State Persistence in Noir

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Author: **Emmanuele Lattuada**

Student ID: 990034
Advisor: Prof. Alessandro Margara
Co-advisors: Prof. Gianpaolo Cugola, Luca De Martini
Academic Year: 2022-23

Abstract

Noir is a new distributed system designed to process large amounts of data, it is scalable, efficient, and fast. With its functionalities, it is possible to easily implement different kinds of analyses and computations in just a few lines of code, while delivering excellent performance in terms of execution times.

This thesis focuses on the study and implementation of state persistence in Noir, aiming to make the system fault-tolerant. In the context of applications like Noir, involving long-term executions on large datasets or streaming data, fault tolerance becomes a crucial factor. By periodically saving the state of the system, the algorithm ensures that in the event of a failure, such as a network error, the computation can resume from the last saved state, avoiding the need to start from scratch.

The algorithm for state persistence has been designed to maintain Noir's original architecture and design while allowing the periodic saving of a system snapshot to a reliable external source. This enables recovery from the last saved state in case of a failure.

The algorithm has been tested, and the thesis includes a study and analysis of the experiments conducted to understand the effects of the algorithm on Noir's performance in various situations and with different parameters.

Keywords: distributed systems, stream processing, state management, fault-tolerance

Abstract in lingua italiana

Noir è un recente sistema distribuito progettato per processare grandi quantità di dati in maniera scalabile, efficiente e veloce. Le funzionalità che offre permettono di definire in modo semplice e in poche righe di codice diversi tipi di analisi e computazioni e, allo stesso tempo, di ottenere ottime performance sui tempi di esecuzione.

Il lavoro di questa tesi si concentra sullo studio e sull'implementazione della persistenza dello stato in Noir, l'obiettivo è di rendere il sistema tollerante ai guasti. Il contesto di applicazione di un sistema come Noir prevede esecuzioni molto lunghe su moli di dati considerevoli, per questo motivo la tolleranza ai guasti diventa un fattore chiave. Salvando periodicamente lo stato del sistema, nel caso in cui si verifichi un guasto come un errore di rete, diventa possibile riprendere la computazione dall'ultimo stato salvato, evitando così di dover ripartire da zero.

È stato progettato un algoritmo di persistenza dello stato che mantiene inalterato il design e l'architettura di Noir e che allo stesso tempo permette di salvare periodicamente uno snapshot del sistema su una sorgente esterna affidabile in modo che in caso di guasto sia possibile recuperare l'ultimo stato salvato e ripartire da quello.

Questo algoritmo è stato testato e nella tesi è presente uno studio ed un'analisi degli esperimenti effettuati per cercare di comprendere gli effetti dell'algoritmo sulle performance di Noir in diverse situazioni e al variare di diversi parametri.

Parole chiave: sistemi distribuiti, stream processing, gestione dello stato, tolleranza ai guasti

Contents

Introduction

The amount of data and information being generated is increasing rapidly every day. Various scenarios, such as social media, the Internet of Things, commercial transactions, and advertising, are constantly generating vast amounts of data that need to be processed and analyzed efficiently. To address this need, faster and more efficient software is required to process and extract insights and models from this data.

A very effective paradigm that has developed and spread in recent years is dataflow. This approach focuses on data that are modeled as a flowing stream. Operations applied to the data are functions and transformations that take an input data stream and produce an output stream containing processed data. The collection of all functions applied to the stream forms a directed graph of operators, where each node represents an operator that performs a specific function or transformation. The edges in the graph represent the communication channels between these operators. It is a directed graph because the data flows in one direction, from the initial nodes of the graph, passing through various operators for processing and manipulation, and reaching the final nodes of the graph where the computation results are produced.

Compared to the traditional imperative programming model, the dataflow paradigm does not have a single shared state or an explicit flow of control. Once the network of operators and communication channels are defined and created, each operator works independently with its own local state and control flow. This independence allows for easy and efficient computation distribution across a cluster of hosts, as communication and synchronization are managed automatically by the communication channels. Furthermore, the dataflow model is highly scalable. You can handle higher workloads simply by adding more hosts to the cluster and instantiating more operators without changing the system architecture.

Noir is a streaming and batch processing framework based on dataflow and developed by a research team at Politecnico di Milano. It provides a set of APIs for conducting various types of analyses and offers excellent performance when compared to other dataflow frameworks.

This thesis work focuses on state persistence in Noir. The state in Noir consists of the collection of internal and local states of each individual operator. To extract this state during execution, a snapshot algorithm is required. A snapshot is a picture of the system, it captures the states of the operators and the messages being exchanged through communication channels.

The main purpose of state persistence is to make the system fault-tolerant. Reliable periodic snapshots of the global state enable the system to restart from the last saved state in case of failure. In the context of large volumes of data and resource-intensive executions, as in the case of real-time stream processing, having fault-tolerance mechanisms to avoid restarting processing from scratch in the event of an operator or connection failure is essential. Persistence also enables the implementation of other features, such as the ability to provide a queryable state and the capability to stop and resume execution from an intermediate point.

The first chapter of this thesis introduces Noir, describing its architecture and key functionalities. The second chapter explains what a snapshot algorithm is and provides an overview of some snapshot algorithms, comparing the solutions implemented in other frameworks. The third chapter details the snapshot algorithm we implemented in Noir, examining the states of individual operators and the procedures we put in place to obtain a consistent state. The fourth chapter presents the experiments conducted to study the impact of the snapshot algorithm on the performance of Noir.

1 | Noir

Noir[18] is a processing system for batch and streaming analysis written in Rust[7] and based on the dataflow[15] paradigm. Its goal is to provide a high level of abstraction and expressiveness but also to reach performances close to custom and ad-hoc implementation written with tools and libraries at lower abstraction levels, like MPI[21] (Message Passing Interface). Its design and some of its API are inspired by Apache Flink[20], a popular open-source framework for stream processing also based on dataflow. Apache Flink has also been the reference for evaluating Noir's performances together with the custom implementation of the same benchmarks written in MPI. Noir showed that it can achieve a throughput up to x30 compared to Apache Flink and for some tasks, it has performances close to custom MPI implementation.

Noir can handle bounded and unbounded streams of any type, the only required constraints are defined by the Data trait and ExchangeData trait depending on the specific type of operator. This means that the user can create a stream of desired type but he has to provide the implementation of the functions requested by these traits. Noir can also manage keyed-streams that are streams of Key-Value tuples, this kind of stream is composed of logically independent substreams, one for each key. In other words, if two tuples have the same key they belong to the same substream, otherwise they are processed separately as they belong to different substreams. To go into detail view the section Partitioning of Operators. Another interesting stream is the one composed of tuples Value-Timestamp, where each value is associated with a time. This can be used for event processing as explained in the section Timestamp of Operators.

As in the dataflow model, Noir generates a network of operators with a directed graph. The first nodes in the network are called Sources, last ones are called Sinks. Stream tuples flow from one operator to the next one through communication channels. The type StreamElement incorporates both tuple and metadata necessary for the correct execution of the flow. In particular, StreamElement has these instances:

- Item(Out): tuple of the stream. Out is the generic type;
- Timestamped(Out, Timestamp): tuple of type Value-Timestamp;

- Watermark(Timestamp): special timestamp, view section Timestamps;
- FlushBatch: used to mark the end of a batch of data;
- FlushAndRestart: used to finish an iteration, view section Iterations;
- Terminate: used to notify the end of the stream and the end of the computation, this is the last token received from each operator.

1.1. Architecture

This section explains the abstract model and the main concepts behind this framework.

1.1.1. Job graph and execution graph

The job graph is a directed graph that represents the operators and the relations between them. Each node is an operator and each edge is a communication channel through which data are exchanged from one operator to the next one.

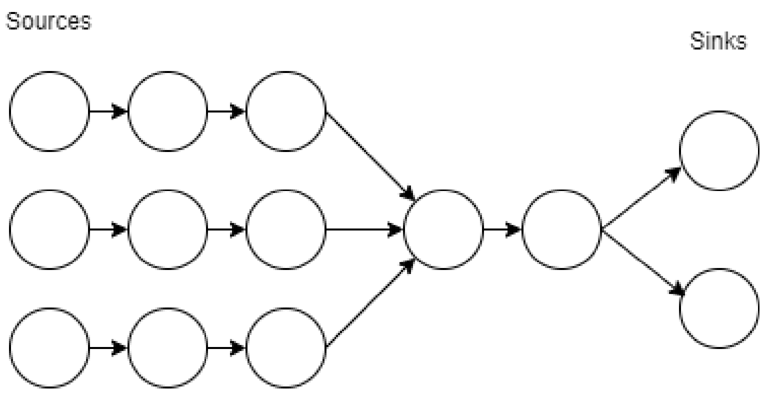

Figure 1.1: Simple job graph.

Sources are the first operators in the graph while the last ones are the Sinks. The former has the role of feeding the network with data, so it produces the tuples and sends them to the next operators. Tuples can be generated at runtime (i.e. a sequence of numbers) or can be taken from an external source like a file or a TCP channel. The latter is in charge of handling computation results, it can collect results or do other actions specified by the user.

The job graph can present cycles if there are iterative operators, this introduces some complications as explained in later sections.

The execution graph is derived from the job graph. Most of the operators can be parallelized and replicated to improve throughput, the execution graph contains all the replicas of each operator in the job graph and all the connections between them. The number of replicas of each operator is related to the specific operator, indeed there are some operators that cannot be replicated, for example, a Source that reads data from a TCP channel. The execution graph depends on several things:

- the number of hosts and maximum parallelism limit of each of them fix the maximum number of operator replicas that each host can instantiate,
- the replication constraints of each individual operator that specify the number of replicas that each operator can have, operators with limited replication cannot be replicated and have a single instance while operators with unlimited parallelism can have an arbitrary number of replicas that is defined by the maximum available parallelism given from the previous point,
- the communication mode from an operator to the following one defined by the Next strategy (see next section) which specifies the connection topology.

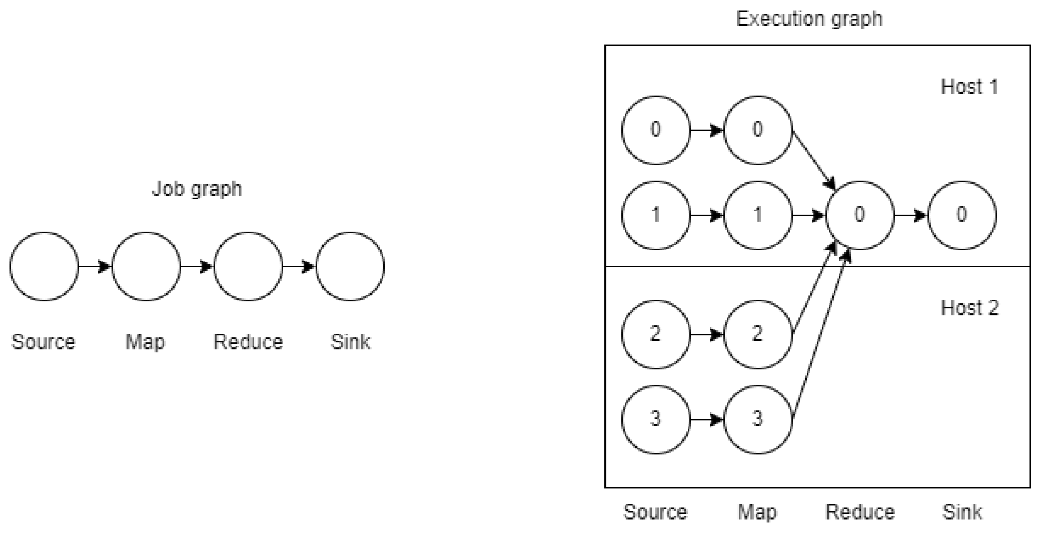

Figure 1.2: Job graph and execution graph

In figure 1.2 we can see a simple job graph on the left and a possible corresponding execution graph on the right. The cluster is composed of 2 hosts with maximum parallelism equal to 2 for both. The numbers associated with each operator on the right are the replica indexes. For Source and Map operators there is unlimited parallelism while Reduce and Sink cannot be replicated. More details on these operators are provided in the section Operators.

1.1.2. Next strategy and blocks

The Next strategy specifies the connection topology between an operator p and the next operator n in the execution graph. There are 4 possible strategies:

- random: the data is sent to a replica of n randomly chosen,
- group by: the replica to which send data is deterministically chosen depending on the data,
- only one: p and s have the same number of replicas and a replica of p sends all data to only its corresponding replica of n, this means that each replica of p has just one output channel and each replica of n has just one input channel to have a 1 to 1 connection,
- broadcast: each replica of p sends data to all replicas of n.

To optimize the execution, Noir groups operators into blocks. The grouping is based on the Next strategy, indeed inside a block, there are only operators with Next strategy "only one" and that don't need the network layer. Each block starts with a special operator called Start or with the Source operator and ends with another special operator called End or the Sink operator. Operators inside a block are serially connected, one after the other, and they make a so-called Operator Chain.

1.1.3. Communication channels

The communication channels between operators can be local or remote. The local ones are in-memory channels that are used between operators on the same host. Remote channels are constructed over TCP channels and they are used for communication between operators on different hosts. Using local channels simplifies communication by avoiding message serialization and deserialization and the overhead due to the network stack. Remote channels are optimized and it is used the same TCP socket for multiple remote channels. In particular, just one TCP socket is used to connect two blocks on different hosts regardless of the number of instances (replicas) of the two blocks. In-memory channels are based on the paradigm of multiple-producers and single-consumer and are provided by crate Flume[3] (default) or crate Crossbeam[2] (available as a feature).

1.2. Scheduling and execution

Noir keeps all necessary information for the execution in a structure called Environment. The creation of an Environment is the first thing to do to start a Noir program, it needs a configuration as a parameter. The configuration specifies the execution mode, which can be local or remote. Local execution does not need a cluster, all the code is executed locally. This mode is mainly used for tests and debugging. Remote execution requires a list with all hosts belonging to the cluster, corresponding ports to be used for connections, and credentials to establish an SSH connection. It is also necessary to specify for each host the maximum allowed parallelism, i.e. the maximum number of replicas that can be instantiated for each operator. This parameter is also required in local execution mode and is typically set to the host number of cores.

Once the Environment has been created, the next step is the cluster spawning with the function "spawn_remote_workers()". For each host listed in the configuration, an SSH connection is established through which the executable and the configuration file can be sent, after that the target host can launch the program.

The Environment structure also allows defining the Stream, it starts with the creation of the Source and then gradually adds operators following the order of the flow, thus the job graph is built. When you add an operator, the division into blocks is automatically carried out and the various operators are scheduled. The stream must always end with a Sink.

After the entire Stream is defined we can call the function "execute_blocking()", this function actually starts the data processing. First, the execution graph is generated, then all connections are created on the base of the execution graph, and finally, through the scheduler, all the blocks are instantiated and so also all the operators contained in each block. Each block is handled by a dedicated worker and therefore by a dedicated thread. The worker manages the block processing through the functions "setup()" and "next()" provided by the operator chain.

```rust
// Get the configuration from command line arguments
let (config, _args) = EnvironmentConfig::from_args();
// Create a new environment
let mut env = StreamEnvironment::new(config);
// Spawn workers on remote hosts
env.spawn_remote_workers();
// Create a Source operator
let source = IteratorSource::new(0..100);
// Define the stream
let result = env
    .stream(source)
    .group_by(|i| i % 5)
    .fold(Vec::new(), Vec::push)
    .collect_vec();
// Start the processing
env.execute_blocking();
// Read and print result
if let Some(result) = result.get() {
    println!("{result:?}");
}
```

Listing 1.1: Simple Noir program

1.3. Operators

Operators are a fundamental part of Noir because they contain all the logic needed for data processing. All operators implement the trait Operator which defines the basic functions that each operator must provide, in particular, the two most important functions are:

- **setup()**: function to initialize the operator, all that metadata generated after the whole flow has been instantiated is passed to this function, this information includes block coordinates, block replicas, and network topology.
- **next()**: function used to process the Stream, it is modeled as an iterator. This means that this function always returns a StreamElement but all the logic and the code necessary to get it depends on the operator and the specific implementation of this function. Typically this function retrieves one or more StreamElement from

the previous operator, does some processing, and returns another StreamElement.

As mentioned earlier, the operators within a block form an Operator Chain where each operator keeps track of the previous one. This model implies that each block has direct access only to the last operator of the chain (an End or Sink operator). When setup() and next() functions are called on an Operator Chain, these calls are done on the last operator of the chain and then are recursively propagated to the previous operators up to the first operator in the chain (a Start or a Source operator). In this way the program control passes from the last to the first operator in the chain, this means that in the Operator Chain, a StreamElement is not sent from one operator to the next one but is requested from an operator to the previous one. The flow of next() calls is in the opposite direction from the StreamElement flow through the Operator Chain. This is a pull-based architecture.

The main operators available in Noir are described below, they are grouped according to the kind of operations they perform.

1.3.1. Sources

The role of these operators is to inject tuples into the flow. Sources are the first nodes in the job graph and they don't have in-going edges but only out-going ones. Replication constraints of this type of operator depend only on the specific Source. There are different Source operators depending on the actual external source of the data:

- **Channel**: reads data from a channel, the channel can be an in-memory channel or a network channel (TCP channel). Due to the single instance of the channel this Source cannot be replicated.

- **File**: reads data from a file. The file is read line by line. This Source can be replicated as long as the file is in the same location on all hosts. The file is split into chunks and each replica reads only from its own chunk.

- **Csv**: reads data from a CSV file, similar to FileSource but specific to this format, indeed it provides several configuration parameters to read and extract tuples from the CSV format. Similar to FileSource, this Source can be replicated if the file is present in the same location on all hosts.

- **Iterator**: generates data from an iterator, this type of Source is very flexible and useful because, through the abstraction of the iterator[9], it is possible to generate all kinds of data sequences. This Source cannot be replicated because the iterator cannot be parallelized.

- **Parallel Iterator**: similar to IteratorSource. This Source requires a generation function as a parameter, the function is used to generate the iterators for each replica of these operators. The generation function uses the replica ID and the overall number of replicas to produce the right iterator for each instance.

1.3.2. Sinks

These operators are the nodes at the end of the job graph and they don't have outgoing edges. They receive and manage processing results. Like Sources, the replication constraints depend on the specific type of Sink. There are two main types of Sinks:

- **ForEach**: is used to perform operations on each element that arrives at the operator. In other words, it is possible to specify a procedure to be applied to each result that arrives at the ForEach operator. This Sink can be replicated since the operations to be performed on each element depend only on the element itself.
- **Collect**: as the name suggests, is used to collect all processing results into a single structure that can be returned to the user. Since it is a collection operation it is required to be performed on a single instance and therefore this Sink cannot be replicated. There are several implementations of the Collect operator that differ in how results are collected. For example, the CollectVec operator collects all results in a Vec while the CollectCount operator sums all results it receives and returns to the user only the sum of all the results.

1.3.3. Data manipulations and filters

This kind of operator includes simple operators such as:

- **Map**: applies a transformation to each tuple it processes. An example of a transformation might be an arithmetic operation if the tuples are numbers or a transformation to uppercase letters if the tuples are literal characters or strings.
- **Filter**: applies a filter to each tuple. If the filter condition is satisfied then the tuple is forwarded to the next operator otherwise the tuple is removed from the stream. An example might be a filter with a threshold where tuples with a value above the threshold are discarded.
- **Flatten**: is used to decompose a tuple into multiple tuples, for example, the operator can receive as input a vector of integers and have single integers as output, when the operator receives the vector it returns as output every single integer contained

in the vector.

These operators have no state because the operations they perform depend only on the tuple being processed at that time and not on those processed in the past. For the same reason they have no replication constraints and they can have an arbitrary number of replicas.

These operators are used to implement other operators such as FilterMap and FlatMap. The former is the union of Map and Filter operations, the map function must return an Option where the tuples that are discarded by the Filter become None. The latter is the union of Map and Flatten operators, thus the map function can return 0, 1, or n tuples.

Sometimes it is necessary to do operations on the current tuple that also consider past received tuples. In this case, the operator needs to keep a state of the computation. Rich operators have been developed to support this kind of scenario. These operators keep an internal state that is used to process tuples and can be updated for each processed tuple. An example scenario that requires the use of rich operators is the following: we want to filter out all sequential duplicates, in other words, we don't want two consecutive identical tuples in the output stream. The simple Filter operator is not sufficient to implement this condition, but RichFilterMap allows us to do it. The state to be maintained is the last tuple processed by the operator, and the filter condition is that the current tuple must be different from the saved one.

Rich operators don't have replication constraints but keep in mind that the state of these operators is local and is not shared among replicas. Consider the previous example, if the operator RichFilterMap is replicated and the stream without duplicates is entirely collected by a CollecVec, the collected stream may contain two consecutive identical tuples if those tuples are flowed in two different replicas of RichFilterMap.

Rich operators in Noir are:

- **RichMap**: rich version of Map. This operator takes a FnMut closure that specifies the mapping function. The mapping function is cloned for each replica.
- **RichFilterMap**: rich version of FilterMap, is implemented with RichMap.
- **RichFlatMap**: rich version of FlatMap, is implemented with RichMap.

1.3.4. Aggregates

These operators are used to extract aggregate data from the stream. They require a function that describes how to aggregate all the tuples they receive into a single accumulator.

They need to process the entire stream to compute the final result and as soon as the result is ready is forwarded to the next operator. There are two types of operators: **Fold** and **Reduce**. Fold takes as input the initial value of the accumulator and the aggregation function; the type of the accumulator and the type of the tuple can be different. Reduce, on the other hand, takes as input only the aggregation function, and the accumulator is initialized to the first tuple of the stream. Both operators are blocking and cannot be replicated because they need all the tuples of the stream to produce the final result.

To mitigate this constraint there is a distributed version of these operators where the aggregation operation is split into two phases. In the first phase, a partial calculation of the aggregate is performed by each replica on its portion of the stream; in the second phase, the partial results are aggregated. The computation of partial results can be done in parallel while the computation of the final result cannot because all partial results are needed to compute the final result. AssociativeFold and AssociativeReduce require two aggregation functions: the first to compute the partial results, and the second to produce the final result from all partial results.

