State Management in Apache Flink®

Consistent Stateful Distributed Stream Processing

Paris Carbone[†] Stephan Ewen[‡] Gyula Fóra[⋆]
Seif Haridi[†] Stefan Richter[‡] Kostas Tzoumas[‡]

[†]KTH Royal Institute of Technology
{parisc,haridi}@kth.se

[⋆]King Digital Entertainment Limited
gyula.fora@king.com

[‡]data Artisans
{stephan,s.richter,kostas}
@data-artisans.com

ABSTRACT

Stream processors are emerging in industry as an apparatus that drives analytical but also mission critical services handling the core of persistent application logic. Thus, apart from scalability and low-latency, a rising system need is first-class support for application state together with strong consistency guarantees, and adaptivity to cluster reconfigurations, software patches and partial failures. Although prior systems research has addressed some of these specific problems, the practical challenge lies on how such guarantees can be materialized in a transparent, non-intrusive manner that relieves the user from unnecessary constraints. Such needs served as the main design principles of state management in Apache Flink, an open source, scalable stream processor.

We present Flink's core pipelined, in-flight mechanism which guarantees the creation of lightweight, consistent, distributed snapshots of application state, progressively, without impacting continuous execution. Consistent snapshots cover all needs for system reconfiguration, fault tolerance and version management through coarse grained rollback recovery. Application state is declared explicitly to the system, allowing efficient partitioning and transparent commits to persistent storage. We further present Flink's backend implementations and mechanisms for high availability, external state queries and output commit. Finally, we demonstrate how these mechanisms behave in practice with metrics and large-deployment insights exhibiting the low performance trade-offs of our approach and the general benefits of exploiting asynchrony in continuous, yet sustainable system deployments.

1. INTRODUCTION

Traditionally, when implementing data-driven applications and services, *state* was separated from the application logic that performs the computation on data. The typical architecture has the state centralized in a database management system shared among applications that are either stateless, or rely on the database for data consistency and scalability among others. Recently, stream processing has been gaining tremendous attention in the industry as a paradigm to implement both analytical applications on "real-time" data, but also as a paradigm to implement data-driven applications and services that would otherwise interact with a shared external database for their data access needs. The stream processing paradigm is more friendly to modern organizations that separate engineering teams vertically, each team being responsible for a specific feature or application, as it allows state to be distributed and co-located with the application instead of forcing teams to collaborate by sharing access to the database. Further, stream processing is a natural paradigm for *event-driven* applications that need to react fast to real-world events and communicate with each other via message passing.

In point of fact, stream processing is not a new concept; it has been an active research topic for the database community in the past [29, 26, 17, 21] and some (but not all) of the ideas that underpin modern stream processing technology are inspired by that research. However, what we see today is widespread adoption of stream processing across the enterprise beyond niche applications where stream processing and Complex Event Processing systems were traditionally used. There are many reasons for this: first, new stream processing technologies allow for massive scale-out, similar to MapReduce [31] and related technologies [46, 20, 22]. Second, the amount of data that is generated in the form of event streams is exploding. Processing needs now spread beyond financial transactions, to user activity in websites and mobile apps, as well as data generated by machines and sensors in manufacturing plants, cars, home devices, etc. Third, many modern state of the art stream processing systems are open source allowing widespread adoption in the developer community.

Earlier attempts to distributed stream processing [8] provided distributed programming model semantics but focused on the challenge of producing real-time, perhaps approximate results that would later be augmented or corrected by more reliable, periodic (e.g., overnight) batch compute jobs (e.g., Lambda Architecture [41]). While this addresses real-time compute on data records, most challenges related to consistent state management remain a concern of the user and typically rest upon external database management systems or traded off for further scalability.

We identified a set of hard challenges, faced daily by developers that architect critical continuous applications in the real world. First, the lack of explicit computational state abstractions in stream processing systems forces them to declare and maintain state externally, decoupled from computational logic. Hence, the burden of ensuring data consistency lies in the application logic, coordinating computation with external database systems. Often, this is complex to maintain as the code-base is divided across the state it manages. Second, transactions with external storage can become

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

the bottleneck of the whole application. Finally, operational challenges arise when there is need to scale in or out, deal with partial failures or to simply change application logic with software patches. While several of these important state management issues have been previously researched and applied in production systems, most known approaches fall short of doing so in a transparent manner. For example, micro-batching techniques for reliable continuous processing (e.g. Apache Spark and Trident [47, 16]) sacrifice programming model transparency and processing latency by enforcing batch-centric application logic. Other proprietary continuous processing system solutions [18] on the other hand, build on heavy transactional per-record processing. This pushes critical complexities outside the system relying on high-performance key-value stores, special hardware and optimized network infrastructure.

Apache Flink [23, 7] is a stream processing system that addresses these challenges by closely integrating state management with computation. Flink's dataflow execution encapsulates distributed, record-centric operator logic to express complex data pipelines. Consistent application state is a first-class citizen in data processing pipelines written in Flink and is persisted using a modular state backend. Furthermore, the system manages operations on state and orchestrates failure recovery and reconfiguration (scale-out/in) whenever necessary without imposing heavy impact on the execution or violating consistency. The core of our approach in Apache Flink builds on *distributed snapshots*, a classical concept that is proliferating anew today. Distributed snapshots enable rollback recovery of arbitrary distributed processes [33] to a prior globally consistent execution state. Several distributed computing systems have used different variations of snapshotting mechanisms [42, 40], though adopting sub-optimal protocols that impose global synchrony and thus halt computational progress while also persisting more state than required (i.e. records within network buffers).

In this work, we present a complete, continuous state management solution that builds on distributed snapshots. Flink's snapshotting mechanism has been in use since version 0.9 (June 2015) and therefore hardened throughout frequent releases of the framework and extensively tested in production at some of the largest stream processing deployments in the world, on thousands of nodes managing hundreds of gigabytes of state (see Section 5). The state snapshotting mechanism is coordinated and pipelined, similarly to the classical Chandy-Lamport's protocol [27]. However, it is fine-tailored for weakly connected dataflow graphs and superimposes the acquisition of consistent snapshots without heavily impacting throughput. More importantly, snapshots are compacted, limited to minimal computational state with the exception of cyclic dataflow graphs where the partial inclusion of records in-transit is necessary. In situations where relaxed processing guarantees are acceptable (i.e. at-least once processing guarantees) a totally asynchronous version of the protocol can be selected on-demand. Moreover, state and output that can be accessed from outside the system (e.g. queryable state, pipeline output) is provided under different isolation levels (read committed or uncommitted) in order to satisfy the required trade off between consistency and latency. This paper is the first principled description of the techniques that are implemented in Apache Flink for state management. The main goal of this work is to accurately describe these techniques and their significance[1]. To summarize, this paper's contributions:

[1]Most authors have been involved in the conception and implementation of these core techniques. Yet, the full credit for the evolution of Flink's ecosystem goes to the Apache Flink community, currently having more than 250 contributors.

- We provide a complete end-to-end design for continuous stateful processing, from the conceptual view of state in the programming model to its physical counterpart implemented in various backends.
- We show how to naturally pipeline consistent snapshots in weakly connected dataflow graphs and capture minimal state, skipping in-transit records when possible, without impacting general system progress.
- We demonstrate how snapshots can be utilized for a large variety of operational needs beyond failure recovery such as software patches, testing, system upgrades and exactly-once delivery.
- We encapsulate different processing guarantees and isolation levels for externally accessing partitioned operator state and output, using snapshots.
- We describe large-scale pipeline deployments that operate 24/7 in production and rely heavily on stateful processing coupled with runtime metrics and performance insights.

The rest of the paper is organized as follows: Section 2 gives an overview of the Apache Flink stack and the basic principles behind distributed snapshots and guarantees for dataflow execution graphs. In Section 3 we describe the core state management mechanisms of Flink, namely its stateful programming abstractions, the snapshotting protocol and its practical usages. Section 4 summarizes further implementation concerns such as backend support, concurrent snapshots, the ability to query application state as well as end-to-end guarantees. Finally, Section 5 describes existing large-scale deployments and discusses metrics related to Flink's snapshots, followed by related work in Section 6, and our conclusions coupled with future work and acknowledgements summarized in Section 7.

