FLIP-76: Unaligned Checkpoints

Status

Current state: *Implemented*

Discussion thread: http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/DISCUSS-FLIP-76-Unaligned-checkpoints-td33651.html

JIRA: ~~FLINK-14551~~ - Unaligned checkpoints CLOSED

Released: MVP in Flink 1.11, fully implemented in Flink 1.13

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

Motivation

Currently, triggering a checkpoint will cause a checkpoint barrier to flow from the sources through the DAG towards the sinks. This checkpoint barrier guarantees a consistent snapshot of the DAG at a given point in time.

For each operator with multiple input channels, an alignment phase is started whenever the checkpoint barrier is received on one channel, which causes further inputs of that channel to be blocked until the checkpoint barrier from the others channels are received.

This approach works fairly well for modest utilization but exhibits two main issues on a back-pressured pipeline:

- Since the checkpoint barrier flows much slower through the back-pressured channels, the other channels and their upstream operators are effectively blocked during checkpointing.
- The checkpoint barrier takes a long time to reach the sinks causing long checkpointing times. A longer checkpointing time in turn means that the checkpoint will be fairly outdated once done. Since a heavily utilized pipeline is inherently more fragile, we may run into a vicious cycle of late checkpoints, crash, recovery to a rather outdated checkpoint, more back pressure, and even later checkpoints, which would result in little to no progress in the application.

This FLIP proposes a way to perform checkpoints with a non-blocking alignment of checkpoint barriers. It provides the following benefits.

- Upstream processes can continue to produce data, even if some operator still waits on a checkpoint barrier on a specific input channel.
- Checkpointing times are heavily reduced across the execution graph, even for operators with a single input channel.
- End-users will see more progress even in unstable environments as more up-to-date checkpoints will avoid too many recomputations.
- Facilitate faster rescaling.

The key idea is to allow checkpoint barriers to be forwarded to downstream tasks before the synchronous part of the checkpointing has been conducted (see Fig. 1). To that end, we need to store in-flight data as part of the checkpoint as described in greater details in this FLIP.

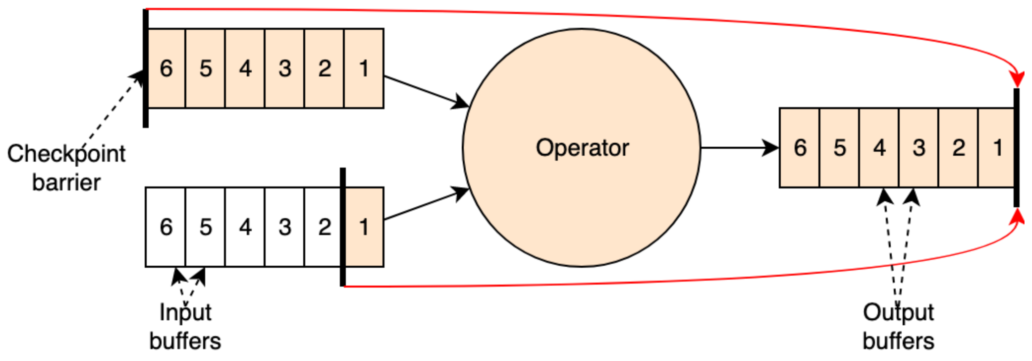


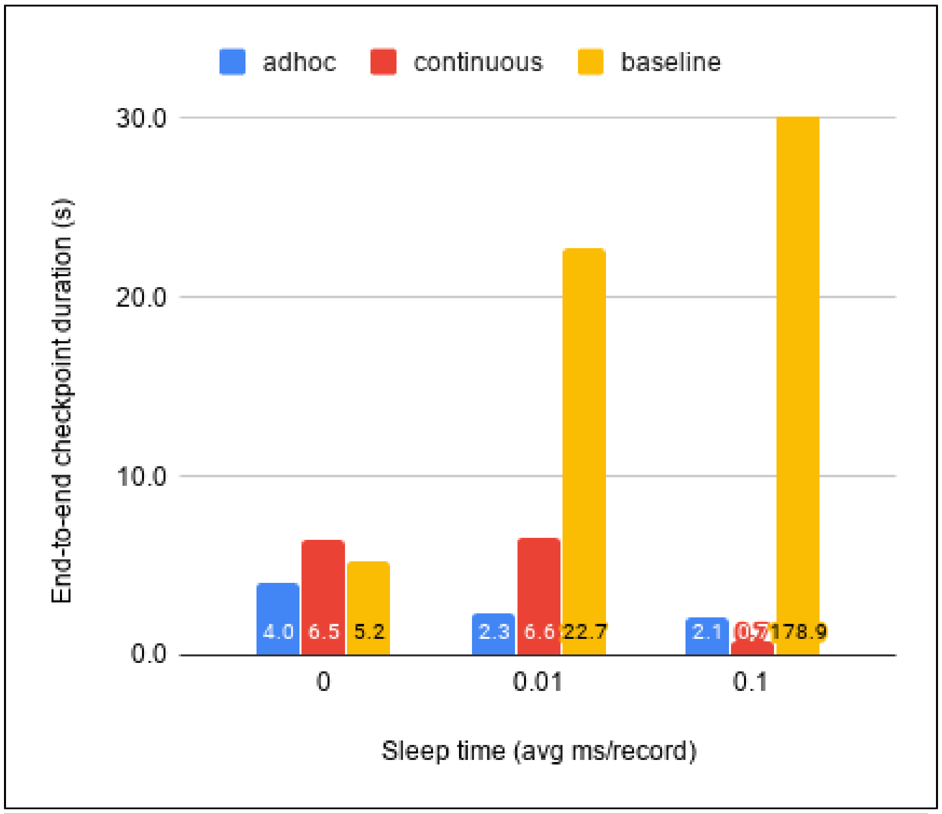
Figure 1: Checkpoint barriers overtake in-flight records. Colored buffers need to be persisted together with the operator state.

