

TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING



Migration state among jobs in Apache Flink

Baikousis Ioannis

Thesis Committee

Prof. Antonios Deligiannakis (Supervisor)

Prof. Minos Garofalakis

Prof. Vasilios Samoladas

Abstract

On a daily basis more and more data is produced and needs to be processed to extract useful information. Specifically, processing data streams is vital and requires high-performance resources to query them in real time. Big Data processing frameworks have been developed to handle efficient complex queries on data streams. As the amount of data expands, the frameworks have to adapt to processing requirements, thus it is essential to support updates and upgrades both in hardware and software infrastructures over time. Therefore, migration mechanisms have been developed in order to give the ability to frameworks to evolve, guaranteeing no data losses.

In this diploma thesis we provide a migration algorithm in Apache Flink which gives you the opportunity to manage many operator states among different Flink jobs and submit them into another cluster with no data losses. Additionally, it enables us to merge, split or rescale jobs in order to adapt to processing requirements. Our algorithm is based on the State Processor API which is provided by Flink and it is implemented on RapidMiner Studio which gives us the ability to design workflow easily and quickly.

To validate our approach, we designed some workflows using simple operators on RapidMiner studio and present a complete detailed cluster-mode execution with many test-cases as merging and splitting the workflows and migrating the state without data losses proving the correctness of our migration algorithm.

Περίληψη

Καθημερινά, τόσο και περισσότερα δεδομένα παράγονται κάνοντας την επεξεργασία αυτών μείζονος σημασίας ουτωςώστε να εξάγουμε την χρήσιμη πληροφορία. Ειδικότερα η επεξεργασία ροών δεδομένων είναι ζωτικής σημασίας και απαιτεί υψηλών επιδόσεων υπολογιστικούς πόρους ώστε να μπορέσουμε να κάνουμε επερωτήσεις σε πραγματικό χρόνο σε αυτά. Για αυτόν τον λόγο, έχουν δημιουργηθεί big data frameworks τα οποία έχουν την δυνατότητα να επεξεργαστούν και να χειριστούν αποδοτικά σύνθετα επερωτήματα πάνω σε ροές δεδομένων. Όσο ο όγκος των δεδομένων μεγαλώνει τα frameworks πρέπει να προσαρμόζονται στις απαιτήσεις που χρειάζονται για την επεξεργασία τους, συνεπώς είναι σημαντικό να δέχονται τακτικά αναβαθμίσεις σε υλικό και λογισμικό. Για τον λόγο αυτό έχουν αναπτυχθεί μηχανισμοί που δίνουν την δυνατότητα να μπορούν να αναβαθμιστούν χωρίς καμία απώλεια δεδομένων.

Σε αυτήν την διπλωματική εργασία παρουσιάζεται ένας αλγόριθμος μεταφοράς του state χρησιμοποιώντας το Apache Flink framework, ο οποίος μας δίνει την δυνατότητα να επεξεργαστούμε operator states ανάμεσα σε διαφορετικά Flink jobs και να μεταφέρουμε το state σε άλλα clusters χωρίς καμία απώλεια δεδομένων. Ακόμα μας δίνει την δυνατότητα να ενώσουμε, να διαχωρίσουμε ή να κάνουμε rescale τα jobs ώστε να προσαρμοστούν στις επεξεργαστικές απαιτήσεις. Ο αλγόριθμος βασίζεται στο State Processor API που παρέχεται από το Flink και είναι υλοποιημένος στο RapidMiner Studio το οποίο μας δίνει την δυνατότητα να σχεδιάσουμε εύκολα και γρήγορα workflows.

Για να επικυρώσουμε την ορθότητα της υλοποίησης μας, έχουμε σχεδιάσει μερικά workflows στο RapidMiner Studio τα οποία περιέχουν απλούς τελεστές και παρουσιάζουμε μια αναλυτική περιγραφή της εκτέλεσης σε cluster-mode παίρνοντας περιπτώσεις όπως η ένωση και ο διαχωρισμός τους χωρίς να χάνονται δεδομένα καθώς και η μεταφορά του state αποδεικνύοντας έτσι την ορθότητα του αλγορίθμου μας.

Acknowledgments

First of all, I would like to thank my supervisor professor, Prof. Antonios Deligiannakis, who directed and guided me properly for this diploma thesis. I am also grateful to the other two committee professors, Prof. Minos Garofalakis and Prof. Vasilios Samoladas for their support.

In addition, I am thankful to my parents and friends who helped me to great extent to finish this diploma thesis and supported me over the years.

Contents

Abstract.....	3
Περίληψη.....	4
Acknowledgments.....	5
Contents.....	6
Figure Table.....	7
1. Introduction.....	8
1.1. Thesis outline.....	9
2. Apache Flink.....	10
2.1 DataSet API.....	10
2.2 DataStream API.....	10
2.3 Programs and Dataflows.....	10
2.4 Parallel Dataflows.....	12
2.5 Stateful Operators	13
2.6 Checkpoints and Savepoints	13
2.7 State Backend	13
2.7.1 Memory State Backend	14
2.7.2 FileSystem State Backend	15
2.7.3 RocksDB State Backend	16
2.8 Unify binary format	17
2.9 State Processor API	18
3. RapidMiner Studio.....	19
3.1 RapidMiner Extensions	19
4. Implementation.....	22
4.1 The Migration Algorithm.....	22
4.1.1 Steps of the Algorithm.....	22
4.2 RapidMiner Studio.....	24
4.2.1 Stateful operators.....	24
4.2.2 Restart Implementation.....	25
5. Experimental Execution.....	27
5.1 Split-Merge example.....	27
5.2 Migration example.....	35

Conclusion.....	37
References.....	38

Figure Table

Figure 1: Execution of a stream.....	11
Figure 2: Parallel execution.....	12
Figure 3: Memory State Backend diagram.....	14
Figure 4: File System State Backend diagram.....	15
Figure 5: RocksDB State Backend diagram.....	16
Figure 6: Unify binary format schema.....	17
Figure 7: A workflow consists of one job.....	21
Figure 8: Implementation of the Streaming Nest operator.....	21
Figure 9: The Streaming Nest operator to create the job.....	27
Figure 10: A workflow consists of two Aggregate Stream operators.....	27
Figure 11: The Json file to run a job without restart.....	28
Figure 12: The submitted job on the cluster.....	28
Figure 13: The results of the new job after data production.....	29
Figure 14: A workflow consists of two jobs.....	30
Figure 15: Implementation of the first Streaming Nest operator.....	30
Figure 16: Implementation of the second Streaming Nest operator.....	31
Figure 17: The Json file in order to split a workflow to two jobs.....	32
Figure 18: The submission of the new jobs.....	32
Figure 19: The generated files on hdfs both for the jobs1 and job2.....	33
Figure 20: The results of the new jobs after the data production.....	33
Figure 21: The merged workflow that consists of two Aggregate operators.....	34
Figure 22: The json file in order to merge two jobs.....	34
Figure 23: The generated files on hdfs for the job1.....	35
Figure 24: The submission of the new job and the cancellation of the previous ones.....	35
Figure 25: The results which are produced by the new job.....	35
Figure 26: The results of the produced data.....	36
Figure 27: The Json file to migrate the state.....	37
Figure 28: The results which are produced on the Technical University cluster.....	37

1. Introduction

Nowadays, the amount of data and information is increasing rapidly and it is vital to process and analyze it in real-time. More and more data is produced on a daily basis, from different sources like social media or IoT, making its analysis necessary in order to extract useful information and features. For this reason, frameworks have been developed, which enable us to process huge amounts of data very fast guaranteeing high accuracy and efficiency.

In this day and age, data science is one of the most significant scientific fields which constitutes an indispensable part of machine learning and data mining, sciences which are essential for technological evolution. It is not possible to express the definition of data science using strict terms but in simple words:

“Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from many structural and unstructured data.”[0]

Frameworks that enable developers to develop high efficient workflows to process data, for example Apache Flink which we use in this diploma thesis, become more and more vital in data science. Thus, developers with high skills on these frameworks are necessary in every data analysis process. Many technological companies took advantage of this fact and developed softwares, like RapidMiner studio, in order to give the ability to everyone with basic knowledge of data analysis to design high efficient data analysis workflows.

