

FLIP-XXX: Store large operator coordinator state outside the checkpoint metadata

Discussion thread	https://lists.apache.org/thread/c9yb5hdy3rpng04ntywo8x46r2mm3xzl
Vote thread	(to be added)
JIRA	(to be created)
Release	(to be decided)
Author	Spoorthi Basu
Prototype	https://github.com/spoorthibasus/flink/tree/flip-coordinator-state-prototype

Motivation

Operator coordinators run on the JobManager and checkpoint their own state. Every FLIP-27 source has one, which runs its split enumerator. State written by tasks follows the rule described for `execution.checkpointing.data-inline-threshold`: "All state chunks smaller than that are stored inline in the root checkpoint metadata file." Larger state goes to its own file and the metadata only references it.

Coordinator state does not follow this rule. It is stored inline in `_metadata` whatever its size. `OperatorCoordinatorCheckpoints` wraps it in a `ByteStreamStateHandle`, `OperatorState` holds it as that type, and the metadata serializer writes its bytes directly into the metadata file. State from checkpoint master hooks (`MasterTriggerRestoreHook`) is also always inline; this FLIP does not change it.

The reason was threading rather than size. FLINK-17674 typed coordinator state as `ByteStreamStateHandle` because restoring state to coordinators "happens in the JobManager's main thread and must consequently not do any potentially blocking I/O operations". It also noted: "If state restoring becomes an asynchronous operation we can relax this restriction." For sources, restoring has been partly asynchronous since Flink 1.12. Every FLIP-27 source coordinator is wrapped in `RecreateOnResetOperatorCoordinator` (FLINK-18039, released in 1.11; the reset became asynchronous in 1.12). When it is reset to a checkpoint, it closes the old coordinator in a separate thread, creates and resets a new one once the close completes, and defers all calls to the coordinator in the meantime. The restriction on coordinator state stayed in place. This FLIP makes reading the state part of that asynchronous reset.

Enumerator state grows with the number of splits, and several sources keep a lot of it:

- Flink CDC keeps metadata for every finished snapshot split, which caused a JobManager OOM. Flink CDC 3.7 adds an option, off by default, that releases it once it is no longer needed (FLINK-39775, FLINK-40697, FLINK-40732).
- FLINK-39754 reports an `IcebergEnumeratorState` with about 1.87 million splits serialized past 1 GB in production. `apache/iceberg#14464` reports serialization failing at the 2 GB

array limit.

- In continuous mode, the FileSource enumerator keeps every path it has processed (FLINK-25672). One report describes that history pushing JobManagers past the 16 GB limit of the pool they run in.

Keeping that state inline has three costs.

First, the JobManager keeps extra copies of it. Every retained checkpoint keeps its coordinator state as a byte array on the JobManager heap, next to the live enumerator.

Second, with JobManager high availability, each completed checkpoint is also Java-serialized into the HA storage directory by FileSystemStateStorageHelper, which both the ZooKeeper and the Kubernetes HA services use. Coordinator state is part of that object, so it is written a second time on every checkpoint, and a recovering JobManager reads it back once per retained checkpoint.

Third, every tool that reads only the metadata reads the coordinator state too: disposing of a savepoint, inspecting one with the State Processor API or the `savepoint_metadata` table function. FLINK-36679 added a metric for `_metadata` size after the file bloated "up to 100MB~3GB causing job slow and/or fail to start", in cases involving task state rather than coordinator state.

Measured on a prototype against the master branch of 25 September 2026, with a source whose enumerator holds a payload of 600-byte entries. The job is stopped with a savepoint, the savepoint directory is moved elsewhere, and the job is restored from the new location:

enumerator state	_metadata today	_metadata with this FLIP (option enabled)
30 MB	30,201,424 bytes	1,416 bytes
120 MB	120,801,424 bytes	1,416 bytes

All state came back in both cases. In a single-JVM MiniCluster running a job with 30 MB of enumerator state, used heap after a full GC is 116 MiB today with two retained checkpoints and 145 to 146 MiB with three, since each retained checkpoint holds a copy of about 29 MiB; with the change enabled it stays at 57 to 58 MiB. For the HA copy, 64 KiB of coordinator state makes the serialized completed checkpoint about 68 KB with the change disabled and about 3 KB with it.

This FLIP does not change how much memory the enumerator itself uses, or the 2 GB limit on a single coordinator's state, which comes from coordinator state being one byte array:

`OperatorCoordinator#checkpointCoordinator` produces one, and so does the enumerator's `SimpleVersionedSerializer`. Lifting that limit needs coordinator state to be written and read as a stream. Storing the state in its own file makes that possible, but it is left to a separate proposal.

