CHAPTER

3

Reliable Data Stream Processing

3.1 Introduction

Scalable data stream processing platforms such as Apache Flink are distributed systems. A distributed system consists of multiple processes connected through a network, that send and receive messages. Distributed systems are typically designed to make all concerns related to their distributed nature transparent to the user, offering the view of a single entity. The same transparency principle simplifies not only the programming model but also the work of a system designer. In this chapter, we address a crucial design challenge when it comes to distributed stream processing, that of guaranteeing a reliable unbounded execution. Reliable processing relates to guarantees offered by a continuously running distributed system despite partial failures or reconfiguration phases that can occur.

In this study, we first revisit the concept of consistent snapshots, which are replicas of the global configuration (i.e., state) of a system at a specific point of its distributed execution. Consistent snapshots capture a complete distributed state that can be used as a single atomic reference, to inspect [28], recover [60] or even alter a distributed computation. Most existing snapshotting protocols are distributed algorithms that make certain assumptions regarding the structure of a system (e.g., strongly connected process graph) to collect all process and channel (in-transit messages) states which are part of a snapshot.

A variety of operational challenges in distributed stream processing relate to guaranteeing that every process in the system consumes its input messages, updates its internal state and generates output messages without loss, even while independent process failures occur or the configuration of the system changes [11, 9, 61]. Past approaches [11, 62, 63] aimed to tackle this challenge via fine-grained communication protocols between individual processes to re-conciliate lost state (e.g., replaying input), filter out duplicates or restricting semantics to idempotent

operations [9]. While some of these techniques solve individual special cases of reliable processing, they do not offer a clear specification of the guarantees they implement. More importantly, they miss one of the most fundamental needs in stream processing, that of a unified state management mechanism. Unified state management should cover all aspects of application state, such as migrating, reconfiguring, querying, recovering and versioning the unbounded execution of a stream processing application.

In this chapter, we propose an execution model for stateful stream processing based on the notion of epochs. Epochs define concrete points in a distributed stream execution where state is atomically committed and thus, they can be used to provide unified state management. We argue that epochs do not enforce a discretized execution [12] and can instead be committed asynchronously using a special form of distributed snapshotting aligned on epochs. Starting from Chandy and Lamport's original algorithm, we present all necessary modifications needed to respect epoch order and prove the respective provided properties. Our final distributed protocol [30, 29] manages to capture the complete state of a weakly connected stream process graph [20, 64, 9, 54, 11] after an epoch, limiting channel state logging (in-transit messages) to graph cycles that optionally exist. Finally, this chapter serves as a self-contained specification of the epoch-based stream processing, its asynchronous implementation with snapshotting and its foundations. Chapter 4 presents a concrete usage and implementation of epoch-based snapshots for various operational needs in Apache Flink.

The chapter's outline goes as follows: section 3.2 offers an overview of consistent snapshotting in the fail-stop process model and related safety and liveness properties. Our preliminary analysis covers the concept of marker-based snapshotting, having Chandy and Lamport's protocol and its assumptions as a starting point, followed by several direct yet important generalizations. In section 3.3 we model the execution of a data stream processing system as a special case of interest and discuss the problem of reliable processing guarantees in that model as well as shortcomings of the related state of the art. Section 3.4 presents the specification and protocol for epoch-based stream processing, our proposed solution to reliable data stream processing. Finally, section 3.5 offers a summary of our approach.

3.2 Preliminaries

In this section we present a complete model and specification for consistent snapshots, identifying the necessary properties and known distributed algorithms that solve this problem on any message-passing system. Then, in section 3.3, we look into a more restricted version of snapshots, relevant to the problem of epoch-based stream processing.

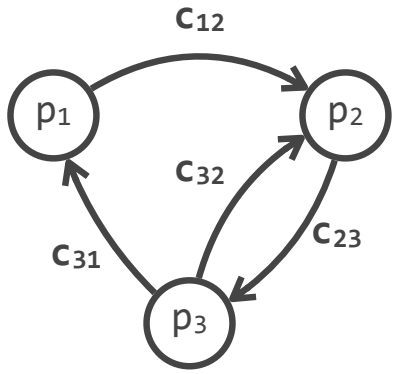


Figure 3.1: A process graph with three processes.

3.2.1 System Model

A distributed system model [60, 28] consists of two core components: processes and network channels. We assume a finite set of processes Π as well as a finite set of network channels $\mathbb{E} \subseteq \Pi \times \Pi$. For any two processes $p_i, p_j \in \Pi$ p_i can send messages to p_j iff $(p_i, p_j) \in \mathbb{E}$. In principle, we can represent a distributed system as a directed graph $G = (\Pi, \mathbb{E})$ with Π being its vertices and $\mathbb{E}$ representing its edges. For brevity, we will denote a channel $(p_i, p_j) \in \mathbb{E}$ simply as c_{ij}. In Figure 3.1 we depict the process graph $G = \{\{p_1, p_2, p_3\}, \{c_{12}, c_{23}, c_{31}, c_{32}\}\}$ as an example.

Process Model: Each process $p \in \Pi$ has its own local volatile state s_p that is initially empty. Furthermore, it is statically initialized with a number of input channels $\mathbb{I}_p = \cup_{x \in \Pi}\{c_{xp} | (x, p) \in \mathbb{E}\}$ and output channels $\mathbb{O}_p = \cup_{x \in \Pi}\{c_{px} | (p, x) \in \mathbb{E}\}$. In certain cases we will identify the channel on which a message is sent or received (e.g., m_{qp} or $m_{c \in \mathbb{O}_p}$), when that is necessary.

Channels and State: Channels are communication endpoints, implementing certain behavior (e.g., FIFO delivery) between processes and we decouple them from the in-channel state M, a multiset of all in-transit messages in the system (i.e., all messages sent but not yet delivered). Each channel $c_{pq} \in \mathbb{E}$ supports two types of events: $\langle send, m \rangle$ and $\langle recv, m \rangle$. Sending a message m results into $M \rightarrow M \cup \{m\}$ while receiving that message results into $M \rightarrow M/\{m\}$. For simplicity, in this context we assume that point to point network channels are reliable and have unbounded capacity, thus, they cannot overflow or lose messages that are in transit. Specification 2, presented later in the chapter, provides a formal description of a channel interface and properties for FIFO reliable channels.

Execution Model: The complete state or *configuration* of the system can be summarized by $\{\Pi_*, M\}$, where $\Pi_* = \{s_p | \forall p \in \Pi\}$ and M its in-transit messages. An execution of a system can be modeled via the use of transitions to its configuration $\{\Pi_*, M\} \rightarrow \{\Pi'_*, M'\}$, where each transition is caused by a primitive step or *action* executed by a single process $p \in \Pi$ and is summarized as (s_p, m, M_{OUT}, s'_p) as follows:

- s_p: The local state of p before the action.
- m: An input message $m \in M \cup \{\varnothing\}$.
- M_{OUT}: A set of output messages M_{OUT}.
- s'_p: The local state of p after the action.

For a process p, an action consists of individual events that respect the following event flow: 1) The reception of an input message m that causes that action (if $m = \{\varnothing\}$ then it is an *internal* action), 2) An internal computation: $(m, s_p) \rightarrow (s'_p, M_{OUT})$ and 3) a set of *send* events that put messages M_{OUT} into outgoing channels. After an action on $\{\Pi_*, M\}$ the new system configuration $\{\Pi'_*, M'\}$ contains the updated local process state $\Pi'_* = \Pi_* \cup \{s'_p\}/\{s_p\}$ and a new set of in-transit messages $M' = (M \cup M_{OUT})/\{m_{IN}\}$. For simplicity, in the most part we will refer to single-event actions such as $\langle send, m_{qp} \rangle_q$ and $\langle recv, m_{qp} \rangle_p$, each of which results into a new process state, unless stated otherwise.

Local and Global Execution: Sequences of events in a system form an execution E. An execution contains all events that have occured from the initial state of the system up to the events of the latest action. We often differentiate between E_p, the subset of an execution that contains events occured in process p and the global execution $E = \cup_{p \in \Pi} E_p$ that contains all individual events occurred across all processes in a distributed system.

Local Event Order: For each respective process $p \in \Pi$ computation follows a natural sequence of state transitions from an empty or initial state s_p^0 as such: $s_p^0, s_p^1, \ldots, s_p^n$. The respective events that cause these transitions $e_p^0, e_p^1, \ldots, e_p^i$ are totally ordered by a local causal order relation $\leq_p$, instrumented by its strict underlying local execution $E_p = \{e_p^0, e_p^1, \ldots, e_p^i, \ldots\}$ such that $e_p^i \leq_p e_p^j$ iff $i < j$. Since local event order is a total order it satisfies antisymmetry, totality and transitivity.

Failure Model: We assume a fail-stop model according to which a process stops and loses its current volatile state when a fault occurs. We will further refer to a process as *correct*, always in respect to an *observed* execution E when no fault occurs within E. We further denote a set of correct processes $\Lambda \subseteq \Pi$. The observed execution is context specific and contains all events that occur within the execution of a protocol (e.g., the consistent snapshotting protocol in subsection 3.2.3).

3.2.2 Consistent Cuts and Rollback Recovery

The concept of rollback recovery has many usages in distributed computing. It is, in essence, a reconfiguration process that aims to restore the execution of a full distributed system back to an intermediate point that was captured during a failure-free execution. A full system rollback can often be used as a mechanism for fault recovery. This is especially important in long-running system executions in order to avoid a complete re-execution of a computation that could take days or months. The main prerequisite of rollback recovery is to first be able to capture and copy the global state (configuration) of a system to stable storage, i.e., all process states as well as messages that are being in-transit.

Evidently, in a fail-stop model with volatile local state it is impossible to ensure that a complete system configuration will be recorded at the same instant due to the absence of a global atomic clock [28]. What is otherwise achievable by global state snapshotting techniques is to capture a "valid" configuration, one that can possibly occur during any failure-free execution [65]. A more acceptable formal interpretation of which global state can be considered valid lies on the definitions of causal event order and consistent cuts.

3.2.2.1 Global Causal Order

Given a global execution $E = \cup_{p \in \Pi} E_p$ including all individual events occurred across all processes in a distributed system, it is impossible to derive a total order relation for all events happening concurrently without a global atomic clock. However, there exist a partial order relation, known as *causal order* [66], given in Definition 3.2.1.

Definition 3.2.1. Given an execution E and $e, e', e'' \in E$, the *causal partial order* relation $\prec$ satisfies the following:

(1) $e \leq_p e' \implies e \prec e'$.

(2) if e and e' correspond to $\langle send, m_{pq} \rangle_p$ and $\langle rcvd, m_{pq} \rangle_q$ then $e \prec e'$.

(3) $(e \prec e') \wedge (e' \prec e'') \implies e \prec e''$ *(Transitivity)*.

The transitivity of causal partial order is especially important for specifying the order relation between two events that have otherwise occurred across two different processes in the system. Event diagrams, as the one depicted in Figure 3.2 are often used to visually represent executions of events in a distributed system where dots on each process line show events in the local execution of each process and arrows pointing out the causal dependencies of respective send and receive events in that execution between processes. In Figure 3.2 we highlight the casual relation between event a and d, b, e, f, g, h all of which belong to a's transitive closure of $\prec$ along the

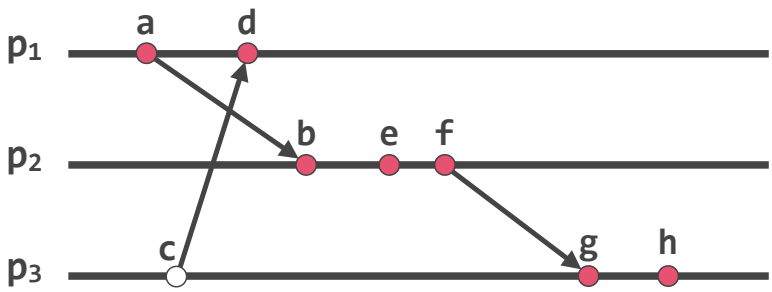


Figure 3.2: An event diagram highlighting events causally related to a.

causal relation path in the diagram. In the case of event c we cannot specify any causal order with respect to a since c is not within the transitive closure of a and vice versa, thus, in that case $a \| c$ (i.e., a and c are concurrent).

3.2.2.2 Valid Configurations and Consistent Cuts

A concept that is strongly related to the notion of distributed snapshots is a *distributed cut*. Definition 3.2.2 covers the most general case of a distributed cut. Notice that here we are including only local events preceding e.

Definition 3.2.2. Given an execution E and $e, e' \in E$, a distributed cut is a set $\mathcal{C} \subseteq E$ which satisfies the following invariant: $(e \in \mathcal{C} \wedge e' \leq_p e) \implies e' \in \mathcal{C}$

Example: A cut can be visually represented as a line that "cuts" through an execution, marking a frontier where all *locally* preceding events are in the cut. Figure 3.3 depicts two such cuts based on the process graph of Figure 3.1. In the case of cut C_1 (Figure 3.3(a)), only the events $\langle send, m \rangle_{p1}$ and $\langle rcvd, m' \rangle_{p1}$ are included in the cut. While C_1 is a possible distributed cut it cannot represent a valid state of the system since according to the included events, message m' was received but never sent, which is impossible. There is simply no possible execution E that can lead to such configuration. On the other hand, cut C_2 (Figure 3.3(b)) represents a valid configuration in the same execution, one where every event received was previously sent (event m' was only sent according to C_2 which does not violate any of the properties).

A consistent cut, described in Definition 3.2.3, is one that respects causal partial ordering and therefore, captures the notion of a "valid" system configuration.

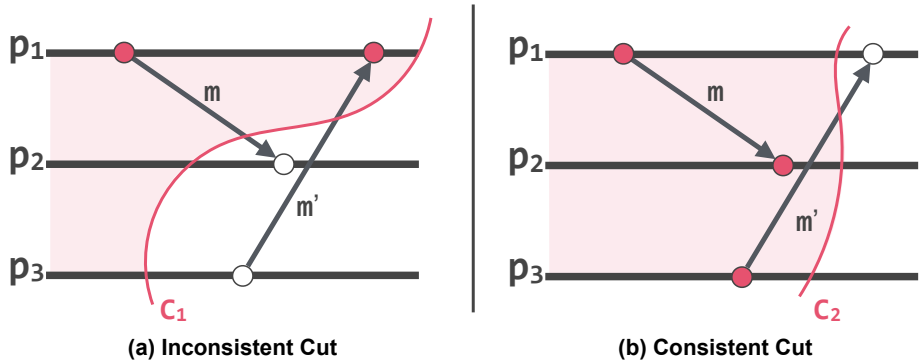


Figure 3.3: An example of an inconsistent (C_1) and a consistent cut (C_2).

Definition 3.2.3. Given an execution E and $e, e' \in E$, a *consistent distributed cut* is a set $\mathcal{C} \subseteq E$ which satisfies the following invariant: $(e \in \mathcal{C} \wedge e' \prec e) \implies e' \in \mathcal{C}$. Furthermore, a system configuration $\{\Pi_{\mathcal{C}}, M_{\mathcal{C}}\}$ after an execution bounded by a consistent cut $\mathcal{C}$ is also a *valid configuration*.

3.2.3 Consistent Snapshots: Specification and Fundamentals

Snapshotting protocols are distributed algorithms that capture a valid configuration of a distributed system during its execution. Since snapshots are distributed they have to be partially executed by each respective process in a coordinated manner which results into a consistent cut. Optimally, as described by Chandy and Lamport [28], the execution of a snapshotting protocol should not interfere with an application but run concurrently with it, capturing states when necessary while the application is running. In practice, a snapshot $S_{\mathcal{C}} = \{\Pi_{\mathcal{C}}, M_{\mathcal{C}}\}$ captures a valid configuration for *any* consistent cut $\mathcal{C}$. For brevity, we will imply that all snapshots are "consistent", unless stated otherwise.

Example: Consider a snapshot implementing the consistent cut C_2 that was presented previously in Figure 3.3(b). If we consider all events, messages and state transitions occurred, as depicted in Figure 3.4, it is more clear to visualize what the exact configuration of the system would be based on C_2. When it comes to process states, C_2 interleaves with processes p_1, p_2 and p_3 at states s_1^1, s_2^1 and s_3^1 respectively. Furthermore, message m' was in transit in the same cut (as being sent but never received). Thus, for completeness, message m' is also included in the snapshot. The full snapshot can be summarized as such: $S_{C_2} = \{\Pi_{C_2} = \{s_1^1, s_2^1, s_3^1\}, M_{C_2} = \{m'\}\}$.

Now let us define more formally the specification of consistent cuts, in terms of the interface of the protocol as well as the required safety and liveness properties that it should satisfy.

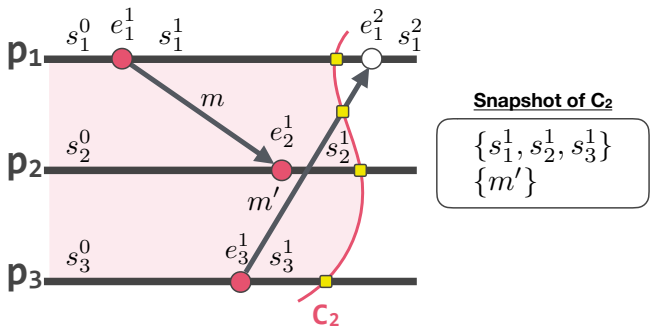


Figure 3.4: Contents of a snapshot for consistent cut C_2.

Specification 1: Consistent Snapshotting (csnap)

Event Interface:

Request: $\langle snapshot \rangle$: Initiates a consistent snapshot.

Indication: $\langle record|p, s_p^k, M_p \rangle : s_p^k \in \Pi_{\mathcal{C}} \wedge M_p \subseteq M_{\mathcal{C}}$.

Properties:

CSNAP1: *Termination*: $\Lambda = \Pi \implies \langle record|p, \_ \rangle \in E_p, \forall p \in \Pi$.

CSNAP2: *Validity*: Configuration $S_{\mathcal{C}} : \{\Pi_{\mathcal{C}}, M_{\mathcal{C}}\}$ is *valid*, where $\Pi_{\mathcal{C}} = \cup_{p \in \Pi} s_p$ and $M_{\mathcal{C}} = \cup_{p \in \Pi} M_p$.