The processing frameworks are usually set up in distributed computer clusters and the most important features which are vital to provide a framework is a fault tolerant mechanism and a restart strategy plan which ensures non data losses. In many cases we take the advantage of these mechanisms to build a migration scheme in order to migrate the state in another computer cluster or update the current one.

In this diploma thesis we introduce a flexible migration mechanism in order to give users the ability to migrate a Apache Flink framework workflow implemented in RapidMiner studio without any data loss. Additionally, this mechanism enables you to reformat any workflow, adding or subtracting operators to provide new features in data analysis, rescale the parallelism of a workflow to adapt to the process requirements and merge or split different workflows to new ones, guaranteeing no loss of data.

1.1. Thesis outline

- Chapter 2

In this chapter we provide an overview of the Apache Flink framework and the usage of DataStream and Dataset APIs in it. Moreover, we analyze how Apache Flink manages the data via dataflows and how it parallelizes its processing. In addition, we will describe in simple terms what is State and State Backend and how we get snapshots. Finally, we present the State Processor API and give some code examples.

- Chapter 3

In the second chapter, we present the RapidMiner Studio and analyze some useful abilities that it provide us. Furthermore, we see the extension of the RapidMiner Studio and give a workflow example.

- Chapter 4

This chapter contains the main implementation and it is divided into two sub-chapters. In the first sub-chapter we will describe what changes we made in the RapidMiner Extension and in the second one we present the Migration Algorithm and how it works.

- Chapter 5

In the fifth chapter we will see some execution examples which include important circumstances in order to handle the migration of a state. Additionally, we will analyze a step by step way to merge and split workflows in order to create a new one, which is a combination of others.

- Chapter 6

The final chapter contains a recap both of the provided algorithm and the implementation in RapidMiner Studio as a conclusion.

2. Apache Flink

Apache Flink[1] is a distributed processing engine and an open source data analytics framework for batch and stream processing in real time. Apache Flink allows stateful computation over unbounded and bounded data streams at any scale, in high performance computations at in-memory speed. Furthermore, it is designed to run in all common environments and provides data distribution and fault tolerance mechanism. It requires compute resources in order to execute applications and integrates with all common cluster resource managers or as stand-alone cluster.

Flink provides APIs[2] in order to process data, followed by dedicated libraries, for common use cases.

2.1 DataSet API[3]

It enables you to apply transformations like filters, mapping, grouping and others on data sets which are bounded and created from different sources as files or collections. Furthermore, the results of the transformations are returned via sinks into distributed files or standard output.

2.2 DataStream API[4]

DataStream API provides a range of operators which are very common in stream processing such as windowing, and is based on functions like `map()`, `aggregate()` and `reduce()` giving us the ability to interact with streams. It [5] also uses a data stream class to represent collections of data in a Flink program which can be bounded or unbounded. It is essential to add at least one source to read and apply DataStream API methods to a data stream and one or more sinks as output.

2.3 Programs and Dataflows[6]

An example of a simple dataflow is provided in this section. The dataflows are basically directed acyclic graphs (*DAGs*) with a start (source) and an end (sink).

In this example which is provided below, the data is read by a kafka source as a stream of string which is our source operator of the dataflow. After this, data is passed into a map operator in order to be transformed into an Event stream. Now, another transformation is applied in the stream which uses a keyby operator, a window operator and finally a function to process and transform the stream into a Statistics DataStream. In the last step, the stream was written into a Bucketing sink.

```

DataStream<String> lines = env.addSource(
    new FlinkKafkaConsumer<>(...)); } Source

DataStream<Event> events = lines.map((line) -> parse(line)); } Transformation

DataStream<Statistics> stats = events
    .keyBy("id")
    .timeWindow(Time.seconds(10))
    .apply(new MyWindowAggregationFunction()); } Transformation

stats.addSink(new BucketingSink(path)); } Sink

```

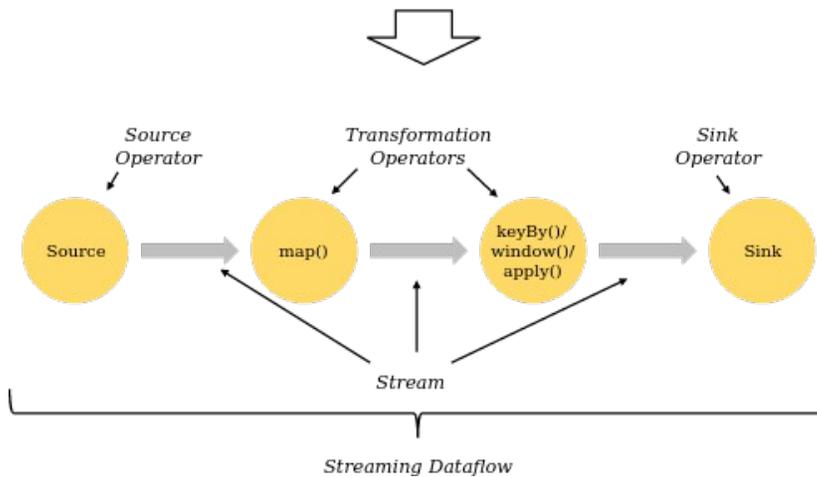


Figure 1: Execution of a stream.

2.4 Parallel Dataflows[7,8]

A Flink program consists of multiple tasks and every task is split into several parallel instances for execution and each parallel instance processes a subset of the task's input data. Parallelism is the number of parallel instances of a task.

As mentioned in the Flink documentation programs in Flink are parallel and distributed. During execution, a stream has one or more stream partitions, and each operator has one or more operator subtasks. The operator subtasks are independent, and executed in different threads.

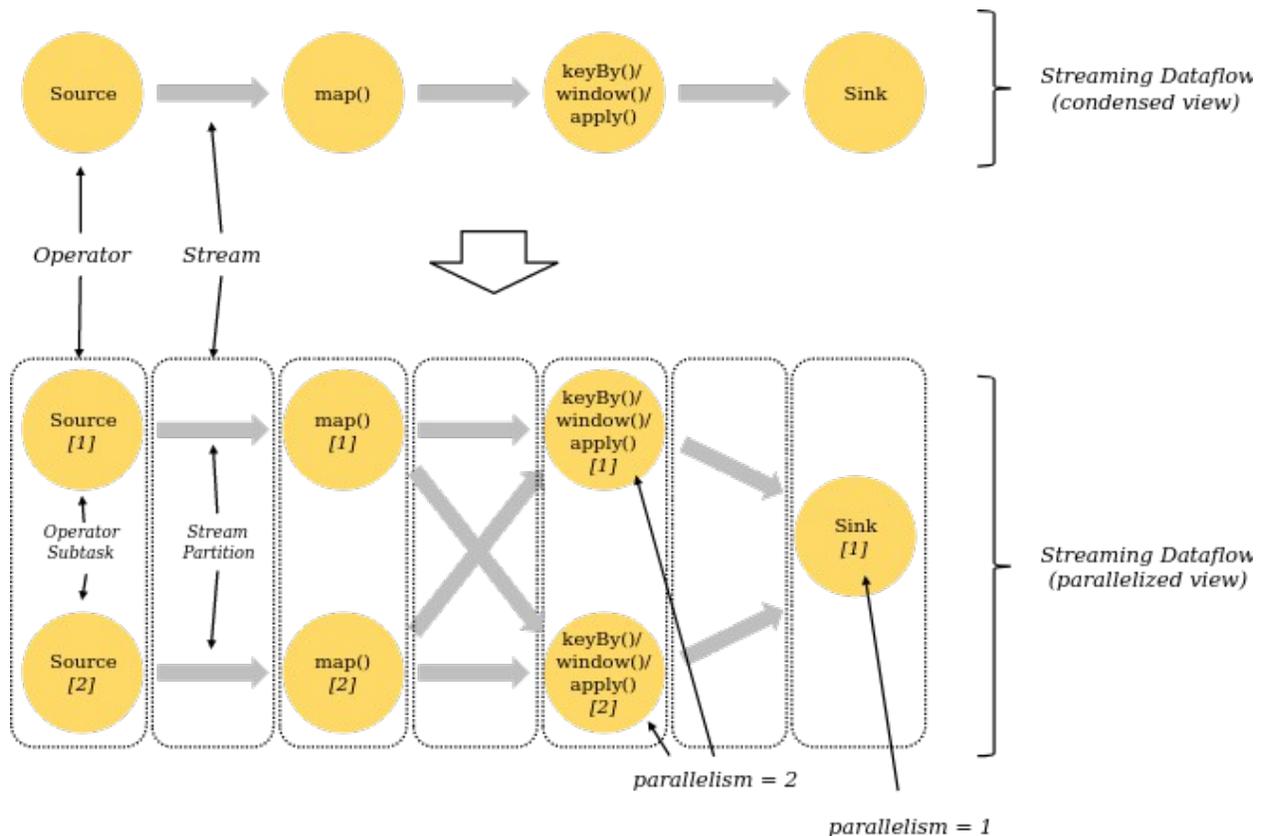


Figure 2: Parallel execution.