Public Interfaces

A new configuration option, `execution.checkpointing.coordinator-state-in-file.enabled` (default false), controls whether coordinator state may be stored in its own file. While it is disabled, every checkpoint and savepoint the job takes stays in the format Flink writes without this FLIP.

A second new option, `execution.checkpointing.coordinator-state-inline-threshold`, sets the size below which coordinator state stays inline when the first option is enabled. It has no default: if it is not set, `execution.checkpointing.data-inline-threshold` applies, as its description already suggests. If it is set, it replaces `data-inline-threshold` for coordinator state only, so users can balance JobManager memory against the number of files without changing how task state is

stored. Unlike `data-inline-threshold`, which cannot be set above 1 MB, it has no maximum, since a larger value only keeps more coordinator state inline, as Flink does today. Coordinator state below the threshold that applies, and the state of coordinators that do not restore asynchronously, is stored as it is today.

Checkpoint and savepoint metadata gets a version 7. It is written only when at least one coordinator state was stored in its own file; otherwise the metadata stays at version 6 and is identical to what is written today. Version numbers here are those of the master branch of 25 September 2026; see the note on FLIP-600 under compatibility.

The following changes are internal (the classes are `@Internal` or carry no annotation):

- `OperatorCoordinator` gets two default methods, following `supportsBatchSnapshot()`: `supportsCoordinatorStateInFile()`, which returns false, and `resetToCheckpointFromHandle(long, StreamStateHandle)`, which a coordinator that returns true implements without blocking the caller. `RecreateOnResetOperatorCoordinator` implements both.
- `OperatorState#getCoordinatorState()` returns `StreamStateHandle` instead of `ByteStreamStateHandle`, and `setCoordinatorState` takes one. Code compiled against the old signatures will not link against the new ones.

The `lastCheckpointMetadataSize` metric no longer includes coordinator state that is stored in its own file. Checkpoint statistics only add up the state that subtasks report, so among checkpoint metrics it was the only one that reflected coordinator state. To keep it visible, the size of coordinator state stored in a file is added to the checkpoint's state size and checkpointed size (`lastCheckpointFullSize` and `lastCheckpointSize`). It belongs to no vertex, so the per-vertex figures no longer add up to these totals. Coordinator state kept inline stays part of the metadata size, as today.

Proposed Changes

Which coordinators

A coordinator's state may be stored in its own file only if the coordinator reports, through `supportsCoordinatorStateInFile()`, that it can read the state back without blocking the caller, and `execution.checkpointing.coordinator-state-in-file.enabled` is true.

`RecreateOnResetOperatorCoordinator` does, which covers every FLIP-27 source and other coordinators built on it, such as the sink statistics coordinators of Iceberg, Paimon and Fluss. It reads the file itself and passes the same bytes as today to the coordinator it wraps, so those coordinators need no changes. The state of all other coordinators stays inline, and restoring them works as it does today, unless they opt in.

Writing coordinator state

When a checkpoint completes, `PendingCheckpoint#finalizeCheckpoint` writes the state of those coordinators, except state that stays inline under `execution.checkpointing.coordinator-state-inline-threshold`, through the checkpoint location's `CheckpointStateOutputStream` in exclusive scope, just before it writes `_metadata` and on the same thread, which is never the `JobManager` main thread. If `execution.checkpointing.coordinator-state-inline-threshold` is not set, the stream applies the storage's inline threshold, as for task state. Below it, the stream hands back an in-memory

handle and the state stays inline as today. Above it, the stream writes a file into the checkpoint directory and returns a handle to that file, and the completed checkpoint keeps only that handle. If the new threshold is set, state below it stays inline without being written to a stream, and other state is written through a stream whose inline threshold is zero, so it goes to a file unless it is empty. Streams that apply the storage's threshold get a buffer at least that large; the new threshold sets no buffer size, so a large value needs no large buffer.

Savepoint locations return relative handles, so savepoints stay relocatable, with the same exception as for task state: file systems that inject entropy into paths get absolute paths.

This applies to file system checkpoint storage only. Memory-backed storage caps every piece of state at its maximum state size, 5 MB by default, and fails the checkpoint above it. Coordinator state has never been subject to that cap, so on memory-backed storage, and on checkpoint storage implementations that are not file based, it stays inline.

