSt to harmonize the behavior and capabilities of diverse distributed file systems. The UFS provides a logical file view to Flink engines and delegates file management operations. A key benefit of this abstraction is the facilitated physical file sharing between active state files and recovery (checkpoint) files on DFS.

By leveraging the UFS, we provide a fast and lightweight checkpoint mechanism in Flink 2.0. While in Flink 1.x, the checkpointed

files on the DFS and the state files on the local disk are managed separately by the Job Manager and the state backend, respectively. Flink 2.0 *streams states continuously* to the DFS. To allow direct utilization of these physical state files for checkpoints without management conflicts, the UFS provides file operation delegation, maintaining logical links between state and checkpoint files, as shown in Figure 4. This results in a lightweight metadata update when checkpointing, while maintaining the existing state and checkpoint file lifecycle management. As we show in Section 6.1, checkpoints in Flink 2.0 consistently complete within a few seconds. Recovery and rescaling benefit as well, as loading large state from remote storage can be avoided by leveraging the UFS. Finally, the UFS enables the option of *remote compaction*, which reduces interference with normal processing. Remote compaction is currently an experimental feature in Flink [36]. We present Flink 2.0's new state management layer, ForSt, in Section 5.

4 ASYNCHRONOUS EXECUTION MODEL

A transition from local to remote state comes with an inevitable impact on read latencies. Table 1 shows the access latency across different storage mediums. Reading from remote storage, like HDFS or object storage (OSS, S3), is two orders of magnitude slower than reading from a local disk. As discussed in Section 2.2, stream tasks in Flink 1.x follow a strictly sequential per-record execution. That includes CPU-bound transformations and blocking state reads as one atomic operation executed by the main thread (writes are logged locally asynchronously). Evidently, employing this sequential execution model with remote state access would severely limit performance and lead to I/O bandwidth underutilization. To address the limitations of blocking I/O, Flink 2.0 introduces an *asynchronous execution model*, involving key runtime updates. Specifically, by leveraging task-level parallelism and allowing records to be processed asynchronously and out-of-order, the system can overlap CPU-bound computations with slow remote I/O operations, amortizing latency penalties while improving throughput.

4.1 Asynchronous Execution Overview

Asynchronous execution in Flink 2.0 decouples state access (I/O) from data processing (CPU), moving blocking communication out of the critical path of the main task thread. The processing lifecycle of each input record is refined into three separate processing stages centered around state access: (1) *non-state transformations*, (2) *state access*, and (3) *post-state access callbacks*. Non-state transformations and post-state callbacks are executed in the main thread as before. However, the state access stage is now handled asynchronously by a separate thread pool and can involve a chain of multiple state operations per input record. This is implemented by linking individual state access calls through callbacks, forming an asynchronous pipeline. If an input record does not require state access, only the non-state transformation is executed, similar to Flink 1.x. Finally, post-state callbacks take precedence over new records' non-state transformations to guarantee the timely completion of pending records before progressing further.

Shifting to asynchronous execution poses several challenges, particularly in maintaining Flink's core system guarantees under the new runtime. In the rest of this section, we first detail changes to Flink's programming model (§ 4.2), using the streaming join operator of the logistics use case (cf. § 3.1). We then provide a rigorous overview of additional mechanisms implemented to preserve Flink's core guarantees. Namely, we discuss the *Async Execution Controller (AEC)* that schedules async processing to preserve per-key FIFO order (§ 4.3), the *Async Draining Mechanism* that preserves exactly-once processing with checkpoints (§ 4.4), and, the *Epoch Manager*, that enforces low-watermark event-time order (§ 4.5).

Table 1: Access Latency for Different Storage Mediums

Local Read Latency		Remote Read Latency	
NVMe	ESSD PL1	HDFS on NVMe	OSS
68 µs	199 µs	1.5 ms	23 ms

Algorithm 1: Async join of order and shipping updates.

```
1 Record cast to Order type;
2 State_order.asyncUpdate(Record);
3 State_ship.asyncGetEntries()
4     .THEN (ship_events -> {
5         Joined_Records = applyJoin(Record, ship_events)
            output(Joined_Records); })
```

4.2 Asynchronous Programming Model

We showcase the async programming model using the streaming join operator of Figure 3. The join processes either *Order* updates or *Shipping* updates, which are joined against two state tables: $State_{ship}$ and $State_{order}$, respectively. To enable async state access, operators need to provide async *hints* to the runtime executor.

Algorithm 1 illustrates how an *Order* update is joined with the $State_{ship}$ table within the asynchronous processing model. Non-state transformations (Line 1) remain unchanged from Flink 1.x. We first update the $State_{order}$ with an $asyncUpdate$ (Line 2). For state accesses with additional actions, we use the $THEN$ method (Line 4) to define the callback execution (Line 5). The callback function is triggered when the async state request is finished. In the context of the streaming join, accesses to $State_{order}$ and $State_{ship}$ are independent and can be initiated concurrently. However, when dependencies exist between state access operations, the $THEN$ method ensures sequential chaining and preserves the correct order across state accesses. We refer the reader to the Apache Flink wiki [12] for a comprehensive list of the Flink 2.0 asynchronous APIs.

4.3 Preserving Per-Key Processing Order

Flink's existing per-key sequential processing order guarantee (§ 2.2) is fundamental for correctness in offering single-writer atomic access to states, as well as for its capabilities, allowing frameworks such as complex event processing to build on top. To preserve this property under thread-parallel state access conditions, Flink 2.0 introduces a new component internal to stream tasks, the *Asynchronous Execution Controller* (*AEC*). The *AEC* enforces a single active in-flight computation per key, within each task, at a time.

Stream input arriving at each stream task maintains per-key FIFO order guarantees inherently, through Flink's existing channel management mechanism. This ensures that records belonging to

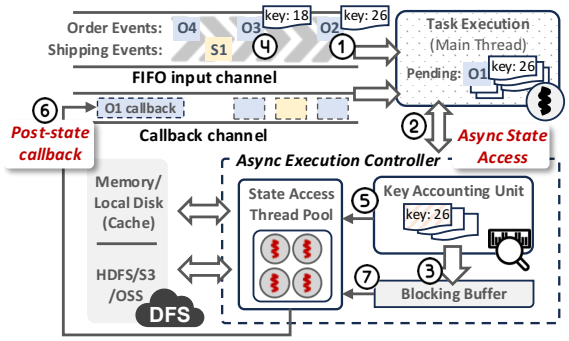

Figure 5: Asynchronous execution controller to ensure per-key processing order.

the same key will appear in their order of origin at the derived task sequential input. The *AEC* extends this guarantee during processing, by ensuring that intra-task scheduling maintains the same per-key partial processing order while allowing concurrency across independent keys. To achieve this, the *AEC* tracks the keys of all in-progress computations within its *Key Accounting Unit* and permits the scheduling of new input records only when there are no pending executions on the same respective key. In-progress computations include non-state executions, state access, and callbacks. Non-state executions and callbacks are performed in the main thread, whereas, state accesses are executed by the separate state access thread pool.

