Epoch alignment in stateful streams

Niels van Kaam

Master Thesis
Computer Science
Software Technology

Web Information Systems

TU Delft

Epoch alignment

in stateful streams

by

Niels van Kaam

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday April 18, 2019 at 11:00 AM.

Student number:	1358324	
Project duration:	August 1, 2018 – April 18, 2019	
Thesis committee:	Dr. Ir. A. Bozzon,	TU Delft, chair
	Dr. A. Katsifodimos,	TU Delft, supervisor
	Dr. Ir. G. Gousios,	TU Delft
	Dr. J.S. Rellermeyer,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

The basis for this thesis is my passion for functional programming, data science and software architecture. I am charmed by the concepts of streaming platform Apache Flink, which exposes a very simple API to define distributed stateful data streams. Where the definition of new jobs is made relatively simple by modern technologies, maintenance of an already deployed web of connected jobs is non-trivial. I think the maintenance of these webs should be simplified or automated, so that skilled engineers can focus on developing new exciting features rather than complex maintenance. The epoch alignment concept proposed by this thesis provides a strong base towards automatic maintenance of streaming jobs.

The project was initiated to continue work on the Codefeedr[27] project. Codefeedr aims to offer a platform that enables real-time analysis on data-sets and data-streams. Users can write queries in an expressive domain specific language (DSL) and publish the query to the platform exposing the real-time results for further usage. With the platform users are able to share their (real-time)data by publishing a connector to their systems. In the Codefeedr concept, streaming jobs are maintained automatically by the platform. This thesis further contributes towards this automated maintenance of streaming jobs.

This thesis has been made possible thanks to many people. First of all, I like to thank my girlfriend Aleksandra, our dog Florie and my parents for providing motivation and unconditional support throughout the project. Secondly, I would like to thank my committee, with responsible professor Alessandro Bozzon, supervisors Georgios Gousios and Asterios Katsifodimos and external expert Jan Rellermeyer, for always being available to give advice on the challenges that arose during the project. I would like to specially thank Paris Carbone for explaining the fundamental concepts of Apache Flink and providing the idea to use epoch alignment in combination with queryable state. Finally, I would like to thank my employer, Processfive, for supporting me during the project and providing flexibility. This allowed me to work on the project when needed.

Niels van Kaam
Delft, April 2019

Contents

List of Figures

1

Abstract

While the amount of data and variability in data produced by numerous systems in a modern company continues to increase, users desire real-time and consistent results from complex analyses across a large variety of event sources. In industry, stream processing systems are emerging to process events with low latency in a scalable and reliable fashion. As more and more stream processing jobs are processing mission critical events, older jobs are subject to maintenance and have to be upgraded or replaced. These upgrade operations include a snapshot-restore operation, where between the snapshot and restore a non-trivial state conversion has to be performed. Such an operation requires a lot of technical expertise and imposes significant downtime on the job itself and all jobs that depend on it.

This thesis proposes a mechanism to align the progress of multiple independent jobs sharing common event sources. The mechanism is an extension of the checkpoint protocol proposed by Carbone et al [22]. Not only does this mechanism simplify maintenance of streaming jobs by allowing hot-swap operations with exactly-once processing semantics, but it can also be used to provide consistency of queryable state. By implementing a proof of concept we show that this so called epoch alignment can be achieved with minimal additional costs over exactly-once processing semantics.

2

Introduction

The last decade, big data processing has been a dynamic research area with large shifts in processing models. The increasing amount of data has driven industry away from traditional one-size-fits-all database systems in search for more scalable and fault-tolerant solutions.

The first model widely used by industry, moving away from traditional database systems and towards scalable distributed event processing, is MapReduce [24]. Based on the MapReduce concept, multiple platforms such as Hadoop [2], Pig [26], Hive [47] and Spark [50] have evolved. Although these MapReduce frameworks solve the scalability and fault-tolerance issues from traditional database systems, they have significant limitations. The MapReduce model is limited to acyclic stateless computations and is not capable of producing real-time incremental results. This makes MapReduce unsuitable, for example, for Machine Learning (ML) [12], which requires both stateful and iterative computations. The delay inherited from the batch-processing design is also a major challenge in use cases such as real-time decision making, such as identified in [28].

The next major shift in processing models is the introduction of scalable stream processing systems. In a stream processing system, data is not modelled as a finite batch, but as a continuous, possibly infinite, stream of events. The concept shows similarities to event-driven models such as Reactive Extensions [38], but applied to a distributed system.

The first generation of stream processing systems, such as Storm [9], focussed on low latency and made sacrifices on reliability. To deal with the reduced reliability, the so-called lambda architecture [7] was born. The lambda architecture uses both a stream processing system (speed layer) and a batch processing system (batch layer). The batch processing system periodically recomputes a reliable view of the data, which is merged with the real-time view from the stream processing system. Combined, these systems provide both real-time results from the speed layer, and reliable results from the batch layer. Any inconsistencies in the results of the speed layer will be corrected once the batch layer performs its next recomputation. At the time of writing, this lambda architecture is deployed in many applications in industry.

The double layers of the lambda architecture, however, come at a cost. Firstly, including both a batch and speed layer requires double implementation of a process, which not only costs effort but also comes with the risks of inconsistencies between the two implementations. Secondly, the speed layer might produce unreliable results which have to be corrected once the results from the batch layer arrive. Finally, a component has to be created which merges the results from the speed layer and batch layer, adding an additional step and complexity to the overall process.

Recent research has shown that it is possible to provide reliability guarantees in stream processing systems with very minimal costs in terms of latency [21]. Therefore, the newer generation of stream processing frameworks include mechanisms to provide reliability guarantees, making the batch layer of the lambda architecture redundant. The removal of the batch layer, which is called the kappa architecture [6] by some sources, greatly simplifies the architecture of scalable and reliable stream-processing systems.

Apache Flink is a recently released stream processing system that uses a checkpointing mechanism to provide such reliability guarantees. The checkpointing mechanism is designed to create restore points to recover the process upon failure. This thesis continues on this checkpointing mechanism and proposes an extension, which does not only use the checkpoints as recovery points for a single job, but also to align the progress of individual jobs in a larger ecosystem.

2.1. Stream processing

Stream processing systems continuously process events produced by one or more event sources. In most deployments, before events enter the stream processing framework, events are written to a publish-subscribe system. This publish-subscribe system temporarily stores events, providing a backup of events in case of a failure in the stream processing system. This publish-subscribe middleware can also multi-cast events to multiple stream processing jobs. It is common to have streaming queries both read and write data to a publish-subscribe system such that multiple streaming jobs are chained after one-another with the publish-subscribe system in between.

Figure 2.1 shows a simple example of a deployment of an application with a stream processing back-end. The example shows the position of the stream processing framework both reading and writing from the publish-subscribe middleware, and front-end applications using No-SQL stores. Figure 2.1 serves as an example, there are many other viable deployments such as writing data from the stream processing framework to the No-SQL store directly, or replacing the No-SQL store with any queryable storage that suits the desired application.

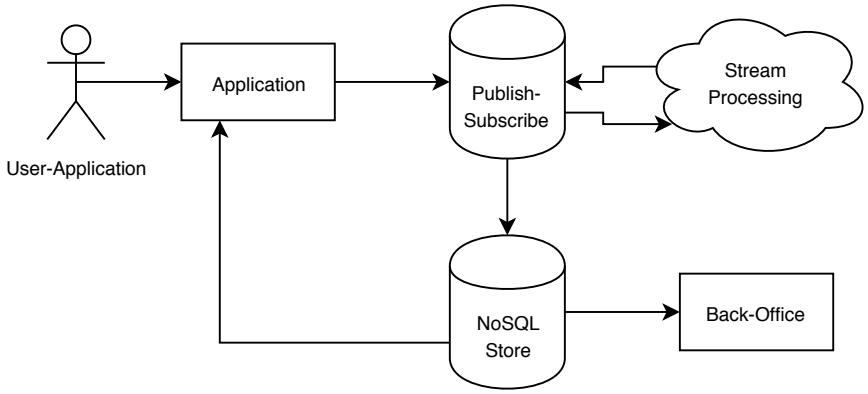


Figure 2.1: Example of a deployment of a stream processing framework.

A stream processing job, as shown in Figure 2.2, can be modelled as a directed graph, where each edge represents a channel through which events can flow, and each node represents an operator which performs an operation upon receiving an event. This operation can be sending one or more new events to outgoing channels and/or modifying the state of the operator.

The example from Figure 2.2 shows an important aspect of stream processing jobs that might be unfamiliar to readers from the relational database field. In streaming systems, data is considered to be an infinite stream of events. However, there is also the notion of time-windows [37], which split the infinite stream of events into an infinite stream of finite buckets. The example of Figure 2.2 uses a tumbling window to aggregate the sum of sales in each category. A tumbling window splits an event stream into non-overlapping buckets of a fixed time-length. There are many types of windows, time-based or event-based and overlapping and non-overlapping. Windows are necessary in streaming applications to perform aggregations that require a finite set of events, such as sum and count. Without windows these aggregations would never produce a result.

Figure 2.2: Example of a simple stream processing job from a classical webshop. Orders are grouped by category, and summed in a tumbling window. All channels are directed left to right, but to reduce clutter only two arrows are drawn.

The windowing introduces another aspect of stream processing jobs. In order to calculate the sum of each category within a window, the operator has to keep track of the sum of each category for the duration of the window. This means the operator carries information that does not belong to a specific event. This is called the "state" of an operator, or "Operator state". Operator state can be small such as in the example of a sum operator, where the state only consists of the sum so far. However, the state can also be large, such as when

joining two separate event streams on a key within a window. In that case the state contains all events of both streams within the active windows.

The previous model, shown in Figure 2.2, is the definition of a streaming job. The actually deployed job can be scaled up by the stream processing framework, such as shown in Figure 2.3. Now the "group by" operator, which seemed to have no purpose in Figure 2.2, distributes events to either instance of the window operator based on the group key (category), because the aggregation requires each event of the same category to be processed by the same operator. The transformations for scalability and distribution of operators in a cluster are in general abstracted away from the user writing streaming queries.

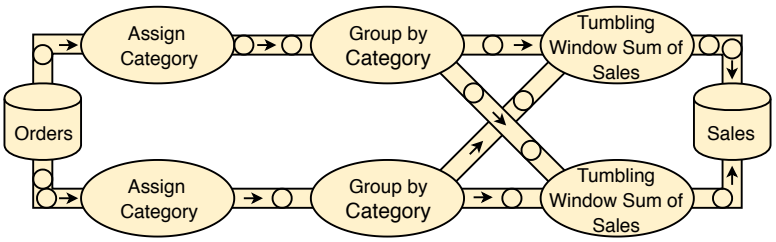


Figure 2.3: Example of a deployment of a stream processing framework.

In the examples given so far, the operators have the structure of a Directed Acyclic Graph (DAG). This is not always the case, as operators in some stream processing frameworks are allowed to send events back into previous operators, which is desirable in, for example, machine learning [35, 39]. In the case of iterative operations, it is the responsibility of the developer to prevent events from looping around the cycle forever.

2.2. Epoch Alignment

To provide consistency, stream processing frameworks employ checkpointing mechanisms, which provide restore points in case of a failure. There are different checkpointing mechanisms providing different levels of consistency, explained in more detail in Chapter 3.

For epoch alignment, this report focusses on a specific mechanism used by Apache Flink [22], in which checkpoint markers are passed through the job graph from operator to operator in absolute order. In this checkpointing mechanism the stream processing platform itself decides when to inject markers into the stream. Because jobs are executed independently of each other, this means markers in related jobs have unrelated positions.

This thesis proposes a method to align the markers of related streaming jobs while maintaining independent execution. With aligning we mean that when two independent jobs are chained after one another, the checkpoint markers flow through the intermediate system from the upstream job to the downstream job. In the case where two independent jobs consume the same event streams, we mean that both independent jobs inject the markers at the exact same place. Because the markers represent progress in event streams, we call this *Epoch Alignment*. The concept and properties of epoch alignment are explained in more detail in Chapter 4.

2.3. Motivation

There are multiple issues that can be solved with epoch alignment. The first application, which also was the initial motivation for this research project, is the capability of hot-swapping streaming jobs. Hot-swapping streaming jobs would bring us one step closer to a stream processing management system where the deployment of jobs is abstracted away from the user, such as the ideal situation for the Codefeedr [27] platform. Two more motivations of epoch alignment, described later in this section, are providing an idempotent queryable state interface, and the application of hot-swapping streaming jobs in multi-query optimization.

2.3.1. Hot-swapping jobs

Recently, there has been a lot of research on domain specific query languages for stream processing jobs simplifying the definition of new jobs [16–18, 25]. However, little research has been done into further automating maintenance of already running jobs. The capability to hot-swap running jobs would make this maintenance easier, because it would not only allow the system to move running jobs for resource management, but also update running jobs to a newer version without significant downtime.

Let us assume there are two jobs with aligned epochs, where the "old" job is pushing its output to the target destination, and the "new" job is voiding its output. Both jobs need to agree on a moment to perform the hot-swap. With aligned epochs this operation becomes simple. Some external coordinator decides to perform the hot-swap on a specific future epoch, named the "swap" epoch. This "swap" epoch is communicated to both jobs via a two-step commit protocol, where the pre-commit and commit happen in separate epochs of both jobs (such that upon failure and recovery of either job the hot-swap still continues). Once the new job starts with the next epoch after the "swap" epoch, it waits for the old job to finish the "swap" epoch. Because both jobs run with exactly-once processing guarantee, it can be assumed this will happen at some point in the future. Once the old job has completed the alignment epoch, the new job can start pushing data to the subject. After passing the "swap" epoch, the old job will no longer push its data to the subject (and void its data).

2.3.2. Queryable state

Another motivation for researching epoch alignment are its potential applications for queryable state. Queryable state is a recent concept introduced into streaming systems, where a system outside of a streaming job (which can also be a second streaming job) accesses the state of one or more operators in a streaming job. The native way to support similar lookups on the state of a job is to output the desired information as a side-stream of the job, and to use these events to update the state in a queryable storage, such as a relational database. The external system can then query the storage with some latency. Queryable state can be used to replace the relational database, reducing latency and simplifying the architecture by removing a component from the ecosystem.

In the context of queryable state, the relation between epoch and operator state provided by epoch alignment, can be exploited to improve consistency. This is especially the case when the state query from the external system results from an event published by the stream processing system in question. In that situation, it is likely that the information of the external system is already outdated compared to the state of the stream processing system to be queried, resulting in an inconsistent view on the state. Figure 2.4 shows an example of this situation, where the external system is modelled as a second job.

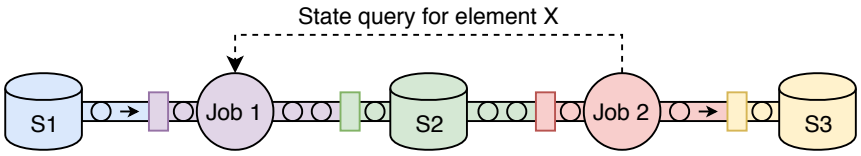


Figure 2.4: Example of state query from "Job 2" to "Job 1". If "element X" was introduced in the red epoch, and removed in the purple epoch, the state query from Job 2 might fail when the state of element X has already been removed.

Another situation that might occur is shown in Figure 2.5, where two jobs using the same source run in parallel. In this example, "Job 1" queries the state of an element in "Job 2", which "Job 2" has not yet seen, because it has a backlog.

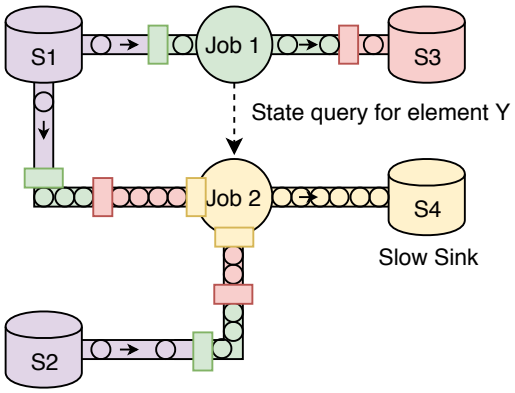


Figure 2.5: Example of state query from "Job 1" to "Job 2". "Job 2" has some backlog, so elements that "Job 1" has seen, are not present yet in "Job 2".

The two given examples are base cases for more complex issues that might occur when using queryable state, where one job has seen more or less events than another job, even when the query does not involve querying for a specific element. The general characteristic in all these situations is that the state query is not idempotent, as the state of a job mutates over time.

This idempotency is where epoch alignment can be used. If the state query would be performed in the context of a specific epoch, the query would be idempotent. Because the state of an epoch will never change when the epoch has been completed. This does require the epoch alignment implementation to provide a mapping of relations between epochs of one job to epochs of another job. This mapping allows the job or external system to add a reference to a specific epoch to the state query. Thus, under the assumption that the state back-end supports lookups in historical state, epoch alignment can make queryable state lookups idempotent.

2.3.3. Multiple query optimization

In the case of a Relational Database System (RDBS), whenever a query is fired, statistical information about the database is used to identify a cost optimal execution plan for the query [31]. Streaming queries can be optimized in a very similar way, for example by using Apache Calcite. However, in contrary to queries in a RDBS, streaming queries run indefinitely while the environment changes. This means the execution plan of a query might have been optimized at the moment of deployment, but might no longer be optimized at a later moment. Thus, at a later moment it might be relevant to redeploy a job with a different execution plan.

When looking at a larger scope, when multiple jobs are running alongside each other in the same ecosystem, optimizations could also involve merging jobs or splitting jobs. Combining multiple queries into a common execution plan is called multiple query optimization. This has been a research topic in the context of RDBS's for decades [36, 41, 44].

In a RDBS, multiple query optimization involves a step where queries are batched before executed, which introduces additional latency and therefore reduces its effectiveness in a lot of use cases. On streaming platforms, multiple query optimization is much more relevant than on a RDBS, because queries run indefinitely. Especially on a platform such as desired for Codefeedr [27], it is likely that multiple users define similar queries. Merging overlapping queries could potentially save a lot of resources.

Because the epoch alignment allows hot-swapping of streaming jobs with exactly-once guarantee, this also opens the road to having a continuous query optimizer running and restructuring the jobs, without compromising throughput, latency or processing semantics. This continuous optimizer could use metrics from the running job to further optimize execution plans, and then hot-swap them in place. Whenever a user deploys a similar query to one already existing, the optimizer could deploy a merged version with the existing query with two outputs, instead of deploying an entirely new query. Further research will be needed before a system such as described in this paragraph will be production ready, but epoch alignment brings it one step closer.

2.4. Research Questions

To be able to perform epoch alignment and apply it in the cases we described in the previous section, we need to find an answer to the following research questions:

- The main research question for this thesis follows directly from the topic: *Can checkpoints of stream processing jobs be aligned with exactly-once processing semantics?*

- Streaming platforms are used in applications where low latency is required. Therefore, in all of the applications of epoch alignment that have been mentioned, low latency is preferred. *Can latency be kept minimal when jobs are running aligned?*

- The most important reason to have independent jobs, is to have separate components which can be restored individually upon failure. The epoch alignment would be much less useful if it would compromise the independence of jobs, and could cause multiple jobs to fail because of a single problem. In that case, one might as well publish everything as a single job. *Can two aligned jobs remain independent in the context of failure and recovery?*

- Because modern stream processing frameworks provide a lot of flexibility to developers, we do not expect to find a solution that works for all possible types of streaming jobs or deployments. For example,

cyclic dependencies between jobs make epoch alignment more complex. The final research question therefore is: *What are the limitations of running two jobs in aligned mode?*

2.5. Outline

This chapter introduced stateful stream processing, the concept of epoch alignment. We also gave a motivation for the research on epoch alignment, and identified research questions for this thesis.

Chapter 3 describes the related work on checkpointing mechanisms for both stream processing systems and distributed systems in general. Chapter 4 explains the conceptual idea of epoch alignment and proposes a mechanism how epoch alignment can be achieved. Chapter 5 describes how this mechanism was implemented in our proof of concept. Chapter 6 explains the experiments we have conducted to measure the costs of epoch alignment and validate our implementation. Chapter 7 and 8 summarize the findings during the thesis and provide suggestions for future work to continue the research.

3
Related Work

This chapter discusses the related work on checkpointing mechanisms both for streaming applications and distributed systems in general. We start with the classical Chandy-Lamport protocol which forms the base for many checkpointing mechanisms, and move up in time towards modern checkpointing protocols specialized for distributed stream processing systems.

3.1. Preliminaries

When discussing reliability and checkpointing in stream processing frameworks, the term *processing semantics* shows up. The processing semantics define what assumptions the user of a stream processing framework can make about whether at all, and how many times events are processed. In stream processing frameworks, the three most common processing semantics are:

- At-most-once: Each event may or may not be processed, but each event may never be processed more than once.
- At-least-once: Each event is guaranteed to be processed, but events may be processed more than once.
- Exactly-once: Each event is guaranteed to be processed once and only once.

The checkpointing mechanisms described in this chapter are designed to guarantee one or more of these processing semantics on a distributed system. In general this is guarantee is provided by producing a snapshot, or checkpoint, which can be used as a restore point to recover a from a failure. During normal execution, and during failure and recovery, the processing semantics are guaranteed to hold.

This is non-trivial due to the fact that operators can contain state. Simply periodically snapshotting the state of each operator would not produce a valid snapshot of the entire process, as it would require the snapshot to be taken at the exact same time in each distributed operator, which is practically impossible [23].

The related work described in this chapter covers various of these checkpointing mechanisms, including the one used in Apache Flink [1], which forms the base of our epoch alignment method.

3.2. Chandy-Lamport

The problem of epoch alignment or checkpointing in general falls under a more generic problem, namely detection of a global state of a distributed system. This has been a relevant research topic for a long time. Fundamental concepts for global state detection in a distributed system were proposed by Chandy and Lamport [23].

To obtain an exact snapshot of a distributed system, a global clock would be required such that each process can snapshot its state at the exact same time. At present, there is no reasonably affordable distributed system with a global clock. Chandy and Lamport define an algorithm which captures a "meaningful" state from the distributed system. Meaningful means that the state in the snapshot did not necessarily occur, but the current state of the system is reachable from the state in the snapshot.

Chandy and Lamport define the following model of a distributed system: The system consists of distributed processes connected by channels. Each channel has an infinite buffer size, maintains message order,

and delivers messages in a finite time. A process maintains a state. During an event the process is allowed to alter its state, and send or receive a message from one of its channels. In Chapter 4 we will show this model is still relevant for modern stream processing systems.

Upon receiving a marker across a channel c, the Chandy-Lamport protocol performs the following steps:

```
if(p has not recorded state) {
    p records its state
    p sends a marker on all outgoing channels
} else {
    p records all messages received along c after the state was recorded
        and before p received the marker along c
}
```

The algorithm is designed for strongly connected graphs, and relies on logging in-transit messages during the protocol. The latter is an issue for modern stream processing systems, because due to the large amount of events, these are a significant component in the checkpoint size, even if only a fraction of the messages ends up in the checkpoint. Still, the Chandy and Lamport protocol can be viewed as the base concept from which many more recent checkpointing mechanisms have been deducted.

3.3. Checkpointing in the MPI

The Message Passing Interface (MPI) [45] is a standard for parallel computing, focussed on scalability and portability, which is still in use today. MPI defines low-level operations for communication between processes in a distributed system. Processes are part of a communicator, which groups a number of processes that communicate with each other. The base version of MPI defines communication methods for both point-to-point communication and broadcasts. As MPI is designed for heterogeneous distributed systems, it also defines a protocol for communicating data types between processes.

An extension of MPI, called MPI-2 [29], focusses on features for more dynamic deployments, allowing processes to spawn new child processes. It also defines a number of one-sided communication methods which give processes direct access to parts of the memory of another process.

MPI can be considered a more generic and low-level early alternative to distributed stream processing frameworks. In distributed systems, failures are an inherited risk, which means that also in this early stage of distributed computation, checkpointing was a very relevant research topic. With MPI, checkpoints can be used to either move running computations to other hardware, or to restore upon failure, which is similar to the applications of checkpoints on modern stream processing systems. However, due to the low-level protocol and the less strict processing model, the checkpointing mechanisms for MPI have to be more generic. Inherently, these are less efficient in more specific use cases like stream processing.

