KIP-258: Allow to Store Record Timestamps in RocksDB

Status

Current state: *"Accepted"* *[VOTE] KIP-258: Allow to Store Record Timestamps in RocksDB*

Discussion thread: [DISCUSS] KIP-258: Allow to Store Record Timestamps in RocksDB

JIRA:

- Unable to render Jira issues macro, execution error. (2.3 release)
- Unable to render Jira issues macro, execution error. (2.3 release)
- Unable to render Jira issues macro, execution error.

Released: 2.3 (partially implemented)

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

Motivation

In order to improve the provided stream processing semantics of – and to add new feature to – Kafka Streams, we want to be able to store record timestamps in KTables. This allows us to address multiple issues like

- store timestamps in KTables (KAFKA-6521)
- handling out-of-order data for source KTable (related to KAFKA-5533)
- improve timestamp propagation for DSL operator (KAFKA-6455)
- return the timestamp of the latest update in Interactive Queries (KAFKA-4304)
- add TTL for KTables (KAFKA-4212 and KAFKA-4273)

The challenging part of this KIP is to define a smooth upgrade path with the upgraded RocksDB format. Note that only DSL users will be affected, because we want to switch the default stores.

Public Interfaces

We add three new store types:

- `TimstampedKeyValueStore` that extends the exiting `KeyValueStore`
- `TimstampedWindowStore` that extends the exiting `WindowStore`
- `TimstampedSessionStore` that extends the exiting `SessionStore`

Those new stores will be used by DSL operators. For PAPI users, nothing changes for existing applications. Of course, the new stores can we used, too.

For a seamless single rolling bounce upgrade, we introduce a `TimestampedBytesStore` interface. This interface is be used by byte-stores, to indicate that they expect the new value+timestamp format. Additionally it provides a static helper method to convert `byte[]` arrays in old "plain value" format to new "value+timestamp format".

Proposed Changes

To make use of the new timestamp that can be stored, we need to add new class to the existing store interfaces that allow to read/write the timestamp and the value at once.

```
package org.apache.kafka.streams.state;

// new classed and interfaces

public class ValueAndTimestamp<V> {
    private ValueAndTimestamp(final V value, final long timestamp); // use `make()` instead

    public V value();
    public long timestamp();
    public static <V> ValueAndTimestamp<V> make(final V value, final long timestamp); // returns `null` if
`value==null`
    public static <V> V getValueOrNull(final ValueAndTimestamp<V> valueAndTimestamp); // returns `null` if
`valueAndTimestamp==null`
}

public interface TimestampedKeyValueStore<K, V> extends KeyValueStore<K, ValueAndTimestamp<V>> {}

public interface TimestampedWindowStore<K, V> extends WindowStore<K, ValueAndTimestamp<V>> {}

public interface TimestampedSessionStore<K, V> extends SessionStore<K, ValueAndTimestamp<V>> {}

public interface TimestampedBytesStore {
    static byte[] convertToTimestampedFormat(final byte[] plainValue);
}

// extend existing classes (omitting existing method)

public final class Stores {
    public static KeyValueBytesStoreSupplier persistentTimestampedKeyValueStore(final String name);

    public static WindowBytesStoreSupplier persistentTimestampedWindowStore(final String name,
                                                                             final Duration retentionPeriod,
                                                                             final Duration windowSize,
                                                                             final boolean retainDuplicates);

    public static SessionBytesStoreSupplier persistentTimestampedSessionStore(final String name,
                                                                               final Duration retentionPeriod);

    public static <K, V> StoreBuilder<TimestampedKeyValueStore<K, V>> timestampedKeyValueStoreBuilder(final
KeyValueBytesStoreSupplier supplier,
                                                                                                      final
Serde<K> keySerde,
                                                                                                      final
Serde<V> valueSerde);


    public static <K, V> StoreBuilder<TimestampedWindowStore<K, V>> timestampedWindowStoreBuilder(final
WindowBytesStoreSupplier supplier,
                                                                                                  final
Serde<K> keySerde,
                                                                                                  final
Serde<V> valueSerde;

    public static <K, V> StoreBuilder<TimestampedSessionStore<K, V>> timestampedSessionStoreBuilder(final
SessionBytesStoreSupplier supplier,
                                                                                                    final
Serde<K> keySerde,
                                                                                                    final
Serde<V> valueSerde);
}

public final class QueryableStoreTypes {
    public static <K, V> QueryableStoreType<ReadOnlyKeyValueStore<K, ValueAndTimestamp<V>>>
timestampedKeyValueStore();
```

```
    public static <K, V> QueryableStoreType<ReadOnlyWindowStore<K, ValueAndTimestamp<V>>>
timestampedWindowStore();

    public static <K, V> QueryableStoreType<ReadOnlySessionStore<K, ValueAndTimestamp<V>>>
timestampedSessionStore();
}


// test-utils package


public class TopologyTestDriver {
    public <K, V> KeyValueStore<K, ValueAndTimestamp<V>> getTimestampedKeyValueStore(final String name);

    public <K, V> WindowStore<K, ValueAndTimestamp<V>> getTimestampedWindowStore(final String name);

    public <K, V> SessionStore<K, ValueAndTimestamp<V>> getTimestampedSessionStore(final String name);
}
```

New stores (RocksDB, InMemory) will be added that implement the corresponding new interfaces. Note, we keep the existing stores as-is and only add new stores that can be used by PAPI users too; by default, PAPI users would need to rewrite their application to switch from existing store usage to new stores if desired.

All users upgrade with a single rolling bounce per instance.