Figure 5 illustrates the *AEC* operations for a task of the logistics streaming join operator. The task execution thread consumes *Order* and *Shipping* records from the FIFO input channel and submits state access requests to the *AEC*. Assume record O1 representing *Order* with key 26 is currently waiting for state access. This operation is offloaded to the *AEC* state access thread pool, while the main task thread keeps processing records from the input channel. The *Key Accounting Unit* has recorded key 26 to prevent concurrent access with the same key. The following operations take place, as shown in Figure 5: ① The task execution thread consumes the next input record - *Order* record O2 with key 26. ② O2's state request is sent to the *AEC* for async execution. ③ The *Key Accounting Unit* identifies a conflict on key 26 caused by the pending record O1 and places O2 in the *Blocking Buffer* until all preceding operations on the same key are complete. ④ The main task thread proceeds with the next record - *Order* O3 with key 18. ⑤ Since key 18 currently has no in-progress computations, O3's state request passes through the *Key Accounting Unit* and acquires a state thread for execution. ⑥ The async state request for pending record O1 eventually completes. The callback function (Line 4-5 in Algorithm 1) along with the fetched state is enqueued into the *Callback Channel* for further execution. Records in the callback channel are prioritized over those in the input channel by the main thread to enable faster end-to-end progress. ⑦ After O1 completes, the *AEC* resumes the pending state requests on key 26 in FIFO order. The number of in-flight records and the AEC's blocking buffer are configurable to limit memory consumption. By default, this is set to 6000 records [11], leading to several MBs of memory per operator. If the blocking buffer saturates, the AEC blocks new records, triggering backpressure.

Correctness argument. Per-key FIFO order in Flink 2.0 is provided via a combination of two mechanisms I and II:

I - Inter-Task Ordering (Channel Management) As established in Flink's original design (Section 2.2), records belonging to the same key k arrive at each task in FIFO order. Thus, $r_1 \prec r_2$ in the stream input if r_1 is produced before r_2, given a stream order relation $\prec$.
II - Intra-Task Ordering (AEC). Within each task, the AEC allows only one *in-flight* computation per key at a time. That is, any record r_2 that shares a key k with an ongoing record r_1 must wait in the blocking buffer until r_1 finishes. Consequently, r_1 completes all CPU and I/O steps *before* r_2 can start.

4.4 Exactly-Once Processing Compatibility

As we discussed in Section 2.2, Flink's checkpointing process ensures that all computations (and no more) between two consecutive checkpoint markers are committed [19]. This fundamental mechanism remains intact in Flink 2.0's asynchronous execution model to guarantee exactly-once processing semantics. In Flink 1.x, when an aligned checkpoint marker triggers a local checkpoint in the stream task, all records preceding the marker have been fully processed, and, thus, all state changes have been reflected in the state store. However, the same precondition cannot be guaranteed by default during async execution, as state accesses and post-state callbacks may still be pending upon processing a checkpoint marker. To address this, Flink 2.0 introduces an *async draining mechanism* to ensure that all ongoing state accesses and callbacks are finished before a local checkpoint operation is triggered.

The draining procedure uses the *AEC* to track pending computations while blocking any input that follows the checkpoint marker until all preceding actions are complete. Preceding actions include processing in-flight records and records residing in the blocking buffer, as well as, any derivative state accesses and callbacks. Once all processing has been completed, it is safe to take the local state checkpoint, and emit a new checkpoint marker in the output. The draining procedure prolongs checkpoint duration; however, in most cases, the *AEC*'s management of the maximum number of in-flight records ensures this prolongation is controllable. As shown in Section 6.1, checkpoints can consistently finish within several seconds. From a compatibility perspective, the draining mechanism preserves Flink 2.0's portability with Flink 1.x checkpointing, making it possible to enable or disable asynchronous execution through simple reconfiguration. Finally, the alternative "Chandy-Lamport" checkpoints from Flink 1.x which bypass alignment, are still compatible with Flink 2.0 and are recommended for network-intensive pipelines. Similar to the Flink 1.x version, no draining is necessary but in-flight messages need to be checkpointed as part of the state.

Correctness argument. Let C_k be the aligned input checkpoint marker for checkpoint k. Given an input FIFO network order relation $\prec$, we must ensure that (I) every record r for which $r \prec C_k$ completes before issuing checkpoint k, and (II) no record r' where $C_k \prec r'$ is processed before checkpoint k. Once aligned marker C_k arrives, the operator blocks any subsequent records r' for which $C_k \prec r'$. This satisfies property II. Meanwhile, only when the AEC detects that no operations remain pending for those earlier records does it finalize checkpoint k and emit C_k downstream. This guarantees property I. In the same atomic step, the task also unblocks its regular input, allowing normal processing to resume seamlessly.

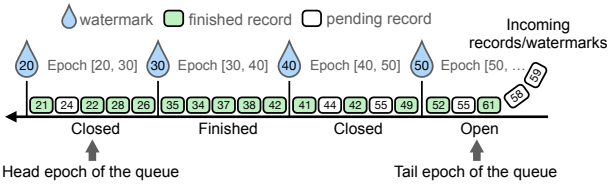

Figure 6: The event-time epoch mechanism in Flink 2.0.

4.5 Watermark Handling with Async Execution

Watermarks provide a consistent mechanism to track global progress across all substreams, by offering monotonicity and completeness in out-of-order processing. Flink's FIFO channels are instrumental in ensuring monotonicity when generating combined watermarks from multiple input streams, as explained. Completeness, on the contrary, relies on synchronous execution. An input watermark w extracts a promise that if a record r originates before the watermark, i.e., $r.t < w$, then that record has been already processed. Many key mechanisms, such as event-time timers and windows, rely on this guarantee to trigger their final computations. However, under async execution, it is possible that there are still pending computations originating earlier than a watermark w at the time of its reception. Hence, either triggering timers or disseminating w further would violate low-watermarking ordering semantics.

To address this, we introduce the notion of event-time "epochs", representing periods between two consecutive watermarks. Epochs are used for asynchronous progress tracking, i.e. tracking pending records per epoch until their completion. An epoch transitions through three distinct stages according to its progress: i) *OPEN*: initially, an epoch begins in that state, awaiting the arrival of the next watermark. During this stage, newly arriving input records are assigned to this open epoch. ii) *CLOSED*: upon the arrival of the next combined watermark, the epoch transitions to this state. Closed epochs cannot be assigned more records, whereas, a new epoch is opened to accept new inputs. iii) *FINISHED*: a closed epoch whose records have been fully processed.