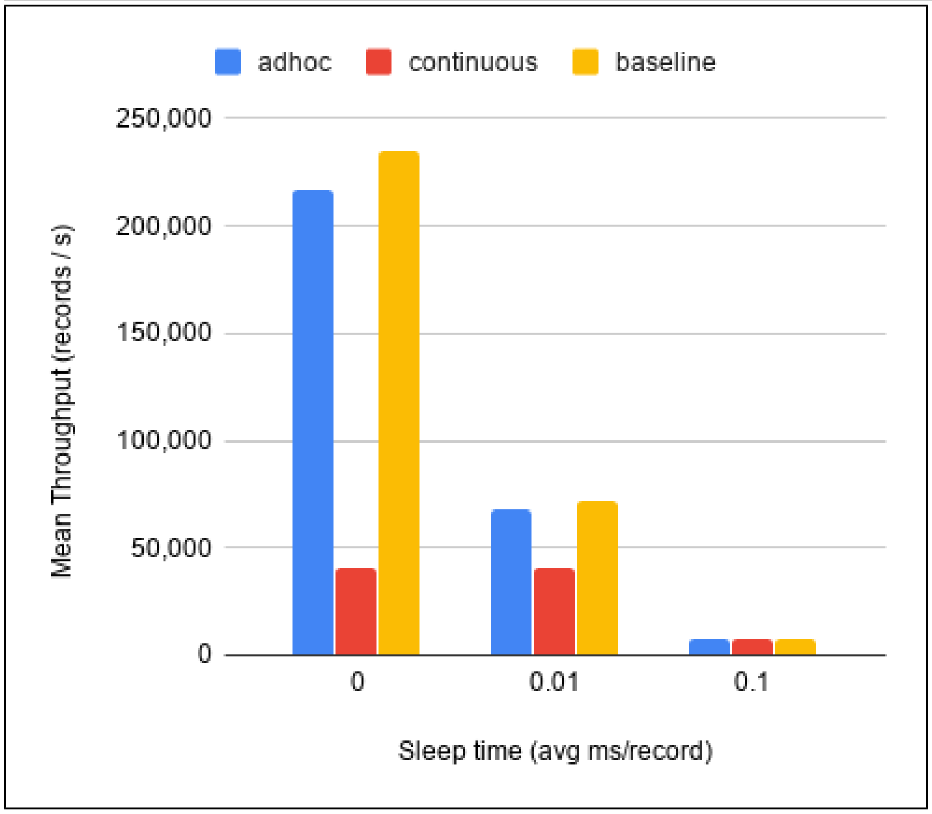
Preliminary work

We first evaluated with POCs two options when to persist:

- Persist in an **adhoc** fashion on checkpoint. While keeping the data volume to a minimum this approach could potentially overload external systems during checkpointing.
- Persist **continuously** to avoid such overload on the cost of storing much more data and potentially decreasing throughput.

In the following experiments, we had a simple topology (source map map sleepy map measure map), where each channel was a random shuffle. The sleepy map slept on average 0, 0.01, and 0.1 ms per record to induce backpressure. We compared a POC for adhoc spilling, a POC for continuous spilling, and the base commit in the master (1.11-SNAPSHOT). For each approach and sleep time, we measured 3 times and took the median while persisting on HDFS in an EMR cluster with 1 master and 5 slaves, each having 32 cores and 256 GB RAM, with a total parallelism of 160.





