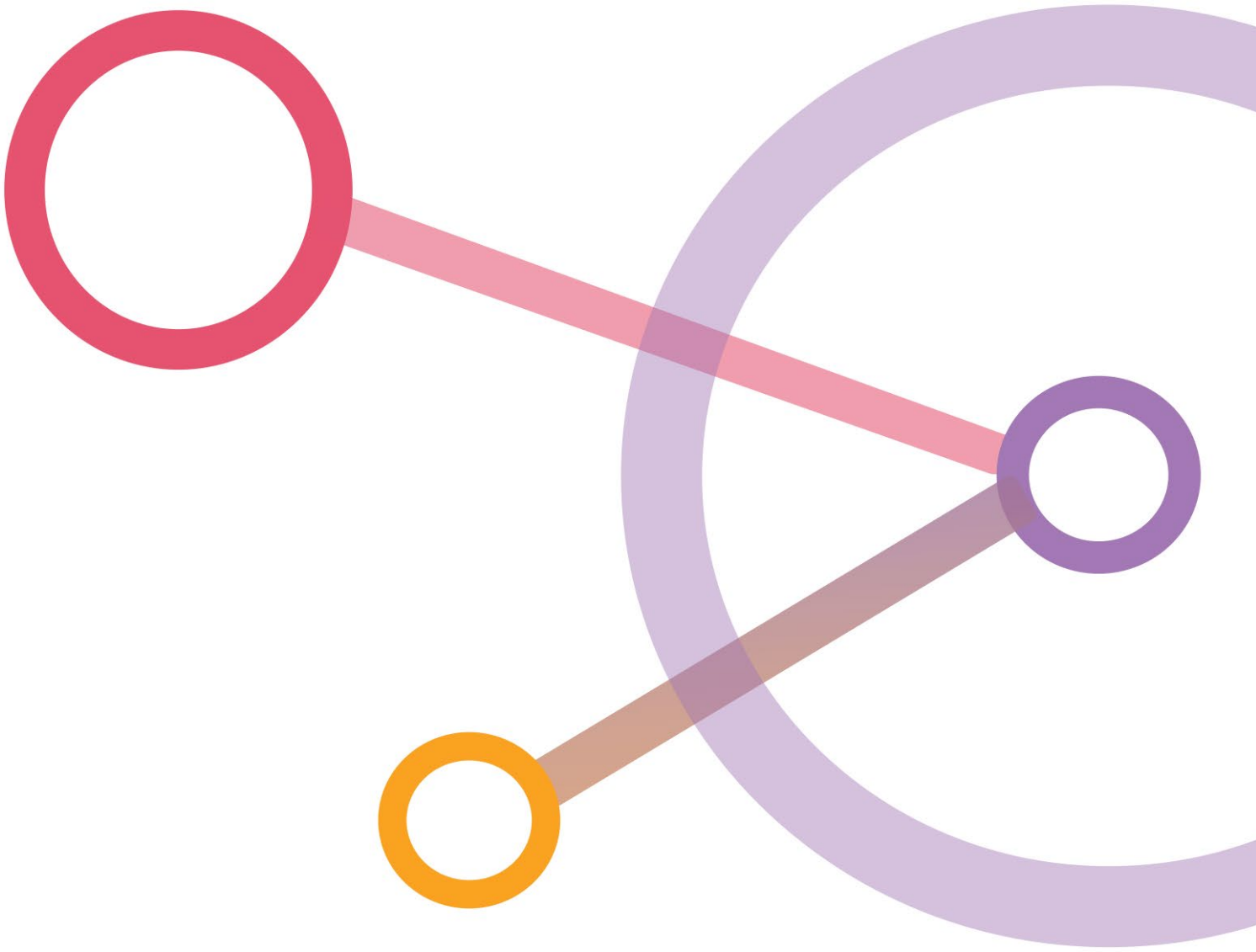

A beginner's Guide to checkpoints in **Apache Flink**

Dawid Wysakowicz, Software Engineer | Ververica

Provided by ververica

Table of Contents

Why are checkpoints necessary in event streaming applications?