2. PRELIMINARIES

2.1 The Apache Flink System

The Apache Flink system [7] is an open-source project that provides a full software stack for programming, compiling and running distributed continuous data processing pipelines (Figure 1(a)). Pipelines can be written as a series of data-centric transformations expressed in a fluid, functional programming API (in Scala, Java or Python) inspired by Flume Java[25], Dryad LINQ[45], Naiad[42] and based in its majority on Google's Dataflow Model [19] (i.e., identical windowing semantics and out-of-order processing logic). At the core of the model there are two basic abstract data types, the *DataSet* and *DataStream* representations which target bounded and unbounded datasets respectively. Computation declared using the available higher-level domain-specific libraries such as the SQL and Machine Learning (ML) packages, translates into a logical pipeline using these core representations. A major distinctive trait of the Flink programming model compared to state of the art is the capability to declare local or partitioned, persistent application state within continuous user-defined transformations through managed data collections with diverse properties (append-only, mutable, etc.). Flink's runtime ensures that consistency is guaranteed for any managed state declared despite potential partial failures or reconfigurations, such as updates to the application code or changes in execution parallelism.

Logical pipeline representations are optimised at the client and shipped to Flink's runtime, a distributed, continuous dataflow execution environment. The runtime derives a physical deployment of

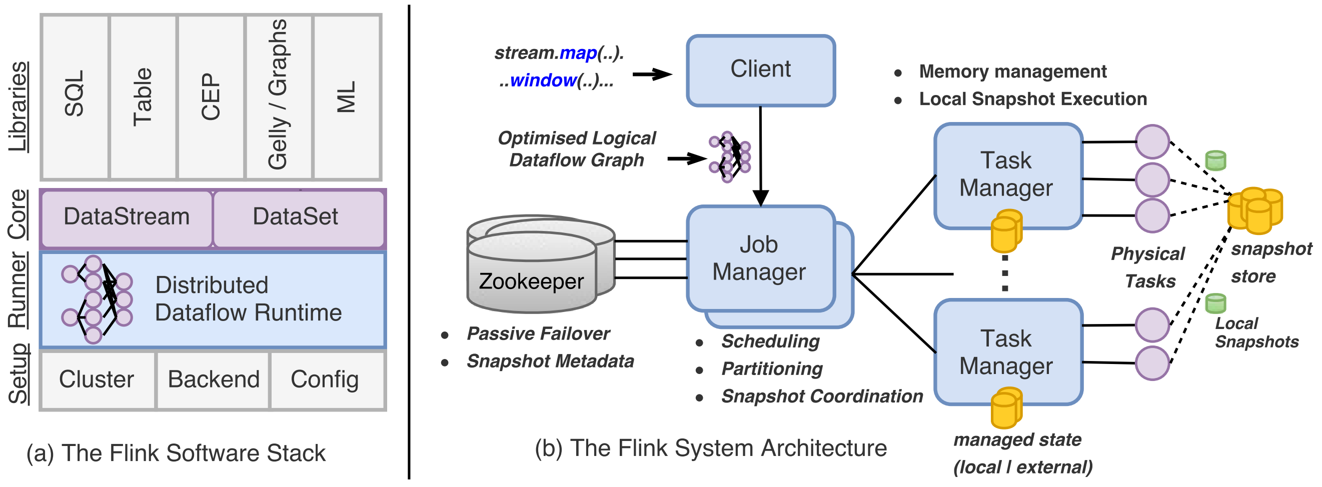


Figure 1: An Overview of the Apache Flink System Model and Architecture.

tasks and manages their continuous execution as depicted in Figure 1(b). As with most distributed data processing systems, Flink's runtime consists of a *JobManager*, a master process that holds the metadata of active pipelines and coordinates execution by communicating with worker processes, the TaskManagers. Communication between the the JobManager and TaskManagers respects an asynchronous RPC-based communication protocol, consisting of periodic status updates (heartbeats) to the JobManager and scheduling requests back to the TaskManagers. In contrast to batch-centric job management [47] which prioritizes reconfiguration and coordination, Flink employs a schedule-once, long-running allocation of tasks. However, the system is flexible to reconfigure pipelines to more or less workers and re-allocate application state on-demand. This approach minimizes management overhead while still allowing for further adaptation to hardware or software changes or partial failures that can potentially occur. Finally, pipeline deployments in Flink are highly available, thus, tolerating even master failures via leader election and passive failover in Zookeeper. All underlying mechanisms for state partitioning, snapshotting and maintenance are the main focus and covered thoroughly in this paper.

2.2 The Global Snapshotting Problem

Distributed systems are typically designed to hide concerns related to their distributed nature from the user, offering the view of a single entity. For a distributed compute system like Flink we often have to reason about the state of a pipeline in production at any time during its execution. Referring to the complete distributed state of a computation as an atomic unit, is essential to correctly rollback its full execution to the point in time when that global state was captured. This is crucial when reconfiguration is required or a partial failure caused a violation of the correct execution of the pipeline. Generally, this approach is also known as rollback recovery [33].

Distributed snapshotting [27] protocols enable rollback recovery by producing a correct, complete state replica of a distributed execution which can be used to restore the system to an earlier point in time. In principle, a distributed system is a set of processes connected via data channels, abstractly represented as a directed graph of nodes and edges respectively. At any time during continuous system execution the complete state is reflected in the nodes and edges of that graph (i.e., internal state of processes and in-transit events). A consistent snapshot should capture the complete state while respecting causal execution dependencies so that no computational state or data are lost.

Existing snapshotting protocols are tailored to specific types of graphs and vary in terms of complexity and performance. For example, Chandy and Lamport's original protocol [27] is designed for strongly connected directed graphs (there exist a path between any two processes) and it is transparently pipelined together with the normal execution of the system through the use of special markers, without affecting its operation. On the other hand, the same approach relies on aggressively logging any records that are in transit in the duration of the protocol and it is incapable of terminating on weakly connected graphs.

Weakly connected graphs are inherently relevant to distributed dataflow processing systems [42, 38, 18, 25, 24]. Data records are typically inserted to the system through special *source* vertices while exiting through special *sink* vertices and cycles can optionally exist. Proposed protocols for snapshotting weakly connected dataflow graphs such as Naiad's two-phase commit [42] and IBM Streams' multi-stage snapshotting halt the regular operation of the system to complete the snapshot and also end-up logging in-transit records unnecessarily. In our approach, described thoroughly in Section 3, we show how it is possible to naturally pipeline the snapshotting process in weakly connected graphs and capture minimal state, skipping in-transit records when possible, without halting the overall progress of the system.

3. CORE CONCEPTS AND MECHANISMS

3.1 System Model

Each processing pipeline in Flink is first defined as a logical directed graph $G = (\mathcal{T}, \mathcal{E})$ where $\mathcal{T}$ is a set of vertices representing compute tasks and $\mathcal{E}$ is a set of edges representing data subscriptions between tasks in $\mathcal{T}$ (Figure 2(a)). Data subscriptions can apply arbitrarily between tasks in $\mathcal{T}$, addressing the dependencies prescribed directly or indirectly via the programming model (e.g., forward, shuffle and hash partitioning). A task $t \in \mathcal{T}$ can encapsulate the logic of a single operation (e.g., `map`, `filter`, `fold`, `window`). However, standard logical dataflow optimisations such as fusion [36, 25] are also applied in an intermediate step allowing multiple operators to share the same task in $\mathcal{T}$ (Figure 2(b)). Each logical graph is directly mapped to a physical, distributed graph G^* upon deployment or rescaling (Figure 2(c)). In the rest of this section we are going to introduce the concept of managed state in Apache Flink, followed by physical state partitioning and a description of how state and data are being allocated to tasks.

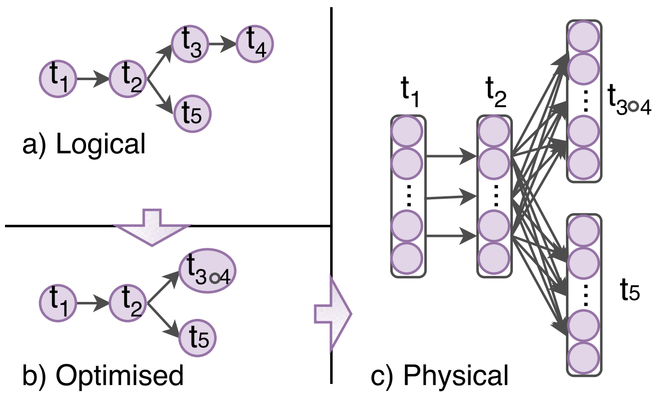


Figure 2: Dataflow Graph Representation Examples.