Data can be transported via streams between two operators in a one-to-one pattern which preserves the partitioning and ordering of the elements., or in a redistributing pattern in which streams change the partitioning of streams. At the Redistributing pattern each operator subtask sends data to different subtasks, depending on the selected transformation. For example keyBy() repartition the stream depending on the key. In a redistributing exchange the ordering among the elements is only preserved

within each pair of sending and receiving subtasks, but the parallelism is different regarding the order in which the aggregated results for different keys arrive at the sink.

2.5 Stateful Operators [\[9\]](#)

Stateful operator is an operator that has the ability to store and access information of previous multiple events in case you want to collect or extract features independently of any window operator and with no data loss guarantee. The state has a key/value store format and is partitioned and distributed with the streams which are provided in the stateful operator.

2.6 Checkpoints and Savepoints [\[10\]](#)

Apache Flink provides a flexible fault tolerance mechanism based on distributed checkpoints, which is very useful in case of failure. A checkpoint is an asynchronous snapshot of the states of operators which are automated by the program. When the application fails, the program which uses checkpoints, resumes the process from the last completed checkpoint in order to ensure no data losses when it recovers.

Savepoint is a state snapshot mechanism which is non automated like checkpoints. A user can trigger a savepoint manually to take a state snapshot to update the application or migrate it in another cluster.

2.7 State Backend [\[11\]](#)

The state backend is responsible for the representation of the state internally and “how and where” it is persisted upon checkpoints. Apache Flink has three available state backends:

- *MemoryStateBackend*
- *FsStateBackend*
- *RocksDBStateBackend*

2.7.1 Memory State Backend [12,14]

This storage stores the data in the memory of each task manager's Heap as objects. Some disadvantages of its use are that the size of each state is limited and an aggregate state must fit into the JobManager memory to avoid overloading. In contrast, the Memory State Backend is extremely fast due to the fact that it keeps the state in memory.

In addition, it is encouraged for local development and debugging. This is the default backend used by Flink in case nothing is configured. The Memory State Backend should be used only experimentally, in local setups and cases with very small states as a result of the limitations.

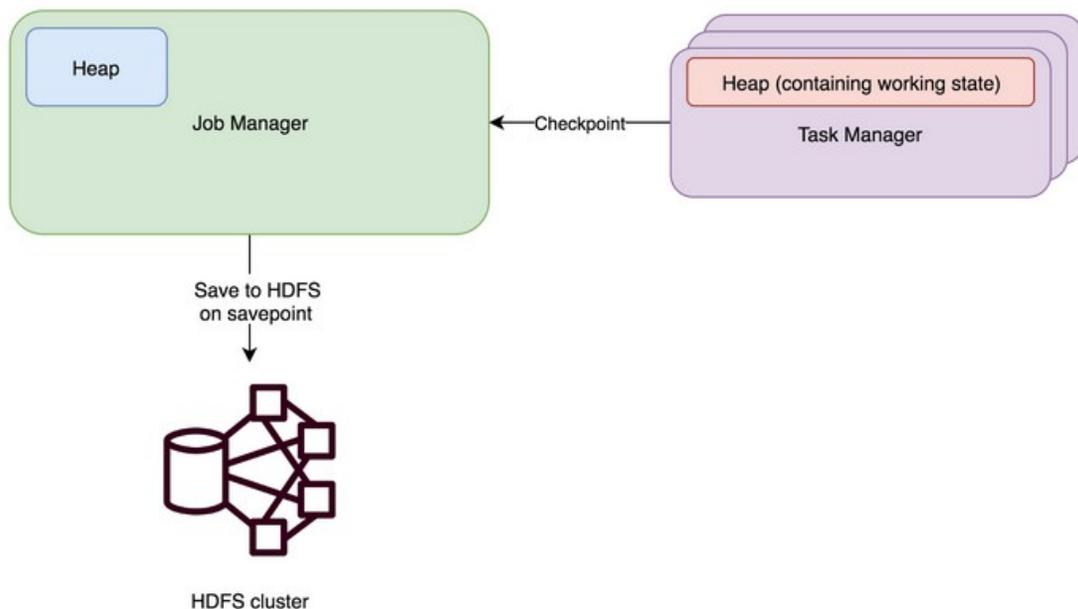


Figure 3: Memory State Backend diagram.

2.7.2 FileSystem State Backend [13,14]

FsStateBackend holds in-flight data of the current state in the TaskManager's memory but it stores the checkpoint on the filesystem (HDFS/S3) rather than job manager memory. JobManager's heap memory holds only useful metadata (in case of high availability). It is at your own risk to configure it properly in order to avoid large checkpoint metadata because the state backend stores small state chunks directly in metadata.

The Filesystem State Backend allows us to work with jobs with states more flexibly but is also limited by heap memory and is recommended to be used in cases with less data and high-performance requirements.

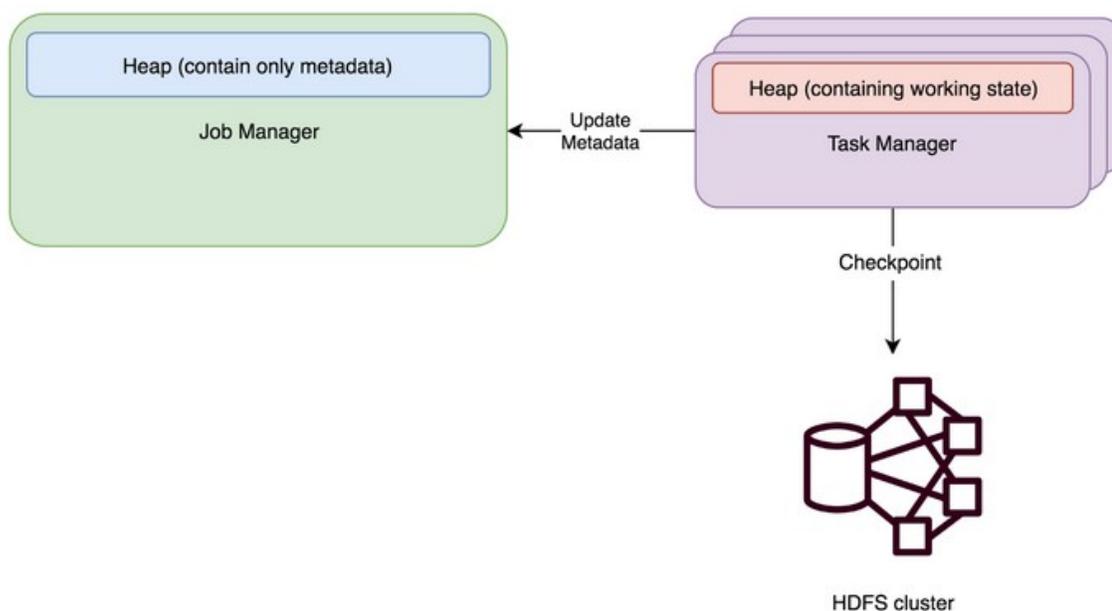


Figure 4: File System State Backend diagram.

2.7.3 RocksDB State Backend [15,14]

The RocksDB state Backend holds in each TaskManager's memory a small amount of data in a RocksDB database which is important for accessing into the FileSystem. The RocksDB creates checkpoints into a durable file system (like HDFS) directories which are defined by the user. Only a little metadata is stored in JobsManager's heap memory in case of high availability. It is the only FileSystem that offers incremental checkpoints which are essential to reduce checkpointing time due to the fact that incremental checkpoint record only changes between a previous and current checkpoint.