Every stream processing application, whether this is a streaming data pipeline, or a streaming SQL application can be stateful, meaning that it involves some sort of state. It's rather straightforward to explain how operations such as event aggregation or the calculation of an average include state in the form of a counter for incoming events or other aggregates.

But even if we take a simpler example, of a Map transformation, changing a single object to a different one, we will still experience that source operators need to store some state to keep track of the offsets that they have consumed so far and determine the position in the input data (for example in a Apache Kafka partition) up until all records have been consumed, and decide what's the next record to be processed.

State is particularly important in the case of failures. Especially for stream processing applications, re-processing events in the case of failure is time consuming (reprocessing and replaying the event sequence could take hours or even days) so rebuilding the state we have accumulated so far might be a hard task to do. To persist state in an easy-to-manage way and recover from a failure, Apache Flink implements a mechanism that allows reprocessing only the events from a specific point in time (previous stored state) instead of replaying the entire history of the application.

There are usually two options for doing that:

- **External State:**

Using an External State would mean that Apache Flink would be utilizing an external persisted system and every access and update would need to be stored to the external persisted storage (i.e., a database or distributed file system)

- **Internal State:**

Using an Internal State mechanism means that Apache Flink persists state locally while only parodically takes snapshots of the accumulat-

-ed state of the application to an external storage or persisted file system to recreate and reproduce the state when necessary.

Each one of the approaches comes with its own advantages and drawbacks. For example, using External State means that scaling/rescaling of the state is independent from the processing logic but at the same time tends to be significantly slower because each access and update to the state needs to go over the network which impacts how quick the response might be.

Additionally, securing exactly-once guarantees with an external storage system tends to be troublesome because the technology would need to coordinate different environments with different processing semantics and guarantees. On the other hand, choosing an Internal State means that every state access or update is significantly faster since it's local and does not have to travel through the network. Additionally, using Internal State allows the creation of a highly consistent, distributed state snapshot of a stream processing application, which is the case for Apache Flink (see below for details).

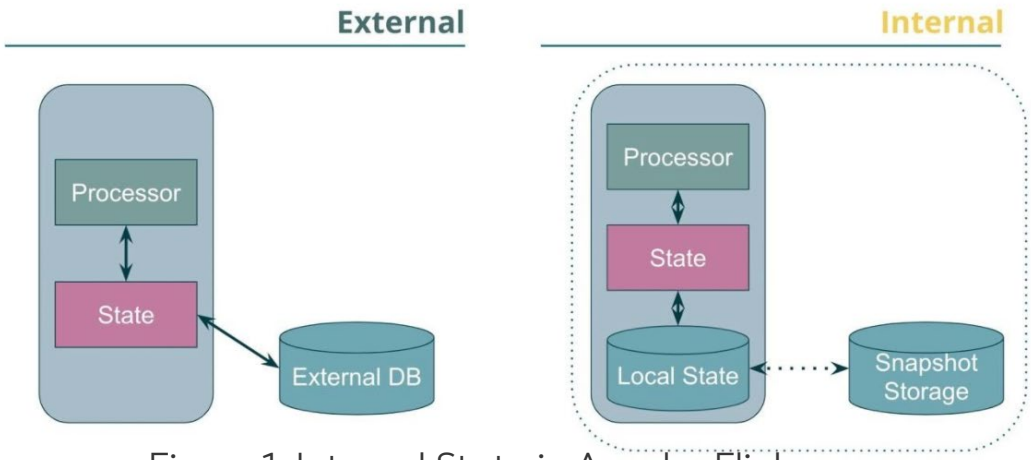

Figure 1: Internal State in Apache Flink

How checkpointing in Apache Flink works - Distributed Snapshots

Apache Flink recovers from failures without the need to reprocess every event from the beginning using a Distributed Snapshots mechanism. Distributed Snapshots in Apache Flink work in a similar fashion to the Chandy–Lamport algorithm.

