KIP-889: Versioned State Stores

Status

Current state: *Accepted*

Discussion thread: here

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

The state stores used by Kafka Streams today maintain only the latest value associated with each key. This prevents Kafka Streams from providing proper temporal join semantics for stream-tables joins.

Consider the following example of A join B, where A is a stream and B is a table, and records (for a particular key) arrive in the following order:

```
                    B: (time=0, v=b0)
A: (time=1, v=a1)                       --> output 1: (a1, b0)
                    B: (time=3, v=b3)
A: (time=4, v=a4)                       --> output 2: (a4, b3)
A: (time=2, v=a2)                       --> output 3: (a2, b3) should be (a2, b0) instead
```

Note that the last record (a2) is out-of-order by timestamp (relative to the other messages in stream A). The first two join result records are as expected: the a1 record is joined against b0, and the a4 record is joined against b3. However, when the out-of-order a2 record arrives, it should be joined against the b0 record since that's the latest record on the B-side as of its record timestamp `time=2` but this is not what happens. The state store for table B has already been updated to contain b3 and the b0 record was replaced in the process. As a result, the only record available in the store for the a2 record to join against is the b3 record, which is why Kafka Streams outputs the join result (a2, b3) today.

For an example stream-table join application where proper temporal join semantics is critical, imagine that B is table of real-time currency conversion rates and A is a stream of transactions. It's important that each transaction in A is joined with the conversion rate *at the time that the transaction was issued,* not the latest conversion rate seen so far.

A similar temporal semantic gap exists with table-table foreign key joins today as well. The subscription store may not reflect an accurate set of foreign key records at the time of the join record's timestamp, resulting in incorrect join results.

To address this class of gaps in Kafka Streams stateful processing semantics, this KIP proposes to introduce versioned state stores. A versioned state store will track multiple record versions for the same key, rather than the single latest record per key as is the case for existing stores today.

Public Interfaces

Introducing versioned stores covers a large surface area: the store interfaces, store implementations, updating DSL processors to use them, interactive queries, metrics/monitoring, etc. The scope of this first KIP is limited to:

- defining the basic interface for versioned stores
- introducing a RocksDB-based implementation for the interface
- additional methods necessary for allowing users to pass versioned stores to the DSL (via StreamsBuilder and Materialized) and PAPI.

Everything else will be deferred to follow-up KIPs.

The interface changes proposed in this KIP are:

- a new interface for versioned stores, and a helper class: `VersionedKeyValueStore<K, V> extends StateStore` , with helper `VersionedRecord`
- a new interface for versioned store suppliers, and a helper interface: `VersionedBytesStoreSupplier extends KeyValueBytesStoreSupplier` , with helper `VersionedBytesStore`
- three new methods in Stores.java:
  - two for creating a persistent, versioned store supplier: `Stores#persistentVersionedKeyValueStore(...)` plus an overload
  - another for creating a StoreBuilder from a versioned supplier: `Stores#versionedKeyValueStoreBuilder(...)`
- a new method in TopologyTestDriver.java for getting a versioned store from a topology
- a new static method in ValueAndTimestamp.java for creating ValueAndTimestamp instances where the value may be null: `ValueAndTimestamp#makeAllowNullable(...)`

Proposed Changes

Versioned Store Interface

The new interface for versioned stores is as follows:

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

/**
 * A key-value store that stores multiple record versions per key, and supports timestamp-based
 * retrieval operations to return the latest record (per key) as of a specified timestamp.
 * Only one record is stored per key and timestamp, i.e., a second call to
 * {@link #put(Object, Object, long)} with the same key and timestamp will replace the first.
 * <p>
 * Each store instance has an associated, fixed-duration "history retention" which specifies
 * how long old record versions should be kept for. In particular, a versioned store guarantees
 * to return accurate results for calls to {@link #get(Object, long)} where the provided timestamp
 * bound is within history retention of the current observed stream time. (Queries with timestamp
 * bound older than the specified history retention are considered invalid.)
 * <p>
 * The store's "history retention" also doubles as its "grace period," which determines how far
 * back in time writes to the store will be accepted. A versioned store will not accept writes
 * (inserts, updates, or deletions) if the timestamp associated with the write is older than the
 * current observed stream time by more than the grace period.
 *
 * @param <K> The key type
 * @param <V> The value type
 */
public interface VersionedKeyValueStore<K, V> extends StateStore {

    /**
     * Add a new record version associated with this key.
     * <p>
     * If the timestamp associated with the new record version is older than the store's
     * grace period (i.e., history retention) relative to the current observed stream time,
     * then the record will not be added.
     *
     * @param key       The key
     * @param value     The value, it can be {@code null};
     *                  if the serialized bytes are also {@code null} it is interpreted as a delete
     * @param timestamp The timestamp for this record version
     * @throws NullPointerException If {@code null} is used for key.
     */
    void put(K key, V value, long timestamp);
```