Since records are processed in parallel, multiple epochs with unfinished records may coexist. To estimate completeness and coordinate watermark operations, Flink 2.0 features an *Epoch Manager* component, which maintains a queue of epochs and their information. New epochs and watermarks are enqueued at the tail, and emitted from the head. An epoch can only be dequeued if it is in the *FINISHED* state, and a watermark can only be emitted downstream if it reaches the head of the queue. As shown in Figure 6, the finished epoch [30, 40] and its watermark cannot be dequeued as its preceding epoch [20, 30] still has pending records. The queue ensures that the propagation of the watermarks and their order semantics remain consistent with the synchronous execution.

Correctness argument. Completeness is defined as the guarantee that every record r such that $r.t < w$ is fully processed by the time watermark w is emitted downstream. Flink 2.0 ensures completeness via its epoch management mechanism. Upon processing watermark w, the task *seals* the current epoch E to a *CLOSED* state in the same atomic operation. The Epoch Manager tracks all in-flight records within that epoch and blocks w from propagating until 1) every in-flight record with $r.t < w$ completes processing (E is in *FINISHED* state) and 2) E is at the head of the queue of the Epoch Manager. As a result, once w is emitted downstream, it is guaranteed that there are no unprocessed records with timestamps below w, preserving completeness.

5 ForSt: DISAGGREGATED STATE BACKEND

We now describe Flink 2.0's new state management layer and introduce ForSt [28] (short for "*For Streaming*"), a disaggregated state backend designed for streaming data. ForSt overcomes the limitations of localized state, such as limited storage capacity, resource spikes, and checkpointing and reconfiguration bottlenecks, which are prevalent issues in workloads with large state. ForSt addresses these challenges by decoupling storage from compute through a disaggregated architecture and leveraging scalable remote storage. This design brings the following novelties in data streaming: i) active state operations directly in DFS, eliminating local storage limitations while exploiting local caching; ii) a unified file system (UFS) layer, aligning the behavior (API) and capabilities (particularly fast file sharing) across diverse distributed file systems; iii) seamless, lightweight checkpointing based on the file-sharing mechanism; iv) rapid reconfiguration and recovery via direct state retrieval from the DFS; v) remote compaction, mitigating CPU spikes with checkpoint-triggered compaction in Flink 1.x; and vi) a local cache, utilizing both memory and disk, to accelerate data retrieval.

5.1 Unified File System (UFS)

ForSt employs an LSM-tree structure to manage state files, which are streamed directly to the DFS via a Unified File System (UFS) layer, as illustrated in Figure 7. Distributed storage systems, such as HDFS and object storage services, like OSS and S3, vary widely in terms of behavior and capabilities and also differ from POSIX-compliant local file systems [41]. As an example, while HDFS provides immediate visibility of file creations and modifications, S3 employs an eventual consistency model, which may result in delayed object visibility. Hard links, a POSIX file system concept, create metadata entries that point to the same data blocks (inode) of an existing file, facilitating efficient file sharing. However, most DFS implementations do *not* natively support hard links. As a result, file duplication often necessitates data copying, leading to increased overhead and potential consistency problems. The UFS abstracts away this complexity and provides a single point of entry to access multiple DFS backends.

Given that both checkpoint and state files are now stored on the DFS, it is imperative for ForSt to offer efficient linking mechanisms across diverse DFS implementations to enable lightweight checkpointing and rapid reconfiguration. To this end, the UFS provides a logical file abstraction and delegates file operations. The UFS maintains the mapping and reference counts between logical files and their physical locations and ensures consistent object visibility. This approach facilitates efficient move or link operations without necessitating physical file relocation or data duplication.

5.2 Fast Checkpointing and Reconfiguration

Faster checkpoints. Flink 1.x's checkpointing design (cf. § 2.2) involves state backends copying and uploading their local state files to the DFS at checkpoint time. This copy and upload process to

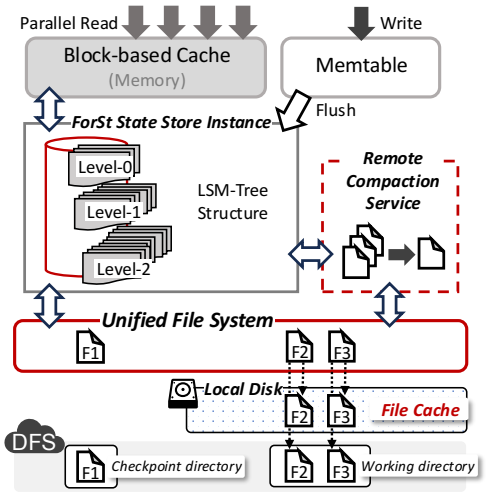

Figure 7: Overall architecture of ForSt.

the DFS is both resource-intensive and time-consuming, particularly for applications with large state sizes, leading to significant resource spikes and prolonged checkpoint time. The Job Manager (JM), acting as checkpoint coordinator, is responsible for managing the lifecycle of these checkpointed files, and state backends are responsible for managing their working state files.

ForSt's disaggregated design optimizes checkpointing in Flink 2.0 by co-locating the working state and checkpoint directories. Leveraging hard-linked file sharing capabilities provided by UFS, the majority of underlying state files are already durably replicated in the DFS when a checkpoint is triggered. This transforms checkpointing from a data-intensive operation to a lightweight reference creation. Specifically, the ForSt checkpointing process comprises the following steps, as Figure 8 illustrates: i) State backends create hard-links for state files to be checkpointed, with the UFS managing logical-to-physical file mappings and reference counts (Figure 8 ①); ii) These hard-linked logical copies are registered in the JM (Figure 8 ②), as in Flink 1.x. Once a checkpoint becomes obsolete, the job manager issues a deletion to the state backend via its hard-linked reference instead of commencing a direct file deletion (Figure 8 ④ and ⑤). iii) Hard-linked copy deletions trigger a reference count reduction in the UFS. This results in safe physical file deletions only when the reference count defaults to zero (Figure 8 ③ and ⑥).

Checkpointing in Flink 2.0 closely follows the original mechanism in Flink 1.x, with one key refinement: instead of directly deleting files, the JM delegates deletion to the UFS. This decouples checkpointing from physical file management, while significantly reducing overhead without compromising the original guarantees.

Seamless recovery and rescaling. Failure recovery and rescaling necessitate state backends to download or copy files from the checkpoint to their working directory, which is slow for large state sizes. As we will show in Section 6.1, recovering a job with 290 GB of state takes over 3 minutes in Flink 1.20, while large-scale cluster recovery or migration scenarios (e.g. during Double 11) can take hours. However, leveraging the hard-linking feature of the UFS, data transfer can be entirely avoided. New instances can recover directly from linked file copies, resulting in near-instant recovery.