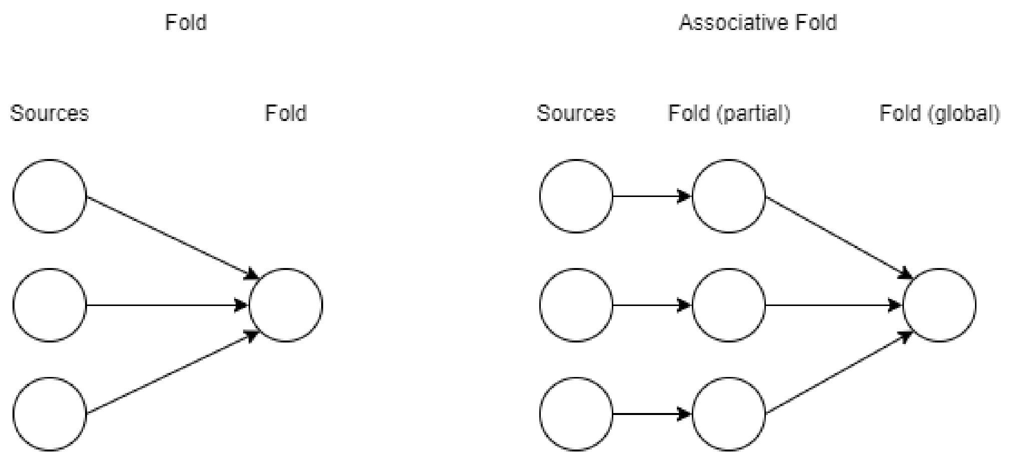

Figure 1.3: Fold vs AssociativeFold

Noir also provides some common useful aggregates like max, min, sum, count, and some others. All these functions are built on top of the Fold operator, also the Reduce operator, and the associative versions are all built on top of the Fold operator.

1.3.5. Partitioning

This category includes those operators that allow splitting the stream into logically independent substreams and redistributing the tuples to all the different replicas of the next operator. There are two ways to split and distribute the stream: random distribution or key-based distribution. The first method is implemented in the **Shuffle** operator, each received tuple is forwarded to one of the next replicas, chosen randomly. This operator allows parallelism changes in the flow, typically it is used after those operators that cannot be replicated to allow parallelization for subsequent operators

Partitioning by key, on the other hand, is done through **GroupBy**. The operator requires a function used to compute the key starting from the tuple, after that the stream becomes a KeyedStream where the elements will be tuples composed of the key and the old element. The key also determines the replica of the next operator to forward the tuple to. All tuples with the same key are sent to the same replica.

The **KeyBy** operator just performs the computation of the key and the transformation of the stream into a KeyedStream. All created tuples are sent to the next operator with OnlyOne strategy, the next operator receives tuples with different keys, and it is not guaranteed that all tuples with the same key arrive at the same replica.

The **UnKey** operator is the opposite of the KeyBy; it turns a KeyedStream into a simple stream by removing the key from the key-value tuple and returning only the value.

1.3.6. Timestamps

When data streams represent a series of events, the concept of time is also important for processing purposes. For example, if the Sources receive readings from sensors, it is useful to know the time at which the reading was taken because it can be a relevant factor for analysis purposes. Two important times can be distinguished: event time and processing time. The former represents the time when the event occurred, for example, the time when the reading was taken from the sensor. The event time is fixed and depends only on the event. The latter represents the time when the data is processed, unlike the first this is not fixed and can change each time the data passes through the operators.

Noir implements the **AddTimestamps** operator that allows you to specify the logic to get the event time from each tuple, for example by extracting the value from one of the tuple attributes. This operator also modifies the token that contains the tuple in the flow, changing it from Item(item) to Timestamped(item, event time) so that subsequent operators can properly handle the event.

It may happen that events flowing through the network are not sorted by event time and that event A arrives before event B despite the event time of A being later than the event time of B. This is why there are special mechanisms for dealing with late arrivals:

watermarks.

The **Watermark** is a special token that represents a barrier. When an operator receives a watermark it records the timestamp contained in it and from that instant all events that arrive with an event time lower than the watermark are discarded. In other words, the watermark establishes a validity limit for events late arrivals. If an event arrives too late, it is no longer relevant for analysis.

1.3.7. Join

Another useful operation that is frequently applied to streams is the join. If we consider two streams as two tables with a potentially unbounded number of entries, then the join operation between two streams is similar to the SQL join command applied to the two tables. Also in this context, the join is applied on a specific field of the tuple and we can distinguish 3 modes:

- **Inner**: the output consists of all the tuples that have a match in both the left stream and the right stream, the produced tuple has type <Left, Right>
- **Left**: all tuples in the left stream are present in the output, while only those tuples of the right stream that match are present, the produced tuple has type <Left, Option<Right». The tuples of left that do not match in output become (Left, None).
- **Outer**: all tuples of left and right are present, and the produced tuple has type <Option<Left>, Option<Right».

All 3 types of joins require the specification of the ship strategy and the local strategy. The ship strategy defines how the tuples of left and right streams are distributed among all the replicas of the Join. These are the options:

- Hash Repartition: as with the group-by operator, tuples are distributed by key.
- Broadcast-forward: tuples in the left stream keep their own distribution while tuples in the right stream are broadcast to all replicas.

The algorithm used to check tuple matching can be chosen from the following local strategies:

- Hash Join: the tuples of the two streams are maintained in two hash tables indexed by the key.
- Sort and Merge: the tuples of the two streams are maintained in memory and sorted

by key.

For timestamped streams, there is also the **IntervalJoin** operator, which allows tuples of two streams to be associated based on their timestamps. Specifically, a left stream tuple with timestamp tl is associated with a right stream tuple with timestamp tr if and only if tl - lower_bound < tr < tl + upper_bound, where lower_bound and upper_bound are the parameters passed to the operator. In other words, two tuples are joined if their event times are close enough.

1.3.8. Windows

The window is another useful and common abstraction used for stream analysis. It allows to split the stream into portions, called windows, and to perform independent processing on each window. The size of windows can be defined by three different properties:

- **Count**: the number of tuples is the parameter to get the window, so the order in which tuples arrive influences the content of the windows.
- **EventTime**: windows are made on the base of the event time. In this case, the role of the watermark becomes crucial since it can sign the closing of the windows.
- **ProcessingTime**: windows are sized on the processing time.

The ways in which windows are created are:

- **Sliding**: windows are defined by 2 parameters: size and step. The first indicates the length of the window, the second indicates how often a new window is created. If step < size the windows have overlapping, which means that a tuple can be present in multiple windows.
- **Tumbling**: windows are defined only by the size parameter that specifies the length, when a window is closed, immediately a new window is opened. These windows don't have overlaps.
- **Session**: this type requires the window to contain all consecutive elements that have a distance less than or equal to the gap parameter. This mode is available only for temporal windows, it is not supported by Count Windows.

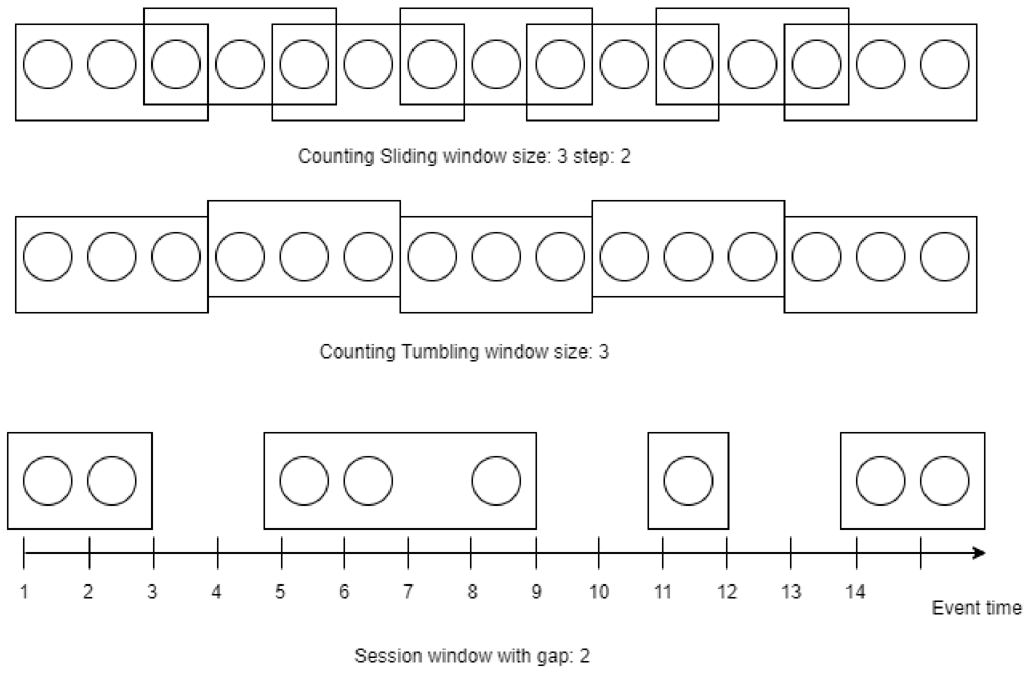
Counting Sliding window size: 3 step: 2
Counting Tumbling window size: 3
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Event time
Session window with gap: 2

Figure 1.4: Windows examples

1.3.9. Iterations

Iterative operators are a special kind of operators that require a separate discussion because they introduce loops into the dataflow graph. This implies that a tuple may flow through the same operator several times, which may cause some issues and some properties such as proper termination and exit from the loop, assurance that no tuple is lost in the loop, and proper handling of watermarks need to be guaranteed. Typically iterations are stateful in the sense that at each iteration the state is used for computation and a new state is created. This leads to an additional problem: a mechanism is needed to synchronize state access by all replicas to ensure consistency of shared state and avoid deadlocks.

Noir provides two operators for iterative computations: **Iterate** and **Replay**. The former transforms the stream at each iteration and has two outputs: the transformed stream and the final state. The latter always iterates over the initial stream and has only the final state as output. The structure of these operators is similar; Iterate and Replay receive data from the previous blocks and, at each iteration, send it to the body. All computation and processing that has to be repeated at each iteration is contained in the body of the loop; it consists of a dataflow in an independent scope with respect to the operators outside the loop. The task of the body is to generate the delta updates, which are the updates to the state obtained at each iteration. For the Iterate operator, the body also has the role of generating the new tuples to be used at the next iteration. The delta updates are sent to the Iteration Leader, which is a special, non-replicated operator (it has a single instance) that cannot be controlled from the outside. The task of the Iteration Leader is to compute the new state at each iteration via the delta updates. Once all the stream tuples have been sent to the body, Iterate or Replay sends the FlushAndRestart token to the body. When the body operators receive the FlushAndRestart they know that the iteration has ended, so they send the computed delta update to the IterationLeader and reset their internal state to be ready to start a new iteration. The Iteration Leader waits for delta updates from all body replicas, after which it updates the state and sends it to all replicas of the Iterate or Replay operator. Both operators are blocking and work only on bounded streams. In fact, they need to receive all stream elements before they can complete even the first iteration, and they need the entire stream to complete the iteration and compute the new state. The incoming stream is stored by both operators, Replay holds it until the end of the computation and uses an index to scroll it while Iterate replaces it with the new stream generated at each iteration. After sending the FlushAndRestart token, the Iterate or Replay operator must wait for a new state from the IterationLeader before starting a new iteration. This architecture ensures state consistency and synchronization for state reads and writes, and also allows the creation of nested iterations. At the end of the computation, when the Terminate token flows through the body and arrives at the IterationLeader, the loop ends, and the final state contained in the IterationLeader is sent to subsequent operators in the network. The Iterate operator also has a second output through which it sends the new stream generated by the last iteration. The body also can execute the Join operator with a stream from outside the loop, called side input. The side input needs to be a bounded stream and it is entirely read and cached during the first loop iteration similar to what happens to the input stream of Iterate and Replay.

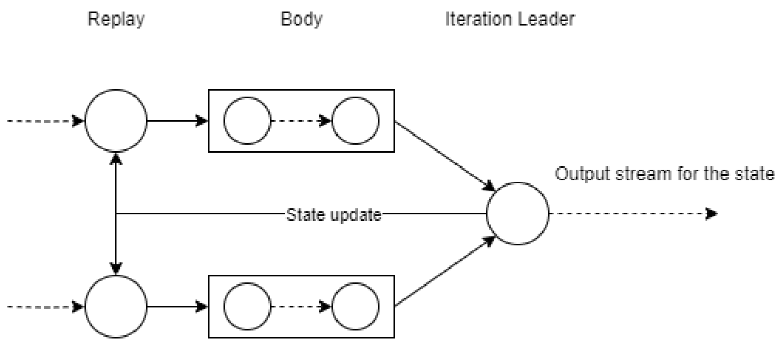

Figure 1.5: Replay operator

The figure 1.5 represents the iteration structure of the Replay operator with replication equal to two. The Body is composed of some operators. The state update edge represents the channel used to send the newly computed state from the IterationLeader to all replicas of Replay.

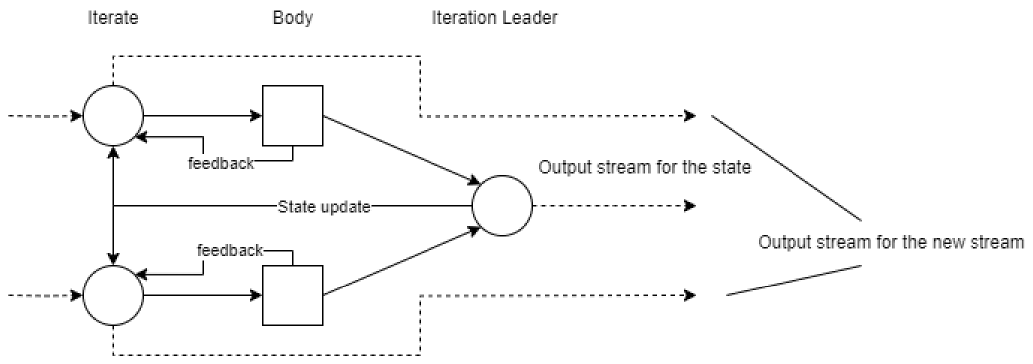

Figure 1.6: Iterate operator

The figure 1.6 represents the Iterate structure. This presents some differences with respect to Replay. The feedback edges from the Body's replicas to Iterate's replicas are the channel used to send the tuples computed by the Body for the new stream. The dashed edges going out from Iterate represent the channels used to output the new stream with the tuples generated during the last iteration.

2 | Snapshot algorithms

A simple program generally consists of variables and instructions that operate on them; the content of the variables at a given time of execution is called the state of the program. If we think of a bit more complex system, the state can be a database. In the dataflow paradigm, on the other hand, there is no shared global state but each operator has its own independent local state, the only data shared between operators are stream tuples and other metadata necessary for the proper functioning of the system that are exchanged via message. The state of a distributed dataflow system consists of the internal states of each operator and the messages that are being exchanged.

Snapshot algorithms[23] aim to save the global state of a distributed system on a durable and reliable source. A snapshot is a kind of picture of the system at a given instant that captures the local state of each individual component and the messages that the components are exchanging at that instant. Typically, snapshot algorithms involve periodic checkpoints.

There are two ways to perform a checkpoint:

- blocking: execution stops, no more messages are exchanged, and only the internal states of the components are saved. If stopping and restarting execution is done correctly without losing or changing the order of tuples, the snapshot is consistent.
- non-blocking: it allows you to take a snapshot without stopping the execution. In this case, it is more complex to guarantee snapshot consistency. In fact, it may happen that checkpoints of individual operators are not performed at the same time for all operators, so different algorithms adopt different strategies to make sure that a consistent snapshot can always be achieved.

The recovery phase involves resetting the state to the last valid snapshot and resuming computation from that point. The choice of algorithm best suited to the specific case depends on the architecture and implementation of the system and the following evaluation metrics:

- checkpoint overhead: the increase in execution time due to the snapshot procedure.

- recovery latency: the time required to recover computation from a saved snapshot.

2.1. Chandy-Lamport

Chandy-Lamport[17] is a distributed snapshot algorithm for a general distributed system and is the base of some other specific algorithms. It considers a distributed system as a set of processes that communicate with each other with messages exchanged through channels. Processes have their local internal state and there is not a global state shared between them. If we represent the system as a graph, the processes are the nodes and communication channels are the edges.

It works under the following assumptions:

- Any two processes of the system are connected by a communication channel (strongly connected graph)
- FIFO communication channels
- Reliable channels and processes

Any process can initiate the snapshot by executing these steps:

1. Record its local internal state
2. Send a specific token on all its outgoing channels
3. Start a local snapshot by recording incoming messages

The same procedure is executed when a process receives the token for the first time. When any process receives a token it stops recording incoming messages from that channel. All messages other than tokens are processed transparently and without any interference from the snapshot. When a process receives the token from each incoming channel it can consider its local snapshot complete and can send it to a single collector of the global state. The algorithm terminates and the global state is available when all processes receive the token on each of their in-going channels, which means when all processes complete their local snapshots.

2.2. Asynchronous Barrier Snapshotting

Apache Flink supports fault tolerance by implementing this snapshot algorithm[16]. From the generic Chandy-Lamport algorithm that works on strongly connected generic graphs has been derived this specific algorithm for directed graphs that covers the case of both acyclic and cyclic graphs. As the name suggests, the idea is to create a snapshot of the global state of the system incrementally with a special token that flows in the operator network and acts as a barrier to delimit the portion of the stream already processed included in the snapshot from the portion not included in the snapshot.

It works under these assumptions:

- FIFO communication channels
- Reliable channels and processes
- Channels can be blocked or unblocked, when they are blocked they can buffer messages but they cannot deliver any message until they have been unblocked

The snapshot procedure in the case of directed acyclic graph (DAG), that is without iterations is the following:

1. A central coordinator injects a special snapshot token into all Sources, that save their state and forward the token to the next operators,
2. When an operator within the network receives a snapshot token from one of its input channels, it blocks that channel and continues to receive messages only from its unblocked channels,
3. When an operator receives the snapshot token from its only unblocked channel, it saves its internal state and forwards the token to the next operators,
4. When all Sinks receive the snapshot token and save their internal state the snapshot is complete.

With this procedure it is possible to avoid saving the tuples received during the snapshot thus the state is lighter, however, the price to pay is a lower asynchronous execution due to the token alignment done by blocking the channels.

In the case of cyclic graphs, thus in the presence of iterations, we have iteration-specific operators that have standard input channels and a back-edge channel from the end of the loop that is used to send the results of an iteration to the head of the loop in order to start a new iteration. For this type of operator, the procedure does not work since the operator would wait indefinitely for the snapshot token from the back-edge but this can never arrive since it has not yet entered the loop. The procedure for cyclic graphs is the same as the acyclic procedure for all operators that do not receive data from back-edges, while for all operators (the iteration-specific operators) that have a back-edge between the input channels it works like this:

1. when it receives the snapshot token from a standard channel it blocks that channel,
2. when it receives the token from the last standard unblocked channel, it makes a copy of its state and forwards the token (it does not unblock standard channels)
3. from that moment it buffers all tuples received from the back-edge
4. when the snapshot token arrives from the back-edge, it saves the state copy and the buffer with all tuples received from the back-edge
5. finally unlocks all standard channels and continues execution

2.3. Other solutions

There are different algorithms and implementations to perform snapshots. Here I will introduce some examples.

Naiad[19] is a distributed dataflow system for batch and stream processing based on timely dataflow that can support cyclic dataflows. It provides fault tolerance with a synchronous blocking snapshot algorithm that works like this:

1. when it's time to do a snapshot it stops message delivery
2. it flushes message queues, messages are saved in buffers
3. each process checkpoints its own local internal state
4. execution restart and message buffers are flushed
5. the checkpoint is complete after it has been saved

To recover the execution after a failure, the last saved checkpoint is taken and each process sets its local internal state to the checkpoint. This snapshot procedure has a high impact on throughput and latency.

Apache Samza[10] is a distributed streaming processing system designed for low-latency, highly scalable data stream processing. Originally developed by LinkedIn, it later became an Apache project. Its primary use cases include streaming ETL, event-based applications, and real-time streaming analytics. In Samza, a Stream is divided into independent partitions, and each partition is an ordered and replayable sequence of tuples. Each tuple is associated with an offset that uniquely identifies it within the partition. The offsets are ordered within the partition, allowing you to identify the position of a tuple within the stream. The stream is very similar to a Kafka[4] topic, which is one of the most commonly used brokers for managing Samza streams. Processing operations are performed through

Tasks. Typically, a task reads tuples from one or more streams and produces an output stream or writes results to a database. A task is uniquely associated with a partition of the input stream. Therefore, if there are three tasks, the input stream must have three partitions. A task can read from a partition starting from any offset, making tasks independent of each other. The checkpointing[11] algorithm leverages stream properties to determine how far the stream has been processed. When it is time to create a checkpoint, it saves the current offsets of all input partitions. When restarting from that checkpoint, Samza starts reading partitions from the saved offsets. Two checkpointing solutions are possible: saving checkpoints locally to disk or using a dedicated stream. The former requires to re-instantiate the task on the same machine to recover the state, while the latter avoids this problem. In most cases, a specific Kafka topic is used. To maintain the internal state[12] of the operator, another dedicated stream called the "changelog" is used. All changes to the internal state are saved in this stream. To recover a specific state, you can read the changelog until the desired offset to retrieve all modifications to the internal state.

Apache Storm[13] is a distributed real-time computation system also based on the dataflow paradigm. Its use case is the reliable processing of unbounded data streams for applications like real-time analytics, online machine learning, ETL, and all other data-intensive contexts. Its operator graph is called "topology" and is composed of Spouts and Bolts. A Spout is similar to a Source in Noir; its task is to produce tuples for processing, getting them from an external source. Bolts, on the other hand, contain the processing functions to be applied to the stream, such as filters, aggregates, joins, and so on. The fault tolerance mechanism is based on checkpoints and tuple acknowledgment. The checkpoint[14] algorithm works as follows:

1. A special Spout called "CheckpointSpout" generates a special tuple called "CheckpointTuple."
2. The CheckpointTuple flows through a separate dedicated stream, which is wired by the topology builder and has the CheckpointSpout as root.
3. When a Bolt receives the CheckpointTuple, it saves its internal state, acknowledges the CheckpointTuple, and forwards the CheckpointTuple to the next Bolt.
4. If a Bolt has multiple input streams, it has to wait for the CheckpointTuple on all its inputs before proceeding with the checkpoint.
5. When all Bolts acknowledge the CheckpointTuple, the checkpoint is complete and committed.

Spouts do not save any state because they use a tuple acknowledgment mechanism to determine which tuples were successfully processed. When a tuple has been fully processed by the entire topology, an acknowledgment is executed on it, indicating to the Spout that the tuple has been successfully processed. In case of a failure, the "fail()" method is executed, indicating to the Spout that it needs to replay that tuple because the processing was not successful.

3 | Snapshotting in Noir

Before going into the details of the snapshot algorithm, we need to consider some aspects of Noir architecture and the goals we want to achieve. The purpose of state persistence is to be able to resume execution from an intermediate point of the computation without having to start from scratch. The ideal use case for which the snapshot algorithm has been modeled is as follows: saving the state at regular time intervals so that the impact on performance is relatively low but at the same time ensures that there is always a global state that is not too outdated for the application context. In other words, this algorithm is designed to be applied to long-term and computationally expensive executions or for unbounded streams (i.e., continuous processing), where snapshots are taken at a frequency that allows the time needed for recovery and reprocessing of lost tuples to be compatible with the specific scenario, and where the impact of snapshots on throughput and latency is acceptable. For example, consider a processing scenario that involves training a machine learning model that takes hours to complete; it may be reasonable to use a snapshot frequency of a few minutes. Another choice made in the implementation of the algorithm is to favor fail-free execution, meaning to limit the time cost of the snapshot procedure. This is why we chose a non-blocking snapshot algorithm to limit the overhead associated with the snapshot procedure. Additionally, Noir does not have a centralized coordinator, making the implementation of a non-blocking algorithm much simpler.