Interface: Consistent snapshotting is more formally defined in Specification 1 (csnap) which summarizes the interface and expected safety and liveness properties to be satisfied by a system component that provides that functionality. The interface of the protocol contains two core messages: a request $\langle snapshot \rangle$ which initiates a consistent snapshot and $\langle record|p, s_p^k, M_p \rangle$ a response message that contains an internal state s_p^k as well as M_p a set of in-transit messages captured by process p.

Properties: We further break down the requirements of consistent snapshotting into two properties: *Termination* and *Validity*. Termination is a liveness property that is eventually satisfied if all processes are correct during an instance of a snapshot. The assumption that $\Lambda = \Pi$ might appear too strong, however, in the context of rollback recovery it is important to capture the complete application state. In case any failures occur, it is required to bring the whole system back to a global state captured during a failure-free execution, thus, no failures are tolerated during the execution of the protocol. The *Validity* property is a safety property that bears

equivalence to the invariant of consistent cuts (see Definition 3.2.3). In essence, for every local event e that precedes the event e_p^k that caused state $s_p^k : e \leq_p e_p^k$ it is implied that $e \in \mathcal{C}$ (Lemma 3.2.1).

Lemma 3.2.1. For any process $p \in \Pi$ with snapshotted state $s_p^k \in \Pi_{\mathcal{C}}$ and event $e \in E_p : e \leq_p e_p^k \implies e \in \mathcal{C}$.

Proof. Lemma 3.2.1 derives from the specification of events as well as Definition 3.2.2. We know that a state s_p^k is caused through a state transition triggered by event e_p^k. Thus, $(1) s_p^k \in \Pi_{\mathcal{C}} \implies e_p^k \in \mathcal{C}$. From Definition 3.2.3 we also know that $(2)(e' \leq_p e'' \wedge e'' \in \mathcal{C}) \implies e' \in \mathcal{C}$. (1) and (2) satisfy Lemma 3.2.1. □

3.2.4 Snapshotting Strongly Connected Graphs

Several algorithms have been proposed for satisfying the specification properties of consistent snapshots, targeting process graphs with specific structural characteristics as well as network assumptions (e.g., tolerating message loss [67]) and initiation strategies. In respect to structural characteristics of a process graph, connectivity is more especially a critical property that needs to be taken into consideration in the design of a snapshotting algorithm. We refer to connectivity as the set of assumptions we can make about the reachability of every pair of processes $p, q \in \Pi$ as expressed in Definition 3.2.4.

Definition 3.2.4. Given a process graph $G = (\Pi, \mathbb{E})$ and processes $p, q, w \in \Pi$, the reachability relation $\sim$ satisfies the following:

(1) $\exists c_{pq} \in \mathbb{E} \implies p \sim q$.

(2) $p \sim q \wedge q \sim w \implies p \sim w$ *(Transitivity)*.

One of the most popular algorithms from classic computer systems literature is the one by Chandy and Lamport [28] (C-L for short). The C-L algorithm captures a valid system configuration and terminates if the protocol is initiated by any given process in a strongly connected graph. A strongly connected graph is a process graph for which every process is reachable by another process (Definition 3.2.5).

Definition 3.2.5. A graph $G : (\Pi, \mathbb{E})$ is *strongly connected* iff $p \sim q, \forall p, q \in \Pi$.

In the most general case, the C-L algorithm relies on a *strongly connected* process graph to guarantee termination. Furthermore, it depends on reliable network channels with strict FIFO delivery order for ensuring validity. Despite having relatively strict connectivity assumptions, the C-L approach has been highly influential to the current work since it exhibits the use of a non-coordinated and non-blocking mechanism for capturing the configuration of any strongly connected

Specification 2: FIFO Reliable Channel (`fiforc`)

Event Interface:

Request: $\langle send, m\rangle : M \rightarrow M \cup \{m\}$.

Indication: $\langle rcvd, m\rangle : M \rightarrow M/\{m\}$.

Properties:

FIFORC1: *Reliable Delivery*: $p, q \in \Lambda : m_{pq} \in M \implies \langle rcvd, m_{pq}\rangle \in E_q$.

FIFORC2: *No Creation*: $\langle rcvd, m\rangle \in E \implies m \in M$.

FIFORC3: *No Duplication*: $\forall \langle rcvd, m\rangle, \langle rcvd, m'\rangle \in E \implies m \neq m'$.

FIFORC4: *FIFO Delivery*: $\forall p \in \Pi, q \in \Lambda$
$\langle send, m\rangle_{pq} \leq_p \langle send, m'\rangle_{pq} \iff \langle rcvd, m\rangle_{pq} \leq_q \langle rcvd, m'\rangle_{pq}$.

component. In this section we describe the C-L, starting from the formal specification of FIFO Reliable Channels.

FIFO Reliable Channels: Specification 2 summarizes the interface and expected properties of a FIFO Reliable channel in the semantics of our model. For brevity, we assume that this component is instantiated per directional channel in the system (one endpoint at the sender and another at the recipient process) and thus omit to include the addresses of the processes in the events of its interface, whenever that is unnecessary. Typically, FIFO reliable channels can be implemented on top of reliable point to point links which in turn build on fair-loss links [68]. In practice, TCP channels adequately satisfy the properties of this abstraction.

Algorithm Semantics: Regarding the notation of the algorithms throughout this part of the dissertation, we adopt an event-based algorithmic specification [68] describing the logic executed by the protocol within message-handling function literals (**upon** <msg> **on** <interface>). We assume that message-handlers are executed by a single thread, thus, there are no concurrent executions of respective handlers when it comes to a single component instance. Events are triggered on interfaces (e.g., `fiforc` and `csnap`) and respect a provided specification. Furthermore, we use <variable> $\leftarrow$ <value> to mark assignments and <interface> $\rightarrow$ <message> for triggering messages on respective interfaces. In fact, every event handler implements an atomic action for that process (altering internal and network state). The internal computation (application logic) is executed within the `process:` $(m, s_p, \mathbb{O}_p) \rightarrow (s'_p)$ function and yields a new internal state and output messages (triggered on the channels passed).

3.2.4.1 Analysis of Chandy-Lamport Snapshots

In Algorithm 3 we present the C-L protocol by integrating its complementary message-handling logic within the regular operation of a process. The core intuition of the C-L protocol is to disseminate a special marker message "$\odot$" in the computational graph which acts as a seperator between those events that precede a consistent cut from those that come after the cut. This technique aids processes to decide with purely local information on when they can trigger a local copy of their state (at the instant they first receive a marker) as well as which messages should be part of the global snapshot (in case they causally precede a marker). In Specification 1 we describe the consistent snapshotting algorithm, as integrated logic in the process model.

3.2.4.2 Termination

The *Termination* property as formulated in Specification 1 indicates that the if all processes are correct and the protocol is initiated on any process of a strongly connected graph then the global recorded state should eventually be indicated along all processes in the graph: $\Lambda = \Pi \implies \langle record|p, \_\rangle \in E_p, \forall p \in \Pi$. Termination can follow a direct proof based on the deterministic order of local events, the reliable delivery of FIFO channels and the reachability properties of a strongly connected graph as follows:

Proof. Every process in the C-L algorithm indicates its recorded state once a marker has been received through all of its input channels. Thus, it trivially follows that every process indicates a full recorded state iff a marker is received through all existing channels in $\mathbb{E}$:

$$\langle record|p, \_\rangle \in E_p, \forall p \in \Pi \iff \langle rcvd, \odot\rangle_{\forall c \in \mathbb{E}} \in E \tag{3.1}$$

Thus, it suffices to prove that

$$\Lambda = \Pi \implies \langle rcvd, \odot\rangle_{\forall c \in \mathbb{E}} \in E \tag{3.2}$$

Now let us assume that the protocol is initiated at any process $p \in \Pi = \Lambda$. Via deterministic local ordering of events process p broadcasts a marker through its output channels that is eventually received through FIFO reliable channels:

$$\begin{aligned} &\langle snapshot\rangle \in E_p \implies \langle send, \odot\rangle_{\forall c \in \mathbb{O}_p} \in E_p \\ &\implies \cup_{q \in \Pi : c_{pq} \in \mathbb{O}_p} \{\langle rcvd, \odot\rangle \in E_q\} \text{ (FIFORC1)} \end{aligned} \tag{3.3}$$

Given the reachability of strongly connected graphs (Definition 3.2.4) we can provide via induction the following generalization:

Algorithm 3: Chandy-Lamport Consistent Snapshots

Implements: csnap, **Requires**: fiforc $(\mathbb{I}_p, \mathbb{O}_p)$

1: $(\mathbb{I}_p, \mathbb{O}_p) \leftarrow$ configured_channels;
2: $s_p \leftarrow \varnothing$; ▷ volatile local state
3: $\mathrm{Recorded} \leftarrow \emptyset$; ▷ channels under logging
4: $s_p^* \leftarrow \emptyset$; $M_p \leftarrow \emptyset$; ▷ state in snapshot

---

5: **Upon** $\langle \mathrm{rcvd}, m \rangle$ *on* $c_{qp} \notin \mathrm{Recorded}, m \neq \odot$
6: $\quad s_p \leftarrow \mathrm{process}(m, s_p, \mathbb{O}_p)$; ▷ regular process logic
7: **Upon** $\langle \mathrm{rcvd}, m \rangle$ *on* $c_{qp} \in \mathrm{Recorded}, m \neq \odot$
8: $\quad M_p \leftarrow M_p \cup \{m\}$; ▷ record in-transit message
9: $\quad s_p \leftarrow \mathrm{process}(m, s_p, \mathbb{O}_p)$;
10: **Upon** $\langle \mathrm{rcvd}, \odot \rangle$ *on* $c_{qp} \in \mathbb{I}_p$
11: $\quad$ **if** $s_p^* = \mathrm{empty}$ **then**
12: $\quad\quad \mathrm{startRecording}()$;
13: $\quad$ Recorded = Recorded $-\{c_{qp}\}$;
14: $\quad$ **if** Recorded $= \emptyset$ **then**
15: $\quad\quad \mathrm{csnap} \rightarrow \langle \mathrm{record}|\mathrm{self}, s_p^*, M_p \rangle$;
16: **Upon** $\langle \mathrm{snapshot} \rangle$ *on* csnap
17: $\quad \mathrm{startRecording}()$;
18: $\quad$ **if** Recorded = $\emptyset$ **then**
19: $\quad\quad \mathrm{csnap} \rightarrow \langle \mathrm{record}|\mathrm{self}, s_p, \emptyset \rangle$;
20: **Fun** *startRecording()*
21: $\quad s_p^* \leftarrow s_p$; ▷ record local state
22: $\quad$ **foreach** $\mathrm{out} \in \mathbb{O}_p$ **do**
23: $\quad\quad \mathrm{out} \rightarrow \langle \mathrm{send}, \odot \rangle$;
24: $\quad$ Recorded $\leftarrow \mathbb{I}_p$

$$(3.3) \iff \ldots \iff \cup_{q \in \Pi : p \sim q} \{\langle \mathrm{send}, \odot \rangle \in E_q\} \tag{3.4}$$

and hence, via deterministic processing and FIFO reliable channels:

$$\begin{aligned} (3.4) &\iff \cup_{q \in \Pi : p \sim q} \{\langle \mathrm{rcvd}, \odot \rangle_{\forall c \in \mathbb{I}_q} \in E_q\} \\ &\iff \cup_{q \in \Pi : p \sim q} \{\langle \mathrm{send}, \odot \rangle_{\forall c \in \mathbb{O}_q} \in E_q\} \text{ (local event order)} \\ &\iff \langle \mathrm{rcvd}, \odot \rangle_{\forall c \in \mathbb{E}} \in E \text{ (FIFORC1)} \end{aligned} \tag{3.5}$$

□

3.2.4.3 Validity

We should prove that every snapshot acquired via the C-L protocol is valid. It suffices to show that the events that are included in the cut compose a consistent cut in the execution of a system as it was shown in Theorem 3.2.1.

Proof. Given two events $e_p, e_q \in E$ for which $e_p \prec e_q$ we need to prove that if $e_q \leq_p e_q^k$ then this implies that $e_p \leq_p e_p^l$. From Lemma 3.2.1 we know that this condition satisfies the requirement of consistent cuts.

Let $e_p^{\odot}$ represent the local snapshot aquisition step within a process p. Based on the marker-forwarding logic of the C-L protocol, process p captures its local state s_p^k and forwards the marker further in the same computational step $e_p^{\odot}$, therefore $e_p^k = e_p^{\odot}$. No other local event can possibly occur between a $\langle rcvd, \odot \rangle$ and a respective $\langle send, \odot \rangle$ event to alter the state of process p. It therefore suffices to prove the following: for two events $e_p, e_q \in E$ for which $e_p \prec e_q$, if $e_q \leq_q e_q^{\odot}$ then this implies that $e_p \leq_p e_p^{\odot}$. We identify two possible cases (I) $(p = q)$ and (II) $(p \neq q)$.

(I) $p = q$

Given that $p = q$, let us call the two events e_p and e'_p, so that $e_p \leq_p e'_p$ (1). If e'_p is part of the snapshot, then $e'_p \leq_p e_p^{\odot}$ (2). From (1) and (2) it derives directly that $e_p \leq_p e'_p \leq_p e_p^{\odot}$ and therefore, e_p should also be part of the the cut.

(II) $p \neq q$

Assume that p and q are directly connected through a FIFO reliable channel $c_{pq} \in \mathbb{E}$. We need to prove the following for causal consistency to be satisfied:

$$e_q \leq_q e_q^{\odot} \implies e_p \leq_p e_p^{\odot}. \tag{3.6}$$

By contradiction, suppose that the following statement is true:

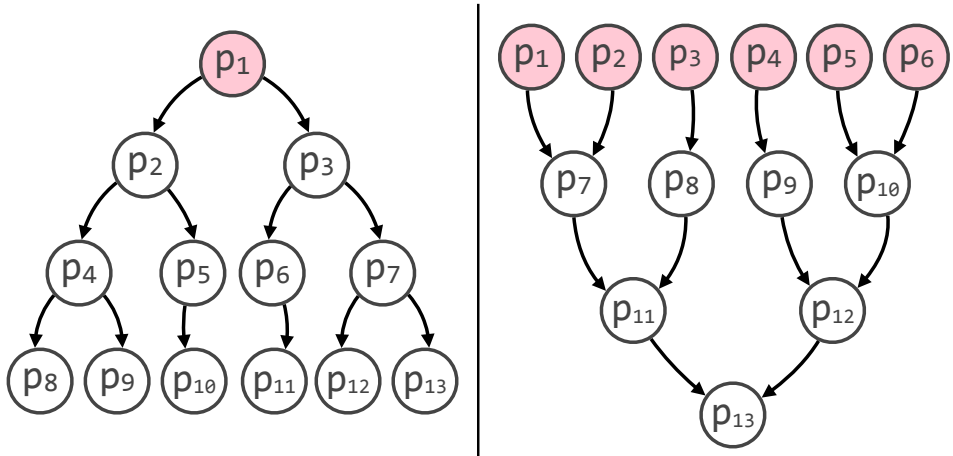
$$e_p^{\odot} \leq_p e_p \wedge e_q \leq_q e_q^{\odot} \tag{3.7}$$

Given that $e_p \prec e_q$ and p and q are directly connected processes we can assume that $e_p = \langle send, m \rangle_{pq}$ $e_q = \langle rcvd, m \rangle_{pq}$. Since c_{pq} implements a FIFO Reliable Channel (Specification 2) we know that every message m sent is delivered (FIFORC1) and all messages in a channel maintain FIFO delivery order (FIFORC4).

$$e_p^{\odot} \leq_p \langle send, m \rangle_{pq} \iff e_q^{\odot} \leq_q \langle rcvd, m \rangle_{pq} \tag{3.8}$$

If we however infer FIFO delivery order (3.8) on (3.7) we arrive to the following contradiction.

$$e_q^{\odot} \leq_q \langle rcvd, m \rangle_{pq} \wedge \langle rcvd, m \rangle_{pq} \leq_q e_q^{\odot} \not\Leftrightarrow . \tag{3.9}$$



(a) Weakly Connected Process Graph **(Single Initiator)**

(b) Weakly Connected Process Graph **(Multiple Initiators)**

Figure 3.5: Examples of weakly connected graphs with possible protocol initiators.

Finally, by using (3.6) as an induction step we can trivially generalize for any causally related events that can occur along the chain of reachable processes from p to q, $p, q \in \Pi$ where $p \sim q$.

$$\begin{aligned}
&(c_{pq} \in \mathbb{E} \wedge e_p \prec e_q) \implies (e_q \preceq_q e_q^{\odot} \implies e_p \preceq_p e_p^{\odot}) \\
&\iff \ldots \iff \text{(induction)} \\
&(p \sim q \wedge e_p \prec e_q) \implies (e_q \preceq_q e_q^{\odot} \implies e_p \preceq_p e_p^{\odot}
\end{aligned} \tag{3.10}$$

Since we assume a strongly connected graph Equation 3.10 is satisfied for $\forall p, q \in \Pi$, therefore in that case causal consistency can never be violated. Furthermore, in Theorem 3.2.2 we summarize a general observation regarding validity in marker-based snapshotting protocols.

Theorem 3.2.2. Marker-Based Snapshotting: A valid snapshot can be acquired via the use of a marker-forwarding logic in a process graph with FIFO reliable links. According to that logic, a local snapshotting action $e_q^{\odot}$ in a process $q \in \Pi$ is causally related to the snapshotting action $e_p^{\odot}$ of an adjacent process p (i.e., $e_p^{\odot} \prec e_q^{\odot}$) via the the forwarding of a special marker $\odot$ included in that action that separates all events the precede and follow a consistent cut.

□

3.2.4.4 Applicability of Marker-Based Snapshotting

The marker-based protocol of C-L creates a single snapshot and guarantees termination if initiated by a single process in a strongly connected graph. We will now examine alternative protocol initiation strategies that apply to non-strongly connected graphs. As it was shown in the termination analysis in paragraph 3.2.4.2, termination is based on strong connectivity, i.e., $p \sim q$ for any pair of processes $p, q \in \Pi$ (Definition 3.2.5). However, the applicability of the protocol can be generalized further by lifting two of its main assumptions (I) *strongly connected process graphs* and (II) *single initiator*.

