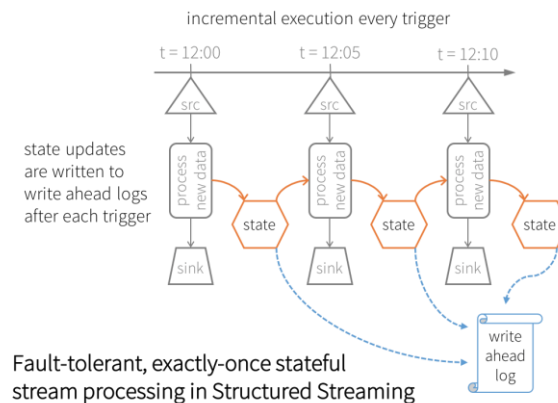


## 6.2 Runtime Adaptation – Spark Clusters

### 6.2.1 Background on Spark Checkpointing Mechanisms

Stateful Stream Processing is stream processing with state. While many operations in a physical simply look at one individual tuple at a time, some operators remember information across multiple tuples (for example window operators). These operators are called stateful. The main difference with stateless processing is that tuples can combine and are not independent. State can be thought of as collection of keys and current value pairs. Example categories of stateful Spark Structured Streaming operators are<sup>17</sup>: (i) Stream Aggregation, (ii) Arbitrary Stateful Streaming Aggregation, (iii) Stream-Stream Join, (iv) Stream (De)duplication, (v) Streaming Limit operators.



**Figure 27: Structured Streaming - Incremental Stateful Query Updates.**<sup>18</sup>

During the execution of stateful streaming queries (parts of physical workflows in terms of INFOR), SparkSQL internally maintains an intermediate state for fault tolerance.


Fault tolerance is achieved via checkpoints. The basic concept of checkpointing does not differ much from the conventional checkpointing mechanism used in database systems. The intermediate state is stored versioned inside Spark executors' memory and also backed up in a user defined checkpoint location using write-ahead logs (Figure 27). The checkpoint location is a path to a fault-tolerant file system like HDFS. Every query with defined checkpointing parameters reads the previous state and saves the updated one in memory and in the write-ahead log. In case of failure, the latest completed state restores from the checkpoint location, and the query resumes its execution from that point on. The Structured Streaming API ensures exactly once guarantees for stateful stream processing when input sources are replayable, and streaming sinks are idempotent to handle reprocessing.

To initiate streaming computation the `Datastream.writeStream()` function is used which accomplishes that by returning the required `DataStreamWriter` interface with the following properties:

- Output sink information: Specifies the sink format (file, Kafka, console, memory), output location path, etc
- Output mode: Specifies the content written to the external sink. Append is the default mode. Check the Programming Model section for more details regarding the available modes.
- Query name: Specifies an ID-like distinct query name. This property is optional.
- Trigger interval: Specifies the trigger interval. This property is optional, and if absent, the system checks for availability of new data right after the completion of previous processing

<sup>17</sup> <https://jaceklaskowski.gitbooks.io/spark-structured-streaming/content/spark-sql-streaming-stateful-stream-processing.html>

<sup>18</sup> <https://databricks.com/blog/2017/05/08/event-time-aggregation-watermarking-apache-sparks-structured-streaming.html>

 <p>Project supported by the European Commission Contract no. 825070</p>	<p><b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b></p>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public

- Checkpoint location: Specifies a directory location in an HDFS-compatible, fault-tolerant file system to store all checkpoint information. This property is optional as well.

To start executing an operator, the code (i.e., generated by RapidMiner Studio in INFOR's Architecture – Deliverable D4.2) must also use the `start()` method, which produces a final `StreamingQuery` object. It can be used later for monitoring purposes. In many cases, to prevent the execution from stopping while the running query is alive, the method `awaitTermination()` is called. An exemplary query is presented in the following snippet.

```
val query = example
  .writeStream()
  .outputMode("append")
  .queryName("randomness")
  .format("json")
  .option("checkpointLocation", "randomPath/dir")
  .option("path", "NewRandomPath/dir")
  .start();
```

In the above code snippet, the following line performs the checkpointing:

```
.option("path", "NewRandomPath/dir")
```


Each query defines a checkpoint location, and while the query is alive, Spark constantly writes metadata to the checkpoint path (in HDFS). The specified checkpoint location stores four types of data:

- Source Files: that maintain information about all input sources processed in the query. For instance, a Kafka source preserves information about partitions and offsets.
- Offsets: Files that maintain offset details for each particular batch (watermark, timestamp, configurations). Internally it is represented as  
`org.apache.spark.sql.execution.streaming.OffsetSeqLog`
- Commits: Files that contain commit information about batch metrics. Internally it's represented as  
`org.apache.spark.sql.execution.streaming.CommitLog`
- Metadata: A File that contains metadata information regarding the query as a query ID. Internally it's represented as `org.apache.spark.sql.execution.streaming.CommitMetadata`
- State: These files exist only after stateful stream processing. State data is stored as LZ4-compressed objects (delta and snapshot files) that contain key,value pairs for each stored state.

## 6.2.2 Migrating State between Spark Clusters

The Connection Component of the INFOR Architecture (Deliverable D4.2) is accountable for the connection with each computer cluster and this connection is achieved via REST APIs. In Spark's case, Apache Livy REST API is used. Therefore, when we refer to changes in Spark configuration, we refer to changes in Apache Livy configuration or the request sent in Spark. Apache Livy accepts requests as .jar files carrying information about Spark cluster configurations and the operators executed in the requested workflow.

The logical operators provided by the Streaming Extension of the RapidMiner studio that are interpreted in stateful operators upon translating a logical workflow to a physical one are: (i) only Aggregate Stream, (ii) Join Stream, (iii) Connect Stream and (iv) Union Stream. So, when we refer to state migration we refer to these operators upon they are included in a given workflow. It is essential to notice that all of these Operators follow the Sliding Window Model, and their State perseveres for the duration of window time. The user determines the duration of the window.

 Project supported by the European Commission Contract no. 825070	<b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public

A critical detail is each the name of each such operator. The RapidMiner Studio does not allow two instances of the same operator to have the same name in a given workflow. That means that all the operators in the same RapidMiner process possess unique names which can be used as unique identifiers (UIDs) for storing and retrieving the state of the respective operators. This is a fundamental observation that we exploit as will be explained shortly.

When the Optimizer assigns part of a workflow to a Spark cluster, either the first time a workflow is submitted or upon a runtime adaptation process, a new Spark Job (Streaming Nest with Spark connection in RapidMiner Studio) must be submitted. In case of a migration process, the first step of its physical execution relies on processing a custom JSON file we define with critical parameters on how the migration process should be performed.

The details and structure of the aforementioned JSON file are as follows [Kalog20]:


restart

- Data Type: Boolean/String(\true", \false" values allowed)
- Description: the job restarts after a migration procedure (true) or it is the initial physical plan for that part of the workflow (false)

jobs

- Data Type: List
- Description: Contains multiple different jobs with different properties as follows:
  - job name
    - Data Type: String
    - Description: The name of the job that the properties below apply
  - checkpoint location
    - Data Type: String
    - Description: Full checkpoint path storing all required information for restarting
  - merge
    - Data Type: Boolean/String(\true", \false" values allowed)
    - Description: Flag property that Enables/disables merge jobs
  - merge jobs
    - Data Type: comma-delimited Strings
    - Description: When enabled, moves all the available (full) paths to the checkpoint location
  - remote
    - Data Type: Boolean/String(\true", \false" values allowed)
    - Description: Flag property that Enables/disables remote connection
  - remote connection
    - Data Type: String in the format \host:port"
    - Description: When enabled, initiates the new Spark job in the given configuration
  - remote location
    - Data Type: String
    - Description: Copy the contents of the checkpoint location to a different DFS location (full path) and use that as a new checkpoint location. The property is used only when the checkpoint location already exists and will not be initiated in this execution. The property is disabled by leaving its value empty \ " or by removing the property altogether

All job name properties included in the JSON file are scanned, and all the related properties of the job mentioned in JSON and matching the new, under submission, job name are fetched [Kalog20]. The first property to investigate is the checkpoint location. This property is used in the migration algorithm's last steps but needs to be examined first for appropriately setting the property in the code of the job under submission. Checkpoint location points to the full HDFS path, where all the information required for the examined part of the workflow to restart, after migration, will be stored. More specifically, each stateful operator creates a subdirectory to store its state by utilizing the operator's name (UID as mentioned previously). However, this unique operator name cannot directly apply to name a directory due to HDFS name constraints. According to HDFS documentation, all characters in URLs that are not a-z, A-Z, 0-9, '-', '.', '\_' or '/' must first be converted to URL encoding and so plenty of special characters including spaces heavily adopted by users will cause issues. For that purpose, we use a polynomialRollingHash function to transform the UID into a unique ID number [Kalog20].

