KIP-892: Transactional Semantics for StateStores

Status

Current state: Adopted

Discussion thread: Thread

JIRA: ⚠ Unable to render Jira issues macro, execution error.

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

Motivation

As described in KIP-844, under EOS, crash failures cause all Task state to be wiped out on restart. This is because, currently, data is written to the StateStore before the commit to its changelog has completed, so it's possible that records are written to disk that were not committed to the store changelog.

This ensures consistency of local stores with their changelog topics, but can cause long delays in processing while it rebuilds the local state from the changelog. These delays are proprotional to the number of records in the changelog topic, which for highly active tables, or those with a very high cardinality, can be very large. Real-world use-cases have been observed where these delays can span *multiple days*, where both processing, and interactive queries, are paused.

In KIP-844, it was proposed to create an alternative type of StateStore, which would enable users to opt-in to "transactional" behaviour, that ensured data was only persisted once the changelog commit has succeeded. However, the design and approach outlined in KIP-844 unfortunately did not perform well when tested (with a write throughput that was approximately only 4% of the regular RocksDB StateStore!).

This KIP explores an alternative design that should have little/no performance impact, potentially performing better than the status quo, and can thus be enabled for all stores. This should bound state restore under EOS to less than 1 second, irrespective of the size of the changelogs.

Public Interfaces

New configuration

Name	Default	Description

default.state.isolation.level	READ_UNCOMMITTED	The default isolation level for Interactive Queries against StateStores. Supported values are READ_UNCOMMITTED and READ_COMMITTED.
statestore.uncommitted.max.bytes	67108864 (64 MB)	Maximum number of memory bytes to be used to buffer uncommitted state-store records. If this limit is exceeded, a task commit will be requested. No limit: -1. Note: if this is too high or unbounded, it's possible for RocksDB to trigger out-of-memory errors.

Changed Interfaces

- org.apache.kafka.streams.processor.StateStore
- org.apache.kafka.streams.processor.StateStoreContext

Changes:

org.apache.kafka.streams.processor.StateStore

```
    /**
     * Flush any cached data
     *
     * @deprecated Use {@link org.apache.kafka.streams.processor.api.ProcessingContext#commit()
ProcessorContext#commit()}
     *             instead.
     */
    @Deprecated
    default void flush() {
        // no-op
    }

    /**
     * Commit all written records to this StateStore.
     * <p>
     * This method MUST NOT be called by users from {@link org.apache.kafka.streams.processor.api.Processor
processors},
     * as doing so may violate the consistency guarantees provided by this store, and expected by Kafka Streams.
     * Instead, users should call {@link org.apache.kafka.streams.processor.api.ProcessingContext#commit()
     * ProcessorContext#commit()} to request a Task commit.
     * <p>
     * When called, every write written since the last call to {@link #commit(Map)}, or since this store was
{@link
     * #init(StateStoreContext, StateStore) opened} will be made available to readers using the {@link
     * org.apache.kafka.common.IsolationLevel#READ_COMMITTED READ_COMMITTED} {@link
     * org.apache.kafka.common.IsolationLevel IsolationLevel}.
     * <p>
     * If {@link #persistent()} returns {@code true}, after this method returns, all records written since the
last call
     * to {@link #commit(Map)} are guaranteed to be persisted to disk, and available to read, even if this
{@link
     * StateStore} is {@link #close() closed} and subsequently {@link #init(StateStoreContext, StateStore) re-
opened}.
     * <p>
     * If {@link #managesOffsets()} <em>also</em> returns {@code true}, the given {@code changelogOffsets} will
be
     * guaranteed to be persisted to disk along with the written records.
     * <p>
     * {@code changelogOffsets} will usually contain a single partition, in the case of a regular StateStore.
However,
     * they may contain multiple partitions in the case of a Global StateStore with multiple partitions. All
provided
     * partitions <em>MUST</em> be persisted to disk.
     * <p>
     * Implementations <em>SHOULD</em> ensure that {@code changelogOffsets} are committed to disk atomically
with the
     * records they represent.
     *
     * @param changelogOffsets The changelog offset(s) corresponding to the most recently written records.
     */
    default void commit(final Map<TopicPartition, Long> changelogOffsets) {
        flush();
```

```
    }

    /**
     * Returns the most recently {@link #commit(Map) committed} offset for the given {@link TopicPartition}.
     * <p>
     * If {@link #managesOffsets()} and {@link #persistent()} both return {@code true}, this method will return
the
     * offset that corresponds to the changelog record most recently written to this store, for the given {@code
     * partition}.
     * <p>
     * This method provides readers using the {@link org.apache.kafka.common.IsolationLevel#READ_COMMITTED}
{@link
     * org.apache.kafka.common.IsolationLevel} a means to determine the point in the changelog that this
StateStore
     * currently represents.
     *
     * @param partition The partition to get the committed offset for.
     * @return The last {@link #commit(Map) committed} offset for the {@code partition}; or {@code null} if no
offset
     *         has been committed for the partition, or if either {@link #persistent()} or {@link
#managesOffsets()}
     *         return {@code false}.
     */
    default Long committedOffset(final TopicPartition partition) {
        return null;
    }

    /**
     * Determines if this StateStore manages its own offsets.
     * <p>
     * If this method returns {@code true}, then offsets provided to {@link #commit(Map)} will be retrievable
using
     * {@link #committedOffset(TopicPartition)}, even if the store is {@link #close() closed} and later re-
opened.
     * <p>
     * If this method returns {@code false}, offsets provided to {@link #commit(Map)} will be ignored, and
{@link
     * #committedOffset(TopicPartition)} will be expected to always return {@code null}.
     * <p>
     * This method is provided to enable custom StateStores to opt-in to managing their own offsets. This is
highly
     * recommended, if possible, to ensure that custom StateStores provide the consistency guarantees that
Kafka Streams
     * expects when operating under the {@code exactly-once} {@code processing.mode}.
     *
     * @return Whether this StateStore manages its own offsets.
     */
    default boolean managesOffsets() {
        return false;
    }

   /**
     * Return an approximate count of memory used by records not yet committed to this StateStore.
     * <p>
     * This method will return an approximation of the memory that would be freed by the next call to {@link
     * #commit(Map)}.
     * <p>
     * If no records have been written to this store since {@link #init(StateStoreContext, StateStore)
opening}, or
     * since the last {@link #commit(Map)}; or if this store does not support atomic transactions, it will
return {@code
     * 0}, as no records are currently being buffered.
     *
     * @return The approximate size of all records awaiting {@link #commit(Map)}; or {@code 0} if this store
does not
     *         support transactions, or has not been written to since {@link #init(StateStoreContext,
StateStore)} or
     *         last {@link #commit(Map)}.
     */
    @Evolving
    default long approximateNumUncommittedBytes() {
```

```
        return 0;
    }
```

