

KIP-844: Transactional State Stores

- [Status](#)
- [Motivation](#)
- [Public Interfaces](#)
- [Proposed Changes](#)
 - [Background](#)
 - [Overview](#)
 - [StateStore changes](#)
 - [Behavior changes](#)
 - [Configuration changes](#)
 - [Interface Changes](#)
- [Compatibility, Deprecation, and Migration Plan](#)
- [Test Plan](#)
- [Transactions via Secondary State Store for Uncommitted Changes](#)
 - [Rejected Alternatives](#)
 - [RocksDB in-memory Indexed Batches](#)
 - [RocksDB Optimistic Transactions](#)
 - [Method to control transaction lifecycle in StateStore](#)

Status

Current state: *Accepted*

Discussion thread: [here](#)

JIRA: [here](#)

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

Motivation

Right now, a stream processor with EOS has to delete all data from the local state stores after crash failure because the state stores might be in a partially updated state. The partial update of a state store can happen during a crash failure because the changes to the local state are not atomic with respect to Kafka Streams commit. If an application with EOS crashes between commits, it cannot reset the state to the previously committed, so it wipes the state stores and replays the changelog from scratch.

This KIP proposes making writes to the state stores transactional so that they atomically commit only after the corresponding changes are committed to the changelog topic. As a result, Streams applications configured with EOS will no longer need to wipe the state stores on crash failure.

Public Interfaces

Changed:

- `org.apache.kafka.streams.processor.StateStore`
- `org.apache.kafka.streams.state.Stores`
- `org.apache.kafka.streams.kstream.Materialized.StoreType`

Proposed Changes

Background

This section briefly describes relevant parts of a stateful task's write lifecycle with EOS.

1. The task ([StreamTask](#), [StandbyTask](#)) registers its state stores. State stores load offset metadata from the checkpoint file ([link](#)). That step aims to establish a mapping between data in the state store and the offset of the changelog topic.
 - a. In case of crash failure, if the state store has data, but the checkpoint file does not exist, `ProcessorStateManager` throws an exception for EOS tasks. This is an indicator to throw away local data and replay the changelog topic ([link](#)).
2. The task processes data and writes its state locally.
3. The task commits EOS transaction. `TaskExecutor#commitOffsetsOrTransaction` calls `StreamsProducer#commitTransaction` that sends new offsets and commits the transaction.
4. The task runs a `postCommit` method ([StreamTask](#), [StandbyTask](#)) that:
 - a. flushes the state stores and
 - b. updates the checkpoint file ([link](#)) for non-EOS tasks ([link](#)).
5. Go to step 2 until task shuts down. During shutdown, the task stops processing data, then writes its current offset to the checkpoint file and halts.

If the failure happens at steps 2 or 3, the state store might contain records that have not yet been committed by EOS transaction. These uncommitted records violate the EOS guarantees and are the reason why Kafka Streams deletes state store data if EOS is enabled.

Overview

This section introduces an overview of the proposed changes. The following sections will cover the changes in behavior, configuration, and interfaces in detail.

This KIP introduces persistent *transactional state stores* that

1. distinguish between uncommitted and committed data in the state store
2. guarantee atomic commit, meaning that either all uncommitted (dirty) writes will be applied together or none will.

These guarantees are sufficient to prevent the failure scenario described in the previous section.

This proposal deprecates the `StateStore#flush` method and introduces 2 other methods instead - `StateStore#commit(changelogOffset)` and `StateStore#recover(changelogOffset)` that commit the current state at a specified offset and recover from the crash failure to a previously checkpointed offset accordingly. With these changes, the lifecycle of a stateful task with transactional state stores becomes:

1. The task registers its state stores. The state stores call `StateStore#recover` that discards uncommitted data.
2. The task processes data and writes new records as uncommitted.
3. The task commits the EOS transaction.
4. The task runs a `postCommit` method that:
 - a. commits dirty writes.
 - b. updates the checkpoint file.
5. The task shuts down.