 <p>Project supported by the European Commission Contract no. 825070</p>	<p><b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b></p>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public

## JSON property file snippet example [Kalog20]:

```
{
  "restart": "true",
  "jobs":[
    ...,
    {
      "job name":"newOperators",
      "checkpoint location": "hdfs://45.10.26.123:9000/apps/",
      "merge":"false",
      "merge_jobs":["hdfs://45.10.26.123:9000/apps/Checkpoint/"],
      "remote":"true",
      "remote_connection":"45.10.26.123:58090"
    },
    ...
  ]
}
```

The target path responsible to store each operator's state follows the pattern "checkpoint location \polynomialRollingHash(operatorName)". By inspecting the hash function result, RapidMiner Studio or the Optimizer (if needed) will not be able to distinguish the initial operator names (UIDs) between them. To allow other components or applications to retrieve the UID of each operator, when an operator name contains parentheses (common in RapidMiner Studio), the contents between them concatenate with the hash result using underscore to form the target path. For example, if operatorName="SKtest(mytest)" and polynomialRollingHash(operatorName)="9123223", the final path for that operator will be checkpoint location\9123223\_mytest\. In case this target path already exists, the state recovers, migration completes and execution resumes for the respective part of the newly prescribed physical workflow. In case the target path does not exist, Spark creates a new target path, and execution starts while preserving state information. When in production and not in debug mode, an additional directory is created with the formatted checkpoint location\polynomialRollingHash(operatorName) RES to store the result files produced during the execution.


On the other hand, remote and remote connection properties take effect while handling Spark configuration. When the remote property is enabled ("true"), remote connection is used. Remote connection overwrites the current connection configuration by setting a new host and port. Remember that remote connection property follows the pattern "host:port". This property is used to allow each job to migrate in another, remote Spark cluster.

merge, and merge jobs properties can be applied to concatenate directories in the same HDFS. In case the merge property is enabled("true"), all the directories included in the merge job property move their content to the checkpoint location path.

Finally, the remote location property copies the checkpoint location's contents to a different HDFS, i.e., of the remote cluster where the part of the physical workflow migrates. After copying all the contents to the new HDFS, it sets the checkpoint location = remote location to restart the workflow after migration with state information directly derived from that location. Note that the checkpoint location directory should already exist and not get created for the first time during this execution. When the value of this property is empty " " or does not exist at all, it is ignored.

Remarkably, by the way we execute the migration process we have no loss of state. This is because we manage to simply copy directories and files as bytes, and we change only the absolute paths to point to remote hosts.

Algorithm 5 sums up all steps followed during migrating parts of physical workflows (and cancelling old, submitting new jobs) and properties analyzed above.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p><b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b></p>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public

## Algorithm 5: State Migration Algorithm in Apache Spark

```


1 Properties job_properties = {job_name, checkpoint_location, remote_location,
   merge, merge_jobs, remote, remote_connection}
2 Input json_config = {jobName |  $\Phi(\text{jobName}) \subseteq \text{job\_properties}$  }
3 foreach job do
4   if cur_job_name exists in json_config then
5     fetch cur_job_properties  $\leftarrow \Phi(\text{cur\_job\_name})$ 
6     if remote then
7       | Create job in host:port
8     end
9     if merge then
10      | foreach merge_jobs directory do
11        | move directory into checkpoint location
12      | end
13    end
14    if remote_location exists || not empty then
15      | Copy checkpoint location to remote_location
16      | Set checkpoint_location  $\leftarrow$  remote_location
17    end
18    foreach stateful operator do
19      | if  $\backslash \text{checkpoint\_location} \backslash \text{polynomialhash}(\text{uniqueOperatorName})$  exists
20        | then
21        | | restore state from directory
22      | else
23      | | create directory and store state
24      | end
25      | continue execution
26    end
27 end

```

Algorithm 5: State Migration Algorithm in Apache Spark [Kalog20]

### 6.2.3 Discussion on Technical Issues while Migrating Spark State

During state migration in Spark, remote location property induces massive overhead slowing down the execution of runtime adaptation. Recall that the remote location property copies an HDFS directory's contents into a remote, new HDFS directory. The checkpointing mechanism in the Structured Streaming API produces thousands of small files to be stored on these directories. On the other hand, HDFS is optimized to operate under large files and thus large block sizes. Therefore, small checkpoint files of few KBs produced by Spark Structured Streaming, upon being stored in blocks of MBs each: (i) unnecessarily occupy large space without using it, (ii) cause lots of seeks during migration which significantly increases the time it takes to perform a runtime adaptation step.

