FLIP-158: Generalized incremental checkpoints

Discussion thread	http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/DISCUSS-FLIP-158-Generalized-incremental-checkpoints-tp47902.html
Vote thread	http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/VOTE-FLIP-158-Generalized-incremental-checkpoints-td48485.html
JIRA	~~FLINK-21352~~ - FLIP-158: Generalized incremental checkpoints RESOLVED ~~FLINK-25842~~ - [v2] FLIP-158: Generalized incremental checkpoints RESOLVED
Release	1.15

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

- Motivation
- High-level Overview
- Public Interfaces
- Proposed Changes
  - General design
    - Performing a checkpoint
    - Recovery and rescaling
    - Cleanup
  - StateBackend
    - Tracking the changes
    - Materializing state changes
  - StateChangelog
    - DFS write latency
    - API
    - Implementation notes
    - File contents layout
    - State change lifecycle
  - Example
- Compatibility, Deprecation, and Migration Plan
- Test Plan
- Future directions
- Rejected Alternatives
  - StateChangelog implementations

Motivation

The goal of this FLIP is to establish a way to drastically reduce the checkpoint interval for streaming applications, across state backends, reliably for both small and large scales. We are aiming at intervals in the order of few seconds even for larger scales (> 100 nodes, TBs of state).
Depending on user adoption of this feature and further requirements, the architecture here can also serve as the a foundation to further reduce the checkpoint interval in the future.

A faster checkpoint interval has a number of benefits for streaming applications:

- Less work on recovery. The more frequently the checkpoint, the fewer events need to be re-processed after recovery.
- Lower latency for transactional sinks: Transactional sinks commit on checkpoint, so faster checkpoints mean more frequent commits.
- More predictable checkpoint intervals: Currently the length of the checkpoint depends on the size of the artifacts that need to be persisted on the checkpoint storage.
  For example, if RocksDB created only a new Level-0 SST since the last checkpoint, the checkpoint will be fast.
  But if RocksDB finished a new compaction and created a large SST for Level-3/-4/-5, the checkpoint will take longer.
- A frequent checkpoint interval allows Flink to persist sink data in a checkpoint before writing it to the external system (write ahead log style), without adding too much latency. This can simplify the design of sinks for systems that don't expose transactional APIs well. For example the exactly-once Kafka sink is currently quite complex, due to way Kafka's transactions work, specifically the lack to recover transactions well (and rather relying on transactions timing out).

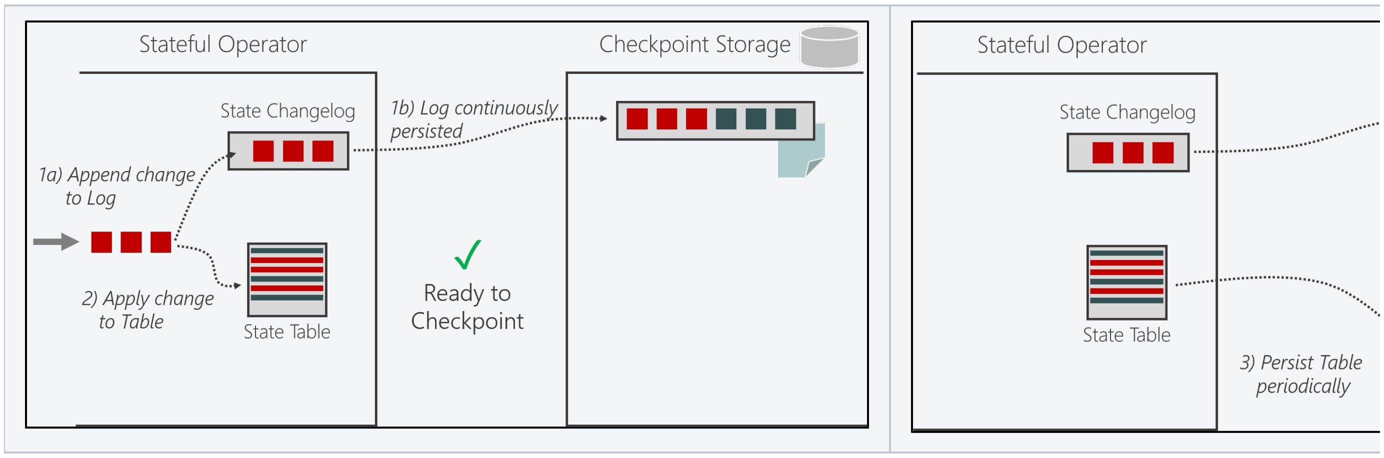
In addition, the here proposed approach will also help to reduce the small file fragmentation issue that can occur with using RocksDB with incremental checkpoints.

High-level Overview

The core idea of this proposal is to introduce a ***state changelog***; this changelog allows operators to persist state changes in a very fine-grained manner, as described below:

- Stateful operators write the state changes to that log (*logging the state*), in addition to applying them to the *state tables* in RocksDB or the in-mem Hashtable.
- An operator can acknowledge a checkpoint as soon as the changes in the log have reached the durable checkpoint storage.