Key checkpointing mechanisms for MPI are CoCheck [46] and LAM/MPI [43]. CoCheck [46] is implemented as a wrapper around MPI. CoCheck is built on top of MPI and exposes the same interface as MPI. CoCheck solves two main issues when checkpointing a distributed application using MPI.

The first issue solved by CoCheck is that addresses of processes might change whenever a process is restored. To deal with this dynamic property, CoCheck keeps a routing table of virtual addresses and physical addresses, which is updated upon restoration of a process. In the checkpointing mechanism of stream processing frameworks this issue is not relevant, because the developer of streaming jobs is not allowed to directly address processes by the name or address. The developer defines an execution graph, and the connections between operators in the graph are fully managed by the stream processing framework, and thus can be properly reconstructed by the framework upon recovery.

The second challenge is getting a meaningful snapshot of the state of the channels at the moment of a checkpoint. Here, CoCheck uses a mechanism similar to the Chandy-Lamport protocol. Because with MPI the developer decides which process communicates with which process, the checkpointing mechanism has to assume all-to-all communication. This is dealt with by letting each process send a ready message (or checkpoint marker) to all other processes. Each process buffers all messages between sending out the ready markers and receiving ready markers from all processes. These buffered messages are stored alongside the state of the process, and restored alongside the state after recovery.

For distributed stream processing engines the mechanism used by CoCheck is not optimal. The mechanism works fine for heavy computational tasks with minimal communication. Stream processing jobs are characterized by heavy communication (due to a lot of events) and relatively low amounts of computation

per event. In that case, similar to the original Chandy-Lamport protocol, the buffers that are stored as part of the state can become very large, resulting in a large checkpoint size.

A more recent approach, LAM/MPI [43], uses a different mechanism. Instead of buffering input channels, after receiving a checkpoint trigger, out-of-band communication is used to communicate a bookmark (or offset) of messages that have been sent to other processes. The receiving processes, before taking a snapshot, use these bookmarks are to drain incoming channels. Outbound messages triggered during the draining are buffered.

From the MPI perspective the approach in LAM/MPI seems a clear improvement. Receiving messages is done with callbacks, and it is likely that these callbacks do not directly trigger new outbound messages. Therefore, this approach should significantly reduce the size and costs of the checkpoint. From a streaming perspective, however, each inbound message is very likely to produce one or more new outbound message. Therefore, in streaming applications, also LAM/MPI has the downside of resulting in a large checkpoint size.

3.4. Borealis Distributed Stream Processing System

In the earlier days of stream processing, a common way to provide consistency was to run the same operation multiple times in parallel, and compare the results. A very relevant method is the "Delay, Process, Correct" (DPC) protocol for the Borealis stream processing system proposed by Balazinska [11, 19].

The model used in DPC assumes that each operator has sufficient processing and network capacity for its job, such that there is no significant buffer or latency due to backlogs of events. The user defines a maximum delay within which the system can attempt to recover upon failures. A lower delay increases the risk of tentative results, which might have to be corrected at a later stage. This way the DPC lets the user chose between consistency and availability.

DPC considers output to be consistent when each operator has processed the events in an order that would be valid if the system would have ran without failures. Events which are emitted to meet the availability requirements, but for which this consistency cannot be guaranteed, are called "tentative". Events for which consistency can be guaranteed are called "stable". DPC provides eventual consistency, which means eventually all replications of all operators have processed the same events in the same order, and produced the same output events.

With DPC each operator is assigned a timespan which it can use to detect failures of upstream operators, and guarantee it is connected to replications of its upstream operators that are producing stable events. If it fails to find a stable replication for one of its upstream operators within the timespan, it will, if available, use an upstream operator producing tentative events. If also no upstream operator producing tentative results is available, it will just continue processing events from the other still working upstream operators. In both cases, the events emitted by the operator are marked as tentative, as the consistency requirement cannot be guaranteed.

When an operator detects all of its upstream operators are producing stable events again, it performs a roll-back to the state of the lastly produced stable event and replays all events since then. To meet the high-availability requirements, at least one replication of the operator waits with the roll-back and replay operation until another replication that has performed the roll-back has fully catched up again.

Balazinska's approach is very relevant to our approach of epoch alignment, since Balazinska is able to hot-swap operators at any time of the computation. This is possible by using only deterministic operators, and sorting events on event-time, such that all events are processed in a deterministic order in every operator. This does, however, require the event source to assign an event-time that can be sorted deterministically. To avoid creating large buffers, the latency between event-times of different event sources has to be small, implying the event sources have to run a somewhat synchronized clock. DPC also requires each operator to consume events from more than one upstream operator to buffer and sort all incoming events on event-time. This means, When multiple instances of the same operator are running in parallel, sorting must also take place after every shuffle.

3.5. MillWheel and Dataflow model

MillWheel is a stream processing framework developed by Google [13] in which, similar to the Borealis system, a streaming query is modelled as a DAG of individual operators. To implement operators, MillWheel exposes a low-level API, which gives the user access to event data and also to the managed state. MillWheel provides exactly-once processing semantics on a framework level, meaning that processing semantics are guaranteed if the user implements the API exposed by MillWheel in a deterministic fashion, and manages

operator state by using the state constructs exposed by the framework.

In [14], MillWheel Akidau et al. propose the *Dataflow Model*, which addresses how stream processing systems can be designed to deal with concepts such as unbound data streams, event-time and windows. Akidau et al. propose a number of high level building blocks that can be exposed through an API. These building blocks are in principle independent from the underlying stream processing framework, and make reasoning about data streams following the proposed semantics more natural and accessible to users that are less familiar with stream processing semantics.

MillWheel uses a per-operator checkpoint mechanism similar to DPC [19]. However, where in DPC the downstream operator is responsible for retrieving data, in MillWheel the upstream operator is responsible for delivering a message with exactly-once guarantee. Upon receiving a batch of records, the operator performs its operations, and writes output events and state mutations in a single atomic operation. This single operation also forms the checkpoint. After this operation has completed, the sender is acknowledged of the proper retrieval of events. Finally the events are emitted to the next downstream operator.

To provide at-least-once processing guarantee, the upstream operator is responsible for continuing to send records until an acknowledgement has been received. To provide exactly-once guarantee, the downstream operator has to deduplicate events by comparing incoming events with the local check-pointed state. This is possible because the local state is updated before the acknowledgement is sent to the upstream operator. All these operations are handled by the framework, making most computations in user code idempotent. This method, where checkpoints are completed before events are sent to the next upstream operator, is called *strong production*.

Because MillWheel does not have to maintain exact ordering of events, the checkpoint mechanism is more efficient than DPC. However, because not any form of message ordering can be guaranteed, there might never be a moment when a downstream operator has processed the exact same set of events as an upstream operator. Therefore, the checkpointing mechanism of MillWheel seems to be unable to provide the properties we need for hot-swapping and epoch alignment.

3.6. Flink

Apache Flink, formerly known as the Stratosphere platform [15], is a framework and distributed processing engine for stateful computations over bounded and unbounded data [1]. Flink combines data stream processing and batch processing in a single API [21]. Batch programs are considered to be a special case of streaming, where the stream is finite. Flink, a streaming or batch process is modelled as a directed graph, consisting of stateful operators connected by data streams. In this graph, operators are parallelized into subtasks, and streams are split into partitions.

The APIs exposed by Flink follow various abstraction levels. The lowest level allows the developer to implement a custom operator and directly access the state managed by Flink, which is a similar abstraction level as MillWheel [13]. Flink also exposes higher level APIs, by providing implementations of building blocks that follow similar semantics as the DataFlow Model [14]. At the highest level of abstraction, Flink exposes a SQL API, where streaming jobs are specified in a tailored SQL dialect.

Flink provides processing guarantees by using a checkpoint mechanism which is very similar to the Chandy-Lamport algorithm [20]. Instead of checkpointing each individual operator, Flink's mechanism takes a meaningful snapshot of the entire job with minimal interruption of the process. This is done by periodically injecting so called checkpoint markers at the sources of a job. Whenever an operator in the DAG receives one of these markers, it takes a snapshot of its current state before passing the marker to all of its outputs, similar to the Chandy-Lamport protocol. When an operator with multiple input streams receives a marker on one of its input streams, it will not read any more data from that specific input stream until it has received the same marker from all of its input streams. This temporarily blocks a part of the input stream, but also makes sure that if the job is acyclic all channels are drained before taking the snapshot, significantly reducing the checkpoint size compared to the Chandy-Lamport protocol.

To deal with cycles in the job graph, Flink removes the cycles by creating an imaginary source and sink, cutting a stream that would otherwise close the cycle in two [22]. The sink outputs its elements directly to the source. However, this source does get notified whenever a checkpoint should be taken. As soon as the source gets notified of a checkpoint, it passes the marker along the stream. Until it receives the same marker it sent, it will record all data it sent to the next operator. When it receives the checkpoint marker, it will record all records it has sent between sending and receiving the checkpoint marker as its state.

An important effect of Flink's checkpointing mechanism is that the checkpoint markers travel through

the stream as barriers, without events passing back or forward through the barrier at any point of the process. Between each barrier, deterministic ordering of events does not have to be guaranteed, making the process more efficient than the DPC approach. The checkpoint barriers themselves follow exact ordering and provide detectable moments at which a downstream operator has processed the same events as an upstream operator with exactly-once guarantee. These checkpoint barriers are able to provide the moments where epoch alignment can be performed, which is why we chose to build our proof of concept of epoch alignment on top of Apache Flink.

3.7. Other stream processing systems

The last decade, many other stream processing frameworks have emerged with their own checkpointing mechanisms. Samza [40] is a stream and batch processing engine to work in cooperation with Kafka [33] or another replayable messaging system. In addition to scalability, Samza focusses on fast recovery upon failure and a single API for both batch and stream processing. At the time of writing, Samza does not support *exactly-once* processing semantics.

The original batch processing engine Spark [8] has also implemented real-time queries with Spark Streaming [42, 49]. Where Flink considers batch processing as a special case of streaming, in which a batch is a finite stream, Spark implements streaming as a continuous flow of micro batches. Exactly-once processing semantics are provided by the underlying Resilient Distributed Datasets (RDD's). Each micro batch reads its input from an RDD, and writes its output and state back to an RDD in a single transaction. This mechanism is optimized using *lineage graphs*, which let the output RDD track how it was computed. Thus, instead of storing every RDD, the Spark engine can recompute the required events in case of a failure. Spark seems a valid alternative to implement epoch alignment, especially since epoch alignment includes a *lineage graph* of aligned epochs. We chose Apache Flink for our proof of concept, because it uses a continuous model. With Spark we would have to rely on micro batches. Integrating epoch alignment with RDD's and *lineage graphs* would be interesting future research.

Storm [48] is a stateless stream processing engine developed by Twitter. Storm natively supports at-most-once and at-least-once processing semantics, but also supports stateful exactly-once semantics with its Trident API. For the at-least-once semantics Storm uses acknowledgements that backtrack to the source of an event. This event source is responsible for storing events until the acknowledgement arrives. The event source has to re-emit events if no acknowledgement arrives, or when a failure response is received. On top of Storm, Trident supports stateful processing with exactly-once semantics by splitting the input into batches, and run these batches in a transactional way.

As a successor to Storm, Twitter has been working on Heron [34]. The API exposed by Heron is compatible with the Storm API. Heron improves the processing transparency and traceability of errors, performance, and supports back-pressure. At the time of publication however, just like Storm, Heron does not support exactly-once semantics out of the box [34].

3.8. Summary

We discussed work related to checkpointing mechanisms, both for stream processing systems and distributed systems in general. Stream processing frameworks have specific properties that can be exploited to design more efficient checkpointing mechanisms than general purpose checkpointing mechanisms for distributed systems, such as MPI. While checkpointing stream processing systems is a popular research area, there has been little research on relating the checkpoints individually deployed jobs (epoch alignment). Spark's lineage graphs come closest to this concept, but lineage graphs are built on top of a batching mechanism rather than a streaming mechanism. Moreover, lineage graphs remain within the scope of a single deployment.

4

Epoch Alignment

This chapter explains the properties of epoch alignment and proposes a method to achieve these properties. Firstly, section 4.1 explains the concept of aligned epochs. Next, section 4.2 explains the meaning of equivalent position, and explains why equivalent positions are desired for aligned epochs. Section 4.3 discusses the relation of Flink's checkpointing mechanism to the Chandy-Lamport protocol, and explains how Flink's checkpointing mechanism can be mapped to the Chandy-Lamport protocol. In section 4.4 we use this model to explain how epoch alignment can be implemented as an extension to Flink's checkpointing mechanism. Finally, in sections 4.5 and 4.6 we investigate and validate properties of the resulting aligned epochs.

4.1. Aligned epochs

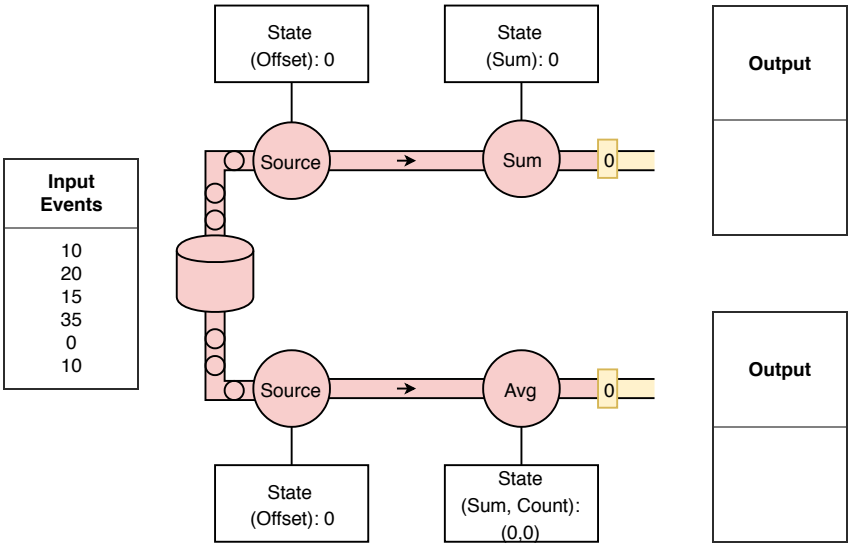
An epoch in the context of stream processing, refers to all events that have been processed up to a specific checkpoint. For example, epoch 2 includes all data from checkpoint 0, checkpoint 1 and checkpoint 2 together. The concept of epoch alignment proposed by this thesis is to have multiple independent jobs using the exact same epochs in their snapshots. Epochs can be aligned either when two jobs are using the same sources in parallel, or when one job consumes the data published by another job.

Snapshots and epochs are best explained visually. Figures 4.1 to 4.3 show an example of the state of operators and the content of snapshots of two parallel jobs as time progresses. The jobs shown in the figures contain two operators. The first operator in both jobs is the Source, which connects to the external system producing the events. The Source has to keep track of its progress of reading the external source. This progress is stored in the operator state as Offset.

For the job on the top, the second operator is a sum operator for all events it has seen so far. To calculate a sum when a new event arrives, the operator just needs the sum of all events that have been processed so far. Therefore, the state of the second operator of the top job is the Sum.

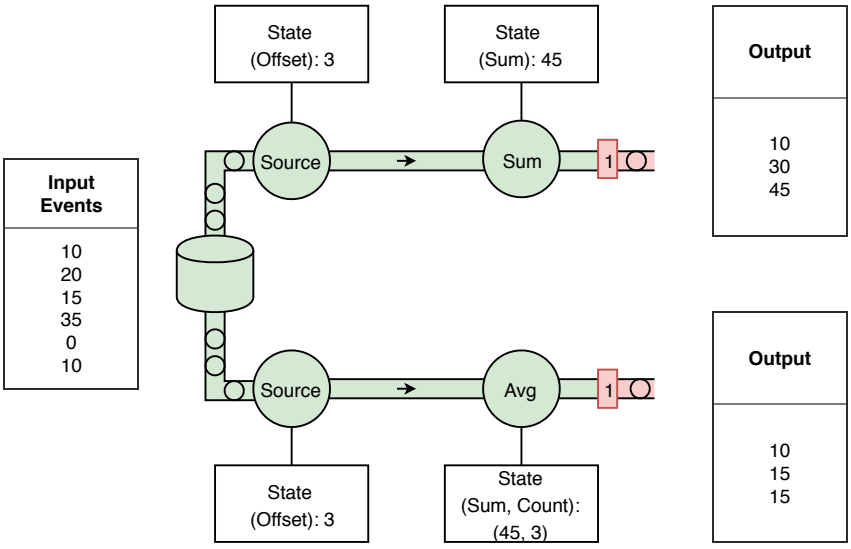
The bottom job calculates the average of all events that have been processed so far. To calculate the average, the operator needs both the number of events that have been processed, and the sum of these events. Therefore, the state of the second operator of the bottom job contains both the amount of events that have been processed, and the sum of these events.

The table on the bottom of the figures shows the content of every checkpoint that has completed. The checkpoint content consists of the state of both operators. The tables on the left and on the right show the data in the relevant storage. The checkpoints in both jobs are aligned, because at every checkpoint the offsets of the sources are equal. This indicates that both jobs have read the same set of elements at every checkpoint.



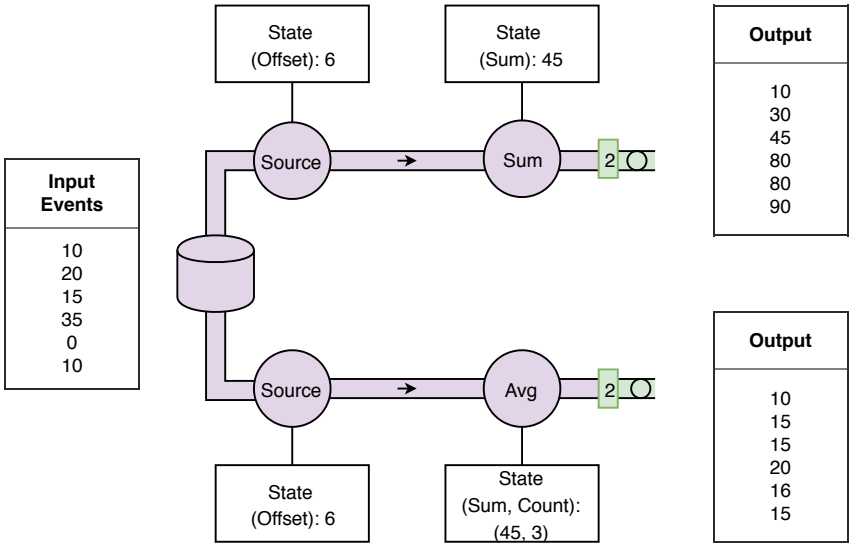
	Checkpoint 0		
Job 1	Offset: 0 Sum: 0		
Job 2	Offset: 0 (Sum, Count): 0,0		

Figure 4.1: Checkpoint 0/2: Example of epoch alignment with two parallel jobs. The job on the top calculates the sum of all elements. The job on the bottom calculates the average of all elements.



	Checkpoint 0	Checkpoint 1	
Job 1	Offset: 0 Sum: 0	Offset: 3 Sum: 45	
Job 2	Offset: 0 (Sum, Count): 0,0	Offset: 3 (Sum, Count): (45, 3)	

Figure 4.2: Checkpoint 1/2: Example of epoch alignment with two parallel jobs.

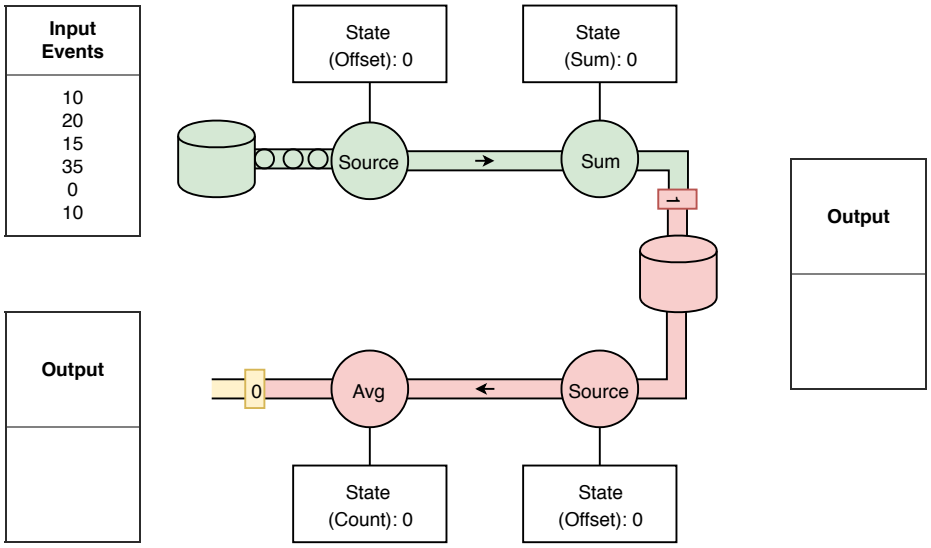


	Checkpoint 0	Checkpoint 1	Checkpoint 2
Job 1	Offset: 0 Sum: 0	Offset: 3 Sum: 45	Offset: 6 Sum: 90
Job 2	Offset: 0 (Sum, Count): 0,0	Offset: 3 (Sum, Count): (45, 3)	Offset: 6 (Sum, Count): (90, 3)

Figure 4.3: Checkpoint 2/2: Example of epoch alignment with two parallel jobs.

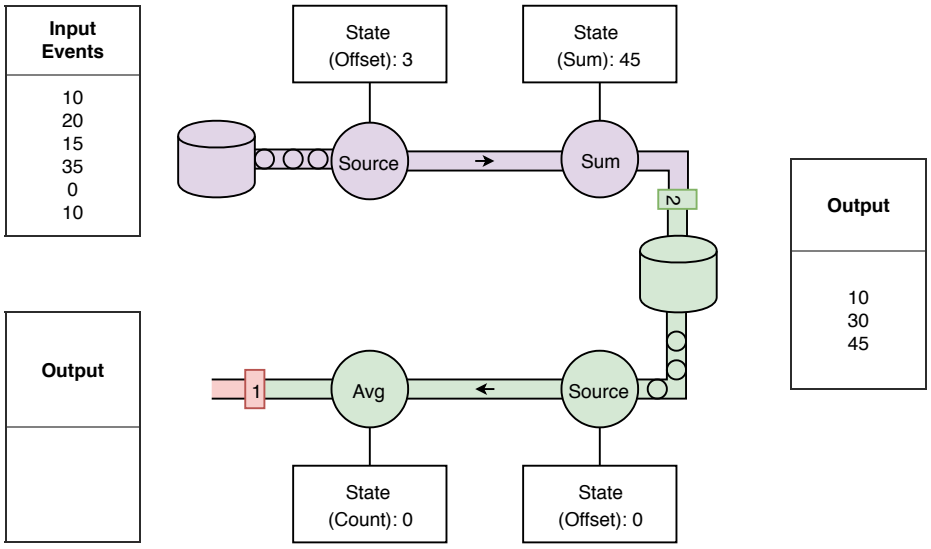
Because the calculation of the average requires both the number of events, and the sum of all events so far, the average can also be calculated using the results from the top job. Figures 4.4 to 4.7 show this case, where the two jobs are running after one another with a storage in between. In this case, the bottom job only needs to track the number of events to calculate the average.

This is an example of a case where one job consumes the data from another job. The jobs are also aligned, but the checkpoint increments are not the same. Checkpoint 1 from the bottom job is aligned with checkpoint 0 from the top job, and consecutively checkpoint 2 from the bottom job is aligned with checkpoint 1 from the top job. This shows that aligned epochs should include a mapping that indicates which epochs from one job are aligned with which epochs from another job. In concept this is somewhat similar to lineage graphs in Spark [49].