 <p>Project supported by the European Commission Contract no. 825070</p>	<p><b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b></p>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public

Some proposed solutions [Kalog20], incorporate different compaction techniques to handle and discard some of the small files, use of different DFS (HopsFS claims that is designed on top of HDFS to tackle small file issue) or even use of a different type of storage (such as HBase, Cassandra, ScyllaDB).

## 6.3 Runtime Adaptation – Flink Clusters

### 6.3.1 Background on Flink Savepoints and Checkpointing Mechanisms

Similarly to Spark, Apache Flink provides a flexible fault tolerance mechanism based on distributed checkpoints. Complementarily to checkpoints, savepoints are state snapshots kept by a mechanism which is not automated like checkpoints<sup>19</sup>. A savepoint is a consistent image of the execution state of a streaming job, created via Flink's checkpointing mechanism. A Flink program can use savepoints to stop-and-restart, split, merge or update the configuration of Flink jobs. Savepoints consist of two parts: a directory with (large, holding the state) binary files on persistent storage (i.e., HDFS, S3 etc) and a (relatively small) meta data file. The metadata file of a savepoint contains (primarily) pointers to all files on persistent storage that are part of the savepoint, in form of relative paths.

As explicitly described in Flink's documentation, savepoints are different from checkpoints in a similar way that backups are different from recovery logs in traditional database systems. The primary purpose of checkpoints is to provide a recovery mechanism in case of unexpected job failures. A checkpoint's lifecycle is managed by Flink, i.e. a checkpoint is created, owned, and released by Flink, without user code interaction apart from enabling checkpoints. As a method of recovery and being periodically triggered, two main design goals for the checkpoint implementation are i) being as lightweight to create and ii) being as fast to restore from as possible. Optimizations towards those goals can exploit certain properties, e.g. that the job code does not change between the execution attempts. Checkpoints are usually dropped after the job is cancelled (except if explicitly configured as retained checkpoints). In contrast to all this, savepoints are triggered, owned, and deleted by the application. Their use-case is for planned, manual backup and resume.

Flink's state backed has undergone considerable changes in the latest version of the Big Data platform. However, the legacy `FsStateBackend` used to maintain checkpoints and savepoints in HDFS is equivalent to using `HashMapStateBackend` and `FileSystemCheckpointStorage` in the latest version.

Therefore, for the purposes of runtime adaptation over Flink clusters in the scope of INFOR, we work with savepoints. Useful for our purposes is also the State Processor API of Flink. The State Processor API introduced in Flink v1.9, is an API that enables CRUD (Create/Read/Update/Delete) operations on savepoints and checkpoints using Flink's DataSet API. It allows working both with partitioned and non-partitioned state and provides a public interface to load an existing savepoint with a command like:

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


Additionally, Flink programs can access the state data by specifying the operator UID, the state name and the type information. Note that the UID functionality described for Spark is also useful here as UIDs for streaming operators pass as parameters to savepoints as shown below:

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

DataSet<Integer> listState =
    savepoint.readUnionState<>(" uid", "union-state", Types.INT);

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

<sup>19</sup> <https://ci.apache.org/projects/flink/flink-docs-master/docs/ops/state/savepoints/>

 <p>Project supported by the European Commission Contract no. 825070</p>	<p><b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b></p>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public



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 as it provides increased flexibility to interact with states and snapshots and enables 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, the program can add multiple operators in a current savepoint or create a new one with new features and write it at any desirable location.