- The state tables are persisted periodically, independent of the checkpoints. We call this the *materialization* of the state on the checkpoint storage.
- Once the state is materialized on checkpoint storage, the state changelog can be truncated to the corresponding point.



This approach mirrors what database systems do, adjusted to distributed checkpoints:

- Changes (inserts/updates/deletes) are written to the transaction log, and the transaction is considered durable once the log is synced to disk (or other durable storage).
- The changes are also materialized in the tables (so the database system can efficiently query the table). The table are usually persisted asynchronously (blocks are flushed to storage at a later point).
- Once all relevant parts of the changed tables have been persisted, the transaction log can be truncated from the perspective of that particular transaction. That truncation procedure is commonly, and not coincidentally, called a "checkpoint"; the similarities here really go quite far.

We will call the component that manages the *state changelog* henceforth **"Durable Short-term Log" (DSTL)**. This name is chosen to clarify and emphasize the difference in requirements and usage compared to logs as implemented by systems like Apache Kafka, Apache Pulsar, or Pravega:

- The DSTL is always written to, but rarely read. It holds data for seconds to minutes.
- Logs Kafka, Pulsar, Pravega are often read more than they are written (have multiple subscribers/consumers) and hold data for hours to months.

The section "StateChangelog implementations" has more details on the requirements and comparison of log implementations.

Public Interfaces

- No changes to the public APIs (except for configuration)
- Configuration
  - Toggle to enable
  - StateChangelog: persist delay, retry policy, file prefix
  - State backends: materialization interval, file prefix
- Monitoring
  - StateChangelog: DFS request size, latency, number of “Logs” per request, errors
  - State backends: DFS request size, latency, errors

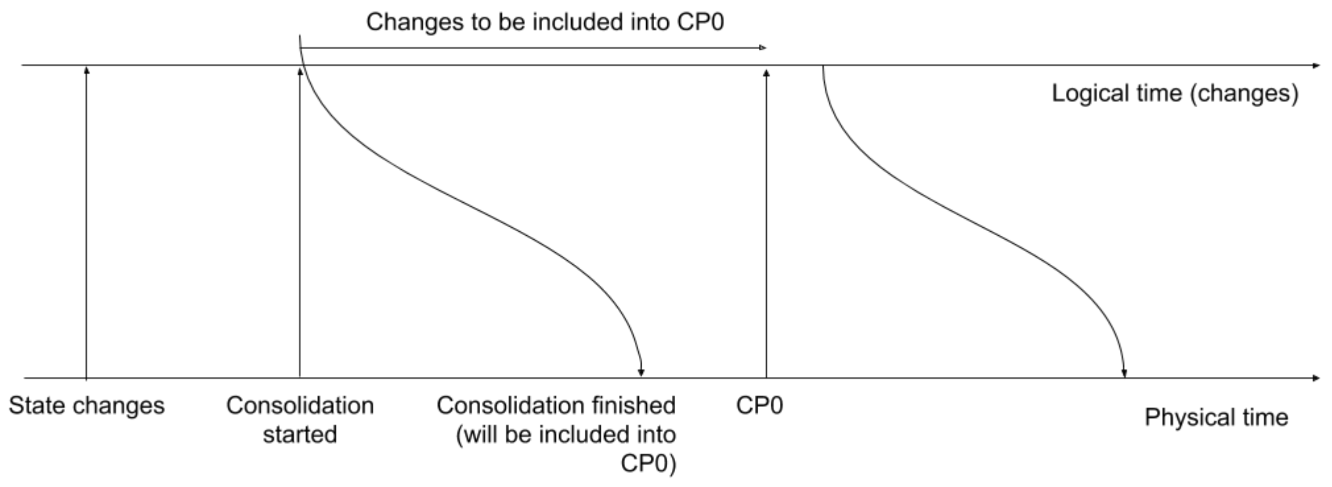
Proposed Changes

General design

The idea to get faster checkpoints is similar to the database write-ahead log:

1. For immediate durability, the changes are logged durably (WAL, TXN-LOG, etc.); this is done as fast as possible to allow further processing while allowing to restore in case of crash
2. For efficient querying and storing, "materialization" into the primary structure (table or index) happens at some point later. The durability of that structure can then be fully decoupled (because there is already the log)

Applying this to Flink: all state changes are written to a log first. To acknowledge the checkpoint, only these changes need to be persisted durably. In the background, a slower materialization activity is triggered; upon its completion, the log head can be truncated.


Changes to be included into CP0
Logical time (changes)
State changes
Consolidation started
Consolidation finished (will be included into CP0)
CP0
Physical time

Two new components are introduced:

1. A new state backend that
    a. saves all state changes in a StateChangelog - in addition to applying them to its internal structures
    b. snapshots these internal structures (materializes) - independently from a checkpoint
2. StateChangelog: durably persist the changes received from StateBackend (and, depending on the design, provides the read access)

StateBackend maintains a logical timestamp to draw a boundary between the consolidated and unconsolidated changes.