(I) Supporting Weakly Connected Graphs

A directed graph is *weakly connected* if by replacing all of its directed edges with undirected ones it produces a strongly connected graph. In principle, any graph that forms a connected component is a weakly connected graph. Consider the process graph of Figure 3.5(a). Since not every process is reachable from any other process (e.g., $p_5 \not\sim p_2$) the marker-based protocol is not guaranteed to terminate if started from any single process. However, in the same process graph $p_1 \sim p, \forall p \in \Pi$ thus, if initiated upon p_1 the C-L protocol will terminate. Lemma 3.2.3 summarizes this observation.

Lemma 3.2.3. Given a weakly connected process graph $G = (\Pi, \mathbb{E})$, a marker-forwarding snapshotting protocol can terminate with a single initiator iff there exists $p \in \Pi$ so that $\forall q \in \Pi$ q is reachable from p (i.e., $p \sim q$).

(II) Supporting Multiple Initiators

We can generalize the applicability of C-L snapshots even further, to any weakly connected process graph for which we can coordinate more than a single initiating process. In the simplest case we can naively initiate the protocol on all processes which is guaranteed to terminate on any graph trivially. However, it is feasible to select a minimal set of initiators $\mathbb{A} \subseteq \Pi$ that can, in combination, reach every other process. This observation follows trivially from Lemma 3.2.3 and generalizes it further since the union of reachable processes via $\mathbb{A}$ yields the remaining processes in the process graph. We summarize this generalization in Definition 3.2.6.

Definition 3.2.6. Given a weakly connected process graph $G = (\Pi, \mathbb{E})$, a marker-forwarding protocol can terminate with any set of initiators $\mathbb{A} \subseteq \Pi$:
$\Pi/\mathbb{A} \subseteq \{q \in \Pi | \exists p (p \in \mathbb{A} \wedge p \sim q)\}$.

In the example of Figure 3.5(b) consider $\mathbb{A} = \{p_1, p_2, p_3, p_4, p_5, p_6\}$. From Definition 3.2.6 we know that we can reach $\{p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}\} = \Pi/\mathbb{A}$. Notice that if we exclude e.g., p_5 from $\mathbb{A}$ then we can only reach $\Pi/\{\mathbb{A}, p_5\}$, however, $\Pi/\mathbb{A} \not\subseteq \Pi/\{\mathbb{A}, p_5\}$. In other words, while the same processes can be reached as

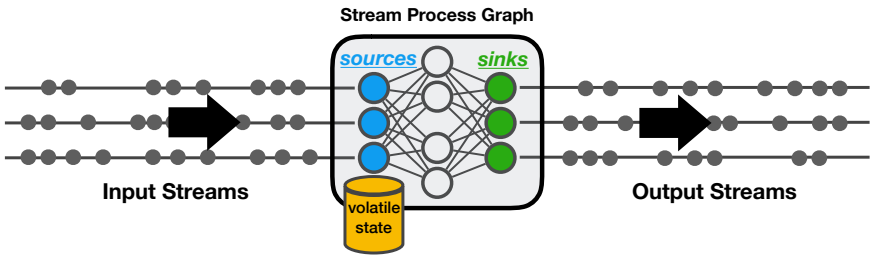


Figure 3.6: A Stream Process Graph.

before without p_5 e.g., $p_6 \sim p_{10}$, no process can reach p_5 itself and thus, p_5 should be part of the initiators in $\mathbb{A}$. In this example, $\mathbb{A}$ is also a minimal set.

3.3 Issues in Reliable Stream Processing

Stateful, stream processing graphs are special-purpose distributed systems that are used for pipelining computational tasks and their interdependencies in continuous, possibly infinite executions. In this section, we present the special properties of stateful stream processing, as well as open problems that can be reduced to the core issue of reliable processing in long-running executions.

3.3.1 Specification of Stateful Stream Processing

First, we make a set of model assumptions for data stream processing which we distill from the common characteristics of known scalable stream processors (e.g., Flink [43] , Apex [39], IBM Streams [56], Storm [5], SEEP [10, 11] and Heron [69]).

3.3.1.1 Stream Process Model

A stream processing system (depicted in Figure 3.6) can be modeled as a special purpose process graph $G = \{\Pi, \mathbb{E}\}$ that, as a single unit, receives ordered sequences of messages as an input, updates its volatile state and generates ordered sequences of output messages. Internally, individual processes that we call "tasks" follow a strict message-driven execution of computational steps, as described already in section 3.2. We further call each step a *stream process action* and denote $\langle \mathtt{proc}, m, M_i \rangle_p$ a process action of a task p that is triggered by an input message $m \in M$ (i.e., $m \neq \{\varnothing\}$) and produces a set of output messages M_i associated with that action.

The internal computation logic within a *stream process action* is encapsulated within a function `process:` $(s_p^i, m_i) \rightarrow (s_p^{i+1}, M_i)$ that typically maps to the invocation of a higher-order function (e.g., map, filter, fold) and its corresponding user-defined function literals (see Flink's model as a reference in chapter 2). More

complex functions can be composed on top such as binary stream logic (e.g., join, co-map, co-flatmap) as well as blocking operator logic that is common in stream processing (see stream windows and iterations in Chapters 5 and 6). For example, event-time windows are also invoked via a *stream process action* triggered by watermarks which are input messages, discussed in detail throughout chapter 6. In this chapter, we will consider stream computation at the lowest processing level (i.e., that of a *stream process action* and corresponding events). Finally, we assume that all tasks operate on a fail-stop processing model, and thus, they stop their execution upon any failure and lose their volatile state.

3.3.1.2 Stream Process Graphs

The process graph of a stream processing system, also referred to as "stream process graph" is, in the most basic case, a connected directed acyclic graph[1] which contains two special types of tasks: sources and sinks.

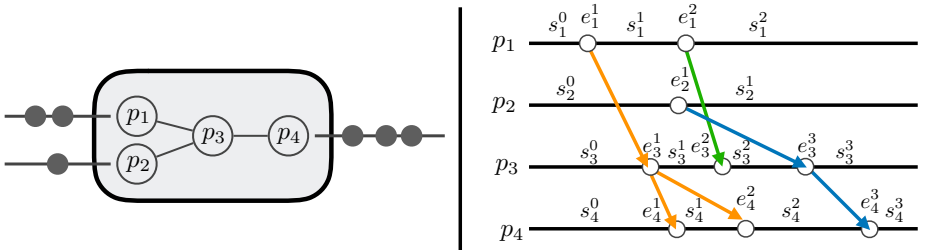
Sources: We term as "sources" a *non-empty* set of tasks that contain no input channels from other tasks and can, in combination, reach all other tasks in the graph. Since sources share the same properties as the initiators in Definition 3.2.6 we will use the same notation $\mathbb{A} \subseteq \Pi$. In practice, sources serve as the main entry point for input streams in the graph and typically bind to message queues that are external to the system. However, it is also possible for sources to generate streams deterministically. In either case, message generation internally occurs within the regular *stream process action* of the sources and follows a strictly deterministic execution. This is achieved either via pulling data from logged input streams (using a partitioned log [51, 70]), a deterministically generated sequence of records (e.g., Fibonacci sequence) or a mixture of logged and deterministically generated input (e.g., records and derived low watermarks).

Sinks: We define another special set $\Omega \subset \Pi$ of tasks that have no output channels to other tasks in G, referred to as "sinks". In practice, sinks serve as an exit point of a stream process graph by pushing sequences of messages outside the system, e.g., to file systems, message queues or DBMSs via external communication mechanisms. Sinks are especially important for committing external side effects (i.e., output message streams) consistently, as it will be shown in chapter 4.

3.3.2 Productions and the Problem of Reliable Processing

An inherent relation between actions on the strictly message-based execution model of stream processing tasks is the one of *productions*. In Figure 3.7(a) we depict a simple process graph consisting of two sources p_1 and p_2 and a single sink p_4. A

[1] section 3.4.4.4 covers cycles as a special case, while chapter 6 elaborates further on semantics and incorporation of arbitrarily nested cyclic computation in stream process graphs.



(a) A stream process graph (b) An event diagram of a stream computation

Figure 3.7: Example of a stream process graph and a possible execution

corresponding event diagram in Figure 3.7(b) shows a possible execution based on that process graph. Notice that every stream of messages initiated via p_1 and p_2 respectively creates a tree of *stream process actions* often along the whole diameter of the process graph. We call each relation within a tree a "production" and define it further as a partial order relation $\rightsquigarrow$ in a stream execution E in Definition 3.3.1.

Definition 3.3.1. Given an stream process graph execution E and $e, e', e'' \in E$, the *production partial order* relation $\rightsquigarrow$ satisfies the following:

(1) if e and e' correspond to $\langle proc, m_i, M_i \rangle$ and $\langle proc, m_j, M_j \rangle$ respectively where $m_j \in M_i$ then $e \rightsquigarrow e'$.

(2) $(e \rightsquigarrow e') \wedge (e' \rightsquigarrow e'') \implies e \rightsquigarrow e''$ *(Transitivity)*

In Figure 3.7(b) we highlight the transitive closure of each production with a separate color and call each of them, a *production tree*. We can further generalize that each *production tree* is in fact a subset of an execution, related to the transitive closure of an input event (occurring at a source) since no internal events can occur. Furthermore, due to the finite, directed acyclic structure of a stream process graph, productions can only be finite sets, e.g., in the previous example $e_1^1 \rightsquigarrow \{e_3^1, e_4^1, e_4^2\}$ and $e_1^2 \rightsquigarrow \{e_3^2\}$. Event productions imply causality but the inverse is not true. In fact, due to the local event order there is a causal relation between most events that occur from the very beginning of a computation up to its indefinite execution. We can further reduce the causal ordering relation to stream process graphs as defined in Definition 3.3.2 (modification in blue).

Definition 3.3.2. Given an stream process graph execution E and $e, e', e'' \in E$, the *causal partial order* relation $\prec$ satisfies the following:

(1) $e \leq_p e' \implies e \prec e'$.

(2) $e \rightsquigarrow e' \implies e \prec e'$.

(3) $(e \prec e') \wedge (e' \prec e'') \implies e \prec e''$.

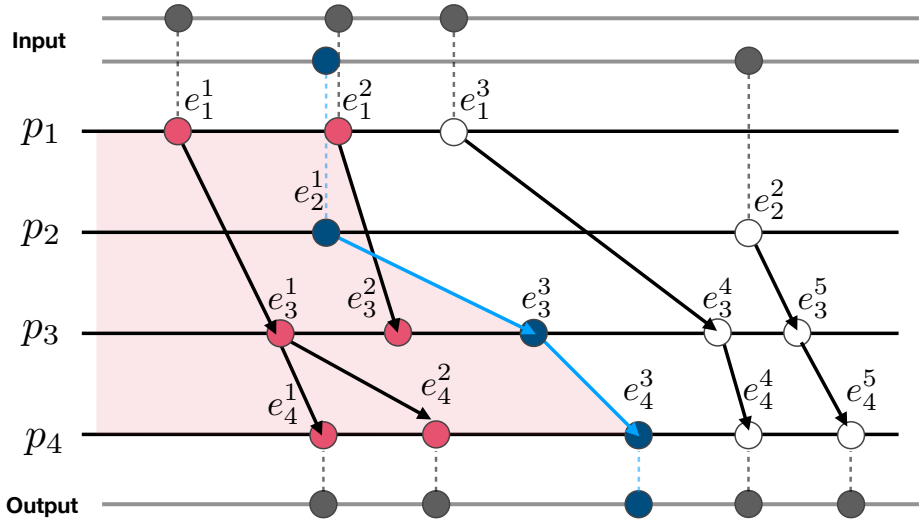


Figure 3.8: A highlight of all events that produce (blue) and cause (red) e_4^3.

To visualize the two relations, consider a continuation of the previous execution of Figure 3.7(b) including mappings of source and sink events to input and output streams as depicted in Figure 3.8. A selected event e_4^3 is part of the production tree of e_2^1 since $e_2^1 \rightsquigarrow \{e_3^3, e_3^4\}$ and causally succeeds most events of the distributed execution up to this point (as shown in red in the figure). It is noticeable that throughout an execution every individual event contributes to side-affects both to the stream output with its productions but also to all causally succeeding events via internal state transitions.

Problem Intuition: Given the unbounded nature of stream processing executions and the large number of tasks allocated in data centers, we expected failures to occur especially often during the lifetime of an application. If we ought to strongly rely on such an execution (e.g., processing medical records or financial transactions) and the correctness of its corresponding results we have to ensure that the system hides inconsistencies from the outside observer. This means the observed system execution (inferred by looking at each individual state transition) should hide events that correspond to incomplete or abnormal state that might have occurred at its tasks (e.g., incomplete productions, undetected event patterns etc.). Before providing a clear definition of the problem of reliable stream processing we will first retrospect over the state of the art as well as the basic modern needs for reliable processing.

3.3.3 Reliable Stream Processing: Past and Current Considerations

While in our design we have adopted a fail-stop failure model, there have been several approaches in that past that assume a fail-recovery model, where tasks can restart independently from failures and employ different mechanisms [63, 62, 9, 11, 10] to amend their execution to reproduce computations that were possibly lost. We briefly discuss two such related approaches and their considerations.

Task-Level Rollback Recovery: One approach to provide fault tolerance as well as aid reconfiguration is to allow processes to checkpoint independently their channel and internal states to stable storage during arbitrary points in their execution and recover from there independently. However, this creates several non-trivial complications. Consider in the previous example of Figure 3.8 a failure of task p_3 right after executing event e_3^3.

Assuming a fail-recovery model, p_3 could restart from a previously captured state, e.g., s_3^1 and it would have to ensure that 1) pending messages are re-executed strictly in the same order (i.e., processing the message sent from p_2 after the two messages sent by p_1) and 2) consumer tasks do not execute duplicate computation. In our example, task p_4 would have to avoid processing duplicate messages since that would result in an abnormal execution that does not reflect the case of no failures occurring (e.g., if e_4^3 has already already occurred). Several existing approaches consider extensive input logging and complex reconciliation protocols with reachable processes [11, 71, 62] in the graph that is often infeasible with one-directional communication. Furthermore, given the fine-grained nature of task-level rollbacks, from the user- or consuming service-level there is no clear distinction on which parts of the global computation have been committed, thus, no clear guarantees can be made trivially on the state of the application and globally committed output.

Transactional Productions and Idempotency: Another related approach to the problem is to commit each single production as a transaction using an external fault-tolerant storage system, before delivering it to a consumer service or user. Google's Millwheel stream processing system [9] is using BigTable[72] for that purpose which provides support for distributed transactions, committing a production [2] of an input record carrying a specific key with the same transaction identifier (for idempotency). This works trivially in purely data-parallel computations (per-key) where task-level causal order can be possibly violated and all that is needed is to guarantee that each operation per key has succeeded. However, since the more general stream processing model presented here addresses the complete state of each task (across all keys) this approach cannot be generally applicable. Furthermore, the same approach suffers from non-trivial optimisations (e.g., deduplication, batching

[2] Millwheel calls this concept a strong production and differentiates it from weak productions that related to task-level rollback recovery and avoid pre-commits for naturally idempotent productions.

transactions) and also falls short of offering broader application-level guarantees if anything more than committed aggregates per-key are needed.

3.3.4 Related Open Problems

Evidently, there is a need for an application-level consistency model for reliable data stream processing that goes beyond the scope of task or procuction-level fault tolerance. We have identified a set of emerging system needs that relate indirectly to the same problem but cannot be collectively covered or composed by prior approaches.

Support for Reconfiguration: The stream process graph G as well as user-defined logic (e.g., function literals) that is being executed within the tasks is not necessarily static. Given that a stream system does not typically run for a few minutes or hours but is meant to do continuous processing it is often required to apply system-level changes. For example, there is commonly a need to increase or decrease the number of task instances that execute a logical operation in parallel, also known as scale-out or scale-in operation in a cloud computing infrastructure. This problem also demands flexibility on how state is partitioned and re-allocated to different numbers of tasks. Changes in task logic (the *process* function) is also a common case of reconfiguration, associated with "bug fixing" or issuing upgrades on the application logic ran by specific tasks. The main challenge of any type of reconfiguration, is to apply all changes in a reliable and transparent way, similarly to fault recovery.

Execution Migration and Provenance: Stream processing executions might be long-running, however, this is not always the case with the underlying infrastructure. Given the vast amount of diverse IaaS platforms for both on-premise and cloud deployments there is an increasing need for flexibility on *where* an execution is physically occurring. For a reliable stream processing system that means that there is an inherent need to first identify, capture and version the global state of a stream execution e.g., E^{v1} in order migrate to E^{v2} where $E^{v1} \subseteq E^{v2}$. To enable this, all production trees would need to maintain causal dependencies from E^{v1} to E^{v2}.

State Access Isolation: Following the recent trend of main-memory databases, many latency-critical applications today rely on fast read access. The state of of individual processors in stream processing graphs poses an attractive alternative to main-memory databases [73], since among other things, it always reflects an up-to-date summary of long sequences of input records. Given that this state unlike DBMSs is not transitionally committed it is restricted, by definition, to dirty reads. That yields a need for a form of version control for task states in stream processing, separating state that is potentially safe to read from volatile state that can be invalidated. That can further allow the possibility of access isolation mechanisms that can guarantee causal consistency to external queries.

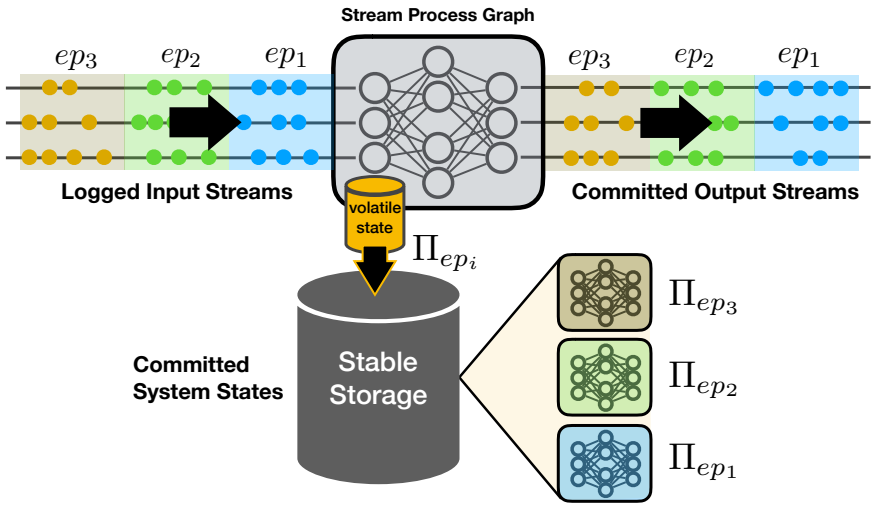


Figure 3.9: Overview of Epoch-Based Stream Processing.