Consider possible failure scenarios:

- The crash happens between steps 1 and 3. The uncommitted data will be discarded. The input records were not committed via the EOS transaction, so the task will re-process them.
- The crash happens between 3 and 4a. The EOS transaction has been already committed, but the state store hasn't. The state store will replay the uncommitted records from the changelog topic.
- The crash happens between 4a and 4b. The state store has already committed the new records, but they are not yet reflected in the checkpoint file. The state store will replay the last committed records from the changelog topic. This operation is idempotent and does not violate correctness.
- The crash happens after step 4b. The state store does nothing during recovery.

There are multiple ways to implement state store transactions that present different trade-offs. This proposal includes a single reference implementation via a secondary RocksDB for uncommitted writes.

StateStore changes

This section covers multiple changes to the state store interfaces. This proposal replaces `StateStore#flush` with 2 new methods - `StateStore#commit(Long)` and `StateStore#recover(long)` and adds a boolean `transactional()` method to determine if a state store is transactional.

The reasoning behind replacing `flush` with `commit/recover` is two-fold. First, let's talk about why we don't need both `flush` and `commit`:

1. There is always a single writer in Kafka Streams workloads, and all writes must go to a single currently open transaction.
2. There is always a single reader that queries dirty state from a single open transaction.
3. The state stores already explicitly call `flush` after AK transaction commits before writing to the checkpoint file to make uncommitted changes durable. Adding a separate method will create room for error, such as a missing `commit` call or executing both commands in the wrong order.

In this sense, `flush` and `commit` are semantically the same thing. The separation of concerns between these 2 methods will be ambiguous and it is unclear what is the correct call order.

The purpose of `StateStore#recover(long changelogOffset)` method is to transition the state store to a consistent state after crash failure. This method discards any changes that are not yet committed to the changelog topic and ensures that its state corresponds to the offset that is greater than or equal to the checkpointed `changelogOffset`.

Behavior changes

If `StateStore#transactional()` returns `true`, then the store performs writes via the implementation-specific transactional mechanism. Reads via `ReadOnlyKeyValueStore` methods return uncommitted data from the ongoing transaction.

A transactional state store opens the first transaction during initialization. It commits on `StateStore#commit` - first, the store commits the transaction, then flushes, then starts a new transaction.

There are several places where `StreamTask`, `ProcessorStateManager`, and `TaskManager` check if EOS is enabled, and then it deletes the checkpoint file on crash failure, specifically, when:

- `StreamTask` resumes processing ([link](#))
- `ProcessorStateManager` initializes state stores from offsets from checkpoint ([link1](#), [link2](#))
- `StreamTask` writes offsets to the checkpoint file on after committing ([link](#))
- `TaskManager` handles revocation ([link](#))

If EOS is enabled, we will remove offset information for non-transactional state stores from the checkpoint file instead of just deleting the file.

Configuration changes

StreamsConfig

`default.dsl.store` has a new valid value - `txn_rocksDB` that enables transactional RocksDB state store.

Interface Changes

StateStore

StateStore.java

```
/**
 * Return true the storage supports transactions.
 *
 * @return {@code true} if the storage supports transactions, {@code false} otherwise
 */
@Evolving
default boolean transactional() {
    return false;
}

/**
 * Flush any cached data
 *
 */
@Deprecated
default void flush() {}

/**
 * Flush and commit any cached data
 * <p>
 * For transactional state store commit applies all changes atomically. In other words, either the
 * entire commit will be successful or none of the changes will be applied.
 * <p>
 * For non-transactional state store this method flushes cached data.
 *
 * @param changelogOffset the offset of the changelog topic this commit corresponds to. The
 *                        offset can be null if the state store does not have a changelog
 *                        (e.g. a global store).
 * @code null}
 */
@Evolving
default void commit(final Long changelogOffset) {
    if (transactional()) {
        throw new UnsupportedOperationException("Transactional state store must implement StateStore#commit");
    } else {
        flush();
    }
}

/**
 * Recovers the state store after crash failure.
 * <p>
 * The state store recovers by discarding any writes that are not committed to the changelog
 * and rolling to the state that corresponds to {@code changelogOffset} or greater offset of
 * the changelog topic.
 *
 * @param changelogOffset the checkpointed changelog offset.
 * @return {@code true} if the state store recovered, {@code false} otherwise.
 */
@Evolving
default boolean recover(final long changelogOffset) {
    if (transactional()) {
        throw new UnsupportedOperationException("Transactional state store must implement StateStore#recover");
    }
    return false;
}
```