Performing a checkpoint

On checkpoint, StateBackend requests StateChangelog to persist durably all the changes made after the latest (known) materialization (identified by a logical timestamp). The result is a regular (Keyed)StateHandle. StateChangelog guarantees that this handle only provides access to changes that are:

1. Made by this **backend instance** (so we don't have to filter changes or deal with split-brain scenarios)
2. Made **after the materialization** (so the logical timestamp is not needed after the handle is built)

As usual, persisting is done asynchronously, so the usual processing can be continued. Upon completion, subtask acknowledges the checkpoint.

For a single backend, a snapshot consists of:

1. A handle for a (previously) materialized state (roughly the same as the current state handle)
2. If (1) doesn't exist then state handles from the last confirmed snapshot
3. A handle for the changes made since then and saved by the StateChangelog (potentially multiple to support many concurrent checkpoints)

The details of StateChangelog handles are discussed below (API and File contents layout).

Note that everything above is:

- as of time of the sync phase
- included as shared state handles
- optional

For savepoint, the call is just proxied to the corresponding method of the underlying StateBackend.

Recovery and rescaling

On JobMaster, state assignment doesn't need to be changed because regular handles are used (that's also true for rescaling).

On StateBackend, the previously materialized state is restored first (if any); and then the changelog is applied. It doesn't have to be filtered for already materialized changes - that's ensured during state handle creation. However, it has to be filtered out for irrelevant records on upscaling. For that, state changes must be complemented with the keygroup (the non-keyed state is not supported in the initial version).

Cleanup

Cleanup when subsuming old checkpoint

Existing SharedStateRegistry mechanisms suffice here: once a checkpoint is subsumed, reference counters for its state objects are decremented; and once they are zeroed, state objects are destroyed.

Preventing excessive metadata growth

There can be a case when the data rate is low and checkpointing is frequent. For example, 1 event per checkpoint and 10 checkpoints per second. That means there will be 10 new handles per second per backend instance. If consolidated, this can pile up quickly and blow JobMaster. There are several ways to deal with it:

1. Merge handles on JM into a single handle
   a. Depending on StateChangelog implementation, it may or may not reduce the actual amount of data in the handle. With a DFS-based approach proposed below, handles will refer to different files, which couldn't be merged. Using name pattern can pose some other challenges
   b. Requires exposing StateChangelog implementation details
   c. Performing more work on JM (not much though)
2. Trigger materialization at a fixed rate, rather than on underlying backend compaction. For Incremental RocksDB, materialization will be cheap (not many changes). But other backends could be constantly uploading slightly updated state, which may still not be fast enough.

The 2nd option is much simpler and with DFS-based StateChangelog should be enough.

Cleanup on shutdown

During a normal shutdown, StateBackend asks StateChangelog to delete any state changes which weren't requested to be persisted durably. StateBackend also removes any uploaded materialized files that are not included in any checkpoint.

Cleanup after recovery

There is a chance that the above step didn't complete, e.g. due to a crash. Orphaned files on DFS can include materialized and non-materialized changes (depending on StateChangelog implementation).

To remove them, upon recovery JM lists the files in the job checkpointing directory and removes everything not referenced by the checkpoints. To prevent the removal of new pending files, this has to be done before checkpointing starts.

Note, that this is an existing issue, so we intend to implement the cleanup in the subsequent version. Existing (non-changelog) StateBackends will also benefit from this mechanism.

StateBackend

Tracking the changes

This can be done in the respective "State" implementations: MapState, ValueState, ListState, etc. For that, a wrapping layer can be added around KeyedStateBackend (and probably some supporting classes, such as RegisteredKeyValueStateBackendMetaInfo).

These state changes are sent to StateChangelog but not necessarily durably persisted yet.

Materializing state changes

As discussed in "Preventing excessive metadata growth", materialization must be triggered periodically (and at most one at a time).

Conceptually, it consists of the following steps:

1. Remember the current logical timestamp
2. Snapshot underlying state (by calling SnapshotStrategy.snapshot()). This can trigger snapshot or checkpoint for RocksDB or a full snapshot for Incremental Heap backend
3. (Asynchronously) upload the snapshot to DFS by executing the RunnableFuture obtained in the previous step
4. Save the timestamp from step (1) as the last materialized timestamp

Steps 1 and 2 need to be done atomically; no state changes can be performed in between. Therefore, snapshotting (step 2) should be fast. For synchronization, Task Mailbox can be used (steps 1+2 and 4).

Note: for Incremental RocksDB, materialization does not necessarily trigger the compaction.

StateChangelog

At least for the first version, a **DFS**-based solution was chosen (see the Alternatives section for a discussion of other approaches). In this approach, StateChangelog is a stateless layer which bundles writes and translates them to write requests to DFS.

Colocation inside TMs with operators simplifies coordination and reduces latency.