3.4 Epoch-Based Reliable Stream Processing

The concept of epoch-based stream processing offers a uniform solution to reliable streaming and fulfills all relevant state management needs discussed in subsection 3.3.4. In this section, we introduce the concept and describe several design approaches to materialize it, while aiming to satisfy principal design properties (uncoordinated and blocking-free execution, transparency and compositionality).

3.4.1 Concept Overview

In Figure 3.9 we visualize the essential idea of epoch-based processing at a conceptual level. We assume, as before, a stream process graph G with a fail-stop model and a deterministic event sequence at its sources (e.g., logged streams). The main goal is to expose a continuous execution that satisfies reliable processing, i.e., maintaining all event productions and respecting causality. However, instead of reasoning about individual events, we distinguish discrete stages, or "epochs" that represent finite intermediate subsets of a continuous execution E as such $E^{ep_1} \subset E^{ep_2} \subset E^{ep_3} \ldots \subset E$.

Conceptually, each epoch represents a part of the computation that is atomically processed (either completes or restarts). This allows us to lift all concerns regarding reliability from the level of individual records or productions to the coarse-grain level of epochs. Once an epoch has been committed all the states of the process

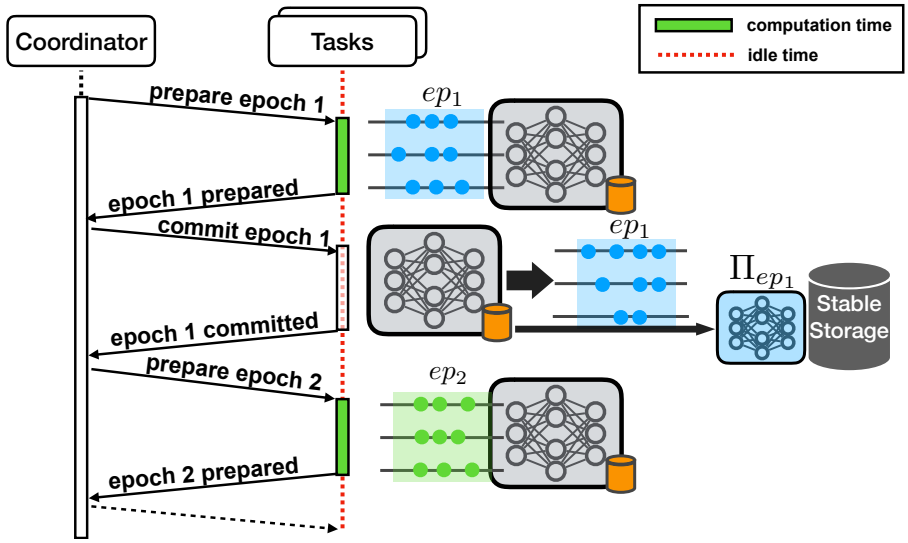


Figure 3.10: Example of Synchronous Epoch Commit.

graph in addition to external output streams reflect the full state of an execution up to that point. In case of failure during the execution of an epoch ep_n we can simply rollback the deterministic input and global state of the process graph to a previously completed epoch (e.g., ep_{n-1}). A core requirement for employing an epoch-based execution is to be able to capture all individual task states of the system upon the completion of an epoch on stable storage. It is, in essence, the configuration of the system at the exact point that the input of epoch ep_i (and nothing more) has been fully processed. Thus, a snapshot S_{ep_i} of that configuration would consists only of local task states $S_{ep_i} = \{\Pi_{ep_i}\}$. It is important to note here that there are many possible executions up to an epoch completion, e.g, $E_{ep_i} \neq E'_{ep_i}$, since no total processing order is guaranteed in a distributed stream execution. However, a committed epoch includes the result of a single possible execution up to an epoch. This notion relates to the informal use of *exactly-once* [9, 29] end-to-end processing semantics which, in this case, it specifically refers to "exactly one" epoch commit.

3.4.2 Synchronous Epoch Commit

In the simplest case, epoch-based stream processing can be achieved via staging the underlying execution. Staging is typically implemented using an atomic commit protocol between a coordinator process and the tasks of the graph.

In Figure 3.10 we show how a synchronous (two-phase) atomic commit protocol can be used to stage epochs in a stream processing execution. The first phase

ensures that all computation has been completed and the second phase completes once all side effects have been persisted to stable storage (i.e., task states and stream output). While this approach works in practice, there is one major downside, it enforces a blocking coordination protocol. The core of the problem of staging is that most tasks will have to remain idle 1) until every other task has finished computing and 2) while every other task commits its state and side effects to stable storage. The latter can be potentially avoided with asynchronous copying methods, however, if epochs are frequent the communication overhead of the protocol itself can often overtake the actual computation time. On the other hand, if epochs are infrequent, tasks are less idle but the end-to-end latency between the ingestion of input streams and the time results are committed can be too high and thus, too late for many applications (e.g., critical event monitoring).

3.4.2.1 The Case of Micro-batching

Stream micro-batching [12] is a mechanism that emulates stream processing capabilities on batch processing systems which employ short-lived task execution (i.e., Spark [27], MapReduce [8]). In essence, micro-batching is equivalent to the synchronous epoch commit protocol. The main idea is to have a coordinating process (i.e., the driver in Spark) that periodically divides a stream into batches and then schedules a job (process graph of short-lived tasks) per batch. In the case of micro-batching, a new set of tasks will be scheduled to overtake the computation on each epoch. In case of a failure, a batch is re-executed by the same or a new set of tasks (parallel recovery[12]). This grants the flexibility to re-allocate and reconfigure the computation per batch, however, it also introduces additional scheduling overhead on top of the synchronous epoch commit costs mentioned before. Furthermore, the original version of discretized streams enforced the use of time-discretized batches in its programming model, making the whole architecture non-transparent to the user-facing programming model. This problem can be attributed to an initial design choice in the context of batch execution rather than being an inherent property of epoch-based stream processing, as newer versions of Spark Streaming offer a more fluid stateful programming model (e.g., Structured Streaming [74]).

3.4.3 Asynchronous Epoch Commit

We seek for non-blocking mechanism for committing epochs in a stream computation, one that does not halt the regular execution of the stream processing graph and allows tasks to progress asynchronously across epochs without the need to remain idle for blocking synchronization. A possible solution to satisfying this fundamental requirement lies at the use of consistent snapshots, regulated to respect epochs in order to bypass the need for staging the overall execution.

3.4.3.1 Epoch Events and Cuts

First, we will examine the notion and implications of epochs in a continuous execution. An epoch marks a logical time for a distributed stream computation known at all sources of a stream process graph. We will denote as ep_i the epoch of time $i \in \mathbb{Z}$ and further assume that sources are notified about the completion of an epoch, e.g., ep_n through special *epoch events* : $\langle ep_n \rangle$.

Epoch events can be either issued by an external coordinator process to all source tasks, or instructed by a user or simply inferred through special punctuation events pulled from incoming data streams. For generality, we will make no special assumptions at this point on how epoch events are triggered, though, we will assume that they are eventually triggered at all sources $\mathbb{A} \subseteq \Pi$ of a stream process graph and are monotonic (given in Definition 3.4.1).

Definition 3.4.1. Epoch Event Monotonicity: Given a source task $p \in \mathbb{A}$ and two local epoch events in E_p: $e_p^k = \langle ep_i \rangle$ and $e_p^l = \langle ep_j \rangle$ then $i < j$ iff $k < l$.

The notion of epoch events can help us reason about a complete preceding computation in a stream process graph. In Definition 3.4.2, we introduce "Epoch Cuts", a stricter form of a consistent cuts expressing not only a causally consistent, but also a *complete* execution of a stream process graph in respect to an epoch.

Definition 3.4.2. Epoch Cut: An Epoch Cut $\mathcal{C}_{ep_i}$ satisfies the following invariants in stream process graphs (acyclic) for $e, e' \in E$ and $p \in \mathbb{A}$:

(1) $(\langle ep_i \rangle \prec e) \implies (e \notin \mathcal{C}_{ep_i})$

(2) $(e \prec \langle ep_i \rangle) \implies (e \in \mathcal{C}_{ep_i})$

(3) $((e \in \mathcal{C}_{ep_i}) \wedge (e \rightsquigarrow e')) \implies (e' \in \mathcal{C}_{ep_i})$

(4) $((e \in \mathcal{C}_{ep_i}) \wedge (e' \prec e)) \implies (e' \in \mathcal{C}_{ep_i})$ (consistent cut)

We will be using invariant (4) in order to describe *Causal Consistency* and (1-3) as the necessary conditions for *Epoch Completeness* in a cut.

Example: Consider the execution illustrated in Figure 3.11 which is based on the process graph of Figure 3.1 and includes epoch events for ep_n. The events marked as green denote everything that locally precedes epoch events, including their productions, while all succeeding production trees are marked as red. Based on Definition 3.4.2, the epoch cut $\mathcal{C}_{ep_n}$ should contain exactly all green events including the epoch events. Both C_1 and C_2 satisfy *Causal Consistency*, yet, only C_2 satisfies *Epoch Completeness* and therefore, $C_2 = \mathcal{C}_{ep_n}$. In the case of C_1 we identify two invariant violations for Epoch Cuts. First, events $e_1^4, e_2^3 \in C_1$ while $\langle ep_n \rangle \prec_{p1} e_1^4$ and $\langle ep_n \rangle \prec_{p2} e_2^3$ which violates invariant (1) of Definition 3.4.2. Second, $e_4^3 \notin C_1$, however, according to Definition 3.4.2(3) it should be part of the epoch cut since $(e_1^2 \in C_1) \wedge (e_1^2 \rightsquigarrow e_4^3)$. Therefore, $C_1 \neq \mathcal{C}_{ep_n}$.

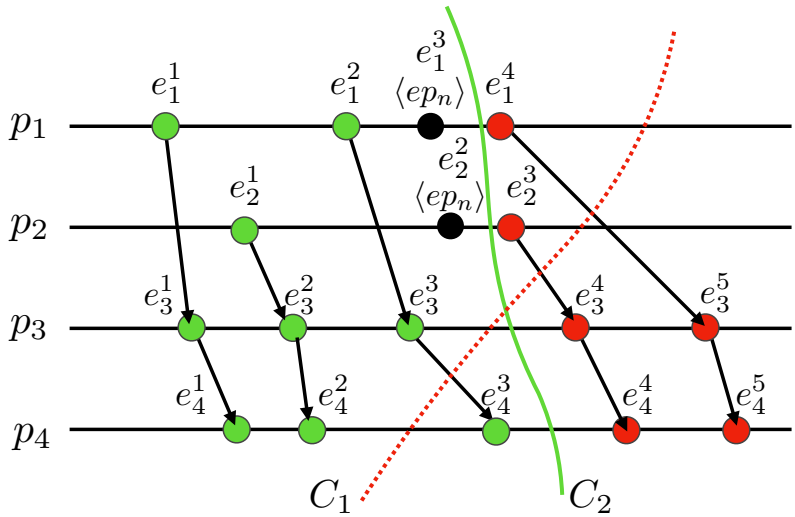


Figure 3.11: An example of two consistent cuts where $C_2 = \mathcal{C}_{ep_n}$.

3.4.3.2 Feasibility of Epoch Cuts

Evidently, epoch cuts are a quite strict form of consistent cuts since they impose constraints on the possible executions where they can be feasible. In Figure 3.12 we show another possible execution based on the process graph of Figure 3.1 as before. The problem in this execution is that no consistent cut can satisfy all invariants of Definition 3.4.2. For example, by invariant (1) and (4) it should be true that $e_3^4 \in \mathcal{C}_{ep_n}$ since $e_1^2 \prec_{p1} \langle ep_n \rangle$ and $e_1^2 \rightsquigarrow e_3^4$. However, this contradicts with $e_3^4 \notin \mathcal{C}_{ep_n}$ imposed by invariant (2) since $\langle ep_n \rangle \prec e_3^4$. (i.e., $\langle ep_n \rangle \prec e_2^3 \prec e_3^3 \prec e_3^4$).

Intuitively, in order to make any epoch cut feasible, an execution E should always respect event-ordering imposed by epochs. Thus, no event that is not part of an epoch should precede an event of that epoch. We call this execution property "Epoch Feasibility" and summarize it in Definition 3.4.3.

Definition 3.4.3. Epoch Feasibility: An execution E of a stream process graph satisfies Epoch Feasibility iff $\forall e, e' \in E : ((e \in \mathcal{C}_{ep_n}) \wedge (e' \notin \mathcal{C}_{ep_n})) \implies e' \not\prec e$.

3.4.4 Snapshots for Asynchronous Epoch Commit

If we can assure the feasibility of epoch cuts throughout a long-running continuous execution then it is possible to prepare and commit epochs asynchronously, as depicted in Figure 3.13. This can be enabled through the use of snapshots, though, we need a protocol that implements not just any consistent cut but epoch cuts.

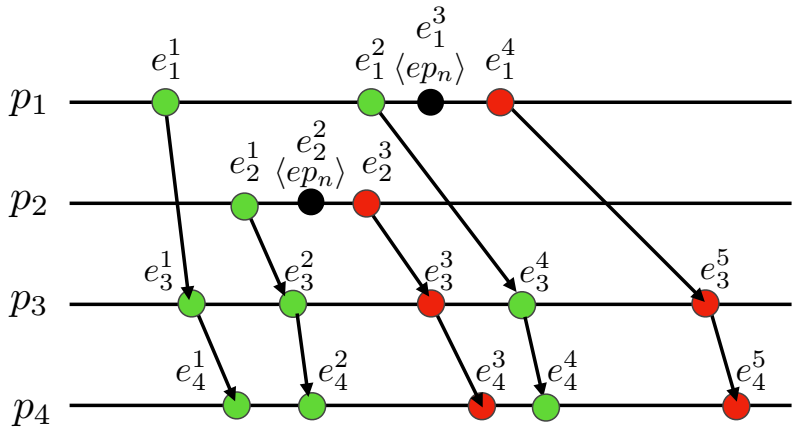


Figure 3.12: An example of an execution where $\mathcal{C}_{\mathrm{ep_n}}$ is infeasible.

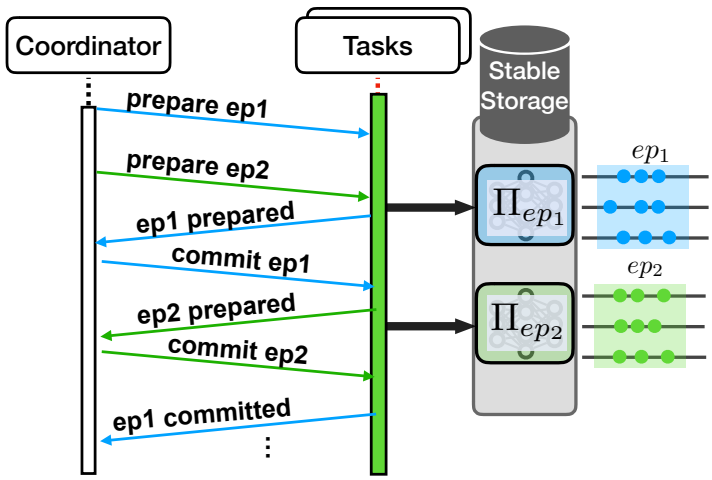


Figure 3.13: Asynchronously coordinated epochs with no idle times.

Specification 4 describes the expected behavior of "Epoch Snapshotting". Epoch snapshots are consistent snapshots that capture an *epoch-complete* configuration $S_{\mathcal{C}_{\mathrm{ep_n}}}$ in respect to an epoch cut $\mathcal{C}_{\mathrm{ep_n}}$. The specification essentially describes a recurring consistent snapshot that terminates per epoch in a continuous execution. When it comes to safety properties an epoch snapshot should satisfy *Validity* (identical to the consistent snapshot specification) as well as *Epoch Completeness*.

With epoch completeness we refer to the notion of capturing the (local) execution of an epoch ep_n in a process p, also denoted as $E_p^{ep_n}$. Given that every full epoch-complete snapshot implements an epoch cut, all message productions are included in the snapshot and therefore, no messages in transit are considered in a captured configuration. Mind that in this specification snapshots are not single-shot but recurring and should satisfy *Termination* if no failures occur up to the complete execution of an epoch.

Specification 4: Epoch Snapshotting (esnap)

Event Interface:

Indication: $\langle \text{record}|p, n, s_p^i \rangle$

Properties:

ESNAP1: *Termination*: $\Lambda = \Pi$ in $E^{ep_n} \implies \langle \text{record}|p, n, \_ \rangle \in E_p, \forall p \in \Pi$

ESNAP2: *Validity*: Configuration $S_{\mathcal{C}_{ep_n}} = \{\Pi_{ep_n}\}$ is valid

ESNAP3: *Epoch-Completeness*: Configuration $S_{\mathcal{C}_{ep_n}}$ is epoch-complete

Snapshotting Approach: The in-flight marker-based snapshotting mechanism by Chandy-Lamport that we analyzed previously in subsection 3.2.3 demonstrates how to capture a consistent snapshot concurrently to the execution without the need for blocking coordination. However, in the context of epoch-based processing it is not capable of offering epoch-cuts as is. In the rest of this section, we present step-by-step an alternative marker-forwarding protocol that satisfies validity (Theorem 3.2.2) similarly to Chandy Lamport while also guaranteeing epoch completeness. For better understanding, we break down our approach into three parts: 1) Snapshot Initiation, 2) Epoch Alignment and 3) Cyclic State.

3.4.4.1 I) Snapshot Initiation

The Termination property of esnap hints that the protocol has to eventually yield a snapshot per epoch. Therefore, we need a mechanism that initiates a snapshot per epoch and eventually terminates for every task in Π. In Theorem 3.2.6 we generalized the applicability of marker-forwarding snapshotting protocols to any weakly connected graph. More concretely, we have proven that the marker-based protocol terminates if initiated on a set of processes $\mathbb{A}$ that can in combination reach the full graph. In the context of stream process graphs, the set of source tasks satisfies this property. That means that an instance of the marker-based protocol would eventually reach all processes if executed on a set of initiator processes A.