The code below shows how a new Savepoint with specific State Backend and operators can be created:

```
Savepoint
    .create(new StateBackend(), maxParallelism)
    .withOperator("uid1", transformation1)
    .withOperator(...)
    .write(savepointPath);
```

The following code shows how one can modify the savepoint, for instance, upon splitting part of a workflow migrating the different part to different computer clusters:

```
existingSavepoint
    .removeOperator("uid1")
    .withOperator("uid1", transformation1)
    .write(savepointPath);
```

## 6.3.2 Migrating State between Flink Clusters

As was the case with Spark, when the Optimizer assigns part of a workflow to a Flink cluster, either in the first time a workflow is submitted or upon a runtime adaptation process, a new Flink Job (Streaming Nest with Flink connection in RapidMiner Studio) must be submitted. In case of a migration process, the first step of its physical execution relies on processing a custom JSON file we define with critical parameters on how the migration process should be performed, but this time modified to match Flink job migration requirements.

The details and structure of the aforementioned JSON file are as follows [Baiko20] (recall that we now have both a checkpoint directory and a savepoint one):

restart

- Data Type: Boolean/String(“true”, “false” values allowed)
- Description: the job restarts after a migration procedure (true) or it is the initial physical plan for that part of the workflow (false)

fromExistingSavepoint

- Data Type: Boolean/String(“true”, “false” values allowed)
- Description: true if we want to restart a job from an existing savepoint.

AllowNonRestoreStates


- Data Type: Boolean/String(“true”, “false” values allowed)
- Description: allows to restore the state if any operator is missed.

cancel job

- Data Type: Boolean/String(“true”, “false” values allowed)
- Description: allows to cancel the running job after the snapshot

checkpoint dir

- Data Type: String
- Description: the new checkpoint directory (it is important in case that the checkpoint directory is not configured in the new job)

 <p>Project supported by the European Commission Contract no. 825070</p>	<p><b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b></p>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public


## jobs

- Data Type: List
- Description: Contains multiple different jobs with different properties as follows:
  - job name
    - Data Type: String
    - Description: The name of the job that the properties below apply
  - job ids
    - Data Type: List <String>
    - Description: the job ids for which we need to get the savepoints. Multiple values in case of merging parts of workflows
  - new savepoint path
    - Data Type: String
    - Description: the new savepoint path (it fails to restart if it exists)
  - existing savepoint paths
    - Data Type: List <String>
    - Description: the existing savepoint paths from which the job is restarted (in case the "fromExistingSavepoint" is true)
  - host
    - Data Type: String
    - Description: the host name of the cluster in which the job ids run
  - port
    - Data Type: String
    - Description: the port of the host cluster in which the job ids run
  - target directory
    - Data Type: String
    - Description: the directory in which the snapshots will be saved (important in case the savepoint directory is not configured)

### JSON property file snippet example [Baiko20]:

```
{
  "restart": "false",
  "fromExistingSavepoint": "false",
  "cancel job": "false",
  "jobs": [
    {
      "job name": "example",
      "job ids": ["ac50c3e9bf9900013b659cdc0735", "fb8179c889ac096816044577f93a"],
      "new savepoint path": "hdfs://45.10.26.123:9000/apps/savepoint/savepoints/example1",
      "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"
}
```

Note that the fact that Flink provides an advanced (the State Processor) API for state handling, and the ability to work with job ids instead of file names, makes the migration processes easier to handle. For instance, in order to split a part of a workflow that is currently running in a single Flink cluster, it suffices to remove the operator states that are not involved in the newly prescribed (two or more) workflow parts that will be transferred to other, remote Flink clusters. Assume we have a single physical workflow WoF1 running in a Flink cluster. In case WoF1 is split into two physical plans WoF2, WoF3 to be executed in two other, remote Flink clusters it suffices to transfer the state of WoF1 to those clusters and remove, from the delivered savepoint, the state of operators (example presented earlier in our discussion) that will not be executed in WoF2 or WoF3 respectively. Moreover, when we want to

 <p>Project supported by the European Commission Contract no. 825070</p>	<p><b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b></p>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public




merge parts of workflows we can do so using their job ids instead of working with file name identifiers as was the case with Spark.

The implementation of the migration algorithm for Flink clusters is based on the State Processor API. 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.

