


FLIP-34: Terminate/Suspend Job with Savepoint

| | |
|-------------------|--|
| Discussion thread | https://lists.apache.org/thread/7d8xx1d2qqxp41pwgmw8kdgyk2c8gcv7 |
| Vote thread | |
| JIRA |  FLINK-11458 - Add TERMINATE/SUSPEND Job with Savepoint (FLIP-34) CLOSED |
| Release | 1.9 |

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

In many scenarios, we want to finish a job with a savepoint. This can be, for example, because we want to upgrade our cluster, or because we finished processing the important part of our data and we want to simply terminate the job, but make sure that our output is safely stored to our sink storage system, e.g. using the `StreamingFileSink`.

Currently, Flink offers the functionality of cancelling a job with a savepoint. Under the hood, this entails two tasks. These are:

1. take a savepoint, and when the state of the checkpoint is safely stored,
2. cancel the job.

When applied to the current exactly-once sinks, this approach is problematic, as it does not guarantee that side-effects will be committed by Flink to the 3rd party storage system. To understand the problem in more detail, see Appendix A below.

This document targets fixing the end-to-end semantics when stopping jobs and proposes the addition of two shutting down modes which ensure that no events come after the checkpoint. In the remainder of the document we clarify the semantics of the two shutdown methods and discuss the implementation of the feature.

Goals

We propose the addition of the following two shutdown modes, namely:

- SUSPEND for temporarily stopping (like upgrades), and
- TERMINATE for terminal shut down

The remainder of the paragraph presents the similarities and differences of the two modes.

SIMILARITIES

Both modes shut down with a checkpoint/savepoint and ensure all side effects go through. This will guarantee exactly-once semantics end-to-end. In addition, both modes ensure that no events come after the checkpoint. This will allow for better guarantees of at-least once sinks. Finally, in both cases, the JOB STATE should be FINISHED to indicate that the operation that the user requested completed successfully. This differs from how the current “cancel with savepoint” is implemented, which ends the Job in state CANCELED. Having the job state being CANCELED would make it impossible for the user to differentiate between success and failure of the requested operation, i.e. it can happen that someone cancels the job while a TERMINATE or SUSPEND is happening.

DIFFERENCES

Event Time Semantics

In SUSPEND, the user just wants to “pause” the job rather than stop it. For a job with event-time timers, this would imply that we do not want the registered (event-time) timers to fire, as the job will resume in the future. In this case, we should not send MAX_WATERMARK (for event-time).

In TERMINATE, we want the timers to fire, as this will flush out any buffered state associated with timers, e.g. non-fired windows. In this case, we should emit MAX_WATERMARK for event-time timers. For processing time timers, there is not much we can do for now.

System Perspective

The following table illustrates the differences between the two new operations from an operator/task perspective.

| | Source OPS | Task Status | Job Status |
|-----------|---|-------------|------------|
| SUSPEND | Checkpoint Barrier, End Of Stream | Finished | Finished |
| TERMINATE | MAX_WATERMARK, Checkpoint Barrier, End Of Stream | Finished | Finished |

User-Facing Changes

- Expose the new SUSPEND and TERMINATE operations through REST to the user
- Change event time semantics in accordance with the above table

Open questions

1. Any better suggestions about the names of TERMINATE and SUSPEND
2. Failure in the TERMINATE case:
 - a. We launch a TERMINATE, so MAX_WATERMARK is emitted
 - b. Event-time Timers fire
 - c. Savepoint (state persistence phase) succeeds
 - d. An exception is thrown during the notifyOnCheckpointComplete()

Appendix A: Cancel with Savepoint shortcomings

To understand the problem, first we will explain how an “exactly-once” sink is implemented in Flink in the general case. This holds true for every sink that implements a flavor of a “two-phase commit” protocol. Such a sink follows the pattern:

1. on the onElement(), the sink normally keeps data in a temporary buffer (not necessarily in Flink’s state) and updates the necessary metadata. This metadata is stored in Flink and will be included in the next checkpoint. In case of a failure, the sink will need it to recover without breaking the exactly-once guarantees.
2. onCheckpoint() the sink makes sure that the metadata is successfully persisted
3. onCheckpointComplete(), i.e. when all parallel tasks have finished storing their state, the sink will try to commit/publish the data that were buffered up to the successfully committed checkpoint. It is safe to do it now, as even in the case of failure, Flink guarantees that the job will re-start from its “last successful checkpoint”. In addition, the sink will clear the state associated with that checkpoint, as it is no longer needed.

In the above description, the actual commit phase of the “two-phase commit” takes place in the onCheckpointComplete(), i.e. after the checkpoint is successfully acknowledged to the JobManager.

In its current implementation, the `onCheckpointComplete()` notification is implemented as a “best-effort” mechanism. This is due to the fact that the completion of the callback is not acknowledged to the JobMaster.

In addition, when performing a cancel-with-savepoint, the cancel command is issued right after the acknowledgement of the successful persistence of the checkpointed state. This means that with high probability, the “commit” phase will not have been executed by all tasks by the time the job is cancelled, thus leaving the outside view of the system in an inconsistent state.

Public Interfaces

There is no expected change in public interfaces

Initial Implementation Plan

An initial sketch (by no means final) of the implementation can look like this:

1. the Job Manager triggers a synchronous savepoint at the sources, that also indicates one of TERMINATE or SUSPEND
2. sources send a MAX_WATERMARK in case of TERMINATE, nothing is done in case of SUSPEND
3. the Task Manager executes the checkpoint in a SYNCHRONOUS way, i.e. it blocks until the state is persisted successfully and the `notifyCheckpointComplete()` is executed.
4. the Task Manager acknowledges the successful persistence of the state for the savepoint
5. the Job Manager sends the notification that the checkpoint is completed
6. The Task Manager unblock the synchronous checkpoint execution.
7. Finishing the job propagates from the sources, i.e. they shut down and EOS messages propagate through the job.
8. The Job Manager waits until the job state goes to FINISHED before declaring the operation successful.

Proposed Changes

The implementation plan (which will be further updated) roughly describes the components to be touched.