3.1. Algorithm

We start with some assumptions about the guarantees provided by Noir:

- Communication channels are reliable and guarantee FIFO orders.
- Processes (operators) are reliable.

These assumptions satisfy the requirements of the ABS algorithm but not the ones of Chandy-Lamport. In fact, it is not true that all operators are connected by a communication channel, but this is not a problem. Using an approach similar to ABS, a special snapshot message can be propagated through the network of operators from Sources to

Sinks. After all operators have saved their states, we obtain a global snapshot of the system.

The special message consists of a snapshot token containing the identifier of the snapshot, referred to as **SnapshotId**.

```
pub struct SnapshotId {
    snapshot_id: u64,
    terminate: bool,
    pub(crate) iteration_stack: Vec<u64>,
    pub(crate) iteration_index: Option<u64>,
}
```

Listing 3.1: SnapshotId structure

The SnapshotId has the role to uniquely identify the global state taken during that snapshot, and it contains the following:

- snapshot_id: a sequential index starting from 1 and incremented by 1 each time a new snapshot is started,
- terminate: a flag that indicates whether it is a Terminate Snapshot (explained further below),
- iteration_stack: a secondary index used for iterations (see the Iterations section),
- iteration_index: another index to distinguish iteration loops (see the Iterations section).

The indices of SnapshotIds must always guarantee the following properties:

- **Monotonicity**: It is not possible to take two snapshots with the same index, and it is not possible to take a snapshot with an index lower than a previously taken snapshot. Otherwise, an operator could save two different states in the same snapshot, leading to an inconsistent and invalid snapshot.
- **Regularity**: The index of the snapshot must be equal to the index of the last taken snapshot plus one. This is crucial because it allows us to determine if an operator has taken a specific snapshot by knowing only the last taken snapshot. In other words, if the last snapshot taken by an operator has an index of “i”, then the operator has also taken all previous snapshots with an index less than “i”.

Some examples of sequences of SnapshotId:

S1, S2, S2	Wrong: violate monotonicity property and regularity property
S1, S2, S4	Wrong: violate regularity property, S3 has not been taken
S2, S3, S4	Wrong: the index must start from 1, the only case where it does not start from 1 is when restarting from a saved snapshot, in which case it starts from the index following the snapshot since all previous ones are still saved and valid
S1, S2, S3	Ok: complies with all the properties

Table 3.1: Properties of SnapshotId.

Snapshot tokens are generated by Sources independently; there is no central coordinator responsible for producing and injecting tokens into the dataflow. The generation of snapshot tokens can be configured based on the number of generated tuples or at regular time intervals (see the Configuration section for more details). However, it is important to note that, given the pull-based architecture of the operators, token generation only occurs when the operator immediately downstream of the Source calls the "next()" function on the Source.

Snapshot generation is implemented and managed by the SnapshotGenerator structure. With each "next()" call, the Source checks if the conditions for generating the snapshot token (either a specified number of generated tuples or an expired time interval) are met. If the conditions are met, the SnapshotGenerator is reset, setting the number of produced tuples back to 0 and resetting the timer for the interval. Then, it returns the snapshot token. If the conditions are not met, it increments the tuple counter and then generates and returns a new tuple.

When an operator receives a snapshot token, it follows the following snapshot procedure:

1. The operator receives a snapshot token with index "i" from one of its input channels "channel1."
2. It makes a copy of its internal state.
3. It sends the snapshot token to the downstream operators.
4. For each input channel other than "channel1," all tuples that arrive before the snapshot token with index "i" are buffered.
5. Both the buffered tuples and the non-buffered ones are processed as usual.
6. When the token with index "i" has been received from all inputs, the state is saved.

In the end, the operator state consists of its internal state and the buffer containing the tuples received during the snapshot.

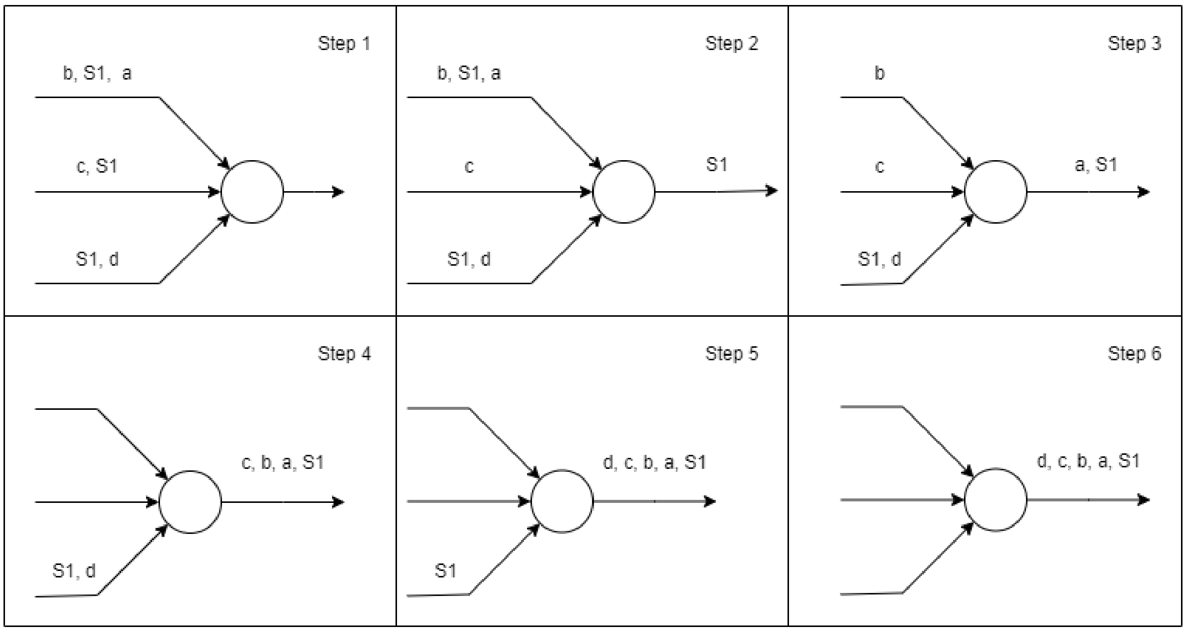

Figure 3.1: Snapshot algorithm

The figure 3.1 shows step-by-step the operations of the algorithm:

1. Initial situation: an operator receiving data from three channels is represented, and the contents of the channels are also shown.

2. The operator reads snapshot token S1 from the second channel, makes a copy of its internal state and forwards S1 to the output channel.

3. The operator reads tuple "a" from the first channel, adds it to the queue for tuples received during the snapshot, processes it, and forwards it to the output channel. Then, the operator also reads token S1 from the first channel. At this point, any tuples arriving from the first channel will no longer be added to the queue. Token S1 is not forwarded because it has already been sent.

4. The operator first reads "b" from the first channel and then "c" from the second channel. In both cases, it simply processes and forwards the two tuples; they are not saved in the queue because both the first and second channels have already sent token S1.

5. The operator reads "d" from the third channel; this channel has not yet sent S1, so the tuple is added to the queue. Then it is processed and forwarded.

6. The operator reads S1 from the third channel. At this point, it has completed the snapshot procedure and can save the state, which consists of the copy of its internal state made in step 2 and the queue with tuples "a" and "d."

This kind of procedure avoids blocking channels as Flink does but it requires saving the queue with the tuples received during the snapshot. This can consume a significant amount of memory and result in very large snapshots. However, in Noir, only the Start operator and the iterative operators have multiple input channels. All other operators have a single input channel, making the procedure much simpler and lightweight:

1. The operator receives a snapshot token "i" from its single input channel.
2. It saves its internal state.
3. It forwards the snapshot token to the downstream operators.

Iterative operators are addressed in the "Iterations" section because they require a dedicated algorithm due to the introduction of cycles within the operator graph and because of their specific implementation and functioning.

A snapshot is considered complete only when all operators in the network have saved their respective states. With this procedure, it is possible for Sinks to save their state before other internal operators in the network. Therefore, the only way to ensure that you have a complete snapshot is to verify that all operators have saved the state corresponding to the SnapshotId contained in the token. This means that by setting a high enough frequency for generating snapshot tokens, it could happen that a new snapshot is initiated without the previous one being completed. However, this is not a problem. Operators with multiple input channels can maintain partial states for each ongoing snapshot. This is explained in detail in the section describing the state of the Start operator.

Another situation that requires attention is when some Sources terminate before others. This is a relatively common event due to the asynchronous and uncoordinated nature of Sources. When this happens, it is possible that the active Sources continue to produce snapshot tokens while the Sources that have terminated can no longer do so. This could lead to the storage of incomplete snapshots and, if the Sources are heavily imbalanced, result in periods without snapshots. For this reason, a special **Terminate Snapshot** has been introduced. Its SnapshotId has the "terminate" flag set to true, indicating that it is the last snapshot taken by an operator before terminating processing. The Terminate Snapshot is implicitly taken before forwarding the Terminate token to the next operator. Terminate SnapshotId does not flow through the network like normal SnapshotIds because it is closely tied to the Terminate token. The index of the SnapshotId is determined by those saved previously, thanks to the properties of monotonicity and regularity.

When an operator with a single input channel receives a Terminate token, it saves its state with the special Terminate SnapshotId and forwards the Terminate token to the next operator.

For operators with multiple input channels, the procedure is a bit more complex:

1. The operator receives a Terminate token on channel "a."
2. If there is an ongoing snapshot, it removes "a" from the list of channels that are waiting for the snapshot token.
3. If there are other channels that still need to send the Terminate token, the operator does not send anything to the downstream operators but waits to receive a message from the other channels.
4. If this is the last Terminate (all other channels have already sent it), the operator does a Terminate Snapshot and forwards the Terminate token to the downstream operators.

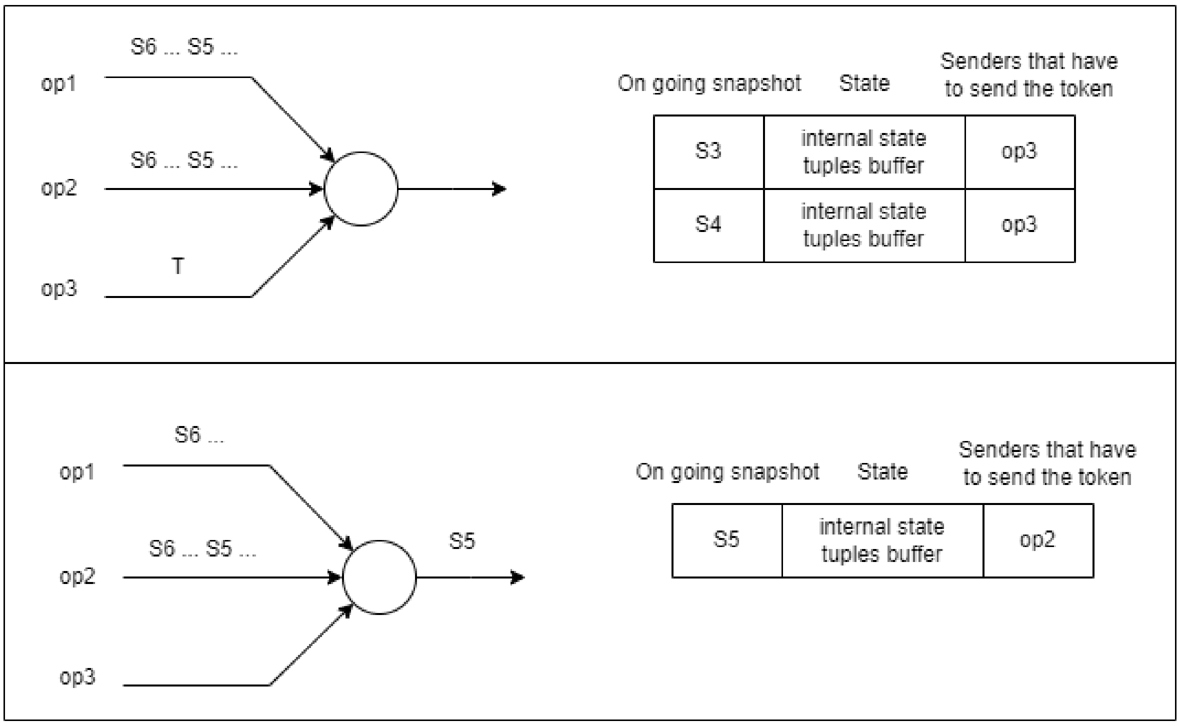
op1
S6 ... S5 ...
op2
S6 ... S5 ...
op3
T
On going snapshot
State
Senders that have to send the token
S3
internal state tuples buffer
op3
S4
internal state tuples buffer
op3
op1
S6 ...
op2
S6 ... S5 ...
op3
S5
On going snapshot
State
Senders that have to send the token
S5
internal state tuples buffer
op2

Figure 3.2: Terminate snapshot

The figure 3.2 represents the evolution of the algorithm in the case of early termination of a Source (op3). In the initial situation (top part of the figure) we have op1 and op2 which are two operators that have taken snapshots until S4 and are going to produce more tuples and more snapshots tokens. On the other hand, we have op3, which has completed its work and has sent the Terminate token. The last snapshot taken by op3 is S2, so the central operator is still waiting for tokens S3 and S4 from op3 in order to save their respective snapshots. When the central operator receives the Terminate token from op3, it knows that no more messages will arrive from that operator. Therefore, it can conclude and save snapshots S3 and S4. If the central operator then reads S5 from op1, as shown in the lower part of the figure, it will follow the normal snapshot procedure but will not expect to receive S5 from op3. In other words, S5 is complete and can be saved as soon as it arrives from op2.

3.2. Operators state

Each operator saves its state autonomously and independently. This state is identified by a key formed by the combination of the operator coordinates and the SnapshotId. The operator coordinates uniquely identify each individual replica of every operator present in the execution graph. The coordinates consist of the following identifiers:

- Block of ownership
- Host on which it is instantiated
- Replica identifier
- Operator index within the operator chain

The state of the operators is a generic type that implements the ExchangeData trait:

ExchangeData: Data + Serialize + for<'a> Deserialize<'a>

"Serialize and Deserialize" are used to serialize and deserialize the state, while "Data" is a super-trait of "ExchangeData" that all types within a stream must implement:

Data: Clone + Send + ‘static

The ExchangeData trait is also implemented by all those types that use the network layer.

We can categorize Noir operators into two categories: **stateless** and **stateful**. The former does not have an internal state; the operations they perform on the tuple depend only on the tuple itself. The data structures of these operators contain only information that can be determined before processing. For example, the operator coordinates are a piece of data that does not change during the computation, it is assigned during setup and remains unchanged. For this reason, it is not necessary to save it in the snapshot. Stateless operators do not save any state to be persisted; when they receive a snapshot token, they simply forward it to the next operator. Some examples of stateless operators are Map, Filter, and Flatten.

Here are the main stateful operators and the state persisted in the snapshot.

3.2.1. Sources

In general, the state of a Source is used to precisely determine the tuples between snapshots, providing the ability to regenerate any lost tuples not included in the snapshot. The state of Sources depends on the specific type of Source:

- **Channel**: It is stateless and does not keep the received tuples in memory. Therefore, it has no way of receiving the same tuple again.
- **CSV**: The state is the byte position within the CSV file at which the Source has reached during reading.
- **File**: The state is the byte position within the file at which the Source has reached during reading.
- **Iterator**: The state is the number of elements generated up to that point. To return to the saved state, you simply need to consume the iterator again for the saved number of times.
- **Parallel Iterator**: The state is analogous to the simple version. For each replica, the number of elements generated up to the snapshot moment is saved. Since the function that generates the iterator for each replica is deterministic and based on the replica index, we are ensured that the same iterators are always assigned to the same replicas.

All stateful Sources have a state of fixed size because they only save a kind of index that allows identification of the snapshot point. The type of Source affects the type of guarantees that can be provided. A stateless Source, not having the ability to regenerate previous tuples, can provide at-most-once semantics. Stateful Sources, on the other hand, can provide exactly-once semantics as they can regenerate tuples from the exact snapshot point.

3.2.2. Sinks

The state of Sinks, if provided, contains the partial results generated by the network up to the snapshot moment. As suggested by the name, Collect-type Sinks maintain results and have a state:

- **CollectVec**: The state is the vector containing the results received up to the snapshot moment. The state size is variable and depends on the number of elements produced by the processing.
- **CollectCount**: The state is an accumulator with the sum of the elements received up to the snapshot. Therefore, the state size is fixed.

The **ForEach** operator, on the other hand, is stateless; it performs the specified procedure on each result it receives without maintaining any state.

Again, the type of Sink affects the guarantees. A stateless Sink can provide at-least-once semantics because it may produce the same result multiple times. A stateful Sink, on the other hand, can provide exactly-once semantics. When considering both Sources and Sinks together, we can determine the overall guarantees that can be offered by the network:

Source	**Sink**	**Overall guarantees**
stateless	stateless	none
stateless	stateful	at-most-once
stateful	stateless	at-least-once
stateful	stateful	exactly-once

Table 3.2: Sources and Sinks guarantees

3.2.3. Start

It is the only operator that can have multiple input channels (excluding iterative operators). This makes the snapshot procedure more complex, and the saved state includes the internal state of the operator and the buffer with tuples received during the snapshot.

To manage partial states during execution, it uses a HashMap:

SnapshotId - (StartState, Set<Coord>)

The SnapshotId is the key that identifies the ongoing snapshots and the tuple (Start state, Set of coordinates) contains the state to be persisted and the set of coordinates that is used to keep track of previous operators that have not yet sent the token with that SnapshotId.

The Start state includes both the internal state of the operator and the queue with tuples received during the snapshot. It is composed of:

- **missing_flush_and_restart**: The number of previous operators that still need to send the FlushAndRestart token,
- **watermark_frontier**: A structure that manages received watermarks,
- **wait_state**: A flag indicating if the Start operator needs to wait for a state update (this is used for iterations, as explained later),
- **receiver_state**: Start can have a SingleReceiver if it receives data from the replicas of a single operator or a BinaryReceiver if it receives data from the replicas of two different operators (thus receiving data from two different streams, as in the case of Join, for example). This field is used to save the state of the receiver. The differences between the two types of receivers and the state of each are described below,
- **message_queue**: A queue containing the tuples received during the snapshot.

The type of receiver, in addition to influencing the information to be persisted in the state, changes the snapshot procedure.

The SingleReceiver has no state because it only handles receiving new batches of data from the replicas of the previous operator without performing any analysis.

When a snapshot token arrives at the Start operator, the following procedure is executed:

1. Check if this SnapshotId is already arrived
2. If it has, remove the sender from the set of previous replicas for this snapshot
   (a) If the set is empty, the snapshot is complete and can be removed from the HashMap and persisted.
3. If it has not arrived:
   (a) Get current state (tuples in message_queue will be added later)
   (b) Create a set with the coordinates of all previous replicas except the one that sent the token
      i. If the set is empty, persist the state

        ii. Otherwise, add a new entry in the HashMap with the SnapshotId and the tuple (Start state, coordinates set)

    (c) Send the snapshot token to the next operator

Upon the arrival of any tuple, the Start operator checks for ongoing snapshots. If such snapshots exist, it searches for all partial snapshots in which the sender of this tuple is included in the set of coordinates of the previous replicas. For each of those partial snapshots, the tuple is added to the message_queue state field.

When a Terminate token arrives, the coordinate of the replica that sent it is removed from the set of coordinates in each partial snapshot. The arrival of the Terminate token signals that the stream from which it comes has terminated and no more messages will be sent from that stream. Furthermore, all operators through which that Terminate token has flowed have taken a Terminate snapshot. Therefore, all partial snapshots that the Start operator is currently taking do not need to wait for their respective snapshot tokens from that stream. The Start operator, in turn, performs a Terminate snapshot after receiving Terminate tokens from all its previous replicas and before sending the Terminate token to the next operator.

When recovering from a snapshot, the Start operator must process all the tuples within the message_queue before it can receive and process new tuples.

In the case of a BinaryReceiver, things become a bit more complicated. This type of receiver is responsible for receiving new batches of data from two different streams (conveniently referred to as left and right) and performs the following operations:

1. It wraps the elements from the left and right streams in a specific structure defined as BinaryElement.
2. If one of the two streams is cached, it manages the cache by inserting the new tuples or retrieving tuples from the cache if the cached stream has already been fully cached.

In order to execute these operations, the BinaryReceiver maintains a state that can be modified based on the tuples present in the batch it is analyzing. Furthermore, it processes an entire batch of data before forwarding it to the Start operator. To obtain a consistent state, it is necessary to implement a snapshot procedure inside the BinaryReceiver similar to the one explained earlier and to leave the Start operator with the task of receiving the BinaryReceiver state once the snapshot is complete, adding it to its own state, and persisting it. BinaryReceiver will also keep the queue with messages received during the snapshot.

Similar to the Start operator, the BinaryReceiver manages its ongoing partial snapshots using a HashMap:

SnapshotId - (BinaryReceiverState, Set<Coord>)

The state of the BinaryReceiver includes:

- **left**: The state of the SideReceiver for the left stream.
- **right**: The state of the SideReceiver for the right stream.
- **message_queue_left**: A queue containing messages from the left stream received during the snapshot. These messages are not individual tuples but entire batches of tuples.
- **message_queue_right**: A queue containing messages from the right stream received during the snapshot.
- Some other data: necessary to recover a consistent state.

The SideReceiver state on each side is responsible for saving the number of FlushAndRestart tokens that are still pending and the potential cache if the stream is cached.

3.2.4. End

This operator is located at the end of a block and is responsible for sending data to the next operators according to the Next strategy defined by the stream. Stream elements are grouped into batches, with each batch containing the coordinate of the block to which End belongs (to allow the operator receiving the batch to know who sent it) and a queue of stream elements. Since batching does not require any state, this operator is stateless and does not persist any state.

3.2.5. Fold

The state of the Fold operator is composed of:

- **accumulator**: the partial result calculated with all the tuples received before the snapshot token,
- timestamp: the maximum timestamp received so far, if present,
- watermark: the maximum watermark received so far, if present,
- other parameters that allow restoring the operator to a consistent state.

The size of the Fold operator state depends on the type of accumulator.

The KeyedFold operator, designed for keyed streams, has a state composed of:

- **accumulators**: each key belongs to a different substream, so there is an accumulator for each key. A HashMap indexed by key is used to store all the accumulators.
- **ready_elements**: a vector with complete results for all substreams. This is generated after the arrival of the FlushAndRestart token, and the results are moved from the HashMap to this vector to be forwarded to the next operators with future calls to "next()". This field is not present in the simple Fold operator because it produces a single result that it forwards to the next operator as soon as it receives a FlushAndRestart token from the previous operator. Here, instead, a result is produced for each key, necessitating this support vector.
- timestamps: the maximum timestamp received so far, if present. Also here, it needs to be divided by key, so a Key-Option<Timestamp> HashMap is used.
- watermark: the maximum watermark received so far, if present.
- other parameters that allow restoring the operator to a valid state.