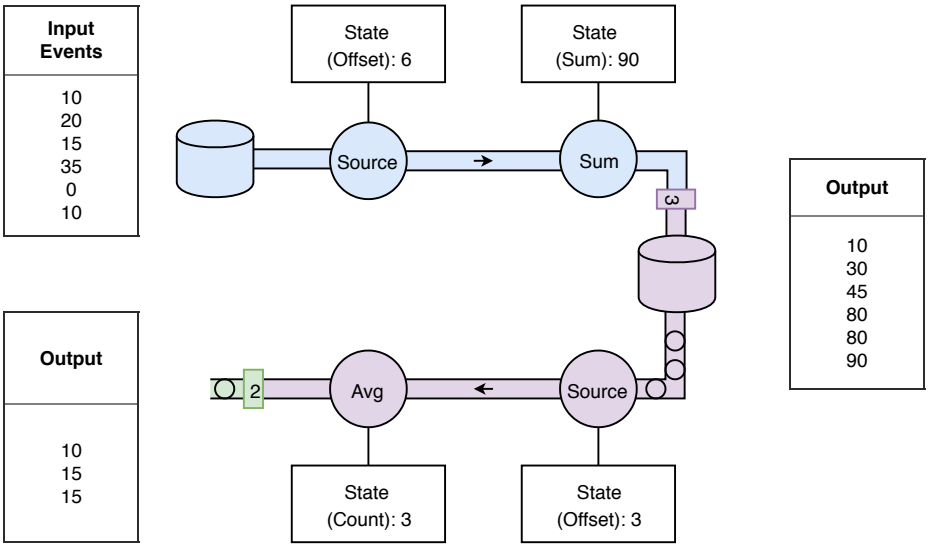
	Checkpoint 0	Checkpoint 1
Job 1	Offset: 0 Sum: 0	Offset: 0 Sum: 0
Job 2	Offset: 0 Count: 0	

Figure 4.4: Checkpoint 0/3: Example of alignment of two sequential jobs. The bottom job reads the events from the top job, through an intermediate storage.



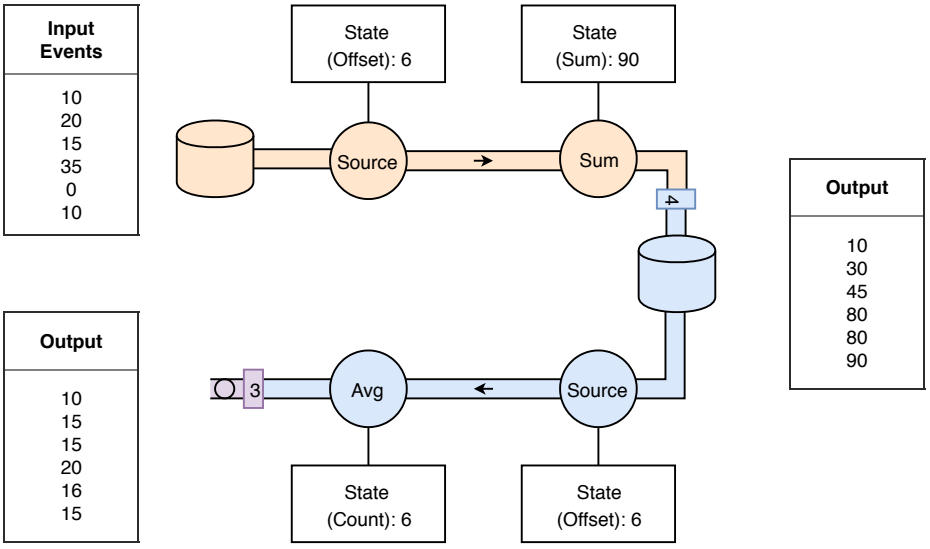
	Checkpoint 0	Checkpoint 1	Checkpoint 2
Job 1	Offset: 3 Sum: 45	Offset: 0 Sum: 0	Offset: 3 Sum: 45
Job 2	Offset: 0 Count: 0	Offset: 0 Count: 0	

Figure 4.5: Checkpoint 1/3: Example of alignment of two sequential jobs.



	Checkpoint 0	Checkpoint 1	Checkpoint 2	Checkpoint 3
Job 1	Offset: 3 Sum: 45	Offset: 0 Sum: 0	Offset: 3 Sum: 45	Offset: 6 Sum: 90
Job 2	Offset: 0 Count: 0	Offset: 0 Count: 0	Offset: 3 Count: 3	

Figure 4.6: Checkpoint 2/3: Example of alignment of two sequential jobs.



	Checkpoint 0	Checkpoint 1	Checkpoint 2	Checkpoint 3
Job 1	Offset: 3 Sum: 45	Offset: 0 Sum: 0	Offset: 3 Sum: 45	Offset: 6 Sum: 90
Job 2	Offset: 0 Count: 0	Offset: 0 Count: 0	Offset: 3 Count: 3	State: 6 Offset: 6

Figure 4.7: Checkpoint 3/3: Example of alignment of two sequential jobs.

The examples shown in Figure 4.1 to 4.7 are base cases where epoch alignment can be applied. From these base cases more complex topologies can be constructed, for example, a job publishing data that is consumed by multiple jobs.

Epoch alignment is non-trivial, because each job should be independent, and because the alignment should not compromise processing semantics. To perform the alignment of two parallel jobs, a certain form of communication is required between the sources of both jobs. This communication is not allowed to compromise the processing semantics or independence of the jobs. This means that a failure of a job should not cause a failure in another job, or compromise the processing semantics when performing a snapshot-restore operation. This thesis shows that it is possible to implement epoch alignment with minimal modification to an existing checkpointing mechanism, without each individual job knowing of the existence of a job it is aligned with.

4.2. Equivalent Position

One of the motivations for Epoch Alignment, which is explained in Section 2.3.1, is its application in hot-swapping streaming jobs. Figures 4.8 and 4.9 show how hot-swapping two streaming applications ("Old" and "New") on aligned epochs could look like. The hot-swap itself looks very simple. The challenge, however, is to carry out the hot-swap with exactly-once processing guarantee. In the context of hot-swapping, "exactly-once processing guarantee" means that for every event that was ingested by both jobs, at least one of the two jobs produced the results to the target, but never both.

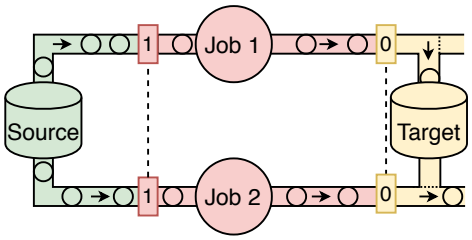


Figure 4.8: Hot-swap to be performed on checkpoint 0, before checkpoint 0 completed. Events from job 1 are sent to the target, events from job 2 are voided.

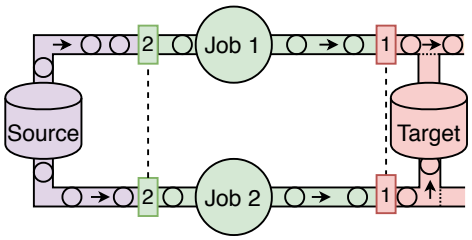


Figure 4.9: Hot-swap was performed on checkpoint 0, after checkpoint 0 completed. Events from job 1 are voided, events from job 2 are sent to the target.

Because stream processing systems are parallel and distributed, in order to provide the exactly-once processing guarantee at the hot-swap, a position in the output stream of each parallel instance of the old job must be found, which is equivalent to a particular position in the output stream of each parallel instance of the new job. Figure 4.10 shows how an equivalent position in two streaming jobs with two parallel operators could look like. All operators in both jobs have to agree on this position to perform the hot-swap at the correct position.

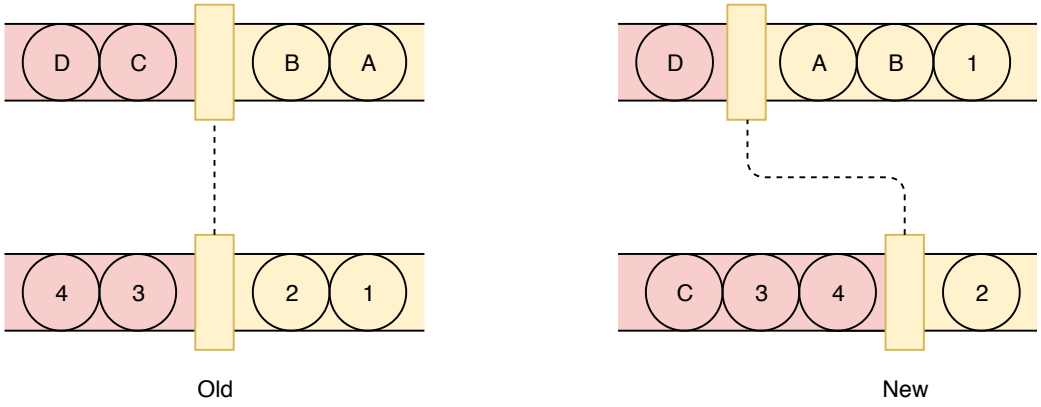


Figure 4.10: Equivalent positions in two data streams. Note that event order and distribution is mixed up, but the marker separates the same sets of events.

The solution proposed by this research is to use the checkpointing mechanism proposed by Carbone et al [22], but when multiple independent systems read from the same source, the markers are injected at the same position in each independent system. The result will be a system of chained streaming jobs, where periodic checkpoints of each individual job could be combined to create a global checkpoint of the entire system.

4.3. System model

To explain our proposal, we first map the processing model of Flink to the model proposed by Chandy and Lamport in [23]. After this, we show the difference between the checkpointing protocol of Flink and the Chandy-Lamport protocol. Using the mapped model, we show that epoch alignment can be achieved by only modifying the topology of the model, and not the protocol.

The checkpointing protocol of Flink and the Chandy-Lamport protocol are explained in sections 3.6 and 3.2 respectively. Figure 4.11 shows an example of an acyclic job in Flink, taken from [22]. In Figure 4.12 we transformed the DAG from Figure 4.11 into a strongly connected graph. The transformation has been done by introducing a "Checkpoint Coordinator", and connecting all sources with outgoing edges, and all sinks with incoming edges. Note that the event source(s) and sink(s), represented by the cylinders, are not part of the system for which the snapshot is computed. With the transformation, the computation itself has not been modified, meaning that no messages other than the markers are passed from and to the checkpoint coordinator.

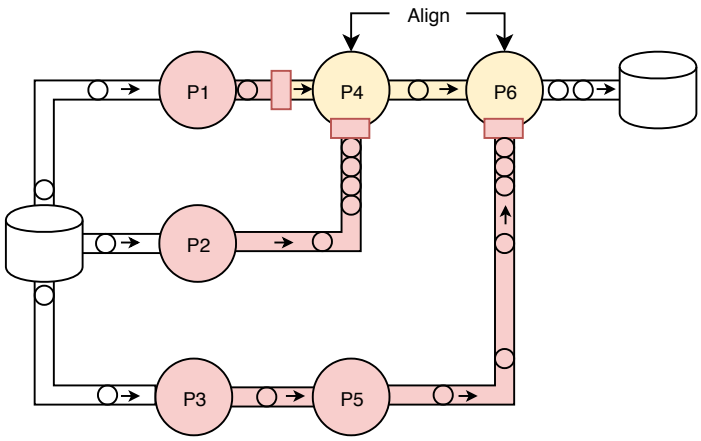


Figure 4.11: Example of an acyclic streaming job in Flink, from [22]. Colours indicate the individual checkpoints which are closed by a checkpoint marker (square). The white components are not part of the system that is being checkpointed.

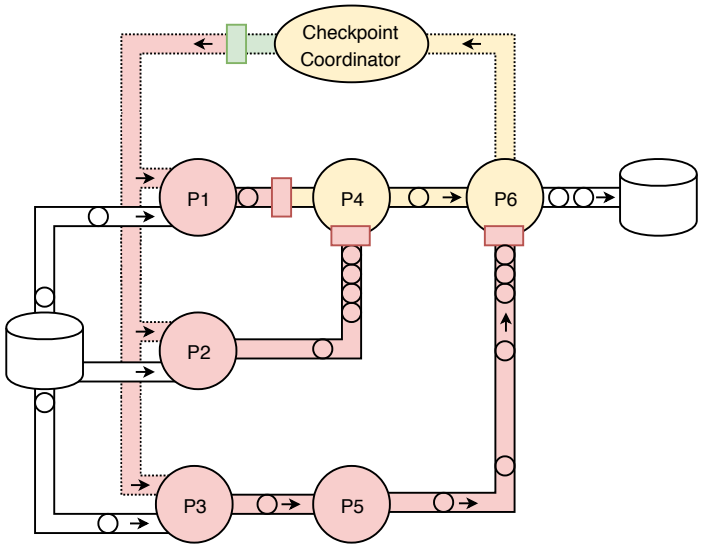


Figure 4.12: Example of the Flink job from 4.11, converted into a strongly connected graph. The dotted channels indicate that these channels are only used for checkpoint markers, and events are never emitted across these channels.

To explain the results, we will follow the Chandy-Lamport protocol step-by-step on the system shown in Figure 4.12. For the initial state, assume no markers are present, and that the Chandy-Lamport protocol is initiated in the checkpoint coordinator. When applying the protocol, the following steps are performed.

- The checkpoint coordinator sends a marker to all outgoing channels. This is shown by the green marker in Figure 4.12.
- Operators P1-P3 receive the marker. The operators snapshot their state. The only channel incoming to operators P1-P3 (from within the system) is the channel from the checkpoint coordinator. From this channel the marker was received, so the channel is recorded as empty. Operators P1-P3 send the marker to all outgoing channels, shown as the red markers in Figure 4.12.
- P4 receives a marker from P2. P4 snapshots its state, stores the channel from P2 as empty, and sends a marker to P6.
- Because the Chandy-Lamport model has arbitrary but finite latency across channels, one potential situation is that P4 receives its marker from P1 directly after it received the marker from P2, without receiving messages from P1 in between. Let's assume this happens, which will be discussed later. In that case, P4 will snapshot the channel from P1 as empty.
- P5 receives a marker from P3. P5 snapshots its state, snapshots the channel from P3 as empty, and sends a marker to P6.
- P6 performs the exact same steps as P4, but sends its marker to the checkpoint coordinator.
- The checkpoint coordinator receives a marker from P6. The state of the channel from P6 to the checkpoint coordinator is empty by design.

By following these steps, we observed that in the evaluation of the Chandy-Lamport protocol, only states of the operators were recorded, and all channels were recorded as empty. This is because we made the assumption that all markers were received at the same time. Carbone et al [22] show that it is safe to enforce this behaviour in streaming jobs that are directed acyclic graphs (DAGs), without the risk of introducing deadlocks. This behaviour is enforced by blocking the channel after a marker has arrived, until markers have arrived from the other incoming channels. Figure 4.12 visualizes this enforced behaviour by drawing a backlog of events at P4 and P6.

Let us now consider a job with a cycle, such as shown in Figure 4.13. The transformed strongly connected graph for the Chandy-Lamport protocol is shown in Figure 4.14. Because of the cycle, the job is no longer a DAG, and we can no longer wait in P1 for all markers to arrive. Following the Chandy-Lamport protocol, this

means that operator P1 in Figure 4.14 buffers all events received from P3 after it received the marker from the checkpoint coordinator and before it received the marker from P3 as state of the channel from P1 to P3, and store these events alongside its state.

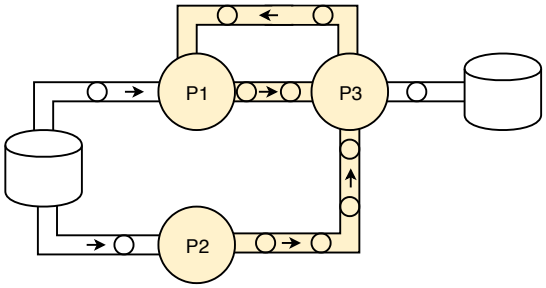


Figure 4.13: Example of a cyclic job.

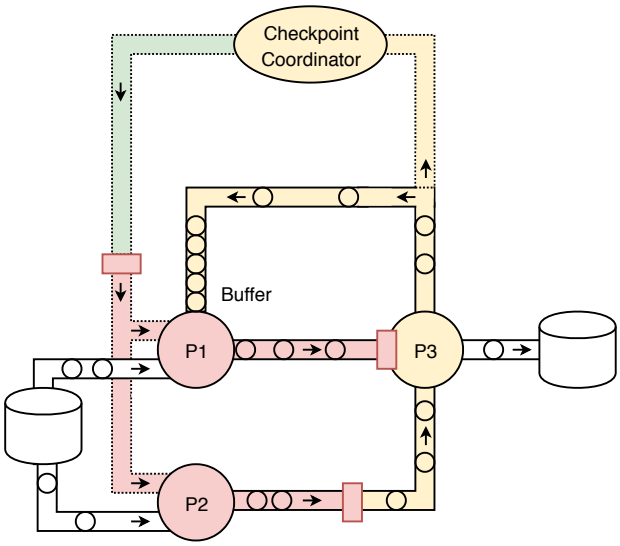


Figure 4.14: Example of the job from 4.13, converted into a strongly connected graph.

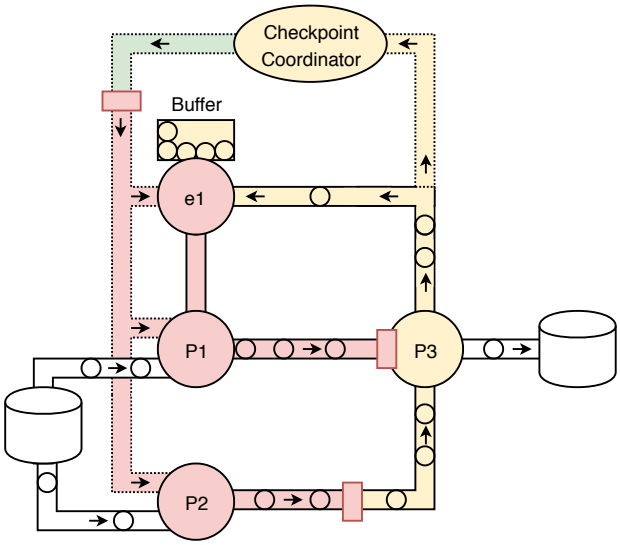


Figure 4.15: Example of the job from 4.13, converted into a strongly connected graph similar to proposed by Carbone et al [22]. Colours indicate the individual checkpoints which are closed by a marker (square).

The protocol proposed by Carbone [22] differs from how the Chandy-Lamport protocol behaves on the model in Figure 4.13. The method proposed by Carbone [22] is to include a sink and a source operator between the channel which is closing the cycle. The artificially created sink sends its events directly to the created source. The created source operator gets notified of new checkpoints by the environment the same way as an ordinary source, and emits a marker to the next operator. Between sending the marker, and receiving a marker from the artificially created sink, the source buffers all events received from the sink, and stores these events as its state. Upon snapshot and restore, these stored events are emitted to the next operator.

Figure 4.15 shows how this behaviour can be simulated with the Chandy-Lamport protocol by introducing a new empty operator e1. Operator e1 behaves exactly like the Chandy-Lamport protocol, where it will buffer all events after receiving its first marker. To prevent deadlocks, e1 should not block until it has received markers from all incoming channels.

The main advantage of the protocol proposed for Flink compared to the Chandy-Lamport protocol seems that in Flink each operator only performs a single snapshot operation, which is either the state of an operator (P1, P2, P3) or a buffer of elements (e1). In Figure 4.14 P1 first records its own state upon receiving the marker from the checkpoint coordinator, and later updates this state with the buffer from P3 to P1 after receiving the marker P3, making the snapshot in P3 a two-step operation.

However, there also seems to be a disadvantage of the protocol proposed in [22] compared to the Chandy-Lamport protocol. If in Figure 4.15 P1 would receive the marker from e1 before it receives the marker from the previous operator (in Figure 4.13 the checkpoint coordinator), it will stop reading elements from e1 and pump elements into the cycle until it receives the marker from the previous operator. All these elements are received by e1 after it has received the first marker, and thus are part of the buffer in the snapshot. Since finding optimizations for the existing protocols is not the goal of this research, we leave further investigation to the reader.

4.4. Epoch Alignment

In section 4.3 we have shown how the checkpointing protocol of Flink relates to the Chandy-Lamport protocol. Let us now continue with epoch alignment across multiple jobs. The idea is quite simple, we just let the markers flow from one job into the next. An example is shown in Figure 4.16. Note that in the second job, the snapshots are not triggered by the checkpoint coordinator, but by the arrival of markers through the source, which is the only modification that would need to be made to Flink's protocol. From Figure 4.16 we can see that if the channel between P6 and Q1 meets the requirements of a channel in the model defined by Chandy-Lamport [23], which is maintaining the message order, the entire system is a strongly connected graph. This means the Chandy-Lamport protocol can be used to obtain a meaningful snapshot of the entire system. The state of the checkpoint coordinators, which are connected with each other to make the global system strongly connected, is always empty by design, because no other messages than markers are sent and received from the checkpoint coordinator.

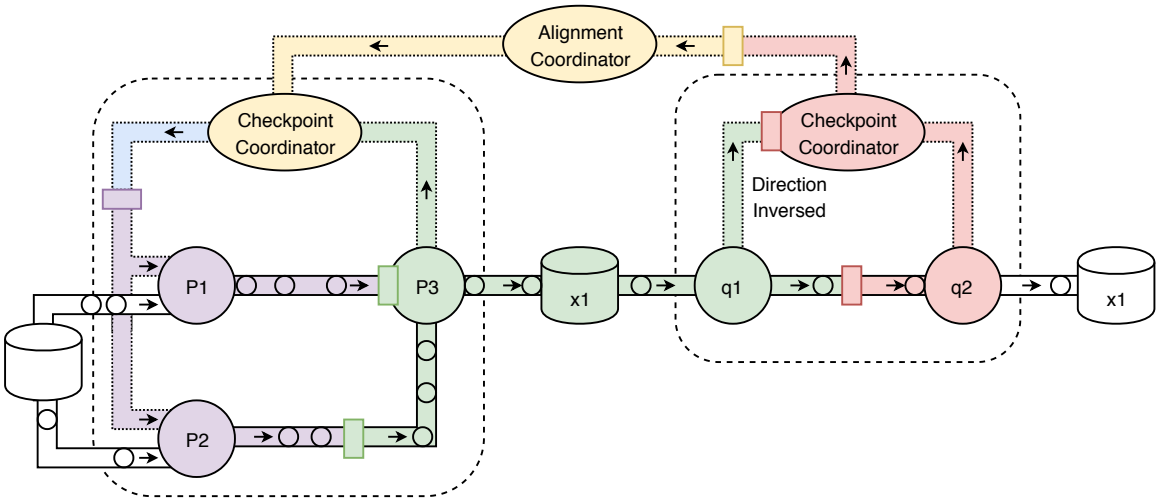


Figure 4.16: Example of two connected jobs passing markers. Colours indicate the individual checkpoints which are closed by a marker (square).

4.4.1. Advanced chains

More complex relations are possible between jobs than just one job reading data from another job. The case where multiple jobs ingest the data from a single job, shown in figure 4.17, can be treated the same way as the previous case from Figure 4.16. This means there are two jobs that have their checkpoints triggered by the previous job. The case where a single job ingests data from multiple jobs, shown in Figure 4.18, is more complex. This figure shows that the markers from both source jobs have to be merged into a single marker for the third job. In Figure 4.18 the markers are drawn with the same color. In practice, these markers have to be linked together.