Metrics

New

- `stream-state-metrics`
    - `commit-rate` - the number of calls to `StateStore#commit(Map)`
    - `commit-latency-avg` - the average time taken to call `StateStore#commit(Map)`
    - `commit-latency-max` - the maximum time taken to call `StateStore#commit(Map)`

Deprecated

- `stream-state-metrics`
    - `flush-rate`
    - `flush-latency-avg`
    - `flush-latency-max`

These changes are necessary to ensure these metrics are not confused with orthogonal operations, like RocksDB memtable flushes or cache flushes. They will be measuring the invocation of `StateStore#commit`, which replaces `StateStore#flush`.

While the `flush` metrics are only deprecated, they will no longer record any data under normal use, as Kafka Streams will no longer call `StateStore#flush()`.

Proposed Changes

To ensure that data is not written to a state store until it has been committed to the changelog, we need to isolate writes from the underlying database until changelog commit. To achieve this, we introduce the concept of transaction Isolation Levels, that dictate the visibility of records, written by processing threads, to Interactive Query threads.

We enable configuration of the level of isolation provided by StateStores via a `default.state.isolation.level`, which can be configured to either:

default.state.isolation.level	**Description**
`READ_UNCOMMITTED`	Records written by the StreamThread are visible to all Interactive Query threads immediately. This level provides no atomicity, consistency, isolation or durability guarantees. Under this Isolation Level, Streams behaves as it currently does, wiping state stores on-error when the `processing.mode` is one of `exactly-once`, `exactly-once-v2` or `exactly-once-beta`.
`READ_COMMITTED`	Records written by the StreamThread are only visible to Interactive Query threads once they have been committed. Under this Isolation Level, Streams will isolate writes from state stores until commit. This guarantees consistency of the on-disk data with the store changelog, so Streams will not need to wipe stores on-error.