Stores

```
/**
 * Create a persistent {@link KeyValueBytesStoreSupplier}.
 * <p>
 * This store supplier can be passed into a {@link #keyValueStoreBuilder(KeyValueBytesStoreSupplier, Serde,
 Serde)}.
 * If you want to create a {@link TimestampedKeyValueStore} you should use
 * {@link #persistentTimestampedKeyValueStore(String)} to create a store supplier instead.
 *
 * @param name name of the store (cannot be {@code null})
 * @param transactional whether the store should be transactional
 * @return an instance of a {@link KeyValueBytesStoreSupplier} that can be used
 * to build a persistent key-value store
 */
public static KeyValueBytesStoreSupplier persistentKeyValueStore(final String name, final boolean
transactional)

/**
 * Create a persistent {@link KeyValueBytesStoreSupplier}.
 * <p>
 * This store supplier can be passed into a
 * {@link #timestampedKeyValueStoreBuilder(KeyValueBytesStoreSupplier, Serde, Serde)}.
 * If you want to create a {@link KeyValueStore} you should use
 * {@link #persistentKeyValueStore(String)} to create a store supplier instead.
 *
 * @param name name of the store (cannot be {@code null})
 * @param transactional whether the store should be transactional
 * @return an instance of a {@link KeyValueBytesStoreSupplier} that can be used
 * to build a persistent key-(timestamp/value) store
 */
public static KeyValueBytesStoreSupplier persistentTimestampedKeyValueStore(final String name, final boolean
transactional)

/**
 * Create a persistent {@link WindowBytesStoreSupplier}.
 * <p>
 * This store supplier can be passed into a {@link #windowStoreBuilder(WindowBytesStoreSupplier, Serde,
 Serde)}.
 * If you want to create a {@link TimestampedWindowStore} you should use
 * {@link #persistentTimestampedWindowStore(String, Duration, Duration, boolean)} to create a store supplier
 instead.
 *
 * @param name name of the store (cannot be {@code null})
 * @param retentionPeriod length of time to retain data in the store (cannot be negative)
 * (note that the retention period must be at least long enough to contain the
 * windowed data's entire life cycle, from window-start through window-end,
 * and for the entire grace period)
 * @param windowSize size of the windows (cannot be negative)
 * @param retainDuplicates whether or not to retain duplicates. Turning this on will automatically disable
 caching and means that null values will be ignored.
 * @param transactional whether the store should be transactional
 * @return an instance of {@link WindowBytesStoreSupplier}
 * @throws IllegalArgumentException if {@code retentionPeriod} or {@code windowSize} can't be represented as
 {@code long milliseconds}
 * @throws IllegalArgumentException if {@code retentionPeriod} is smaller than {@code windowSize}
 */
public static WindowBytesStoreSupplier persistentWindowStore(final String name,
                                                             final Duration retentionPeriod,
                                                             final Duration windowSize,
                                                             final boolean retainDuplicates,
                                                             final boolean transactional) throws
IllegalArgumentException

/**
 * Create a persistent {@link WindowBytesStoreSupplier}.
 * <p>
 * This store supplier can be passed into a {@link #windowStoreBuilder(WindowBytesStoreSupplier, Serde, Serde)}.
```

```

* If you want to create a {@link TimestampedWindowStore} you should use
* {@link #persistentTimestampedWindowStore(String, Duration, Duration, boolean)} to create a store supplier
instead.
*
* @param name                name of the store (cannot be {@code null})
* @param retentionPeriod      length of time to retain data in the store (cannot be negative)
*                             (note that the retention period must be at least long enough to contain the
*                             windowed data's entire life cycle, from window-start through window-end,
*                             and for the entire grace period)
* @param windowSize           size of the windows (cannot be negative)
* @param retainDuplicates     whether or not to retain duplicates. Turning this on will automatically disable
*                             caching and means that null values will be ignored.
* @param transactional       whether the store should be transactional
* @return an instance of {@link WindowBytesStoreSupplier}
* @throws IllegalArgumentException if {@code retentionPeriod} or {@code windowSize} can't be represented as
{@code long milliseconds}
* @throws IllegalArgumentException if {@code retentionPeriod} is smaller than {@code windowSize}
*/
public static WindowBytesStoreSupplier persistentTimestampedWindowStore(final String name,
                                                                    final Duration retentionPeriod,
                                                                    final Duration windowSize,
                                                                    final boolean retainDuplicates,
                                                                    final boolean transactional) throws
IllegalArgumentException

/**
 * Create a persistent {@link SessionBytesStoreSupplier}.
 *
 * @param name                name of the store (cannot be {@code null})
 * @param retentionPeriod      length of time to retain data in the store (cannot be negative)
 *                             (note that the retention period must be at least as long enough to
 *                             contain the inactivity gap of the session and the entire grace period.)
 * @param transactional       whether the store should be transactional
 * @return an instance of a {@link SessionBytesStoreSupplier}
 */
public static SessionBytesStoreSupplier persistentSessionStore(final String name,
                                                                final Duration retentionPeriod,
                                                                final boolean transactional)