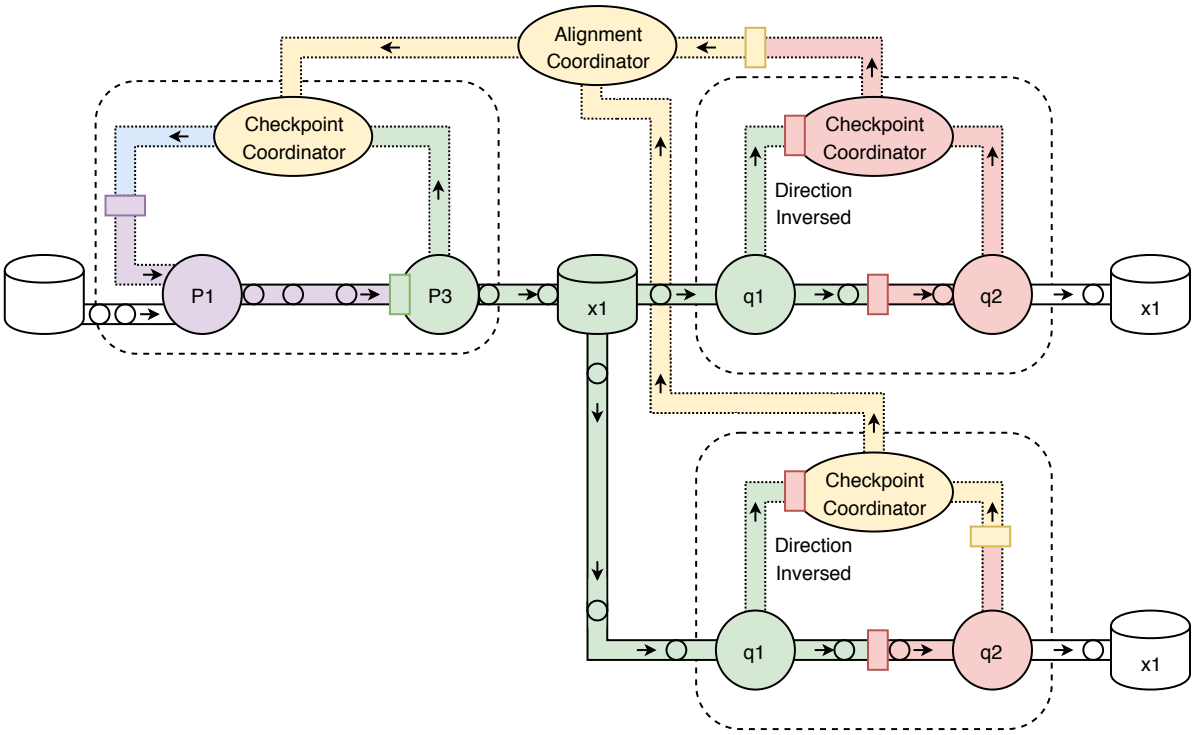


Figure 4.17: Example of connected jobs passing markers, where two jobs read from a single job.

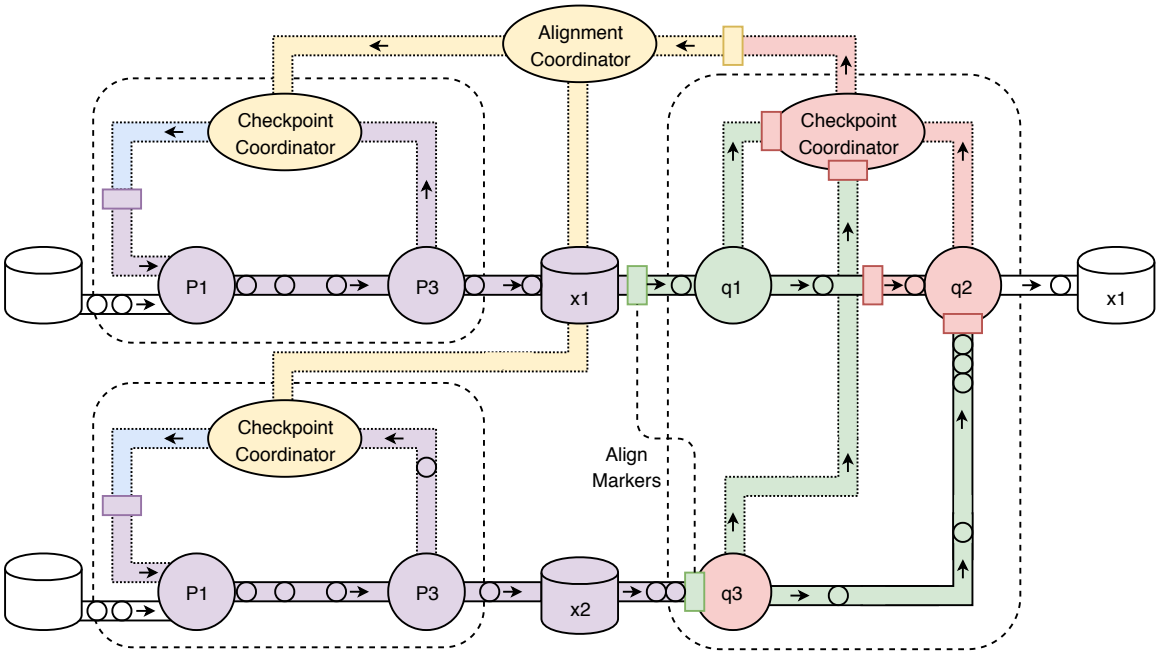


Figure 4.18: Example of connected jobs, one job ingesting data from two previous jobs. Both markers from the previous jobs have to be aligned.

The concept of the marker merging is shown in Figure 4.19. To perform the marker merging, the markers (only markers, not data) from external sources are also passed to the checkpoint coordinator. After receiving a marker, the checkpoint coordinator will block the input channel until it has received markers from every input channel. Once it has received all markers, it emits a special marker, which informs source operators how to transform the markers from the external sources into a single unified marker for the current job.

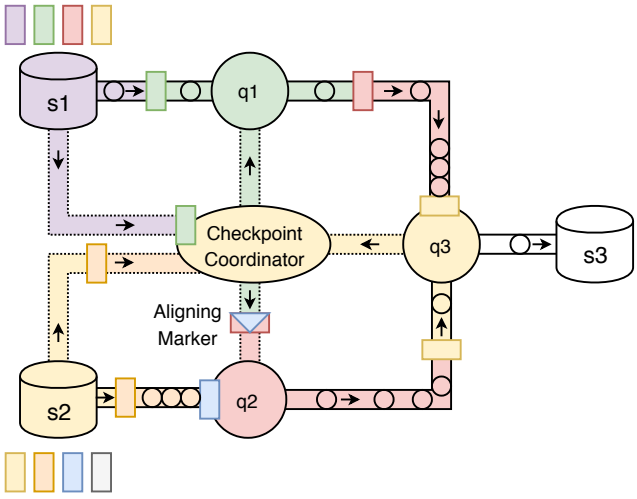


Figure 4.19: Example how markers from different sources can be merged. The colors under the sources indicate their respective checkpoint markers that have to be aligned.

Operators q1 and q2 behave the same way as any operator in the Flink checkpoint protocol. Upon receiving a marker, an input channel from q1 or q2 will be blocked until markers have been received from all input channels. As q1 and q2 are sources, one of these markers will be external, and the other will be the alignment information from the checkpoint coordinator. Using these two markers, q1 and q2 are both able to emit the same transformed marker.

Because we described "blocking" operations in both the checkpoint coordinator and the source operators, we need to proof that the system can not be trapped in a deadlock. If we assume that all external sources

are running, and will periodically emit a marker across all channels, it is trivial that the checkpoint coordinator will always receive a marker via an unblocked channel as long as it waits long enough. This guarantees that for every marker received across any input channel, a merging marker will be sent from the checkpoint coordinator to the source operator. Because the external source sends the same marker to the checkpoint coordinator and the source (q1, q2 in Figure 4.19), the source will always receive a marker from both incoming channels, and thus never block.

This, however, only shows that the marker merging does never completely block. It does not provide guarantees on the latency. One drawback of the protocol shown in Figure 4.19 is that checkpoint markers from each independent source are ingested at an equal rate. If one source is producing more markers than another, the source producing more markers will build up backlog, because it has to wait for markers from the other source to align with. Therefore, we have to add the requirement that each job in the global system emits markers at an equal rate. In Chapter 8, in which future work is discussed, we provide a suggestion on how this limitation can be avoided.

4.4.2. Cycles

In the previous section, the first requirement to prevent deadlocks was that all sources are running and emitting markers. When cycles are included this is not guaranteed, because a job can indirectly depend on itself. The marker merging should work with cycles as long as there are markers present in the loop, which is not guaranteed by the protocol (for example, upon launch no markers are present anywhere). The protocol, therefore, does not support cycles outside of the bounds of a job. As long as the cycle remains within the bounds of a job, there always is an unblocked path from the marker source to an operator with blocked channels, as explained in [22].

In summary, we can guarantee the protocol does not deadlock, under the condition that:

- All jobs that participate in the alignment form a DAG.
- All sources are running and periodically emitting markers at an equal rate.
- The messaging system between jobs delivers messages in the same order as received.

4.5. Properties of Aligned Jobs

Until this point all reasoning has been done around the "meaningful" state introduced by Chandy and Lamport [23]. The meaningful state means that the snapshotted state does not necessarily ever existed, but from the snapshotted state there is a path to the current state of the system. This is very useful when using the snapshot as a restore point, because the system will end up in the current state again. The motivation for aligning streaming jobs, however, is not to make a single restore point of all jobs combined. Most likely, these jobs were split because the user wants to be able to restore them individually in case of a failure. For our applications, we require the equivalent position described in Section 4.2, which a "meaningful" state does not necessarily provide.

The state of an operator in the Chandy-Lamport protocol consists of two components, namely the state of the operator itself, and the state of all incoming channels to that operator. One property of a stream processing framework, such as Flink, is that operators are only allowed to broadcast one or more messages after receiving a message. This leads to the conclusion that if in the snapshot all channels are empty, the system is idle and will only start processing once it receives external input. As explained before, the Flink checkpointing protocol enforces the channels to be empty in the snapshot, as long as the job does not contain a cycle. This means that, together with epoch alignment, an idle point can be identified where the two jobs have processed the exact same set of messages. This idle point is an equivalent position, and meets the requirements to perform a hot-swap of two streaming jobs, while maintaining the exactly-once processing guarantee.

4.6. Async I/O

Aside cycles there is another feature in Flink that is different from the ordinary DAG, namely async I/O. Figure 4.20 shows an async I/O operator. The idea of an async I/O operator is to send non-blocking requests to an external system, for example to enrich events with context stored in a database. When an async I/O operator receives an event, this event is stored in its operator state as "Outstanding request". After this, the operator performs the asynchronous request. Upon receiving an answer from the external service, the related event is removed from the operator state, and the response from the external service is forwarded to the next operator.

Upon receiving a checkpoint marker, the operator takes a snapshot of the state with outstanding requests, and passes the checkpoint marker on to the next operator.

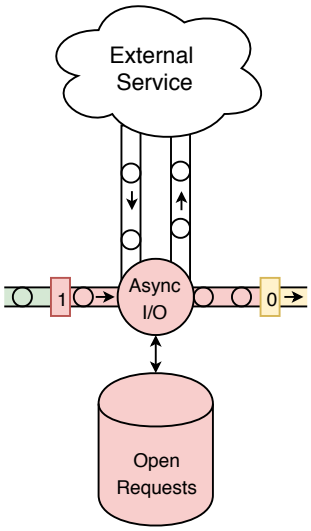


Figure 4.20: Example of an Async I/O Operator in Flink.

This async I/O operation fits in Chandy-Lamport's model and protocol. In [23] an "event" ("event" refers to an event in Chandy-Lamport's model) can consist of an operator performing a state mutation and optionally sending or receiving a message. In Chandy-Lamport's model an operator is allowed to spontaneously trigger an event based on its current state. In the case of async I/O, the state mutations are the addition and removal of the outstanding request list. The removal "event" is, from the jobs perspective, triggered spontaneously, because some event was present in the list of outstanding requests.

In Section 4.5 we made the assumption that one "event" consists of receiving a message, mutating state, and sending one or more messages. The async I/O operator breaks this assumption, because the receiving part and sending part happen independently of each other in separate "events", potentially in different epochs. This means we cannot perform a hot-swap operation with exactly-once (or even at-least-once) guarantee, if a job contains async I/O operators.

5 Implementation

This chapter describes the technical details and designs that were used to implement the epoch alignment protocol proposed in the previous chapter. Firstly, section 5.1 defines a number of terms used throughout this chapter. Next, section 5.2 explains how we used dependency injection in Apache Flink to make most developed components unit testable. Section 5.3 explains how ZooKeeper is used to manage out of job state, including the structure of the ZooKeeper state. The position of Kafka in our implementation is explained in section 5.4. The implementation of running aligned jobs is explained in section 5.5. Section 5.6 explains the protocol we used to transition a job from unaligned to aligned mode. The integration tests we deployed to validate end-to-end consistency in our implementation are described in section 5.7. Finally, section 5.8 describes the limitations of our implementation.

5.1. Overview and Terminology

The proof of concept was implemented by building a custom Kafka connector for Apache Flink. This custom connector has access to a cross-job shared state in Apache ZooKeeper. Since we deal with both Kafka and Flink, it is easy to get confused with terminology. For example, a *Producer* in Kafka is a component which writes data to Kafka. When looking from Flink's perspective, however, the *Producer* is the output of the stream, thus called a *Sink*. To keep the report consistent we define the following terms.

- **Job** refers to a streaming job in Flink. A Flink job can be modelled as a directed graph, where nodes represent operators and edges represent event flow between operators.
- **Operator** refers to a single node in a Flink job graph. An operator maintains a state, consumes incoming events, performs computations on incoming events, and produces output events. When the term operator is used we refer to all parallel instances of this operator in a Flink job.
- **Source** refers to the source operator in a Flink job. In a job graph a source is a special operator that does not consume events, but only maintains a state and produces events. The component that sends events to the source is not part of the job graph. When the term source is used we refer to all instances of this source in a Flink job.
- **Sink** refers to the sink operator in a Flink job. In a job graph a sink is a special operator that does not produce events, but only consumes events and maintains a state. The component that receives events from the sink is not part of the job graph. When the term sink is used we refer to all instances of the sink in a Flink job.
- **Checkpoint** refers to a position in a streaming job in Flink, indicated by checkpoint markers that are passed to the streaming job. Upon receiving a checkpoint marker, an operator takes a snapshot (backup) of its state at the position of the marker. When referred to events from checkpoint 2, we mean the events between the checkpoint marker 1 and 2.
- **Epoch** is used to refer to all data up to a certain checkpoint. For example, when referred to data from epoch 2, we mean all data up to the marker of checkpoint 2. So if a job starts at checkpoint 0, epoch 2 is equivalent to checkpoint 0, 1 and 2 together.

- **Consumer** is derived from the Kafka consumer. Since each parallel instance of a source has its own consumer, we use the term consumer to refer to a single instance of a source in a Flink job.
- **Producer** is derived from the Kafka producer. Since each parallel instance of a sink has its own producer, we use the term producer to refer to a single instance of a sink in a Flink job.
- **Kafka Consumer** is used to refer to the actual Kafka Consumer from the Java Kafka connector.
- **Kafka Producer** is used to refer to the actual Kafka Producer from the Java Kafka connector.
- **Topic** is used to refer to a Kafka topic.
- **Subject** is derived from the term Subject in Reactive Extensions (Rx) [38]. We use the term Subject to refer to a Topic together with its shared state in ZooKeeper. We chose the term Subject, because our implementation allows to use a Topic in a similar fashion to the Rx Subject.

5.1.1. Architecture

To help with building independent and unit testable components, we chose to use dependency injection (DI). For this we applied the cake pattern [30], a common DI strategy when developing in Scala. The main motivation for using the cake pattern is that it purely uses the Scala language features, simplifying integration of DI with API's exposed by Flink. The cake pattern allowed us to independently test the integration with the technologies used (Flink, Kafka and ZooKeeper) and the alignment algorithm. It also allowed us to implement the epoch alignment protocol itself independently from the other technologies.

The list below provides and overview of the components developed for the proof of concept. Figure 5.1 shows how these individual components relate to each other.

- **Data Generators** are components that produce sample elements which are used in the final experiments. They enable us to, based on a seed and offset, generate sample elements at a controllable rate.
- **Configuration** components provide a strongly typed wrapper and default values for the configuration of a deployment.
- **ZooKeeper Client** provides a thin Scala layer around the ZooKeeper API, providing Quality-of-life functionality, such as exposing Scala Futures instead of the callbacks from the native ZooKeeper API.
- **Kafka** components provide a Scala layer around the Kafka API, providing an API that is easier to use from a Flink source or sink. The functionality of this component is described in section 5.4.
- **Subject Library** is a component that exposes an interface for the ZooKeeper store, and is used as an entry point to obtain ZooKeeper Nodes (see below).
- **ZooKeeper Nodes** components provide a strongly typed API around the ZooKeeper state that is used by our proof of concept. The implementation of this component is described in section 5.3.
- **Stream API Components** combine the API's exposed by the other components, and provide the implementations of the Sinks and Sources for Flink. The integration with Flink is described in section 5.2.
- **Table API Components** is a small wrapper around the Stream API Components, to integrate with Flink's Table API. In this case, the type information from the subject is used, allowing direct use of the results from a job in another job without having to declare the data structure.
- **Synchronization Components** provide the implementation of the epoch alignment protocol, which is used by the custom sinks and sources. The implementation of the epoch alignment protocol is described in Section 5.6.

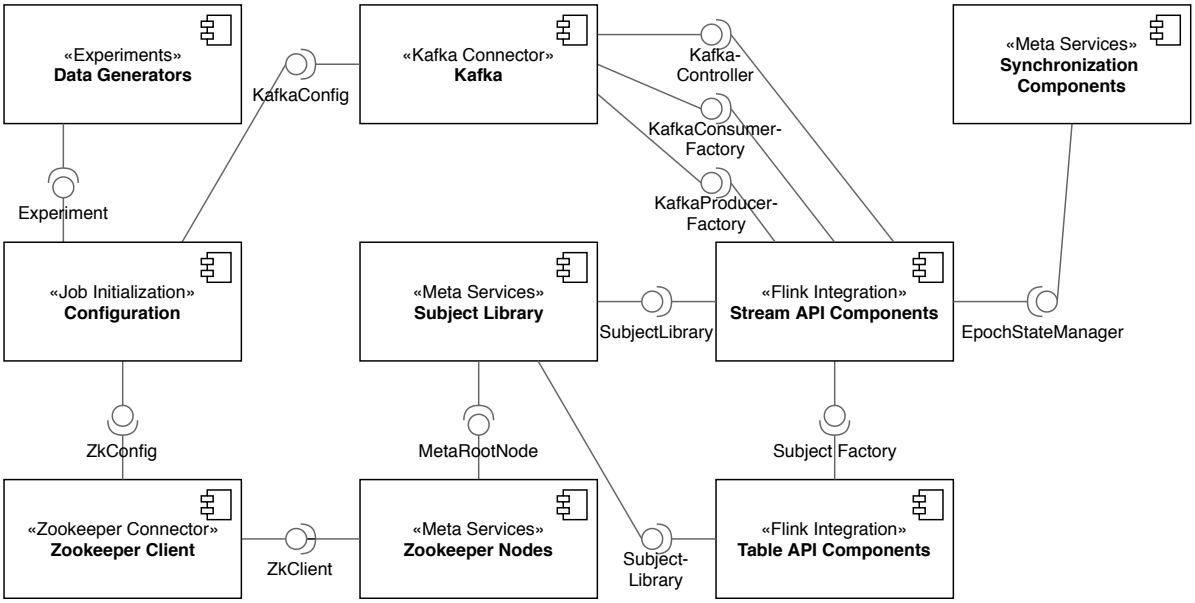


Figure 5.1: Overview of the core components in our proof of concept.

5.2. Flink

The core technology in our proof of concept is Apache Flink. We built our proof of concept by implementing a custom Kafka connector (both source and sink) for Flink, that is able to communicate a shared state with all sinks and sources using the same subject. This custom sink supports the at-least-once, exactly-once and aligned running modes. At-least-once and exactly-once are the native supported processing guarantees provided by Flink for most operators. When our custom Source is launched in exactly-once mode, a command can be passed via ZooKeeper to transition the source into aligned running mode. Because checkpoint markers are inserted at the source, and the only difference between exactly-once and aligned is the location of the checkpoint markers, the sink does not need to transition into the aligned state. The implementation fully supports parallel operators and partitioned streams.

5.2.1. Flink and Dependency Injection

A challenge we ran into when using the cake pattern in combination with Apache Flink, was that Flink operators are distributed, and therefore have to be serializable. This means that when an operator has a dependency to a particular component, that component must either be serialized alongside the operator, or reinitialized after de-serialization. This is recursive, meaning that for a component to be serializable, all its dependencies must also be serializable, or the dependencies must be re-injected upon de-serialization. From an architectural perspective, the latter would be preferred, because it will use the configuration from the target environment when re-injecting components. In addition, the latter would allow the DI framework to determine the creation policy (singleton, non-shared), where de-serializing would always create a new instance thus forcing the non-shared policy. Re-injecting upon de-serialization would, however, require some integration of Flink with the DI framework, which we considered out of scope for this thesis. Therefore, for the proof of concept, all components have to be serializable.

Some components, however, have non-serializable parts, such as the connection to ZooKeeper. An easy way to solve this, is to make these parts lazy volatile, meaning they are re-initialized whenever they are needed after de-serialization. However, as described in the previous paragraph, due to the de-serialization all components become non-shared. In the case of ZooKeeper, this means that any component with some direct or indirect dependency on ZooKeeper would initialize its own client, which introduces a large amount of connection overhead. That is why we pulled the ZooKeeper connection to a static context on the execution environment of a Flink job, while making sure that a single ZooKeeper client is maintained for every parallel instance of a job. That solution, however, breaks the DI pattern, forcing some additional scaffolding to make the ZooKeeper components unit testable again.

5.3. ZooKeeper

To align checkpoint markers, we need the source operators of individual Flink jobs to communicate with each other. This means the jobs share a common state outside the scope of a single job, which is not directly managed by Flink. To maintain processing guarantees, the individual jobs should not depend on this state. This means the shared state is not allowed to have an effect on the output of a job. In our implementation the common state in ZooKeeper only has an effect on the location of the checkpoint markers within the event streams. As long as the streaming job is implemented in such a way that checkpoint markers have no effect on the output, which is a general best practice, our implementation has no effect on the output.

The shared state needs to be fully scalable and distributed along with the Flink operators. The shared state must also be consistent when a job fails and performs a checkpoint restore. We have chosen for Apache Zookeeper [10] to manage the distributed state. It manages a scalable and distributed state, and has the required functionalities to keep this state consistent upon failures. Apache Kafka, described in 5.4, also uses Apache ZooKeeper to store and communicate metadata about its topics, meaning we can reuse the deployment of Apache ZooKeeper.

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services [10]. ZooKeeper stores data in a hierarchical structure, very similar to a file system. ZooKeeper offers a simple low-level API for *Create*, *Read*, *Update* and *Delete* operations, and also operations specific to a distributed state such as *Sync* (wait for data to be propagated) an Watches (observe a node on changes).

Because ZooKeeper, as manager of the common state, has a key position in the implementation of our algorithm, we designed an easy-to-use interface around ZooKeeper. A set of strongly typed hierarchical *Nodes* around the low-level API from ZooKeeper were developed. These nodes allow us to define and navigate the structure of the ZooKeeper state in a strongly typed fashion. The nodes greatly simplify the implementation of the algorithm in our custom sinks and sources. Also, the nodes are mockable, improving the expressiveness of our unit tests.

The strongly typed nodes expose generic implementations of higher level operations, such as distributed locks, aggregate states, leader election, and generic integration with Rx to obtain observables of state changes on ZooKeeper nodes or their children. The nodes itself hold no state or reference to ZooKeeper connectors, are serializable, and can be passed between execution environments. When calling a method on a node that requires access to the ZooKeeper store, access to the store is resolved on the local system that holds the node.