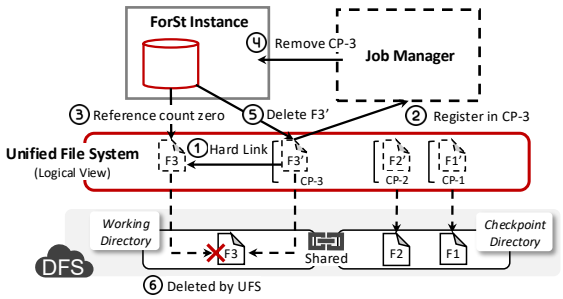

Figure 8: Fast checkpointing on Unified File System.

5.3 Remote Compaction

Compaction is the process of merging immutable files in LSM-Trees to remove obsolete or deleted data, reducing storage overhead and improving read performance. While crucial for read performance and space efficiency, compaction introduces recurring CPU and I/O contention in monolithic architectures. In Flink 1.x, this overhead is further amplified by periodic checkpointing, as each checkpoint captures a full snapshot of the working state, triggering local compaction. Consequently, foreground tasks contend with background compactions for resources, leading to latency fluctuations and occasional resource spikes. In practice, this often results in resource overprovisioning, reducing overall efficiency.

The disaggregated design of ForSt facilitates the decoupling of the CPU-intensive compaction process from Flink compute nodes. Since the active working state is stored on a shared DFS, state files for compaction are directly accessible without disturbing normal processing. This enables ForSt to introduce compaction-as-a-service (Remote Compaction), where dedicated stateless compactor workers execute file compactions triggered by Flink tasks. This design significantly enhances the flexibility of the setup, as the compactor and compute nodes can scale independently, effectively decoupling the two resource types to handle their respective workloads. Given the I/O-intensive nature of compaction, the compactor can be deployed either within the same intranet as the DFS cluster or locally, regardless of the network configuration of the compute nodes. As a shared service, resource spikes from compactions across different jobs can be staggered, leading to more stable CPU usage and a higher overall resource utilization rate. This optimization aligns with the pooling concept prevalent in the cloud-native era. Our implementation [36] adopts the compaction trigerring of Flink 1.x but offloads the actual process. Instead of compacting locally, the ForSt backend sends a compaction request with metadata to a dedicated service. A scheduling node assigns tasks to compactors using a round-robin policy. Once complete, the backend is notified and updates LSM metadata accordingly.

5.4 Local Cache Management

Caching is essential for faster read operations in disaggregated architectures. ForSt incorporates a caching mechanism that uses both memory and local disk resources on compute nodes as a multi-tiered cache hierarchy. ForSt includes a widely adopted block-based Least Recently Used (LRU) cache in memory and a file-based secondary cache on local disks. The secondary cache replicates SSTable files from remote storage and uses a *History-Based Policy*.

Table 2: Minimum Number of CUs and Associated Costs

	Flink 1.20	Flink 2.0
Minimum Required CUs	16	8
Monthly cost ($)	688	344

This History-Based Policy manages cache files using historical access statistics. For eviction, we employ an LRU mechanism to track all currently cached files, evicting the least recently accessed files when the cache reaches its capacity and new files need to be loaded. For loading, access frequency over the preceding minute is monitored. Files on the remote store with an access frequency exceeding a predefined threshold are periodically loaded back into the cache. The LRU-based eviction policy, while simple, has proven effective in our production deployments, while the frequency-based loading policy effectively mitigates the cache thrashing problem. The history-based policy is the default cache policy in Flink 2.0 and in Alibaba's Flink Service. The cache policy is designed to be pluggable, allowing for easy substitution.

6 EVALUATION

Our experimental evaluation consists of two parts. In Section 6.1, we evaluate the benefits of disaggregated state management on the production use case of Figure 3, in Flink 1.20 and Flink 2.0. Our results show that Flink 2.0 leads to lower operating costs, higher checkpointing performance, and reduced reconfiguration duration. In Section 6.2, we use the Nexmark benchmark [55, 60] to further evaluate the performance impact of the asynchronous execution model, the caching mechanism, and the ForSt backend.

6.1 Real-World Workload Evaluation

We replicate the logistics use case of Section 3.1 on a smaller scale, using a subset of production workloads, resulting in a total state size of roughly 290 GB. We adjust the input rate to match the daily production traffic. We use Flink 1.20 as the baseline, with state backed by RocksDB and stored on the local disk.

Experimental setting. We run all experiments on a Kubernetes cluster with 6 `ecs.g7.8xlarge` Elastic Compute Service (ECS) instances. Each instance is equipped with 32 2.7 Ghz Intel Xeon vCPUs, 128 GB of main memory, one network card with 16 Gb/s intra-network bandwidth and one `ESSD PL1` disk, offering 50K IOPS and 350 MB/s throughput. This setup is based on the commercial model for Alibaba Cloud's Real-time Computing service [27].

For storage, we configure an HDFS cluster as a replacement for Alibaba Pangu [48]. We use one NameNode and two DataNodes, each deployed on an `ecs.g6a.2xlarge` ECS instance, equipped with 8 2.6 Ghz AMD EPYC vCPUS, 32 GB of memory, and one network card with 2.5 Gb/s intra-network bandwidth. The HDFS cluster can deliver up to 5 Gb/s throughput, which aligns with our production compute-storage configuration. To determine the optimal disk configuration for DataNodes, we consider both disk latency and throughput. We choose `ESSD PL0` disks, which are sufficient to provide $\approx 1ms$ access latency between compute and storage clusters. We provision 4 disks per DataNode, offering over 5.6 Gb/s of total disk throughput, which exceeds the network bandwidth.

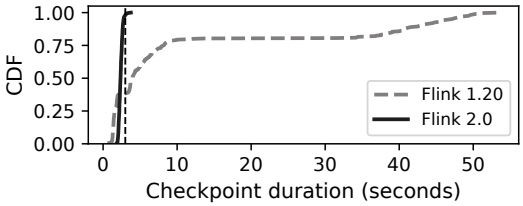

Figure 9: CDF of the checkpoint durations.

Cost evaluation. In this experiment, we compare the operating cost of the two deployments on Flink 1.20 and Flink 2.0. Specifically, we measure the monthly cost in terms of the compute units (CUs) required by each system to sustain the daily traffic of the production workload. Modern containerized service providers bundle resources like CPU, memory, and local disk, rather than selling them separately [30, 44]. Consequently, we have adopted the compute unit (CU) as an integrated measure for evaluating cost. Based on Alibaba Cloud's pricing model [26], the monthly costs for compute and storage clusters are approximately $7300 and $1000, respectively. The cost price of one CU — which includes one CPU core, 4 GB of memory, and 20 GB of disk storage — is roughly $43 per month. Notice that Flink 2.0 does not incur additional costs on HDFS compared to Flink 1.20. This is because both versions store checkpoint data on HDFS; Flink 2.0 simply co-locates and shares checkpoint and active state data without extra storage cost.