We can see that for the baseline, the checkpoint duration strongly increases with sleep time and thus backpressure. For both POCs, the checkpoint duration, however, remained stable and even decreases for higher backpressure because of lower overall data volume. Surprisingly, the checkpointing times for continuous spilling are higher than adhoc for lower sleep times. We suspect that we have additional backpressure for continuously write large amounts of data. With increasing backpressure and thus decreasing data volume, continuous spilling reaches sub-seconds checkpointing times.

Nevertheless, continuous spilling seems to have rather big impact of overall throughput. While adhoc POC showed 10% performance decrease, continuous POC is clearly bottlenecked for higher volume. Primarily for that reason, **we decided to go with the adhoc approach in this FLIP**. While sub-seconds checkpointing times of continuous spilling surely would be a nice asset, the primary goal of decoupling checkpointing times from backpressure is also reached with adhoc spilling.

Proposed Changes

In this FLIP, we

- Change the handling of checkpoint barriers, to allow them to overtake other records,
- Add persist the inflight data inside of checkpoints,
- Recover from the new checkpoint while also allowing new checkpoints to be taken during recovering to guarantee progress, and
- Propose a solution of how the operators can be rescaled in the future

Handling of checkpoint barriers

Whenever we receive checkpoint barriers, we forward them immediately to the output. To achieve a consistent checkpoint, we need to persist all data in the input and output buffers between the received and forwarded checkpoint barrier. In contrast to the current checkpoints, we not only “freeze” the current operator state during the synchronous checkpointing phase, but also “freeze” the channels (see Fig. 1).

For operators with one input channel, the checkpoint barrier directly overtakes all input data, triggers the synchronous parts of the checkpointing algorithm in the operator and is forwarded to the end of the output buffer.

If an operator has multiple input channels, there are two options:

- We start checkpointing directly when we receive the first checkpoint barrier. At that time, we may receive an arbitrarily large number of records on the other channels before seeing the respective checkpoint barriers. We can receive arbitrarily large number of records only if an upstream operator multiplies the records number (like flatMap or join). In other cases the number of records is limited by the size of Flink’s network buffers.
- We wait until we see the last checkpoint barrier and block the other input channels. In comparison to aligned checkpoints, we will block data flow for a shorter amount of time.

Here, the tradeoff is between storage size for in-flight data and checkpoint latency and time where an input channel is blocked. The current state can even be incorporated by waiting for checkpoint barrier to arrive at the operator and inserts the checkpoint barrier before the output.

We want to implement all 3 versions to allow a fine-grain evaluation. Depending on the outcome, we will choose a specific default option and exhibit a (hidden) configuration “checkpointing policy”:

- ALIGNED (current behavior),
- UNALIGNED_WITH_MAX_INFLIGHT_DATA (and secondary integer option “checkpointing max inflight data”),
- UNALIGNED_WITH_UNLIMITED_INFLIGHT_DATA.

Note, that during downscaling the size limit of “max inflight data” might be temporarily exceeded during recovery (see recovery section).

Persistence

For persistence, existing state management primitives will be reused to colocate operator state and inflight data, which offers the following advantages.

- Don’t duplicate code and avoid inconsistencies between channel and operator state distribution (especially keyed)
- **Simplicity on recovery and rescaling**: having state not split into channel and operator parts
- Avoid inconsistencies between operator and channel state (e.g. different retention times)
- Being able to use existing snapshotting mechanism with the possibility to reuse incremental checkpoints

However, some drawbacks may require to use alternatives ways in the future.

- Less flexibility
- Risk of break snapshotting
- Increased checkpoints size

The checkpoint format is only implicitly extended by adding more (keyed) state with conventional naming.

Components

In general, inflight data is stored in state handles per operator sub task that are ultimately managed by CheckpointCoordinator. We need to add or modify the following components.

1. **Checkpoint Metadata**
    a. **Channel StateHandle** extends StreamStateHandle, contains subtask index and information to place it to the correct subpartition /inputchannel
    b. **TaskStateSnapshot**, **PendingCheckpoint**, and **CompletedCheckpoint** - contains a collection of ChannelStateHandle
2. **Writer**. Writes network buffers using a provided CheckpointStreamFactory and returns state handles
3. **Reader**. Reads data from state handle and returns buffers (loads and deserializes)
4. **Buffer (de)serializer**
5. **StateAssignmentOperation** (existing class) - reads snapshot metadata and distributes state handles across subtasks; need to add channel states distribution

TaskStateManagerImpl (existing class) - holds state assigned to task

Recovery

From a high level perspective, inflight data is restored where it was previously spilled:

- Outgoing buffers are restored on upstream side into output.
- Incoming buffers are restored on downstream side into the input.