The structure of the ZooKeeper state as hierarchical nodes is shown in Figure 5.3. This structure of hierarchical nodes is created by implementing one of the four base traits shown in Figure 5.2. (Note that Scala Traits, which are similar to Interfaces in Java, can provide a default implementation for methods defined by the interface). The following subsections describe the generic functionality provided by the components from Figure 5.2. The structure of the state itself is described in section 5.3.5

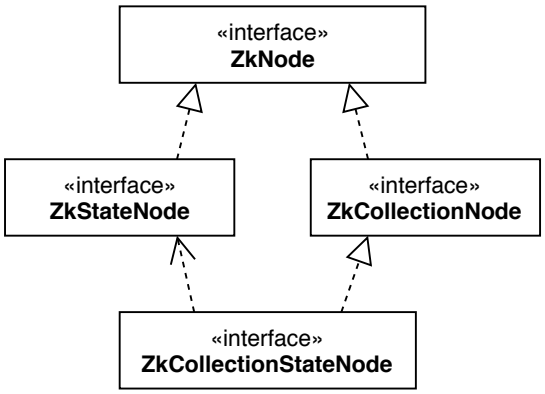


Figure 5.2: Traits providing basic ZooKeeper functionality.

5.3.1. ZkNode[TData]

The base type for all ZooKeeper nodes is `ZkNode`. The `ZkNode` implementation offers basic operations that can be performed on every node. Key methods provided by the `ZkNode` trait are:

- `create(data:TData): Future[TData]` Creates the node with the passed data. Deals with serialisation and racing conditions when the node is created by multiple workers.
- `awaitCondition(condition: TData => Boolean): Future[TData]` Returns a future that completes when the data stored at the node matches the passed condition.
- `observeData():Observable[TData]` Returns a Rx Observable that fires when the data is changed. Does not guarantee to be fired for each individual mutation (at-most-once guarantee).
- `readLock(): Future[ZkReadLock]` Returns a future that completes with a read lock on the node.
- `writeLock(): Future[ZkWriteLock]` Returns a future that completes with a write lock on the node.
- `awaitChild(child: String): Future[String]` Returns a future that completes when a child of the given name has been created.
- `exists(): Future[Boolean]` Returns a future that completes with a boolean if the current node (still) exists in the ZooKeeper state.
- `awaitRemoval(): Future[Unit]` Returns a future that completes when the node is removed.

Apart from the provided functionalities, `ZkNode` also allows overriding of some hook methods, such as `PostCreate` or `PostUpdate`. For example, `PostCreate` is used to automatically create all required child nodes when a new subject is created. `PostUpdate` is used in combination with locks in the PartitionOffset node from Figure 5.1, such that the last process writing its partition/offset of an epoch will also update the partition/offset of the entire epoch. In Figure 5.1 the yellow nodes are general ZkNodes. All other coloured nodes are more specific instances described below.

5.3.2. ZkCollectionNode[TChild,TData]

One of the extensions on the `ZkNode` is the `ZkCollectionNode`. A collection node manages children of a specific type. In Figure 5.1 examples of collection nodes are sources, sinks, producers etc. Most of these are the more specific `ZkCollectionStateNode` described below.

- `getChildren():Future[` Returns a future of the strongly typed children of the node.
- `awaitChildNode(name:String)` Returns a future that completes when a child of the given name has been created. Completes with a strongly typed instance of the child.
- `observeNewChildren` Returns a Rx observable which is fired for each newly created child (exactly-once guarantee).

5.3.3. ZkStateNode[TNode, TState]

Another extension of `ZkNode` is `ZkStateNode`. A `ZkStateNode` does not only hold data, but also has a child of the type `ZkNode[TState]`. `ZkStatNode` is mainly used in combination with `ZkCollectionStateNode` described below. An example of a state node in Figure 5.3 is the source node, where the state is a boolean which indicates whether the source is still active.

`ZkStateNode` exposes methods such as:

- `getState(): Future[Option[TState]]` to retrieve the state
- `setState(state: TState): Future[Unit]` to modify the state
- `watchState(f: TState => Boolean): Future[TState]` returns a future that completes whenever the state meets some condition.

5.3.4. ZkCollectionStateNode[TChildNode <: ZkStateNode[TChild, TChildState], TData, TChild, TChildState, TAggregateState]

`ZkCollectionStateNode` is an extension to the `ZkCollectionNode` when the children of a collection are `ZkStateNodes`. The `ZkCollectionStateNode` provides an interface to aggregate the state of the children into a single state, including hooks and locks to automatically update the state of the aggregate collection when one of the children is updated. An example of this usage in Figure 5.3 is the `Sources` node, which aggregates the boolean state from the source node into a boolean which indicates if a subject still has active sources.

Key methods exposed by `ZkCollectionStateNode` are similar to the `ZkStateNode` methods, but exposing the aggregate state:

- `getState(): Future[TAggregateState]` Returns a Future of the aggregated state of all children.
- `watchStateAggregate(f: TChildState => Boolean): Future[Boolean]` Returns a future that completes when the condition holds true for the state of all children.

5.3.5. Shared State

Using the generic nodes described above, we have designed a tree structure of ZooKeeper nodes shown in Figure 5.3. The central part and root of the tree is the subject node. The subject node holds information about how to connect to the endpoint exposing the subject, and a schema definition for the data exposed by the endpoint. In our proof of concept we will only use Apache Kafka [3] topics as endpoints for our subjects. For Kafka the subject contains a topic name along side the type information.

The tree structure, in Figure 5.3 below the subject node, is created by individual jobs using the subject. A job reading data from the subject will create a source node, and for each parallel instance a consumer node. When the source is running aligned it will also store partition and offset data under the SourceEpochs node, providing a mapping from epochs of the source to epochs of the subject. Each upon initialization source will also create a command node, which allows an external process to trigger alignment of the source with its subject.

A job writing data to a subject will create the sink and producer nodes, and continuously update its progress by writing partitions and offsets to ZooKeeper. In our custom implementation of the sink, the sink will also automatically update the epochs of the subject, using the partitions written by all producers of all sinks that are writing data to the subject.

The epochs of the subject can, however, also be constructed by a separate process, independent from Flink. This is useful when the data on the Kafka topic is produced by an external system, which does not integrate with our ZooKeeper state. In that case, our sources can not align with the external sink, but using the subject-epochs, multiple sources using the same subject can still align with each other.

The ZooKeeper state in Figure 5.3 also includes aliveness of consumers, sources, producers and sinks. This structure was initially designed for statistical analyses on the Codefeedr platform, but has also proven helpful in the integration tests described in section 5.7. However, this aliveness data plays no role in the alignment algorithm itself.

Figure 5.3 also contains a completely separate tree under the "Jobs" node. This tree is used to guarantee that all epochs that contain data are completed before a job terminates, as further described in section 5.7. This part of the tree was kept independent of the main state, because it is only used when running integration tests.

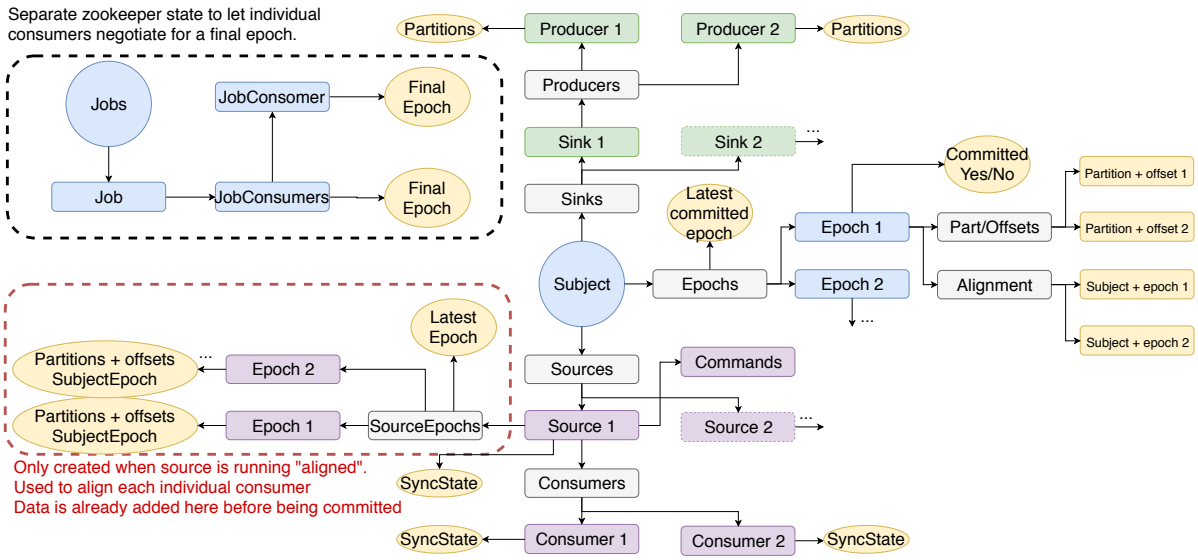


Figure 5.3: Structure of the state kept in ZooKeeper.

5.4. Kafka

A landscape of streaming applications consists of numerous streams (Flink Jobs), which are linked to external systems or to each other. To link streaming jobs to each other, a publish-subscribe mechanism is desired. This publish-subscribe mechanism should be scalable and fault-tolerant, and also guarantee message ordering. Apache Kafka [3] is a popular streaming platform that meets these requirements. Kafka guarantees message ordering only within a so called partition, which has been taken into account while implementing the alignment protocol.

Flink comes with native support for Apache Kafka. However, since we require very specific (aligning) behaviour of the connector, we decided to implement our own sources and sinks. Our custom connectors use the type information stored in ZooKeeper to fully integrate with Flink's Table API and SQL API. This allows us to define chained jobs with SQL Statements only without compiling code.

Using the connector provided by Kafka, the implementation of a custom source and sink is straight forward. In the implementation of the source, Flink's operator state is used to keep track of the consumed offsets. More complexity is introduced by implementing the alignment protocol, explained in the next section.

5.5. Alignment

The alignment of epochs is implemented by aligning every source with the subject-epochs present in the ZooKeeper state. A subject epoch is determined by the sink(s) that produced the events for the subject. When multiple Flink sources use the same subject, aligning these sources with the subject will also align the sources with each other. Because the subject-epochs are determined by the sinks from upstream jobs, the checkpoint markers effectively flow from a sink through the subject to the next source. See Figures 5.4 and 5.5 for a visual representation of the alignment for a subject with two sources and sinks.

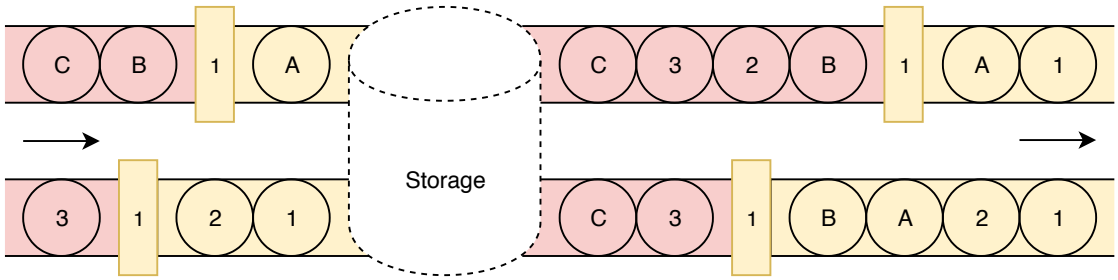


Figure 5.4: A subject with two sinks (left) and sources (right), unaligned.

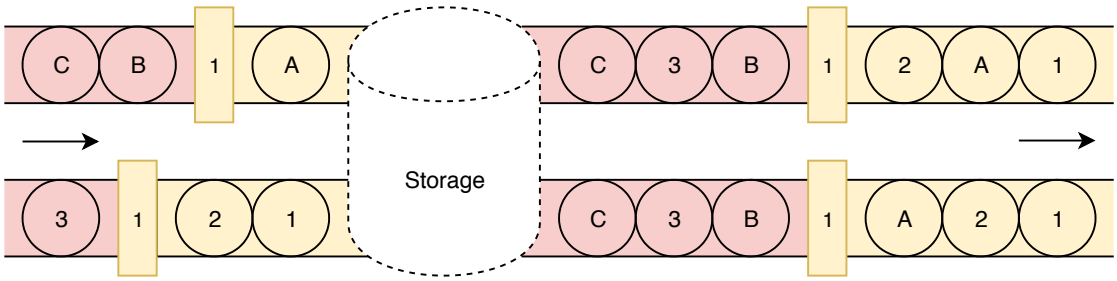


Figure 5.5: A subject with two sinks (left) and sources (right), aligned.

The alignment does not guarantee the order of events from different sources, which is also shown in Figure 5.5. This behaviour actually depends on the Kafka partitions. The order is guaranteed within a single partition, but not across different partitions. Therefore, our implementation of epoch alignment requires each individual sink operator in Flink to publish its events to a separate set of Kafka partitions.

5.5.1. Separate partitions for each producer

As mentioned in section 5.3.5, producers are not allowed to share partitions on Kafka. Having a single producer send data to one or more private partitions is disadvantageous, because it risks unbalancing the Kafka partitions, which in turn risks unbalancing consumers of the next job. This is also why the default behaviour of a Kafka producer is to distribute events round-robin over all partitions. Flink's native Kafka connector, however, does not use round-robin routing, likely because it would remove any processing order guarantee.

Another disadvantage of not publishing data round-robin to partitions, is that jobs publishing data to some topic, can not be added or removed while other jobs are running. In that case, partitions must be added while the system is running, and it must be ensured that the consumers support on-the-fly addition and removal of partitions. Another choice may be to initially create each topic with a large number of partitions. In that case, however, partitions will certainly be unbalanced, since a set of partitions will be empty.

The reason we need separate partitions for each producer is best shown graphically. Figures 5.6 and 5.7 show two instances, which can occur when two producers write to the same partition. Figure 5.7 shows a situation where no checkpoint marker can be injected in the downstream job, because some event from epoch 2 has been emitted before all events from epoch 1 have been emitted.

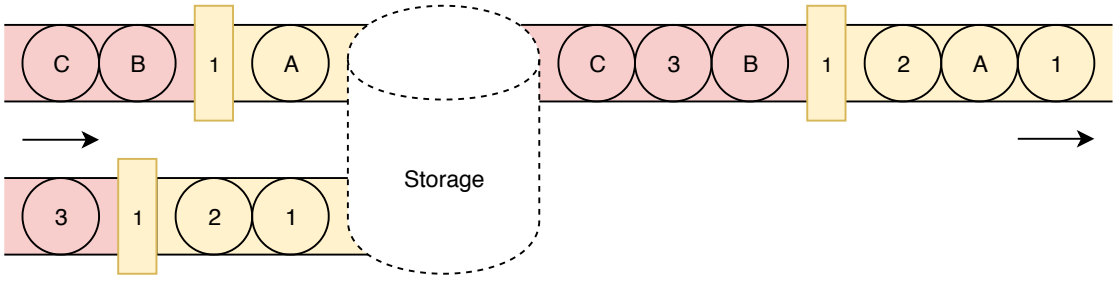


Figure 5.6: Two instances of two producers writing to a single partition. Ideal situation, where a checkpoint marker can be injected.

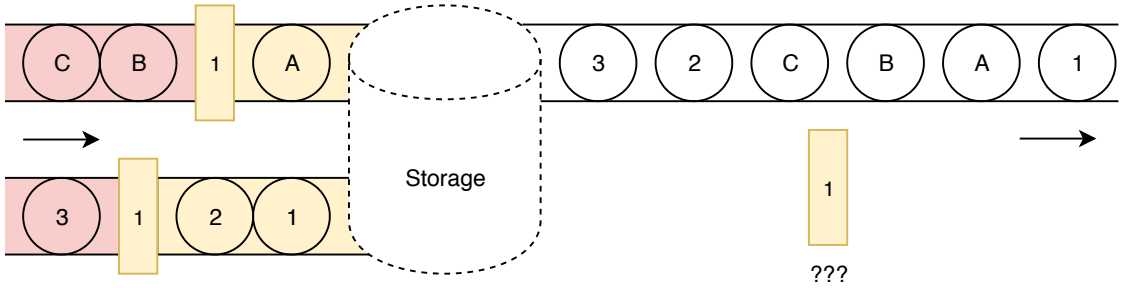


Figure 5.7: Two instances of two producers writing to a single partition. Possible situation, where the checkpoint marker cannot be injected on the downstream job (right), because at every position it will either include one or more messages from epoch 2, or miss one or more messages from epoch 1.

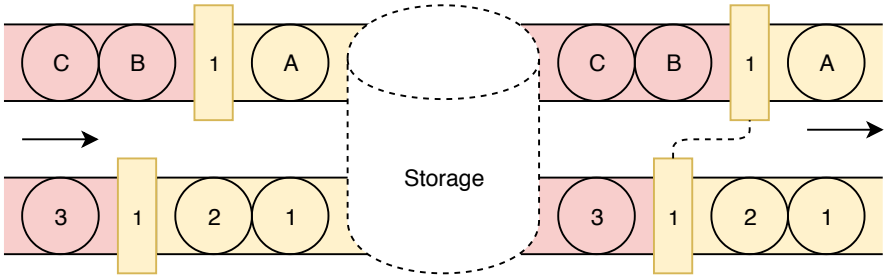


Figure 5.8: Two producers writing to their own partitions, which makes it possible to inject a checkpoint marker for both partitions.

In the case of multiple producers writing to a single partition, Kafka supports atomic reads and writes. This is done by keeping track of a Latest Stable Offset (LSO) for each partition [4, 5]. When the consumer is configured to only read committed messages, all messages past this LSO will be kept in a buffer until the LSO progresses. Mapping this concept to Figure 5.7, message "B" will not be emitted until CP2 of the bottom producer is committed.

This means that Kafka does support atomic writes, but it also means that a message from the next epoch (Message "2" in Figure 5.7) can be emitted before a message from the previous epoch (Message "B" in Figure 5.7). This breaks the message ordering requirement, which is unacceptable for our use case, because we want the messages seen by epoch 1 on the downstream (right) side to be exactly equal to the messages seen by epoch 1 on the upstream (left) side. When we let each producer write to an individual partition, message ordering is guaranteed (See Figure 5.8).

Through configuration, the downsides of having an individual partition per producer can be avoided to a certain extent. The unbalancing problem can be solved by performing a rebalancing step in Flink, before writing the data to Kafka (this may change message ordering, but cannot move messages into a different epoch). However, when multiple jobs produce elements for the same topic at a different rate, unbalanced partitions can not be prevented. In that case, it is still possible to ensure that consumers of a source reading from a topic, are assigned to a balanced set of partitions, such that each consumer receives a balanced amount of events.

5.5.2. Generating subject-epochs

The first part of the alignment is creating epochs on the subject. When using one of our custom sinks, these epochs are created automatically alongside the two-phase commit implementation of the Kafka connector. The producer keeps track of the partitions and offsets of produced elements, because this is required for the commit operation for Kafka.

During the pre-commit the following steps are performed by each producer of the job.

- Obtain a write lock on the EpochCollection node.
- Check whether an epoch is already available for the current checkpoint increment. If not, create the epoch node with the state "False" (indicating the epoch has not been committed).

- Release the write lock on the EpochCollection node.
- For each partition that the current producer wrote to, write a partition/offset node under the created epoch node. This partition/offset node is created with the state "False".

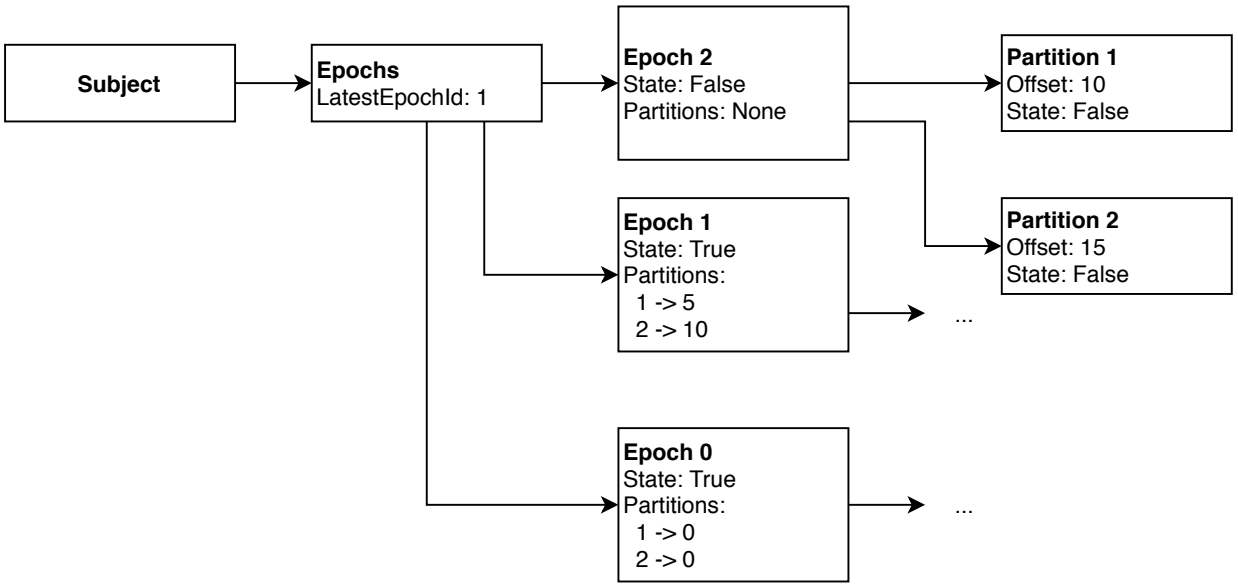


Figure 5.9: ZooKeeper state after performing pre-commit on a subject with two partitions.

After the pre-commit, the offset data has been stored in ZooKeeper, but the epoch is still marked as uncommitted. Each individual partition/offset is also marked as uncommitted. Figure 5.9 shows the ZooKeeper state of a subject and its epochs, after a pre-commit of checkpoint 2 has been made on a sink that writes data to two partitions. The partition nodes are instances of the ZkStateNode described in section 5.3, and the epoch node is a ZkCollectionStateNode.

After the pre-commit phase has completed successfully, commit is called. The commit operation consists of the following steps performed by every producer:

- For every partition under the EpochNode, set the state to "True".
- Obtain a write lock on the EpochNode.
- Check if the state of the epoch node is "True". If so, release the lock and do nothing, because another producer already performed the remaining steps. If not, continue with the remaining steps.
- Obtain all partitions of the epoch, and check if all are in the "True" state. If so, continue with the remaining steps. If not, release the lock and do nothing, because another producer still needs to commit its partition/offsets.
- Collect all partition and offset data from the partition/offset nodes, and store the aggregation as a single collection in the EpochNode.
- Set the state of the epoch to "True".
- Release the write lock on the epoch node.

Figure 5.10 shows the resulting state in ZooKeeper after the commit operation has been completed. The partitions are also stored in the EpochNode, so consumers only need to read one node for each epoch, reducing the amount of ZooKeeper operations.

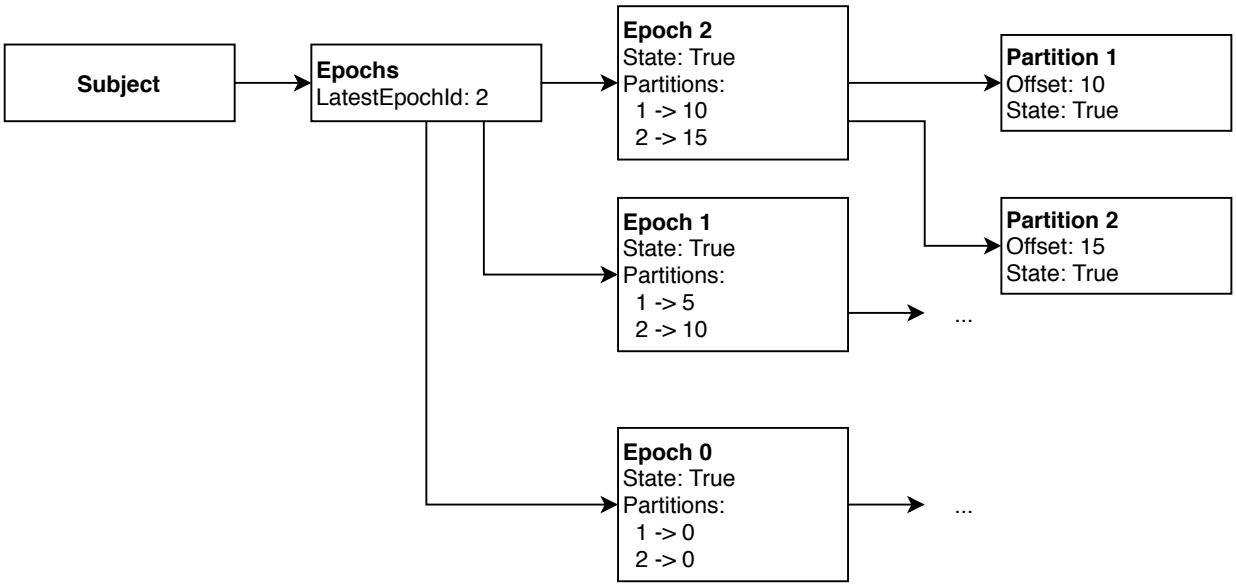


Figure 5.10: ZooKeeper state after performing commit on a subject with two partitions.