This type of State Backend is useful in case you have very large states and long windows because you have no limitations on storage owing to the fact that you are able to use all your disk storage.

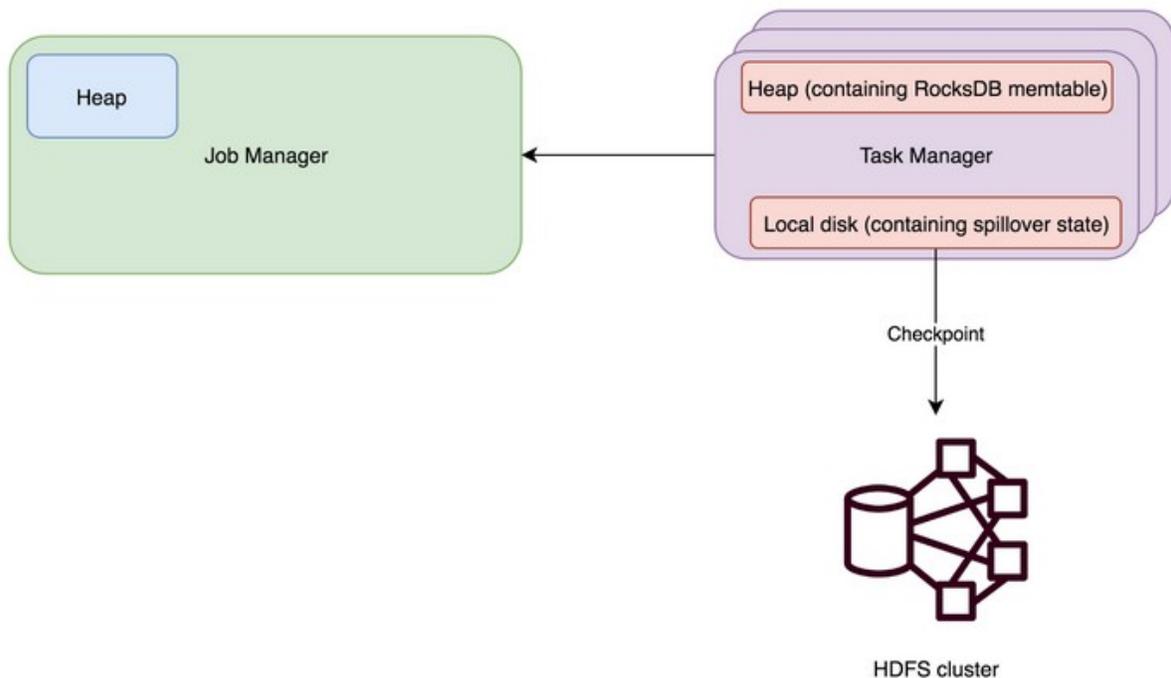


Figure 5: RocksDB State Backend diagram.

2.8 Unify binary format [16]

From the 1.9 version of Apache Flink and thereafter, a unify binary format across state backend savepoints is provided in order to be compatible in case of restart, giving you the ability for changes like backend. Up to 1.8 version of Flink this important ability is not provided and it is at your own risk to create your type serializers in case of changes or migration. The main goal of this change is to unify the savepoint format among all state backends for keyed states with the advantage of being more future-proof and applicable for new state backends.

“We propose to unify the binary layout for all currently supported keyed state backends to match the layout currently adopted by RocksDB, for all levels including the layout of contiguous state values of a key group as well as layout of individual state primitives.”

In Rocks DB it is not possible to know the number of (namespace, partition key) state value mappings when taking a snapshot. For this reason, RocksDB backend writes metadata along with the stream of contiguous state values in a key group. The MSB of the last state value of a keyed state is flipped, to indicate the last value of this state. When it finds this flipped bit it knows that the next value will be a state ID which indicates the next value is another new keyed state. It terminates when it reads an `END_OF_KEY_GROUP_MARK`.

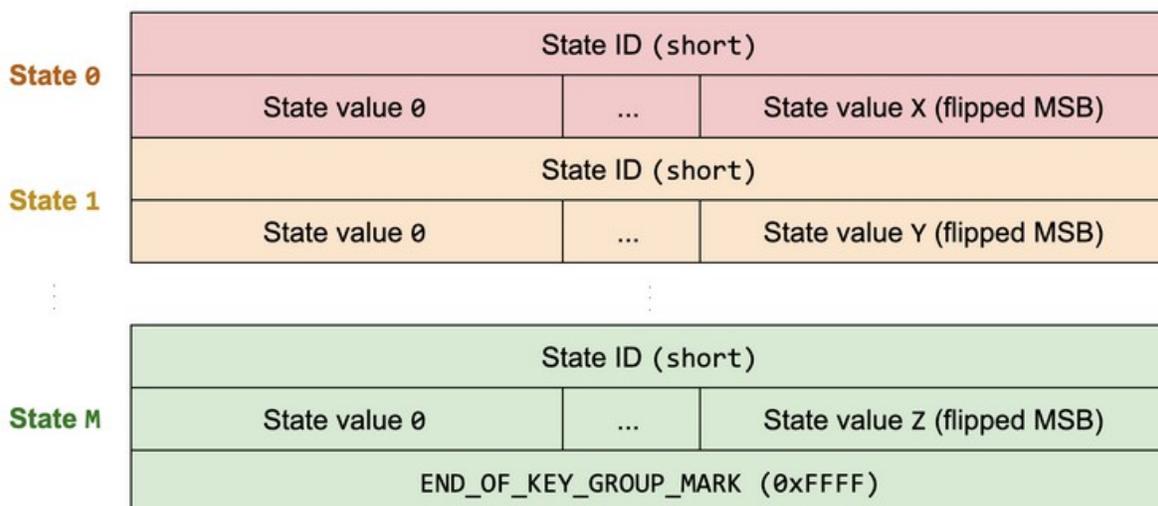


Figure 6: Unify binary format schema.

2.9 State Processor API [17,18]

State Processor API realised in 1.9 version of Apache Flink is an API that enables you to read, write and modify Savepoints and Checkpoints using the functionality of the DataSet API. It provides you with the ability to work with partitioned and non- partitioned state. State Processor API provides a public interface in order to load an existing savepoint with a command like:

```
ExecutionEnvironment bEnv = ExecutionEnvironment.getExecutionEnvironment();
ExistingSavepoint savepoint = Savepoint.load(bEnv, "hdfs://path/", stateBackend);
```

Additionally, you can access the state data by specifying the operator uid, the state name and the type information:

```
DataSet<Integer> listState = savepoint.readListState("uid", "state", Types.INT);
DataSet<Integer> unionState = savepoint.readUnionState("uid", "state", Types.INT);

DataSet<Tuple2<Integer, Integer>> broadcastState = savepoint.readBroadcastState("uid", "state", Types.INT, Types.INT);
```

Two more features which are provided by the State Processor API are the creation of a new savepoint or the extension of the current one. This very useful functionality provides you with the flexibility to interact with states and snapshots and enables you to change the main “body” of a savepoint. For example, it is possible to specify a new state backend and change the maximum parallelism. Furthermore, you can add multiple operators in a current savepoint or create a new one with your new features and write it anywhere you want.

These lines of code show how can you create a new Savepoint with specific State Backend and operators:

```
Savepoint
    .create(stateBackend, maxParallelism)
    .withOperator("uid", transformation)
    .withOperator(...)
    .write(path)
```

The following code shows how can you modify a Savepoint:

```
existingSavepoint
    .removeOperator(oldOperatorUid)
    .withOperator(oldOperatorUid, transformation)
    .write(path)
```

3. RapidMiner Studio

RapidMiner Studio[[19](#)] is a software designer application that enables everyone to build easily and quickly complex analytical processes in the data science sector providing a friendly user graphical interface. RapidMiner studio offers a wide range of extremely useful operators in order to combine them and make your own analytics designs as data preparation, machine learning, deep learning, and predictive analytics. Every process consists of operators and every operator is responsible for one and only task which takes as input the output of the previous operator and offers an output to the next one. The insertion of an operator into the design process is very simple and done with a “drag and drop” of the operator into the process. The next step that you have to make is to connect the new operator to the others and to configure it whether it is necessary. Finally, you have to press the “play” button and your process will be alive.