Table 2 shows the minimum number of CUs required to run the job while avoiding backpressure. Flink 1.20 needs at least 16 CUs, to avoid the state size per CU exceeding the disk limit of 20 GB. Conversely, Flink 2.0 is memory-bound but handles the job with only 8 CUs, cutting costs by half ($344 per month).

Checkpoint performance. We now compare the performance of the checkpointing mechanism in Flink 1.20 and Flink 2.0. Our results indicate that Flink 2.0 provides substantially faster and more consistent checkpointing, key determinants of overall checkpointing performance. We set the checkpoint interval to 1min and enable incremental checkpoints. We run the experiment for five hours and record metrics for 300 checkpoints for each system. Figure 9 shows the CDF of checkpoint durations. Overall, all the checkpoints in Flink 2.0 can finish within 3 seconds, no matter how big the checkpoint size is, while Flink 1.20's checkpoint duration is largely determined by the incremental checkpoint size. In this case, with an average incremental checkpoint size of 1.89 GB, over 19.7% of the checkpoints took more than 30 seconds to complete, and more than 1.5% took more than 50 seconds.

Flink 2.0 is able to complete checkpoints within seconds thanks to its disaggregated state management. Co-locating the ForSt active state and checkpoint directories and leveraging UFS to eliminate file copies significantly reduces the data transfer required during checkpointing, ensuring both speed and stability. To validate the versatility of the UFS-based approach, we replicated the experiment on OSS instead of HDFS, keeping the setup identical. OSS is using the recommended standard zone-redundant type [25]. The findings were consistent: in Flink 2.0, checkpoint durations remained fast and stable, with all checkpoints completed within 4 seconds. In contrast, checkpoint times in Flink 1.20 exhibited significant fluctuations due to the long-tail transfer latency of OSS.

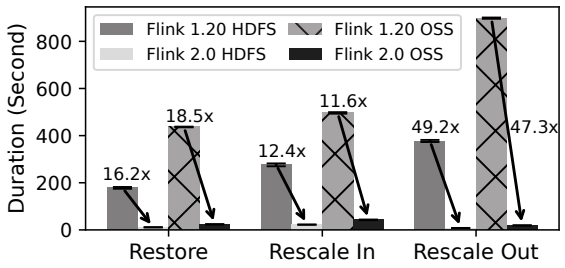

Figure 10: Recovery and rescale durations.

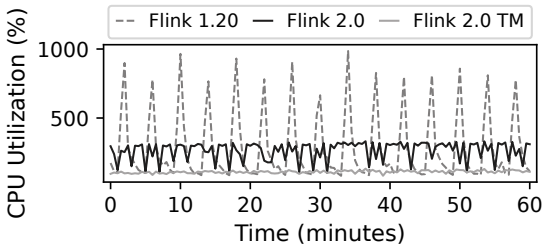

Figure 11: Total CPU utilization over one-hour duration in Flink 1.20 and Flink 2.0 with remote compaction enabled.

Recovery and rescaling. Our next experiment aims to demonstrate how disaggregated state management facilitates faster reconfiguration in Flink 2.0. Flink employs pause-and-restart reconfiguration to recover after a failure and to modify the resource allocation of a job. To achieve state consistency during reconfiguration, Flink temporarily pauses the affected job and instructs workers to load state from the most recent checkpoint. Data processing can resume once all workers have loaded their respective state partitions.

We evaluate three production-grade reconfiguration scenarios representative of peak traffic events, such as Double 11 sales: (i) cross-cluster migration, where we terminate a job on an overloaded primary compute cluster and seamlessly restart it on a backup cluster, (ii) a scale-out scenario, where we double job parallelism (16 to 32) during traffic surges, and (iii) a scale-in scenario, where we half job parallelism (32 to 16) post-peak, to save costs. We repeat each experiment three times on Flink 1.20 and Flink 2.0, testing each version with both HDFS and OSS as the storage, while keeping all other settings identical. OSS is using the recommended standard zone-redundant type [25]. Figure 10 shows the results.

With HDFS, Flink 2.0 significantly shortens the reconfiguration duration and restarts the job within tens of seconds, in all scenarios. Compared to Flink 1.20, it provides 16× faster job recovery, 12× faster scale-in, and 49× faster scale-out. Flink 2.0 offers superior performance thanks to its ability to reconfigure the job without requiring data transfer from remote storage to the compute nodes' local disks. On the contrary, Flink 1.20 takes several minutes to download and rebuild job state in the corresponding local RocksDB instances. For recovery and scale-in, the data transfer volume is roughly 290 GB, that is, equal to the total job state size. In the scale-out scenario, however, the data transfer volume is double, as each scale-out worker (32 parallelism) fetches data stored in one original worker (16 parallelism). As a result, the benefits of disaggregated state management are even more evident for scale-out.

Repeating the experiments with OSS as the distributed file system yielded a similar conclusion: Flink 2.0 achieves significantly shorter reconfiguration durations compared to Flink 1.20. When compared to HDFS, we observe that reconfiguration using OSS in Flink 2.0 takes approximately 10-20 seconds longer. This is because object storage services generally exhibit lower performance than HDFS for metadata lookups and small random I/O operations. Although our UFS approach eliminates the primary cost of copying large data files, loading metadata to rebuild the ForSt instance is still required. In future work, we plan to optimize this by merging metadata reads at startup to further accelerate recovery.

Smooth Resource Utilization. In this experiment, we demonstrate the smooth resource utilization of Flink 2.0 with remote compaction enabled. The compaction service was deployed in an isolated Kubernetes pod. We collect CPU utilization metrics from all nodes throughout a one-hour job execution for each system. The checkpoint interval was set to 1 minute and incremental checkpoints were enabled. Figure 11 shows the results. **Flink 1.20** denotes the CPU usage of all Task Managers (TMs) in Flink 1.20. **Flink 2.0** includes the CPU usage of all TMs plus the compaction service, and **Flink 2.0 TM** represents only CPU usage of TMs in Flink 2.0.