State for each offset

When using only a single job, it seems unnecessary to have each individual partition/offset node also keep a committed state. In the two-phase commit it is not possible to have only a part of the transactional components in the committed state. Once a commit is called on a component, one can assume that for each component commit will be called.

However, it is also possible to have multiple jobs write to the same subject. In that case, the situation may arise that some partitions of the epoch are committed while others are aborted. A potential optimization would be to keep track of which partition/offset belongs to which job, and perform the commit step on job level instead of producer level.

Scalability

A careful reader might notice that the implementation of the pre-commit and commit use a write lock on the ZooKeeper state. This means that some operations that are performed by all producers are executed sequentially and not in parallel. We did not consider this to be a significant issue, because the locks are only used within the pre-commit and commit operations. This means that the factors that ultimately affect performance in terms of scalability, are the checkpoint interval (which determines the commit frequency), and the parallelism of the job (which determines how many locks are required per commit), and not the number of events.

Apache ZooKeeper supports tens of thousands of writes and reads per second, and the lock and state mutations performed within the lock require only a few operations. This should push the amount of parallel workers that is required for the locks become a significant bottleneck beyond a realistic number, even with low checkpoint intervals.

However, in 6 we are observing an increasing latency with an increasing parallelism. The current high amount of ZooKeeper reads and writes is a potential cause of this issue. The ZooKeeper operations could be further optimized, and be performed asynchronously, without blocking the operator. In the current version, however, Flink's PreCommit does not allow asynchronous operations. This subject should be investigated in future work.

External producers

If the data is written to Kafka by an external system, the epoch nodes can also be created periodically by an independent process. In that case, multiple sources using the same subject can still align with each other, but are not aligned with the external system producing the data.

5.5.3. Aligning the source

Having a subject with epochs stored in ZooKeeper, the sources only need to align with the epochs of the subject. As addressed in section 4.4, because the sources align with the same epochs, the sources are auto-

matically aligned with each other.

The operations performed by each individual consumer at the beginning of each epoch are shown in Figure 5.11. It seems that a lot of operations have to be performed within a lock on the SourceEpochCollection, however, only the first instance to enter the lock has to deal with a non-existing epoch node on the first decision. All remaining instances will exit the lock immediately and retrieve the data created by the first instance. In addition, this could be further optimized by having a designated leader, but we did not consider it necessary for the proof of concept.

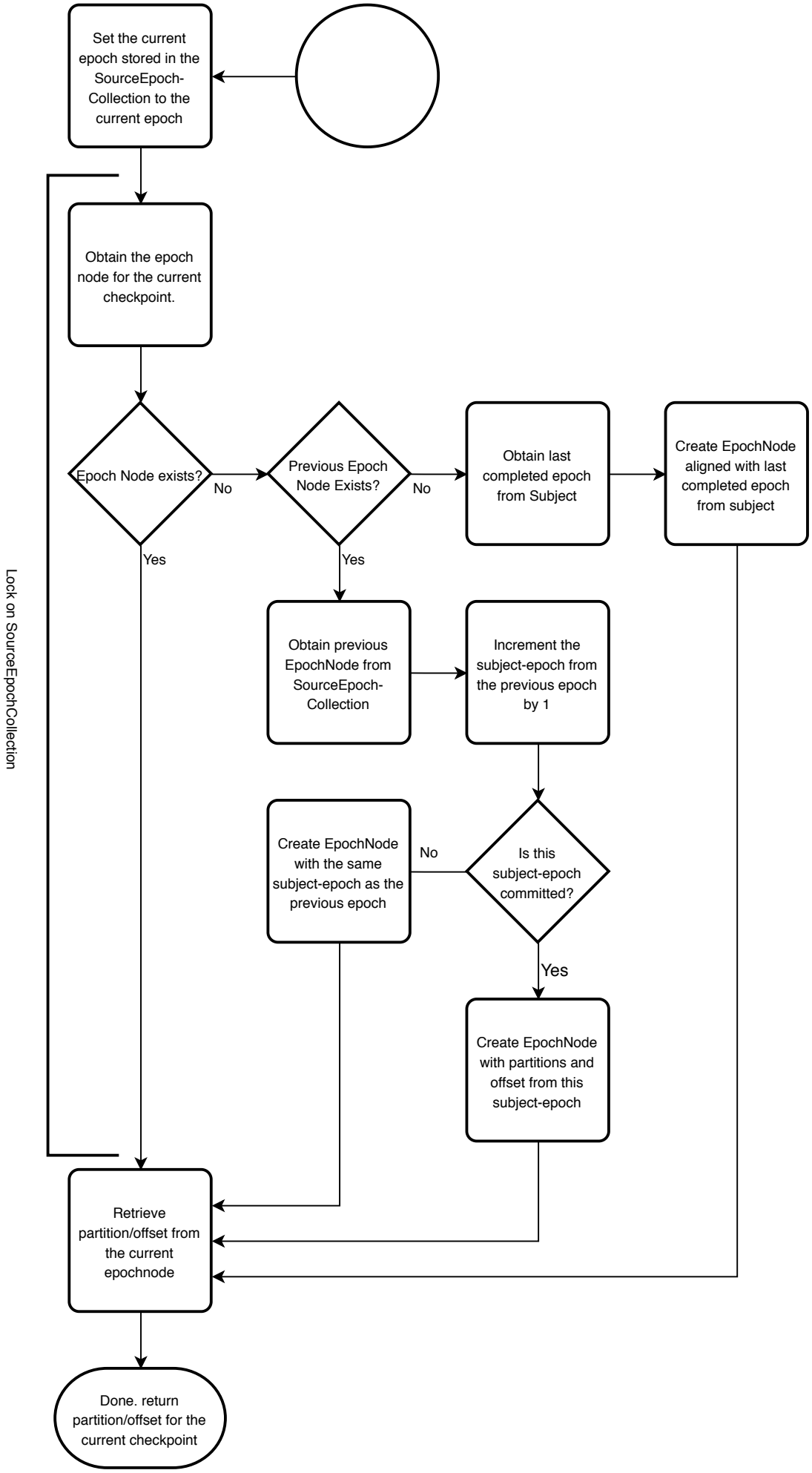


Figure 5.11: Operations performed by each consumer between each checkpoint.

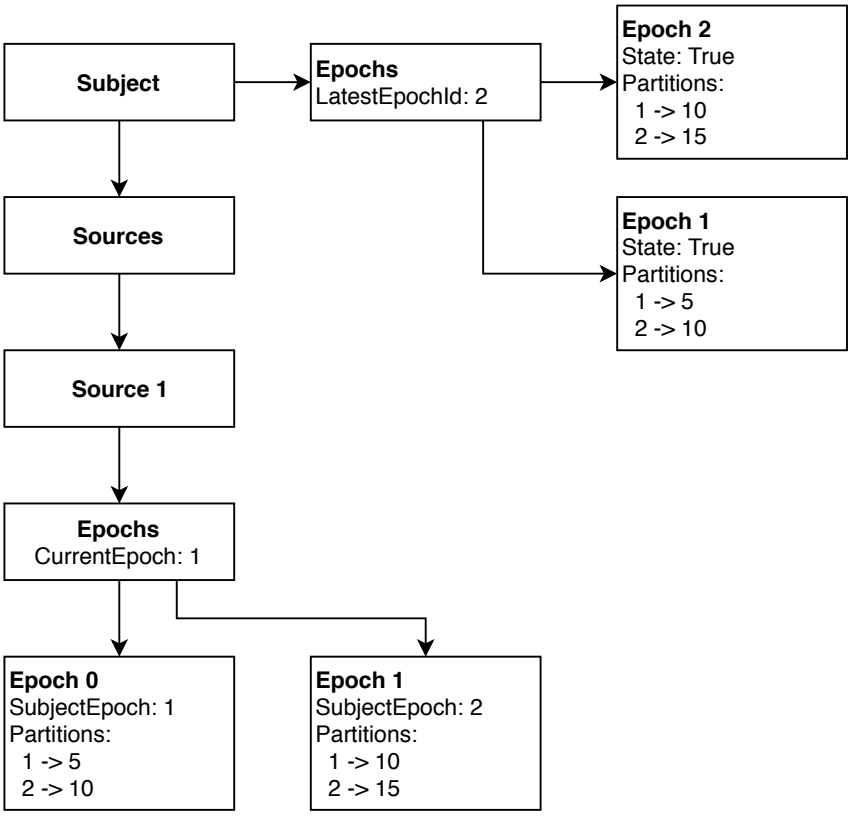


Figure 5.12: Example of the ZooKeeper state for Source 1 after performing the checkpoint for Epoch 0.

In normal operation, the first instance to enter the lock will increment the subject epoch from the previous epoch by 1, and create an epoch node with the partitions belonging to that subject epoch. An example of the ZooKeeper state after performing these steps is shown in Figure 5.12. The other paths of the flowchart in Figure 5.11 deal with the base case where there is no previous epoch, and the case where the subject-epochs are created at a slower rate than the current job is performing its checkpoints. If the subject is producing epochs at a higher rate than the consumer is performing checkpoints, the consumer will get behind, which is discussed in section 5.8.

After performing the initial steps, the consumer has obtained offsets it must reach for the current checkpoint. The consumer will keep retrieving data from Kafka until it has reached exactly these offsets. After that, it will no longer read any data until the next checkpoint starts. Vice versa, it will not accept a checkpoint when it has not yet reached the offsets. This means that at each checkpoint the epochs of the source are aligned with the epochs of the subject. The state in ZooKeeper contains a mapping from the source epoch to the subject epoch.

5.6. Alignment protocol

The previous sections described the behaviour of producers and consumers when running in the aligned state, but not how the source reaches this aligned state. Our implementation supports transitioning from an unaligned state into aligned and back, as long as the job runs with exactly-once processing guarantee. This is done by a few state transitions which are coordinated by an, from the jobs perspective, external alignment coordinator.

The ZooKeeper state in Figure 5.3 shows that each source has a CommandNode, and each consumer and source have a synchronization state. The CommandNode is observed at every checkpoint, and allows the alignment coordinator to communicate with the consumers. The synchronization state in its turn is observed by the alignment coordinator. Figure 5.13 shows the state transitions during the alignment process.

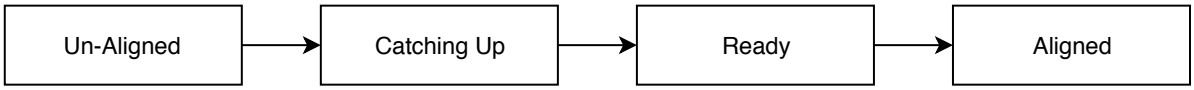


Figure 5.13: States of a source when aligning.

Initially, the coordinator will send the "CatchUp" command to the source. Upon receiving the command, each consumer will transition into the "Catching Up" state. In this state, after each checkpoint, the consumer will read offsets of the second last completed epoch of the subject. If the consumer is ahead on all partitions compared to those offsets, the consumer will transition into the "Ready" state. Once all consumers of the source have transitioned into the "Ready" state, the source itself will transition into the "Ready" state.

When the source has transitioned into the "Ready" state, the alignment coordinator chooses a future epoch increment and sends the "Align(EpochNr)" command. Upon receiving the "Align" command, all consumers transition into the "Align" state when the chosen epoch increment is started. When a consumer is in the "Aligned" state, it will perform the steps described in section 5.5.3, and run aligned with the subject.

The alignment coordinator is allowed to give the Abort command during any state of the alignment process, also when the source is in the aligned state. Because the alignment algorithm does not affect the results of the streaming job itself, aborting will just put the source back into the unaligned state. In the unaligned state, the source allows Flink to inject the checkpoint marker whenever it wants. The alignment coordinator is responsible for cleaning up the state in ZooKeeper after Abort is called.

5.7. Testing

The functionality of the individual components was tested with unit tests written in ScalaTest. Because we chose to use the cake pattern, writing unit tests for the individual components was straightforward. Because the key functionality of the proof of concept is in the communication between individually running jobs, automated integration tests were also employed.

For these integration tests an environment with Flink, Kafka and ZooKeeper was deployed with Docker. In this environment we created jobs with Kafka topics in between. Each parallel instance of a sink (producer) updates its ZooKeeper state when it completes. When all producers of a subject are completed, the subject is marked as inactive. When a source is connected to an inactive subject, it will retrieve all data from the subject and then terminate. This mechanism ensures that chained jobs with Kafka in between are naturally terminated. The natural termination allows us to wait until a job is finished in our tests, which allows us to compare the results in a finite environment.

One challenge we had to overcome in this process was that all communication with ZooKeeper is hooked to the checkpoint mechanism. Flink guarantees that its execution environment naturally terminates when all sources have terminated, and all events are processed. However, Flink does not guarantee that the last epoch that contains data, will fully complete. This means that, when a job is executed with exactly-once semantics, the last commit is not guaranteed to happen. The Kafka connector does not read uncommitted data when running in exactly-once semantics. This means that a consecutive job will not read all data when the source job terminates. In a production environment, streams are generally assumed to be indefinite, so this is not a problem. In our controlled test environment, however, this leads to issues, because we compare the results of finite streams.

This challenge was solved by creating an additional ZooKeeper tree for every independent job, also shown in Figure 5.3. Every producer and consumer of a job register in this tree. At every checkpoint, each registered operator will update its current checkpoint increment in ZooKeeper. When a consumer completes, it will not notify Flink of its completion until it observes that all operators of the same job committed the last epoch for which they had produced data. Once a producer has confirmed that all operators have committed the last epoch, it will naturally terminate and thereby notify Flink of its completion. This implementation allowed us to run streaming integration tests with finite datasets.

5.8. Limitations

In Chapter 5 we mentioned a number of limitations caused by the implementation of the alignment protocol. This section summarizes the limitations and, where possible, proposes potential solutions to these limitations.

The first limitation is that each producer is required to push data to its own private set of Kafka partitions

as described in 5.5.1. Solving that problem would require a publish-subscribe system that exposes the exact transactions from the publisher to the subscriber. It is likely, however, that such a system would introduce more overhead than the workaround proposed in 5.5.1, by using a large number of partitions, and balancing the partitions so that consumers receive a balanced number of events.

Secondly, for the current proof of concept, no work has been done to clean up the ZooKeeper state. In our experiments, the ZooKeeper state never becomes large enough to cause problems. For a production ready solution, however, the old epoch nodes should be cleaned up or reused.

Another limitation is that we only support deployments where both the upstream jobs and downstream jobs run at the same checkpoint interval. Future research should investigate how the offset selection can be made more flexible, while still producing aligned epochs at a regular interval. At the end of this report we provide ideas that could form the base for this future work.

The final limitation is that the subject epoch to be aligned with is determined at the start of a checkpoint at the downstream job. This can potentially increase the latency, when the upstream job commits its epochs just after the downstream job starts a new one. This has been confirmed by our experiments, and is explored in detail in Section 6.4. Also for this limitation, more research is required to investigate how the offset selection can be more flexible, while still producing aligned epochs at regular intervals.

6 Experiments

This chapter describes the experiments we performed to measure the effects of the epoch alignment protocol on the performance of streaming jobs. We start by explaining the experimental setup in Section 6.1. We continue by obtaining a baseline, by running streaming jobs without Kafka topic in between, and measuring the effects of various parameters in section 6.2. Next, we measure the effects of introducing a Kafka topic between two streaming jobs in section 6.3. We measure the effects of epoch alignment on latency, and provide an analysis of the results in section 6.4. Finally, section 6.5 summarizes the experimental results.

6.1. Experimental Setup

Aside from integration tests validating the correctness of our implementation, performance has been measured via experiments. For these experiments we deployed two machines with sufficient cores and memory such that each parallel instance has its own resources. One machine was used to run Apache Kafka and ZooKeeper, and the second machine to run Apache Flink. For our Flink deployment, we used a single task and job manager, with one task slot for each core of the machine. All experiments were performed within docker containers, allowing us to easily redeploy clean environments for every experiment.

The measurements are taken by a minor extension we made to a Flink operator, which we can wrap around any custom operator in Flink. This extension hooks into the checkpointing behaviour where upon a checkpoint it will write a log line with the following information:

- *CheckpointId:* Current number of the checkpoint.
- *Offset:* Total number of events processed by the operator.
- *Count:* Total number of events processed during the currently measured checkpoint.
- *Latency:* The worst case latency that occurred while processing events during the current checkpoint. Latency is calculated by subtracting the event-time from the local time of the processing operator.
- *Checkpoint Latency:* Worst case latency if all events were processed at the moment the checkpoint marker passed through the current operator. This is calculated by subtracting the oldest event-time processed by the operator during the checkpoint from the local time of the operator when the checkpoint marker was received. When working with exactly-once processing semantics this measurement is more accurate for the latency, as the receiver has to wait for the checkpoint to complete before processing the events.

In our case it is safe to work with local time, because all operators are deployed on a single machine, meaning that the difference in local time between independent operators is insignificant compared to the other effects that will be measured.

We collect the events logged by our operator extension by running an independent system that observes the docker-logs from the machine running Flink and writes the output to Elasticsearch. During the experiments, we observed the real-time performance in dashboards configured in Kibana.

6.1.1. Simulation

In our experiments, we simulate a query that computes *Hotness* of pull-requests. We define *Hotness* as the number of comments that are published directly to the pull-request, combined with the number of comments that are posted to issues that are related to that pull-request. The pull-request-comments, issue-comments and issues are each modelled as independent event sources. This query simulates a use case for the codefeedr [27] project.

The query is shown schematically in Figure 6.1. We simulate an event source producing comments belonging to issues. These comments are reduced within a tumbling window to an *IssueId* and *Count* of number of comments. These reduced values are joined within a tumbling window to the issue itself. The result is mapped to an object containing *IssueId*, *PullRequestId* and *Count*. The resulting values are the number of comments, IssueId and PullRequestID. We call this intermediate result the hot issues.

For pull request comments we follow a similar route, where we perform a reduce within a tumbling window to map each comment to an object with *PullRequestId* and *Count*. Finally, the streams of hot issues and reduced pull requests are unioned, and the results are reduced to a single object containing the *PullRequestId* and *Count*, where the *Count* represents the hotness of the pull request.

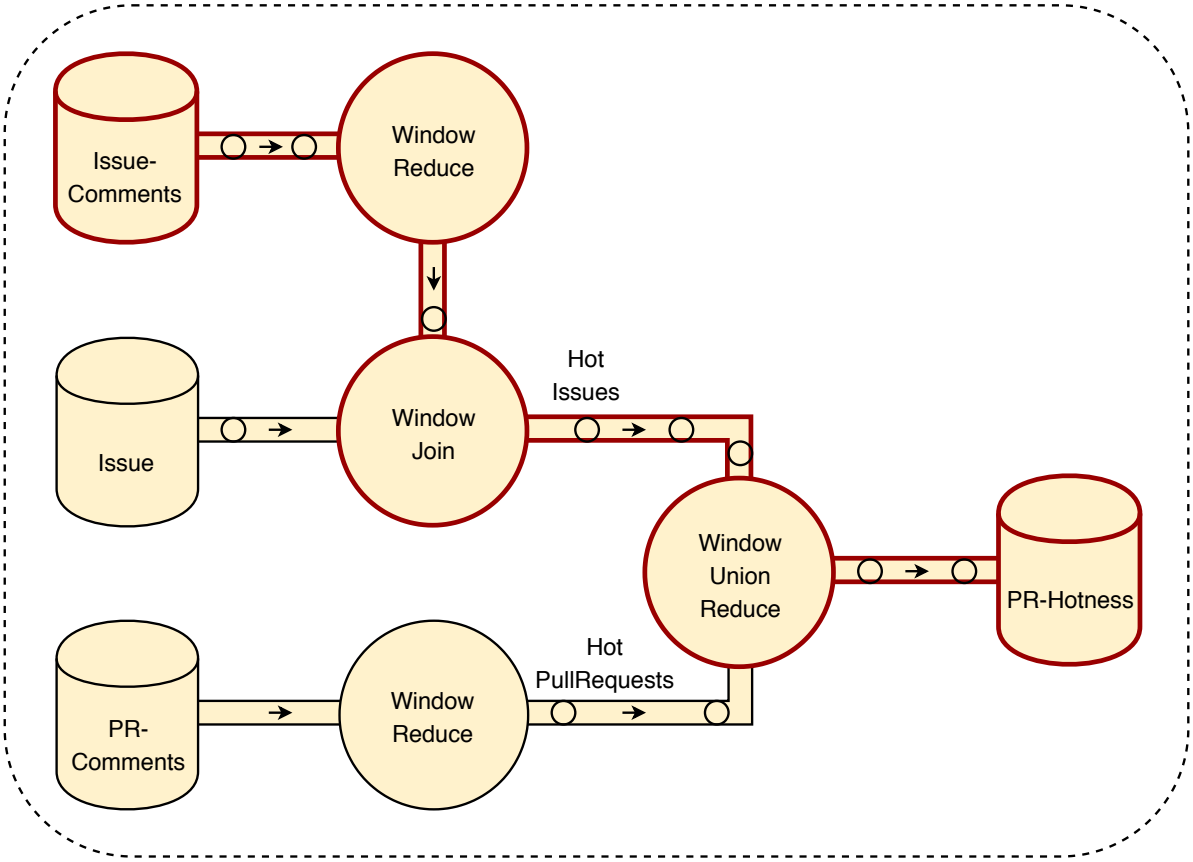


Figure 6.1: Schematic representation of the Hotness query for pull-request. Latency is measured across the red path at every operator (circle).

To test the alignment of two consecutive jobs, we split the query into two separate jobs. The first job computes the hot issues as described in the previous paragraph and outputs the results to Kafka. The second job computes the hot pull requests by merging the hot issues from Kafka with locally generated pull-request-comments. The two independent jobs are shown schematically in Figure 6.2.