Furthermore, the flow design is represented internally in RapidMiner as a XML file and it is responsible for the graphical representation in the user interface. In addition, it provides you with some useful recommendations to correct your design and error message in the event of troubleshoot. It is worth mentioning that a debugging system is offered by RapidMiner studio and gives you the ability to set breakpoints between operators in order to supervise the workflow partially. Thus, the user has full control of the process and is able to design it as he wishes.

3.1 RapidMiner Extensions [[20](#)]

RapidMiner studio contains over 1.500 operations in order to give the opportunity to everyone to implement any professional data analysis process as data partitioning and market-based analysis. Furthermore, every tool that someone needs to process the data is included in it and is developed efficiently to offer the user high performance. Extensions can easily be imported into the standar version of the RapidMiner studio adding new operators and functionality such as the Rapood extension which connects the RapidMiner studio with the Hadoop cluster to push the calculations there from high analytics processes.

RapidMiner develops a new extension for streaming processing using Apache Flink and Apache Spark frameworks and in this diploma thesis we will

focus on Apache Flink implementation. Moreover, regarding the design implementation, the extension contains a StreamingNest operator which is actually the Flink Job. Inside any StreamingNest operator you can insert streaming operators. A wide range of streaming operators are offered to you from the extension and some of the most important are described below:

- **Kafka Source operator:** When you use a Kafka Source operator you have to specify the host, the port and the name of the Kafka topic. It reads a stream of data, at json format, and passes it to the next operator.
- **Kafka Sink operator:** The Kafka Sink operator takes as input a stream of data as json format and writes it into a Kafka Topic. Additionally, as in Kafka Source you have to specify the host, the port and the name of the Kafka topic.
- **Aggregate operator:** The most useful streaming processes (min, max, count, sum, avg) are implemented in this operator. The user must choose one of them in every single aggregate operator in order to create his own functionality. A stream is inserted to it and after a window time, which is specified by the user, an output stream is produced.
- **Join operator:** The Join Operator takes as input two data streams and joins them under a specified key. The produced stream has the values of both the input streams and the join key as json format.
- **Duplicate operator:** This operator takes as input a data stream and splits it into two new streams which are the same as the first. The Duplicate operator is very useful because it gives you the opportunity to process the same stream in different ways.

For example, a workflow is provided below in which there is a streaming Nest operator to initialize the Flink job.

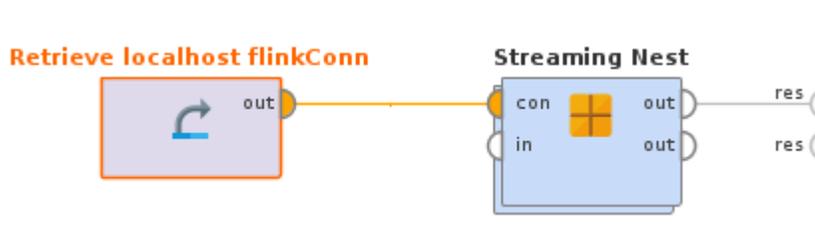


Figure 7: A workflow consists of one job.

Inside it, data is read from a kafka topic using a Kafka Source operator. The data passes in an Aggregate operator that calculates the sum of the value and finally it writes it in another kafka topic via a Kafka Sink operator.

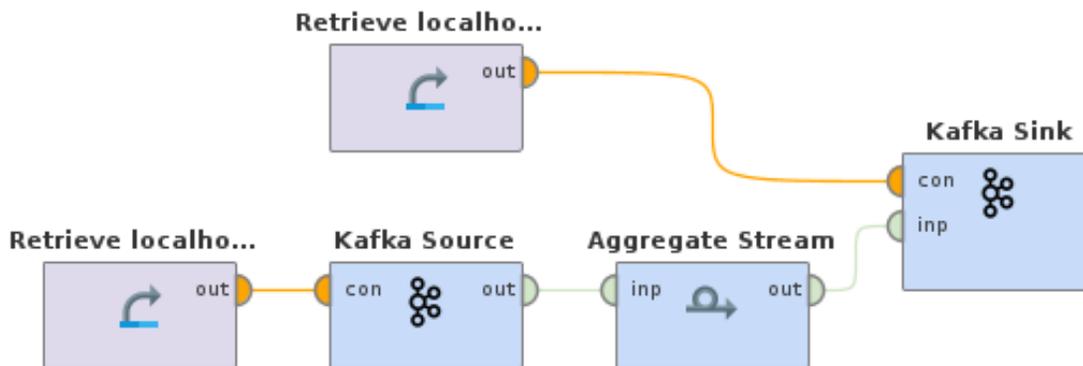


Figure 8: Implementation of the Streaming Nest operator.

4. Implementation

4.1 The Migration Algorithm

The implementation of the migration algorithm is related to the State Processor API because it uses its “under the hoop” classes and methods. The State Processor API requires , as referred in the documentation[18,19], arguments as the name of the State Descriptor and the Type Info which ,in many cases, is impossible to know about and has limited functionality. The Migration Algorithm provides the ability to merge or split savepoints , to change the parallelism and the absolute paths of a savepoint.

The Migration Algorithm takes as input a List of savepoint paths in which the savepoints exist, a List of operator UIDs in which will contain the new savepoint and the new savepoint path in which the produced savepoint will be written. Moreover, the algorithm copies the state file into the new savepoint directory and open it in order to access the data of operators which they are specified by the input UIDs. Then, it changes the absolute paths of the specified operators using the new one and copies the data of them without any change. Finally, it creates the new Metadata file using the new operators and writes it into the new savepoint directory.

4.1.1 Steps of the Algorithm

1. The Apache Flink hashes every operator UID using the murmur3_128 hash function to secure that every UID is distinct. In the first step the algorithm hashes the UIDs via murmur3_128 hash function in order to be able to compare them with the hashed UIDs of the savepoints.
2. In the second step, all the savepoint files are copied into the new savepoint directory, without the metadata file.

3. The savepoints are loaded into a Savepoint List using the SavepointLoader class. In this way we have access to every OperatorState.
4. In the fourth step, it merges all the OperatorStates into a new OperatorStates List in order to manage them in the next step.
5. Select which of the operator states will exist into the new savepoint. To do that we compare the operator UIDs from the list with the produced UIDs from the first step.
6. It changes the absolute paths of the States that will be included in the new savepoint.
 - 6.1. For every Operator state, it changes the raw and the managed state absolute paths.
 - 6.1.1. It gets every state as a List of keyed_state_handle
 - 6.1.2. From the keyed_state_handle List gets the delegate_state_handle as File_state_handle or ByteStream_state_handle.
 - 6.1.3. It constructs a new state_handle and copies the data of the previous one in order to avoid the State Descriptor, the BootstrapTransformation and the Type Info requirements. Additionally, it changes the absolute paths using the new savepoint directory.
 - 6.1.4. It constructs a new key_group_state_handle with the new absolute paths and deletes the old one.
 - 6.2. It adds the new State into the new operator states List.
7. A new Savepoint Metadata is created using the new operator states List and the new savepoint directory.
8. It gets the constructor of the ExistingSavepoint class and constructs the new Savepoint using the new Metadata file.
9. Finally, it copies back the state files into the old savepoint directories because they were deleted. That happens in case another job requires one or more savepoints which we have processed. Thus, the implementation does not take extra snapshots whether an old one exists.

Furthermore, by applying this technique we take a deep dive into Flink state in order to process it without data losses. We have no data losses because we do not get access to real data but we copy it as bytes and we change only the absolute paths in order to get access to different savepoint directories. Thus, if we combine the change of the absolute paths in a

savepoint together with the ability to merge or split operator states from different savepoints we are able to manage multiple savepoints in order to produce a new one with specific characteristics and states.

4.2 RapidMiner Studio

4.2.1 Stateful operators

In this segment it is important to say that the RapidMiner Studio is used for the code design and not for the operator implementation. The algorithm implementation is general and adaptable in any case irrespective of the job's implementation.