In Kafka Streams, all `StateStore` s are written to by a single `StreamThread` (this is the Single Writer principle). However, multiple other threads may concurrently *read* from `StateStore` s, principally to service Interactive Queries. In practice, this means that under `READ_COMMITTED`, writes by the `StreamThread` that owns the `StateStore` will only become visible to Interactive Query threads once `commit()` has been called.

The default value for default.state.isolation.level will be `READ_UNCOMMITTED`, to mirror the behaviour we have today; but this will be automatically set to `READ_COMMITTED` if the processing.mode has been set to an EOS mode, and the user has not explicitly set `deafult.state.isolation.level` to `READ_UNCOMMITTED`. This will provide EOS users with the most useful behaviour out-of-the-box, but ensures that they may choose to sacrifice the benefits of transactionality to ensure that Interactive Queries can read records before they are committed, which is required by a minority of use-cases.

In-memory Transaction Buffers

Many StateStore implementations, including RocksDB, will buffer records written to a transaction entirely in-memory, which could cause issues, either with JVM heap or native memory. To mitigate this, we will automatically force a `Task` commit if the total memory used for buffering uncommitted records returned by `StateStore#approximateNumUncommittedBytes()` exceeds the threshold configured by `statestore.uncommitted.max.bytes`. This will roughly bound the memory required for buffering uncommitted records, irrespective of the `commit.interval.ms`, and will effectively bound the number of records that will need to be restored in the event of a failure. Each `StreamThread` will be given `1/num.stream.threads` of the configured limits, dividing it fairly between them.

It's possible that some Topologies can generate many more new `StateStore` entries than the records they process, in which case, it would be possible for such a Topology to cross the configured record/memory thresholds mid-processing, potentially causing an OOM error if these thresholds are exceeded by a lot. To mitigate this, the `StreamThread` will measure the increase in records/bytes written on each iteration, and pre-emptively commit if the *next* iteration is likely to cross the threshold.

Note that this new method provides default implementations that ensure existing custom stores and non-transactional stores (e.g. InMemoryKeyValueStore) do not force any early commits.

Interactive Queries

Interactive queries currently see every record, as soon as they are written to a `StateStore`. This can cause some consistency issues, as interactive queries can read records before they're committed to the Kafka changelog, which may be rolled-back. To address this, we have introduced configurable isolation levels, configured globally via `default.state.isolation.level` (see above).

When operating under the `READ_COMMITTED` isolation level, the maximum time for records to become visible to interactive queries will be `commit.interval.ms`. Under EOS, this is by default a low value (`100 ms`), but under `at-least-once`, the default is 30 seconds. Users may need to adjust their `commit.interval.ms` to meet the visibility latency goals for their use-case.

When operating under the `READ_UNCOMMITTED` isolation level, (i.e. ALOS), all records will be immediately visible to interactive queries, so the high default `commit.interval.ms` of `30s` will have no impact on interactive query latency.

Error Handling

Kafka Streams currently generates a TaskCorruptedException when a `Task` needs to have its state wiped (under EOS) and be re-initialized. There are currently several different situations that generate this exception:

1. No offsets for the store can be found when opening it under EOS.
2. `OutOfRangeException` during restoration, usually caused by the changelog being wiped on application reset.
3. `TimeoutException` under EOS, when writing to or committing a Kafka transaction.

The first two of these are extremely rare, and make sense to keep. However, timeouts are much more frequent. They currently require the store to be wiped under EOS because when a timeout occurs, the data in the local `StateStore` will have been written, but the data in the Kafka changelog will have failed to be written, causing a mismatch in consistency.

With Transactional StateStores, we can guarantee that the local state is consistent with the changelog, therefore, it will no longer be necessary to reset the local state on a `TimeoutException` when operating under the `READ_COMMITTED` isolation level.

Atomic Checkpointing

Kafka Streams currently stores the changelog offsets for a StateStore in a per-Task on-disk file, `.checkpoint`, which under EOS, is written only when Streams shuts down successfully. There are two major problems with this approach:

- To ensure that the data on-disk matches the checkpoint offsets in the `.checkpoint` file, we must flush the StateStores whenever we update the offsets in `.checkpoint`. This is a performance regression, as it causes a significant increase in the frequency of RocksDB memtable flushes, which increases load on RocksDB's compaction threads.
- There's a race condition, where it's possible the application exits after data has been committed to RocksDB, but before the checkpoint file has been updated, causing a consistency violation.

To resolve this, we move the responsibility for offset management to the `StateStore` itself. The new `commit` method takes a map of all the changelog offsets that correspond to the state of the transaction buffer being committed.