The size of the KeyedFold operator state is variable because it depends on the number of different keys arriving at the operator. For each new key that arrives, a new entry is added to the HashMap. Clearly, the type of accumulator also influences the state size.

3.2.6. Richmap

As explained earlier, this operator applies a function to the input tuple to generate another output tuple. The mapping function is a closure[8] of the type:

$$\text{FnMut}((\&\text{Key}, \text{Out})) \rightarrow \text{NewOut}$$

Here, Key is the generic type for the key, and Out and NewOut are the generic types for the input and output tuples, respectively. The closure type FnMut indicates that this closure can be executed multiple times and captures mutable references from the context where it is defined, which it uses to generate the new tuple. The tuple (&Key, Out) is due to the fact that this type of operator is applied to KeyedStreams. To separately handle substreams created by different keys, the operator maintains a key-closure HashMap. This way, there is a mapping function for each substream, and consequently, the state of the function is independent for each substream.

When a new key arrives at the operator, a new entry is added with the key and a copy of the mapping function. When a tuple with a known key arrives, the closure is retrieved from the HashMap, and the new tuple is computed. In the case of a simple stream, a dummy key is added that is the same for all tuples. This way, you get a KeyedStream consisting of a single substream containing all the tuples.

The state for this operator corresponds to the state of the closures contained in the HashMap. However, from inside the operator, there is no control over the closure state since it is captured from the context in which it is defined. For this reason, a persistent version of the RichMap operator has been implemented. This version is very similar to the original one, except for the closure type. The function type for mapping becomes:

$$\text{Fn}((\&\text{Key}, \text{Out}), \&\text{mut State}) \rightarrow \text{NewOut}$$

The Fn closure type captures only immutable references, so there is no mutable state over which we have no control. The state is now explicitly passed through a mutable reference State. Like the original version, this operator is applied to KeyedStream and handles multiple substreams through a HashMap. The difference is that now the HashMap is of type Key-State and contains the state of each substream, while the mapping function is unique and is applied to all tuples along with the state corresponding to the key. The operator also requires the state with which to initialize the HashMap. The state saved by the operator is the entire map with keys and states present at the time of the snapshot. The size of the state depends on the number of different keys processed and the size of the State type.

3.2.7. Join

As mentioned earlier, Noir supports Inner Join, Left Join, and Outer Join operations, providing various strategies for distributing tuples among replicas and searching for matching tuples. However, concerning simple (non-keyed) streams, there are only two actual operators that perform the Join operation, and they differ from each other on the local strategy used for matching tuples:

- **JoinLocalSortMerge**: This operator uses two vectors to keep the tuples of the two streams in memory. It needs to store all tuples from both streams before it can search for matches and produce output results.
- **JoinLocalHash**: This operator uses two HashMaps to store the tuples. Results are produced as tuples arrive, comparing them with those that arrived earlier.

The ship strategy, which determines how tuples from both streams are distributed among the various Join replicas, only affects the Next strategy of the blocks through which the left and right streams arrive. In the case of Hash Repartition, the final blocks of both left and right streams have a GroupBy Next strategy so tuple distribution is based on the tuple key. However, in the case of Broadcast-forward, the final blocks of the right stream still have a GroupBy Next strategy, while the final blocks of the left stream have an OnlyOne Next strategy indicating that the left stream tuples maintain their existing locality. The type of Join (Inner, Left, or Outer) is passed as a parameter to the two operators and is used to determine how to handle tuples that do not find a match.

State of JoinLocalSortMerge:

- **left**: a vector containing tuples arriving from the left stream.
- **right**: a vector containing tuples arriving from the right stream.
- Other parameters, including two flags indicating if both streams have sent all their tuples.

State of JoinLocalHash:

- **left**: A structure that manages and stores tuples received from the left stream, which contains a HashMap indexed by key to store the tuples.
- **right**: A similar structure is used for tuples received from the right stream.

For KeyedStreams, there are two additional operators that implement the join function:

- **JoinKeyedInner**: Performs an Inner Join between the left and right streams, using local hash as the local strategy. Its state contains two HashMaps that store the tuples from the left and right streams received up to the snapshot moment.
- **JoinKeyedOute**r: Performs an Outer Join between the left and right streams, also using local hash strategy. Therefore, its state contains two HashMaps with tuples from both streams.

The last type of join to analyze is **IntervalJoin**. This operator also needs to store tuples in memory and uses a queue for the left stream and a HashMap for the right stream. As one can imagine, the state of this operator contains both the queue for the left stream and the HashMap for the right stream, along with other necessary information to enable recovery from the saved state.

In summary, regardless of the type of join and how it is implemented, it is necessary to store all or some of the tuples from two streams. Consequently, the state to be persisted contains the stored tuples up to the snapshot moment. This, of course, affects the size of the state saved in the snapshot.

3.2.8. Window

Noir abstracts all the logic and processing of the windows into a single operator called WindowOperator. There are two fundamental traits that characterize and specify the behavior of windows:

- **WindowManager**: This trait defines when to open and close windows according to the specific type of window. The window types can include CountSliding, EventTimeSliding, EventTimeSession, and so on. The process() function is used to process a new tuple arriving at the operator and contains all the logic for managing windows.
- **WindowAccumulator**: This trait specifies the operations to be performed on the elements within the windows. Typically, these operations involve aggregation functions such as Sum, Min, Max, First, and so on. This trait also has a process() function used to process new tuples.

These two traits work together to define how the windowing operations should be handled in Noir.

The WindowOperator works with KeyedStreams, although it can also be used with simple Streams. Similar to other operators, it uses a HashMap indexed by key to manage substreams generated by different keys. This HashMap contains WindowManagers as values, each handling windows for the respective substream. Some examples of WindowManagers include:

- CountWindowManager
- EventTimeWindowManager
- ProcessingTimeWindowManager
- SessionWindowManager

The names of these managers indicate the windowing logic they implement. For the first three, there are both sliding and tumbling versions.

All these types of WindowManagers maintain a queue of open windows, referred to as **Slots**, containing different data, depending on the nature of the window. However, all types of window slots have a field reserved for the Accumulator, which holds the window content. When a WindowManager receives a new tuple, it determines which slot (window) it belongs to and passes the new tuple to the corresponding Accumulator. The Accumulator processes the data incrementally, so the window content corresponds to the result obtained from the tuples belonging to the window that have arrived up to that point. In other words, the Accumulator contains a partial result until the last tuple belonging to the window is processed, at which point the result is complete. The Accumulator then returns this result to the WindowManager, which closes and removes the window from the queue. The WindowManager subsequently returns the result to the WindowOperator.

The state consists of the open windows at the time of the snapshot. However, it depends on the specific WindowManager and WindowAccumulator. For this reason, the WindowManager and WindowAccumulator traits have get_state() and set_state() functions added. This allows the WindowOperator to retrieve and set the state for any implementation of WindowManager and WindowAccumulator.

The state of some WindowManagers and their respective Slots:

- CountWindowManagerState: This state represents CountWindowManager and contains a queue of SlotStates, each corresponding to an open window. Each SlotState includes:
  - The number of tuples already processed
  - The state of the Accumulator
  - The optional timestamp of the last processed tuple
- EventTimeWindowManagerState: This state is specific to EventTimeWindowManager and includes the last received watermark and a queue of SlotStates. For each SlotState, you can find:
  - The window opening timestamp
  - The window closing timestamp
  - The state of the Accumulator
  - A flag indicating whether the window is still active
- ProcessingTimeWindowManagerState: This state is for ProcessingTimeWindowManager and contains a queue of SlotStates. Each SlotState is composed of:
  - The time elapsed since the window opened
  - The time remaining until the window closes
  - The system time at the moment of the snapshot
  - The state of the Accumulator

  - A flag indicating whether the window is still active

- SessionWindowManagerState: For SessionWindowManager, the state includes an optional SlotState specifying:
  - The state of the Accumulator
  - The time elapsed since the arrival of the last tuple
  - The system time at the moment of the snapshot

The state of the Accumulators corresponds to the partial result computed until the snapshot token arrives.

In the case of recovery from a snapshot using the set_state() method, the state of WindowManagers and Accumulators is restored. However, it is important to note that for ProcessingTime windows and Session windows, it is not guaranteed that processing will produce the same results. This is because these types of windows rely on system time, which changes from the moment the state is taken to when recovery is performed.

To maintain window consistency, the time elapsed from the snapshot moment to the recovery moment is considered. Specifically, ProcessingTime windows maintain the start and end times of the window as absolute times, and Session windows maintain the arrival time of the last tuple as an absolute time. In other words, the time that elapses between taking the snapshot and restoring the system is an interval during which no tuples can arrive, and there is no guarantee that the tuples arriving after recovery will have the same processing time as they would have had during normal execution without recovery.

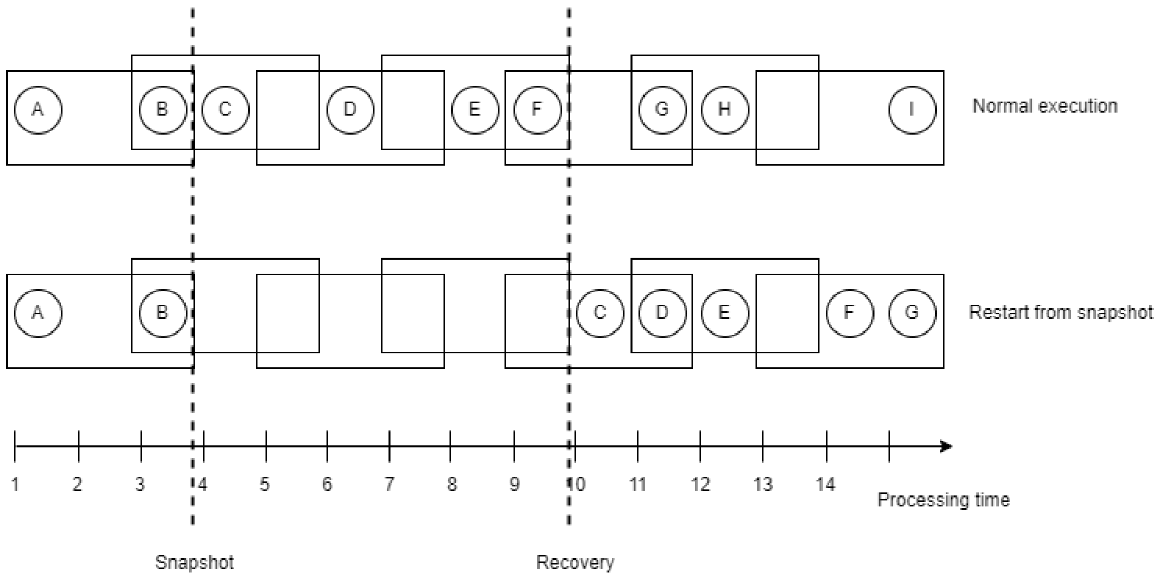

Figure 3.3: ProcessingTime window recovery

In figure 3.3, we can see a ProcessingTime Sliding window (size: 3, slide: 2). The top part represents a normal execution, while the bottom part illustrates the effects of recovering from the state saved in the snapshot. The tuples received after the snapshot no longer have the same processing time, which leads to modifications in the windows after the snapshot moment.

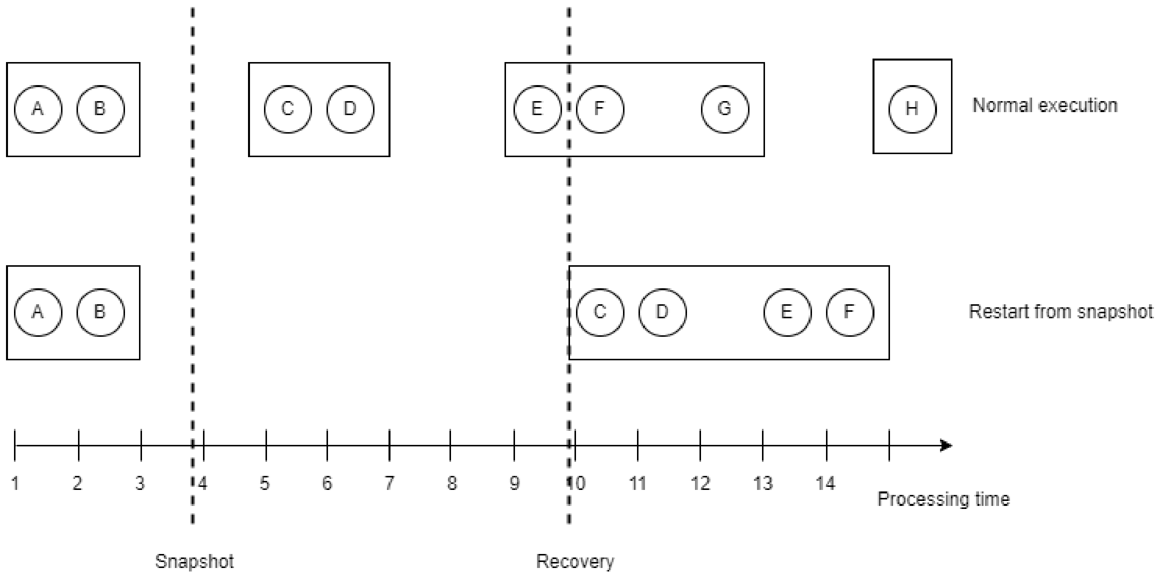

Figure 3.4: Session window recovery

The figure 3.4 shows the possible effects of recovery from a snapshot on a SessionWindow with a gap of duration 2. Here again, the recovery results in a change in the processing time of the tuples after the snapshot, which in turn modifies the content and duration of the windows.

3.3. Iterations

Iterations require a dedicated snapshot procedure due to their unique functioning, as described in Chapter 2.3.9. The main challenges are as follows:

A. Iterations require the entire stream, meaning that to complete the first iteration, you need to consume the entire stream. Additionally, iterations are blocking, in the sense that you cannot start a new iteration until the previous one has finished. This means that when the first iteration is completed, the Sources have already produced the entire stream, and their workers may have terminated. As a result, they can no longer generate and provide snapshot tokens to the flow. This requires finding a way to take snapshots during all subsequent iterations after the first one.

B. Snapshot tokens cannot pass through the same operator more than once because doing so would result in an operator taking a snapshot with a SnapshotId that has been used in the past, violating the monotonicity property of SnapshotIds.

C. The state used within iterations is synchronized among all replicas. Therefore, snapshots must be taken during the same iteration by all replicas of the iterative operator and by all operators contained within the iteration body. If two different replicas of an iterative operator take snapshots in two different iterations, the resulting global state is inconsistent, and recovery from it is not possible.

3.3.1. Algorithm

The algorithm we have designed aims to address the issues described as follows:

A. During the first iteration, snapshot tokens are generated exclusively by the Sources. They flow through the iteration body and reach the IterationLeader, which keeps track of the last persisted snapshot. From the second iteration onward, it is the IterationLeader that takes care of generating new snapshot tokens starting from the last SnapshotId it has tracked.

B. When the snapshot token reaches the end of the iteration, it is not sent back to the beginning of the loop like all other tuples. Instead, it is forwarded to the operators following the loop. This ensures that it does not pass through the same operator more than once.

C. The IterationLeader includes any snapshot tokens in the message used to send the state produced during an iteration. This guarantees that snapshots are taken during the same iteration by all replicas of the operators within the loop.

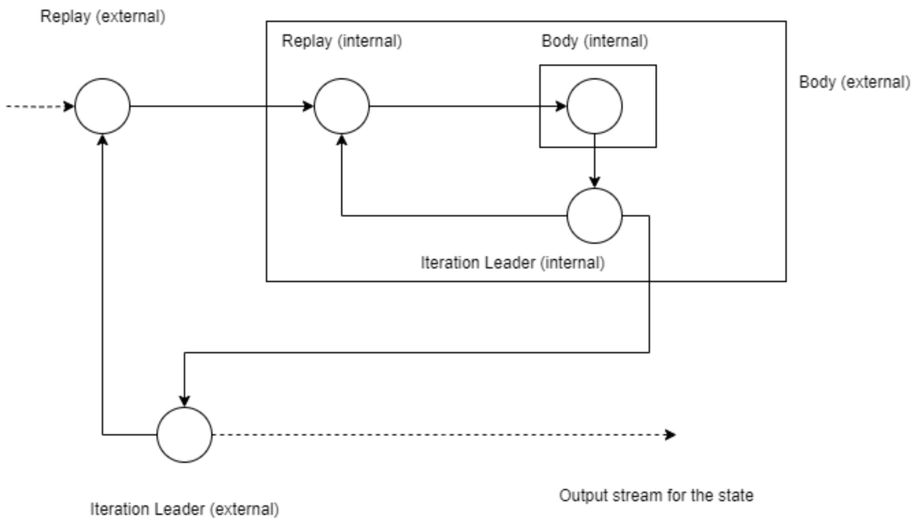
Replay (external)
Replay (internal)
Body (internal)
Body (external)
Iteration Leader (internal)
Iteration Leader (external)
Output stream for the state

Figure 3.5: Nested iterations

In the figure 3.5, there is an example of nested iterations, where the body of one iteration contains another iteration. In this scenario, we have two IterationLeaders generating snapshot tokens, which can cause some issues. When the innermost IterationLeader produces a token, it may not reach the outer Replay before the next outer iteration begins. As a result, operators may take the snapshot during different iterations, and the global state reported for that SnapshotId becomes inconsistent.

To address this, within an iteration, the SnapshotIds contain an additional piece of information indicating the iteration level. Specifically, a stack called "**iteration stack**" is used. All snapshot tokens within a loop have a stack with a length equal to the loop level. Tokens generated outside the loop (by a Source or by an IterationLeader at a lower level) have a "0" in the iteration stack at the position corresponding to the loop level for the current iteration. All snapshots generated by the IterationLeader at a specific iteration have a consecutive number.

If we consider the previous figure, we have three different stack levels:

- Snapshots outside both iterations have an empty stack, so they are of the form S1[], S2[], S3[], ...

- Snapshots inside the first iteration but outside the second have a stack with one level, so they are of the form S2[0], S2[1], S2[2], S3[0], ...
- Snapshots inside both iterations have a stack with two levels, so they are of the form S2[0,0], S2[1,0], S2[1,1], S2[1,2], S2[1,3], S2[2,0], S3[0,0], ...

If we define a new order and a new distance metric between two SnapshotIds, the properties of monotonicity and regularity remain valid. The **new order** between SnapshotIds is as follows:

- Sa < Sb if the index of "a" is smaller than the index of "b," or if the indexes are the same and when scanning the stack from the outermost level to the innermost level, the first position with different indexes has the index of "a" greater than that of "b."
- Sa = Sb if the index of "a" is equal to the index of "b," and when scanning the stack from the outermost level to the innermost level, all levels present in both stacks are equal.
- Sa > Sb in all other cases.

Some examples:

S1 [] < S2 []	lower index
S1 [0] < S1 [1]	lower level 1
S1 [0, 1, 0] < S1 [0, 1, 1]	lower level 3
S1 [0, 1, 0] < S2 [0, 0, 0]	lower index
S1[] = S1[1, 1]	same index and no different levels since the first stack is empty
S1 [0, 2] = S1 [0, 2, 1]	same index and equal common levels of the stack

Table 3.3: Order of SnapshotId with iteration stack.

The new **distance metric** to determine if a SnapshotId Sa is consecutive to Sb is as follows:

- The index of Sa is equal to that of Sb plus one, and the stack of Sa contains only 0.
- Or, when scanning the stack from the outermost level to the innermost level, the first position with a different index has the index of "a" equal to that of "b" plus

one, and all subsequent positions in the stack of Sa contain only 0.

Some examples:

S1 [] → S2 []	index incremented by 1
S1 [0] → S1 [1]	stack level one incremented by 1
S1 [0, 1, 0] → S1 [0, 1, 1]	stack level three incremented by 1
S1 [0, 1] !→ S1 [0, 3]	stack level two incremented by 2
S1 [0, 1] !→ S2 [0, 2]	index incremented by 1 but stack level two is not 0

Table 3.4: Distance of SnapshotId with iteration stack.

Snapshot procedure used during the first iteration:

1. When a snapshot token arrives at the iterative operator from the input channel, a level with a value of 0 is added to its iteration stack. The operator saves its internal state and forwards the new token to the operators in the body.
2. The operators in the body take the snapshot as usual.
3. When the IterationLeader receives the token for the first time, it copies it and saves its internal state. Then, it removes the last level from the iteration stack of the SnapshotId and forwards the new token to the external output channel of the loop.
4. After the IterationLeader has received all delta updates related to the first iteration, it initializes its SnapshotGenerator with the SnapshotId of the last taken snapshot. This last snapshot is also the last snapshot taken by all operators inside the loop.

After the first iteration is completed:

1. Before the IterationLeader sends the message with the new state to the replicas of the iterative operator, it checks if the SnapshotGenerator has produced a SnapshotId.
2. If there is a SnapshotId, the IterationLeader takes that snapshot and then it adds the SnapshotId to the message containing the new state to be sent to the replicas of the iterative operator.
3. When an iterative operator completes an iteration, it waits for a message from the IterationLeader. When this message arrives, it sets the new state and checks if a SnapshotId is also present.
4. If there is a SnapshotId, it follows the procedure to save its internal state and sends a stream element containing the SnapshotId to the operators in the body. This way,

the snapshot token is the first element in the stream that flows into the body.

5. When the token returns to the IterationLeader, it is simply ignored, and it is not sent to operators outside the loop.

With this procedure, snapshots taken during iterations remain invisible to the outside of the loop. In other words, the iterative operator, as seen from the outside, behaves like any other blocking operator (such as the Fold). This snapshot algorithm maintains this isolation because the snapshot tokens generated within the loop do not exit from it.