For hide the format change for DSL user, the new stores will handle the format change internally. We have to distinguish in-memory and persistent stores:

In-Memory stores:

- on restart, data will be read from the changelog topic
- the changelog format does not change
- the record timestamp can be extracted and put into the value on read (`RecordConverter` will take care of this part)

Persistent stores:

- For locally stored data, reads will be served with surrogate timestamp -1 (semantics is "unknown").
- On put, data will be stored using the new format.
- *Key-Value store*: new `TimestampedRocksDBStore` is used. To isolate old and new format, we use two column families. We perform dual put /get operation in new and old column family to lazily upgrade all data from old to new format.
- *Window/Session store*: existing segments will use old format and new segments will be created with the new format.

The described upgrade path works for library provided store implementations out-of-the-box. All new stores implement `TimestampedBytesStore` interfac e, that indicates that the store can handle old and new format and does a seamless upgrade internally.

There is one special case for which users uses a custom `XxxByteStoreSupplier`. For this case the returned store wont' implement `TimestampedByte sStore` and won't understand the new format. Hence, we use a proxy store to remove timestamps on write and add surrogate timestamp -1 (semantics is "unknown") on read (note, that for non-persistent custom stores, we don't need to use the proxy). Using the proxy is required to guarantee a backward compatible upgrade path. Users implementing a custom `XxxByteStoreSupplier` can opt-in and extend their stores to implement `TimestampedBytesS tore` interface, too. For this case, we don't wrap the store with a proxy and it's the users responsibility to implement the custom store correctly to migrate from old to new format. Users implementing a custom stores that implement `TimestampedBytesStore` and that want to upgrade existing data in old format to new format can use `TimestampedBytesStores#convertToTimestampedFormat()` method.

Compatibility, Deprecation, and Migration Plan

Simple rolling bounce upgrade is sufficient. For PAPI uses, nothing changes at all. For DSL users, the internal `RocksDBTimestampedStore` (as one example) will upgrade the store data lazily in the background. Only if users provide a custom `XxxByteStoreSupplier` no upgrade happens (but users can opt-in implementing `TimestampedBytesStore` interface) but the old format is kept. We use a proxy store that removes/adds timestamp on read /write.

To keep interactive queries feature compatible, the new `TimestampedXxxStores` can be queried with and without timestamp. This is important for DSL users, that might query a store as `KeyValueStore` while the DSL switches the store internally to a `TimestampedKeyValueStore`. Thus, we allow to query a `TimestampedKeyValueStore` with queryable story type `"key-value"` and remove the timestamp under the hood on read. Compatibility for window/session stores are handled similarly.

The `TimestampedBytesStore` interface will be implemented by Kafka Streams for the newly added stores. Only users who implement custom key-value /window/session stores are not affected, as their stores will be wrapped by the proxy store. User can "opt-in" of course, and change their custom key-value /window/session store implementation to support the new `TimestampedXxxStore` interfaces by implementing `TimestampedBytesStore` in their own store classes. For other user implemented stores that add a new `StateStore` type, nothing changes for existing code.

Test Plan

This feature can be mainly tested via unit and integration tests:

- unit/integration tests for newly added stores
- Upgrades can be simulated in integration tests by combining PAPI and DSL in one test run

Additionally, existing system tests are extended to perform rolling bounce upgrades.

Rejected Alternatives

- background upgrade (ie, use existing store and migrate to new format in the background; switch over to new format later)
  - requires complex upgrade path with in-place/roll-over configuration and two rolling bounces
- change the RocksDB on-disk format and encode the used serialization version per record (this would simplify future upgrades). However there are main disadvantages:
  - storage amplification for local stores
  - record version could get stored in record headers in changelog topics -> changelog topic might never overwrite record with older format
  - code needs to check all versions all the time for future release: increases code complexity and runtime overhead
  - it's hard to change the key format
    - for value format, the version number can be a magic prefix byte
    - for key lookup, we would need to know the magic byte in advance for efficient point queries into RocksDB; if multiple versions exist in parallel, this is difficult (either do multiple queries with different versions bytes until entry is found or all versions are tried implying does not exist – or use range queries but those are very expensive)
- encode the storage format in the directory name not at "store version number" but at "AK release number"
  - might be confusion to user if store format does not change ("I am running Kafka 1.4, but the store indicates it's running on 1.2").
- use a simpler offline upgrade path without any configs or complex rolling bounce scenarios
  - requires application down-time for upgrading to new format
- use consumer's built-in protocol upgrade mechanism (ie, register multiple "assignment strategies")
  - has the disadvantage that we need to implement two `StreamsPartitionAssingor` classes
  - increased network traffic during rebalance
  - encoding "supported version" in metadata subsumes this approach for future releases anyway
  - if we want to "disable" the old protocol, a second rebalance is required, too
  - cannot avoid a second rebalance that this required for state store upgrade
- only support in-place upgrade path instead of two to simplify the process for users (don't need to pick)
  - might be prohibitive if not enough disk space is available
- allow DSL users to stay with old format: upgrade would be simpler as it's only one rolling bounce
  - unclear default behavior: should we stay on 2.1 format by default or should we use 2.2 format by default?
    - if 2.1 is default, upgrade is simple, but if one write a new application, users must turn on 2.2 format explicitly
    - if 2.2 is default, simple upgrade requires a config that tells Streams to stay with 2.1 format
    - conclusion: upgrading and not upgrading is not straight forward either way, thus, just force upgrade
  - if no upgrade happens, new features (as listed above) would be useless
- Only prepare stores for active task (but not standby tasks)
  - this would reduce the disk footprint during upgrade
  - disadvantage: when switch to new version happens, there are not hot standby available for some time
  - we could make it configurable, however, we try to keep the number of configs small; also, it's already complex enough and adding more options would make it worse
  - it's not an issue for roll-over upgrade and not everybody configures Standbys in the first place
    - people with standbys are willing to provide more disk, so it seem a fair assumption that they are fine with roll-over upgrade, too