3.1.1 Managed State

Each stream operation in Flink can declare its own state and update it continuously in order to maintain a summary of the data seen so far. State is a main building block of a pipeline as it encapsulates, at any time, the full status of the computation. There are conceptually two scopes upon which managed state operates. For purely data-parallel stream operations such as a per-key average, the computation, its state and associated streams can be logically scoped and executed independently for each key. This is similar to how a relational `GROUP_BY` projects rows of the same key to the same set to compute grouped aggregates. We refer to this state as `Keyed-State`. For local per-task computation such as a partial machine learning training model, state can be declared in the level of a parallel physical dataflow task, known as `Operator-State`. Both `Keyed-State` and `Operator-State` are transparently partitioned and managed by the runtime of the system. More importantly, the system can guarantee that update operations on managed state will be reflected exactly-once with respect to the input streams. In Section 4 we explain in detail how the file system facilitates efficient external state persistence of different state types despite the local view exposed to the programmer. Below, we briefly explain how managed state can be declared and the basic intuition for each of the state types.

`Keyed-State`: Any data-parallel stream computation can be mapped to a user-defined key space and as a result any associated state will also be scoped together with the computation. Typically, data-stream records arrive to the system with some domain-specific key such as a user-session identifier, a device address or a geographical location. In the most general case, Flink allows for a user to map any record from its schema domain $\mathcal{S}$ to a given key space $\mathcal{K}$ via the `keyby` $: \mathcal{S} \rightarrow \mathcal{K}$ operation supported by the `DataStream` abstract type. Under key scope, state can be allocated dynamically within a user-defined function by using special collections that the model exposes through the API and vary depending on the nature of the state. For append-only state per key (e.g. for storing a pattern sequence or a window) there is a `ListState` collection supporting an `add` operation. If the state is otherwise a value that mutates during the application logic, there is a `ValueState` type supporting an `update` operation. Other basic state types such as `ReduceState` further allow for one or two-step, on-the-fly, distributive function aggregations on managed state. Finally, the `MapState` state type can support `put` and `get` key-value operations and is preferred over having a custom map declared as a `ValueState` since it avoids a full map deserialization to perform single key lookups.

`Operator-State`: Another scope upon which state and computation can be declared is within the granularity of each parallel instance of a task (task-parallel). `Operator-State` is used when part of a computation is only relevant to each physical stream partition, or simply when state cannot be scoped by a key. A Kafka ingesting source operator instance for example that has to keep offsets to respective partitions in Kafka [39] is using this scope. `Operator-State` adheres to a *redistribution pattern* that allows breaking state into finer-grained units when possible, allowing the system to redistribute state when changing the parallelism of the operator (scale in/out).

3.1.2 State Partitioning and Allocation

Physical Representation: The mapping of a logical graph G to $G^* = \{\mathcal{T}^*, \mathcal{E}^*\}$, the physical, distributed execution graph (Figure 2(c)) occurs when a pipeline is deployed, on its initial run or upon reconfiguration (e.g., for scale-out). During that stage each logical task $t \in \mathcal{T}$ is mapped to a number of physical tasks $t^1, t^2, \ldots, t^\pi \in \mathcal{T}^*$, each of which gets deployed to available containers throughout a cluster (e.g., using YARN [43] or Mesos [35]) up to the decided degree of parallelism $\pi \in \mathbb{N}^+$.

Key-Groups: For tasks that have declared managed keyed state, it is important to consistently allocate data stream partitions or re-allocate in the case of reconfiguration. For flexibility, Flink decouples key-space partitioning and state allocation similarly to Dynamo[32]. Consider a user-defined key space $\mathcal{K}$. The runtime maps keys to an intermediate circular hash space of "key-groups" : $\mathcal{K}^* \subset \mathbb{N}^+$ given a maximum parallelism π-max and a hash function h as such:
$\mathcal{K}^* = \{h(k) \bmod \pi\text{-max} \mid k \in \mathcal{K}, \pi\text{-max} \in \mathbb{N}^+, h : \mathcal{K} \rightarrow \mathbb{N}^+\}$
This mapping ensures that a single parallel physical task will handle all states within each assigned group, making a key-group the atomic unit for re-allocation. The intuition behind key-groups lies in the trade-off between reconfiguration time (I/O during state scans) and metadata needed to re-allocate state (included within snapshots). On one extreme each parallel task could scan the whole state (often remotely) to retrieve the values of all keys assigned to it. This yields significant amounts of unnecessary I/O. On the opposite extreme, snapshots could contain references to every single key-value and each task could selectively access its assigned keyed states. However, this approach increases indexing costs (proportional to num. of keys) and communication overhead for multiple remote state reads, thus, not benefiting by coarse-grained state reads. Key-groups offer a substantial compromise: reads are only limited to data that is required and key-groups are typically large enough for coarse grained sequential reading (if π-max is set appropriately low). In the uncommon case where $|\mathcal{K}| < \pi$-max it is possible that some task instances simply receive no state.

State Re-Allocation: To re-assign state, we employ an equal-sized key-group range allocation. For π parallel instances, each instance $t^i \in \mathcal{T}^*, 0 \leq i \leq \pi$ receives a range of key-groups from $\lceil i \cdot \frac{\pi\text{-max}}{\pi} \rceil$ to $\lfloor (i+1) \cdot \frac{\pi\text{-max}}{\pi} \rfloor$. Seeks are costly, especially in distributed file systems. Nevertheless, by assigning contiguous key-groups we eliminate unnecessary seeks and read congestion, yielding low latency upon re-allocation. `Operator-State` entries, which cannot be scoped by a key, are persisted sequentially (combining potential finer-grained atomic states defined across tasks), per operator, within snapshots and re-assigned based on their *redistribution pattern*, e.g., in round-robin or by broadcasting the union of all state entries to all operator instances.

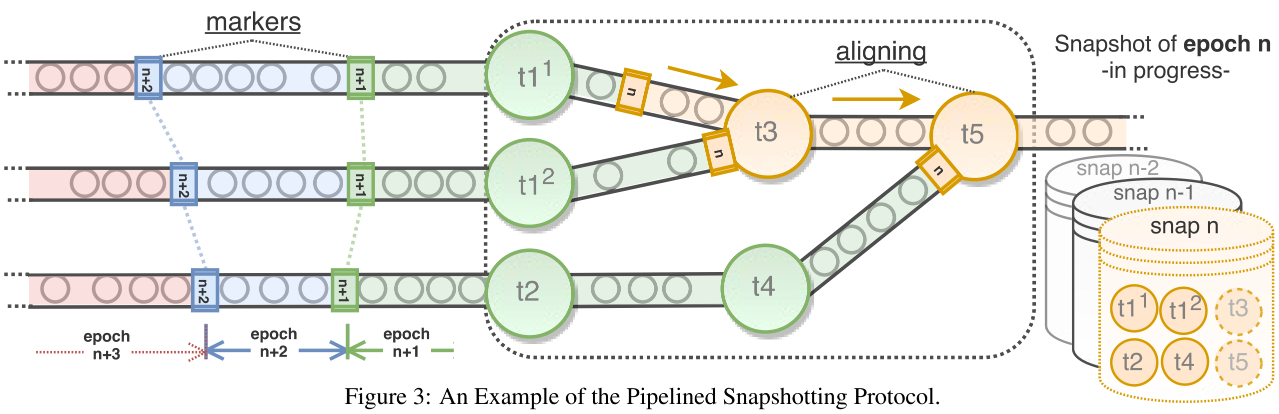


Figure 3: An Example of the Pipelined Snapshotting Protocol.

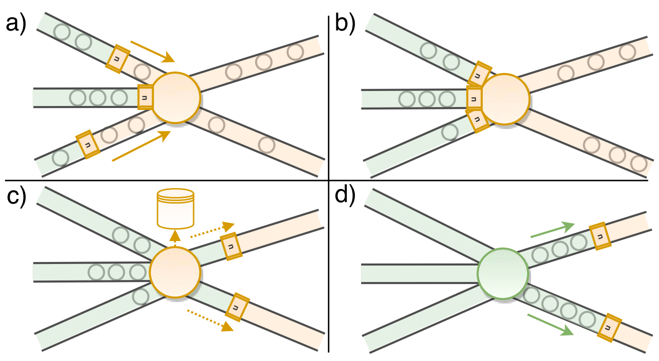


Figure 4: Alignment and Snapshotting Highlights.