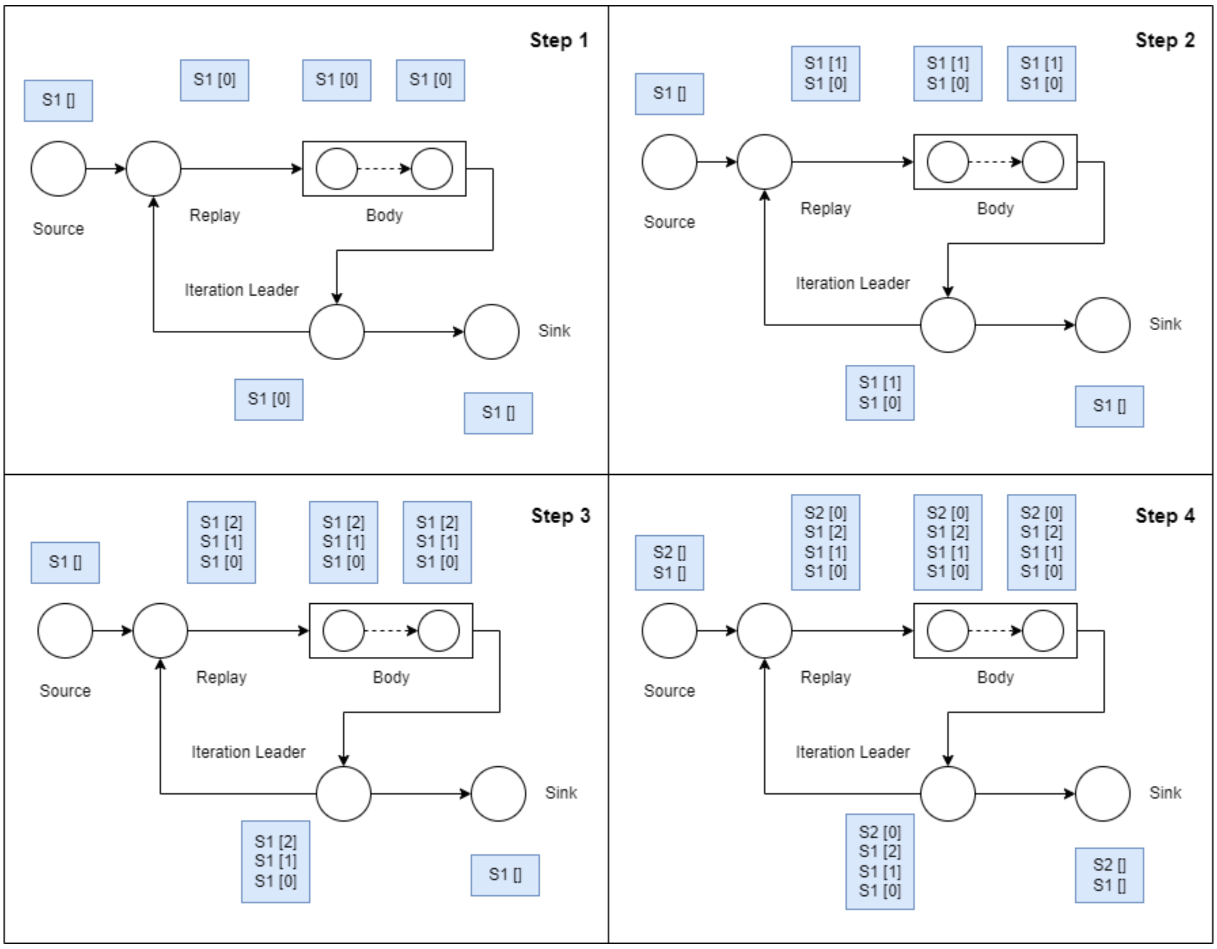

Figure 3.6: Iterations algorithm

The figure 3.6 represents the snapshot algorithm for iterative operators. In the blue rectangles, we can see the queues with SnapshotIds related to the snapshots taken by each operator. We look in detail at what happens at each step:

1. We are in the first iteration, the Source has produced a snapshot token S1[], which has already traversed the entire loop body and has reached the Sink.

2. We are in one of the loop iterations (not the first). The snapshot token S1[1] is produced by the IterationLeader and has already traversed the entire body. The token does not exit the loop.
3. The same situation as step 2 is repeated, and one or more iterations have passed since snapshot S1[1].
4. The loop has completed all iterations and requested new tuples from the Source. The Source has finished and sends the Terminate token, which flows through the network and reaches the Sink. S2 is the Terminate snapshot.

In the case of iteration loops on parallel streams or sequential iteration loops, we want to maintain the independence of each loop. In other words, when we search for the last valid SnapshotId to resume from, we want the iteration stacks of two parallel or sequential loops to be considered separately. This way, they can restart from the last completed iteration independently of the progress of the other loop.

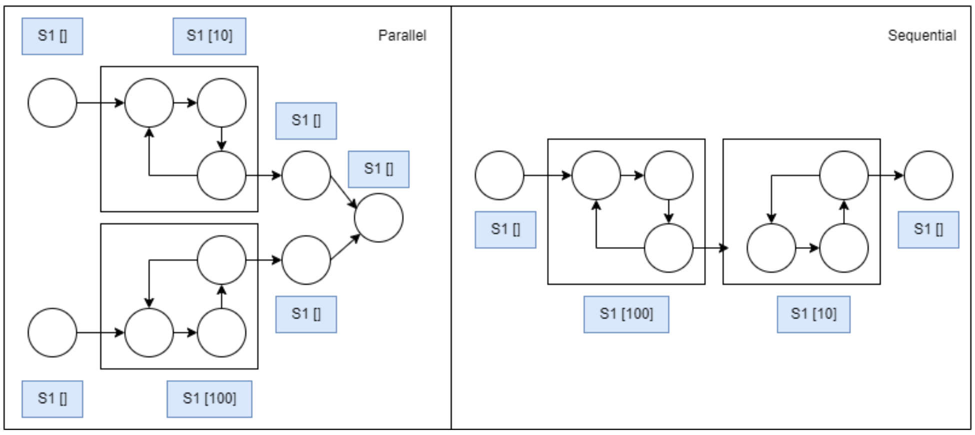

Figure 3.7: Parallel and sequential loops

In the figure 3.7, two situations are depicted:

- Parallel loops: Suppose we take a snapshot at each iteration. The upper loop has reached iteration 10, while the lower one is at iteration 100. If we do not consider the two loops as independent, upon restarting, the lower loop would restart from iteration 10, losing the subsequent iterations.
- Sequential loops: Until the first loop has completed all its iterations, it will not produce any stream to feed into the second loop. In the figure, the first loop has completed 100 iterations and finished, while the second loop is at iteration 10. When we restart, we want to return to the same situation and avoid restarting both loops from iteration 10.

For this reason, the parameter "**iteration index**" was introduced within the SnapshotId. This parameter is used to identify iteration loops within the dataflow. Two parallel or sequential loops have different indexes, while two nested loops have the same index, as the inner loop belongs to the outer loop.

The properties of the SnapshotId remain valid, and the **ordering and distance** between two SnapshotIds change as follows:

- If the iteration index of Sa is the same as that of Sb, use the rules stated previously.
- If it is different, then the iteration stack is not considered for comparison, and Sa and Sb can never be consecutive. In other words, an operator can never save two SnapshotIds with different iteration indexes.

Examples (the iteration index is enclosed in round brackets):

S1 [] () < S2 [] ()	same iteration index (none), what was previously stated holds
S1 [3] (1) = S1 [4] (2)	different iteration index, just consider the main index which is the same
S1 [1] (1) < S2 [1] (2)	different iteration index, just consider the main index which is different

Table 3.5: Order of SnapshotId with iteration index.

S1 [] () → S2 [] ()	same iteration index (none) and index incremented by 1
S1 [0] (1) → S1 [1] (1)	same iteration index and stack level one incremented by 1
S1 [0] (1) !→ S1 [1] (2)	different iteration index

Table 3.6: Distance of SnapshotId with iteration index.

The iteration index is set when the SnapshotId enters and exits the loop, similar to what happens with the iteration stack. When a snapshot token enters a loop, it is set with the specific iteration index of that loop, and when it exits the loop, it is reset to the value it had before entering the loop.

A fundamental requirement of this algorithm is that all replicas of the iterative operator must receive the same snapshot tokens as input. In other words, all replicas must save their state with the same SnapshotId before completing the first iteration. Otherwise, there is an **alignment problem** that invalidates the algorithm.

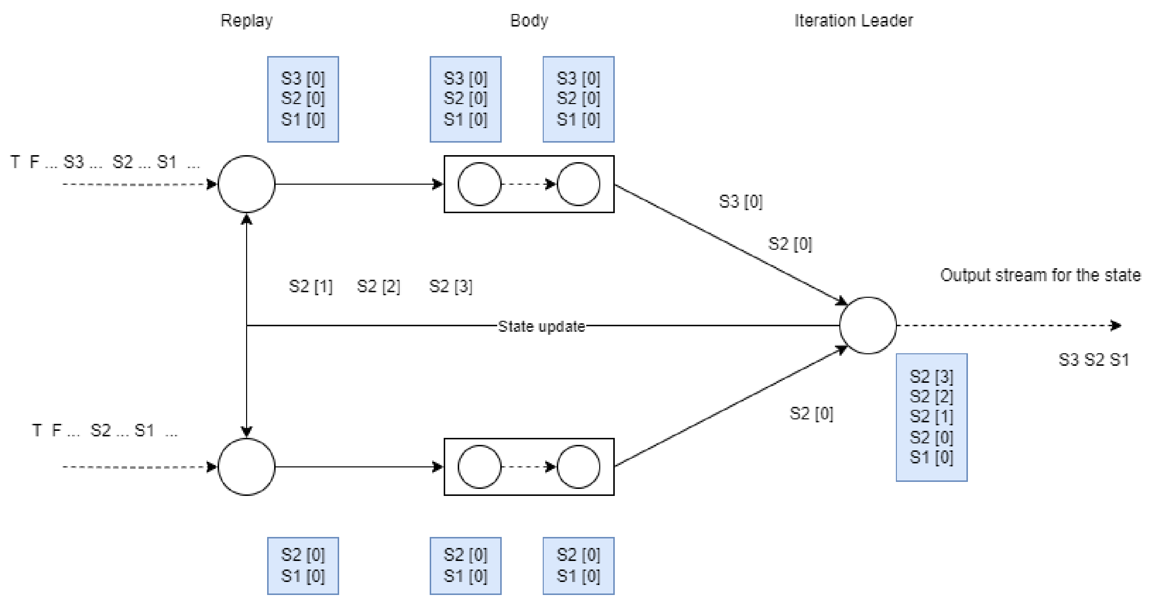

Figure 3.8: Alignment problem

The figure 3.8 represents the alignment problem for the Replay operator in the case of two replicas. Snapshot tokens S1, S2, and S3 arrive from the input stream to the upper replica, while the lower replica only receives tokens S1 and S2. During the first iteration, the tokens pass through Replay and the operators in the loop before reaching the IterationLeader. Note that inside the loop, a position with index 0 has been added in the iteration stack of the SnapshotIds. When the IterationLeader receives the token with SnapshotId S3 [0], it begins the procedure to take a snapshot of S3 [0] and forwards S3 to the output stream operators. However, it cannot complete this snapshot at the end of the first iteration. This means that from the second iteration onwards, the IterationLeader can generate snapshot tokens starting from the SnapshotId S2 [1] and continuing with S2 [2], S2 [3], and so on. When the lower replica of the Replay receives the token S2 [1] from the IterationLeader, it saves the state normally and proceeds. However, when S2 [1] arrives at the upper replica, it cannot save the state because it would violate the monotonicity property of snapshots.

Two solutions have been implemented to solve this problem, which differ in the constraints they require and the contexts in which they can be applied.

The first solution is to block the generation of snapshot tokens from the Sources and perform a single snapshot with index 1 before each Source emits the FlushAndRestart token. In this way, we can ensure that during the first iteration, only the token S1 arrives from all replicas. From the second iteration onwards, the IterationLeader produces tokens like S1[1], S1[2], S1[3], and once the processing is complete, all operators from both internal and external streams to the cycle will perform the Terminate Snapshot S2.

With this solution, the snapshot frequency set through configuration only affects the snapshots generated inside the loop. The external operators before the loop take a snapshot only after consuming the entire stream, while the operators following the loop take a snapshot before receiving the results from the loop with the S1 snapshot and before ending the processing with the Terminate Snapshot.

The second available solution is to insert a block for aligning snapshots between the input stream and the loop. Starting from the input stream, the following operators are added:

- End: with a GroupByReplica strategy, specifically implemented for this context.
- Start: this operator allows snapshot alignment, as all replicas of this Start will forward exactly the same SnapshotIds to the subsequent operators.
- End: with an OnlyOne strategy, to close the block in such a way that it can consume the entire stream and save the special Terminate snapshot, as the iterative operator consumes the stream only until the FlushAndRestart.
- Start: to receive the input stream and forward it to the iterative operator.

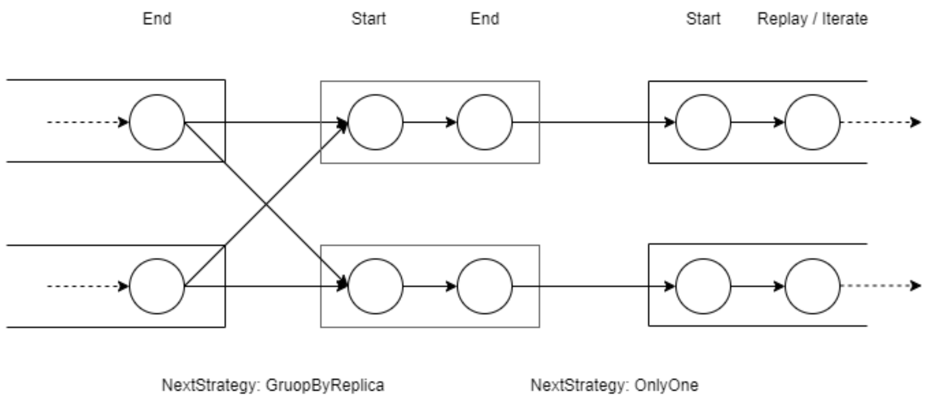

Figure 3.9: Alignment block

The GroupByReplica strategy works similarly to the GroupBy strategy, with the difference that instead of selecting the receiver replica based on the tuple, the End operator forwards all tuples to the replica with the same ID. In this way, the tuples remain on the same host, preserving data locality and reducing the performance impact thanks to in-memory communication channels that avoid the overhead of serialization/deserialization and the network layer.

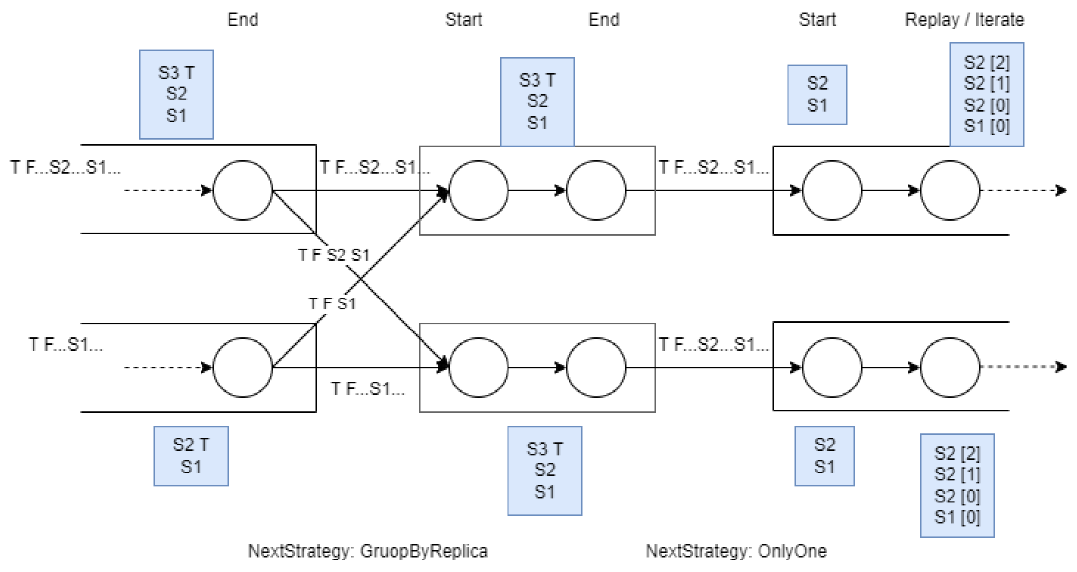

Figure 3.10: Alignment block behavior

The figure 3.10 illustrates how the alignment block allows the same SnapshotIds to reach the loop for each replica.

The first solution, compared to the alignment block, is more efficient because it avoids adding additional operators and blocks. However, it prevents taking snapshots during the first iteration, except for the snapshot before the FlushAndRestart. If the operator graph includes a series of operators that apply many complex functions to the stream before passing it to the loop, then it is preferable to use the second solution, which allows you to take snapshots during that pre-iteration processing phase. In all other cases, the first solution is preferable since, if you do not apply many complex operators and functions before the loop, the time required to obtain the entire stream during the first iteration can become negligible compared to the time spent completing all the other iterations.

To choose which of the two solutions to use, you can use the configuration parameter "iterations snapshot alignment." When set to "true," it selects the first solution, and when set to "false," it implements the second solution with the alignment block.

In the case of loops on parallel streams, it is not advisable to use the alignment block. If the two loops receive SnapshotIds with different indices before starting the iterations, it means that in case of recovery from an intermediate state, one of the two loops must redo all the iterations. For example, if loop1 reaches the snapshot S5 [100], and loop2 reaches the snapshot S6 [100], in case of recovery, you start from S5, and loop2 must redo all the iterations from the beginning.

Another critical situation is related to side inputs, which occur when there is a Join within the body of an iteration with an external stream, referred to as a "side stream" or "side input." During the first iteration, the side stream is read entirely and stored in a cache. From the second iteration onwards, the side stream is read from the cache. Here too, there is the issue of snapshot alignment, especially when the side-stream processes more records than the main-stream.

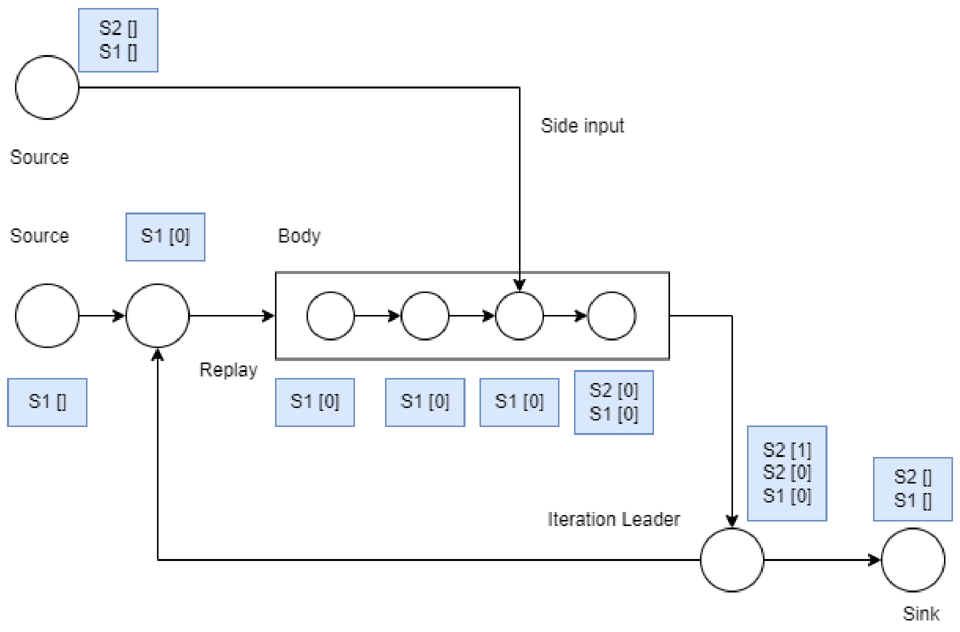

Figure 3.11: Alignment problem with side inputs

The figure 3.11 illustrates a potential critical scenario where the main input produces only S1[], while the Source of the side input generates S1[] and S2[]. The operators located after the Start operator that receives the side input also take a snapshot of S2[0], but this token has never passed through Replay and the first operators in the loop body. When the first iteration is completed, the IterationLeader generates token S2[1] and sends it to Replay, which, however, has S1[0] as the last saved snapshot. In this situation, there are only two possibilities, both of which are incorrect:

1. Saving S2[1], but this violates the regularity property of snapshots and generates an error.
2. Saving S2[0] first and then S2[1], but in this case, S2[0] would be saved during the first iteration for some operators and during the second iteration for others, generating inconsistency.

Therefore, it is essential that both the main-stream and the side-stream produce the same

SnapshotIds. To achieve this, the first solution must be used. If the "iterations snapshot alignment" configuration parameter is set to "false," the program will panic.

3.3.2. Replay

The state of the Replay operator contains two main components:

- The state of the loop taken at the iteration when the snapshot is performed.
- A queue containing the stream that is provided to the body at each iteration.

The snapshot procedure slightly differs depending on the channel from which the snapshot token arrives. When a snapshot token arrives from the input stream (input channel), which is from outside the loop, the procedure is as follows:

1. Modify the SnapshotId contained in the token by adding a level to the iteration stack and setting the iteration index.
2. Copy and persist the state.
3. Forward the token to the body of the loop.

On the other hand, when the snapshot token arrives from the IterationLeader (state update channel):

1. Copy and persist the state.
2. Forward the token to the body of the loop.

Despite this operator having two input channels, they work in mutual exclusion. This means that snapshot tokens generated outside the loop (those with a 0 at the corresponding level of the iteration stack) only arrive from the input channel, while tokens generated inside the loop only arrive from the state update channel.

3.3.3. Iterate

The state of the Iterate operator contains three main components:

- The state of the loop taken at the iteration when the snapshot is performed.
- A queue with the stream sent to the body.
- A queue with the stream received from the operators within the loop (feedback input).

The snapshot procedure for Iterate is a bit more complex than Replay because this operator has a third input channel (feedback channel) where the new stream generated by the body is sent, and it also has a second output channel (output stream channel). When a snapshot token arrives from the input stream (input channel), which is from outside the loop, the procedure is as follows:

1. Modify the SnapshotId contained in the token by adding a level to the iteration stack and setting the iteration index.
2. Copy the state.
3. Forward the token to the body of the loop.
4. Forward the original token (with the unmodified SnapshotId) to the output stream channel.
5. When the same snapshot token arrives on the feedback channel, modify the copy of the state taken earlier by adding all the messages received on the feedback channel before the snapshot token. After that, persist the state.

On the other hand, when the snapshot token arrives from the IterationLeader (state update channel):

1. Copy and persist the state.
2. Forward the token to the body of the loop.

It is not necessary to wait for the token to arrive from the feedback channel because the snapshot token is the first message sent and received on the feedback channel (even in the case of side input, the first processed message always comes from the non-cached stream). Additionally, these snapshot tokens are not sent to the output stream channel because they remain within the loop without ever leaving it.

3.3.4. IterationLeader

The IterationLeader contains the following elements in its state:

- The state of the loop taken at the iteration in which the snapshot is executed.
- Other parameters, such as the number of delta updates yet to be received.

The IterationLeader has two output channels: the "state update channel" for sending data to the iterative operator and the "state output channel" for sending data to external operators following the loop.

During the first iteration of the loop, when a new snapshot token arrives, the IterationLeader executes the following procedure:

1. It saves the last received SnapshotId.
2. It persists its state.
3. It modifies the SnapshotId by removing the last level from the iteration stack, and if it is not in a nested loop, it also removes the iteration index.
4. It sends the token with the modified SnapshotId to the state output channel.