Flink 1.20 shows significant and periodic peaks in CPU usage, aligned with the checkpoint intervals. This is mainly caused by background processes, like checkpoint file uploads and LSM-Tree compactions. In contrast, CPU utilization in Flink 2.0 is more stable. In particular, **Flink 2.0 TM** exhibits consistently low variance and is unaffected by checkpoint operations, due to the improved checkpoint mechanism: 1) leveraging UFS to eliminate file copies significantly reducing the data transfer required during checkpointing (serialization and data copying are CPU-intensive), and 2) compaction tasks are moved from TMs to dedicated compaction nodes. Consequently, Flink TMs exhibited a substantially smaller and smoother CPU usage profile. As for the compaction nodes, their CPU usage is upper-bounded by the resource limits configured for them. This effectively stabilizes the overall cluster CPU usage as long as the compaction service can keep pace with the compaction demands.

Similar smoother usages in I/O and network are observed due to UFS, but not detailed here due to space constraints. In conclusion, Flink 2.0 can prevent excessive resource provisioning by eliminating sudden surges in resource consumption.

6.2 Performance on Nexmark benchmark

In the next set of experiments, we take a closer look at the performance of the disaggregated state design. We aim to understand what the impact is, if any, on steady-state performance and the trade-offs to achieve the higher flexibility and low-latency reconfiguration benefits we demonstrated in the previous section. To this end, we use the well-established Nexmark benchmark [29] to evaluate the performance of the asynchronous execution model, the caching mechanism, and the ForSt backend. Nexmark provides a range of continuous query types, including stateless queries, light state I/O queries, and heavy state I/O queries, allowing us to evaluate performance across different workloads.

Experimental setting. We run all experiments in this section on two clusters hosted on Alibaba Cloud. The compute cluster consists of 4 ECS instances, each equipped with 16 vCPUs and 64 GB of

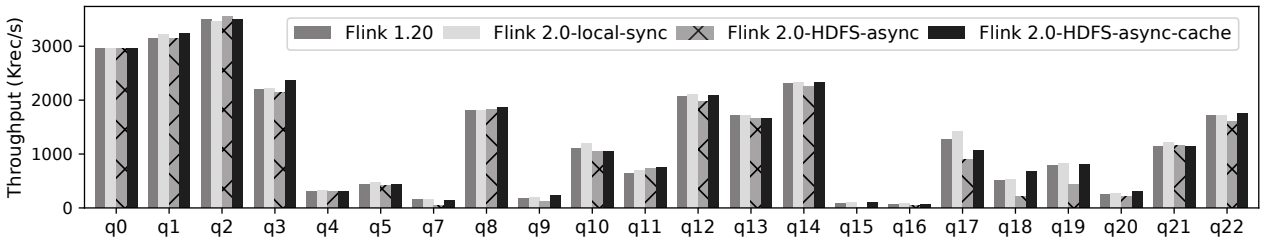

Figure 12: Maximum achieved throughput for all Nexmark queries under various configurations.

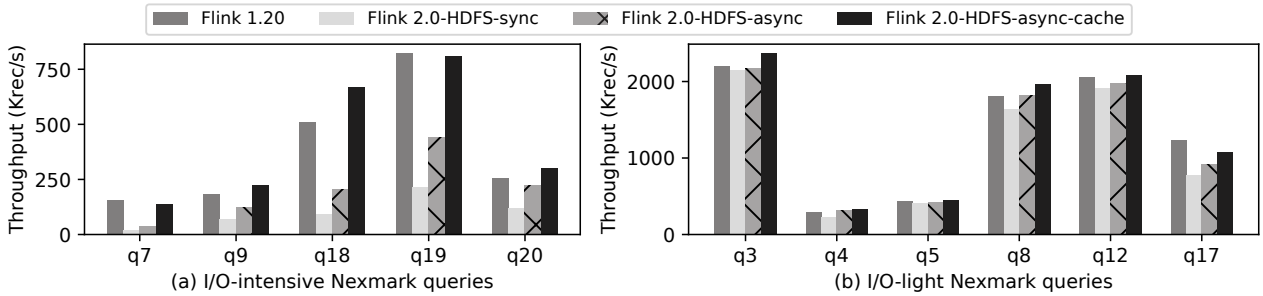

Figure 13: Maximum achieved throughput for I/O-intensive and I/O-light Nexmark queries under various configurations.

memory, one 5 Gbps network adapter, and one 100 GB ESSD PL0 disk, providing 3K IOPS and 125 MB/s throughput. For the HDFS storage cluster, we use one NameNode and 4 DataNodes, each deployed on an ECS instance, equipped with 8 vCPUs, 32 GB of memory, one 2.5 Gbps network adapter, and one 100 GB ESSD PL0 disk, providing 3K IOPS and 125 MB/s throughput.

In all experiments, we deploy Flink 1.20 and various configurations of Flink 2.0 on the fixed set of resources described above. We configure the Nexmark source to generate 200 million records in total and we measure the maximum throughput achieved (records per second). We disabled the bloom filter on both Flink 1.20 and 2.0 to maximize I/O and evaluate performance under high I/O load.

Maximum sustainable throughput across all queries. In the first experiment, we evaluate the overall performance of Flink 1.20 and Flink 2.0 across all Nexmark queries,[1] except Q6, as Flink SQL does not yet support it. The Flink 1.20 nodes were configured with 100 GB of local storage for RocksDB, which is significantly larger than the actual requirements of the queries, to simulate an environment with unconstrained local disk resources. To understand the performance impact of the newly introduced components in Flink 2.0, we evaluate three distinct configurations:

(1) Flink 2.0-local-sync uses 100 GB of local storage but replaces RocksDB with ForSt. Asynchronous execution is disabled.

(2) Flink 2.0-HDFS-async uses the disaggregated state store on HDFS and employs asynchronous execution. The compute nodes do not have any local disk cache.

(3) Flink 2.0-HDFS-async-cache uses disaggregated state, asynchronous execution, and a 1GB local disk cache.

Figure 12 shows the results. When comparing Flink 1.20 and **Flink 2.0-local-sync**, the two configurations with active state on the local disk, we observe that they achieve almost identical throughput. This indicates that Flink 2.0 with ForSt can serve as a seamless replacement for Flink 1.x with RocksDB, maintaining the same performance level. Enabling disaggregated state management without any cache (**Flink 2.0-HDFS-async**) incurs a moderate overhead, degrading throughput by 48% on average for queries with heavy I/O. The primary reason for this degradation is the increased latency introduced by remote I/O operations. Nevertheless, for the majority of queries without heavy I/O, disaggregated state management does not have any significant performance impact. Finally, the results for **Flink 2.0-HDFS-async-cache** show that the caching mechanism can effectively mitigate this impact. Even with a limited 1GB local disk cache, it outperforms local state setup (Flink 1.20 and **Flink 2.0-local-sync**) by 4% on average.