3.2 Pipelined Consistent Snapshots

Flink's snapshotting protocol provides a uniform way to capture the complete state of a pipeline and roll it back whenever that is required. We will first explain its intuition followed by a more formal definition of the assumptions and description of the protocol for directed acyclic and cyclic graphs respectively.

3.2.1 *Approach Intuition*

A continuous stream execution is conceptually divided into logical periods that "cut" a distributed data stream into consecutive finite sets of records (Figure 3), which we call *epochs*. An *epoch* can be triggered on-the-fly, periodically by the system or on-demand by the user and is decoupled from any application logic (e.g., windowing constrains). A snapshot of the computation at *epoch* n refers to a copy of the internal state of each task $t \in \mathcal{T}^*$ after the system fully ingests every input record from the beginning of the computation (*epoch* 0) up to and including *epoch* n. In case of a failure during or before a snapshot of *epoch* n is acquired we can simply revert the global state of the distributed dataflow graph to a previous *epoch* (e.g., $n-1$). A discrete approach to snapshotting would be to let the system fully ingest *epoch* n, log the internal state of each task $t \in \mathcal{T}^*$ and then proceed with *epoch* $n+1$ (similarly to micro-batching [47]). However, this approach raises latency and underutilization costs related to the coordination of a discrete execution which can be hard to amortize. Furthermore, other protocols either disrupt normal execution [42, 38] or are incapable of supporting typical weakly connected graphs [27].

Instead, Flink's snapshotting protocol pipelines progressively the partial acquisition of task states to eventually acquire a complete snapshot, respecting *epochs*, while running concurrently alongside normal operation. Special markers are injected in each data stream partition at the dataflow sources, coordinated by the runtime and get disseminated throughout the dataflow graph as depicted in Figure 3. Markers signal distributed tasks of new epochs and thus aid to establish the appropriate moment to snapshot each local state and proceed with further processing promptly. Tasks with multiple inputs execute an *alignment* phase (e.g., tasks $t3$ and $t5$ in Figure 3) upon which they prioritize exclusively inputs from pending *epochs*. Alignment is decentralized and eliminates the need to fully consume an epoch or log records in transit before snapshotting. As we explain in more detail further, cyclic graphs require partial channel logging only limited to each dataflow cycle. The snapshotting protocol is coordinated centrally by the *JobManager* and each invocation eventually completes or gets aborted (e.g., when a failure occurs). In either case the overall dataflow computation can always progress without interruptions and consecutive snapshots will eventually complete.

3.2.2 *Main Assumptions*

The protocol assumes a fail-recovery, deterministic process model [33] where a partial process failure can be masked by redeployment and restoration of prior operational states. In detail, our protocol builds on the following assumptions:

I: Input data streams are durably logged and indexed externally allowing dataflow sources to re-consume their input, upon recovery, from a specific logical time (offset) by restoring their state. This functionality is typically provided by file systems and message queues such as Apache Kafka [39].

II: Directional data channels between tasks are reliable, respect FIFO delivery and can be blocked or unblocked. When a channel is blocked, in-transit messages are internally buffered (and possibly spilled to disk) and can be delivered on that end once it unblocks.

III: Tasks can trigger a `block` or `unblock` operation on their input data channels and a `send` operation (records or control messages) on their output channels.

3.2.3 *Directed Acyclic Graphs*

Let us consider only directed acyclic graphs (DAGs) for now. The protocol gets initiated at the source tasks of the dataflow by the `TaskManager`, however, for simplicity we assume here that the logic gets initiated upon receiving a special marker event in each and every task (sources would "receive" that first through a Nil channel). Algorithm 1 summarizes the snapshot alignment and pipelining protocol that executes when a marker is received. Mind that markers and records are handled sequentially by the same underlying thread that also invokes user-defined operators.

Algorithm 1: Snapshot Alignment

```
$inputs \leftarrow$ configured_inputs;
$outputs \leftarrow$ configured_outputs;
$blocked \leftarrow \emptyset$ ;
Upon $\langle marker \rangle$ from $in \in inputs$
    if $in \neq Nil$ then
        $blocked \leftarrow blocked \cup in$;
        $in$.block();
    if $blocked = inputs$ then
        foreach $out \in outputs$ do
            $out$.send($\langle marker \rangle$);
        triggerSnapshot();
        foreach $in \in inputs$ do
            $in$.unblock();
        $blocked \leftarrow \emptyset$ ;
```

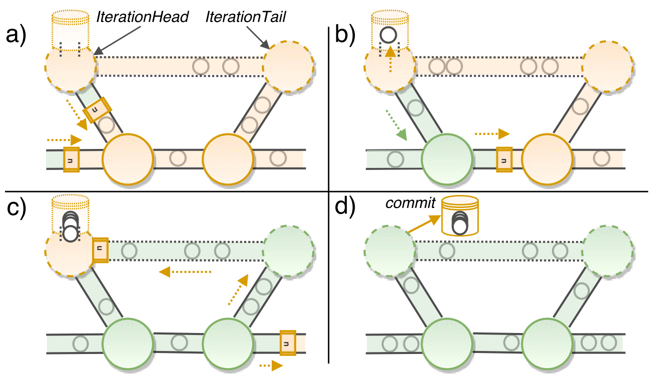


Figure 5: Cycle Snapshotting Highlights.

Alignment: Figure 4 vizualizes the steps prior to and during snapshotting in more detail. When a task receives a snapshot marker on one of its inputs, it blocks that channel since all computation associated with the current epoch has to finish before continuing further (Figure 4(a)). The blocking operation might result into spilling of in-transit records within that channel to disk, if allocated memory for network buffers reaches its limit. Once markers have been received in all inputs (Figure 4(b)) the task can further notify downstream tasks while also proceeding with snapshotting. Markers are first broadcasted forward and then local snapshotting is triggered, both of which can progress concurrently without sacrificing consistency (Figure 4(c)). Depending on the backend support, snapshots can be triggered and executed asynchronously by another thread, thus, minimizing their impact to the overall throughput. Once local snapshotting is initiated (Figure 4(d)) input channels are unblocked and regular operation continues to the next epoch. Overall, reliable FIFO data channels (Assumption II), combined with alignment guarantee that epoch-order is always respected.

Relaxing Consistency: It is possible, if a pipeline allows for relaxed consistency requirements, to disable alignment (i.e., no input blocking). Essentially, this means that a snapshot on epoch n will contain side-effects of input residing in epochs $\geq n$. Upon rollback records succeeding epochs n are going to be ingested again, resulting into multiple state updates. This type of processing guarantees, also known as *at-least-once processing*, can be enabled on Flink to trade-off consistency for practically zero latency impact if the application has no critical requirements.

3.2.4 *Dealing with Dataflow Cycles*

Dataflow graphs in Flink can also support cycles. Cycles are currently defined explicitly through Flink's programming API as asynchronous iterations, though bulk synchronous iterations (e.g., on stream windows) are also considered and can be supported in the future. Cyclic snapshotting is handled as a special case and implemented by system-specific, implicit tasks: an `IterationHead` and `IterationTail`. These tasks act as regular dataflow *source* and *sink* respectively, yet, they are collocated in the same physical instance to share an in-memory buffer and thus, implement loop-back streams transparently.

The default stream alignment logic presented (Algorithm 1) would result into an incomplete distributed snapshot if applied on cyclic graphs. That is due to the fact that records belonging to prior epochs could still remain indefinitely in-transit within a cycle even after a snapshot has been taken over. Thus, it is crucial to persist these records in the snapshot in order to get a complete picture of the correct distributed execution [27, 33]. Alternative approaches to this problem consider a "flushing" phase which enforces the inclusion of all the in-transit records to the internal state of each task [38], however, we argue that this problem is only relevant to the state of a cycle. Thus, we execute a similar special logging protocol (Algorithm 2) that runs solely within the `IterationHead` instances of each cycle.

As described in detail in Algorithm 2 and also visualized in Figure 5, `IterationHead` tasks receive a special marker from the runtime signifying each epoch, same as the sources of the dataflow graph. At that instance, they disseminate markers further within a cycle and start logging in their own managed state all in-transit events that exist within a cycle in that respective partition (Figure 5(a)). Once the marker of the snapshot is received back through their respective collocated `IterationTail` (Figure 5(c)) they trigger a snapshot of that log containing a complete backup of all transmitted records in that epoch (Figure 5(d)). Again, FIFO channels and alignment executed by the rest of the tasks within a cycle (Figure 5(b)) ensures that no records from succeeding epochs will transit prior to the marker. This special logic also restricts channel logging to cycles and does not enforce anything other than task states to be included in the snapshot for the rest of the graph.