Once the first iteration is completed, the SnapshotGenerator is initiated with the last received SnapshotId. Each time an iteration finishes and the new state is computed, it checks whether the conditions for taking a snapshot are met. If affirmative, the IterationLeader:

1. Persists the state.
2. Adds the new SnapshotId to the "state update" message sent via the "state update channel."

Again, this is an internal snapshot within the loop and is not sent to the "state output channel."

3.4. Recovery

The recovery procedure is executed if specified in the persistency configuration. The "try restart" flag indicates whether you want to perform a recovery or start from scratch, while the "restart from" parameter allows you to specify the index of the snapshot from which you want to restart. If not specified, you will start from the most recent complete snapshot. There are some edge cases to consider, such as:

- The snapshot specified in "restart from" is not complete or does not exist at all: In this case, you will start from the latest complete snapshot.
- There are no complete snapshots saved: In this case, you will start from scratch as in normal execution.

The first phase of the procedure occurs once the stream is defined and when the job graph is complete. From the job graph, a list is derived containing the coordinates of all stateful operators in the network. This list is passed to the "find_snapshot()" function, which determines the snapshot to restart from. This function first calculates the most recent complete snapshot and then, if "restart from" is set, selects the minimum between the calculated snapshot and "restart from."

Given the properties of monotonicity and regularity of SnapshotIds, we can be sure that this function always returns a valid SnapshotId. The function for calculating the latest valid snapshot looks for the last saved snapshot for each coordinate and then takes the minimum among all the indexes. Here again, the monotonicity and regularity of the SnapshotIds guarantee a valid result. This function has no impact on the saved snapshots because it only performs read operations. It is important that it remains this way because this procedure is executed in parallel on all hosts, and any modification could lead to invalid results and errors.

If there are loops in the operator graph, it is necessary to determine the iteration stack to restart from for each of them. Once the index of the SnapshotId to restart is determined, a specific function is called to find the iteration stack for each group of operators with different iteration indexes. This function generates a HashMap with the iteration index as the key and the corresponding iteration stack as the value. As a result, each operator restarts from a SnapshotId determined by its specific iteration index.

The index of the snapshot to restart from and the iteration stacks are saved within the PersistencyBuilder structure. After that, the normal process continues with deploying the network by instantiating and starting the various blocks. In the metadata passed to the setup() function of each operator, there is a reference to the PersistencyBuilder. During setup, each operator generates the PersistencyService structure through the Persistency-Builder, which provides the index of the SnapshotId to restart from and the HashMap of iteration stacks to restart from.

This leads to the second phase, in which each operator attempts to restart from the calculated snapshot. This function first determines the correct SnapshotId to restart from by combining the index and the corresponding iteration stack. After that, it deletes all persisted snapshots with a SnapshotId greater than the one from which we want to restart. This is done to avoid conflicts when the operator saves new snapshots because, after recovery, the sequence of SnapshotIds will continue from the one from which we restarted. Once the snapshots are deleted, the function tries to retrieve the state corresponding to the restart SnapshotId and returns it to the operator. The operator sets its internal state to the saved state and is ready to resume processing.

3.5. Persistency Module

This module contains all the needed structures and functions to provide state persistence.

3.5.1. Configuration

PersistencyConfig is a new piece of information added to Noir configuration. It contains all the parameters required to define the settings for executing snapshots and their persistence. When not explicitly specified, the program runs without taking any snapshots.

```
pub struct PersistencyConfig {
    pub server_addr: String,
    pub try_restart: bool,
    pub clean_on_exit: bool,
    pub restart_from: Option<u64>,
    pub snapshot_frequency_by_item: Option<u64>,
    pub snapshot_frequency_by_time: Option<Duration>,
    pub iterations_snapshot_alignment: bool,
}
```

Listing 3.2: PersistencyConfig

Each field has a specific role:

- **server_addr**: Specifies the Redis URL to connect to, including the IP address, port, and password.
- **try_restart**: When set to true, it indicates a desire to restart from a saved state. If set to false, the execution starts from scratch.
- **clean_on_exit**: This parameter is used to clean the datastore once the execution is completed.
- **restart_from**: Indicates the snapshot to restart from. The program resumes from the saved state only if it is a valid and complete snapshot.
- **snapshot_frequency_by_item**: Specifies the frequency of generating snapshots based on the number of tuples.
- **snapshot_frequency_by_time**: Similar to the previous parameter, but time-based. It uses a Duration type to specify the minimum time between the generation of two consecutive snapshot tokens.

- **iteration_snapshot_alignment**: Determines which solution to implement for snapshot alignment in case of iteration. If set to true, all Sources do not generate snapshot tokens except a single snapshot before FlushAndRestart.

As mentioned earlier, snapshot generation occurs asynchronously and independently for each replica of Source operators (and potentially for the IterationLeader). Therefore, when setting the frequency based on the number of tuples, the following scenarios can happen:

- The count is performed independently for each Source. For example, if you have a file with 1000 rows and four Sources reading it, and you choose a snapshot frequency of 50 elements, each Source will generate 5 snapshot tokens since each Source reads 250 rows.
- In cases where one replica, say R1, of a Source reads faster than another replica, say R2, there can be some time between the generation of snapshot tokens by R1 and R2. This time gap can lead to a significant increase in the size of the state for subsequent Start operators that receive tuples from both replicas. This issue is discussed further in the Performance chapter.
- For IterationLeader, the counting is not based on tuples but on the iterations performed.

For these reasons, generating snapshots based on the number of elements is primarily useful for testing and debugging.

Generating snapshots based on time allows for a more aligned generation of snapshot tokens. When the time interval set by the frequency expires, a snapshot token is generated. It is important to note that the pull-based architecture of operators does not allow for perfect alignment, and the token is generated and forwarded to the subsequent operators only when a new tuple is requested to the Source. With time-based generation, in cases where Sources read at different speeds, the number of tuples between two snapshot tokens can vary because the faster Source can produce more tuples compared to the slower one.

3.5.2. PersistencyBuilder

This structure is instantiated within the Scheduler because it performs tasks both before and after processing. This implies that there is an instance of PersistencyBuilder for each host in the cluster.

This structure is responsible for the following tasks:

- Determining the snapshot from which to resume.
- Managing the StateSaver.
- Generating a PersistencyService for each operator.
- Cleaning the storage of all saved snapshots.

The “function find_last_snapshot()” is responsible for determining the snapshot from which to resume. This function is executed before instantiating the blocks with the operators to ensure that no operator can modify the saved snapshots (by saving a new state or deleting a saved state). To determine the last complete snapshot, it is necessary to provide the function with a list of coordinates for all stateful operators in the network, as each state takes snapshots independently. This is crucial because it is the only point in the code where the global snapshot is considered rather than individual operator snapshots. Coordinates are calculated in advance while constructing the stream, and this is a delicate operation. Omitting a coordinate can lead to uncertainty regarding whether the snapshot calculated by the function is indeed the last valid snapshot. On the other hand, if an extra coordinate is added, the function will not find any valid global snapshots.

The generation of PersistencyService for each operator occurs during the setup of each operator. A field has been added to the metadata passed to the setup function, containing a reference to PersistencyBuilder. During the generation, the information needed to retrieve the SnapshotId indicating the state from which to resume is passed to each operator.

If specified in the configuration, PersistencyBuilder deletes every saved state after processing has concluded. For this operation, specifying the list of coordinates from which to delete every saved state is also necessary. However, given that this type of operation modifies the datastore and can be performed in parallel by the instances of PersistencyBuilder present on each host, a reduced list containing only the coordinates of the operators that have been instantiated on that specific host is passed to the function. This way, each instance of PersistencyBuilder, and thus each host, deletes only the snapshots related to the operators instantiated on the same host.

The following table provides an overview of the functions offered by PersistencyBuilder categorized by the Noir execution phase:

Stream definition	**Start execution**	**Operators setup**	**Operators processing**	**End execution**
Compute a list of coordinates of stateful operators	Find the snapshot from which to restart	Generate PersistencyService for each operator		Stop the StateSaver
Create the StateSaver				If needed, clear the datastore

Table 3.7: PersistencyBuilder functions

3.5.3. PersistencyService

This structure provides operators with all the necessary functions for the snapshot and recovery procedures. In particular, the main functions are:

- State saving
- State recovery
- Resuming from a snapshot

For state saving, it is necessary to provide the operator coordinates, the SnapshotId, and the state. There is a specific function for saving a Terminate Snapshot that does not require providing the SnapshotId but calculates it automatically from the last saved SnapshotId. This function serializes the state and sends it to the dedicated StateSaver thread, which is responsible for persisting the snapshots to Redis.

State recovery is performed by specifying the operator coordinates and SnapshotId. This function is blocking and used only during the setup for the recovery procedure.

The "restart_from_snapshot()" function is used to retrieve the actual SnapshotId from which to resume, as explained in the Recovery section.

3.5.4. StateSaver

The purpose of this structure is to manage a worker thread for the asynchronous saving of operator states during snapshots. Each host has its own StateSaver, created and managed by the PersistencyBuilder. The worker thread has an in-memory channel through which it receives states from various operators. When all operators finish processing, a termination message is sent to the StateSaver, which completes processing the messages in the channel and returns. This approach allows the operator to avoid halting computation until the snapshot is saved in Redis. Instead, the operator simply serializes the state, sends it to the StateSaver, and can then continue processing.

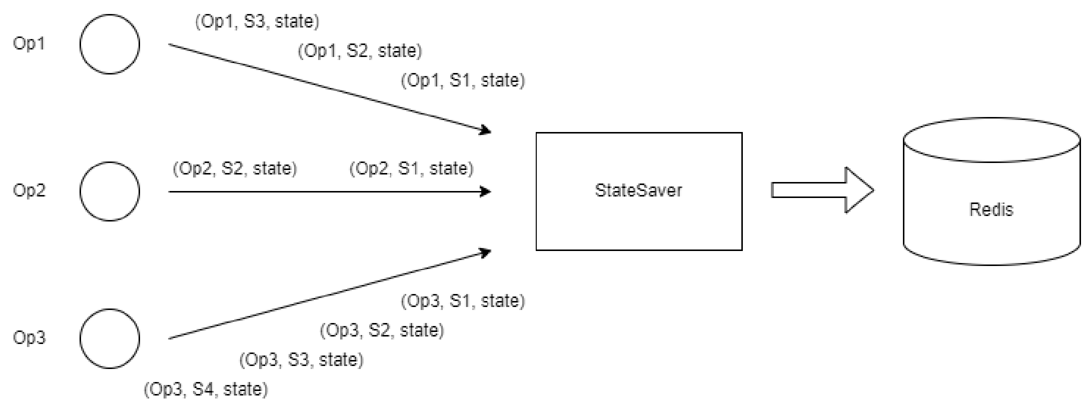

Figure 3.12: StateSaver

The figure 3.12 illustrates the process of saving the state using StateSaver. The operator sends its coordinates, the SnapshotId, and the serialized state in the message. With this information, StateSaver can save the data in Redis, as explained in the subsequent section. The order in which messages arrive at StateSaver is not relevant when they come from different operators. In such cases, it does not matter whether the state of Op1 is saved before that of Op2, and vice versa. However, it is imperative that the order of messages sent by the same operator remains unchanged; otherwise, it would violate the properties of SnapshotId monotonicity and regularity, leading to an error. The in-memory channel used is multiple-producers single-consumer, so it provides this guarantee. This means that if the operator sends messages with the correct SnapshotIds, we can be sure that the order of those messages is maintained.

3.6. Redis

Redis[5] (REmote DIctionary Server) is a key-value store that can be used as a database, cache, or message broker. It offers various data structures such as strings, lists, and sets. Redis manages data in memory, making it highly performing and fast. Simultaneously, it ensures reliability and persistence through periodic data saving to disk. These features make it an excellent choice as a datastore for high-throughput, low-latency applications, as seen in the case of Noir.

The Redis-rs[6] crate integrates Redis into Rust, providing a rich and comprehensive set of features and APIs. The r2d2 feature enables the creation of pooled connections that optimize and speed up connections to Redis. By using pooled connections, you can avoid creating a new connection for each operation. Once a new connection is created and used, it remains active and can be reused for any future operations.

The RedisHandler structure is responsible for interfacing with Redis and providing Noir with all the necessary functions to implement the snapshot algorithm. This structure requires a configuration string that specifies the address, port, and password, which is set through the "server addr" field of PersistencyConfig.

RedisHandler offers the following functions:

- State save: Given the operator's coordinate, SnapshotId, and serialized state, it saves the state.
- Find the last saved SnapshotId: Given the operator coordinates, it finds the last saved SnapshotId.
- Find the last iteration stack corresponding to the last saved SnapshotId, given the SnapshotId index.
- Find the saved state: Given the operator coordinate and SnapshotId, it retrieves the saved state.
- State delete: Given the operator coordinate and SnapshotId, it deletes the saved state.

Every time an operator saves its state in Redis, the following operations are performed:

1. The serialized state is saved with a key that is a concatenation of the operator coordinates and the SnapshotId.
2. The SnapshotId is appended to a list with the key being the operator coordinates.
3. If the SnapshotId has a non-empty iteration stack, it is also appended to another list with the key being the operator coordinates and the SnapshotId index.

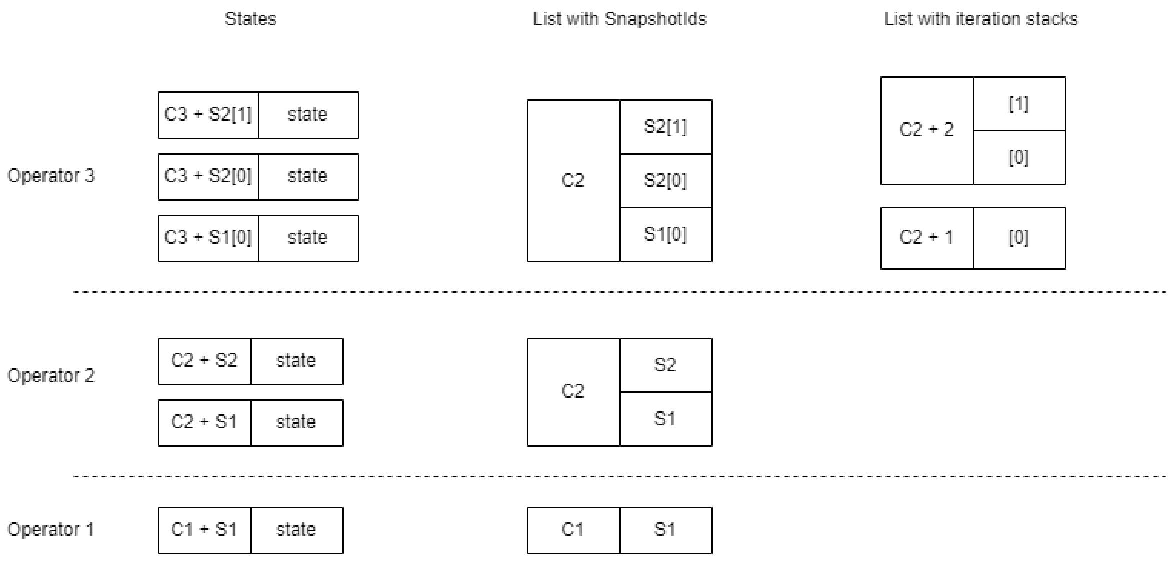

Figure 3.13: Redis data structure

The figure 3.13 represents the data saved in Redis as a result of the snapshots taken by various operators:

- Operator 1 has taken a single snapshot with SnapshotId S1, and the key under which the state is saved is C1 + S1, which is the concatenation of the operator coordinate and the SnapshotId. The list of SnapshotIds contains only S1.
- Operator 2 has saved two states, and the keys used for them differ based on the SnapshotId. The list contains two SnapshotIds: S1 and S2. For both Operator 1 and Operator 2, the SnapshotIds did not include an iteration stack, so the second list was not used.
- Operator 3 has taken three snapshots: S1[0], S2[0], and S2[1]. In this case, the list for iteration stacks is used. Specifically, a list is created for each different index of SnapshotIds.

The list of SnapshotIds allows you to retrieve the last SnapshotId taken without having to scan the entire list, as it is always found in the last position. The list of iteration stacks allows you to immediately retrieve the last iteration stack corresponding to the index of the SnapshotId. Thanks to the properties of SnapshotIds, this allows us to determine with certainty what the last saved SnapshotId is and whether an operator has saved a state with a specific SnapshotId.

The functions for finding the last SnapshotId and the last iteration stack respectively read the last positions of the two lists. The function for finding the state returns the state with the key formed by the operator coordinate and the specified SnapshotId. The function that deletes a specific snapshot not only removes the state but also deletes the corresponding entries in the two lists.

4 | Performance evaluation

One of Noir's key strengths lies in its excellent performance. Therefore, we conducted some experiments to study and to try to quantify the impact of the snapshot algorithm on execution times. The impact of the snapshot algorithm on execution times is influenced by several factors:

- The operators in the job graph: stateless operators add minimal overhead, while stateful operators vary in terms of procedures and states.
- The execution graph: the number of operator replicas affects the state of the Start operator and the complexity of its snapshot procedure.
- Snapshot frequency: a higher frequency results in more snapshots, which, in turn, increases execution time.
- The size of the states of the operators.
- The speed of Redis requests.

The following sections present some of the experiments conducted along with an analysis of the obtained results. The benchmarks used for these experiments are Nexmark[22] and Wordcount.

Nexmark is a set of queries commonly used to evaluate the performance of stream processing systems. These queries are based on an auction system modeled with three entities: person, auction, and bid. There are eight queries that require various analyses and operations, including filtering, joining, and windowing. Below there are the operator graphs for each of the 8 queries.