I/O-intensive queries. We now select the five I/O-intensive Nexmark queries for closer study. We run these queries on the same Flink 1.20 configuration as before and gradually enable disaggregated state management features to assess the benefits of asynchronous execution and caching. We evaluate the following settings:

(1) Flink 2.0-HDFS-sync uses the disaggregated state store on HDFS with *synchronous* state access.

(2) Flink 2.0-HDFS-async enables asynchronous execution but has no local disk cache.

(3) Flink 2.0-HDFS-async-cache uses a 1GB local disk as cache.

Figure 13 (a) shows the results. As expected, naively employing disaggregated state management with synchronous execution (**Flink 2.0-HDFS-sync**) leads to severe performance degradation. **Flink 2.0-HDFS-async** improves throughput by $\approx 2\times$, demonstrating that the asynchronous execution model effectively mitigates the I/O bottleneck. **Flink 2.0-HDFS-async-cache** further improves performance by up to 3.7×, indicating the critical role of caching. Note that the 1 GB local disk cache is not enough to hold the state

[1]Queries Q0-Q2 are stateless queries, which complete within tens of seconds, so small disturbances of JVM GC will have a noticeable impact on the results.

size for any of the queries (q7: 2.25 GB, q9: 4.48 GB, q18: 1.05 GB, q19: 1.52 GB, q20: 2.95 GB). However, even with only 1 GB of cache, disaggregated state management can achieve performance that is comparable to or better than Flink 1.20 with a local disk. Empirically, the extra bandwidth consumption caused by remote state access does not pose a problem compared to available intra-network bandwidth in modern DC environments. For example, OSS offers a default bandwidth of 5 Gbps (which aligns with our experimental setup) and can scale further according to demand.

Queries with light I/O. We further evaluate six queries with state sizes from 10MB to 400MB, where states fit in the block cache rendering disk I/O negligible. As shown in Figure 13 (b), throughput differences across Flink 2.0 configurations are much smaller than in high-I/O scenarios (cf. Figure 13 (a)). For CPU- and memory-bound tasks, async threading and caching offer little benefit, as dispatching overhead exceeds memory access gains.

CPU overhead. We evaluated the overhead of the disaggregated state model with asynchronous data access. Nexmark benchmark results demonstrate a 30% average increase in CPU utilization for stateful operators. This increase is primarily attributed to: (i) context-switching cost for asynchronous access (30%); (ii) AEC intra-task scheduling cost (20%); (iii) batching I/O classification and parallel execution for remote access (20%); and (iv) additional Java garbage collection for Future objects (30%). While Flink 2.0 with disaggregated state management benefits large-scale, I/O-intensive stateful jobs where CPU is typically not the bottleneck, we plan to further reduce this CPU cost in future work. In contrast, for workloads with minimal or no I/O, the added CPU overhead from async execution and DFS access may outweigh benefits. Flink 2.0 supports both sync and async modes, letting users choose based on workload. In practice, overhead is often low, as stateless operators skip these steps. Most jobs mix stateful and CPU-intensive stateless operators (e.g., sources, sinks). Our logistics use case (Figure 11) shows Flink 1.20 using 9% more CPU than Flink 2.0, regardless of remote compaction. This is due to Flink 1.20 requiring 16 task managers (TMs) versus 8 in Flink 2.0, thus, increasing TM framework overhead. Asynchronous overhead is not dominant in this scenario.

7 RELATED WORK

Many database systems architectures decouple the storage engine from the compute engine [2, 8, 10, 16, 33, 34, 46, 61, 62, 64, 66]. While these systems share the same underlying motivation with Flink 2.0, their functional and operational requirements differ significantly from those of streaming systems. In the rest of this section, we only review closely related works on stream processing systems.

Stream processing systems with external state management. Various stream processing systems separate the compute layer from state, by pushing state management responsibilities to a distributed key-value store. MillWheel [4] uses BigTable [22] and Spanner [31] as remote storage, coupling per-key state modifications and checkpoints in the same atomic write. While this approach achieves fault tolerance at task and record granularity, remote state access is performed synchronously, on the critical path of record processing, to guarantee exactly-once semantics and idempotent updates. Meta (formerly Facebook) [23] employs ZippyDB [50] for remote storage, reducing failure recovery time by avoiding complete state loading during restarts. However, this model is primarily suited for monoid processors [14], when updates can be performed as local append-only operations. Storm [59] and S4 [54] offload state management and durability to users, without providing framework support for direct remote state access. Recent SQL streaming database systems [15, 45] also offer native state caches and support for remote cloud storage. RisingWave's architecture [45] is perhaps the closest one to Flink 2.0, using remote storage as the source of truth and employing remote compaction. While these approaches benefit from disaggregation, they focus on SQL APIs and offer limited support for custom state and user-defined event-based logic.

Asynchronous and out-of-order execution. Stream processors rely on event time and watermarks to maintain order and ensure correctness [4, 5, 19, 42, 47, 53, 57]. However, disorder can also be introduced internally, after the event's arrival. StreamBox [51] and Cameo [65] offer out-of-order processing and dynamic scheduling to either leverage parallelism within a single multi-core machine or prioritize events according to user-defined latency deadlines. Flink 2.0 allows out-of-order processing across watermark boundaries, without incurring additional scheduling overhead.

Evolution of stream processing architectures. Data stream processing research can be traced back to systems like Tapestry [58], TelegraphCQ [21], STREAM [9], NiagaraCQ [24], Aurora / Borealis [1, 13, 20], Gigascope [32], and S-Store [49]. These systems established the core principles of modern stream processors, paving the way for today's commercial and open-source success. Dataflow architectures continue to evolve, driven by emerging applications, cloud computing, and hardware trends [18, 38].

8 CONCLUSION

Over the past decade, Apache Flink has become a popular choice for distributed stateful applications due to its low-latency, high-performance state management. However, its tightly-coupled architecture limits checkpointing and reconfiguration speed at large state sizes. Flink 2.0 overcomes these limitations through disaggregated state management while making no compromises to existing guarantees. Our evaluation exhibits fast reconfiguration and state scalability with ForSt, an LSM-tree structured state store targeting remote storage systems built on top of Flink's new asynchronous execution model, utilizing non-blocking state access.

9 ACKNOWLEDGMENTS

The work in this paper was made possible primarily by the Apache Flink community's dedicated and collaborative efforts. Special thanks go to Jark Wu, Lincoln Lee, and Xuyang Zhong for their work on rewriting and adapting asynchronous SQL operators. We also sincerely thank Jinzhong Li, Hangxiang Yu, Piotr Nowojski, Gyula Fóra, Guowei Ma, Xintong Song, Gen Luo, Yunfeng Zhou, Jane Chan, and Jeyhun Karimov for their invaluable input and contributions. Our thanks extend to all contributors not listed due to space limitations and to our anonymous reviewers. The work of Lei Huang and Vasiliki Kalavri was partially supported by the National Science Foundation under Grant No. 2440334. Paris Carbone was funded by the Wallenberg Launchpad and WASP-NEST.