3.3 Usages and Consistent Rollback

Consistent snapshots, described previously in Section 3.2 form the basis for a variety of operations using the Apache Flink system. Periodic snapshots are automatically triggered by Flink's runtime as a form of per-job "checkpoint" for the purposes of consistent fail recovery whenever partial failures occur. However, the usages of snapshots go beyond fault tolerance needs. For a system that is widely deployed in a cloud infrastructure, having the ability to scale resources in or out and lease containers is nowadays a necessity. In principle, failover and re-scaling are two operations that share the same underlying need for consistent reconfiguration support [24]. In this section we describe in more detail the operational benefits that distributed snapshots make feasible, as well as the rollback reconfiguration schemes that are currently supported.

3.3.1 *Snapshot Usages*

Flink's snapshots define consistent points across parallel operators and thus, are suitable points for reconfiguration. The metadata of a snapshot contains all necessary information required to retrieve the complete pipeline state from an associated durable storage backend such as references and indexes to operator state partitions (i.e., key-groups) and upstream logs (in

Algorithm 2: Snapshotting in Cycles

```
$outputs \leftarrow$ configured_outputs;
$isLogging \leftarrow false$ ;
$log \leftarrow \emptyset$ ;
Upon $\langle marker \rangle$
    if $isLogging$ then
        triggerSnapshot($log$) ;
        $log \leftarrow \emptyset$ ;
        $isLogging \leftarrow false$ ;
    else
        $isLogging \leftarrow true$ ;
        foreach $out \in outputs$ do
            $out$.send($\langle marker \rangle$);
Upon $record$
    if $isLogging$ then
        $log \leftarrow log \cup record$;
    foreach $out \in outputs$ do
        $out$.send($record$);
```

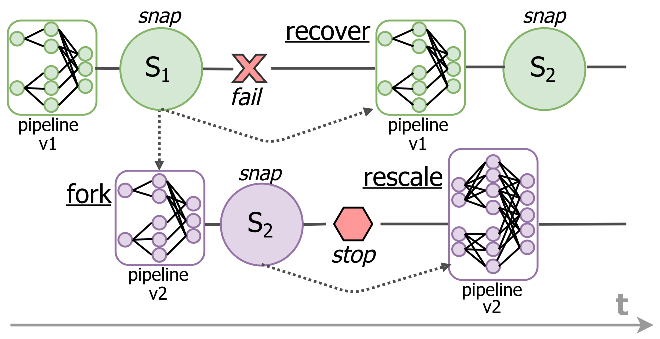


Figure 6: Snapshot usage examples.

case of cyclic graphs). Common causes of reconfiguration are: 1) application logic updates (e.g., software patches) of already running jobs by replacing operators accessing the same snapshotted state or adding new state entries instead and 2) application versioning, allowing forking running pipelines (an example depicted in Figure 6). In practice, reconfiguration follows a `checkpoint-stop-modify-restore` cycle, initiated externally by the user. However, it is also possible to be triggered topologically by attaching reconfiguration commands to snapshot markers which reconfigure operators (replace code) at the point where the snapshot is taken.

3.3.2 Consistent State Rollback

Rollback recovery gets initiated upon a task failure or when there is a need to rescale. Typically, the latest snapshot is used to restart the application from the beginning of the latest committed epoch. Depending on the rollback cause, different recovery schemes are employed. For example. during a full restart or rescaling, all tasks are being redeployed, while after a failure only the tasks belonging to the affected connected component (of the execution graph) are reconfigured. In essence, known incremental recovery techniques from micro-batch processing [47] are orthogonal to this approach and can also be employed. A snapshot epoch acts as synchronization point, similarly to a micro-batch or an input-split. On recovery, new task instances are being scheduled and, upon initialization, retrieve their allocated shard of state. In the case of `Iteration Head` recovery, all records logged during the snapshot are recovered and flushed to output channels prior to their regular record-forwarding logic. Eventually, all states within the pipeline are progressively retrieved and applied to reflect an exact, valid distributed execution at the restored epoch.

It is additionally required for all data sources to rollback input from the epoch where the snapshot occurred. Flink's data sources provide this functionality out-of-the-box by maintaining offsets to the latest record processed prior to an epoch from external logging systems. Upon recovery the aggregate state of those sources reflects the exact distributed ingestion progress made prior to the recovered epoch. This approach assumes that external logging systems, that sources communicate with, index and sequence data across partitions in a durable manner (e.g., Kafka, Kinesis, PubSub and Distributed File Systems).

4. IMPLEMENTATION AND USAGE

With explicit managed state and consistent snapshotting at its core, Flink maintains a rich ecosystem of backends, connectors and other services that interplay seamlessly and benefit from Flink's core mechanism. In this section, we summarize how most of these subsystems build on-top of Flink's core architecture, while expanding it with asynchronous communication, delivery guarantees and external state querying support.

4.1 State Backend Support

Managed state consistency is coordinated by Flink's snapshotting algorithm (Section 3.2), however, it is the responsibility of the state backends to handle state access and snapshotting for their respective state partitions. We distinguish two main classes of state backends: 1) *Local State Backends*, where access to state is kept and controlled in the same physical node, and 2) *External State Backends*, where state access is internally coordinated with an external system such as a database or key/value store.

Local state backends maintain all state in local memory or out-of-core, within an embedded key-value database such as RocksDB [13]. Out-of-core access is preferred in production deployments as in that case state size is only limited by the quota of the local file system allocated to each node. When a snapshot operation is triggered, a copy of the current local states is written to a durable storage layer (e.g., a distributed file system directory). The local state backends can support asynchronous and incremental snapshotting (Section 4.2) and yield considerable read/write performance as they exploit data locality while eliminating the need for distributed or transactional coordination.

External state backends coordinate state access with external database management systems. There are two possible variations, depending on the properties of the external database: 1) **Non-MVCC** (Multi-Version Concurrency Control) databases, can be supported by maintaining within each task a write-ahead-log (WAL) of pending state changes per epoch. The WAL can be committed to the database once an epoch has been completed. In more detail, each snapshot is one distributed bulk 2-phase commit transaction: during a checkpoint, changes are put into the transaction log (WAL) and pre-committed when `triggerSnapshot()` is invoked. Once the global snapshot is complete, then pre-committed states are fully-committed by the JobManager in one atomic transaction. This approach is feasible even when the database does not expose the necessary program hooks to log, pre-commit, and fully commit. 2) **MVCC-enabled** databases allow for committing state across multiple database versions. This can integrate with Flink's snapshotting mechanism by associating each state update with the undergoing epoch. Once a snapshot is fully committed the version is atomically incremented. Likewise, a failed snapshot epoch decrements the current version. A general advantage of external state backends is that rollback recovery does not require any I/O to retrieve and load state from snapshots (contrary to local state backends). This benefit becomes particularly impactful when state is very large, by avoiding any network I/O upon reconfiguration, thus, making it a suitable choice under low latency requirements. Finally, another benefit that comes out-of-the-box with all external backends is support for incremental snapshotting, since, by definition, only changes are committed externally.

4.2 Asynchronous and Incremental Snapshots

One of the assets of the pipelined snapshotting protocol presented in Section 3.2 is that it only governs "when" but not "how" snapshots are internally executed. The `triggerSnapshot()` call by each task is expected to create an identical copy of the current state of that task. However, the copy is not required to be a physical copy, it can be a *logical* snapshot that is *lazily* materialized by a concurrent thread. This type of operation is supported by many *copy-on-write* data structures. More concretely, Flink's local backends allow for asynchronous snapshots as such:

The out-of-core state backend based on RocksDB [13] exploits the LSM (log-structured merge) tree, internal representation of data in RocksDB. Updates are not made in-place, but are appended and compacted asynchronously. Upon taking a snapshot, the synchronous `triggerSnapshot()` call simply marks the current version, which prevents all state as of that version to be overwritten during compactions. The operator can then continue processing and make modifications to the state. An asynchronous thread iterates over the marked version, materializes it to the snapshot store, and finally releases the snapshot so that future compactions can overwrite that state. Furthermore, the LSM-based data structure also lends itself to incremental snapshots, which write only parts to the snapshot store that changed since the previous snapshots.