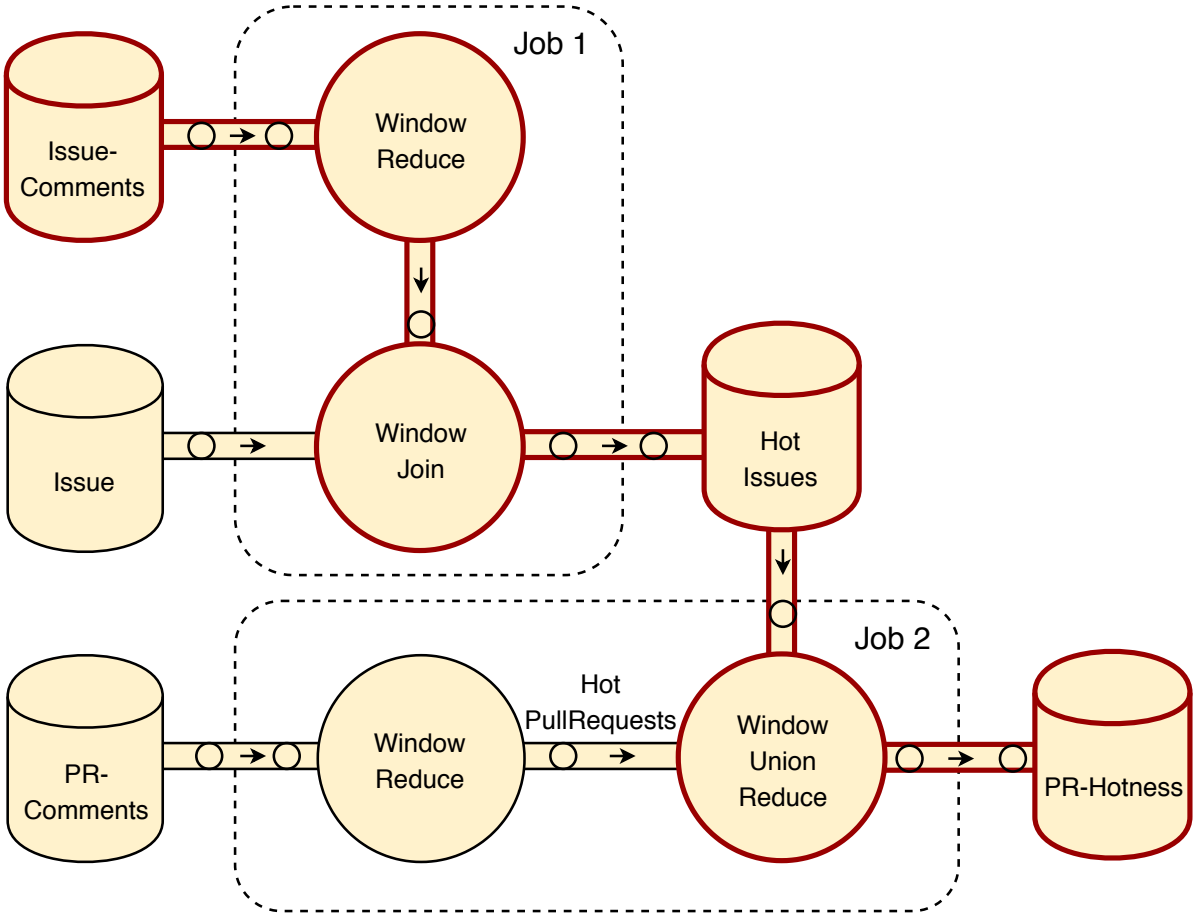


Figure 6.2: Schematic representation of the Hotness query for pull-request with an intermediate Kafka topic. Latency is measured across the red path at every operator (circle).

We use a custom window trigger, which causes windows to emit the updated value for every new event, so that we can measure latency continuously. The default behaviour is to emit window results only at the end of the window, which would let the window duration contaminate the experiments. For all tumbling windows, we use a window duration of 8 seconds, so that window state size does not have a significant impact on the measurements.

To get a consistent end-to-end comparable latency for both the single and split jobs, we only measure latency across a single path in the query. We chose to measure the latency of the issue-comment events, as shown by the red path in Figures 6.1 and 6.2. This is achieved by modelling event-time as optional, and excluding all events with no event-times from the measurements. We measure both the maximum latency and the average latency that occurred during a checkpoint, at all operators highlighted in red in Figures 6.1 and 6.2.

For all sources, we use random generators that use a seed based on a random base value for each individual operator and the offset of the produced element. The random base value and offset are part of the operator state of the generator, such that generated events are consistent upon failure and recovery. We generate a fixed number of 1000 pull requests and issues in each checkpoint. For the pull-request-comments and issue-comments, we use special seeded generator functions that include the current checkpoint id, which guarantees that no comments are generated for an issue or pull request that has not yet been produced. Furthermore, comments are more likely to relate to recently produced issues or pull requests, following a Pareto distribution. The Pareto distribution makes it less likely that comments are generated for issues or pull requests that were generated in older epochs. This means that most events can find a join partner within the current window.

The experimental set-up gives us the following control variables:

- *Parallelism:* Number of parallel instances of each operator in the job graph

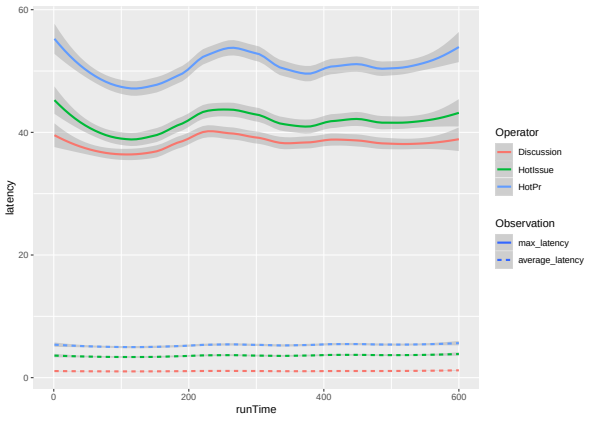
- *Throughput:* Number of events generated per parallel instance of the generator
- *Single or Double:* Single job or Kafka topic in between (Figure 6.1 or 6.2)
- *Checkpoint Interval:* Interval between checkpoints
- *Processing semantics:* At-least-once, Exactly-once or Aligned

Initially, we ran some exploration experiments without limiting the event generation, thus letting Flink's native back-pressure implementation handle the throughput. These experiments showed that our infrastructure was able to handle around 150 thousand events per parallel instance per generator. The back-pressure mechanism in Flink, however, has significant effects on the latency [32]. We therefore decided to run our remaining experiments with less than 50 thousand events per parallel instance, such that the back-pressure mechanism does not have a significant effect on our measurements.

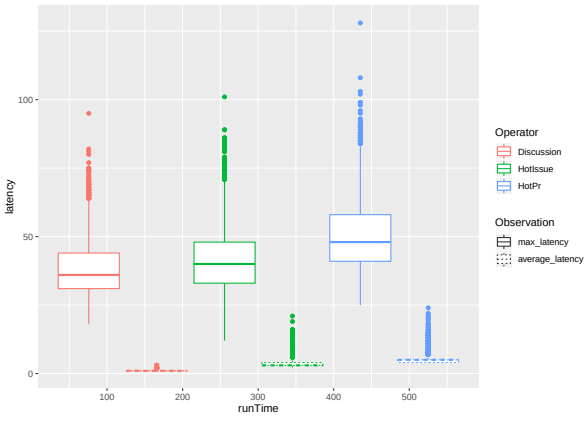
In the exploration phase, we also observed that, with low throughput, latency seems to increase. This can be explained by Flink using a buffer with a time-out between operators, and with low throughput the time-out becomes more dominant. We therefore ran our experiments with a buffer time-out of 2 milliseconds (the default is 100 milliseconds). In addition to this and the experiment's control variables, we have used Flink's default configuration.

6.2. Baseline

Our first experiments have the goal to provide a baseline for the actual experiments on aligned epochs. We therefore deployed the single job without Kafka topics from Figure 6.1 with at-least-once semantics with parallelism of 4, and 50 thousand events per second. Figure 6.3a shows both the average latency and maximum latency that occurred in every observed operator. Figure 6.3b shows a box-plot of the same data. These results show that the latency is in the order of milliseconds, and slowly increases from one operator to the next. In addition, the results show a significant difference between maximum latency and average latency. This is likely caused by the checkpointing mechanism, where the latency is higher during the checkpointing operation. *Latency increases through operators of a job, and is a few tens of milliseconds higher during the checkpointing operation.*



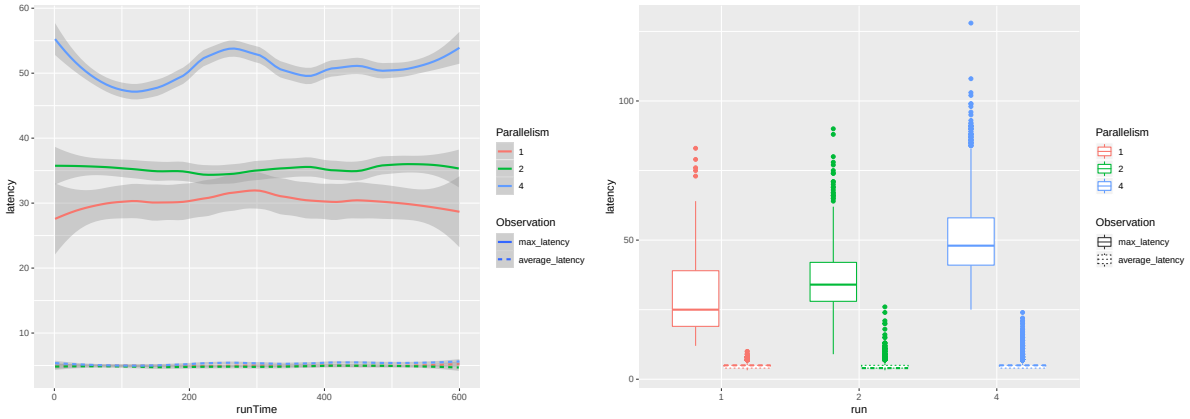
(a) Line-chart of average and max latency for every operator. The observations seem unstable, but note that the difference is a few milliseconds.



(b) box-plot of average and max latency for every operator.

Figure 6.3: Baseline measurement with the following parameters: Parallelism 4, 1 second checkpoint interval, at-least-once processing guarantee. Measuring average and maximum latency during checkpoint intervals over a period of 10 minutes for every operator highlighted in Figure 6.1.

To better understand the effects we are dealing with, we want to explore the effect of parallelism on the latency. Figure 6.4a shows the maximum and average latency on the last operator with various values for parallelism. All other control variables are kept the equal to those in Figure 6.3a. Figure 6.4b shows a box-plot of the same measurements. From these figures we can see that as the parallelism increases, so does the maximum latency. This is in line with our expectation, as we expected the majority of the latency to be due to the checkpointing mechanism, which has to perform a blocking wait when the parallelism is higher than 1. *Parallelism has a significant effect on maximum latency, but less so on the average latency.*

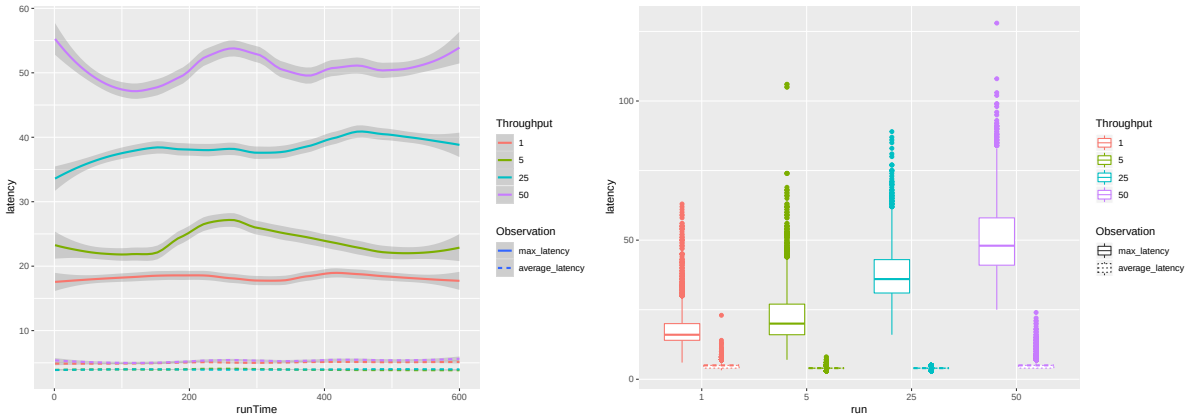


(a) Line-chart of average and max latency.

(b) Box-plot of average and max latency.

Figure 6.4: Measurement of average and maximum latency of the last operator from 6.1 with the following parameters: 1 second checkpoint interval, 50k events per parallel instance per second, at-least-once processing guarantee and variable parallelism.

Next, we want to explore the effect of throughput on the latency. We therefore keep all parameters equal again to Figure 6.3a, but run the experiment with throughputs limited to 1, 5, 25 and 50 thousand events per parallel operator per second. The results were somewhat surprising, because we expected throughput to have little effect on the latency, since we are running far below the capacity of the system. Figures 6.5a and 6.5b, however, show that throughput has a significant effect on the maximum latency, and less so on the average latency, similar to the effect of parallelism. Because the effect is also more visible on the maximum latency than on the average latency, we also expect the checkpointing mechanism to be the cause. *Throughput has a significant effect on maximum latency, but insignificant effect on average latency.*

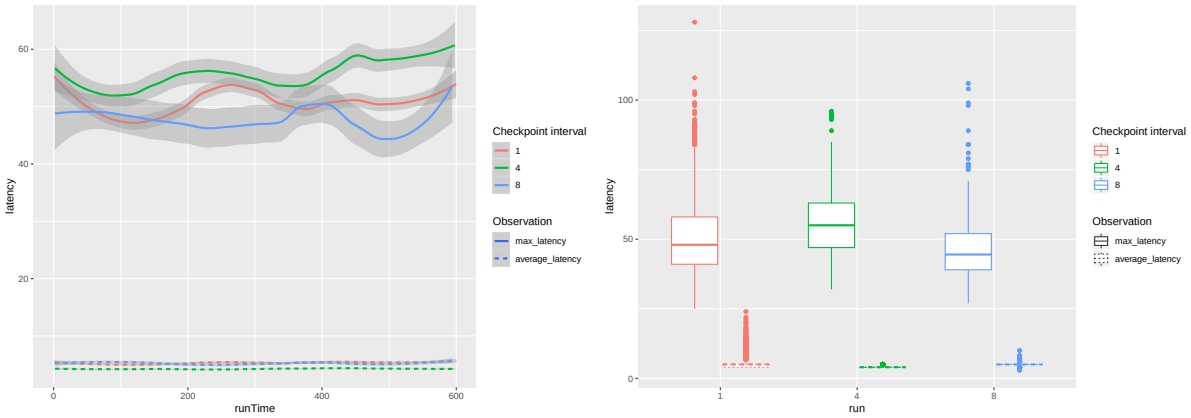


(a) Line-chart of average and max latency.

(b) Box-plot of average and max latency.

Figure 6.5: Measurement of average and maximum latency of the last operator from 6.1 with the following parameters: 1 second checkpoint interval, parallelism of 4 and at-least-once processing guarantee with variable throughput.

Finally, we perform the same experiments with variable checkpoint intervals. The results, which are shown in Figures 6.6a and 6.6b, match our expectations. There is no significant effect on maximum latency, since we measure the maximum latency for the duration of an entire checkpoint. Since earlier experiments already showed that the effect of the checkpointing mechanism on the average latency is very small, our experiments do not provide sufficient accuracy to observe a significant effect on the average latency. *The checkpoint interval has no significant effect on maximum or average latency.*



(a) Line-chart of average and maximum latency.

(b) Box-plot of average and maximum latency.

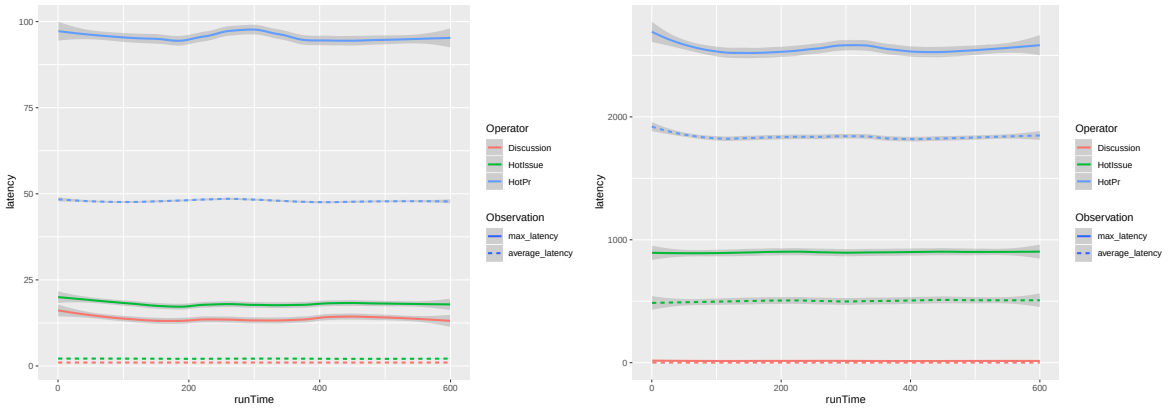
Figure 6.6: Measurement of average and maximum latency of the last operator from 6.1 with parameters: Parallelism of 4, 1 second checkpoint interval, parallelism of 4 and at-least-once processing guarantee with variable checkpoint interval.

6.3. Effect of Kafka topic between jobs

In the epoch alignment experiment we will use two connected jobs with a Kafka topic in between. The next experiments, therefore, explore the effects of introducing this Kafka topic. Here we also have to differentiate between exactly-once processing guarantee and at-least-once processing guarantee. Since exactly-once processing guarantee will force the consumer to wait until a checkpoint has completed, we expect the instance with at-least-once processing guarantee to show significantly less latency than exactly-once.

We first need to benchmark our maximum throughput again, because the (de)serialization to Kafka introduces costs which are likely higher than our sample query. We observed a maximum throughput where back-pressure becomes dominant to be around 20 thousand events per parallel operator per second, independent of the parallelism. This is relatively low, but efficient serialization for high throughput was not the focus of the project, as we are mostly interested in latency. For the further experiments a throughput of 5 thousand events per parallel operator per second is used as default configuration, avoiding back-pressure to affect the latency.

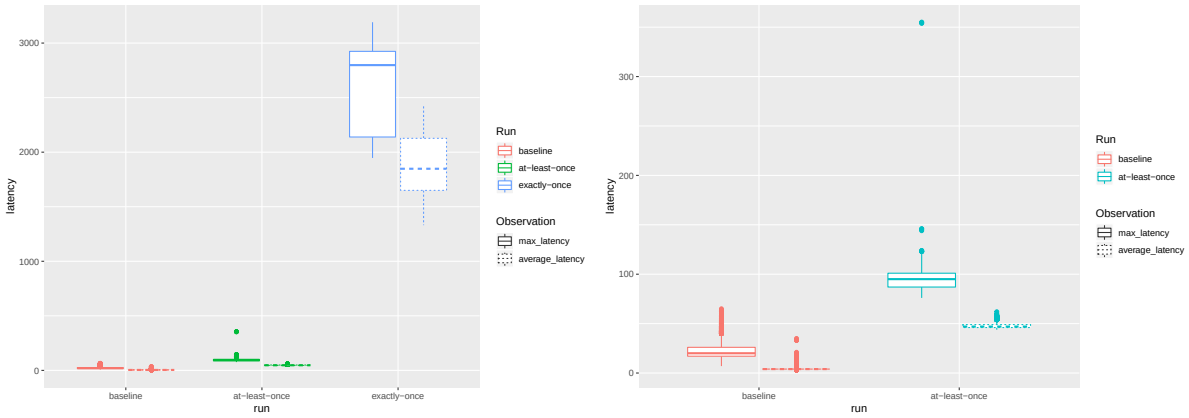
In all experiments, we keep the number of partitions on Kafka equal to the parallelism, such that each producer writes to a single private partition. We first measure the average and maximum latency per operator for *at-least-once* and *exactly-once* in Figures 6.7a and 6.7b, using default configuration. The differences between having a Kafka topic and processing semantics are shown in the box-plots in Figures 6.8a and 6.8b. *Introducing a Kafka topic has significant effect on latency. With at-least-once semantics this is in the order of milliseconds, but with exactly-once processing semantics this latency is in the order of seconds.*



(a) *At-least-once* semantics.

(b) *Exactly-once* semantics.

Figure 6.7: Measurement with Kafka topic in between using default configuration with *at-least-once* and *exactly-once* processing semantics. Measuring average and maximum latency during checkpoint intervals over a period of 10 minutes for every operator highlighted in Figure 6.2.



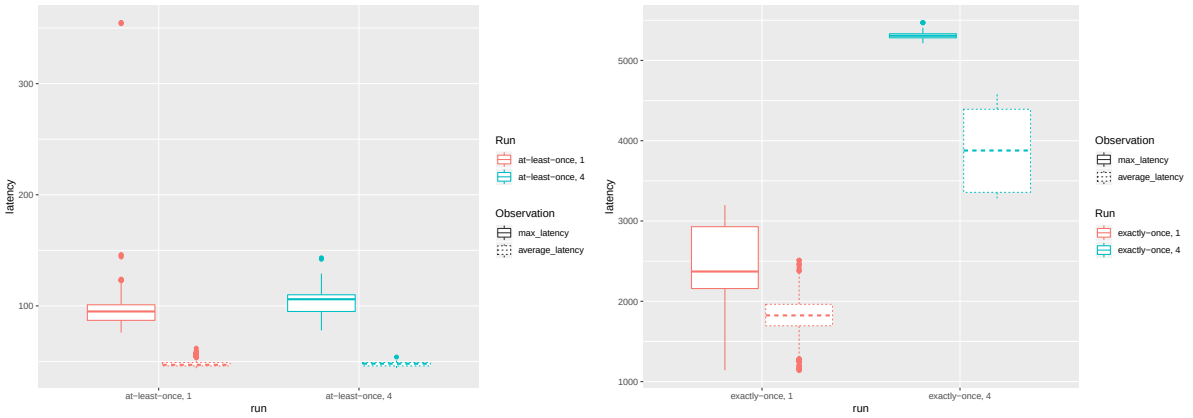
(a) Average and maximum latency of every semantic.

(b) Same plot as 6.8a without *exactly-once*.

Figure 6.8: Box-plot of the latency at the final operator, using the default configuration with variable processing semantics.

As Figures 6.7a, 6.7b, 6.8a and 6.8b show, the introduction of the Kafka topic has increased the latency significantly. The most striking observation is the large latency increase between *at-least-once* and *exactly-once* processing semantics. Where the latency for *at-least-once* remains in the order of milliseconds, *exactly-once* increases the maximum latency with multiple seconds. We expected a significant increase, because with *at-least-once* the consumer can read uncommitted records from Kafka, while with *exactly-once* the consumer has to wait for the checkpoint to be committed before it can start consuming records.

The latter means that when a record arrives just after the checkpoint, it needs to wait nearly the whole checkpoint interval, for the consumer to be allowed to process the record. This effect is shown clearly by Figures 6.9a and 6.9b, where increasing the checkpoint interval does not have a visible effect on the latency with *at-least-once* semantics, but clearly affects the latency with *exactly-once* semantics. *With exactly-once processing semantics the checkpoint interval increases the latency linearly, while it has no effect on latency with at-least-once processing semantics.*



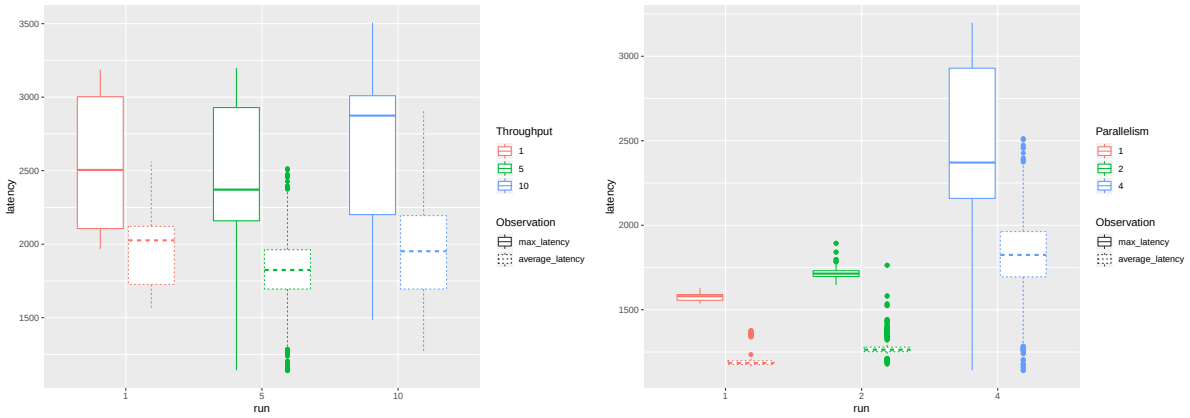
(a) *at-least-once* semantics.

(b) *exactly-once* semantics.

Figure 6.9: Measurements with a Kafka topic between jobs, default configuration and a variable checkpoint interval.