To reduce the number of files and requests to DFS, StateChangelog can **batch** requests on multiple levels:

1. All changes from a single StateBackend for a single checkpoint
2. All backends in a single subtask (OperatorChain)
3. All subtasks in a single TM (the same job)
4. *In the future we may also decide to batch requests from multiple TMs (see Future directions)*

To satisfy the requirements discussed in the previous sections:

1. Each StateBackend instance is associated with a unique LogID upon startup to isolate instances from one another (see Performing a checkpoint)

2. StateChangelog maintains a mapping between logical timestamps and state changes to be able to include only changes after the materialization (see Performing a checkpoint)
3. keygroup is passed and written along with the changes (see Rescaling)
4. StateHandle key used in SharedStateRegistry should only be based on the file name and not on backend UUID or keygroup (multiple handles can refer to the same file and making keys unique will effectively disable sharing)

DFS write latency

From the experiments, the latency of writing 2.5Gb to S3 (500 streams of 5Mb objects) is:

p50	p90	p95	p99	p999
459ms	740ms	833ms	1039ms	3202ms

If everything below second is considered acceptable on this scale then tail latencies become the major problem. Below are some techniques to reduce it.

Request hedging

The idea is to aggressively retry a small portion of requests that take much longer than others (see https://www2.cs.duke.edu/courses/cps296.4/fall13/838-CloudPapers/dean_longtail.pdf).

With a simple implementation, p999 latency decreases by 47% (with smaller requests the improvement is bigger):

p50	p90	p95	p99	p999
495ms	733ms	807ms	937ms	1689ms

Assuming 500 writers, these request latencies translate to checkpoints latencies of p50=1000ms and p99=1700s. For .5Gb state increment, checkpoints latencies would be p75=500ms and p99=1125ms (assuming 250 writers).

Other techniques

Additionally, these techniques can be used:

1. Adjusting aggregation: change target request size (or total request count); this is a tradeoff between median and tail latencies. Implies distributed implementation; automatic adjustment can be unstable.
2. Multipart upload (for S3, files bigger than 5Mb). Can be useful only with aggressive aggregation; otherwise, requests are likely to be smaller than 5Mb.
3. Use multiple buckets (in S3, throttling is applied on bucket level until re-balance). Probably makes sense only for very large setups

These are already implemented or can be configured:

1. Unique prefixes (“folders”)
2. S3 DNS load balancing for S3

API

Note: this is an internal API and may change in the future.

To enable StateChangelog to flush intermediate data, StateBackend should be able to append changes as they happen and only request to persist them durably on the checkpoint. Therefore, StateChangelog must be aware of the backend logical timestamp (SQN in code).

Furthermore, to avoid too fine granular SQN-to-change mappings and more efficient batching, SQN should be generated by StateChangelog and exposed to the backend via the getLastAppendedSqn() method. So there are two usages of it:

1. Materializing state changes - Remember the current logical timestamp
2. Performing a checkpoint (to provide SQN for persistDurably)

```
/** Scoped to a single entity (e.g. a SubTask or OperatorCoordinator). */
interface StateChangelogClient {
    StateChangelogWriter createWriter(OperatorID operatorID, KeyGroupRange keyGroupRange);
}

/** Scoped to a single writer (e.g. state backend). */
interface StateChangelogWriter {

  void append(int keyGroup, byte[] value);

  CompletableFuture<StateChangelogHandle> persistDurably(SQN after);

  void truncate(SQN before); // called upon checkpoint confirmation by JM

  /** Close this log. No new appends will be possible. Any appended but not persisted records will be lost. */
  void close();

  SQN lastAppendedSqn();
}

interface StateChangelogHandle extends KeyedStateHandle {

  /** Enumerate changes per key group. */
  CloseableIterable<byte[]> getStateChanges(int keyGroup);
}
```

Implementation notes

- Because many backends may request to persist changes durably at different times (even for the same checkpoint), StateChangelog waits for some time after the first request and only then batches the changes and sends them to DFS. Besides that, it sends a request as soon as it gets persist requests from all the backends registered with it. Additionally, size threshold may be used.
- Upon receiving a request (from StateBackend) to persist durably starting from some SQN, changes before that SQN can NOT be dropped - JM can still decline the checkpoint
- Request hedging can be implemented in StateChangelog, Flink FS layer, custom FS layer per-provider, custom per-provider configuration. Implementing in StateChangelog gives a good trade-off between efficiency, portability, and effort. It also doesn't affect other parts of the system.