Flink's in-memory local state backend implementation is based on hash tables that employ chain hashing. During a snapshot, it copies the current table array synchronously and then starts the external materialization of the snapshot, in a background thread. The operator's regular stream processing thread lazily copies the state entries and overflow chains upon modification, if the materialization thread still holds onto the snapshot. Incremental snapshots for the in-memory local backend are possible and conceptually trivial (using delta maps), yet not implemented at the current point.

Finally, another feature that is provided to tasks as an optional asynchronous subscription-based mechanism is to trigger notifications about completed snapshots back at the tasks that request them. This is especially useful for garbage collection, discarding write ahead logs or for coordinating exactly-once delivery sinks as we explain further in Section 4.4.

4.3 Queryable State

A recent addition among Flink's state management features is the ability to directly query managed state from outside the system. External systems can access Flink's keyed-state in a similar way as that of a key/value store, providing read-only access to the latest values computed by the stream processor. This feature is motivated by two observations. First, it is required by many applications to grant ad-hoc access to the application state for faster insights. Secondly, eager publishing of state to external systems frequently becomes a bottleneck in the application as remote writes to the external systems cannot keep up with the performance of Flink's local state on high-throughput streams.

Queryable state allows for any declared managed state within a pipeline, currently scoped by key, to be accessed outside the system for asynchronous reading, through a subscription-based API. First, managed state that allows for query access is declared in the original application. Upon state declaration, introduced in Section 3.1.1, it is possible to allow access from external queries by simply setting a flag in the descriptor that is used to create the actual state, having an assigned unique name for this specific state to be accessed, as such:

```
//stream processing application logic
val descriptor: ValueStateDescriptor[MySchema] = ...
descriptor.setQueryable("myKV")
...
val mutState: ValueState[MySchema] =
     ctx.getState(descriptor)
```

Upon deployment, a state registry service gets initiated and runs concurrently with the task that holds write access to that state. A client that wishes to read the state for a specific key can, at any time, submit an asynchronous query (obtaining a `future`) to that service, specifying the job id, registered state name and key, as shown below:

```
//client logic
val client = QueryableStateClient(cfg);
var readState: Future[_] = client.getKVState(job,
     "myKV", key);
```

The current implementation of queryable state supports point lookups of values by key. The query client asks the Flink master (JobManager) for the location of the operator instance holding the state partition for the queried key. The client then sends a request to the respective TaskManager, which retrieves the value that is currently held for that key from the state backend. From a traditional database isolation-level viewpoint, the queries access *uncommitted state*, thus following the *read-uncommitted* isolation level. As future work, we plan to add *read-committed* isolation support by letting TaskManagers hold onto the state of committed snapshots, and use that state to answer adhoc queries.

4.4 Exactly-Once Delivery Sinks

So far we have considered consistency guarantees associated with the internal state of the system. However, it is most often important to offer guarantees regarding the side effects that a pipeline leaves to the outside world, whether that is a distributed database, file system or message queue. A pipeline interfaces with the outside world mainly via its dataflow sinks. Therefore, it is crucial that sinks can offer exactly-once delivery guarantees. The feasibility of achieving "read-committed" isolation guarantees to external writes depends on the properties of the system upon which sinks commit output and typically comes at a higher latency cost (this can, at-times, violate strong SLAs on latency). A pipeline can always be halted between snapshots after a failure or an urgent reconfiguration request and both input and state can be rolled back consistently, as it was described in Section 3. However, the same cannot always be guaranteed about the output. If sinks are connected, for example, to a printer that instantly flushes data on paper, a rollback would possibly print the same or alternating text twice. Flink's programming model is equipped with two main types of sinks that facilitate exactly-once delivery and build on the snapshotting mechanism: 1) Idempotent and 2) Transactional Sinks.

Idempotent Sinks: Idempotency is a property used extensively by many systems at the presence of failures in order to encourage repeatability and alleviate bookkeeping efforts and complex

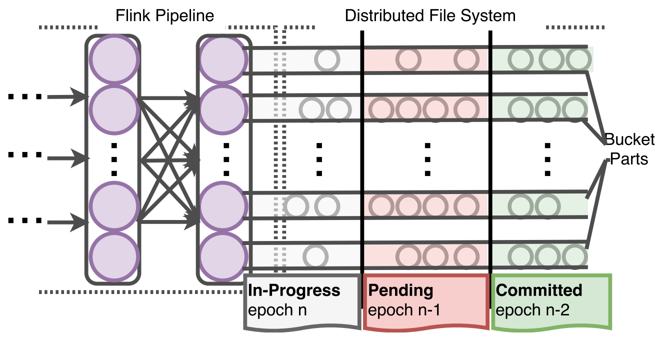


Figure 7: A visualization of Bucketing File Sinks.

transactions to offer delivery guarantees[3, 18]. In some cases deterministic pipeline logic and idempotency can already be offered by the stream application (e.g., not involving stream interleaving or other forms of non-determinism). For those cases, sinks need to take no further action to achieve exactly-once delivery guarantees than eagerly publishing their output. However, if the end-to-end application logic cannot tolerate uncommitted reads a write-ahead-log (WAL) needs to be coordinated with asynchronous snapshot notifications to publish changes to the external system once a snapshot for an epoch is complete. Flink's Cassandra database sink maintains a WAL of prepared query statements as part of its state. Once an asynchronous notification (Section 4.2) arrives for an epoch, the database sink commits all pending writes to the database. The idempotency property of the queries guarantees that even if publishing is disrupted, output will be consistent by simply re-committing the same queries and thus, eventually, leaving the same side effects upon subsequent system reconfigurations.

Transactional Sinks: When idempotency cannot be guaranteed, committing output to external systems (e.g., File Systems, DBMSs) has to be made in a coordinated, transactional way. We identify two variants of this approach:

1) Maintaining a WAL at the sinks and eventually publishing it when a snapshot notification arrives locally using a 2-phase commit with the external system. An example of this approach is Flink's transactional SQL sink which includes in the sink's state the WAL to be committed. Eventually, all changes are atomically committed across partitions after the first successful snapshot.

2) Alternatively, another approach that works well in conjunction with distributed file systems is to append uncommited output eagerly by every sink. On snapshot, output partitions can be pre-committed (e.g., by flagging a file directory) and eventually committed once the global snapshotting process is complete and notified across partitions. Upon failure changes can be rolled back. An example of this approach is Flink's *bucketing* file sink (depicted in Figure 7) which eagerly appends stream output within uncommitted distributed file directories which group (or "bucket") file partitions by time-period. After a pre-configured inactivity time-period `in-progress` directories become `pending` and are ready to be committed. The Bucketing File Sink integrates with Flink's snapshotting algorithm and associates epochs with buckets. Once a file bucket is under `pending` mode and an asynchronous notification for an associated epoch has been received, it can be moved to a `committed` state via an atomic `rename` operation. Potential disruptions between epochs resolve into a `truncate` (Posix) operation, which is currently supported by major distributed file systems and conveniently reverses append operations to the right epoch.

4.5 High Availability & Reconfiguration

All metadata associated with the active state of long-running pipelines is kept within the `JobManager` node. This makes the importance of this central node quite high for the general functionality of the system but also a single-point-of-failure. A `JobManager` failure would halt the coordination of the snapshotting protocol, the collection and management of snapshot metadata as well as making further job deployments and rescaling requests unavailable, thus, eliminating the purpose of any fault tolerance mechanism. To deal with such a critical failure we employ a passive-standby high availability scheme where configured standby nodes can take undertake the coordination role of the failed master node via distributed leader election. When this mode is enabled, the coordination of vital decisions such as job deployment and scheduling are undertaken via a distributed decision protocol which logs committed operations atomically (currently utilizing a `Zookeeper` quorum executing the `zab` protocol [37]). This introduces some additional latency for such critical operations, however, non-critical decisions such as the persistence of the most recent snapshot metadata for a job, are asynchronously committed and logged. The worst scenario of a potential failure during non-critical commits would simply result into, a yet valid restoration of active pipeline state from consistent snapshots that correspond to earlier epochs that have been fully committed prior to the failure.