The RapedMiner extension does not offer stateful operators and the interaction with states is not possible, thus it is our responsibility to convert the non-stateful operators into stateful and set to a unique uid that is produced from the core automatically. Firstly, to make the operators stateful we add a ValueState in the main process of the operator. This state is in Json format, to be adoptable with the whole design which RapidMiner Studio provides and it contains all the useful information about the process. After that we are able to take a snapshot of the operator state and process it.

It is essential to say that every name of the operators which are added to the workflow is distinct. This fact gives us the opportunity to set this disting name as the unique ID of the process state. Thus, we have a unique ID generated automatically from the system and, in addition, we know exactly which id corresponds to which operator state due to the fact that the RapidMiner Studio enables us to supervise the design.

Additionally, we set the state Backend using the setSatateBackend instruction of Flink framework. It is essential to initialize the state Backend due to the fact that many clusters do not include it in the configuration file. Furthermore, when you set the state Backend, you have to define the checkpoint directory which is vital in case you want to take snapshots. It is not possible to take a snapshot of the state if a cluster configuration does not include the checkpoint directory or if you do not set it on your own.

4.2.2 Restart Implementation

What changes we have made to implement the restart and migration strategy in RapidMiner extension are described in this section. Because we have no full access to RapidMiner studio we use a Json file as input which contains some arguments which are vital for our design and produced easily from the core.

Json file fields:

- **restart**: type: boolean -> true if you want to restart a job
- **fromExistingSavepoint**: type: boolean -> true if you want to restart a job from an existing savepoint
- **AllowNonRestoreStates**: type: boolean -> allow to restore the state if any operator is missed
- **cancel job**: type: boolean -> true if you want to cancel the running job after the snapshot
- **checkpoint dir**: type: String -> the new checkpoint directory (it is important in case that the checkpoint directory is not configured)
- **jobs**: type: List< job name, job ids, new savepoint path, existing savepoint paths, host, port, target directory >
 - **job name**: String-> the name of the job that you want to restart (if there is only one you can skip this field by set as Default value "")
 - **job ids**: List <String> -> the job ids from which you get the savepoints
 - **new savepoint path**: String -> the new savepoint path (It fails to restart if it exists)
 - **existing savepoint paths**: List<String> -> the existing savepoint paths from which you restart the job (in case the "**fromExistingSavepoint**" is true)
 - **host**: type:String -> the host name of the cluster in which the job ids runs
 - **port**: type: Integer -> the port of the host cluster in which the job ids runs
 - **target directory**: type: String -> the directory in which the snapshots will be saved (it is important in case the savepoint directory is not configured)

In case of restart we check if there are existing savepoints that take as input arguments into the main migrate algorithm. If we do not already have the savepoints to create the new ones, the restart scenario takes place and takes care of this. In this case, savepoints are taken for every job id, which are included in the json file, using the REST API of Flink and the job canceled if it is necessary. The RequestBody of the POST and GET requests, which are used to trigger the savepoint and retrieve the savepoint path, is completed via information that the json file provides. When it has all the appropriate savepoint paths, the migrate algorithm is called and takes as input arguments the savepoint paths along with all the operator uids, which exist in this StreamingNest operator, and the new savepoint directory.

5. Experimental Execution

In this section we provide some examples and we will see how to split, merge and migrate jobs which are designed in RapidMiner studio and how our implementation is adapted in any required situation.

5.1 Split-Merge example

Firstly, we have designed a simple workflow using a Flink Retrieve connection and a Streaming Nest operator. A KafkaSource operator connected via a Kafka Retrieve is included into the Streaming Nest operator. Its output connects with the input of an Aggregate Stream operator which calculates the sum of the input. Subsequently, its output is connected with another Aggregate Stream operator which calculates the sum of the input again. Finally, the output of the second Aggregate Stream operator is connected with a Kafka Sink operator which writes the output in a topic. The connection of the Kafka Sink operator being via a Kafka Retrieve connection.

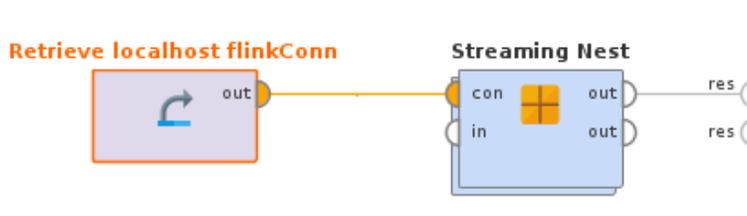


Figure 9: The Streaming Nest operator to create the job.

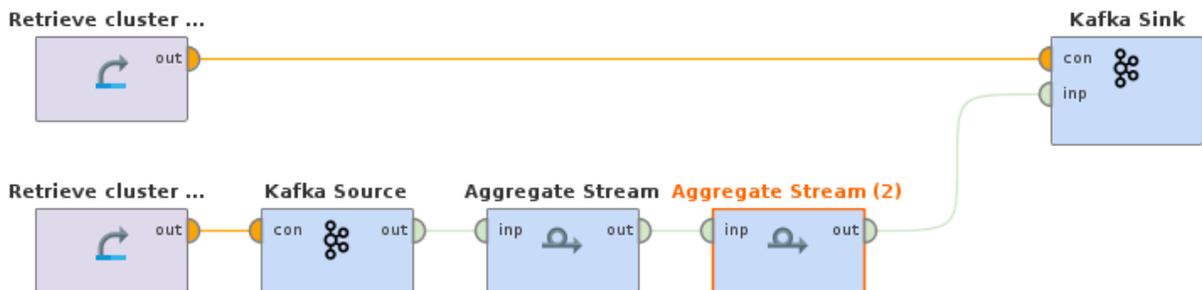


Figure 10: A workflow consists of two Aggregate Stream operators

After the signing of the workflow we have to configure the Json file. We do not want to restart our workflow using other jobs, thus both the restart parameter and the “fromExistingSavepoint” will be “false”. In addition, we must configure the checkpoint dir correctly. We don't care about the other parameters.

```
{
  "restart": "false",
  "fromExistingSavepoint": "false",
  "cancel job": "false",

  "jobs": [
  ],

  "checkpoint dir": "hdfs://45.10.26.123:9000/apps/savepoint/checkpoints",
  "AllowNonRestoreStates": "true"
}
```

Figure 11: The Json file to run a job without restart.

We submit the job on the cluster and produce some data via a Kafka producer into the Source topic.

Job Name	Start Time	Duration	End Time	Tasks	Status
job3	2020-10-27 16:35:50	31s	-	3 / 3	RUNNING

Figure 12: The submitted job on the cluster

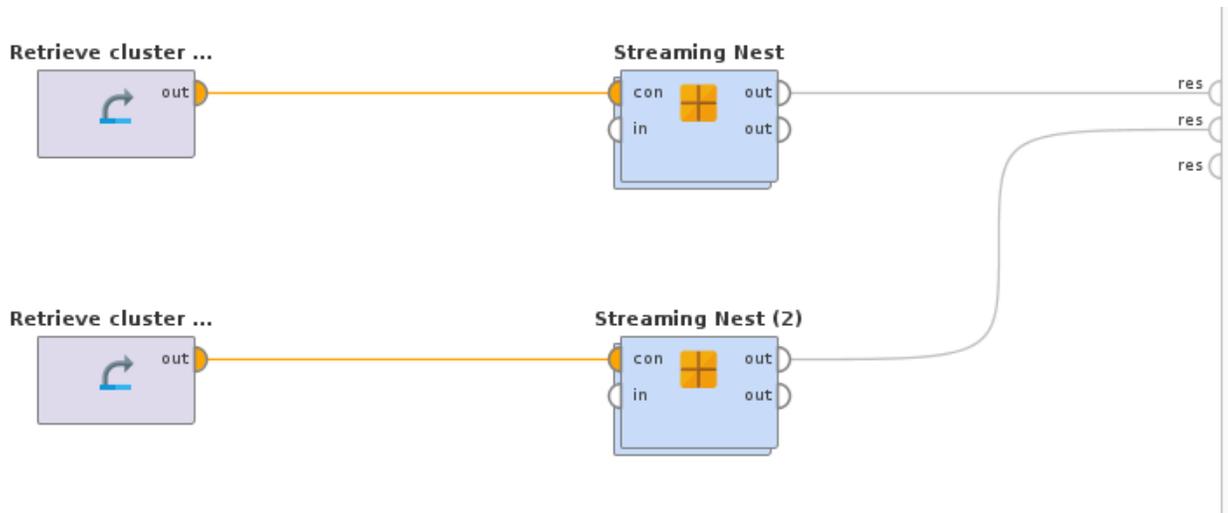


Figure 14: A workflow consists of two jobs.