Algorithm 5: Epoch-Based Snapshots (Sources)

Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel $(\mathbb{I}_p, \mathbb{O}_p)$
Algorithm:

1: $\mathbb{O}_p \leftarrow$ configured_channels;
2: $s_p \leftarrow \varnothing$; ▷ volatile local state

3: /* Source Task Logic */
4: **Upon** $\langle rcvd, m \rangle$
5: $\quad (s_p) \leftarrow process(s_p, m, \mathbb{O}_p)$;
6: **Upon** $\langle ep|n \rangle$
7: $\quad esnap \rightarrow \langle record|self, n, s_p \rangle$;
8: $\quad$ **foreach** $out \in \mathbb{O}_p$ **do**
9: $\quad\quad out \rightarrow \langle send, \odot_n \rangle$;

Given that every source task eventually becomes aware of an epoch change at the moment it processes an epoch event (e.g., $\langle ep_n \rangle$ for epoch n), it is natural to also initiate the protocol at that very instant. In algorithm 5 we summarize the epoch-based snapshot initiation logic for the source tasks. Mind that despite the fact that source tasks always follow a strictly deterministic order of events, they do have internal state. The internal state of the sources typically encapsulates the exact point in their deterministic execution. For example, if a source task reads messages from a logged message queue, its state is the read offset or pointer in that queue. Furthermore, we attach the epoch number to markers, in order to make the current epoch known to all receiving tasks and later simplify the collection of states based on their respective epoch that they were captured. Overall, given that source tasks have no input channels, the logic here is identical to that of the C-L algorithm.

3.4.4.2 II) Epoch Alignment

As we have concluded in Theorem 3.2.2, the marker-based "snapshot-and-forward" logic can guarantee validity. It remains to examine the necessary modifications needed to satisfy epoch completeness. In Figure 3.14 we depict a possible cut consistent and associated (valid) snapshot based on the previous example of an execution where epoch cuts are infeasible. Mind that message m is captured as part of the snapshot. Instead, an epoch-complete cut should incorporate all remaining productions $e_1^2 \rightsquigarrow \{e_3^4, e_4^4\}$. We can do so via a form of prioritization that we call "epoch alignment".

The epoch alignment mechanism allows tasks to complete all computation associated with an epoch before proceeding further in an orthogonal manner to

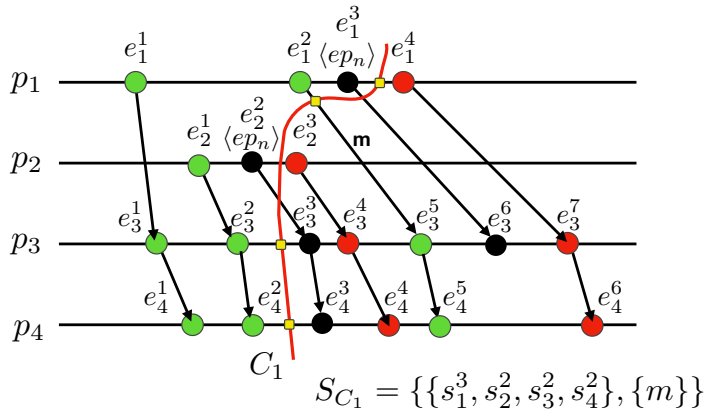


Figure 3.14: A consistent but not epoch-complete snapshot using C-L.

Algorithm 6: Epoch-Based Snapshots (Regular Tasks)

Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel $(\mathbb{I}_p, \mathbb{O}_p)$
Algorithm:

```
1:  (I_p, O_p) ← configured_channels;
2:  Enabled ← I_p ;
3:  s_p ← ∅;                                          ▷ volatile local state
---
4:  /* Common Task Logic                                                  */
5:  Upon ⟨rcvd, m⟩ on c ∈ Enabled
6:      s_p ← process(s_p, m, O_p);
7:  Upon ⟨rcvd, ⊙_n⟩ on c ∈ Enabled
8:      esnap → ⟨record|self, n, s_p⟩;
9:      Enabled ← Enabled/{c};
10:     if Enabled = ∅ then
11:         foreach out ∈ O_p do
12:             out → ⟨send, ⊙_n⟩;
13:         Enabled ← I_p ;
```

the marker-forwarding logic that guarantees validity. The goal is to turn any execution into a feasible one for epoch-cuts and therefore, obtain a snapshot that is both causally consistent (valid) and epoch-complete. In algorithm 6 we describe the complete logic of epoch alignment, as a minor modification of the core C-L protocol. Epoch markers are again disseminated throughout the graph, though,

with alignment we first make sure that all markers have been received along the inputs before capturing the task state and disseminating further.

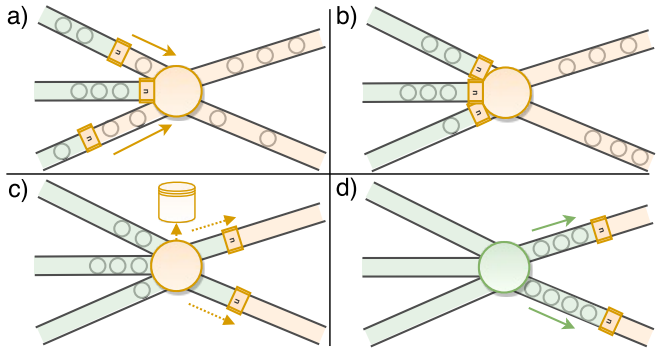


Figure 3.15: Alignment and Snapshotting Highlights.

Figure 3.15 depicts the steps prior to and during snapshotting in more detail. When a task receives a marker on one of its channels, it removes that channel from the "enabled" ones since all computation associated with the current epoch has to complete before continuing further (Figure 3.15(a)). Once markers have been received in all inputs (Figure 3.15(b)) the task can further capture its full state and notify downstream tasks based on the marker-forwarding logic that maintains causal consistency along the recording process of the global snapshot in the graph.

Example: In Figure 3.16 we visualize how epoch alignment can make epoch cuts feasible in the context of the execution visited previously. Alignment takes place in task p_3 once it processes the epoch marker in event e_3^3. At that point e_{23} is removed from the pending channels of p_3, prioritizing messages through e_{13}. Eventually, the last epoch marker is being processed in e_3^5 resulting into p_3 dropping alignment, snapshotting its state and propagating the epoch marker further to p_4. The resulting cut C_2 satisfies all properties of an epoch cut and thus $C_2 = \mathcal{C}_{ep_n}$.

3.4.4.3 Alignment vs Synchronous Epoch Commit

Epoch alignment effectively results into equivalent executions as the synchronous epoch-based approach (commit per epoch). However, in contrast to the synchronous approach, alignment does not hinder blocking synchronization since all tasks continue their regular operation, while prioritizing pending work to complete an epoch in a coordination-free fashion. The sole cost of alignment is limited to extra in-transit latencies for messages within non-enabled channels (measured further in chapter 4 section 4.4). In a typical push-pull messaging model that is employed in most stream processing systems, channel omission can lead, in the worst case, into disk spilling when allocated memory for network buffers reaches its limit. However,

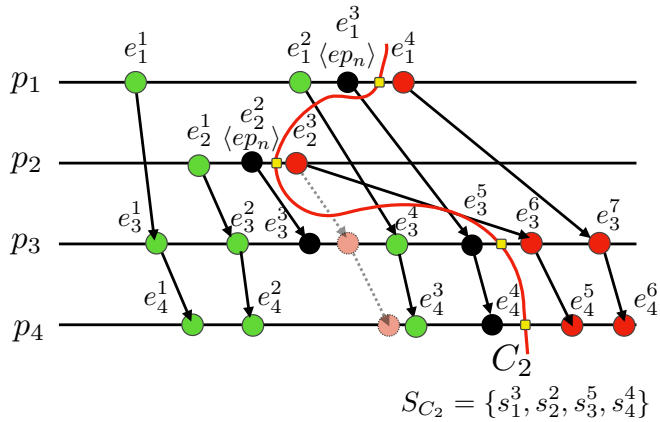


Figure 3.16: An epoch-complete snapshot using epoch alignment.

credit-based flow control can also be issued in order to avoid pushing messages on channels that are non-pending from receiving tasks.

3.4.4.4 III) Cyclic State

So far we have only considered the case where the stream process graph is a directed acyclic graph. However, it is often required to incorporate closed loops of tasks in stream processing graphs (e.g., in machine learning and graph processing applications). We identify two main challenges related to cyclic graphs : 1) A production tree within a cycle is not guaranteed to be bounded and thus, it can potentially progress infinitely. That would in turn mean that an epoch is not guaranteed to complete. 2) Epoch alignment deadlock as-is if there existed a loop in the graph. That is due to the fact that the protocol progresses only if all preceding computation on an epoch has completed. Hence, a circular dependency would make such a condition unsatisfiable.

Given that explicit loops disallow epoch-based execution we can only deal with such issues indirectly. Consider a process graph with cycles, such as the one depicted in Figure 3.17. Loops are formed by strongly connected components in a stream process graph. Conceptually, if we collapse each strongly connected component into a single task (e.g. loops A and B in Figure 3.17) we get a directed acyclic graph. Assuming that we are able to employ any epoch-based execution in that conceptual level, the complete state of Loop A and B at the pass of an epoch would in fact be their in-progress computation, a combination of internal states and in-transit messages. This concept encapsulates our approach to dealing with loops which we describe in a methodology comprising of two steps: 1) Back-Edge Identification and 2) Loop Expansion, as described below. Our methodology is general and it can be used to break loops in any strongly connected component.

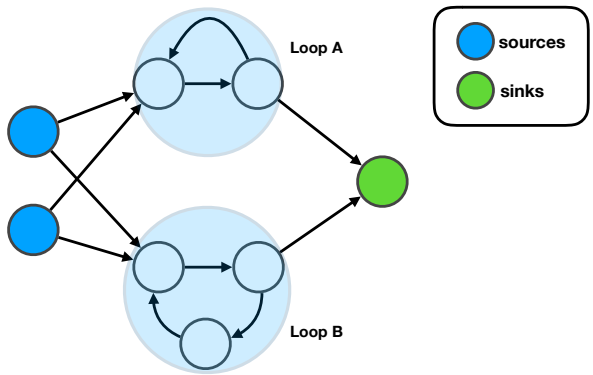


Figure 3.17: An example of a process graph with two loops.

Back-edge Identification: In the best case, loops can be explicitly defined at a higher level programming model (see chapter 6). However, there also exist compositional dataflow programming systems that allow loops to be composed arbitrarily (e.g., Apache Storm [38]), similarly to go-to statements in imperative programming. In the most general case each loop can be inferred using a typical strongly connected component detection algorithm [75] in a depth-first traversal of the process graph at compilation time. Given a set of tasks per loop it is then possible to define an entry task based on the highest dominance value [76]. Typically, in stream process graphs the entry task corresponds to the one that maintains the most input edges external to the loop (since all stream flow passes through that node). An edge (channel) within the cycle that points to the entry task is also known as a back-edge. In Figure 3.18(a) we highlight in red each selected back-edge per loop.

Loop Expansion: Given that we all back-edges are identified, the next step is to eliminate cycles via a process we call "Loop Expansion". We replace each back-edge with two tasks, a source task called "Loop Head" and a sink task called "Loop Tail". Each pair of Head and Tail is interconnected with a hidden channel that we call phantom channel. All messages that traverse the phantom channel are processed by the Head as regular input records. This way our graph is a directed acyclic graph and can respect epochs since each Loop Head is able, as every other regular source to process epoch events and therefore determine which part of the in-transit (cyclic) computation belongs to an epoch. This approach works seamlessly for nested loops as well since every cycle will be transformed into an acyclic graph with a source and a sink, eliminating any cyclic dependencies and allowing epoch-based progress.

The Loop Head tasks incorporate our final variant of the epoch-based snapshotting protocol, summarized in algorithm 7 which guarantees to capture the current state of each loop (messages in transit) at the moment an epoch changes. The overall logic is identical to Chandy-Lamport snapshots since all in-transit messages are

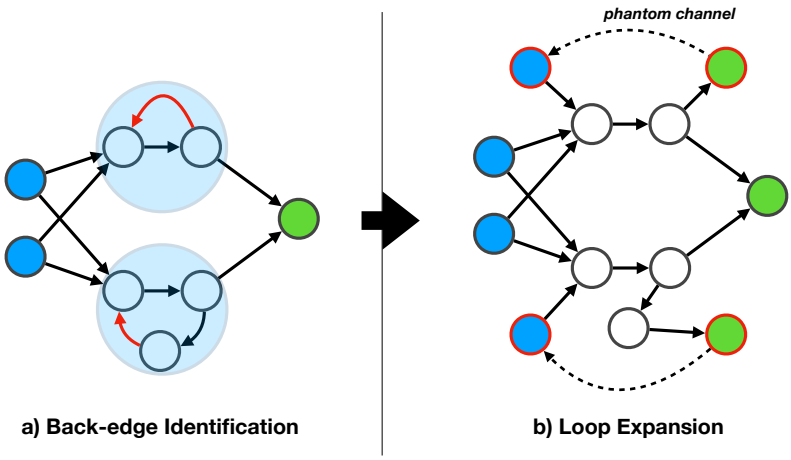


Figure 3.18: Process Graph Transformation Steps for Loops

recorded up to the completion of an epoch. However, we only apply this logic at Loop Head nodes, where that functionality is necessary in the context of a strongly connected component.

We visualize each step of this algorithm variant in Figure 3.19, starting at the instant that a loop head task receives an epoch change event. As depicted in Figure 3.19(a)), markers are first inserted within the respected cycle through the head task. From that point until a marker is processed back at the head (after traversing the whole cycle) all messages forwarded at the head through the phantom channel are being also logged into the head's snapshotted state since they precede the marker and therefore belong to preceding epochs (similar to the C-L algorithm). Once the marker is received back at the head (Figure 3.19(c)) its message log is committed as part of its internal state and the protocol terminates (Figure 3.19(d)).

3.4.5 Analysis of Asynchronous Epoch Snapshots

The epoch-based snapshotting protocol presented inherits the core invariants of the C-L protocol when it comes to termination and validity, while also satisfying epoch completeness. In this section, we will examine these properties and prove them in the context of epoch-based stream processing.

3.4.5.1 Termination

In section 3.2.4.2 we have proven the termination property of the marker-forwarding logic in the C-L protocol which we further generalized to weakly connected graphs with multiple initiators in Definition 3.2.6. According to these observations, any

Algorithm 7: Epoch-Based Snapshots (Loop Heads)

Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel ($\mathbb{O}_p$)
Algorithm:

1: $(\mathbb{O}_p) \leftarrow$ configured_channels;
2: recording $\leftarrow$ false;
3: $s_p \leftarrow \emptyset$; ▷ logged in-transit state

4: /* Loop Head Logic */
5: **Upon** $\langle ep|n \rangle$
6: recording $\leftarrow$ true ;
7: **foreach** out $\in \mathbb{O}_p$ **do**
8: out $\rightarrow \langle send, \odot_n \rangle$;
9: **Upon** $\langle rcvd, \odot_n \rangle$ *on phantom_channel*
10: esnap $\rightarrow \langle record|self, n, s_p \rangle$;
11: $s_p \leftarrow \emptyset$;
12: **Upon** $\langle rcvd, m \rangle$ *on phantom_channel*
if recording **then**
$s_p \leftarrow s_p \cup m$;
foreach out $\in \mathbb{O}_p$ **do**
13: out $\rightarrow \langle send, m \rangle$;

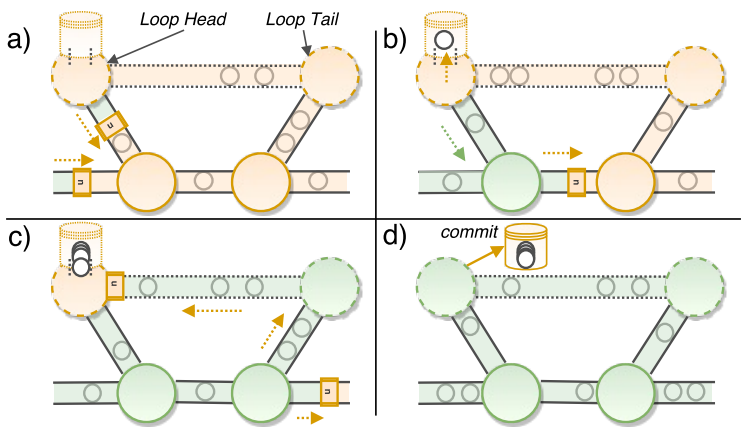


Figure 3.19: Cycle Snapshotting Highlights.

marker-forwarding protocol for snapshotting can terminate if a marker reaches all channels in a process graph. The alignment logic that we have added for epoch completeness affects only prioritization across channels and does not alter the fact that a marker will be in fact delivered across a channel if it has been previously sent (including the phantom channel we have introduced in loops).

Proof. It suffices to show that the following two conditions are satisfied:

1 There exists a set of initiating processes $\mathbb{A} \subseteq \Pi$:
$\Pi/\mathbb{A} \subseteq \{q \in \Pi | \exists p(p \in \mathbb{A} \wedge p \sim q)\}$

2 The protocol is instantiated on every initiating process per epoch.

Condition 1. is satisfied by the properties of a stream process graph itself, including the transformations we introduced for strongly connected components. Effectively, the set of regular sources and the loop heads satisfy the necessary reachability condition.

Condition 2. is satisfied by Algorithms 5 and 7 that describe the initiating logic in both regular sources and loop heads. Given that an epoch event is guaranteed to be received during an epoch change $\langle ep|i\rangle \in E_p, \forall p \in \mathbb{A}$, the algorithm proceeds with the regular marker-forwarding logic within the same action. Furthermore, based that monotonicity property of epoch events given in Definition 3.4.1 it is guaranteed that the protocol will be initiated and terminated in strict epoch order.

Finally, we should add that similar to the C-L algorithm, the termination of an instance of the protocol can only be guaranteed if all of the participating processes are correct during the system execution, up to its completion. □

3.4.5.2 Validity

Despite the addition of the alignment mechanism, epoch-based snapshots preserve the same marker-forwarding logic of C-L according to Theorem 3.2.2 and also rely on FIFO reliable channels to guarantee validity.