5. LARGE-SCALE DEPLOYMENTS

Flink is one of the most widespread open source systems for data stream processing, serving the data processing needs of companies ranging from small startups to large enterprises. It is offered commercially by several vendors, including data Artisans, Lightbend, Amazon AWS, and Google Cloud Platform. A number of companies have published case studies on how they use Flink for stateful stream processing at large scale in the form of blog posts or presentations at industrial conferences and trade shows. For example, Alibaba, the world's largest e-commerce retailer company, deploys Flink on over 1000 nodes to support a critical search service [2] and keep search results and recommendation as relevant to their active retail catalogue as possible. Uber, the world's largest privately held company at the time of writing is building a stream processing platform based on SQL and Flink, called AthenaX [1] to allow access to timely data to the data scientists in the company. Netflix is building a self-serve, scalable, fault-tolerant, multi-tenant Stream Processing as a Service platform leveraging Apache Flink with the goal to make the stream of more than 3 petabytes of event data per day available to their internal users [14]. Other use cases come from the banking sector [15], telecommunications [12] and others [2].

In the remainder of this section, we present live production metrics and insights related to Flink's state management mechanism on a use-case by King (King Digital Entertainment Limited), a leading mobile gaming provider with over 350 million monthly active users.

5.1 A Real-Time Analytics Platform

The Rule-Based Event Aggregator (RBEA) by King [4], is a reliable live service that is implemented on Apache Flink and used daily by data analysts and developers across the company. RBEA

[2]For the interested reader, the annual Flink Forward series of conferences contains much of these industry presentations [9] and a recent survey of the Flink community conducted by data Artisans [10] provides further statistics on how Flink is used within enterprises.

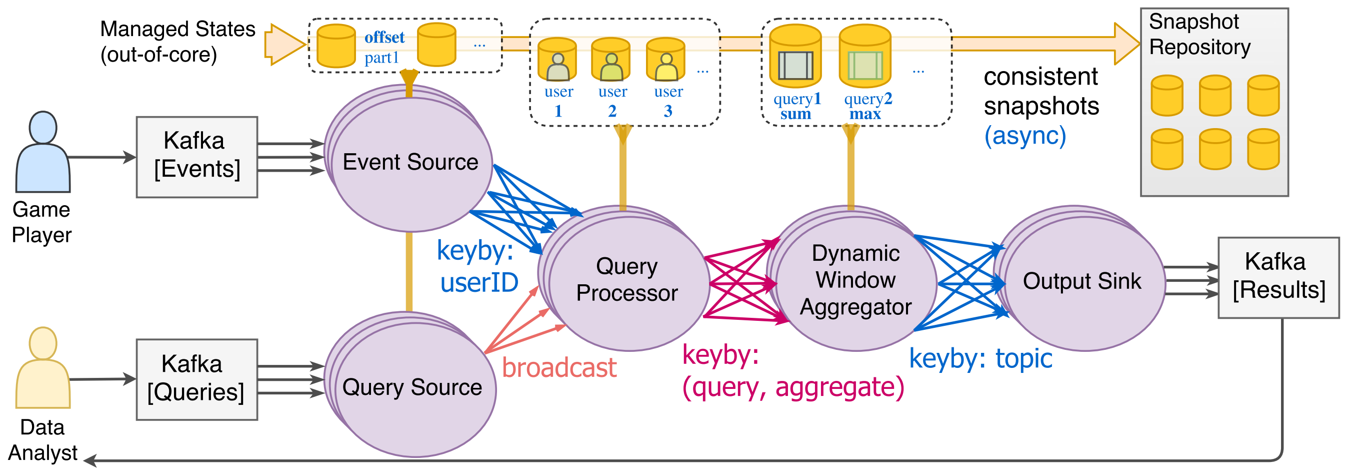


Figure 8: Overview of the Flink pipeline implementing an adhoc standing query execution service at King

showcases how Flink's stateful processing capabilities can be exploited to build a highly dynamic service that allows analysts to declare and run standing queries on large-scale mobile event streams. In essence, the service covers several fundamental needs of data analysts: 1) instant access to timely user data, 2) the ability to deploy declarative standing queries, 3) creation and manipulation of custom aggregation metrics, 4) a transparent, highly available, consistent execution, eliminating the need for technical expertise.

5.1.1 The RBEA Service Pipeline

Figure 8 depicts a simplified overview of the end-to-end Flink pipeline that implements the core of the service. There are two types of streams, ingested from Kafka: a) an `Event` stream originating from user actions in the games (over 30 billion events per day) such as `game_start/game_end` and b) a `Query` stream containing standing queries in the form of serialized scripts written by data analysts through RBEA's frontend in a provided DSL (using Groovy or Java). Standing queries in RBEA allow analysts to access user-specific data and event sequences as well as triggering special aggregation logic on sliding data windows.

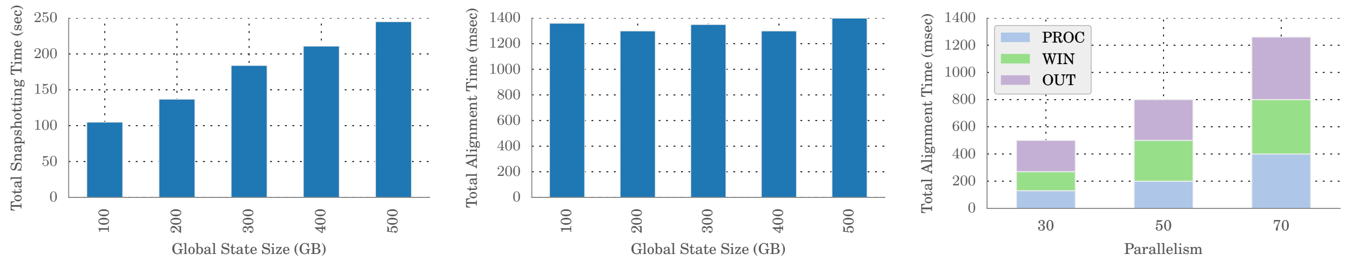
Standing queries are forwarded and executed inside `[Query Processor]` instances which hold managed state entries per user accumulated by any stateful processing logic. A "broadcast" data dependency is being used to submit each query to all instances of the `[Query Processor]` so it can be executed in parallel while game events are otherwise partitioned by their associated user ids to the same operator. Aggregation calls in RBEA's standing query DSL trigger output events from `[Query Processor]` operator which are subsequently consumed by the `[Dynamic Window Aggregator]`. This operator assigns the aggregator events to the current event-time window and also applies the actual aggregation logic. Aggregated values are sent to the `[Output sink]` operator which writes them directly to an external database or Kafka. Some details of the pipeline such as simple stateless filter or projection operators have been omitted to aid understanding as they don't affect state management.

5.1.2 Performance Metrics and Insights

The performance metrics presented here were gathered from live deployments of RBEA over weeks of its runtime in order to present insights and discuss the performance costs related to snapshotting, as well as the factors that can affect those costs in a production setting. The production jobs share resources on a YARN cluster with 18 physical machines with identical specification each having 32 CPU cores, 378 GB RAM with both SSD and HDD. All deployments of RBEA are currently using Flink (v.1.2.0) with local out-of-core RocksDB state backend (on SSD) which enables asynchronous snapshotting to HDFS (backed by HDD). The performance of Flink's state management layer, that we discuss bellow, has been evaluated to address two main questions: 1) What affects snapshotting latency?, and 2) How and when is normal execution impacted?

1) What affects snapshotting latency?
We extracted measurements from five different RBEA deployments with fixed parallelism $\pi = 70$ ranging from 100 to 500 GB of global state respectively (each processing data from a specific mobile game). Figure 9(a) depicts the overall time it takes to undertake a full snapshot asynchronously for different state sizes. Mind that this simply measures the time difference between the invocation of a snapshot (epoch marker injection) and the moment all operators notify back they have completed it through the asynchronous backend calls. As snapshots are asynchronously committed these latencies are not translated into execution impact costs, which makes alignment the sole factor of the snapshotting process that can affect runtime performance (through partial input blocking). Figure 9(b) shows the overall time RBEA task instances have spent in *alignment* mode, inducing an average delay of 1.3 seconds per full snapshot across all deployments. As expected, there are no indications that alignment times can be affected by the global state size. Given that state is asynchronously snapshotted, normal execution is also not affected by how much state is snapshotted.

2) How and when is normal execution impacted?
Alignment employs partial blocking on input channels of tasks and thus, more connections can introduce higher runtime latency overhead. Figure 9(c) shows the total times spent aligning per full snapshot in different RBEA deployments of fixed size (200GB) having varying parallelism. Evidently, the number of parallel subtasks π affects the alignment time. More concretely, the overall alignment time is proportional to two factors: 1) the number of shuffles chained across the pipeline (i.e., RBEA has $3\times$ `keyby` for the `PROCESSOR`, `WINDOW` and `OUTPUT` operators respectively), each of which introduces a form of alignment "stage" and 2) the parallelism of the tasks. Nevertheless, occasional latencies of such a low magnitude ($\sim$1sec) are hardly considered to be disruptive or breaking SLAs, especially in highly utilized clusters of such large-scale deployments where network spikes and CPU load can often cause more severe disruptions.