Metadata

A new `MetadataV7Serializer` is identical to version 6, except that coordinator state may be any `StreamStateHandle`. The serializers for earlier versions reject such state, both when writing and when reading. Earlier version bumps made the new version the default for every checkpoint; here `Checkpoints#storeCheckpointMetadata` chooses version 7 per checkpoint, only when some coordinator state is in a file.

Restoring

`CheckpointCoordinator#restoreStateToCoordinators` passes each coordinator its state handle instead of its bytes. State held in memory is passed on as bytes, as today. State in a file goes to `resetToCheckpointFromHandle`. `RecreateOnResetOperatorCoordinator` reads it in a separate thread, in parallel with closing the old coordinator. Once both are done it creates the new coordinator and resets it with the bytes. Calls to the coordinator are deferred until then, as for any reset today. If a later reset replaces this one, or the coordinator is closed, while the file is still being read, the read is stopped by closing its stream from a separate thread, since closing a stream can block. A stream still being opened is closed as soon as it is open. The result is discarded, and the job does not fail. If the file cannot be read, the job fails and restarts according to its restart strategy, as when a coordinator fails to reset.

No file is read on the `JobManager` main thread, and nothing reads coordinator state files when checkpoints are loaded: restoring from a savepoint or retained checkpoint, recovering the checkpoint store after a `JobManager` failover, and cleaning up after a finished job read the metadata and the HA entries as they do today. Only the coordinator being restored reads its own file.

If the read fails before the scheduler has initialized the coordinator, which is the case for the initial restore and for every restart of the adaptive scheduler, the failure is held and fails the job as soon as the coordinator is initialized.

When the state is in a file, the `JobManager` no longer keeps a copy, so every global failover, and every restart or rescale of the adaptive scheduler, reads the state from its file. The time this adds grows with the state size and the throughput of the checkpoint storage, as it does for task state, which is read the same way unless local recovery is enabled. Keeping the latest copy in memory instead would, with the default of one retained checkpoint, keep the whole heap cost. The read has no timeout of its own and relies on the file system client's timeouts, like reads of

task state.

State Processor API

SavepointWriter copies the files of the operators it carries over from an existing savepoint into the new one. StatePathExtractor, which lists those files, now includes the coordinator state file. Without this, a rewritten savepoint would reference a file it does not contain and would fail to restore.

Documentation

The description of `execution.checkpointing.data-inline-threshold` gains that it also applies to coordinator state of coordinators that support it, when `execution.checkpointing.coordinator-state-in-file.enabled` is true and `execution.checkpointing.coordinator-state-inline-threshold` is not set. The checkpoint monitoring and metrics pages note that coordinator state in a file is counted in the checkpoint's full and checkpointed size, but in no vertex. The release notes describe version 7, the new options, and rolling back.

Cleanup

The file is an exclusive handle of its checkpoint. It is discarded with the checkpoint by the existing `OperatorState#discardState`: when the checkpoint is subsumed or aborted, or when the savepoint is disposed. The file of the checkpoint being restored cannot be subsumed before the reset finishes: until then, the coordinator's checkpoint call is deferred, so no newer checkpoint can complete. A reset that a later one has replaced may still be reading when its file is deleted; its result is discarded.

Compatibility, Deprecation, and Migration Plan

- Checkpoints and savepoints written by earlier versions restore as before.
- A checkpoint whose coordinator state all stays inline is written as version 6, readable by the same earlier versions as today (2.1 and later).
- A checkpoint with coordinator state in a file is written as version 7. Earlier versions reject it with "Unrecognized checkpoint version number: 7". Restoring a newer checkpoint on an older version is not supported in general, and Flink 2.0 and 2.1 each changed the metadata version for every checkpoint. Here the version changes only when `execution.checkpointing.coordinator-state-in-file.enabled` is true, and only for checkpoints in which a coordinator that restores asynchronously has state too large to stay inline.
- Rolling back an upgrade: once a checkpoint has coordinator state in a file, an earlier version can neither restore it (version 7) nor read the entry that references it in the HA store, which fails with a `ClassCastException`. Entries without coordinator state in a file stay readable. For this reason, `execution.checkpointing.coordinator-state-in-file.enabled` is disabled by default, so this FLIP does not affect rolling back unless the option is enabled. Users can enable it once they no longer need to roll back, and enabling it by default can be reconsidered once there is more production experience.
- Savepoints stay relocatable.