REFERENCES

[1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB J.* 12, 2 (2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z

[2] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, et al. 2020. POLARIS: the distributed SQL engine in azure synapse. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3204–3216.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[5] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills, and Dan Sotolongo. 2021. Watermarks in stream processing systems: semantics and comparative analysis of Apache Flink and Google cloud dataflow. *Proc. VLDB Endow.* 14, 12 (July 2021), 3135–3147. https://doi.org/10.14778/3476311.3476389

[6] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[7] Alibaba. 2025. *Alibaba Realtime Compute.* https://www.alibabacloud.com/product/realtime-compute Last access: March 2025.

[8] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data.* 1743–1756.

[9] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. 2003. STREAM: the stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data.* 665–665.

[10] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data.* 2205–2217.

[11] Asynchronous Execution Model 2024. https://cwiki.apache.org/confluence/x/S4p3EQ.

[12] Asynchronous State APIs 2024. https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=293046857.

[13] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. 2005. Fault-tolerance in the Borealis distributed stream processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 13–24. https://doi.org/10.1145/1066157.1066160

[14] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: a framework for integrating batch and online MapReduce computations. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1441–1451. https://doi.org/10.14778/2733004.2733016

[15] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1601–1614.

[16] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.

[17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[18] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond analytics: The evolution of stream processing systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data.* 2651–2658.

[19] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[20] Uğur Çetintemel, Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Samuel Madden, Anurag Maskey, Alexander Rasin, et al. 2016. The aurora and borealis stream processing engines. In *Data Stream Management.* Springer, 337–359.

[21] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Frederick Reiss, and Mehul A. Shah. [n.d.]. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data.* 668. https://doi.org/10.1145/872757.872857

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. 26, 2, Article 4 (June 2008), 26 pages. https://doi.org/10.1145/1365815.1365816

[23] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1087–1098. https://doi.org/10.1145/2882903.2904441

[24] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. [n.d.]. NiagaraCQ: A scalable continuous query system for internet databases. ([n. d.]), 379–390. https://doi.org/10.1145/342009.335432

[25] Alibaba Cloud. 2025. *Alibaba Cloud OSS.* https://www.alibabacloud.com/help/en/oss/user-guide/overview-53 Last access: July 2025.

[26] Alibaba Cloud. 2025. *Alibaba Cloud Pricing.* https://www.alibabacloud.com/en/product/ecs-pricing-list/en Last access: July 2025.

[27] Alibaba Cloud. 2025. *Resource Setup in Alibaba Cloud's Real-time Computing Service.* https://www.alibabacloud.com/help/en/flink/product-overview/basic-concepts Last access: July 2025.

[28] ForSt Community. 2024. *ForSt Project.* https://github.com/ververica/ForSt/ Last access: July 2025.

[29] Nexmark Community. 2020. *Nexmark Github repo.* https://github.com/nexmark/nexmark/ Last access: July 2025.

[30] Compute Units in Alibaba Cloud 2024. Metering method of Realtime Compute for Apache Flink in Alibaba Cloud. https://www.alibabacloud.com/help/en/flink/product-overview/billable-items.

[31] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI.*

[32] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. [n.d.]. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data.* 647–651. https://doi.org/10.1145/872757.872838

[33] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data.* 215–226.

[34] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, et al. 2020. Taurus database: How to be fast, available, and frugal in the cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1463–1478.

[35] Apache Flink. 2025. *Powered by Flink.* https://flink.apache.org/what-is-flink/powered-by/ Last access: July 2025.

[36] Flink Remote Compaction 2025. https://github.com/AlexYinHan/flink/tree/remote_compaction_feature.

[37] Apache Software Foundation. 2025. *Apache Flink Committee.* https://projects.apache.org/committee.html?flink Last access: July 2025.

[38] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. 2024. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (2024), 507–541.

[39] Google. 2025. *Google Cloud Dataflow.* https://cloud.google.com/dataflow Last access: March 2025.

[40] Fabian Hueske and Vasiliki Kalavri. 2019. *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications.* O'Reilly Media.

[41] IEEE and The Open Group. 2018. *The Open Group Base Specifications Issue 7, 2018 edition.* https://pubs.opengroup.org/onlinepubs/9699919799 Last access: July 2025.

[42] Gabriela Jacques-Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. 2016. Consistent regions: Guaranteed tuple processing in ibm streams. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1341–1352.

[43] Keynote Flink Forward 2024. The Past, Present, and Future of Apache Flink. https://www.alibabacloud.com/blog/the-past-present-and-future-of-apache-flink_601867.

[44] Kinesis Processing Units in AWS 2024. Managed Service for Apache Flink application resources in AWS. https://docs.aws.amazon.com/managed-flink/latest/java/how-resources.html.

[45] RisingWave Labs. 2025. *RisingWave.* https://risingwave.com/ Last access: March 2025.

[46] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[47] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment* 1, 1 (2008), 274–288.

[48] Qiang Li, Qiao Xiang, Yuxin Wang, et al. 2023. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 331–346. https://www.usenix.org/conference/fast23/presentation/li-qiang-deployed

[49] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, et al. 2015. S-store: Streaming meets transaction processing. *arXiv preprint arXiv:1503.01143* (2015).

[50] Meta. 2021. *ZippyDB: Facebook's key value store.* https://engineering.fb.com/2021/08/06/core-infra/zippydb/ Last access: July 2025.

[51] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: Modern Stream Processing on a Multicore Machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 617–629. https://www.usenix.org/conference/atc17/technical-sessions/presentation/miao

[52] Microsoft. 2025. *Azure Stream Analytics.* https://azure.microsoft.com/en-us/services/stream-analytics/ Last access: March 2025.

[53] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[54] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed Stream Computing Platform. In *2010 IEEE International Conference on Data Mining Workshops*. 170–177. https://doi.org/10.1109/ICDMW.2010.172

[55] Nexmark Queries [n.d.]. NEXMark benchmark. http://datalab.cs.pdx.edu/niagaraST/NEXMark.

[56] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

[57] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible time management in data stream systems *(PODS '04)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/1055558.1055596

[58] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. [n.d.]. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. 321–330. https://doi.org/10.1145/130283.130333

[59] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[60] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. *NEXMark—A Benchmark for Queries over Data Streams.* Technical Report. OGI School of Science & Engineering at OHSU.

[61] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[62] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 449–462.

[63] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, et al. 2021. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 international conference on management of data*. 2602–2613.

[64] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated database systems. In *Companion of the 2023 International Conference on Management of Data*. 37–44.

[65] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2021. Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 389–405.

[66] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.