The new workflows contain the same Aggregate Stream operators as the first one. It is vital in case of restart that the operator's name has to be the same as the first due to the fact that the name is the UID of the operator. The Kafka Source (2) operator reads data from the topic of the Kafka Sink operator.

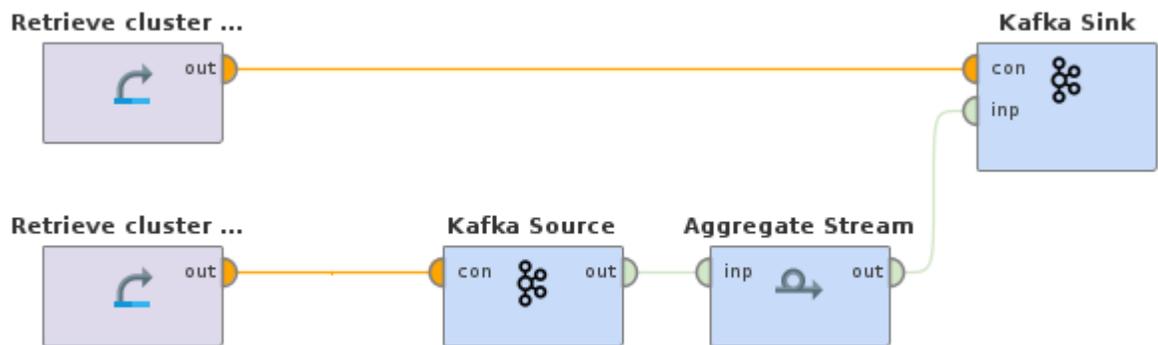


Figure 15: Implementation of the first Streaming Nest operator.

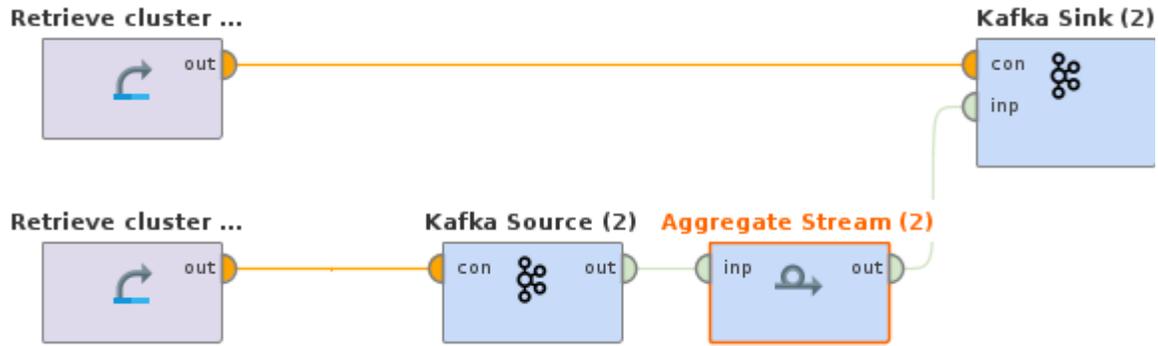


Figure 16: Implementation of the second Streaming Nest operator.

Thereafter, we configure the Json file to restart the new workflows using the states of the first one. Thus the restart flag has to be “true”, the fromExistingSavepoint “false”, in case we want to cancel the running job, the cancel job has to be “true” , otherwise “false”, the AllowNonRestoreStates has to be “true” and the checkpoint dir must be configured correctly. In the jobs field we have two elements, each of them must have the name of our job as job name, as job ids we have one element with the job id of the running job. Additionally, we have to configure the host, the port and the target directory properly.

```

{
  "restart": "true",
  "fromExistingSavepoint": "false",
  "cancel job": "true",

  "jobs": [
    {
      "job name": "job1",
      "job ids": ["7b65f3be38316109440253f2b10b34ca"],
      "new savepoint path": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test1",
      "port": "8081",
      "host": "clu01.softnet.tuc.gr",
      "target directory": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints",
      "existing savepoint paths": [""]
    },
    {
      "job name": "job2",
      "job ids": ["7b65f3be38316109440253f2b10b34ca"],
      "new savepoint path": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test2",
      "port": "8081",
      "host": "clu01.softnet.tuc.gr",
      "target directory": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints",
      "existing savepoint paths": [""]
    }
  ],

  "checkpoint dir": "hdfs://45.10.26.123:9000/apps/savepoint/checkpoints",
  "AllowNonRestoreStates": "true"
}

```

Figure 17: The Json file in order to split a workflow to two jobs.

Job Name	Start Time	Duration	End Time	Tasks	Status
job2	2020-10-27 16:49:21	13s	-	2 / 2	RUNNING
job1	2020-10-27 16:48:46	49s	-	2 / 2	RUNNING

Completed Job List

Job Name	Start Time	Duration	End Time	Tasks	Status
job3	2020-10-27 16:35:50	12m 47s	2020-10-27 16:48:37	3 / 3	CANCELED

Figure

18: The submission of the new jobs.

Afterwards, we will provide an example in which we will merge the previous two jobs into the first one.

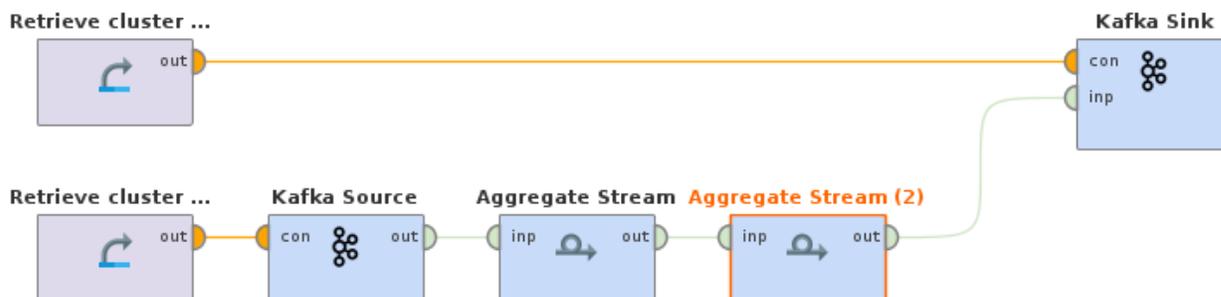


Figure 21: The merged workflow that consists of two Aggregate operators

Thus, we have to configure the Json file properly in order to take savepoints from every running job. We need the restart flag to be “true”, the fromExistingSavepoint “false”, the cancel job or has to be “true” or “false” in order to cancel the jobs or not, the checkpoint dir has to be configured with the checkpoint directory properly and the AllowNonRestoreStates must be “true”. Additionally, the jobs field consists of one element with the job name as the job name of the new job and the job ids consists of the job ids of the running jobs. The other fields as the host, the port, the new savepoint path and the target directory have to be defined correctly.

```
{
  "restart": "true",
  "fromExistingSavepoint": "false",
  "cancel job": "true",
  "jobs": [
    {
      "job name": "job3",
      "job ids": ["9f30f94253d3926bef064bb577177f7f", "f507749506ff4319dea00fa4819fb2d7"],
      "new savepoint path": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test3",
      "port": "8081",
      "host": "clu01.softnet.tuc.gr",
      "target directory": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints",
      "existing savepoint paths": [""]
    }
  ],
  "checkpoint dir": "hdfs://45.10.26.123:9000/apps/savepoint/checkpoints",
  "AllowNonRestoreStates": "true"
}
```

Figure 22: The json file in order to merge two jobs.

