Executive Summary of the Thesis

State Persistence in Noir

Laurea Magistrale in Computer Science Engineering - Ingegneria Informatica

Author: Emmanuele Lattuada

Advisor: Prof. Alessandro Margara

Co-advisors: Prof. Gianpaolo Cugola, Luca De Martini

Academic year: 2022-2023

1. Introduction

The continuous growth in data volumes demands more efficient solutions capable of handling and processing large datasets to extract meaningful models and statistics. Distributed systems have been developed to distribute computation across clusters of hosts, and new software architecture paradigms have been proposed.
These systems are typically used for long-term and complex computations on very large datasets or data streams. Therefore, having fault-tolerance mechanisms is crucial to prevent the loss of work in case of failure. A common approach to this problem involves periodically saving the system's state to create checkpoints for recovery in case of failure.
This thesis focuses on designing a snapshot algorithm for Noir, a promising streaming and batch processing system. The snapshot algorithm enables the periodic saving of the system's state to a reliable external source, facilitating recovery to the last valid state in the event of a failure. Some experiments have been conducted to try to understand the algorithm's impact on Noir's performance.

2. Noir

Noir [3] is a streaming and batch processing system developed by a research team at Politecnico di Milano and implemented in Rust. It enables the analysis of both bounded and unbounded generic streams and is based on the dataflow paradigm. Noir has a rich set of API inspired by Apache Flink, but it can achieve better performance. It provides various functions and abstractions such as map, reduce, filter, join, windows, and iterations. Additionally, it can handle partitioned key-value streams and timestamp-value event streams.
The functions and transformations applied to the stream result in a directed graph of operators, where the data stream enters through the initial operators, known as Sources. After traversing the entire graph and being processed, the stream exits as the output of the final operators, known as Sinks. The edges of the graph represent communication channels used by operators to receive stream tuples and send newly generated tuples to subsequent operators.
In Noir, the job graph is initially created, containing an instance for each operator applied to the stream. Subsequently, the execution graph is generated, which includes the final graph of operators. The key distinction from the job graph

is that operators are replicated to fully leverage available parallelism. The replication of operators depends on factors such as the number of hosts in the cluster, the available parallelism for each host, and the nature of the operator itself, as some operators cannot be replicated.
Operators are grouped into blocks, and within a block, operators form a chain where each operator is preceded by a single operator and followed by another single operator. At the beginning of the block, there is always a Source operator or a Start operator. The former produces stream tuples, while the latter receives data from the previous blocks. At the end of the block, there is always a Sink operator or an End operator. The former receives and manages the results produced by the computation, while the latter sends tuples to the subsequent blocks. The blocks are replicated exactly like the operators they contain, so all operators within a block have the same replication constraints.
The block has a pull-based architecture in which control passes from the last operator of the block to the previous ones. Each operator has two main control functions: setup() and next(). The setup() function is used to set up the operator before starting processing, while the next() function performs the processing, always returning the processed tuple. Communication within a block occurs only through function calls and return values. Communication between two blocks occurs through communication channels, specifically in-memory channels if the two blocks are instantiated on the same host or TCP channels if they are instantiated on two different hosts.
The execution phases of a program in Noir are as follows:

- Create the environment through the configuration: local if you want to run only on localhost or remote by specifying the hosts belonging to the cluster.
- Spawn remote workers: in the case of remote execution, the program is sent and launched on all hosts in the cluster.
- Define the functions to apply to the stream: in this phase, the job graph is generated.
- Start execution: in this phase, the execution graph is computed, blocks are instantiated, and operators are set up.
- Processing: the stream is produced by Sources, processed by operators, and the results are received and managed by Sinks.

3. Implementation

We chose to implement a non-blocking snapshot algorithm without a central coordinator to preserve the original architecture of Noir. Each operator independently saves its state, and the procedure is triggered by a special snapshot message. This algorithm is inspired by Chandy-Lamport [2] and Asynchronous Barrier Snapshotting (ABS [1]) but has differences from both.
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.

3.1. Snapshot algorithm

The snapshot is initiated by Sources, which periodically inject a special snapshot message into the stream. Since each Source is independent and autonomous from the others, the snapshot frequency is defined over a time interval. This ensures that each Source generates the snapshot token at the same moment (within the precision of the system time). Snapshot tokens contain a monotonically increasing ID.
The snapshot procedure is as follows:

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.

The figure 1 shows step-by-step the operations of the algorithm, at step 6 the operator persists the state containing its internal state taken at step 2 and the message queue with "a" and "d".