`RocksDBStore` will store these offsets in a separate Column Family, and will be configured to atomically flush all its Column Families. This guarantees that the changelog offsets will always be flushed to disk together with the data they represent, irrespective of how that flush is triggered. This allows us to remove the explicit memtable `flush()`, enabling RocksDB to dictate when memtables are flushed to disk.

The existing `.checkpoint` files will be retained for any `StateStore` that does not set `managesOffsets()` to `true` , and to ensure managed offsets are available when the store is closed. Existing offsets will be automatically migrated into `StateStores` that manage their own offsets, iff there is no offset returned by `StateStore#committedOffset`.

Required interface changes:

- Add methods `void commit(Map<TopicPartition, Long> changelogOffsets)`, `boolean managesOffsets()` and `Long committedOffset(TopicPartition)` to `StateStore` .
- Deprecate method `flush()` on `StateStore.`

Offsets for Consumer Rebalances

Kafka Streams directly reads from the Task `.checkpoint` file during Consumer rebalance, in order to optimize assignments of stateful Tasks by assigning them to the instance with the most up-to-date copy of the data, which minimises restoration. To allow this to continue functioning, Kafka Streams will continue to write the changelog offsets to the `.checkpoint` file, even for stores that manage their own offsets.

Offsets will be written to `.checkpoint` at the following times:

1. During StateStore initialization, in order to synchronize the offsets in `.checkpoint` with the offsets returned by `StateStore#committedOffset(TopicPartition)`, which are the source of truth for stores that manage their own offsets.
2. When the StateStore is closed, in order to ensure that the offsets used for Task assignment reflect the state persisted to disk.
3. At the end of every Task commit, if-and-only-if at least one StateStore in the Task is persistent and does *not* manage its own offsets. This ensures that stores that don't manage their offsets continue to have their offsets persisted to disk whenever the StateStore data itself is committed.
    - Avoiding writing `.checkpoint` when every persistent store manages its own offsets ensures we don't pay a significant performance penalty when the commit interval is short, as it is by default under EOS.
    - Since all persistent StateStores provided by Kafka Streams will manage their own offsets, the common case is that the `.checkpoint` file will not be updated on `commit(Map)`

Tasks that are already assigned to an instance, already use the in-memory offsets when calculating partition assignments, so no change is necessary here.

Interactive Query .position Offsets

Input partition "`Position`" offsets, introduced by KIP-796: Interactive Query v2, are currently stored in a `.position` file by the `RocksDBStore` implementation. To ensure consistency with the committed data and changelog offsets, these position offsets will be stored in RocksDB, in the same column family as the changelog offsets, instead of the `.position` file. When a `StateStore` that manages its own offsets is first initialized, if a `.position` file exists in the store directory, its offsets will be automatically migrated into the store, and the file will be deleted.

When writing data to a `RocksDBStore` (via `put`, `delete`, etc.), the input partition offsets will be read from the changelog record metadata (as before), and these offsets will be added to the current transactions `WriteBatch`. When the `StateStore` is committed, the position offsets in the current `WriteBatch` will be written to RocksDB, alongside the records they correspond to. Alongside this, `RocksDBStore` will maintain two `Position` maps in-memory, one containing the offsets pending in the current transaction's `WriteBatch`, and the other containing committed offsets. On `commit(Map)`, the uncommitted `Position` map will be merged into the committed `Position` map. In this sense, the two `Position` maps will diverge during writes, and re-converge on-commit.

When an interactive query is made under the `READ_COMMITTED` isolation level the `PositionBound` will constrain the committed Position map, whereas under `READ_UNCOMMITTED`, the `PositionBound` will constrain the uncommitted Position map.

RocksDB Transactions

When the isolation level is `READ_COMMITTED`, we will use RocksDB's `WriteBatchWithIndex` as a means to accomplishing atomic writes when not using the RocksDB WAL. When reading records from the `StreamThread`, we will use the `WriteBatchWithIndex#getFromBatchAndDB` and `WriteBatchWithIndex#newIteratorWithBase` utilities in order to ensure that uncommitted writes are available to query. When reading records from Interactive Queries, we will use the regular `RocksDB#get` and `RocksDB#newIterator` methods, to ensure we see only records that have been committed (see above). The performance of this is expected to actually be *better* than the existing, non-batched write path. The main performance concern is that the WriteBatch must reside completely in-memory until it is committed, which is addressed by `statestore.uncommitted.max.bytes`, see above.