Proof. Each local snapshotting action $e_q^{\odot}$ in a process $q \in \Pi$ is causally related to the snapshotting action $e_p^{\odot}$ of an adjacent process p in a stream process graph (i.e., $e_p^{\odot} \prec e_q^{\odot}$) via the the forwarding of a special marker $\odot$ included in that action that separates all events the precede and follow a specific epoch cut. In Section 3.2.4.3, we have proven that this principle is equivalent to maintaining causal consistency during snapshot acquisition (summarized in Theorem 3.2.2) and therefore it is guaranteed that the final configuration of each epoch snapshot is valid. □

3.4.5.3 Epoch Completeness

Epoch-completeness is satisfied via the the special initiation of the protocol based on epoch events as well as the alignment mechanism. In fact, the action of an epoch change at a source task corresponds to a marker-based snapshot initiation. Then alignment makes sure that all pending productions of an epoch, relevant to a task, are being processed before any message that belongs to a succeeding epoch.

Proof. There are three invariants associated with epoch-completeness that have to be satisfied by a snapshot $S_{\mathcal{C}_{ep_i}}$ of an epoch ep_i .

Inv. 1. $(\langle ep_i \rangle \prec e) \implies (e \notin \mathcal{C}_{ep_i})$

Inv. 2. $(e \prec \langle ep_i \rangle) \implies (e \in \mathcal{C}_{ep_i})$

Inv. 3. $((e \in \mathcal{C}_{ep_i}) \wedge (e \rightsquigarrow e')) \implies (e' \in \mathcal{C}_{ep_i})$

Proof [Inv. 2]: Given that epoch change events occur solely on source tasks, and no event at a source task is causally dependent on events occurring on other processes, the condition $e \prec \langle ep|i \rangle$ applies only to local event order at a source. Therefore, we only need to prove that for each source task $p \in \mathbb{A}, (e \leq_p \langle ep|i \rangle) \implies (e \leq_p e_p^{\odot})$. Based on the source task logic described in algorithm 5, the snapshotting action $e_p^{\odot}$ occurs at the reception of an epoch marker and thus, $e_p^{\odot} = \langle ep|i \rangle$. It therefore derives directly that invariant 2 is always satisfied. Note that the same epoch change occurs on Loop Head source tasks. The only difference in the case of Loop Heads is the in-transit logging of messages in a loop which correspond to pending productions of the currently snapshotting epoch (until the marker arrives back via the phantom channel), thus it is guaranteed that $(e \prec \langle ep_i \rangle) \implies (e \in \mathcal{C}_{ep_i})$ even in that case.

Proof [Inv.1+3]: Inv.1 refers to the exclusion of events corresponding to messages after an epoch change, while, Invariant 3 refers to the inclusion of all events caused by messages preceding an epoch change. We need to prove that both are satisfied on every task in the system. Given the algorithm variants we further separate the cases of source and regular tasks.

I) Source Tasks$(p \in \mathbb{A})$: Given the strict local execution order and the fact that $e_p^{\odot} = \langle ep|i \rangle$, inv.1 is always satisfied. Furthermore, Inv.3 considers event productions and hence it can never be violated for source tasks given that they don't consume input messages.

II) Regular Tasks$(p \in \Pi/\mathbb{A})$: The epoch alignment mechanism of algorithm 6 with FIFO channels suffice to satisfy both. According to epoch alignment, an input channel c_{pq} of a task $q \in \Pi$ becomes disabled from the time a marker arrives in a channel until the last marker is received and the process p executes the snapshotting

and marker-forwarding action. For brevity, we will refer to those two events as $e_q^{\odot pq}$ and $e_q^{\odot}$ respectively and summarize the alignment condition in 3.11.

Alignment Condition: Given adjacent processes $p, q \in \Pi/\mathbb{A}$ the alignment mechanism enforces the following condition:

$$\nexists e_q = \langle proc, m_{pq}, M \rangle_q \in E_q : e_q^{\odot pq} \leq_q e_q \leq_q e_q^{\odot} \tag{3.11}$$

Via (3.11) the proof of Invariants 1 and 3 follows a similar logic: For Inv.1, assume by contradiction that $\langle ep|i \rangle \prec e_q \wedge e_q \in \mathcal{C}_{epi_i}$. Based on the marker forwarding logic in a channel, let that be c_{pq}, it is known that $e_q^{\odot pq} \prec e_q$, since $e_q = \langle proc, m_{pq}, M \rangle$ (1). From (1) and (3.11) it can only be true that $e_q^{\odot pq} \leq_q e_q^{\odot} \leq_q e_q$ and thus, $e_q \notin \mathcal{C}_{epi}$ which leads to a contradiction $\Rightarrow\!\!\!\!\!\!\not\Leftarrow$.

Similarly, for invariant 3, given an event $e_p \in E_p$ and the production $e_p \rightsquigarrow e_q$, let us assume by contradiction the following: $(e_p \in \mathcal{C}_{epi}) \wedge (e_q \notin \mathcal{C}_{epi})$. Since $e_p \in \mathcal{C}_{epi}$, that event precedes the snapshotting event on process p: $e_p \leq_p e_p^{\odot}$ (2). Both of these events create productions in process q: $e_p \rightsquigarrow e_q$ and $e_p^{\odot} \rightsquigarrow e_q^{\odot pq}$. Via FIFO delivery (FIFORC4) we can maintain the delivery order of (2) to their productions from p to q given the fifo property of c_{pq}. Therefore, $e_q \leq_q e_q^{\odot pq}$ (3). However by applying the epoch alignment condition (3.11) to (3) we have $e_q \leq_q e_q^{\odot}$ and finally arrive to contradiction $e_q \in \mathcal{C}_{epi} \wedge e_q \notin \mathcal{C}_{epi} \Rightarrow\!\!\!\!\!\!\not\Leftarrow$.

We thus know that Inv.3 is satisfied for a single production. Via the transitive property of productions the proof extends trivially to arbitrary production trees via induction on the path of channels from p to q where $p \rightsquigarrow q$.

□

3.5 Summary

In this chapter, we introduced the fundamental execution model primitives and specifications of reliable data stream processing. Our analysis addresses a set of universal challenges when it comes to reliable stream processing, including the ability to recover from failures, reconfigure a continuous stream processing execution as well as to version and identify side effects of a distributed, event-based stream computation in a causally consistent manner. To that end, we proposed the Epoch-Based Stream Processing model according to which a stream computation is divided into a series of epochs that atomically commit the state of an execution. Furthermore, we showed how the epoch commit protocol can operate asynchronously to the stream computation via epoch-based snapshotting, a stricter form of a causally consistent snapshot that extracts a valid system configuration while respecting epoch order, a necessary property in our execution model. Our proposed marker-based mechanism can work on any weakly connected, static graph of processes with optional cycles and it has been proven to satisfy all necessary safety

and liveness properties of epoch-based snapshotting. Given that this chapter focuses solely on the fundamentals, we skip the discussion on design principles and instead summarize the complete analysis in chapter 4. The integration of asynchronous epoch commits in Apache Flink further showcases how this underlying execution model can support complex operations in a widely deployed, production-grade scalable stream processing system.

CHAPTER

4

State Management in Flink

As presented previously in chapter 3, asynchronous epoch commits via snapshotting suffice to offer transactional processing guarantees without blocking synchronization and coordination in the context of distributed data streaming. This chapter presents in more detail a concrete adaptation of these techniques within Apache Flink's end-to-end runtime. We describe how the principles we introduced work in practice and drive stateful processing inside and outside the Flink system. Flink's runtime can support continuous long-running task executions without interruptions, while allowing side effects of epochs to commit asynchronously, outside the critical path of the computation. Aside the core snapshotting mechanism, many other related operational mechanisms are made transparent to the user, posing no restrictions on Flink's expressive programming model and thus allowing arbitrary operations on partitioned state.

The outline of the chapter goes as follows: section 4.1 describes the internal runtime components of Flink that participate in the state aquisition mechanism and the concrete end-to-end epoch commit protocol as well as its interactions with various backends developed by the Flink community. Then, section 4.2 [1] offers an in-depth overview of Flink's reconfiguration mechanism and related choices regarding repartitionable state in the system. Section 4.3 introduces the concept of external query isolation and application execution provenance tracking, two useful operations that build on epoch commits. Then, section 4.4 presents the costs and benefits of epoch-based snapshots based on production statistics gathered during a long large-scale deployment of Flink. Finally, section 4.5 discusses existing and potential optimisations such as incremental snapshots and dynamic reconfiguration, followed by acknowledgements in section 4.7 and a summary in section 4.8 covering the use of our core design principles.

[1] Sections 4.2 and 4.4 include previously-published content [29] as-is.

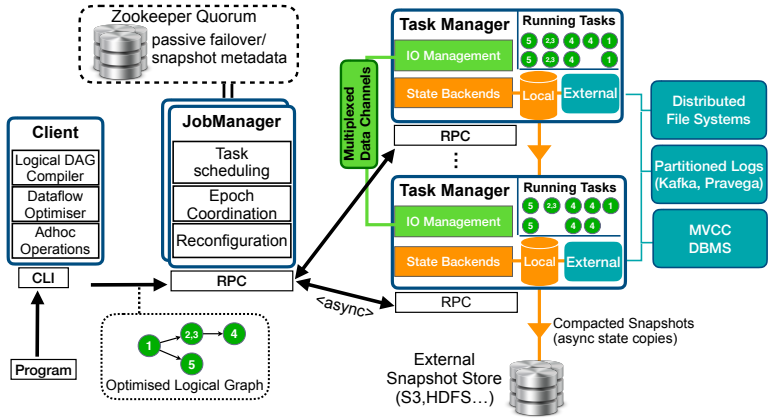


Figure 4.1: A Detailed Overview of Flink's Runtime.

4.1 Asynchronous Epoch Commit Integration

Flink's distributed runtime has a master-slave architecture similar to Spark and Hadoop MapReduce. Yet, in contrast to batch-centric application management [12] that builds on staged, short-running computation, Flink employs a schedule-once, long-running allocation of tasks. Nevertheless, through the use of epochs and corresponding state backends that abstract operations on state the system is able reconfigure applications (e.g., scale-out) and re-allocate application state on-demand while offering reliable processing guarantees (via atomic epoch commits). This approach minimizes management overhead while allowing for further adaptation to hardware or software changes or partial failures that can potentially occur. We first explain the overall design choices and then focus on each respective mechanism related to epoch-based snapshots in more detail.

4.1.1 Architecture Breakdown

In Figure 4.1 we provide an overview of all Flink runtime components and their purpose, which we describe in more detail below.

Client: The client library provides support for static type checking and compilation of Flink programs. In chapter 2 we sketched most operational semantics of stream operations from the programmer's perspective. An operation corresponds to a unary (e.g., map, filter, fold, window) or n-ary higher-order function (e.g., join, co-map, co-flatmap) and is parametrized with user-defined function literals. The Flink client compiles operations to a logical graph of tasks, encapsulating each independent application component and its data dependencies to other components.

A task in the final optimised graph corresponds to a single or multiple (chained) logical operators compiled by Flink's graph generator and optimiser (see chapter 2). The client provides a command line interface (CLI) to compile, submit, monitor and reconfigure running applications and thus serves as a complete interface to Flink's distributed runtime.

JobManager: The JobManager is a JVM process that maintains a global view and control of each running application, including corresponding tasks and their locally managed states. The role of the JobManager is crucial to epoch-based execution since it is the intermediary that triggers epoch-change events, collects recorded snapshots and notifies back tasks about completed epochs, as part of the asynchronous epoch commit protocol. Furthermore, it employs heartbeat-based failure-detection and monitoring of all Flink cluster resources. Every action of the Job Manager that alters the metadata of an application (e.g., scheduling, reconfiguration, epoch completion etc.) is first committed in a Zookeeper quorum [77]. This allows for passive-standby deployments that can guarantee a consistent system execution which can deal with all types of failures, including master node failures. All communication with the client and the local workers (TaskManagers) respects an asynchronous RPC-based protocol that does not interfere with data channels used by the application tasks.

TaskManager: Each logical task of an application is scheduled and physically executed in parallel across multiple workers. Physical tasks are granted access to two major resources: network channels and state. The overall resources available in a worker should be shared efficiently and in isolation by multiple physical tasks (typically assigned to dedicated containers using YARN [48] or Mesos [49]). This is the work of the TaskManager JVM process, which also serves as the main proxy between physical tasks and the JobManager. JobManager processes are stateless and employ policy-based networking and state access while executing requests received from the JobManager. Data channels between task threads are multiplexed into shared TCP connections with shared buffers for serialization/deserialization needs and deadlocks are typically avoided through in-flight flow control. State backends are a modular way to make state operations transparent to the physical state's location and representation. Flink supports plug-in backends for locally embedded databases (e.g., RocksDB), external partitioned logs (Kafka, Pravega, Kinesis etc.) and external multiversion concurrency control-enabled (mvcc) databases without requiring any changes in the user program. Most importantly, physical state is almost always kept outside the heap and managed externally to the JVM for scalability and performance (no garbage collection).

4.1.2 Task Design and Process Model

Physical tasks in Flink adopt the stream process model presented in subsubsection 3.3.1.1 and therefore follow a strict message-based control flow, manipulating

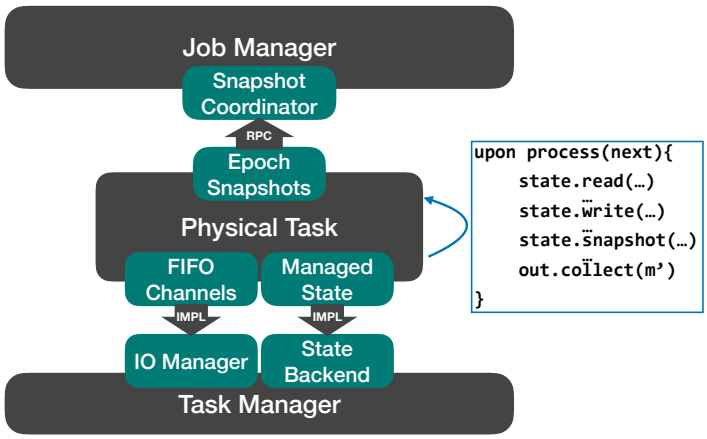


Figure 4.2: Component Design of Physical Tasks.

state and generating output records within an atomic action triggered by an input message. In an epoch-based execution with reliable processing guarantees no indirect communication or external connections are supported in that critical path. However, external asynchronous communication mechanisms are used for auxiliary purposes such as the epoch commit protocol presented later.

In Figure 4.2 we depict how physical tasks are modeled from a component design point of view. The logic within a physical task gets invoked per input message received from one of its FIFO channels. A task can read, write or issue snapshotting operations on its managed state as well as send output messages through a collector interface ("collect" flushes messages downstream). Both implementations of FIFO channels and managed state are provided by the TaskManager. Furthermore, at the end of a complete epoch, each task provides a reference of its snapshotted state which is collected via asynchronous RPC issued by the Job Manager. For convenience, the channel prioritization logic required by the alignment phase of the epoch-based protocol is implemented by the FIFO channels provided by the IOManager component.

4.1.3 Protocol Implementation

Flink makes use of the Asynchronous Epoch Commit protocol (see chapter 3 section 3.3) , guaranteeing that all events in an execution up to an epoch and their internal and external state operations are atomically committed to stable storage. The implemented protocol includes all asynchronous communication steps between a Snapshot Coordinator process (JobManager), Physical Tasks (TaskManager) and respective state backends. An instance of the protocol runs per epoch change and is initiated by the coordinator while supporting concurrent instances of the protocol.

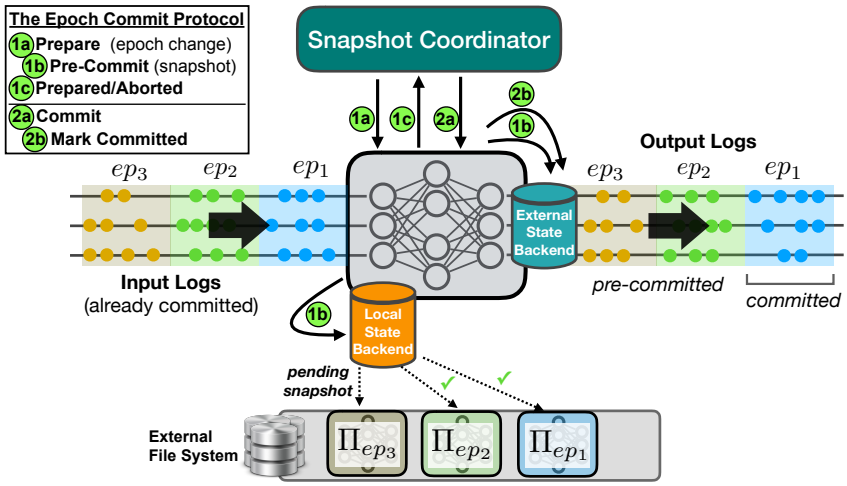


Figure 4.3: Overview of Epoch Commit with Snapshots.

If an instance gets aborted (e.g., due to a partial failure) the system rolls-back all pre-committed changes and the execution restarts from the latest committed epoch. Most importantly, the epoch commit protocol is being executed concurrently with the physical task execution and does not influence the critical part of the computation. In Figure 4.3 we visualize each communication step of the epoch commit protocol and further explain each step below.

Prepare Phase: The prepare phase starts once the snapshot coordinator issues an epoch change. This can occur periodically (e.g., every 20 seconds) or as an adhoc request from the user. In both cases, the coordinator broadcasts an epoch change message (1a) to all TaskManager nodes which in turn initiates the epoch-based snapshotting protocol described in section 3.4 at all source tasks including marker dissemination and epoch alignment. Eventually, if no failures occur, every physical task triggers a local snapshot on its managed state and an acknowledgment is sent back to the coordinator (record notification). We call each local snapshotting operation a pre-commit step (1b). In the case of operators with a local state backend the pre-commit is a copy operation of the state to an external file system. However, in the case of an external state backend, a pre-commit locks remote changes to allow for no further operations and to prepare it for the final commit. Nevertheless, neither the snapshotted states nor the externally pre-committed states should be accessible at this point since the epoch is not yet committed. The prepare phase ends once all tasks have notified the coordinator with a "prepared" acknowledgment (1c). That can only happen after the snapshotting algorithm has finished. If a partial error (abort message) or global timeout occurs during the process the protocol

aborts.