(a) Snapshot Duration vs Total Size [π:70, `state`:[100:500GB], `hosts`:18]

(b) Alignment Time vs Snapshot Size [π:70, `state`:[100:500GB], `hosts`:18]

(c) Alignment Time vs Parallelism [π : [30 : 70], `state`:200GB, `hosts`:18]

Figure 9: RBEA Deployment Measurements on Snapshots

6. RELATED WORK

Reliable Dataflow Processing: Flink offers coarse grained, job-level snapshot maintenance which grants various operational benefits. Several proposed reconfiguration and state management schemes [24] are restricted to fine-grained task management and reconfiguration, thus, lacking the benefits, applications and scale of global snapshots (Section 3.3.1). IBM Streams employs a pipelined checkpointing mechanism [38] that executes in-flight with data streams as with Flink's, tailored to weakly connected graphs with potential cycles. The most distinct difference to Flink's approach is that IBM Stream coordinates a two-phase snapshotting protocol: 1) First, all records in transit are consumed in order to make sure that they are reflected in the global state while blocking all outputs. 2) All operators trigger their snapshot in topological order, using markers as in our technique and resume normal operation. Flink's protocol only drains records within cycles without affecting regular processing whatsoever. Furthermore, Flink's alignment is a local operation and does not halt global progress or hold up output in an execution graph making it more transparent and non-intrusive. Finally, IBM Streams supports language abstractions for selective fault tolerance. On Flink, the choice of snapshotting state is achieved by simply using managed state versus unregistered state, without requiring further user intervention. In the scope of a pipeline/component, snapshots can also be enabled or disabled through Flink's configuration.

Apache Storm [8] initially offered only guaranteed record processing through record dependency tracking. However, the most recent releases of Storm (and Apache Apex[5]) incorporated a variant of Flink's algorithm to its core in order to support exactly-once processing guarantees. Meteor Shower [44] employs a similar alignment phase to Flink. However, it cannot incorporate cyclic dataflow graphs which is a common case for online machine learning [30] and other applications. The same solution does not cover state rescaling and transparent programming model concerns. Naiad [42] and the sweeping checkpointing technique [15] enforce in-transit state logging even in subgraphs where cycles are not present. Moreover, Naiad's proposed three phase commit disrupts the overall execution for the purpose of snapshotting. Finally, MillWheel [18] offers a complete end-to-end solution to processing guarantees, similarly to Flink. However, its heavy transactional nature, idempotency constraints and strong dependence on a high-throughput, always-available, replicated data store [28] makes this approach infeasible in commodity deployments. In fact, Apache Flink's distributed dataflow runtime serves today as a feature-complete runner of Apache Beam[6], Google's open-source implementation of the Dataflow Model[11].

Microbatching: Stream micro-batching or batch-stream processing (e.g. Spark Streaming [47], Comet [34]) emulates continuous, consistent data processing through recurring deterministic batch processing operations. In essence, this approach schedules distinct epochs of a stream to be executed synchronously. Fault tolerance and reconfiguration is guaranteed out-of-the-box through reliable batch processing at the cost of high end-to-end latency (for re-scheduling) and restrictive model, limited to incremental, periodic immutable set operations. Trident [16], a higher level framework built on Apache Storm offered exactly-once processing guarantees through a similar transactional approach on predefined sets but executed on long-running data stream tasks. While fault tolerance is guaranteed with such techniques, we argue that high latency and such programming model restrictions make this approach non-transparent to the user and often fall short in expressivity for a significant set of use-cases.

7. CONCLUSION AND FUTURE WORK

We presented Apache Flink's core mechanisms for managing persistent, large-scale pipelines with large application state in production. Flink is a flexible, reconfigurable distributed system which runs continuous, analytical, data-centric computation offering strong state consistency guarantees. A distinct snapshotting mechanism acquires a global view of the system periodically or upon demand which allows for coarse grained rollback recovery in a asynchronous, transparent and efficient manner. Snapshots allow for fundamentally practical reconfiguration usages, ranging from partial failure recovery to application versioning and debugging. Flink's ecosystem of modules and services built on its core offer different flavours of external state access and isolation while abstracting reliability concerns from the programmer. Finally, we discussed usages of Flink and the low execution overhead of Flink's snapshots in large-scale production deployments.

Future Work: Our main future focus on Apache Flink is to further improve its state management capabilities with incremental in-memory snapshots and automated system estimation of throughput and reconfiguration latency trade-offs for optimized incremental snapshots. Furthermore, we are planning to include the capability of auto-scaling pipelines according to runtime requirements without user circumvention. Finally, we want to support flexible state representations for iterative analysis through efficient bulk synchronous processing on streams.

Acknowledgments: We would like to thank here the whole community of Flink's contributors, peer-reviewers and the financial support from the End-to-End Clouds project granted by Stiftelsen för Strategisk Forskning (RIT10-0043) and the Streamline project funded by European Union's Horizon 2020 (688191).

8. REFERENCES

[1] AthenaX : Ubers stream processing platform on Flink. `http://sf.flink-forward.org/kb_sessions/athenax-ubers-streaming-processing-platform-on-flink/`.

[2] Blink: How Alibaba Uses Apache Flink. `http://data-artisans.com/blink-flink-alibaba-search/`, 2016.

[3] Introduction to Spark's Structured Streaming. `https://www.oreilly.com/learning/apache-spark-2-0--introduction-to-structured-streaming`, 2016.

[4] Rbea: Scalable Real-Time Analytics at King. `https://techblog.king.com/rbea-scalable-real-time-analytics-king/`, 2016.

[5] Apache Apex. `https://apex.apache.org`, 2017.

[6] Apache Beam. `https://beam.apache.org/`, 2017.

[7] Apache Flink. `http://flink.apache.org/`, 2017.

[8] Apache Storm. `http://storm.apache.org/`, 2017.

[9] Flink Forward. `http://flink-forward.org/`, 2017.

[10] Flink Survey. `http://data-artisans.com/flink-user-survey-2016-part-1/`, `http://data-artisans.com/flink-user-survey-2016-part-2/`, 2017.

[11] Google Cloud Dataflow. `https://cloud.google.com/dataflow/`, 2017.

[12] Real-time monitoring with Flink, Kafka and HB. `http://2016.flink-forward.org/kb_sessions/a-brief-history-of-time-with-apache-flink-real-time-monitoring-and-analysis-with-flink-kafka-hb/`, 2017.

[13] Rockdb. `http://rocksdb.org/`, 2017.

[14] Stream processing with Flink at Netflix. `http://sf.flink-forward.org/kb_sessions/keynote-stream-processing-with-flink-at-netflix/`, 2017.

[15] StreamING models, how ING adds models at runtime to catch fraudsters. `http://sf.flink-forward.org/kb_sessions/streaming-models-how-ing-adds-models-at-runtime-to-catch-fraudsters/`, 2017.

[16] The Trident Stream Processing Programming Model. `http://storm.apache.org/releases/0.10.0/Trident-tutorial.html`, 2017.

[17] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDBJ*, 2003.

[18] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[19] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *VLDB*, 2015.

[20] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *The VLDB Journal – The International Journal on Very Large Data Bases*, 2014.

[21] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. *Book chapter*, 2004.

[22] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130. ACM, 2010.

[23] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, page 28, 2015.

[24] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736. ACM, 2013.

[25] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*. ACM, 2010.

[26] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[27] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[29] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[30] G. De Francisci Morales and A. Bifet. Samoa: Scalable advanced massive online analysis. *The Journal of Machine Learning Research*, 16(1):149–153, 2015.

[31] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[33] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[34] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 63–74. ACM, 2010.

[35] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[36] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[37] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9, 2010.

[38] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce. Consistent regions: guaranteed tuple processing in ibm streams. *Proceedings of the VLDB Endowment*, 9(13):1341–1352, 2016.

[39] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. NetDB, 2011.

[40] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[41] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[42] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *ACM SOSP*, 2013.

[43] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[44] H. Wang, L.-S. Peh, E. Koukoumidis, S. Tao, and M. C. Chan. Meteor shower: A reliable stream processing system for commodity data centers. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012.

[45] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.

[46] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[47] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.