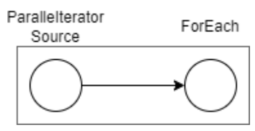

Figure 4.1: Nexmark query 0 job graph

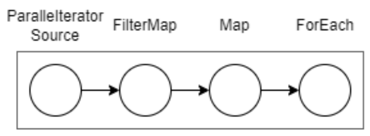

Figure 4.2: Nexmark query 1 job graph

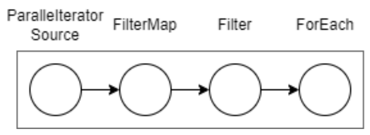

Figure 4.3: Nexmark query 2 job graph

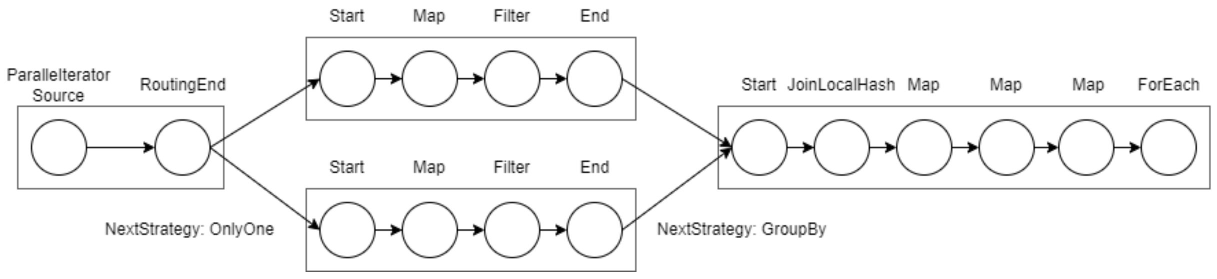

Figure 4.4: Nexmark query 3 job graph

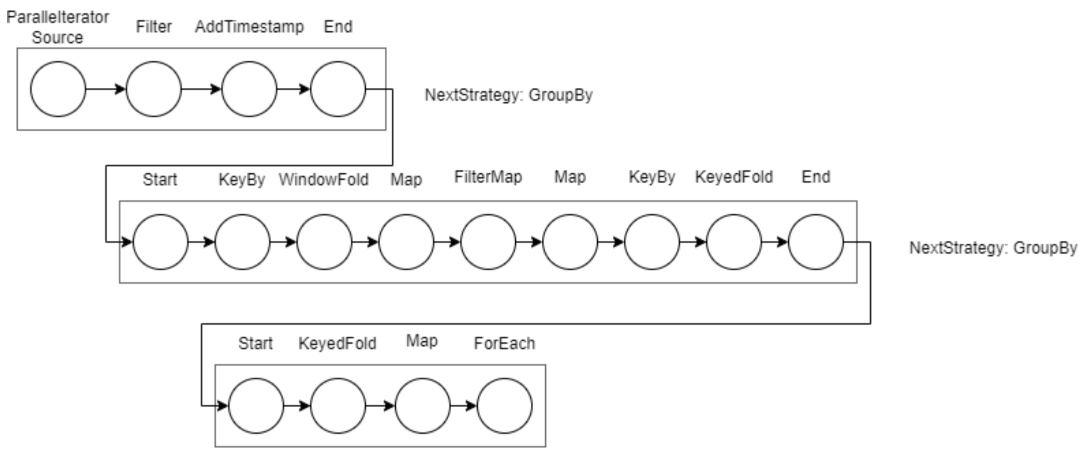

Figure 4.5: Nexmark query 4 job graph

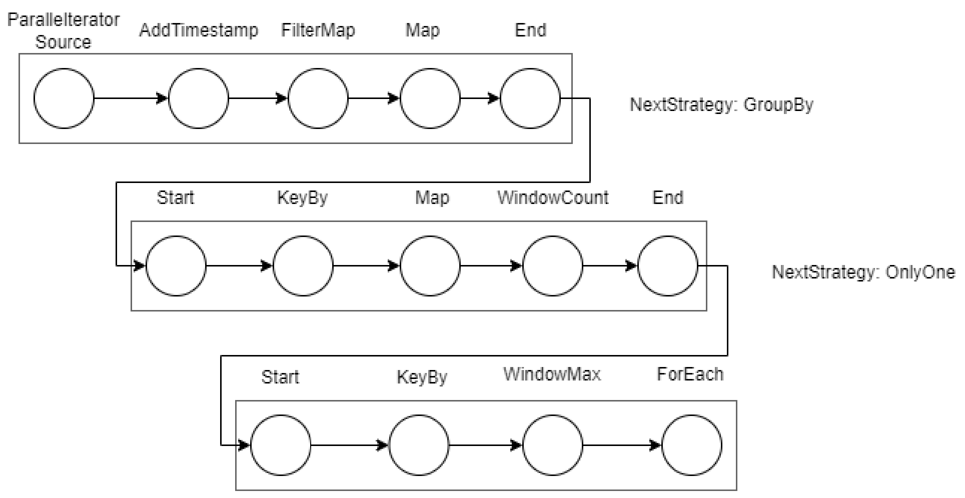

Figure 4.6: Nexmark query 5 job graph

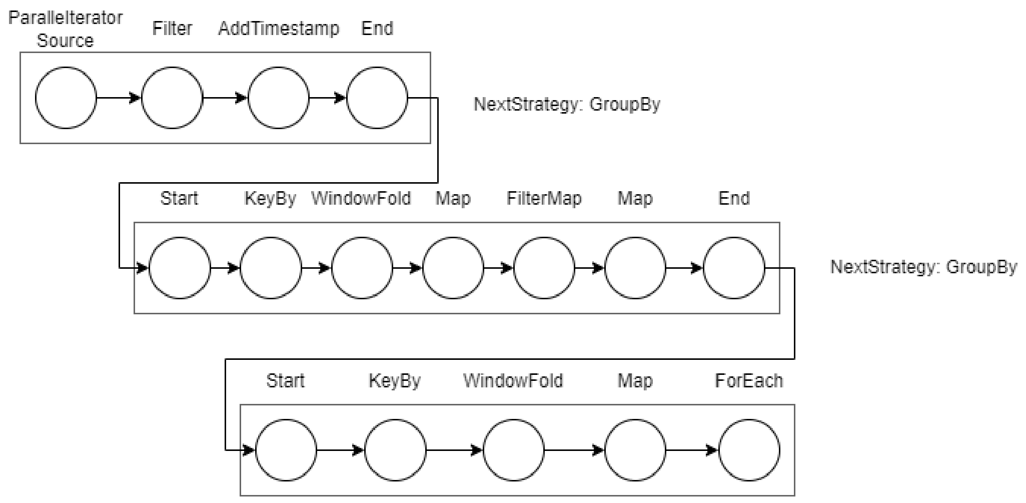

Figure 4.7: Nexmark query 6 job graph

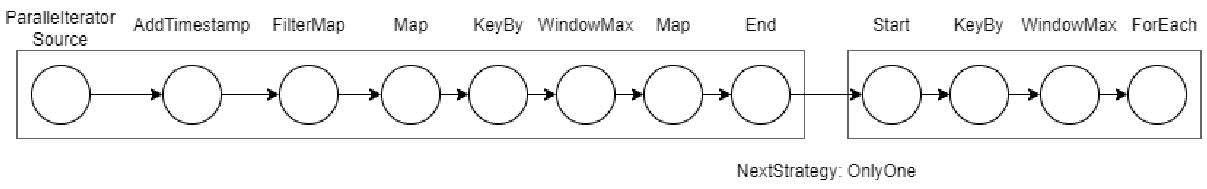

Figure 4.8: Nexmark query 7 job graph

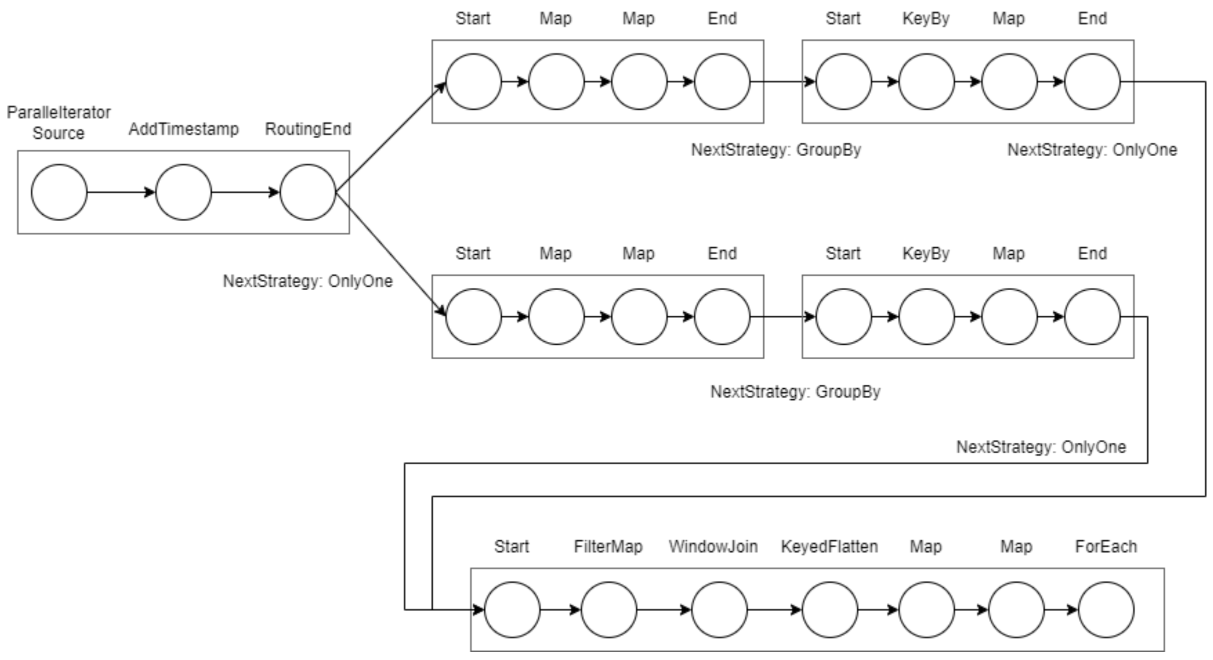

Figure 4.9: Nexmark query 8 job graph

Wordcount is a benchmark that aims to calculate the word occurrences in a text document. There are many possible implementations in Noir for this problem, but for these experiments, three, in particular, have been chosen:

- wc-fold: each word is mapped with a single occurrence, and these occurrences are sent to the Fold operator, which sums them for each word.
- wc-fold-assoc: this version uses associative folding to avoid a network shuffle.
- wc-fast: also using associative folding, but the local folds process an entire line of the file.

Below there are the operator graphs:

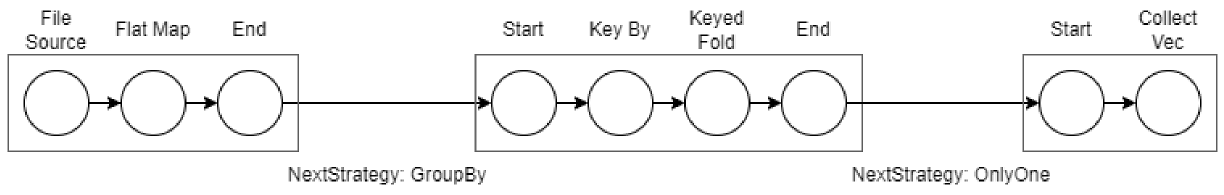

Figure 4.10: Wordcount fold version

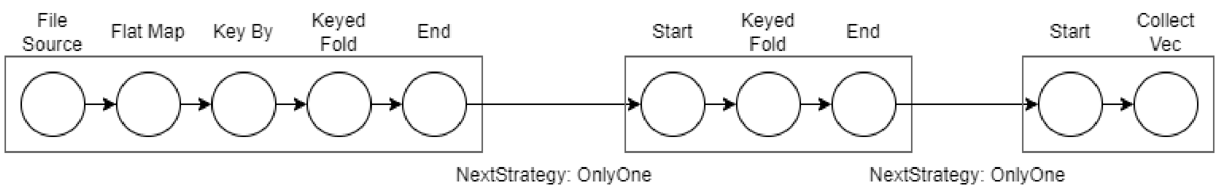

Figure 4.11: Wordcount associative fold version

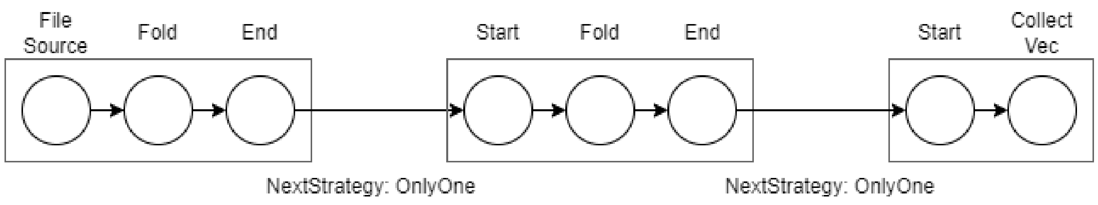

Figure 4.12: Wordcount fast version

All benchmarks have been run on remote server sola1.dei.polimi.it:

Operating System	Debian GNU/Linux 11 (bullseye)
Kernel	Linux 5.10.0-23-amd64
Architecture	x86-64
CPU	Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
CPU Cores	12
CPU Threads	24
RAM	64 GB

Table 4.1: Remote server specifications

The Redis server runs locally with this configuration:

```
io-threads 4
appendonly no
save
```

Listing 4.1: RedisConf

Using the Criterion[1] crate, statistics on multiple consecutive runs are obtained to analyze the average time, throughput, distribution, outliers, and other useful metrics. In addition to these insights, the number of taken snapshots and the total size of all states saved in Redis are recorded. The number of taken snapshots corresponds to the maximum number of snapshots taken by any operator within the network, so it is possible that some operators took fewer snapshots with respect to the retrieved number. The total size of the states is calculated as the sizes of all the data saved in Redis during the execution.

4.1. Comparison with original Noir

The comparison with the original version of Noir, without state persistence, is performed by comparing the results of the Nexmark and Wordcount benchmarks between the original version of Noir and this version with unconfigured persistence (disabled). In the absence of configured persistence, the following steps are skipped:

- Creation of the PersistencyBuilder structure and, consequently, the creation and start of StateSaver (meaning no connection to Redis).
- Computation of the list with the coordinates of stateful operators.
- Creation of PersistencyService for each stateful operator during the setup phase.
- No state saving.

The first experiment focused on the Nexmark benchmark, which was executed with maximum available parallelism (12) and an input stream of 100'000 tuples, and later with 1'000'000 tuples.

nexmark query	**original**	**persist None**	compare
nx 0	6,76 (± 0,08) ms	7,04 (± 0,01) ms	4%
nx 1	6,94 (± 0,01) ms	7,08 (± 0,01) ms	2%
nx 2	6,99 (± 0,01) ms	7,11 (± 0,01) ms	2%
nx 3	8,90 (± 0,02) ms	9,28 (± 0,01) ms	4%
nx 4	19,3 (± 0,1) ms	19,9 (± 0,1) ms	3%
nx 5	7,99 (± 0,02) ms	8,58 (± 0,01) ms	7%
nx 6	19,5 (± 0,1) ms	20,2 (± 0,1) ms	4%
nx 7	7,18 (± 0,01) ms	7,69 (± 0,01) ms	7%
nx 8	9,72 (± 0,01) ms	9,99 (± 0,01) ms	3%

Table 4.2: Nexmark 100K original vs persistence disabled

nexmark query	**original**	**persist None**	compare
nx 0	65,7 (± 0,9) ms	68,3 (± 0,1) ms	4%
nx 1	67,4 (± 0,1) ms	68,5 (± 0,1) ms	2%
nx 2	67,6 (± 0,1) ms	68,9 (± 0,1) ms	2%
nx 3	82,6 (± 0,1) ms	86,6 (± 0,1) ms	5%
nx 4	244 (± 3) ms	246 (± 2) ms	1%
nx 5	76,8 (± 0,1) ms	86,5 (± 0,1) ms	13%
nx 6	252 (± 3) ms	255 (± 2) ms	1%
nx 7	69,2 (± 0,1) ms	73,8 (± 0,1) ms	7%
nx 8	85,8 (± 0,1) ms	89,5 (± 0,1) ms	4%

Table 4.3: Nexmark 1M original vs persistence disabled

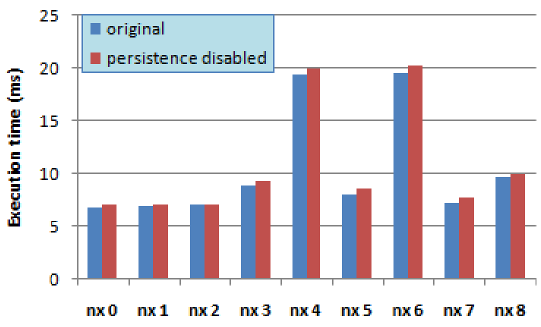

Figure 4.13: Nexmark 100K original vs persistence disabled

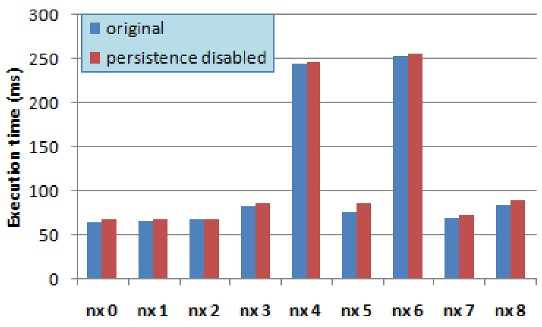

Figure 4.14: Nexmark 1M original vs persistence disabled

If we exclude nx-5 with an input stream of 1,000,000 tuples, for all the others, the overhead remains below 10%.

Wordcount was also executed with the maximum available parallelism (12) using a text document with 100,000 lines and then with 1,000,000 lines.

wordcount	**original**	**persist None**	compare
wc-fast	40,1 (± 0,1) ms	49,7 (± 0,1) ms	24%
wc-fold-assoc	183 (± 1) ms	185 (± 1) ms	1%
wc-fold	221 (± 1) ms	232 (± 1) ms	5%

Table 4.4: Wordcount 100K original vs persistence disabled

wordcount	**original**	**persist None**	compare
wc-fast	397 (± 1) ms	492 (± 1) ms	24%
wc-fold-assoc	1,83 (± 0,01) s	1,83 (± 0,01) s	0%
wc-fold	2,17 (± 0,01) s	2,26 (± 0,01) s	4%

Table 4.5: Wordcount 1M original vs persistence disabled

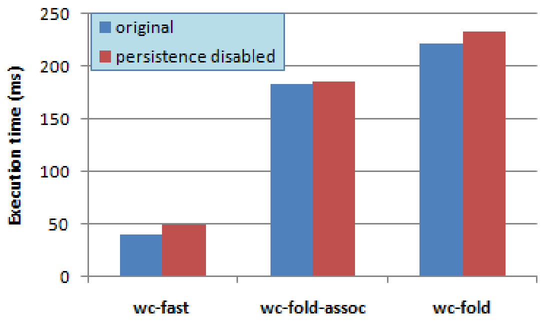

Figure 4.15: Wordcount 100K original vs persistence disabled

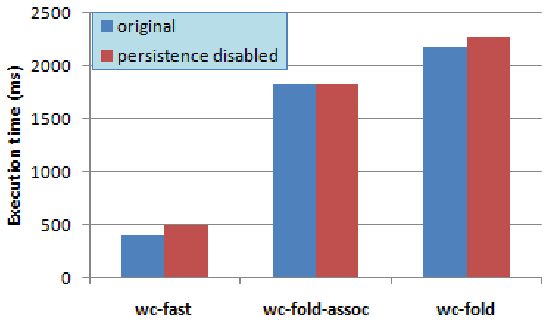

Figure 4.16: Wordcount 1M original vs persistence disabled

The overhead of wc-fold and wc-fold-assoc is in line with the results of Nexmark, while the 24% overhead of wc-fast is hard to explain, especially when compared to an execution with persistence enabled in which multiple snapshots are taken (see the tables in the next section).

4.2. Snapshot frequency by item vs by time

As explained in the previous chapters, snapshot frequency should ideally be set based on time because the one using tuple counting is primarily used for testing and debugging as it does not allow for generating aligned snapshot tokens. However, it can be interesting to quantify the difference in terms of execution time that is achieved by using time-based frequency versus item-based frequency.

	wc-fast	wc-fold-assoc	wc-fold
snapshot frequency	4 ms	18 ms	25 ms
execution time	483 (± 3) ms	1,91 (± 0,02) s	2,67 (± 0,05) s
mean taken snapshots	116	102	106
size of stored data	5,09 MB	3 MB	71 MB

Table 4.6: Wordcount snapshot by time

	wc-fast	wc-fold-assoc	wc-fold
execution time	468 (± 4) ms	1,90 (± 0,01) ms	12,2 (± 0,2) s
mean taken snapshots	101	101	101
size of stored data	5,03 MB	3,03 MB	1,46 GB

Table 4.7: Wordcount snapshot by item

For all experiments, a text file with 1,000,000 rows and maximum available parallelism (12) was used. Different behaviors are observed depending on the benchmark. For wc-fast and wc-fold-assoc, the execution times are nearly the same (according to the number of taken snapshots), while for wc-fold, the execution time changes by an order of magnitude. The same pattern occurs in the size of states stored in Redis. This behavior is due to the arrangement of Start operators within the job graph and the operators that precede them.

In the case of wc-fast and wc-fold-assoc, there are two Start operators, both with the NextStrategy: OnlyOne. The first Start operator receives tuples only from the corresponding replica of the preceding block, so it has only one input. The second Start operator, which is not replicated, receives data from all replicas of the preceding block, meaning it has multiple inputs and maintains a message queue in the state. However, the presence of the Fold operator implies that the stream is fully consumed before the Fold emits tuples with results. This means that the second Start operator will always save a state with an empty message queue until the entire stream has been consumed. Therefore, if non-aligned snapshot tokens flow through the network, their effects are negligible.

In the case of wc-fold, there are also two Start operators, but the first has the NextStrategy: GroupBy and receives all the tuples generated by the Sources. This implies that the Start state always contains a message queue, and the queue size is determined by the alignment of the incoming snapshot tokens. If Sources produce tuples at different rates, the item-based frequency causes the fast Source to generate many snapshot tokens that arrive quickly at the next block, while the slow Source generates fewer tokens that arrive later than those from the fast Source. This creates long message queues, leading to increased execution times and larger states saved, as shown by data.

4.3. Variations of different snapshot frequencies

Below are presented the results of some experiments where the snapshot frequency varies with different values. The Nexmark benchmark was executed with the maximum available parallelism (12), an input stream of 1,000,000 tuples, and a batch size of 1024. Different snapshot frequencies are used for Query 4 and Query 6. We can group Nexmark queries into three categories according to the overhead caused by the snapshot procedure:

- nx-0, nx-1, nx-2, nx-7
- nx-3, nx-5, nx-8
- nx-4, nx-6

For nx-0, nx-1, nx-2, and nx-7, the impact of the snapshot algorithm is low. nx-0, nx-1, and nx-2 have only the ParallelIterator Source operator that persists the state, and its state has a fixed size, consisting only of an integer. nx-7 includes other stateful operators like Window and Start operators. The size of the state of the Window operator is limited by the number of open windows at the time of the snapshot, and the size of the state of the Start operator is fixed since Start receives data only from one replica, keeping the message queue always empty. Table 4.8 presents the obtained results.

For nx-3, nx-5, and nx-8, the variation in snapshot frequency has more pronounced effects, partly due to the presence of Start operators that receive data from multiple replicas. Table 4.9 shows the obtained results.

Similarly, for nx-4, and nx-6, the variation in snapshot frequency has pronounced effects, as indicated by the data in Table 4.10.

nx query	snapshot frequency	execution time	taken snapshots	datastore size
nx 0	T	61,2 (± 1,4) ms	1	3,56 KB
	50 ms	71,0 (± 0,1) ms	2	5,06 KB
	20 ms	71,2 (± 0,1) ms	4	8,25 KB
	10 ms	72,0 (± 0,2) ms	7	13,6 KB
	5 ms	74,3 (± 0,2) ms	14	24,2 KB
nx 1	T	68,3 (± 0,1) ms	1	3,56 KB
	50 ms	71,5 (± 0,3) ms	2	5,06 KB
	20 ms	71,6 (± 0,1) ms	4	8,25 KB
	10 ms	72,3 (± 0,2) ms	7	13,6 KB
	5 ms	74,7 (± 0,2) ms	14	24,2 KB
nx 2	T	69,0 (± 0,1) ms	1	3,56 KB
	50 ms	72,2 (± 0,1) ms	2	5,06 KB
	20 ms	72,3 (± 0,1) ms	4	8,25 KB
	10 ms	74,3 (± 0,3) ms	7	13,6 KB
	5 ms	75,8 (± 0,3) ms	14	24,2 KB
nx 7	T	74,5 (± 0,1) ms	1	11,1 KB
	50 ms	77,2 (± 0,1) ms	2	16,4 KB
	20 ms	78,9 (± 0,3) ms	4	27,5 KB
	10 ms	83,0 (± 0,3) ms	9	50,3 KB
	5 ms	90,4 (± 0,7) ms	18	89,0 KB

Table 4.8: Nexmark q0, q1, q2, q7 various snapshot frequencies

nx query	snapshot frequency	execution time	taken snapshots	datastore size
nx 3	T	83 (± 1) ms	1	20,1 KB
	50 ms	98 (± 3) ms	2	8,27 MB
	20 ms	104 (± 2) ms	5	31,3 MB
	10 ms	125 (± 2) ms	8	51,2 MB
	5 ms	184 (± 3) ms	17	113 MB
nx 5	T	85 (± 1) ms	1	15,3 KB
	50 ms	93 (± 1) ms	2	2,61 MB
	20 ms	103 (± 1) ms	5	10,1 MB
	10 ms	129 (± 1) ms	12	27,1 MB
	5 ms	249 (± 5) ms	(30-40)	86,1 MB
nx 8	T	84 (± 1) ms	1	30,7 KB
	50 ms	88 (± 1) ms	2	2,56 MB
	20 ms	106 (± 1) ms	5	18,2 MB
	10 ms	127 (± 2) ms	9	28,9 MB
	5 ms	192 (± 3) ms	20	57,3 MB

Table 4.9: Nexmark q3, q5, q8 various snapshot frequencies

nx query	snapshot frequency	execution time	taken snapshots	datastore size
nx 4	T	185 (± 1) ms	1	26,8 KB
	90 ms	300 (± 5) ms	3	93,9 MB
	70 ms	359 (± 8) ms	4	142 MB
	50 ms	634 (± 53) ms	8	326 MB
	30 ms	1499 (± 125) ms	(16-30)	942 MB
nx 6	T	188 (± 1) ms	1	59,0 KB
	90 ms	307 (± 7) ms	3	96,3 MB
	70 ms	372 (± 15) ms	4	147 MB
	50 ms	644 (± 66) ms	8	324 MB
	30 ms	1872 (± 335) ms	(16-40)	1,13 GB

Table 4.10: Nexmark q4, q6 various snapshot frequencies

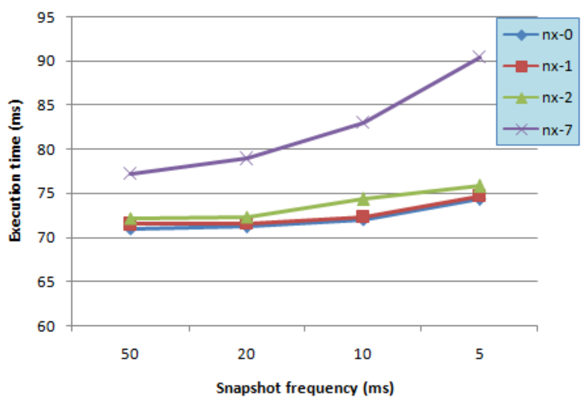

Figure 4.17: Nexmark query 0, 1, 2, 7

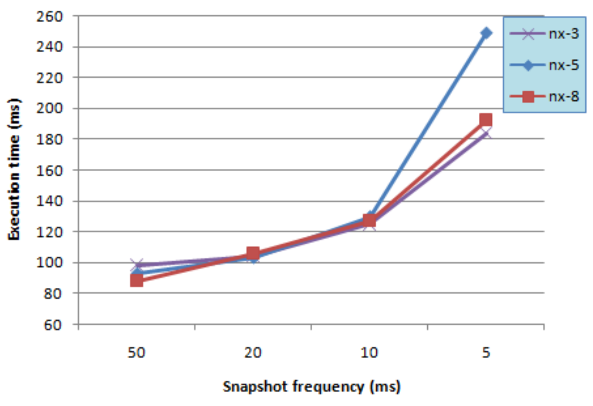

Figure 4.18: Nexmark query 3, 5, 8

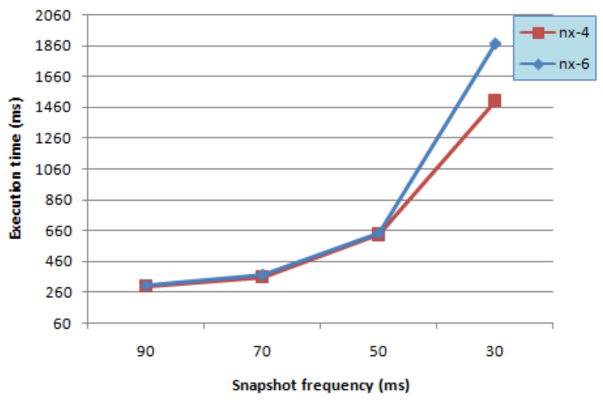

Figure 4.19: Nexmark query 4, 6

In tables 4.9 and 4.10, there are rows highlighted in red, indicating the presence of drift. For those experiments we see a notable increase in execution times and the size of saved states, furthermore, there is a lot of variability in the number of snapshots taken and consequently in execution times.