- Coordinators that do not opt in, and memory-backed checkpoint storage, behave as today.
- As accepted, FLIP-600 (regional checkpoints) lets a checkpoint reference subtask state from an earlier checkpoint, but produces coordinator state anew for every checkpoint. A coordinator state file therefore belongs to exactly one checkpoint, and the cleanup above still holds. FLIP-600 also adds a field to the metadata: its implementation (apache/flink#28963, under review) adds a version 7 of its own and writes it for every checkpoint. If it lands first, the format here becomes version 8 on top of it, still written only when some coordinator state is in a file.
- No migration steps are needed.

Test Plan

The prototype, against the master branch of 25 September 2026, includes unit tests for:

- Writing: large state of a coordinator that restores asynchronously goes to a file when the option is enabled, version 7 is written and the completed checkpoint keeps only the handle; small state, large state of other coordinators, and state on memory-backed storage stay inline with version 6; with the option disabled, no coordinator state may go to a file; with `execution.checkpointing.coordinator-state-inline-threshold` set, state below it stays inline even above the storage's threshold and above 1 MB, state at or above it goes to a file even below the storage's threshold, and it does not apply to other coordinators or while the option is disabled; the new threshold reaches `CheckpointCoordinator` from the job configuration; offloaded bytes are counted in the checkpoint's state size and are not repeated in the checkpoint stored for HA.
- Restoring: the reset reads the file in a separate thread without waiting for it, including a file whose size was not recorded; a failed read fails the job, also before the coordinator is initialized; a later reset takes precedence over one still reading; a later reset, or closing the coordinator, stops a read in progress by closing its stream outside the caller's thread, also while the stream is still being opened; `CheckpointCoordinator` and `OperatorCoordinatorHolder` hand file-backed state to any coordinator that supports it, and only to those.
- Loading and cleanup: savepoint loading and checkpoint store recovery do not read coordinator state files; the file is deleted when its checkpoint is subsumed, and when finalization fails partway.
- Format and tools: version 7 round trips, including relative paths; earlier serializers reject coordinator state in a file, when reading and when writing; a savepoint rewritten with the State Processor API contains the coordinator state file.

Disabling each part of the change in turn makes the tests covering it fail.

Flink's CI workflow (all modules and end-to-end tests, Java 17) passes on the prototype.

End-to-end MiniCluster tests compare stock master with the change: the stop-and-restore run in the Motivation, with the option enabled and disabled and with the new threshold above and below the state size, a global failover inside a running job, a savepoint rewritten with the State Processor API and restored after the original is deleted, and the JobManager heap held by retained checkpoints. Java-serialized `OperatorState` objects, as HA store entries contain them, were also exchanged between the two builds, with the results described under rolling back.

The implementation will also add an integration test with a JobManager failover under HA, and a savepoint compatibility test with a FLIP-27 source, since the existing savepoint migration tests

use the legacy SourceFunction.

Rejected Alternatives

Keep a copy of the state in JobManager memory

This keeps FLINK-17674's restriction for every coordinator: the bytes stay in memory next to the file handle, and state read back from storage is loaded where the checkpoint is read. It keeps the memory cost this FLIP removes. It also makes reading checkpoints depend on coordinator state files: an earlier prototype that loaded them while recovering the checkpoint store would not have been able to recover it after a still-retained NO_CLAIM snapshot was deleted, which the documentation allows once a checkpoint has succeeded.

Make every coordinator restore asynchronously

This would change the OperatorCoordinator restore contract for all implementations. Coordinators that already restore asynchronously cover every FLIP-27 source, which is where large state comes from. Others can opt in by implementing the two new methods.

Decide by the coordinator's class

Checking for `RecreateOnResetOperatorCoordinator` would keep `OperatorCoordinator` unchanged, but would give coordinators built differently no way to opt in.

Read the file on the main thread

That would put blocking I/O on the JobManager main thread, which is exactly what FLINK-17674 rules out.

Use only `execution.checkpointing.data-inline-threshold`

Earlier versions of this FLIP did this. It ties coordinator state to task state: keeping more coordinator state inline, for example to avoid small files just above the 20 KB default, means raising the threshold for task state too, and that threshold cannot be set above 1 MB.

Always store coordinator state in a file

Coordinator state that fits under its inline threshold gains nothing from a file of its own. An inline threshold already makes this decision for task state, and a file per coordinator per checkpoint would add small files for no benefit.

Leave it to each connector

Connectors should still keep their state bounded, and some do: Flink CDC 3.7 can release split metadata once it is no longer needed (opt-in), and FLINK-25672 proposes bounds for the `FileSource`. That limits how large the state grows. It does not change where the state is stored

or how many copies the JobManager keeps.

Always write version 7

Every new checkpoint would become unreadable by earlier versions, even with no coordinator state in a file.