A Flink application consists of a pipeline of task managers executing the operator's code, and a job manager that acts as a single entity coordinating the checkpointing process among other responsibilities. Job manager comes with a checkpoint coordinator that periodically triggers checkpoints by sending trigger requests to all source tasks in the pipeline. Sources, upon request, take a snapshot copy of their state and store it in a distributed storage or file system before they emit checkpoint barriers downstream.

Step 1: Checkpoint barriers trigger taking a snapshot of the operator state in a distributed storage.

Step 2: Checkpoint barriers continue to move downstream through the operator graph in a Flink application.

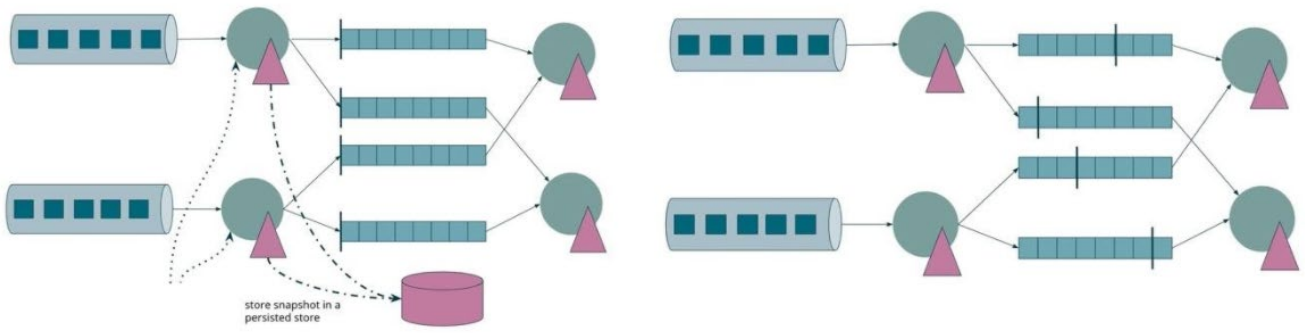

Figure 2: Checkpoint barriers moving through task operators in a Flink application

These barriers are injected into the data stream and flow with the records as part of the event stream. Barriers never (actually they might; see unaligned checkpoints) overtake records, they flow strictly in line. A barrier separates the records in the data stream into the set of records that goes into the current snapshot, and the records that go into the next snapshot.

Taking a closer look into distributed snapshots in a Flink application from the perspective of a single operator, we see that for an operator to create a snapshot of its state, all barriers from all channels should be received so that there is a consistent view from the same point in the application which in turn ensures that all checkpoint barriers across all the channels have arrived in the operator. This process is illustrated in Figure 3 below.

Step 1: Waiting for barriers from all channels to arrive.

Step 2: Snapshotting operators state once we've seen barriers on all channels.