File contents layout

This is an example layout inside a single file:

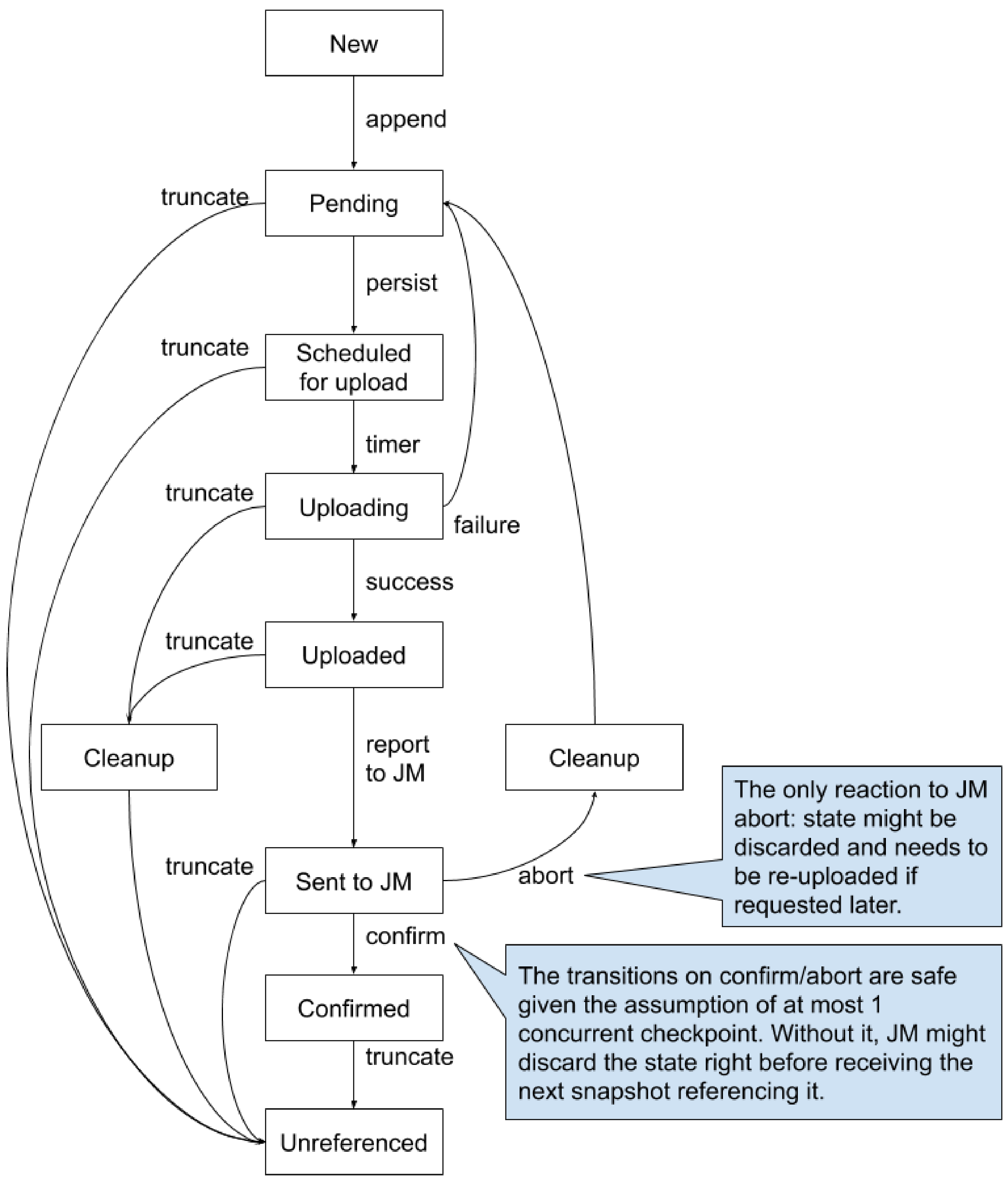
```
+----------------------------------------------------------------------------------------------+
| File: ABCD.bin (all subtasks of TM1 for checkpoint 2)                                        |
| +------------------------------------------------------+   +-------------------------------+ |
| | Log 0 ("group-by" operator)                          |   | Log 1 ("aggregate" operator)  | |
| | +--------------------------------------------------+ |   | +---------------------------+ | |
| | | +---------------------+  +---------------------+ | |   | | +---------------------+   | | |
| | | | KeyGroup 0          |  | KeyGroup 1          | | |   | | | KeyGroup 0          |   | | |
| | | | +-----------------+ |  | +-----------------+ | | |   | | | +-----------------+ |   | | |
| | | | | Changeset 1     | |  | | Changeset 1     | | | |   | | | | Changeset 1     | |   | | |
| | | | | window-contents | |  | | window-contents | | | |   | | | |                 | |   | | |
| | | | | timers          | |  | | timers          | | | |   | | | |                 | |   | | |
| | | | +-----------------+ |  | +-----------------+ | | |   | | | +-----------------+ |   | | |
| | | | +-----------------+ |  | +-----------------+ | | |   | | | +-----------------+ |   | | |
| | | | | Changeset 2     | |  | | Changeset 2     | | | |   | | | | Changeset 2     | |   | | |
| | | | | window-contents | |  | | window-contents | | | |   | | | |                 | |   | | |
| | | | | timers          | |  | | timers          | | | |   | | | |                 | |   | | |
| | | | +-----------------+ |  | +-----------------+ | | |   | | | +-----------------+ |   | | |
| | | | +-----------------+ |  | +-----------------+ | | |   | | | +-----------------+ |   | | |
| | | | | Changeset 3     | |  | | Changeset 3     | | | |   | | | | Changeset 3     | |   | | |
| | | | | window-contents | |  | | window-contents | | | |   | | | |                 | |   | | |
| | | | | timers          | |  | | timers          | | | |   | | | |                 | |   | | |
| | | | +-----------------+ |  | +-----------------+ | | |   | | | +-----------------+ |   | | |
| | | +---------------------+  +---------------------+ | |   | | +---------------------+   | | |
| | +--------------------------------------------------+ |   | +---------------------------+ | |
| +------------------------------------------------------+   +-------------------------------+ |
+----------------------------------------------------------------------------------------------+
```