Compatibility, Deprecation, and Migration Plan

The above changes will retain compatibility for all existing `StateStores`, including user-defined custom implementations. Any `StateStore` that extends `RocksDBStore` will automatically inherit its behaviour, although its internals will change, potentially requiring users that depend on internal behaviour to update their code.

All new methods on existing classes will have defaults set to ensure compatibility.

Kafka Streams will automatically migrate offsets found in an existing `.checkpoint` file, and/or an existing `.position` file, to store those offsets directly in the `StateStore`, if `managesOffsets` returns `true`. Users of the in-built store types will not need to make any changes. See Upgrading.

Users may notice a change in the performance/behaviour of Kafka Streams. Most notably, under EOS Kafka Streams will now regularly "commit" StateStores, where it would have only done so when the store was closing in the past. The overall performance of this should be at least as good as before, but the profile will be different, with write latency being substantially faster, and commit latency being a bit higher.

Upgrading

When upgrading to a version of Kafka Streams that includes the changes outlined in this KIP, users will not be required to take any action. Kafka Streams will automatically upgrade any RocksDB stores to manage offsets directly in the RocksDB database, by importing the offsets from any existing `.checkpoint` and/or `.position` files.

Users that currently use `processing.mode: exactly-once(-v2|-beta)` and who wish to continue to read uncommitted records from their Interactive Queries will need to explicitly set `default.state.isolation.level: READ_UNCOMMITTED`.

Downgrading

When downgrading from a version of Kafka Streams that includes the changes outlined in this KIP to a version that does not contain these changes, users will not be required to take any action. The older Kafka Streams version will be unable to open any RocksDB stores that were upgraded to store offsets (see Upgrading), which will cause Kafka Streams to wipe the state for those Tasks and restore the state, using an older RocksDB store format, from the changelogs.

Since downgrading is a low frequency event, and since restoring state from scratch is already an existing failure mode for older versions of Kafka Streams, we deem this an acceptable automatic downgrade strategy.

Test Plan

Testing will be accomplished by both the existing tests and by writing some new unit tests that verify atomicity, durability and consistency guarantees that this KIP provides.

Rejected Alternatives

Dual-Store Approach (KIP-844)

The design outlined in KIP-844, sadly, does not perform well (as described above), and requires users to opt-in to transactionality, instead of being a guarantee provided out-of-the-box.

Replacing RocksDB memtables with ThreadCache

It was pointed out on the mailing list that Kafka Streams fronts all RocksDB StateStores with a configurable record cache, and that this cache duplicates the function requests for recently written records provided by RocksDB memtables. A suggestion was made to utilize this record cache (the `ThreadCache` class) as a replacement for memtables, by directly flushing them to SSTables using the RocksDB `SstFileWriter`.

This is out of scope of this KIP, as its goal would be reducing the duplication (and hence, memory usage) of RocksDB StateStores; whereas this KIP is tasked with improving the consistency of StateStores to reduce the frequency and impact of state restoration, improving their scalability.

It has been recommended to instead pursue this idea in a subsequent KIP, as the interface changes outlined in this KIP should be compatible with this idea.

Transactional support under READ_UNCOMMITTED

When query isolation level is READ_UNCOMMITTED, Interactive Query threads need to read records from the ongoing transaction buffer. Unfortunately, the RocksDB WriteBatch is not thread-safe, causing Iterators created by Interactive Query threads to produce invalid results/throw unexpected errors as the WriteBatch is modified/closed during iteration.

Ideally, we would build an implementation of a transaction buffer that is thread-safe, enabling Interactive Query threads to query it safely. One approach would be to "chain together" WriteBatches, creating a new WriteBatch every time a new Iterator is created by an Interactive Query thread and "freezing" the previous WriteBatch.

It was decided to defer tackling this problem to a later KIP, in order to realise the benefits of transactional state stores to users as quickly as possible.

Query-time Isolation Levels

It was requested that users be able to select the isolation level of queries on a per-query basis. This would require some additional API changes (to the Interactive Query APIs). Such an API would require that state stores are always transactional, and that the transaction buffers can be read from by READ_UNCOMMITTED queries. Due to the problems outlined in the previous section, it was decided to also defer this to a subsequent KIP.

The new configuration option `default.state.isolation.level` was deliberately named to enable query-time isolation levels in the future, whereby any query that didn't explicitly choose an isolation level would use the configured default. Until then, this configuration option will globally control the isolation level of all queries, with no way to override it per-query.