Because the latency with *exactly-once* semantics is higher than we expected (we expected slightly more than the theoretical minimum explained in section 6.4, 1 checkpoint interval), we performed more experiments. The results are shown in Figures 6.10a and 6.10b. *With exactly-once semantics and a Kafka topic in between, throughput does not have a significant effect on the latency.*

Figure 6.10b, however, shows that the parallelism has a significant effect on the latency. This is somewhat surprising, as the systems we are using are built for scalability. Not only does the latency increase significantly when the parallelism is increased to 4 (which is the maximum we can have, with the currently available resources), but also does the latency become quite unstable. The latter makes observing the effects of parallelism on the epoch alignment implementation more difficult. This effect is worth investigating further, but is left for future work. *Parallelism has a significant effect on latency with exactly-once semantics, and also affects consistency of latency.*



(a) Effects of throughput.

(b) Effects of parallelism.

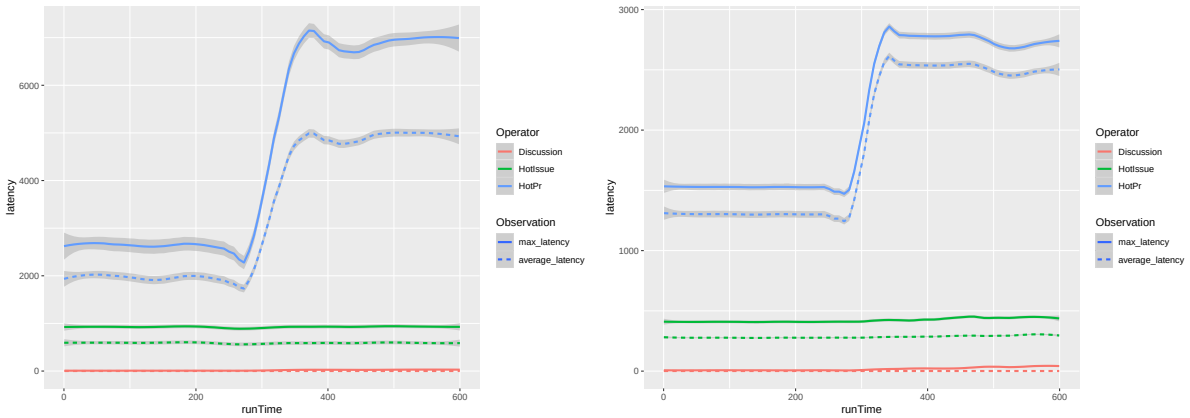
Figure 6.10: Measurement with default configuration, *exactly-once* semantics with a Kafka topic between jobs. Showing effects of parallelism and throughput.

6.4. Alignment

Now we have obtained a large number of baseline experiments and explored the effects of various parameters on the latency, we can start performing measurements on the epoch alignment protocol. We do this by starting in exactly-once processing mode, and then transition into aligned mode. We perform the run with the same default parameters used in the *at-least-once* and *exactly-once* experiments. Figure 6.11a shows the increase in latency as the job is transitioned from *exactly-once* semantics into aligned mode (with also *exactly-once* semantics). With parallelism 4 we observe a significant increase in latency (more than the checkpoint interval), and also a larger increase between average and maximum latency.

The difference between average and maximum latency is larger than the checkpoint interval, which seems strange if checkpoints are aligned. Using the ZooKeeper data we validated that the checkpoints were indeed aligned. It appears that the large difference is caused by the time it takes for the checkpoint markers to pass through all parallel instances of the Kafka connectors. Since we already observed unstable latencies in the base case with exactly-once processing semantics, we cannot conclude whether this is an issue in our implementation of the epoch alignment protocol, or (configuration of) our Kafka sink. Finding what causes the unstable latencies in *exactly-once* semantics in the base case will be high priority for our future research.

The same experiment was performed with a parallelism of 1, for which the results are shown in Figure 6.11b. It shows an increase in latency, and the difference between maximum and average latency remains equal, as we expected. *When switching from unaligned to aligned with parallelism 1, the latency increases and the difference between maximum and average latency remains equal. With higher parallelism the difference between average and maximum latency increases significantly.*

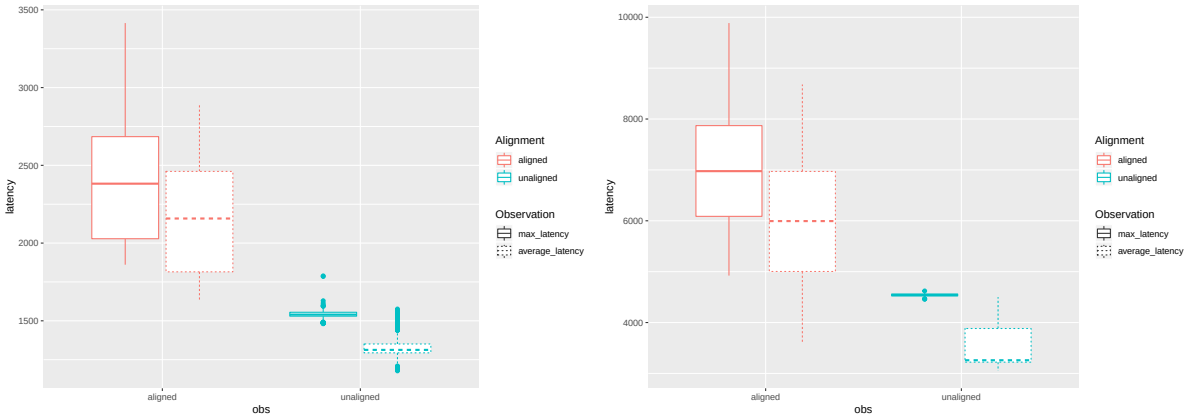


(a) Parallelism: 4

(b) Parallelism: 1

Figure 6.11: Graph showing increase in latency as transitioning from *exactly-once* processing semantics to aligned processing semantics.

While running these experiments we noticed that the observed latency is consistent within a single deployment, but the difference in latency between aligned and unaligned was not consistent across multiple runs. We explored this effect further by measuring the latency in 80 independent runs for both checkpoint intervals 1 and 4. The results are shown in Figures 6.12a and 6.12b respectively. These figures show that the increase in latency by switching from unaligned to aligned does not go further than a little over one checkpoint interval, and averages at a little over half a checkpoint interval. *The increase in latency when switching to aligned running mode is variable, and upper bound by a little over 1 checkpoint interval.*



(a) Checkpoint interval: 1

(b) Checkpoint interval: 4

Figure 6.12: Box-plot showing the latency before an after alignment from 80 independent runs.

Further investigation of this effect leads to the conclusion that the observed differences, when switching between unaligned and aligned mode, are caused by our implementation. In our implementation, the individual consumers of a job communicate an epoch to align with, during the transition from one epoch to another. When choosing an epoch to align with, the consumers only consider completed epochs. This means that in our implementation an epoch of an upstream job must be completed before the next job considers it to align with. The concept is visualized in Figure 6.13.

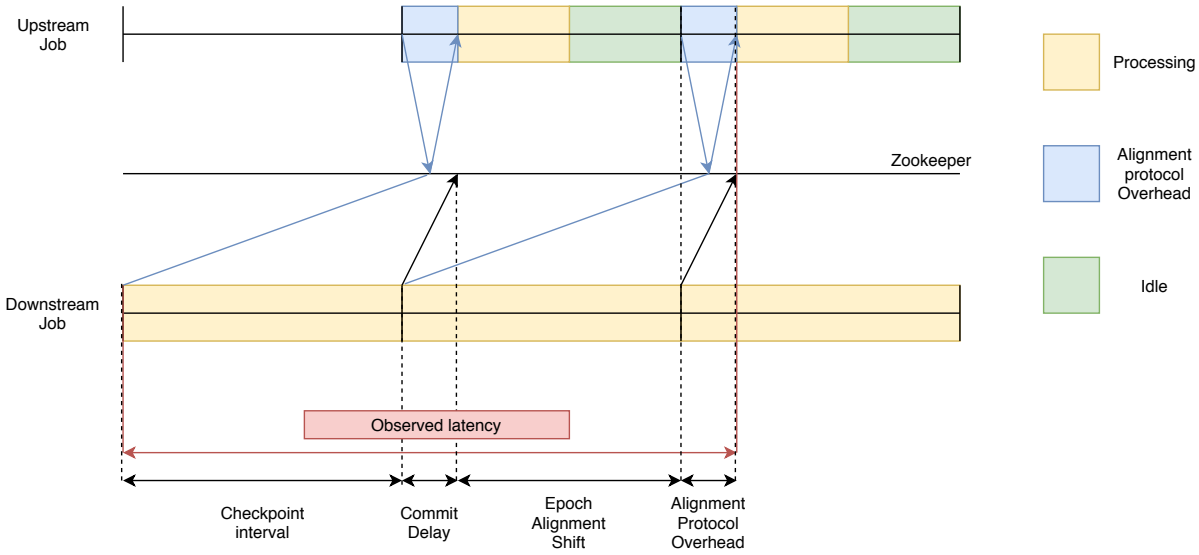


Figure 6.13: Components that determine the latency in two aligned jobs, the worst case scenario.

Figure 6.13 shows that the observed latency has 4 main components. We ignore the latency caused by the processing time of the upstream job, as Figure 6.3a shows this latency is consistent and independent from our epoch alignment protocol. The first component of latency is the checkpoint interval. Since *exactly-once* processing semantics force us to only read committed events, this latency is unavoidable. The second part

is the delay it takes to commit the epoch. For the same reason as the previous component, this latency is unavoidable when using *exactly-once* processing semantics.

The next component of the latency is the component we call *epoch alignment shift*. This component is caused by our implementation, where we only look for newly available epochs at the start of a next checkpoint. This means that once the upstream job has committed its checkpoint, we still need to wait for the next checkpoint of the downstream job, before the data can be consumed. Figure 6.13 shows the scenario where both jobs are checkpointing at the exact same moment. This is not necessarily the case, as both jobs run independently. Figure 6.14 shows the same process, but with shifted checkpointing moments for both jobs. As Figure 6.14 shows, the alignment of checkpoint moments of the individual jobs has impact on the latency.

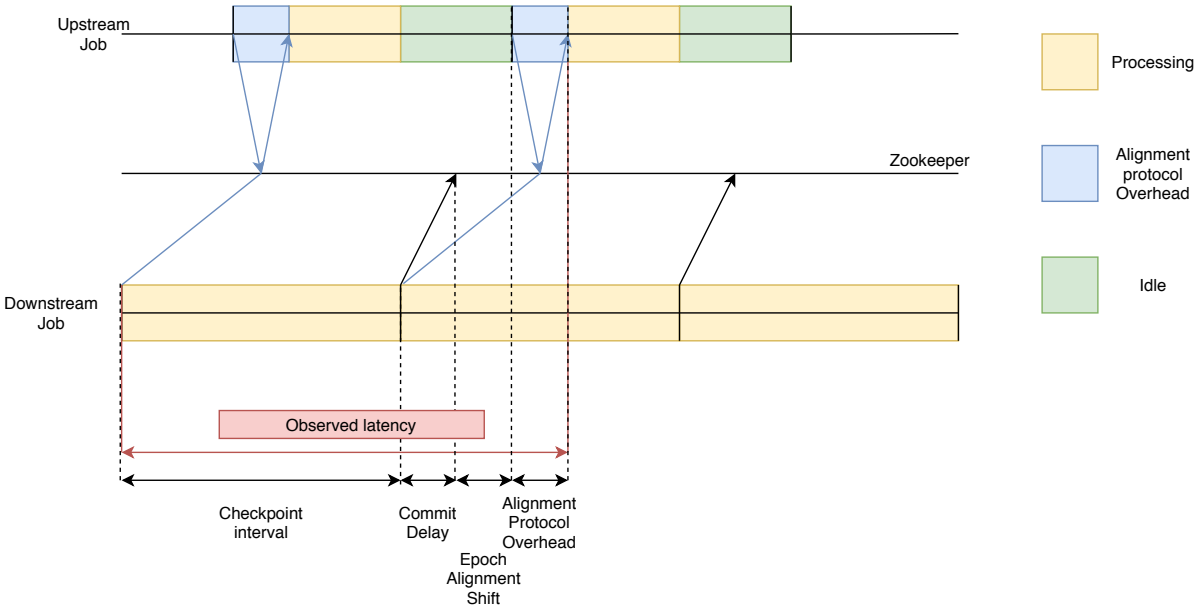


Figure 6.14: Components that determine the latency in two aligned jobs, a more optimal scenario.

We can conclude that the additional latency of aligned jobs, compared to jobs without any processing guarantees, is determined by the sum of the checkpoint interval, commit delay, epoch alignment shift and the alignment protocol overhead. As both overheads are minimal, the most important factors are the checkpoint interval and the epoch alignment shift. The epoch alignment shift is upper bound by the checkpoint interval (if the shift is more than 1 checkpoint interval, the upstream job will align with the next epoch).

When a job is running with *exactly-once* processing semantics, the checkpoint interval and commit delay are unavoidable. This means that compared to *exactly-once* processing semantics, the only additional latency costs are the epoch alignment shift and the overhead of the protocol.

6.4.1. Optimization of epoch alignment shift

In our experiments we relied on the static checkpoint interval provided by Apache Flink. If we would let our downstream checkpoint moments be determined by the upstream job, the epoch alignment shift can theoretically be reduced to 0. However, if there are multiple upstream operators, one cannot perfectly align the checkpoint intervals with all upstream jobs. The upstream jobs cannot simply align their checkpoints with each other, because the processing latency of the jobs might be different. In the case of multiple upstream jobs, one can attempt to align the checkpoint moment with the optimal upstream job, such that the epoch alignment shift is minimal. In the worst case scenario, where the upstream checkpoint intervals are distributed uniformly, the theoretical upper bound on the epoch alignment shift is equal to the *checkpoint interval* - (*checkpoint interval* / *number of upstream jobs*).

6.5. Summary

The experiments have shown that the latency of a streaming job without Kafka topic in between, is in the order of a few milliseconds. Adding a Kafka topic in between increases this latency significantly. When switching

to *exactly-once* processing semantics, the latency increases much more, depending on the configured checkpoint interval, such that the checkpoint interval is the major component of the latency. Increasing parallelism produces unstable results when running with *exactly-once* semantics, which requires further investigation.

Switching to aligned mode increases the latency by a variable amount, with an upper bound dependent on the configured checkpoint interval. From the perspective of *exactly-once* processing semantics, this increase in latency is acceptable. The implementation of the alignment can also be further optimized, such that the increase in latency consistently ends up closer the lower-bound.

7

Conclusion

This thesis proposes the concept of epoch alignment, where checkpoints of streaming jobs flow through the boundaries of a job into consecutive jobs. We propose a protocol for achieving epoch alignment on jobs forming a DAG, by extending Flink's checkpointing mechanism, and staying within the boundaries of the Chandy-Lamport protocol. The concept does not compromise the independence of individual streaming jobs, so that the failure-recovery mechanisms of individual jobs remain independent.

Through proof of concept we have shown the possibility to implement the protocol by modifying the connectors of the Apache Flink stream processing framework. We have shown the capability to transition two sequential jobs from unaligned exactly-once mode into aligned mode and back with minimal effect on latency. Through experiments we have shown that the latency costs of the alignment in our proof of concept is bounded by the duration of a single checkpoint interval, and we provide an explanation how this could be further improved for most use cases.

Finally, we provide applications and limitations of the proposed protocol. The most significant limitation, which is subject to future research, is the requirement of deterministic streams, which our proof of concept cannot provide for iterative computations or async I/O. As one of the applications, we explain how two aligned jobs can be hot-swapped while maintaining exactly-once processing guarantee. Furthermore, we have shown how aligned epochs can be used to make state queries idempotent. Finally, we suggest how the hot-swap capability provides the means to perform optimizations on running queries, paving the path towards multiple query optimization on streaming jobs.

8

Future Work

This chapter covers the future work we envision as continuation of this thesis. Section 8.1 starts with the applications of epoch alignment by using the proof of concept produced for this thesis to solve real-world challenges. Next, in Section 8.2 we cover the limitations of the epoch alignment concept, and suggest potential solutions that can be further investigated.

8.1. Applications

This thesis proposes the concept of epoch alignment. However, it does not yet use epoch alignment to solve real-world challenges. Section 4.2 explains how epoch alignment can be used to perform a hot-swap of streaming jobs. Using this explanation and the epoch alignment implementation from this thesis, a proof of concept for hot-swapping streaming jobs is straight forward to implement. A direct continuation of this thesis should include implementing this hot-swapping method, and explore the gains of hot-swapping streaming jobs.

Chapter 2.3 explained the application of epoch alignment for queryable state. This application requires the state back-end to support queries on a historical state of a specific epoch. Further research is needed to find out how these historical state lookups can be supported in an efficient way. Furthermore, the second example given in 2.3.2 requires a state lookup to wait for an epoch to complete. This can be easily solved by performing the state lookup as async I/O request, but as explained in 8.2.3 this has other drawbacks. Future work might find a more tailored solution for the state-queries from one job to another, avoiding those drawbacks.

Finally, after a proof of concept for hot-swapping streaming jobs has been developed, this feature could be exploited to perform real-time multiple query optimization as explained in 2.3.3. Multiple query optimization has been a research topic for decades [36, 41, 44], but the research on multiple query optimization applied to streaming jobs has barely scratched the surface. Future work will have to determine whether epoch-alignment can be used to build a management system which continuously optimizes its execution plans by hot-swapping jobs.

8.2. Improvements

This section describes future work that could further improve epoch alignment, both in concept and implementation.

8.2.1. Parallelism

Our experiments showed inconsistent results when the parallelism is increased. This is not necessary a problem in the implementation of the epoch alignment protocol, as these inconsistencies already showed up in the base case when just running with *exactly-once* processing semantics. This observation should be further investigated, as latency should not increase in the order of seconds just by adding a few parallel instances. The first direction to look would be the potentially inefficient usage of ZooKeeper reads and writes, that increase unnecessary with the number of parallel instances.

8.2.2. Integration of publish-subscribe system

When checkpoint markers are flowing from one job to another, they flow through the publish-subscribe system in between. This results in a very tight coupling between the stream processing engine and the publish and subscribe system. Section 5.5.1 explained that separate partitions are required for each parallel producer to guarantee the epochs also exist on the publish-subscribe system with an exact offset. Future work could propose a protocol to let the publish-subscribe system handle epoch markers in a very similar way to operators in the checkpointing protocol.

Such a protocol would simplify the responsibilities of the checkpoint coordinator, because alignment of different markers from different jobs pushing to the same subject can already be merged by the publish-subscribe system. This would also allow round-robin routing of events to different partitions, avoiding the unbalanced partition issue.

A completely different approach would to be to implement epoch alignment on the publish-subscribe system instead of the stream processing system. This would simplify the overall architecture, as publish-subscribe systems already include a mechanism to manage offsets on distributed partitions. Future work could explore the feasibility of implementing epoch alignment on a publish-subscribe system, independent from the stream processing system.

8.2.3. Cyclic jobs and async I/O

In section 4 we mentioned that even with aligned epochs we cannot provide exactly-once or even at-least-once guarantee when performing a hot-swap of jobs containing cycles or async I/O operators (Note that here we mean cycles within a single job). The reason is that in such jobs the checkpoint markers can pass events, and thus the checkpoint markers provide no guarantee of an event actually being processed by the job. This can be avoided by introducing some constraints on how cycles or the async I/O operator processes events and markers.

In the case of async I/O, a solution would be to use a fixed "output" epoch that depends on the epoch when the event was received. For example, the response of an async I/O operator will be processed 5 epochs after the event was received. This means the operator will always process the response in the assigned output epoch, and if no response is available yet, will use a "timeout" as response instead. Because an event received in epoch X will always be processed in epoch X+5, this would allow hot-swapping two jobs without potentially missing events.

In terms of cycles a solution would be to let each cycle perform a fixed amount of loops within an epoch. The implementation would need to enforce that all "new" events entering the cycle within an epoch are also looped around an equal amount. A naive way would be to buffer all new events, and only loop "old" events, but there likely are smarter solutions.

Not only would these solutions allow hot-swapping of jobs with cycles and/or asynchronous operators, but in the case of asynchronous operators the epochs would also provide guarantee of events actually being processed. It would be very interesting to research the drawbacks (in terms of latency and snapshot-size) of imposing these restrictions on cycles and asynchronous operators in real-world use cases.

8.2.4. Cyclic dependencies and hybrid alignment state

One of the limitations of the current alignment protocol is that it does not support cyclic dependencies across jobs (for example, job A reads output from job B and job B reads output from job A). The protocol does allow aligning a newer epoch of job A with an epoch from job B, which aligns with some older epoch from job A. However, it does not support bootstrapping the system such that there is an older epoch of job A to align with. Research is needed to determine what extensions can be made to the protocol to bootstrap the system in the case of cyclic dependencies across jobs.

8.2.5. Variable checkpoint interval

The concept only works when all jobs use an equal checkpoint interval. This would not be necessary if we would use some hybrid alignment state, where some of the epochs are aligned but others are just local checkpoints which can be used as recovery in case of a failure. Further research is required to determine what infrastructure and protocol could support such a hybrid alignment state.

Bibliography

[1] Apache flink. https://flink.apache.org/. Accessed: 2017-10-17.

[2] Apache hadoop. https://hadoop.apache.org/. Accessed: 2019-01-12.

[3] Apache kafka. https://kafka.apache.org/, . Accessed: 2018-11-14.

[4] Eos abort index proposal. https://docs.google.com/document/d/1Rlqizmk7QCDe8qAnVW5e5X8rGvn6m2DCR3JR2yqwVjc/edit, . Accessed: 2019-02-01.

[5] Transactions in apache kafka. https://www.confluent.io/blog/transactions-apache-kafka/, . Accessed: 2019-02-01.

[6] Kappa architecture. http://www.kappa-architecture.com/. Accessed: 2019-01-27.

[7] Lambda architecture. http://lambda-architecture.net/. Accessed: 2019-01-27.

[8] Apache spark. https://spark.apache.org/. Accessed: 2019-01-12.

[9] Apache storm. http://storm.apache.org/. Accessed: 2019-01-27.

[10] Apache zookeeper. https://zookeeper.apache.org/. Accessed: 2018-11-28.

[11] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

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

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

[15] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal—The International Journal on Very Large Data Bases*, 23(6):939–964, 2014.

[16] Alexander Alexandrov, Andreas Salzmann, Georgi Krastev, Asterios Katsifodimos, and Volker Markl. Emma in action: Declarative dataflows for scalable data analysis. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2073–2076. ACM, 2016.

[17] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[18] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[19] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[20] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[21] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[22] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[23] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

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

[25] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. Rsp-ql semantics: a unifying query model to explain heterogeneity of rdf stream processing systems. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(4):17–44, 2014.

[26] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2 (2):1414–1425, 2009.

[27] Georgios Gousios, Dominik Safaric, and Joost Visser. Streaming software analytics. In *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*, pages 8–11. ACM, 2016.

[28] Katarina Grolinger, Michael Hayes, Wilson A Higashino, Alexandra L'Heureux, David S Allison, and Miriam AM Capretz. Challenges for mapreduce in big data. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 182–189. IEEE, 2014.

[29] William Gropp, Steven Huss-Lederman, and Marc Snir. *MPI: the complete reference. The MPI-2 extensions*, volume 2. Mit Press, 1998.

[30] John Hunt. *Scala design patterns: Patterns for practical reuse and design*. Springer Science & Business Media, 2013.

[31] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

[32] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream processing engines. *arXiv preprint arXiv:1802.08496*, 2018.

[33] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[34] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[35] Sara Landset, Taghi M Khoshgoftaar, Aaron N Richter, and Tawfiq Hasanin. A survey of open source tools for machine learning with big data in the hadoop ecosystem. *Journal of Big Data*, 2(1):24, 2015.

[36] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 666–677. IEEE, 2012.

[37] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM, 2005.

[38] Erik Meijer. Reactive extensions (rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM, 2010.

[39] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

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

[41] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.

[42] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on apache spark. *International Journal of Data Science and Analytics*, 1(3-4):145–164, 2016.

[43] Sriram Sankaran, Jeffrey M Squyres, Brian Barrett, Vishal Sahay, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *The International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[44] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1): 23–52, 1988.

[45] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[46] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 526–531. IEEE, 1996.

[47] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[48] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[49] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[50] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.