A handle returned to StateBackend contains the file name and Log offset. If StateChangelog decides to create multiple files (in case of too many changes) it can be an ordered collection of tuples (file, offset) or handles.

Changes themselves are serialized by StateBackend and are not transparent to StateChangelog. However, a header may be needed to hold some metadata (such as state IDs or serializer states).

There will be likely a single changeset per keygroup but there is no such guarantee (each time StateBackend requests lastAppendedSqn a new changeset is started).

State change lifecycle


New
append
Pending
truncate
persist
Scheduled for upload
truncate
timer
Uploading
truncate
failure
success
Uploaded
truncate
Cleanup
report to JM
Cleanup
Sent to JM
truncate
abort
The only reaction to JM abort: state might be discarded and needs to be re-uploaded if requested later.
confirm
The transitions on confirm/abort are safe given the assumption of at most 1 concurrent checkpoint. Without it, JM might discard the state right before receiving the next snapshot referencing it.
Confirmed
truncate
Unreferenced

Example

Let's consider a job with one stateful operator having a list state in RocksDB. Assume max retained checkpoints: 1.

On startup, the state is empty StateChangelog.**sqn = T0** StateBackend.lastMaterializedSqn = T0

list.add(a) During normal processing, operator updates its state and simultaneously adds state changes to the StateChangelog. Logical time is not updated.
CP1 triggered StateBackend calls StateChangelog.persisDurably(after = lastMaterializedSqn = T0). StateChangelog**.sqn = T1** (as a result of the call above) StateChangelog returns a Future representing the completion of write of (T0:T1 = a) and remembers it internally.
list.add(b)
State materialization is triggered StateBackend obtains the sqn from StateChangelog (T2) StateChangelog.**sqn = T2** (as a result of the call above) StateBackend flushes RocksDB memtables to SST files; starts the async phase passing it obtained sqn=T2 and snapshot (~list of files). Materialized state will be (T0:T2 = a,b)
list.add(c)
CP2 triggered StateBackend calls StateChangelog.persistDurably(after = lastMaterializedSqn = T0). StateChangelog.**sqn = T3** (as a result of the call above) StateChangelog finds the previous Future (T0:T1 = a) and combines it with a new one (T1:T3 = b,c).
CP1-Future completes It contains a shared StateHandle pointing to a file on S3. This handle is included in TaskSnapshot and sent to JM.
CP2-Future completes It contains a combined StateHandle pointing to two files on S3. StateHandle is included in TaskSnapshot and is sent to JM.
JM finalizes CP1, and then CP2 CP1 is subsumed but nothing is removed yet.
State materialization completes StateBackend stores the consolidated state handle (T0:T2 = a,b) StateBackend.lastMaterializedSqn = T2 StateChangelog.truncate(before = T2) // remove (T0:T1 = a) object reference (not any state that is already saved or is being saved)
list.add(d)
CP3 is triggered StateBackend calls StateChangelog.persisDurably(after = lastMaterializedSqn = T2). StateChangelog.**sqn = T4** (as a result of the call above) StateChangelog searches for reference matching the requested (T2:*) range. It finds completed (T1:T3 = b,c). To be able to filter out (T1=b) on recovery, the returned handle specifies the range (T2:T4). Files also specify the range for each changeset internally. Found Future is combined with a new one for (T3:T4 = d).
CP3-Future completes StateBackend combines its result with the consolidated handle and sends to JM: • (T0:T1 = a,b) - consolidated changes • (T1:T3 = b,c) - previously saved changelog (“b” filtered on recovery) • (T3:T4 = d) - the newly saved changelog

JM finalizes CP3

CP2 is subsumed, the state handle pointing to the file with (T0:T1 = a) is not referenced anymore and so removed.