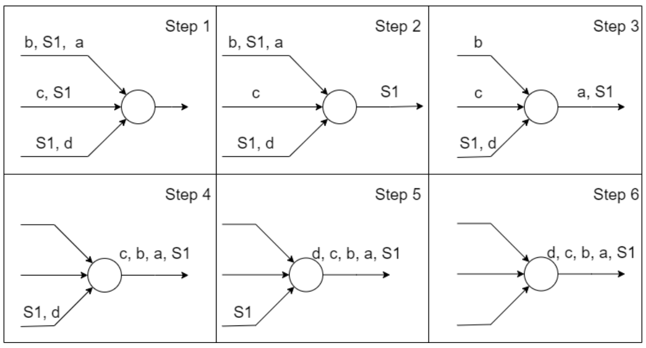

Figure 1: Snapshot algorithm

The persisted state consists of the internal state of the operator and the message queue. In Noir, only the Start operator maintains the message queue; all other operators (except for iterative ones, as explained later) have a single input channel. Therefore, upon receiving the snapshot token, they simply save their state and forward the token to the next operator.
The internal state of each operator varies depending on the specific operator and contains only information that changes and depends on the tuples processed previously. The size of the state also depends on the specific operator; for some, it is fixed, while for others, it depends on the tuples processed previously. Each operator independently saves its state on Redis using a key that combines the operator's coordinates, uniquely identifying it within the execution graph, and the ID of the snapshot.

3.2. Iterations

Iterative operators introduce cycles within the operator graph, they are blocking and they consume the entire stream during the first iteration. Another critical aspect of loops is the shared state among the loop replicas, which is updated at each iteration by the IterationLeader operator. The IterationLeader is not replicated and represents the unique synchronization point between iterations.
For these reasons, a dedicated procedure has been developed, which can be distinguished into two phases. During the first:

1. When a snapshot token arrives at the iterative operator from the input channel, the operator copies its internal state and forwards the new token to the operators in the body.
2. If the iterative operator has a feedback channel through which it receives tuples generated by the loop body, it copies and buffers all tuples arriving on the feedback channel before the snapshot token. When the token arrives on the feedback channel, it can save the state (internal state and message queue).
3. The operators in the body take the snapshot as usual.
4. When the IterationLeader receives the token for the first time, it copies it and saves its internal state. Then, it forwards the token to the external output channel of the loop.
5. After the IterationLeader detects the end of 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:

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 leader. 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.

The figure 2 represents the evolution of 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. The number within square brackets is a secondary index that distinguishes the snapshots taken within the loop.
With this algorithm, the snapshot tokens generated within the loop do not exit it, making the algorithm suitable for nested loops. Additionally, if the dataflow contains multiple loops in parallel or in sequence, the snapshots of the different loops are taken and considered independently.
This algorithm requires that all replicas of the iterative operator receive the same snapshot tokens during the first iteration. For this reason, two possible token alignment solutions have been implemented. The first solution blocks the generation of tokens from the Sources, ensuring that a single token is generated at the end of the stream tuples. The second solution adds a special block before the loop that aligns the snapshots. This solution should be used only if the operators before the loop are the prevalent part of the computation. It is mandatory to use the first solution in case of side inputs.

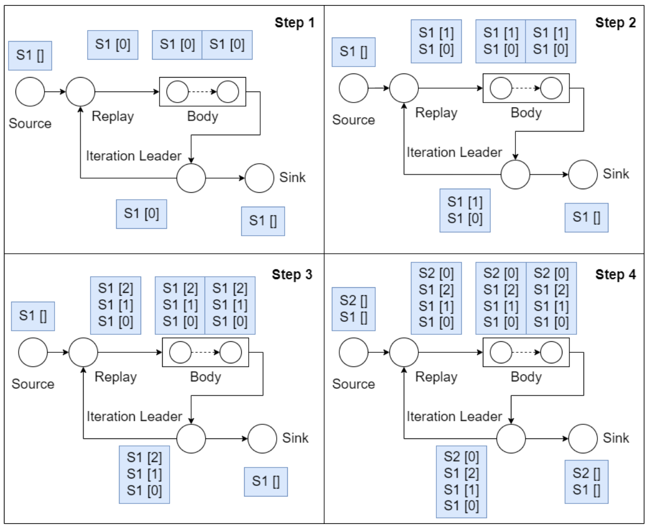

Figure 2: Snapshot algorithm for iterations

3.3. Recovery

The recovery procedure determines the last complete saved snapshot and allows resuming from it or from a specific snapshot chosen by the user. Since the saving occurs autonomously and in a decentralized manner, a snapshot is considered complete only when all operators in the graph have saved the snapshot. This means that to determine the last complete snapshot or if the user-requested snapshot is valid, the coordinates of all operators in the network are necessary. This is the only part of the code where the global snapshot is considered, not the individual saved states of the operators. Once the ID from which to resume has been determined, this information is passed to the operators, which, during the setup phase, retrieve the state and resume from it. Start operators that have saved the message queue must process those messages before being able to receive and process new tuples.