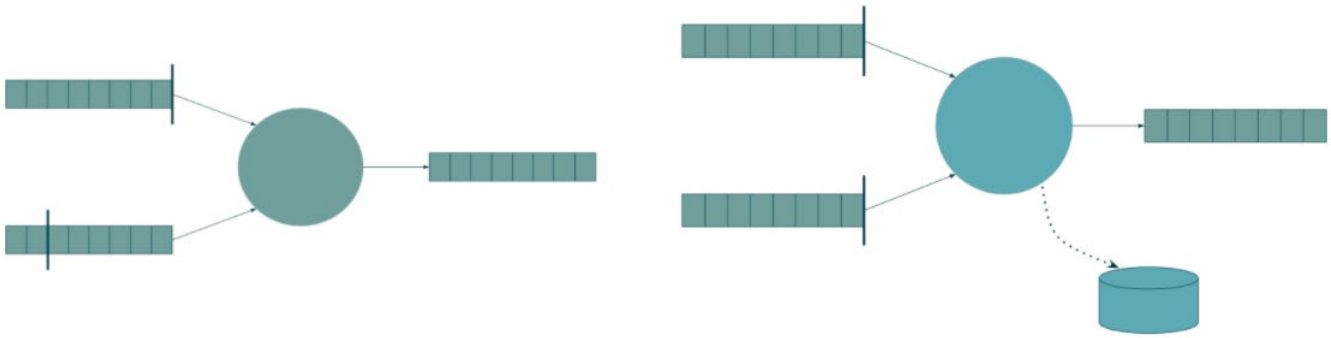

Figure 3: Checkpoint alignment process in a Flink application

Configuring checkpoints in Apache Flink

Now that we have a common understanding of how checkpointing works in Apache Flink, in this section we are discussing some important configuration parameters when setting up checkpointing for your Flink application.

Choosing State Backend

One of the primary considerations when you configure checkpoints in a Flink application is related to the chosen state backend. State backends include local data that can be accessed by each operator in Flink, and they are the ones defining where this 'working' state of the application is kept and how it can be accessed by operators. Apache Flink offers two different state backends, the *`HashMap State Backend`* and an *`EmbeddedRocksDBStateBackend`*, each of them coming with their own advantages and specific characteristics. We take a closer look at both state backends below.

- **HashMapStateBackend**

The *`HashMapStateBackend`* keeps all the 'working' state in memory which naturally results in a faster operation compared to the RocksDB

state backend because each access happens in memory and there is no need for data serialization/deserialization. At the same time, this state backend is limited by the amount of available memory in our application.

- **EmbeddedRocksDBStateBackend**

The `RocksDBStateBackend` uses a co-located key-value store to keep the 'working' state of your Apache Flink application while it spills data into disks. Consequently, RocksDB needs to serialize and deserialize data for your application resulting in a 10X slower operation compared to the HashMap state backend described above. However, RocksDB memory size is practically 'unlimited' since we are no longer restricted by the available memory but only by the disk size which can be updated as needed.

To configure your selected state backend for your Flink application, you can either do this in Flink's configuration parameters like the following:

```
# The backend that will be used to store operator state
checkpoints application
```

Alternatively, you can programmatically configure the state backend on a per job basis/level with the following command:

```
StreamExecutionEnvironment env =
   StreamExecutionEnvironment.getExecutionEnvironment();

env.setStateBackend(new HashMapStateBackend());
// or
env.setStateBackend(new EmbeddedRocksDBStateBackend());
```

Switching State backend over a savepoint (since Flink 1.13)

Starting from Apache Flink 1.13 and later versions, developers and data engineers can switch between the two state backends using a savepoint and can restore their application switching from one state backend to another, a feature that gives greater flexibility to engineering teams.

For developers starting a new Flink application, we would recommend starting with the HashMap State Backend, keeping your state in memory, and only switching to the RocksDB State Backend when you see you are running out of available memory, or you want to move your application to a production environment.

Choosing Checkpoint Storage

A second configuration parameter that data engineers and software developers need to consider when configuring a Flink application is choosing the appropriate checkpoint storage, meaning defining where Apache Flink will store the distributed snapshots. Apache Flink provides two options for storing checkpoints: the JobManagerCheckpointStorage and the FileSystemCheckpointStorage which we describe below.

- **JobManagerCheckpointStorage**

As a default, Apache Flink uses the JobManagerCheckpointStorage which will use the memory of Flink JobManagers to store the checkpoints. Although this ensures that the checkpoints are persisted in the cases of TaskManager failures, when a JobManager fails the stored checkpoints will also disappear.