For every job in jobs do:

- Step 1. Hash every operator UID using the murmur3\_128 hash function to check every UID is distinct.
- Step 2. Compare hashing results with the hashed UIDs of the savepoints to find matches.
- Step 3. Copy all the savepoint files into the new savepoint directory, without the metadata file.
- Step 4. Load the savepoints into a Savepoint List using the SavepointLoader class (This gives access to every OperatorState).
- Step 5. Merge all OperatorState into a new OperatorStates List.
- Step 6. Select which OperatorState in the list will exist into the new savepoint (compare the UIDs from the list with the produced UIDs from Step1).
- Step 7. Change the absolute paths of the States that will be included in the new savepoint.
  - Step 7.1. For every OperatorState, change the raw and the managed state absolute paths as follows:
    - Step 7.1.1. Get every state as a List of keyed\_state\_handle.
    - Step 7.1.2. From keyed\_state\_handle List get the delegate\_state\_handle as File\_state\_handle or ByteStream\_state\_handle.
    - Step 7.1.3. Construct a new state\_handle and copy the data of the previous one (to alleviate the migration algorithm from the burden of setting the State Descriptor, the BootstrapTransformation and the Type Info requirements).
    - Step 7.1.4. Change the absolute paths using the new savepoint directory.
    - Step 7.1.5. Construct a new key\_group\_state\_handle with the new absolute paths.
    - Step 7.1.6. Delete the old one.

	Project supported by the European Commission Contract no. 825070	<b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b>	Doc.nr.:	WP5 D5.2
			Rev.:	1.0
			Date:	31/08/2021
			Class.:	Public

Step 7.2. Add the new State into the new OperatorStates List.

- Step 8. Create a new savepoint metadata file using the new OperatorStates List and the new savepoint directory.
- Step 9. Get the constructor of the ExistingSavepoint class and construct the new savepoint using the new metadata file.
- Step 10. Copy back the state files into the old savepoint directories (because they were deleted and another job later on as we progressively read the JSON may require one or more of the involved savepoints).

### Algorithm 6: State Migration Algorithm in Apache Flink [Baiko20]

In case of restarting a workflow, we check if there are existing savepoints. If the savepoints do not already exist, the migration process 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 is 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 we have all the appropriate savepoint paths, the migrate algorithm is called and takes as input arguments (i) the savepoint paths (ii) all the operator UIDs, which exist in RapidMiner's StreamingNest operator's code, and (iii) the new savepoint directory.

Algorithm 6 sums up all steps followed during migrating parts of physical workflows in Flink. Because Flink provides the State Processor API which makes the discussion throughout the current subsection more tied to it, instead of providing pseudocode, we prefer a more explicit description referring to specific variables, functions and methods provided by the involved Flink APIs [Baiko20].

Again, we have no data losses because we do not get access the data. We just copy the corresponding bytes and we change only the absolute paths in order to get access to different savepoint directories. Thus, the change of the absolute paths in a savepoint together with the ability to merge or split operator states from different savepoints is the way to achieve management of multiple savepoints in order to produce a new one with desired characteristics and State [Baiko20].

### 6.3.3 Discussion on Technical Issues upon Migrating Flink State


Compatibility issues exist (not only for Flink, but also for Spark) since the respective APIs including undergo thorough changes and revisions in the latest versions of Flink. Versioning issues are undoubtedly challenging and may cause problems during job migration for computer clusters running different version of Flink. Similarly, only clusters operating under the same Spark versions (or similar versions) can perform job migration because Spark can undergo major updates between updates.

## 6.4 Choosing When to Adapt

Besides the technical details on how to perform runtime adaptation and ensure that the adapted workflow will continue its execution unaffected, preserving its state, there is an important design decision we need to make. This decision involves prescribing when an adaptation event will take place.

Consider that we have a running physical workflow that was initially prescribed by some algorithm of those in Section 4.5 or Section 5. The workflow is deployed, possibly along with a number of other workflows that have been submitted before/after/concurrently with the said one, and we monitor its execution by collecting runtime statistics (Section 4.2). The question is when the runtime adaptation procedures we introduced in Section 6.3, 6.2 will be executed.

We note that this is a design choice that is orthogonal to the optimization algorithms and runtime adaptation procedures we present. It does not affect the execution of the algorithm, the goodness of the prescribed physical

 <p>Project supported by the European Commission Contract no. 825070</p>	<p><b>WP5 T5.1 - T5.3</b> <b>Deliverable D5.2</b></p>	Doc.nr.:	WP5 D5.2
		Rev.:	1.0
		Date:	31/08/2021
		Class.:	Public