The drift phenomenon represents a misalignment in the generation of snapshot tokens due to the fact that Sources produce data at a faster rate than the subsequent blocks take to process it. This leads to a pause in the generation of tuples from the Sources to allow the subsequent blocks to process and consume previously generated data. However, if the time during which the Source is paused exceeds the snapshot interval, once the Source resumes producing tuples, its SnapshotGenerator generates snapshot tokens that it could not produce when the Source was inactive. This can result in the generation of several snapshot tokens very close together and delayed from the time they were supposed to be produced.

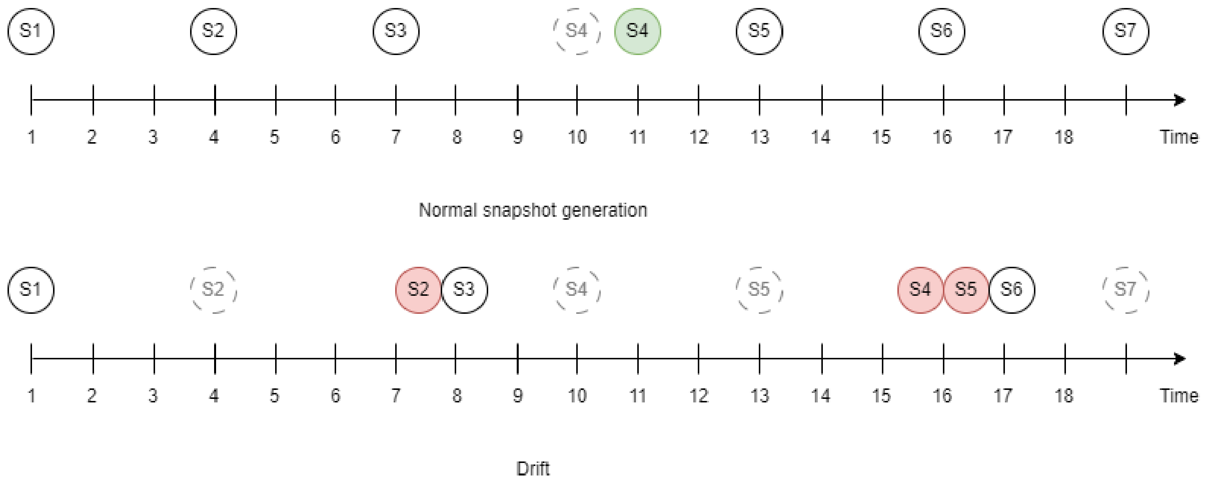
S1
S2
S3
S4
S4
S5
S6
S7
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 Time
Normal snapshot generation
S1
S2
S2
S3
S4
S5
S4
S5
S6
S7
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 Time
Drift

Figure 4.20: Snapshot generation drift

In the figure 4.20, we observe the representation of snapshot generation in normal time (top) and with drift (bottom). In normal generation, there may be delays, as seen for S4, but the delay is less than the snapshot interval and is recovered. With drift, the delay exceeds the snapshot interval and causes the close emission of multiple snapshot tokens. The misalignment of snapshot tokens leads to an increase in the overhead of the algorithm and the size of the saved states, similar to what occurs with item-based frequency in the previous section.

This can result in a further slowdown of subsequent blocks due to the snapshot procedure triggered by these tokens. The Start operators at the beginning of the block have to deal with several snapshots simultaneously that arrive non-aligned in time. This increases the number of messages saved in the queues, leading to higher memory consumption and, consequently, an increase in time. If the slowdown of subsequent blocks increases the time during which the Sources do not produce tuples, the delay of snapshot tokens accumulates, triggering a cascading effect that leads to an exponential increase in execution times and saved snapshots.

Drift can be detected from the logs; a warning is logged every time the SnapshotGenerator timer has a duration greater than twice the snapshot interval. This indicates that the SnapshotGenerator will produce two or more closely spaced snapshot tokens. The presence of drift depends mainly on the difference in speed between the production of tuples by the Sources and the processing of tuples by subsequent blocks, imposing a lower limit on the snapshot frequency.

However, it is essential to keep in mind that these experiments are conducted in a context different from the ideal use case. In an ideal scenario, snapshot frequencies and execution times are orders of magnitude higher, and it is unlikely that the input stream is a sequence of integers generated at runtime. Typically, the data to be processed is read from an external source, which should make Sources slower, reducing the likelihood of drift occurrence.

The Wordcount benchmark, with maximum available parallelism (12) and an input stream of 1,000,000 tuples, produced the results in table 4.11.

wordcount	snapshot frequency	execution time	taken snapshots	datastore size
wc-fast	T	392 (± 1) ms	1	13,2 KB
	100 ms	397 (± 1) ms	4	168 KB
	50 ms	400 (± 1) ms	8	374 KB
	10 ms	425 (± 2) ms	42	1,97 MB
	5 ms	464 (± 3) ms	90	3,99 MB
wc-fold-assoc	T	1,78 (± 0,01) s	1	17,9 KB
	500 ms	1,81 (± 0,01) s	4	111 KB
	100 ms	1,86 (± 0,04) s	19	563 KB
	50 ms	1,87 (± 0,04) s	37	1,08 MB
	10 ms	1,97 (± 0,01) s	180	5,28 MB
wc-fold	T	2,43 (± 0,04) s	1	16,9 KB
	500 ms	2,45 (± 0,06) s	5	2,85 MB
	100 ms	2,46 (± 0,05) s	25	16 MB
	50 ms	2,53 (± 0,05) s	51	33,7 MB
	10 ms	3,00 (± 0,02) s	297	204 MB

Table 4.11: Wordcount various snapshot frequencies

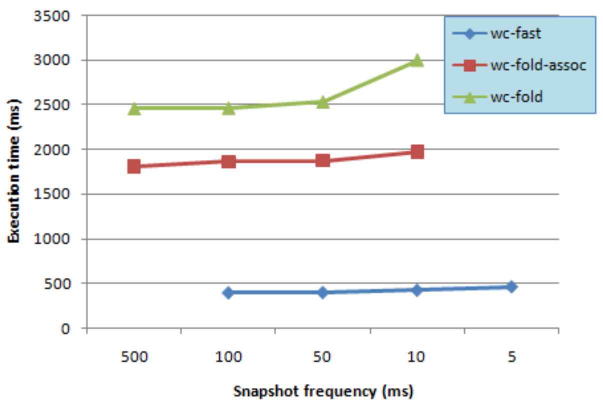

Figure 4.21: Wordcount benchmark

Similarly to Nexmark, the overhead introduced by snapshots varies depending on the specific benchmark and snapshot frequency. For Wordcount, we observe that the cost factor determined by the snapshot algorithm is influenced by the presence of Start operators with NextStrategy: GroupBy. These benchmarks, executed with these parameters, do not exhibit the drift phenomenon.

4.3.1. Drift mitigation

To attempt to eliminate drift, we reconsidered the initial assumptions about the properties of the SnapshotId. In particular, by removing the regularity property, it is possible to avoid generating and injecting tokens too late into the network. Specifically, we want to prevent the accumulation of snapshot tokens. Therefore, the SnapshotGenerator, based on the time elapsed since the last snapshot, directly generates just one token with the higher id compatible with the elapsed time, bypassing all the previous ones. The SnapshotGenerator skips the generation of a snapshot when it can generate one with a higher SnapshotId and if the Source has not emitted any other tuples since the last snapshot token. In other words, it avoids generating snapshots that would lead to saving identical states. In figure 4.22, the generation with drift is depicted at the top, and at the bottom, it illustrates how the generation changes by skipping snapshots that are too late, such as S2, S4, and S5.

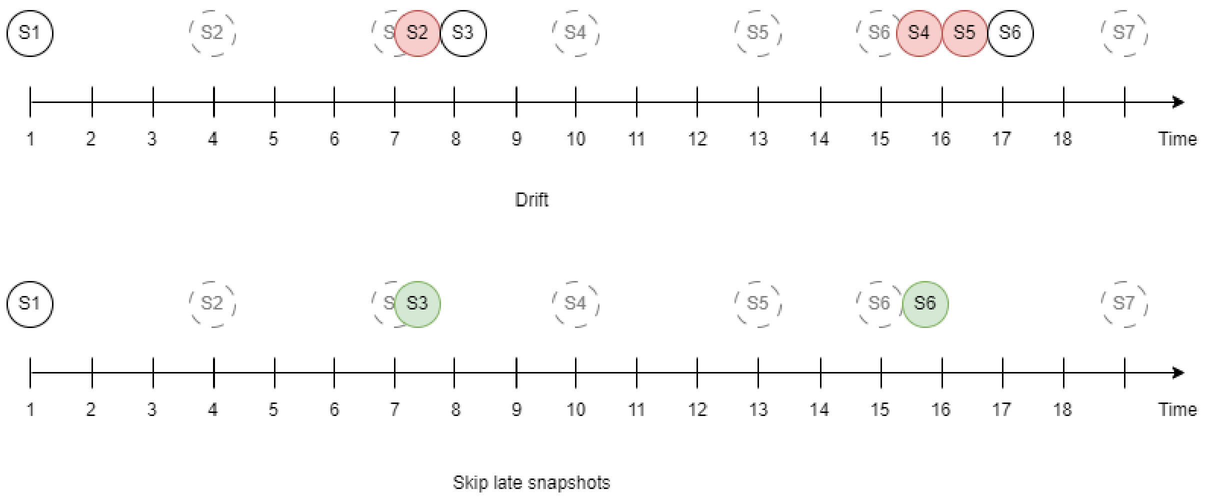

Figure 4.22: Snapshot generation: skip too late snapshots

This modification affects the snapshot procedure performed by the Start operators:

1. Check if this SnapshotId is already arrived
2. If it has not arrived:
    (a) Check if Start has previously received bigger SnapshotIds
    (b) If no:
        i. Copy the current state and add this partial snapshot to the HashMap
        ii. This snapshot token will be sent to the next operator
    (c) If yes, this snapshot won't be taken, this token is only used by the next point to conclude previous ongoing snapshots, if any.
3. Find all ongoing snapshots with SnapshotId less or equal to this one
4. For each of them
    (a) Remove the sender of this snapshot from the set of previous replica coordinates
    (b) If the set is empty, the snapshot is complete, remove it from the HashMap and persist it

For Iterative operators, some modifications are necessary:

- The SnapshotGenerator of the IterationLeader continues to generate snapshot tokens following the regularity property since the generation is synchronized with the iterations.

- The snapshot alignment solution through the alignment block is no longer valid because the Start operator can take different snapshots depending on the order in which it receives snapshot tokens from the preceding replicas. Therefore, in the case of iterations, it is mandatory to use the alignment solution in which the Sources produce only one snapshot token. Given that the behavior of the IterationLeader remains unchanged, everything else remains as before.

The procedure for determining the last snapshot changes too. The function "find last snapshot", which searches for the last valid snapshot, remains unchanged and always returns the minimum of the last snapshots saved for each operator. However, the "restart from" function called by each operator during the setup phase has been modified. In the case where the snapshot calculated by "find last snapshot" has been saved by the operator, it resumes normal operation from that snapshot. On the other hand, if the operator has not saved that snapshot, it restarts from the smallest snapshot greater than that. Consider, for instance, a scenario with three operators that have persisted snapshots as shown in table 4.12. The function "find last snapshot" takes the minimum between S7, S5, S6; which is S5. The function "restart from" returns S7 for op1, S5 for op2 and S6 for op3.

Operator	List of persisted snapshots
op 1	S1, S2, S4, S7
op 2	S1, S2, S4, S5
op 3	S1, S2, S4, S6

Table 4.12: Example of computation of last snapshot

Through the Nexmark benchmark, we can observe the differences between the version with drift and the version in which snapshots too delayed are skipped. Regarding queries 0, 1, 2, 3, 7, and 8, no differences were detected between the two versions, while for queries 4, 5, and 6, the results are presented in the table 4.13.

nx query	snapshot frequency	execution time	taken snapshots	datastore size
nx 4	T	185 (± 1) ms	1	26,8 KB
	90 ms	298 (± 5) ms	3	93,9 MB
	70 ms	370 (± 10) ms	4	146 MB
	50 ms	561 (± 16) ms	8	288 MB
	30 ms	1165 (± 50) ms	(15-20)	717 MB
	15 ms	3048 (± 104) ms	(40-55)	1,92 GB
nx 5	T	85 (± 1) ms	1	15,3 KB
	50 ms	93 (± 1) ms	2	2,61 MB
	20 ms	103 (± 1) ms	5	10,1 MB
	10 ms	128 (± 1) ms	12	27,1 MB
	5 ms	237 (± 5) ms	(30-37)	81,5 MB
	4 ms	307 (± 5) ms	(40-50)	114 MB
nx 6	T	188 (± 1) ms	1	59,0 KB
	90 ms	309 (± 8) ms	3	96,3 MB
	70 ms	368 (± 11) ms	4	147 MB
	50 ms	600 (± 38) ms	8	302 MB
	30 ms	1258 (± 62) ms	(15-24)	775 MB
	15 ms	3051 (± 115) ms	(43-57)	1,93 GB

Table 4.13: Nexmark q4, q5, q6 skip too late snapshots

As one can see, this modification reduces the impact of drift; however, it is not possible to eliminate misalignment entirely, and there is no assurance that all replicas of the Source skip precisely the same snapshots. Due to the independence of the Sources, it can happen that the snapshots skipped by one replica differ from those skipped by another replica, leading to the generation and propagation of misaligned snapshots nonetheless.

4.4. Iterations

For iterative operators, we compare the behavior, varying the number of iterations, between two snapshot alignment solutions: inhibit Source snapshot token generation (isa) and the snapshot alignment block (align-block).

The benchmark consists of a Filter operator that only allows prime numbers to pass through in the input-generated stream. To verify if a number is prime, it searches for all divisors from 2 to the square root of the number, which is a quite computationally expensive operation. The Filter is followed by a loop that iterates over the stream with prime numbers a fixed number of times. This setup allows us to observe the differences between the two alignment solutions and how they change as the number of iterations varies. Specifically, as long as the number of iterations is low, the main computation is performed by operators preceding the loop, resulting in significant differences in execution times and the number of snapshots taken. As the number of iterations increases, most of the computational load switches to the iterative operator, reducing the percentage differences in execution times and the number of snapshots taken.

In the following charts 4.23 4.24 4.25, execution times are plotted against the snapshot frequency in the case of three different executions with 10, 100, and 500 iterations, respectively. The input stream has a size of 10,000,000 tuples, and the maximum available parallelism (12) is utilized. As with the previous benchmarks, the frequency "T" indicates that no snapshot frequency has been set, and only the Terminate snapshot is saved.

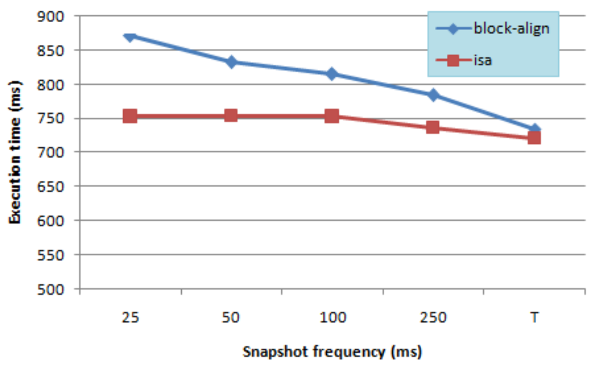

Figure 4.23: Iteration benchmark: 10 iterations

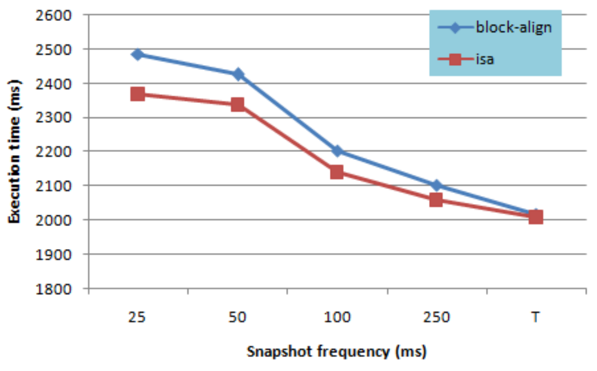

Figure 4.24: Iteration benchmark: 100 iterations

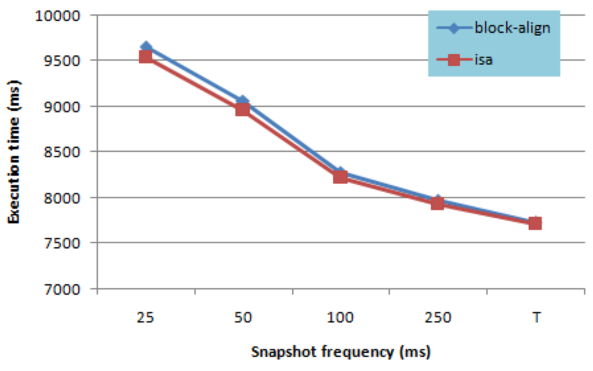

Figure 4.25: Iteration benchmark: 500 iterations

5 | Conclusions and future developments

Noir is an excellent streaming and batch processing framework that enables various types of computations and analyses on bounded and unbounded streams, outperforming even Apache Flink. However, it lacks fault tolerance, and in the event of a failure of an operator or a connection, all ongoing execution results are lost.

With the work of this thesis, it is now possible to periodically persist the state, allowing recovery from the last saved system state in case of failure. The implemented snapshot algorithm works independently and without any central coordinator, preserving Noir's original architecture and design.

The analyses of benchmark results seek to understand and provide an idea of the impact on performance due to state persistence and how it varies based on stream characteristics and specifications. The most significant aspect that emerged from the analyses is the impact of snapshot token alignment among different replicas. This algorithm performs better when the snapshot tokens are generated and propagated in a more aligned manner. This implies that to enhance the performance of this algorithm, strategies should be explored to improve the alignment of snapshot tokens. Alternatively, it is necessary to design a mechanism that efficiently handles misaligned snapshots, such as improving the management and storage of partial snapshots for the Start operator.

A possible future development is to implement state persistence as a feature. With a compile-time feature flag, it is possible to remove the code used for state persistence at compile time, reducing the gap between the execution time of the original Noir and the version with state persistence disabled.

Another useful improvement could be automatic failure detection and restart, masking a failure completely from the user. Although it may not be possible to recover from all types of errors, for example, if the error is inherent to the tuples, it will reoccur even after restart. The fail detection phase must be able to distinguish between recoverable failures and unrecoverable failures.

Bibliography

[1] criterion - rust. URL https://docs.rs/criterion/latest/criterion/.

[2] Crossbeam. URL https://docs.rs/crossbeam/latest/crossbeam/.

[3] Flume. URL https://docs.rs/flume/latest/flume/.

[4] Apache kafka. URL https://kafka.apache.org/.

[5] Redis, . URL https://redis.io/.

[6] redis - rust, . URL https://docs.rs/redis/latest/redis/.

[7] The rust programming language, . URL https://doc.rust-lang.org/book/.

[8] Rust closures, . URL https://doc.rust-lang.org/book/ch13-01-closures.html.

[9] Rust iterator, . URL https://doc.rust-lang.org/std/iter/trait.Iterator.html.

[10] Apache samza, . URL https://samza.apache.org/.

[11] Apache samza - checkpointing, . URL https://samza.incubator.apache.org/learn/documentation/0.7.0/container/checkpointing.html.

[12] Apache samza - state management, . URL https://samza.incubator.apache.org/learn/documentation/0.7.0/container/state-management.html.

[13] Apache storm, . URL https://storm.apache.org/.

[14] Apache storm - state management, . URL https://storm.apache.org/releases/2.5.0/State-checkpointing.html.

[15] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8

(12):1792–1803, aug 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824076. URL `https://doi.org/10.14778/2824032.2824076`.

[16] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. 06 2015.

[17] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, feb 1985. ISSN 0734-2071. doi: 10.1145/214451.214456. URL `https://doi.org/10.1145/214451.214456`.

[18] E. Morassutto and M. Donadoni. Noir : design, implementation and evaluation of a streaming and batch processing framework. Master's thesis, ING - Scuola di Ingegneria Industriale e dell'Informazione, 2021. URL `https://hdl.handle.net/10589/180143`.

[19] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. pages 439–455, 11 2013. doi: 10.1145/2517349.2522738.

[20] C. Paris, K. Asterios, E. Stephan, M. Volker, H. Seif, and T. Kostas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38, 01 2015.

[21] C. The MPI Forum. Mpi: A message passing interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, page 878–883, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0818643404. doi: 10.1145/169627.169855. URL `https://doi.org/10.1145/169627.169855`.

[22] P. A. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark – a benchmark for queries over data streams draft. 2002. doi: 10.1145/2517349.2522738. URL `https://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf`.

[23] X. Wang. A comprehensive study on fault tolerance in stream processing systems. *Frontiers of Computer Science*, 16, 09 2020. doi: 10.1007/s11704-020-0248-x.

List of Figures

Listings

List of Tables

Acknowledgements

I would like to thank Professor Alessandro Margara, Professor Gianpaolo Cugola, and Luca De Martini for their invaluable support and assistance throughout the completion of this thesis.

I also thank my parents, Angela and Giandomenico, and my sister, Anna, for their moral and financial support received throughout my university studies.