- **FileSystemCheckpointStorage**

Because of the above, and especially ***for production use cases, it is recommended that data engineers use the FileSystemCheckpointStorage*** *that stores snapshot files in a highly durable, distributed persisted file system that will keep the snapshot data in either a TaskManager failure or a JobManager failure.*

You can configure your desired checkpoint storage option by following the configuration parameters below:

```
flink-conf.yaml

# configures file system checkpoint storage
state.backend.changelog.storage: filesystem

# Directory for storing checkpoints
state.checkpoints.dir: hdfs://namenode:40010/flink/check-
```

Alternatively, you can programmatically configure the checkpoint storage on a per job mode with the following command:

```
StreamExecutionEnvironment env =
   StreamExecutionEnvironment.getExecutionEnvironment();

env.getCheckpointConfig()
   .setCheckpointStorage("hdfs://namenode:40010/flink/check-
```

Setting checkpoint intervals

When configuring your Apache Flink application, you will also need to set up and configure checkpoint intervals. Checkpoint intervals will essentially tell Apache Flink how frequently you would like checkpoints to be triggered in your application. There is no correct answer or value for how often checkpoints should be triggered as this will very much depend on your specific use case. We recommend asking the following questions to internal teams that will guide you towards choosing an appropriate checkpoint interval value for your Flink application:

- **What is the SLA of your application?**

This will determine how long your team will be comfortable waiting for the Flink application to replay and reprocesses data from the last stored checkpoint in the case of a TaskManager failure, for example.

- **How much load can your TaskManagers sustain?**

Even though the checkpointing process happens asynchronously and in the background while your Flink application is regularly processing

any incoming events, there is some additional overhead on your job's TaskManagers both because of I/O and CPU utilization since Flink will need to retrieve a copy of your state from the distributed file system.

- **How often should your Flink application publish results?**

By determining how often your Apache Flink application should be publishing results, you will get a better understanding of your checkpointing requirements. For example, in the case of exactly once processing, transactions can only be committed once the corresponding checkpoint is completed. This is because in the case of failure, reverting to a previous checkpoint will mean that records might be replayed twice and consequently violating exactly once guarantees.

Checkpoint intervals can be configured either in Flink's config.yaml file:

```
flink-conf.yaml

# how often checkpoints are triggered (value greater than 0 enables
# checkpointing)
execution.checkpointing.interval: 1 s

# additional important settings that affect checkpointing frequency

# how many checkpoints can be pending in the entire graph
execution.checkpointing.max-concurrent-checkpoints: 1

# minimal time that has to pass since the last completed checkpoint
# before triggering a new one
execution.checkpointing.min-pause: 700 ms
```

Or alternatively you can programmatically set up your checkpoint interval on a per-job mode by following the below command:

```
final StreamExecutionEnvironment env =
  StreamExecutionEnvironment.getExecutionEnvironment();

// how often checkpoints are triggered (value greater than 0 enables checkpointing)
env.enableCheckpointing(1_000);

// how many checkpoints can be pending in the entire graph
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);

// minimal time that has to pass since the last completed checkpoint before triggering a new one
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(700);
```

Conclusion

Apache Flink's checkpointing mechanism is one of the key features of Flink as a stream processor, ensuring timely and easy recovery from failures while at the same time enabling Flink to process events with exactly-once guarantees. In the previous sections, we gave an introduction into what checkpoints are in Apache Flink, how they work and how they can be configured for different scenarios (for example, depending on whether your application is in the POC phase or whether it's running in a production environment. For a more thorough presentation and explanation of Flink's checkpointing mechanism as well as some recent (more advanced) additions and changes to checkpointing in Apache Flink, we suggest going over the following resources.

Additional Resources

- **Apache Flink Documentation:**
https://nightlies.apache.org/flink/flink-docs-master/docs/ops/state/checkpoints/
https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/concepts/stateful-stream-processing/

About Ververica

Ververica's mission is to power the core business of every company with cutting-edge real-time stream processing technology. In order to do that, the team at Ververica focuses on building the best technology available for stream processing, while at the same time creating a global and open community around this technology.

We build and develop Ververica Platform, a stream processing platform that enables every enterprise to power their real-time business and use a production-grade streaming infrastructure while at the same time we actively contribute and participate in the open source Apache Flink® community, the underlying technology framework of Ververica Platform itself.

For more information:

@ sales@ververica.com

@VervericaData

www.ververica.com