Commit Phase: The commit phase begins once all "Prepared" commands are received by the coordinator. The purpose of the commit phase is to confirm all pre-commited states for external access. The Snapshot Coordinator initiates the commit phase by broadcasting a "Commit Epoch" message to all tasks (2a) which in turn commit pending epochs at the backends (2b). This action can be invoked concurrently to the task execution by the TaskManager, thus, inducing no performance impact on the critical execution path. The commit phase is especially important for external state backends in order to make all external changes visible to the outside (e.g., pre-committed output streams). Failures can potentially occur during this phase which can lead to incomplete committed changes in an epoch. However, this does not violate any system guarantees. Given that all changes are at least guaranteed to be pre-committed at this point, pending commits can be eventually re-issued during the rollback mechanism (explained below) to finalize the process.

Summary: Epoch commit is a special-purpose two-phase commit protocol that provides transactional processing guarantees. The usage of asynchronous epoch-based snapshots for pre-committing all side-effects yields several important observations. First, no commit operation affects the runtime performance (i.e., throughput) of the system. The commit protocol simply makes all side effects of an epoch externally visible once it is guaranteed that everything is stored to some form of stable storage. Second, it allows pipelining of multiple concurrent epoch commit instances with the use of asynchronous epoch-based snapshots. In the example of Figure 4.3, we can see the case of three possibly concurrent epochs. For epoch ep_1 all side effects are committed while ep_2 has been pre-committed and the commit phase is pending. Finally, ep_3 is in the pre-commit phase which means that the snapshotting algorithms is being currently executed. In the same example, it is safe to rollback from ep_2 given that all states are stored to stable storage at that instant.

4.1.3.1 The Rollback Procedure

Flink's rollback mechanism is initiated either when a partial failure gets detected during normal operation (fail-stop model), or when reconfiguration is requested or upon an aborted epoch commit instance. Rollback respects a "stop, reschedule and restore" procedure whether the reason is failure recovery or reconfiguration (see Figure 4.4). In all cases, the system (JobManager) picks the latest prepared (but not necessarily committed) snapshot to restart an execution from. Eventually, all states within the pipeline are progressively retrieved and applied to reflect an exact, valid distributed execution at the restored epoch. Below, we identify several special cases of a task rollback:

Loop Head Tasks: In the case of Loop Head tasks, all records logged during

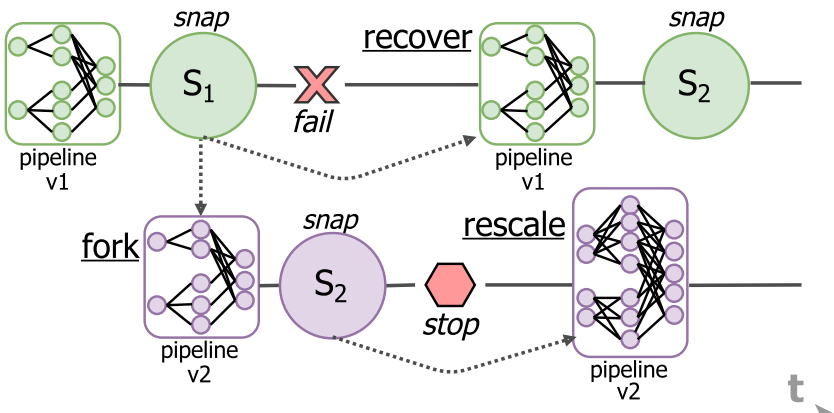


Figure 4.4: Rollback examples.

the snapshot are recovered and flushed to output channels prior to their regular record-forwarding logic. This way, it is guaranteed that the state of the loop returns back to its snapshotted execution (that is, the sum of the in-transit records).

State in External Backends: In order to circumvent the case of permanently uncommitted external states, all external state backends issue a preemptive commit in the beginning of a rollback in order to guarantee that no uncommitted changes will persist past the recovered execution point.

Regular Sources: All data sources need to restore their (deterministic) execution back to the current offset of each stream when 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).

Selective Rollback: Depending on the rollback cause, certain optimized recovery schemes can be 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, if more than one connected components exist.

In essence, known incremental recovery techniques from micro-batch processing [12] are orthogonal to Flink's rollback 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 shared of state back to their respective backends, from

	Start	Pre-Commit	Commit	Abort
Local (RocksdB)	new MemTable	flush MemTable / Snapshot SSTAbles	-	delete snapshot
Local (Heap)	-	deep copy (full snapshot)	-	delete snapshot
Kafka $\geq$0.11	new transaction	close/new transaction	commit transaction	abort transaction
Pravega	new Segment	seal Segment	commit Segment	delete Segment
HDFS	create File in tmp dir	close File (no writes)	move (atomic) file to committed Dir	truncate/delete file
DBMS (non-MVCC)	new WAL	snapshot WAL	execute WAL as transaction	drop WAL
DBMS (MVCC)	-	new version	incr version	decr version

Table 4.1: Examples of Backend-Native Operation Mappings to Flink's Epoch Commit Protocol

externally stored snapshots.

4.1.4 Backend Integration

Several backend implementations have been contributed to Flink by its developer community which integrate seamlessly with Flink Epoch Commit protocol. In Table 4.1 we summarize some of the existing and possible backends and the required operations to integrate with Flink's Epoch Commit. In general, each backend supports four operations: 1) Start, 2) Pre-Commit, 3) Commit and 4) Abort. For several backends that allow some form of transactional writes the operational needs are satisfied out of the box. These include recent versions of partitioned logs such as Kafka ($\geq$ 0.11) and Pravega as well as DBMSs with multi-version concurrency control. For example Apache Kafka 0.11 introduced support for exactly-once delivery for producers. Flink's Kafka 0.11 transactional sink basically issues a new transaction per epoch which can be committed atomically by Kafka itself. Pravega is another example of a newer system which comes with built-in support for transactional cross-partition writes already in its model. Pravega's segments represent self-contained partitioned logs which can be atomically started and sealed, integrating tightly with Flink's epoch-based processing since a segment per epoch can be started, sealed (committed) and deleted in case of an epoch abort.

In the rest of the cases, the implementation of two-phase commit has to be solved indirectly by the developer of the backend. Examples of non-transactional backends are non-MVCC databases and File Systems. For example, the HDFS transactional sinks in Apache Flink persist all state append operations to temporary files and rely on HDFS truncate to abort an epoch and atomic move operations to move a closed HDFS file of an epoch to a "read-committed" directory. In the case of DBMSs or other external store systems a general strategy that works at an additional latency and storage cost is to maintain a Write-Ahead-Log (WAL) for all uncommitted external operations in local state. Upon an epoch commit phase (or recovery) the WAL can be executed and get discarded from the local state.

In general local backends are needed to complement the snapshotting of all metadata needed to persist external two-phase commits. For example, WALs, log offsets, transaction IDs etc. are important metadata that requires bookkeeping to eventually finalize any pending asynchronous message-exchange protocols that have been initiated with external systems.

4.2 System Reconfiguration

Asynchronous epoch snapshots and the rollback procedure cover the needs of reconfiguration but only in part. A typical need in any data-intensive application deployment is to be able to modify the scale (i.e., parallelism) of certain logical tasks. For tasks that have declared managed state we need to consistently allocate data

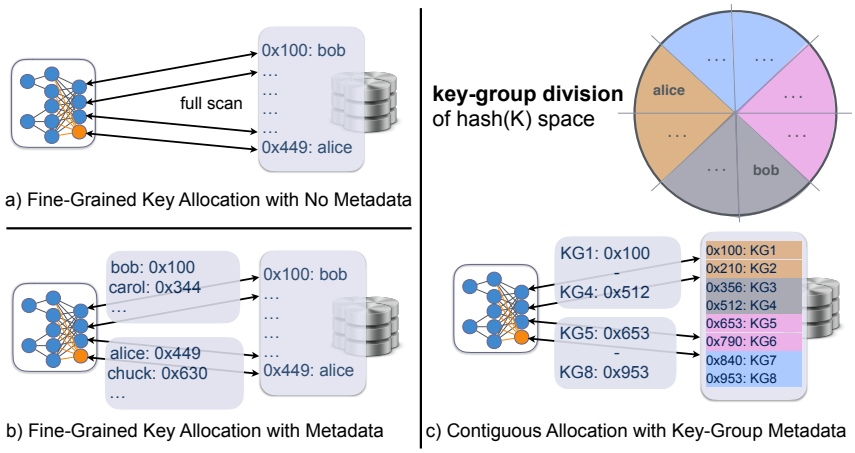


Figure 4.5: State Allocation and Metadata Alternatives

stream partitions and further re-allocate in the case of reconfiguration. In chapter 2 we covered the different managed state representations in Apache Flink. In fact, most scalable operations and respective state are scoped by a user-defined key with a key space K. For load balancing, both streams and respective states are sharded in the space of a consistent hashing function $h : K \rightarrow \mathbb{N}^+$.

4.2.1 Key-Group Partitioning and Allocation

Flink decouples key-space partitioning and state allocation similarly to Dynamo[78]. The runtime maps keys to an intermediate circular hash space of "key-groups" : $K^* \subset \mathbb{N}^+$ given a maximum parallelism π-max and a hash function h as such:
$K^* = \{h(k) \bmod \pi\text{-max} \mid k \in K, \pi\text{-max} \in \mathbb{N}^+, h : K \rightarrow \mathbb{N}^+\}$

Given that snapshots should contain all information needed to find and re-allocate state, there is an evident trade off between the overhead of a rollback (I/O during state scans) and snapshot metadata needed to re-allocate state to different numbers of instances. 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 (Figure 4.5(a)). On the opposite extreme, snapshots could contain references to every single key-value and each task could selectively access its assigned keyed states (Figure 4.5(b)). However, this approach increases indexing costs (proportional to num. of distinct keys) and communication overhead for multiple remote state reads, thus, not benefiting by coarse-grained state reads. Key-groups (Figure 4.5(c)) offer a substantial compromise: reads are limited to data that is required and key-groups are typically large enough for coarse grained reading (if π-max is set appropriately low). In the uncommon case where

$|K| < \pi$-max it is possible that some task instances simply receive no state. Finally, 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.

4.2.2 State Re-Allocation

To re-assign state, we employ an equal-sized key-group range allocation. For π parallel instances, each instance $t^i \in \Pi, 0 \leq i \leq \pi$ 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.

4.3 Operations with Epoch Snapshots

Fault tolerance and reconfiguration are only a subset of the potential needs and benefits covered by epoch-based snapshotting. In this section, we introduce another two novel use-case examples of snapshots, namely external access isolation guarantees for managed state and application provenance both of which have been examined and prototyped in Apache Flink.

4.3.1 External Access Isolation

Flink allows direct adhoc queries to its managed state from outside the system. This way external systems or users 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 can be accessed via a subscription-based API. First, managed state that allows for query access is declared in the original application. Upon state declaration (see chapter 2 section 2.1) 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:

```
1 //stream processing application logic
2 val descriptor: ValueStateDescriptor[MySchema] = ...
```