Compatibility, Deprecation, and Migration Plan

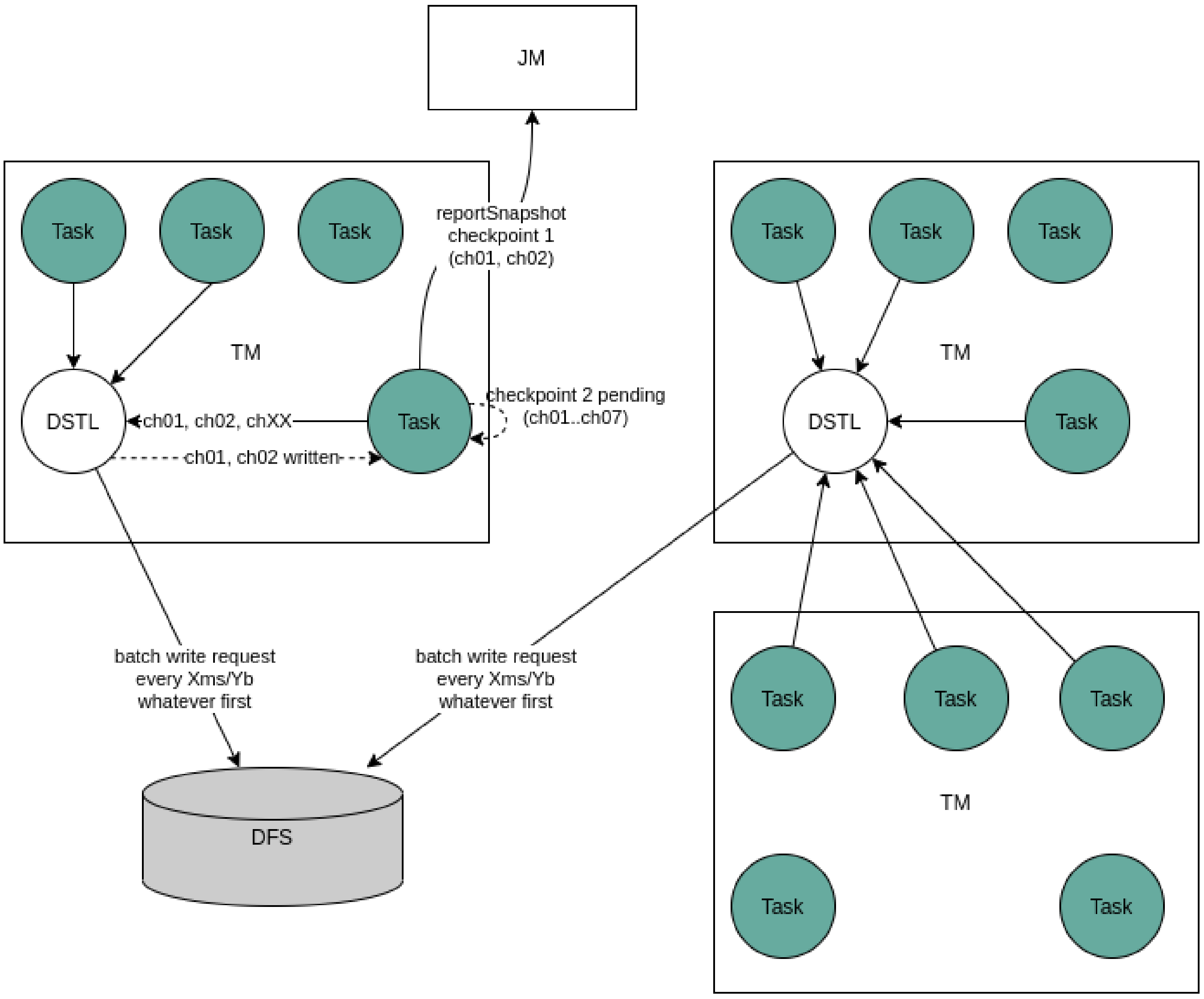
- It should be possible to load the existing state as a consolidated state without any changes (but not vice-versa)
- Only keyed state in the initial version
- At most 1 concurrent checkpoint
- Probably, only RocksDB in the initial version
- Nothing is deprecated and no migration needed
- Changeset of a checkpoint must fit in memory

Test Plan

Existing ITCases with the feature enabled.