```

Materialized.StoreType

Materialized.StoreType enum has a new value TXN_ROCKS_DB that corresponds to a transactional state store implementation based on RocksDB.

Compatibility, Deprecation, and Migration Plan

Transactional state stores will be disabled by default. Both Streams DSL and Processor API users can enable transactional writes in the built-in RocksDB state store by passing a new boolean flag `transactional=true` to Materialized constructor and Stores factory methods. Custom state stores will have an option to enable transactionality by adjusting their implementation according to the `StateStore#transactional()` contract.

`StateStore#flush()` method is deprecated. New `StateStore#commit(changelogOffset)` method will by default fall back to `StateStore#flush()` for non-transactional state stores.

Proposed changes are source compatible and binary incompatible with previous releases.

Test Plan

1. Add a variation for all existing stateful tests to run with enabled transactional state stores.
2. Add tests to ensure that transactional state stores discard uncommitted changes after crash failure.
3. Add tests to ensure that transactional state stores replay missing changes from the changelog topic on recovery.

Transactions via Secondary State Store for Uncommitted Changes

This proposal comes with a reference implementation used in the `Stores#` factory methods used to create transactional state stores. In this implementation, transactionality is guaranteed by batching uncommitted (dirty) writes in a temporary RocksDB instance. On commit, such state store copies uncommitted writes from the temporary store to the main store, then truncates the temporary stores.

All writes and deletes go to the temporary store. Reads query the temporary store; if the data is missing, query the regular store. Range reads query both stores and return a `KeyValueIterator` that merges the results. On crash failure, `ProcessorStateManager` calls `StateStore#recover(offset)` that truncates the temporary store.

The major advantage of this approach is that the temporary state store can optionally use the available disk space if the writes do not fit into the in-memory buffer.

The disadvantages are:

- It doubles the number of open state stores
- It potentially has higher write and read amplification due to uncontrolled flushes of the temporary state store.
- It requires an additional value copy per write to model deletions.

Rejected Alternatives

RocksDB in-memory Indexed Batches

A considered alternative is to make built-in RocksDB state store transactional by using [WriteBatchWithIndex](#), which is similar to `WriteBatch` already used segment stores, except it also allows reading uncommitted data.

The advantage of this approach is that it uses the RocksDB built-in mechanism to ensure transactionality and offers the smallest possible write amplification overhead.

The disadvantage of this approach is that all uncommitted writes must fit into memory. In practice, RocksDB developers recommend the batches to be no larger than 3-4 megabytes ([link](#)) which might be an issue

RocksDB Optimistic Transactions

Another considered alternative is [OptimisticTransactionDB](#). This alternative suffers from the same issues as in-memory indexed batches, but also has greater overhead. It offers more guarantees than Kafka Streams needs, specifically - ensures that there were no write conflicts between concurrent transactions before committing. There are no concurrent transactions in Kafka Streams, so there is no reason to pay for the associated overhead.

Method to control transaction lifecycle in StateStore

A considered alternative is to introduce methods like `StateStore#beginTransaction` and `StateStore#commitTxn` to manage transactions lifecycle. I don't think they are necessary due to stream workloads specifics - there is always a single transaction for a given task and that transaction commits only after the commit to the changelog. Moreover, explicit method calls to begin and commit a transaction introduce possible invalid states, like skipping `beginTransaction` before committing, beginning a transaction multiple times, committing after flushing, etc.