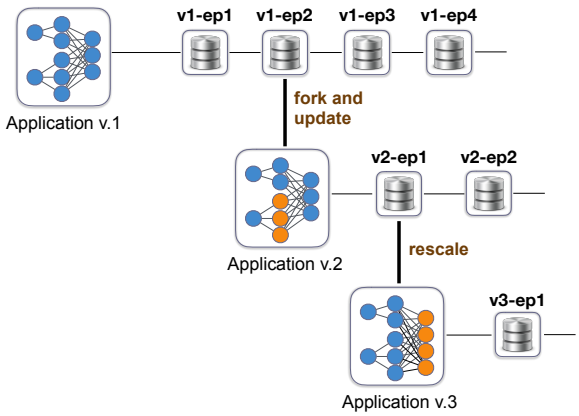


Figure 4.6: Application Provenance with Snapshots

```
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 database query isolation-level viewpoint, such queries access *uncommitted state*, thus following the *read-uncommitted* isolation level. However, via the use of snapshots it is possible to offer *read-committed* isolation support by letting TaskManagers hold onto the state of committed snapshots, and use that state to execute adhoc queries.

4.3.2 Application Provenance and Migration

Epoch-based executions make application provenance possible and more importantly, trivial. This is due to the fact that snapshots mark a clear dependency chain between epochs and reconfiguration actions applied throughout the history of a long-running stream processing application. As shown in Figure 4.6, an application execution dependency diagram resembles that of a version control system (e.g., git), where its distinct change is encapsulated into epoch snapshots. Furthermore, an application can have multiple versions, forked from existing snapshots that are being executed in different configurations (e.g., different parallelism, cluster, logic etc.). This can further ease the development and maintenance of continuous applications as well as granting the freedom to issue bug fixes that can rollback in the past (e.g., at an old epoch snapshots) and thus, re-conciliate erroneous functionality starting from the time it actually occurred.

Another important aspect of snapshots is that they make a continuous execution of an application purely portable. Migrating a full pipeline is as simple as rolling back the application to a committed epoch. Given that snapshots themselves are blobs that can be moved to different locations, this grants an important flexibility to application developers since it makes an entire transition to different cloud providers or on-premise clusters trivial.

4.4 Performance Analysis

At the time of writing, Flink has gone beyond a research prototype and 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 (e.g., Uber, Netflix, Alibaba, Hwawei, Ericsson, King, Zalando etc.). In the remainder of this section, we present a performance analysis derived from live production metrics heavily focused on the performance costs and benefits of asynchronous epoch-based snapshots, which is the core contribution of this work. The data used in this analysis has been extracted from production server logs at King (King Digital Entertainment Limited), a leading mobile gaming provider with over 350 million monthly active users.

4.4.1 A Real-Time Analytics Platform

The Rule-Based Event Aggregator (RBEA) by King [79], is a reliable live service that is implemented on Apache Flink and used daily by data analysts and developers across the company. RBEA 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

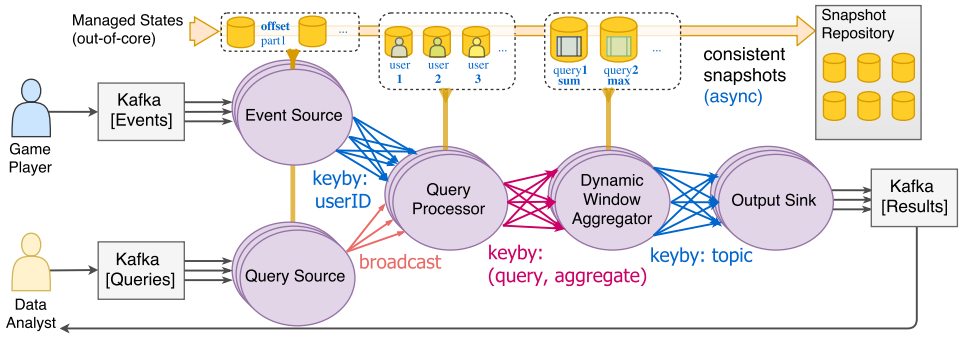


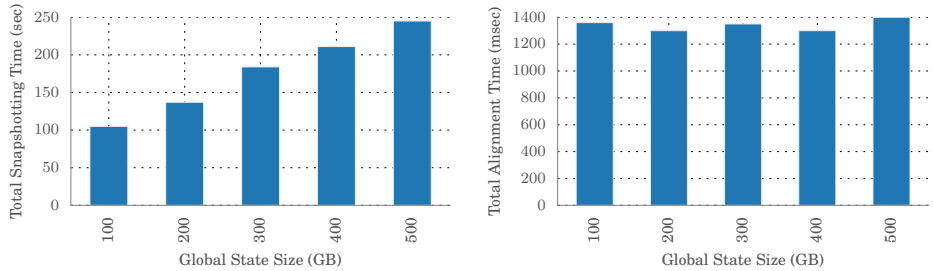
Figure 4.7: Overview of the Flink pipeline implementing an adhoc standing query execution service at King

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.

4.4.1.1 The RBEA Service Pipeline

Figure 4.7 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.



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

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

Figure 4.8: RBEA Deployment Measurements on Snapshots

4.4.1.2 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 local snapshot invocation for copying the full application state to HDFS (see section section 4.5 on asynchronous snapshot invocation). The performance of Flink's state management layer, that we discuss below, 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 4.8(a) depicts the overall time it takes to undertake a full epoch prepare phase (full snapshotting time) asynchronously for different state sizes. Mind that this simply measures the time difference between the invocation of a snapshot (begin of Prepare phase) and the moment all operators notify back they have completed it through the asynchronous backend calls (Prepared). 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 4.8(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

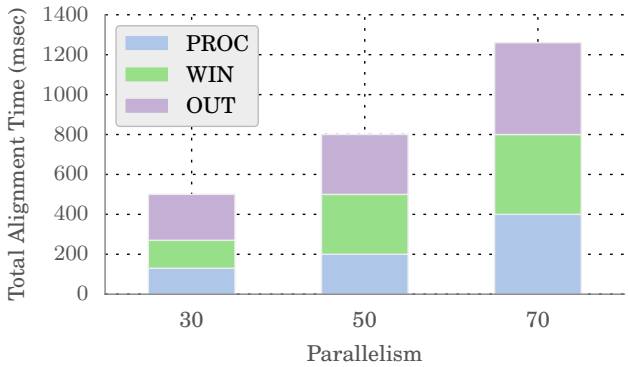


Figure 4.9: Alignment Time vs Parallelism [π : [30 : 70], `state`:200GB, `hosts`:18]

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 4.9 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× `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 (~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.

4.5 Additional Notes and Optimisations

It should be noted that certain performance metrics such as rollback recovery times, the use of incremental snapshots and more are not included in our analysis since they are orthogonal to asynchronous epoch snapshots and vary heavily between different state backends. At the core of asynchronous epoch-based snapshotting there is only a single "cost" within the critical path of an application, that of alignment. Epoch commit latencies depend on how each respective backend commits changes as well as which local backend is used (which affects the speed at which an epoch is

pre-committed). In this section we address several important state management optimisations that have been implemented by the Flink community to boost the system's performance, as well as known optimisations that can be incorporated in the future.

4.5.1 Asynchronous Snapshot Invocation

The asynchronous epoch-based snapshotting protocol (algorithm 6) employed by Flink obtains a recorded state as a local copy. There is yet another important level of asynchrony, orthogonal to the protocol, that of the invocation of the state copy. In reality, most state backends of Flink can execute the local snapshotting operation asynchronously. The snapshotting protocol can still accurately pinpoint the exact instant on which the snapshot can, otherwise asynchronously, be obtained. Flink's core backends such as RocksDB provide native support for asynchronous snapshots. In RocksDB [47] a snapshot operation flushes the memTable (uncommitted operation log) to disk, creating an SSTable (persisted operation log) and a copy process to an external file directory begins concurrently. Once copying is complete the notification logic to the snapshot coordinator triggers back asynchronously without going through the critical path of the application. Asynchronous snapshots are especially important since any synchronous copies can introduce significant latencies to the application, especially when full snapshots are employed. In the performance analysis presented in section 4.4 we only considered snapshots with asynchronous invocation. This allowed us to focus on the actual cost of the protocol (epoch alignment) while eliminating all backend-specific performance concerns.

4.5.2 Incremental Snapshots

So far it might have seemed counterintuitive to copy a full global state of an application per epoch. States can be as big as many Terabytes of data, generated by billions of events throughout an application's long lifetime. While the existence of a full state is important for many of the purposes presented before (e.g., state queries and reconfiguration) it is still possible to replace full snapshots with incremental snapshots. Incremental snapshots encapsulate only the changes applied to the managed state (i.e., writes, appends) between epochs. The actual implementation and usage of incremental snapshots relies on the backend that implements it. For example, Flink enables incremental snapshots through its RocksDB backend. In fact, copying only the "commit log" of operations on state is a natural procedure in RocksDB due to the fact that it already keeps all operations organized hierarchically in commit logs that are backed to disk. Since a full application state is needed periodically for a correct rollback, the system periodically compacts incremental snapshots into a full snapshot, thus, allowing applications to recover from specified epochs. It has been reported that incremental snapshotting can offer orders of

magnitude of speedup, such as from 3 minutes to 30 seconds for applications with Terabytes of state [80]. Therefore, in addition to reduced storage requirements, this feature can further lead to lower epoch commit latency, making side effects accessible in constantly less time.

4.5.3 Omitting Epochs

It is common in production deployments to face errors during a copy of a large snapshot (e.g., due to a temporary HDFS, S3 error). That means that certain instances of the epoch commit protocol might be potentially aborted. To make an application execution less sensitive to temporary external faults, the system can tolerate up to a specified number aborted snapshots. Mind that, when it comes to applications without external side effects (i.e., sinks) an arbitrary number of epochs can be ignored. If, for example, we omit snapshot of ep_2 but successfully obtain a snapshot of ep_3 e.g., Π_{ep_3} we can still offer reliable processing guarantees by omitting Π_{ep_2} since $E^{ep_3} \subset E^{ep_3}$. In fact, for fault tolerance purposes the system holds only the snapshots of the latest committed epoch. In case where transactional sinks are used and we want to maintain reliable processing guarantees all epochs have to eventually be committed in external state backends. In that case, the epoch commit protocol should repeat up to a number of retries until completion.

4.5.4 Relaxed Guarantees

In several application cases, strongly transactional processing is not a strict requirement and users are satisfied with at-least once processing guarantees. Conceptually, at-least-once means that given a deterministic stream input, every event production should occur at least one time, thus duplicate productions are allowed. Flink can provide this type of relaxed guarantees with a minor modification to its snapshotting algorithm, simply by omitting channel prioritization during alignment. If in algorithm 6 we simply keep all channels in the pending list and snapshot, as before, once all input barriers of an epoch ep_n arrive we have a case of a snapshot E' for which $E^{ep_n} \subseteq E'$. If we rollback an application from that snapshot all input records and productions of epochs $\geq n$ will occur and thus, any additional events that were included in the snapshot will be repeated. Flink allows for at-least once processing guarantees as a configuration option which employs this strategy, further eliminating the sole cost of epoch-based snapshots, that of alignment.

4.5.5 Dynamic Reconfiguration

The epoch alignment mechanism makes sure that each physical task completes all work of the current epoch before going further with any processing into the next. For cases of reconfiguration that do not alter the underlying structure of the stream

process graph but are limited to logical changes (e.g., bug fix in a user-defined function) it is possible to dynamically apply all changes without violating validity or epoch completeness across configurations. The main idea is to encapsulate the patch within the epoch markers and apply it at the instant that local check-pointing takes place. This way we can guarantee that conceptually all changes are applied after the completion of the epoch and no task will received records corresponding to the updated logic prior to an epoch completion, again due to alignment. This technique can potentially be integrated on Flink or other systems that use the same algorithm.

4.5.6 Idempotent Sinks

A special, simplified case in the epoch commit protocol is that of deterministic applications. This can be the case when no logical operator in the application is sensitive to order (e.g., restricted to progress-based operators such as event-time windows). In those cases each rollback will yield the exact same external (and internal) side effects on its output sinks. Repeated sink commits can therefore support idempotency (e.g., leading to the same key value store writes or database queries) and the epoch commit protocol can be ignored all together. That is because in that case we will always have the same external side effects no matter how many times all events occur internally due to a rollback.

4.5.7 Hybrid Fail-Stop and Recovery Model

One of the main assumptions made throughout this work has be then one of embedded *volatile* state and the fact that any data stored in the working memory of each task is lost upon a failure. The recent development and possible future adoption of more affordable *non-volatile random-access memory* (NVRAM) as well as Remote Direct Memory Access (RDMA) for compute clusters are some of the factors that can play a key role in the future for next-generation processing systems. These hardware advances would possibly make fine-grained fault-recovery [10, 11] more attractive for fault tolerance given that active state can always be accessed across distributed compute resources even upon process failures. Despite the beneficial usages in fault-recovery the need to address the full application state consistently and execute system-wide operations such as querying committed state or migrating logic would still require a form of a commit mechanism. In that context, our asynchronous epoch-commit protocol could be defacto employed to satisfy these needs using snapshots while allowing for local fine-grained recovery models to also be used seemlesly in par.

4.6 Related Work

Reliable Dataflow Processing: Flink offers coarse grained, job-level snapshot maintenance which grants various operational benefits. Flink bears many similarities to SEEP [11, 10] in terms of employing partitioned embedded state and a form of snapshotting for fault tolerance. Yet SEEP's main focus is not end-to-end transactional processing but rather fault tolerance, scalability and repartitionable state, thus, it has not adopted an application-wide commit mechanism. IBM Streams employs a pipelined checkpointing mechanism [64] 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 Streams adopted a stricter snapshotting scheme: 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 respective cycles. 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 [5] initially offered only guaranteed record processing through record dependency tracking. However, the most recent releases of Storm (and Apache Apex[39]) incorporated a variant of Flink's algorithm to its core in order to support transactional processing guarantees. Meteor Shower [81] employs a similar alignment phase to Flink. However, it cannot incorporate cyclic dataflow graphs which is a common case for online machine learning [82] and other applications. The same solution does not cover state rescaling and transparent programming model concerns. Naiad [20] and the sweeping checkpointing technique 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 [9] 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 [72] makes this approach infeasible in many commodity deployments. In fact, Apache Flink's distributed dataflow runtime serves today as a feature-complete runner of Apache Beam[42], Google's open-source implementation of the Dataflow Model[55].

Microbatching: Stream micro-batching or batch-stream processing (e.g. Spark Streaming [12], Comet [83]) 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 [84], a higher level framework built on Apache Storm offered a form of 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 expressibility for a significant set of use-cases.

4.7 Acknowledgements

The overall work on Flink's asynchronous epoch snapshotting protocol would not be as effective and impactful as it is today without the hard work and commitment of key people in the Flink community. First and foremost, I would like to thank Stephan Ewen, the contributions and influence of which address every small detail in Flink's production-proof state management. Furthermore, I would like to thank Gyula Fóra, Stefan Richter and Till Rohrmann among others for their involvement and implementation of key parts presented in this study such as Flink's backends. Finally, I want to thank Kostas Tzoumas and Flavio Junqueira for all the insightful discussions and the overall effort of the Flink community that keeps enriching Flink's state management with sophisticated optimisations, connectors and backends.

4.8 Summary: A Design Approach Perspective

We summarize the findings of asynchronous epoch-based commits and their applications in Apache Flink in terms of the three of our core design principles.

[D1] Blocking-Coordination Avoidance There are two types of coordination identified in epoch-based stream execution with snapshots. First, we have the centrally coordinated phases of an epoch, i.e., issuying epoch change events and collecting acknowledgements. The snapshotting protocol is another internal coordination mechanism which allows different processes to aquire an epoch-complete global system state using a distributed algorithm. In both cases, blocking synchronization is fully avoided. In the case of master coordination, all communication steps are triggered asynchronously to the underlying execution. Furthermore, the snapshotting protocol does not impose any blocking synchronization since even during epoch alignment all tasks continue their regular execution concurrently.

[D2] Runtime Transparency When it comes to the snapshotting technique, it is fully transparent to the user program. Epoch markers and local snapshotting are handled transparently by Flink's runtime, making zero restrictions on its programming model. In comparison, Spark's microbatching technique initially restricted programmers to express continuous computation into RDD transformations on processing time windows [12] . However, today we observe that these practices are avoided and Flink state management principles are also adopted by the Spark community for continuous processing [85] within Structured Streaming [74].

[D3] Model Compositionality Epoch-based snapshots offer a level of compositionality, not achievable by any other stream processing approach in the past (including regular snapshotting methods). As we analyzed already in section this chapter snapshots can be used for reconfiguration, fault tolerance as well as building sophisticated application provenance schemes on actual stream processing executions. Other design approaches that relied on a fine-grained fail-recovery model [11, 9], which were analyzed before achieved fault tolerance but did not consider such a compositional, application-wide concept as the one of snapshots (e.g., used for snapshot isolation of external queries, complete migration or provenance of continuous applications).

Bibliography

[1] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[2] 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.

[3] 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.

[4] 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*. ACM, 2003, pp. 668–668.

[5] "Apache Storm," http://storm.apache.org/, 2017.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 1–16.

[7] "The Lambda Architecture," http://lambda-architecture.net/, 2017.

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] 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.

[10] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 49–60.

[11] 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*. ACM, 2013, pp. 725–736.

[12] 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*. USENIX Association, 2012, pp. 10–10.

[13] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD Record*, 2005.

[14] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *AMC SIGMOD*, 2006.

[15] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," in *VLDB*, 2015.

[16] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[18] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[19] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, vol. 1, no. 10.4, 2014, p. 3.

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

[21] 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.

[22] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008.

[23] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2004, pp. 263–274.

[24] P. D. Bailis, "Coordination avoidance in distributed databases," Ph.D. dissertation, UC Berkeley, 2015.

[25] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[26] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in *USENIX Annual Technical Conference*, 2014, pp. 305–319.

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

[28] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[29] P. Carbone, S. Ewen, G. Fora, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink: Consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, 2017.

[30] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *arXiv preprint arXiv:1506.08603*, 2015.

[31] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, "Cutty: Aggregate sharing for user-defined windows," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016.

[32] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *VLDB*, 2004.

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

[34] "AthenaX : Uber's stream processing platform on Flink," http://sf.flink-forward.org/kb_sessions/athenax-ubers-streaming-processing-platform-on-flink/.

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

[36] "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.

[37] "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.

[38] "Windowing and State in Storm," https://community.hortonworks.com/articles/14171/windowing-and-state-checkpointing-in-apache-storm.html, 2017.

[39] "Apache Apex," https://apex.apache.org, 2017.

[40] "[spark-20928] continuous processing mode," https://issues.apache.org/jira/browse/SPARK-20928.

[41] "[Proposal Document] Snapshots for Beam," https://docs.google.com/document/d/1UWhnYPgui0gUYOsuGcCjLuoOUlGA4QaY91n8p3wz9MY/edit, 2018.

[42] "Apache Beam," https://beam.apache.org/, 2017.

[43] 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*, p. 28, 2015.

[44] P. Carbone, G. E. Gévay, G. Hermann, A. Katsifodimos, J. Soto, V. Markl, and S. Haridi, "Large-scale data stream processing systems," in *Handbook of Big Data Technologies*. Springer, 2017, pp. 219–260.

[45] P. Carbone, A. Katsifodimos, and S. Haridi, "Stream window aggregation semantics and optimization," 2018.

[46] "Apache Flink," http://flink.apache.org/, 2018.

[47] "Rockdb," http://rocksdb.org/, 2017.

[48] 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*. ACM, 2013, p. 5.

[49] 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.

[50] 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."

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

[52] "Apache Kafka project," http://kafka.apache.org/, 2017.

[53] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 46, 2014.

[54] 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.

[55] "Google Cloud Dataflow," https://cloud.google.com/dataflow/, 2017.

[56] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé *et al.*, "IBM Streams Processing Language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7–1, 2013.

[57] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Nasgaard, R. Soule, and K. Wu, "SPL stream processing language specification," *NewYork: IBMResearchDivisionTJ. WatsonResearchCenter, IBM ResearchReport: RC24897 (W0911 044)*, 2009.

[58] A. Bifet and R. Gavalda, "Learning from time-changing data with adaptive windowing." in *SDM*. SIAM, 2007.

[59] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators," in *International Conference on Database Theory*. Springer, 2005, pp. 37–52.

[60] 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)*, vol. 34, no. 3, pp. 375–408, 2002.

[61] Y.-H. Feng, N.-F. Huang, and Y.-M. Wu, "Efficient and adaptive stateful replication for stream processing engines in high-availability cluster," *IEEE Transactions on Parallel & Distributed Systems*, no. 11, pp. 1788–1796, 2011.

[62] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 1, p. 3, 2008.

[63] M. Balazinska, J.-H. Hwang, and M. A. Shah, "Fault-tolerance and high availability in data stream management systems," in *Encyclopedia of Database Systems*. Springer, 2009, pp. 1109–1115.

[64] 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*, vol. 9, no. 13, pp. 1341–1352, 2016.

[65] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 3, pp. 204–226, 1985.

[66] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[67] T. H. Lai and T. H. Yang, "On distributed snapshots," *Information Processing Letters*, vol. 25, no. 3, pp. 153–158, 1987.

[68] R. Guerraoui and L. Rodrigues, *Introduction to reliable distributed programming*. Springer Science & Business Media, 2006.

[69] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *ACM SIGMOD*, 2015.

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

[71] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 2003, pp. 25–36.

[72] 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)*, vol. 26, no. 2, p. 4, 2008.

[73] A. Kipf, V. Pandey, J. Böttcher, L. Braun, T. Neumann, and A. Kemper, "Analytics on fast data: Main-memory database systems versus modern streaming systems."

[74] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative api for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 601–613.

[75] R. Tarjan, "Depth-first search and linear graph algorithms," in *Switching and Automata Theory, 1971., 12th Annual Symposium on*. IEEE, 1971, pp. 114–121.

[76] S. S. Muchnick, *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[77] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, 2010, p. 9.

[78] 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*, vol. 41, no. 6, pp. 205–220, 2007.

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

[80] "Managing Large State in Apache Flink: An Intro to Incremental Checkpointing," https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html, 2018.

[81] 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.

[82] G. De Francisci Morales and A. Bifet, "Samoa: Scalable advanced massive online analysis," *The Journal of Machine Learning Research*, vol. 16, no. 1, pp. 149–153, 2015.

[83] 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*. ACM, 2010, pp. 63–74.

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

[85] "Low Latency Continuous Processsing Mode in Structured Stream in Apache Spark 2.3.0," https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html, 2018.

[86] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDBJ*, 2006.

[87] "Apache Samza project," http://samza.apache.org/, 2017.

[88] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *ACM SIGMOD*, 2005.

[89] B. Gedik, "Generic windowing support for extensible stream processing systems," *Software: Practice and Experience*, 2014.

[90] T. Grabs, R. Schindlauer, R. Krishnan, J. Goldstein, and R. Fernández, "Introducing microsoft streaminsight," Technical report, Tech. Rep., 2009.

[91] A. Bifet and R. Gavaldà, "Adaptive learning from evolving data streams," in *Advances in Intelligent Data Analysis VIII*. Springer, 2009, pp. 249–260.

[92] Y. Yu, P. K. Gunda, and M. Isard, "Distributed aggregation for data-parallel computing: interfaces and implementations," in *ACM SIGOPS*, 2009.

[93] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," in *ACM SIGMOD*, 2003.

[94] K. Patroumpas and T. Sellis, "Window specification over data streams," in *Current Trends in Database Technology–EDBT 2006*. Springer, 2006, pp. 445–464.

[95] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic, "The DEBS 2012 grand challenge," in *ACM DEBS*, 2012.

[96] J. Li, K. Tufte, D. Maier, and V. Papadimos, "Adaptwid: An adaptive, memory-efficient window aggregation implementation," *IEEE Internet Computing*, 2008.

[97] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues, "Slider: incremental sliding window analytics," in *Proceedings of the 15th International Middleware Conference*. ACM, 2014, pp. 61–72.

[98] "Apache calcite," https://calcite.apache.org/.

[99] "Apache Spark project," http://spark.apache.org/, 2017.

[100] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[101] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," *Journal of Grid Computing*, vol. 10, no. 1, pp. 47–68, 2012.

[102] A. Iyer, L. E. Li, and I. Stoica, "CellIQ: real-time cellular network analytics at scale," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 309–322.

[103] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 85–98.

[104] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 1.

[105] B. Chandramouli, J. Goldstein, and D. Maier, "On-the-fly progress detection in iterative stream queries," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 241–252, 2009.

[106] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi, "Incremental, iterative data processing with timely dataflow," *Communications of the ACM*, vol. 59, no. 10, pp. 75–83, 2016.

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

[108] "Introduction to Kafka Streams," http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple, 2017.

[109] M. A. Inda and R. H. Bisseling, "A simple and efficient parallel fft algorithm using the bsp model," *Parallel Computing*, vol. 27, no. 14, pp. 1847–1878, 2001.

[110] "Apache Hadoop project," https://hadoop.apache.org/, 2017.

[111] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 2.

[112] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[113] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[114] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[115] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM*

international symposium on high performance distributed computing. ACM, 2010, pp. 810–818.

[116] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: a universal execution engine for distributed data-flow computing," in *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2011, pp. 113–126.

[117] 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*, vol. 5, no. 8, pp. 716–727, 2012.

[118] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: A system for real-time iterative analysis over evolving data," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 417–430.

[119] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1268–1279, 2012.

[120] J.-M. Helary, C. Jard, N. Plouzeau, and M. Raynal, "Detection of stable properties in distributed applications," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. ACM, 1987, pp. 125–136.

[121] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2005, pp. 156–165.

[122] A. Lattuada, F. McSherry, and Z. Chothia, "Faucet: a user-level, modular technique for flow control in dataflow engines," in *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM, 2016, p. 2.

[123] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable distributed stream processing." in *CIDR*, vol. 3, 2003, pp. 257–268.

[124] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the Borealis stream processing engine." in *CIDR*, 2005.