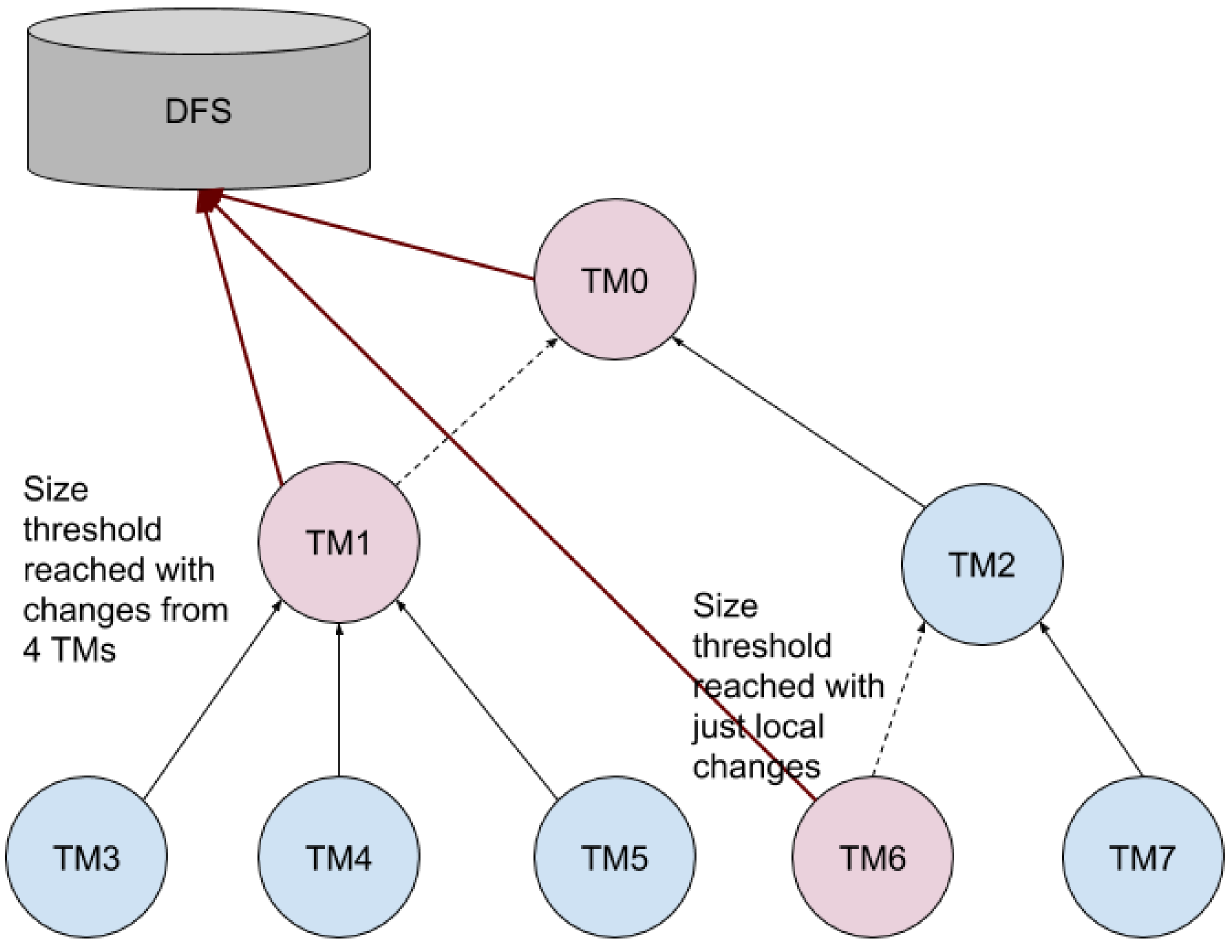
Future directions

The chosen approach allows to implement a distributed StateChangelog which may look like this:


JM
reportSnapshot checkpoint 1 (ch01, ch02)
Task
TM
DSTL
ch01, ch02, chXX
ch01, ch02 written
checkpoint 2 pending (ch01..ch07)
batch write request every Xms/Yb whatever first
DFS

A number of questions arise, such as which node to send data to? Who and how decides to rescale the “active” set of nodes? How is this decision communicated?

One possible approach is to organize distributed StateChangelog nodes into a hierarchy/DAG: each node proxies requests up to the root unless it has accumulated enough data to send straight away to DFS:


DFS
TM0
TM1
TM2
Size threshold reached with changes from 4 TMs
Size threshold reached with just local changes
TM3
TM4
TM5
TM6
TM7

It has the following advantages:

- No need for load-balancing or rescaling
- No centralized coordination
- Making a decision is relatively easy (no additional metrics needed)
- Decisions are dynamic
- The topology is static (which helps in case of a potential single-task failover)
- Each node needs only a single connection to send the data

The disadvantages are:

- Additional round-trips - can be reduced by choosing a higher branching factor

Unnecessary traffic through the intermediate nodes - can be avoided by making a decision upfront for the next N checkpoints on each node and communicating downstream. In the figure above, TM0 address would be propagated to TM2 and TM7; TM1 will send its own address to its descendants.

Rejected Alternatives

StateChangelog implementations

Besides DFS-based, some other options were considered. These are intentionally very rough estimates of different solutions:

	Kafka-based unmanaged	Bookkeeper-based unmanaged	Bookkeeper managed	DFS-based	Custom persistence
1. Latency	Good	Good	Good	Bad (.1 - 1s)	Good

2. Scalability	Good	Unknown	Unknown	Medium	Good
3. Ease of deployment	Medium (many users already have but may still need to adjust the cluster)	Bad	Good	Good	Good
4. Stateless or no new stateful component	Yes (existing)	No	No	Yes (stateless)	No
5. Development time	Good	Medium	Bad	Good	Very bad
6. Other issues	Can truncate changes			High variability (workload, provider, …)	

With the highest priorities for the initial version being (3), (4), (5) DFS was chosen.