This approach facilitates an easy integration into existing checkpointing mechanism. Restored buffers take precedence over all newly produced/received buffers. The following steps explain the assignment of state and mapping it to in-memory data structures (without rescaling) in more detail:

- Channel state consists of upstream and downstream parts; they are recovered by upstream/downstream subtasks separately
- Each part (InputChannel/SubPartition) is represented as StateHandle and included in the snapshot
- Each part in metadata has subtask id and target or source subtask id
- On recovery, checkpoint coordinator distributes state according to subtask index
    1. Without rescaling, we don’t need to do anything to colocate operator and channel state (state placement isn't changed)
    2. Raw state is distributed in the same manner
- Subtasks place state into proper InputChannel/SubPartitions according to target/source id

Rescaling

Since we hook into existing checkpointing mechanism, most of the rescaling challenges are solved in the same way.

Non-keyed

To place a state into a task CheckpointCoordinator uses modulo operation. For example, when scaling out:

1. Upstream: new_par_level % old_src_task_index
2. Downstream: new_par_level % old_dst_task_index

Subtask uses the same logic to determine the right InputChannel/SubPartition.

To match multi-buffer records, both upstream and downstream must sort records in the same order

- i.e., if we have same some channel state on upstream, and channel state on downstream corresponds to it (as ensured by step 4); then it’s enough to sort them by channel id ([src_id : dst_id]); this will ensure the order of records
- there is a (technical?) problem with the proposed solution to match multi-buffer records: we need to alternate load channel and process network operations which can be tricky
- another solution would be to have temporary "virtual" channels to process data from "imported" channels

Keyed

For each key group, we need to place channel and operator state on the same subtask. For **output buffers,** we know (at least conceptually) the keys, so we can store state as KeyedStateHandle and then reuse operator state redistribution code.For **input buffers,** we only know a set of groups - the groups assigned to this task.

But, given that placement of key groups among the tasks is deterministic, we can: 1) compute old and new placements and 2) give each new task input buffers that **could** contain it’s key groups. The task then must filter out irrelevant records by keys (after deserialization). For regular (single-buffer) records it’s trivial.

For multi-buffer record, if it doesn’t belong to this subtask, it will never receive remaining buffers and therefore can’t deserialize and filter it out. Possible solutions:

1. Send restored records from upstream to **all** downstreams that could need it (not according to key); we can use the same logic as in distribution for that
2. Use sequence numbers: if we expect next part of a multi-buffer record on this channel, but receive a buffer with a different SN; then discard this stored buffer
3. Write key in the beginning of each buffer (need to ensure the key itself isn’t splitted)

Guaranteed progress

To guarantee progress even during recovery, we need to implement three features:

- Checkpoint barriers need to be processed even if the unspilling is not completed and all input buffers are taken.

- Ideally, data that has not been completely unspilled will not be written twice. Instead, only the offsets of the checkpointed operators are updated.
- Data unspilling does not require the full data to be copied to the task; that is, persistent data is consumed/unspilled lazily as a stream.

Thus, in an environment with frequent crashes (under backpressure), progress can be made as soon as the operators themselves are recovered fast enough.

Compatibility, Deprecation, and Migration Plan

Unaligned checkpoints will initially be an optional feature. After collecting experience and implementing all necessary extensions, unaligned checkpoint will probably be enabled by default for exactly once.

For compatibility, in documentation, clearly state that users of the checkpoint API can only use unaligned checkpoints if they do not require a consistent state across all operators.

Known Limitations

- State size increase
    - Up to a couple of GB per task (especially painful on IO bound clusters)
    - Depends on the actual policy used (probably UNALIGNED_WITH_MAX_INFLIGHT_DATA is the more plausible default)
- Longer and heavier recovery depending on the increased state size
    - Can potentially trigger death spiral after a first failure
    - More up-to-date checkpoints will most likely still be an improvement about the current checkpoint behavior during backpressure

Test Plan

- Correctness tests with induced failures
- Compare checkpoints times under backpressure with current state
- Compare throughput under backpressure with current state
- Compare progress under backpressure with frequently induced failures
- Compare throughput with no checkpointing

Roadmap

There are two primary goals. First, we want to provide a minimal viable product (MVP) that breaks up the vicious cycle of overload and instabilities of a cluster with slow checkpoints and more accumulated load resulting from recoveries from outdated checkpoints. Second, we aim for the full FLIP implementation that will among other things allow rescaling from unaligned checkpoints to directly counter overload.

In particular, the following improvements will be achieved through full release over MVP:

- Re-scaling on unaligned checkpoints (need to use savepoint in MVP)
- Incremental checkpointing to not write the same buffer multiple times (when the job is backpressured)
- Advanced triggers for unaligned checkpoints such as timeouts on alignment or meeting the maximum threshold of checkpoint sizes
- Support for concurrent checkpoints
- Incremental loading and processing of state
- No additional memory to load channel state: ideally, existing network buffers should be reused
- Reduced number of files: single file could be reused for several checkpoints

We aim to make unaligned checkpoints the default behavior after the full implementation.