In addition, the savepoint of the new workflow are generated by the implementation into the new savepoint path which is defined into the Json file and the new job are submitted on the cluster.

```
john@john-All-Series:~$ hadoop fs -ls hdfs://45.10.26.123:9000/apps/savepoint/savepoints
Found 6 items
drwxr-xr-x - softnet supergroup 0 2020-10-27 16:49 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/savepoint-7b65f3-c023312afb96
drwxr-xr-x - softnet supergroup 0 2020-10-27 16:57 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/savepoint-9f30f9-009142cac8b3
drwxr-xr-x - softnet supergroup 0 2020-10-27 16:57 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/savepoint-f50774-1c9b3bb9ee83
drwxr-xr-x - john supergroup 0 2020-10-27 16:48 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test1
drwxr-xr-x - john supergroup 0 2020-10-27 16:49 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test2
drwxr-xr-x - john supergroup 0 2020-10-27 16:57 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test3
john@john-All-Series:~$ hadoop fs -ls hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test3
Found 3 items
-rw-r--r-- 3 john supergroup 1956 2020-10-27 16:57 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test3/1f60ee25-461f-4fb5-a3a6-d280334df8a9
-rw-r--r-- 3 john supergroup 1940 2020-10-27 16:57 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test3/9c914940-2323-4b24-9dfe-0d54a2cc7dc8
-rw-r--r-- 1 john supergroup 2412 2020-10-27 16:57 hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test3/_metadata
```

Figure 23: The generated files on hdfs for the job1.

Job Name	Start Time	Duration	End Time	Tasks	Status
job3	2020-10-27 16:57:44	2m 41s	-	3 / 3	RUNNING

Completed Job List

Job Name	Start Time	Duration	End Time	Tasks	Status
job2	2020-10-27 16:49:21	8m 5s	2020-10-27 16:57:26	2 / 2	CANCELED
job1	2020-10-27 16:48:46	8m 39s	2020-10-27 16:57:25	2 / 2	CANCELED

Figure 24: The submission of the new job and the cancellation of the previous ones.

Furthermore, some data is produced by a Kafka producer into the Kafka Source operator topic in order to check the results which are written into the Kafka Sink operator topic. The results continue the sums of the previous jobs with no data losses as we can see on the screenshot below.

```
ibaikousis@clu04: /usr/hdp/current/kafka-broker
File Edit View Search Terminal Help
c.gr:6667,clu06.softnet.tuc.gr:6667 --topic test1
>{"word": "a", "value1": "1"}
ibaikousis@clu04: /usr/hdp/current/kafka-broker
File Edit View Search Terminal Help
{"SUM-SUM-value1": "33.0", "word": "a"}
{"SUM-SUM-value1": "43.0", "word": "a"}
{"SUM-value1": "11.0", "word": "a"}
{"SUM-SUM-value1": "66.0", "word": "a"}
```

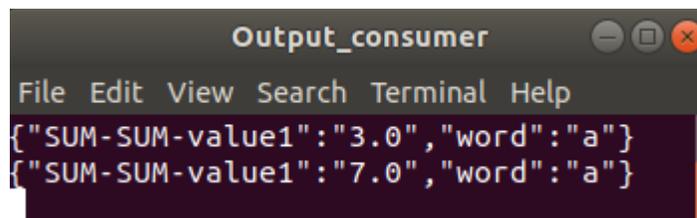
Figure 25: The results which are produced by the new job.

Summarizing, we provided an extended example which includes many cases in order to prove the correctness of the implementation. The cases of the split and merge are checked by the execution example and the results of this are provided on the screenshots successfully.

5.2 Migration example

Finally, we provide a simple example regarding migration of a state in order to test it between different clusters.

Firstly, we want to migrate a job from the localhost Flink setup on the cluster of the Technical University of Crete. We submit a job that contains two `sum` Aggregate Stream operators connected. Then, we produce some data into a Kafka topic of the localhost Flink setup:



```
Output_consumer
File Edit View Search Terminal Help
{"SUM-SUM-value1": "3.0", "word": "a"}
{"SUM-SUM-value1": "7.0", "word": "a"}
```

Figure 26. The results of the produced data on localhost cluster.

In order to migrate the state, we made some changes in the `Json` file. First of all, we specify the host and the port of which the job is running. Moreover, we set the new checkpoint and savepoint directories. The `Json` file provided below:

```

{
  "restart": "true",
  "fromExistingSavepoint": "false",
  "cancel job": "false",
  "jobs": [
    {
      "job name": "job3",
      "job ids": ["d0845026604c64d52e319704318a916b"],
      "new savepoint path": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints/test62",
      "port": "8081",
      "host": "localhost",
      "target directory": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints",
      "existing savepoint paths": [""]
    }
  ],
  "checkpoint dir": "hdfs://45.10.26.123:9000/apps/savepoint/checkpoints",
  "AllowNonRestoreStates": "true"
}

```

Figure 27: The Json file to migrate the state.

After the execution the new job is submitted on the new cluster successfully and some data is produced by us to check that the migration completed with no data losses.

```

ibaikousis@clu04: /usr/hdp/current/ka...
File Edit View Search Terminal Help
{"SUM-SUM-value1": "12.0", "word": "a"}

```

Figure 28: The results which are produced on the Technical University cluster.

As we can see on the screenshot above, the sum continues to increase after the migration of the state on the new cluster.

Conclusion

In this diploma thesis we proposed a Migration Algorithm in Apache Flink framework with the ability to split, merge and rescale operator states. A job can easily migrate on another cluster without data losses and no extra requirements, as the state descriptor and Type info, using our implementation. The algorithm is based on the State Processor API of Flink using many methods and classes of it.

Additionally, we use the RapidMiner Studio in order to apply our algorithm due to the fact that it gives the opportunity to design workflows quickly and easily. Changes have been added in the extension by us in order to automate the restart process using a Json file as input and the REST API which is provided by Flink.

In the final step, we provided execution examples which covers a wide range of use cases of the state migration in combination with the merging and the splitting of workflows. We analyzed ito a extended the steps of the design and the configuration of the Json file in order to restart any workflow using proper parameters.

References

[0] Data science: https://en.wikipedia.org/wiki/Data_science

[1] Apache Flink: <https://docs.cloudera.com/csa/1.2.0/flink-overview/topics/csa-flink-overview.html>

[2] Flink APIs: <https://flink.apache.org/flink-applications.html>

[3] DataSet API: <https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/batch/>

[4] DataStream API: https://ci.apache.org/projects/flink/flink-docs-stable/dev/datastream_api.html

[5] What is a DataStream : https://ci.apache.org/projects/flink/flink-docs-stable/dev/datastream_api.html#what-is-a-datastream

[6] Programs and Dataflows: <https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/programming-model.html#programs-and-dataflows>

[7] Parallel Dataflows: <https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/programming-model.html#parallel-dataflows>

[8] Parallel Execution : <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/parallel.html#parallel-execution>

[9] Stateful Operators : <https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/programming-model.html#stateful-operations>

[10] Checkpoints and Savepoints :
<https://cwiki.apache.org/confluence/display/FLINK/FLIP-47%3A+Checkpoints+vs.+Savepoints>

[11] State Backend : https://ci.apache.org/projects/flink/flink-docs-master/ops/state/state_backends.html

[12] The MemoryStateBackend : https://ci.apache.org/projects/flink/flink-docs-master/ops/state/state_backends.html#the-memorystatebackend

[13] The FsStateBackend : https://ci.apache.org/projects/flink/flink-docs-master/ops/state/state_backends.html#the-fsstatebackend

[14] Here's How Apache Flink Stores Your State data:
<https://towardsdatascience.com/heres-how-flink-stores-your-state-7b37fbb60e1a>

[15] The Rocksdbstatebackend: https://ci.apache.org/projects/flink/flink-docs-master/ops/state/state_backends.html#the-rocksdbstatebackend

[16] Unify binary format : <https://cwiki.apache.org/confluence/display/FLINK/FLIP-41%3A+Unify+Binary+format+for+Keyed+State>

[17] State Processor API : https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/libs/state_processor_api.html

[18] FLIP-43: State Processor API: <https://cwiki.apache.org/confluence/display/FLINK/FLIP-43%3A+State+Processor+API>

[19] RapidMiner Studio: <https://rapidminer.com/products/studio/feature-list/>

[20] RapidMiner Extensions:
<https://marketplace.rapidminer.com/UpdateServer/faces/category.xhtml?categoryId=4>