4. Performance evaluation

Several experiments were conducted to study the impact of this algorithm on performance. In particular, two benchmarks were used: Nexmark and Wordcount. The first is a set of queries commonly used to evaluate the performance of a stream processing system. 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. The second is a benchmark that calculates the number of occurrences of each word in a text document, containing three different implementations performing this analysis.
All experiments were executed with an input stream of size 1'000'000 and with local configuration with replication equal to 12. This setup does not represent the execution scenario for which the snapshot algorithm was designed. However, it is still useful to get an idea of the algorithm's behavior.
The graphs 3 and 4 compare the original version of Noir with this new version but with persistence disabled.

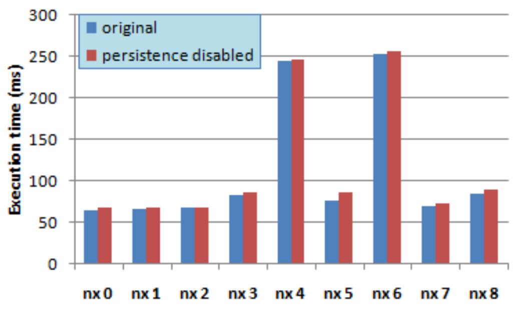

Figure 3: Nexmark benchmark: original Noir vs persistence disabled

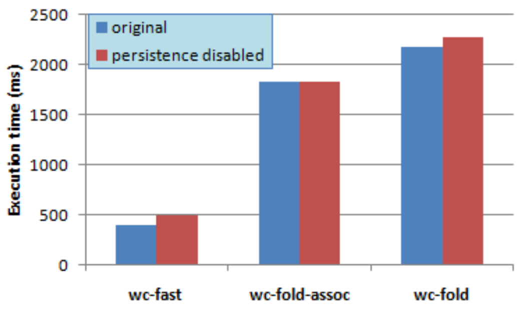

Figure 4: Wordcount benchmark: original Noir vs persistence disabled

The other experiments examine how the execution time of benchmarks evolves with varying snapshot algorithms. In the chart 5, the Wordount benchmarks are represented, while in the charts 6 and 7, the Nexmark benchmarks are presented. Some queries are represented with the same line because the execution times are very close. The overhead introduced by the snapshot algorithm is variable. However, the growth factor of the execution time is mainly determined by the number of Start operators receiving data from multiple replicas.

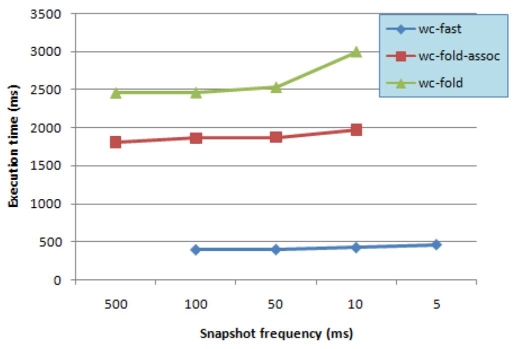

Figure 5: Wordcount benchmark: various snapshot frequencies

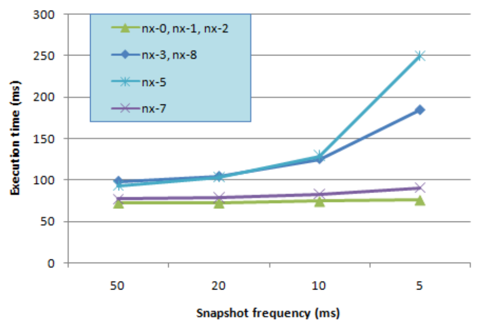

Figure 6: Nexmark query 0, 1, 2, 3, 5, 7, 8: various snapshot frequencies

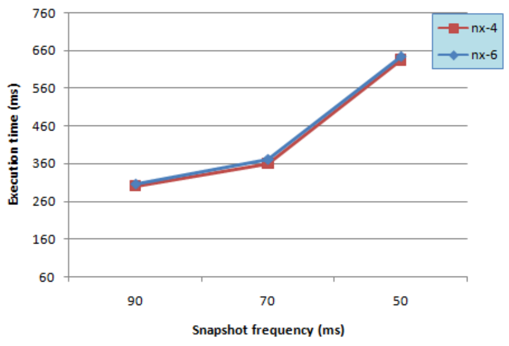

Figure 7: Nexmark query 4, 6: various snapshot frequencies

5. Conclusions

We have designed and implemented a snapshot algorithm that enables periodic state saving in Noir. State persistence marks a significant evolution for Noir as it allows the system to recover from failures by resuming execution from the last saved state, thus avoiding restarting from scratch. The algorithm has been tested and the conducted experiments highlight differences with the original version of Noir and illustrate the evolution of execution times with varying snapshot frequencies.

6. Acknowledgements

Thanks to Prof. Alessandro Margara, Prof. Gianpaolo Cugola and Luca De Martini who guided me during this thesis, and thanks to my family who supported me during my studies.

References

[1] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. 06 2015.

[2] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, feb 1985.

[3] Edoardo Morassutto and Marco 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.