```
    /**
     * Delete the value associated with this key from the store, at the specified timestamp
     * (if there is such a value), and return the deleted value.
     * <p>
     * If the timestamp associated with this deletion is older than the store's grace period
     * (i.e., history retention) relative to the current observed stream time, then the deletion
     * will not be performed and {@code null} will be returned.
     * <p>
     * This operation is semantically equivalent to {@link #get(Object, long) #get(key, timestamp)}
     * followed by {@link #put(Object, Object, long) #put(key, null, timestamp)}, with
     * a caveat that the return value is always {@code null} if the deletion timestamp
     * is older than the store's grace period (i.e., history retention), regardless of
     * what {@link #get(Object, long) #get(key, timestamp)} would return.
     *
     * @param key       The key
     * @param timestamp The timestamp for this delete
     * @return The value and timestamp of the record associated with this key as of
     *         the deletion timestamp (inclusive), or {@code null} if no such record exists
     *         (including if the deletion timestamp is older than this store's history
     *         retention time, i.e., the store no longer contains data for the provided
     *         timestamp). Note that the record timestamp {@code r.timestamp()} of the
     *         returned {@link VersionedRecord} may be smaller than the provided deletion
     *         timestamp.
     VersionedRecord<V> delete(K key, long timestamp);

    /**
     * Get the latest (by timestamp) record associated with this key.
     *
     * @param key The key to fetch
     * @return The value and timestamp of the latest record associated with this key, or
     *         {@code null} if either (1) the store contains no records for this key or (2) the
     *         latest record for this key is a tombstone.
     * @throws NullPointerException       If null is used for key.
     * @throws InvalidStateStoreException if the store is not initialized
     */
    VersionedRecord<V> get(K key);

    /**
     * Get the latest record associated with this key with timestamp not exceeding the specified
     * timestamp bound.
     *
     * @param key           The key to fetch
     * @param asOfTimestamp The timestamp bound. This bound is inclusive; if a record
     *                      (for the specified key) exists with this timestamp, then
     *                      this is the record that will be returned.
     * @return The value and timestamp of the record associated with this key
     *         as of the provided timestamp, or {@code null} if no such record exists
     *         (including if the provided timestamp bound is older than this store's history
     *         retention time, i.e., the store no longer contains data for the provided
     *         timestamp). Note that the record timestamp {@code r.timestamp()} of the
     *         returned {@link VersionedRecord} may be smaller than the provided timestamp
     *         bound.
     * @throws NullPointerException       If null is used for key.
     * @throws InvalidStateStoreException if the store is not initialized
     */
    VersionedRecord<V> get(K key, long asOfTimestamp);
}
```

Note that this proposal intentionally omits most methods from the existing KeyValueStore interface in order to keep the new interface simple. It could be nice to add additional methods in the future, such as `rangeKey()` methods to enable the foreign-key join subscription store use case, but this is deferred to a future KIP in order to align on these basic interfaces first.

The VersionedRecord return type from the `get()` methods is essentially the same as the existing ValueAndTimestamp class today, but is its own separate class so that we can evolve it in the future. For example, we may want to add an additional timestamp to the VersionedRecord class to represent the expiry time of the record version (i.e., the timestamp of the next record version for this key) in addition to the existing timestamp.

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

/**
 * Combines a value from a {@link KeyValue} with a timestamp, for use as the return type
 * from {@link VersionedKeyValueStore#get(Object, long)} and related methods.
 *
 * @param <V> The value type
 */
public final class VersionedRecord<V> {
    private final V value;
    private final long timestamp;

    /**
     * Create a new {@link VersionedRecord} instance. {@code value} cannot be {@code null}.
     *
     * @param value      the value
     * @param timestamp  the timestamp
     * @return a new {@link VersionedRecord} instance
     */
    public VersionedRecord(final V value, final long timestamp) {
        this.value = Objects.requireNonNull(value);
        this.timestamp = timestamp;
    }

    public V value() {
        return value;
    }

    public long timestamp() {
        return timestamp;
    }

    @Override
    public String toString() {
        return "<" + value + "," + timestamp + ">";
    }

    @Override
    public boolean equals(final Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        final VersionedRecord<?> that = (VersionedRecord<?>) o;
        return timestamp == that.timestamp &&
            Objects.equals(value, that.value);
    }

    @Override
    public int hashCode() {
        return Objects.hash(value, timestamp);
    }
}
```

Store Supplier/Builder Interfaces

The new Stores.java methods are as follows:

```
public final class Stores {

    // ... existing methods ...

    /**
     * Create a persistent versioned key-value store {@link VersionedBytesStoreSupplier}.
     * <p>
```

```
     * This store supplier can be passed into a
     * {@link #versionedKeyValueStoreBuilder(VersionedBytesStoreSupplier, Serde, Serde)}.
     *
     * @param name             name of the store (cannot be {@code null})
     * @param historyRetention length of time that old record versions are available for query
     *                         (cannot be negative). If a timestamp bound provided to
     *                         {@link VersionedKeyValueStore#get(Object, long)} is older than this
     *                         specified history retention, then the get operation will not return data.
     * @return an instance of {@link VersionedBytesStoreSupplier}
     * @throws IllegalArgumentException if {@code historyRetention} or {@code segmentInterval} can't be
represented as {code long milliseconds}
     */
    public static VersionedBytesStoreSupplier persistentVersionedKeyValueStore(final String name,
                                                                                final Duration historyRetention)
{
        // ...
    }

    /**
     * Create a persistent versioned key-value store {@link VersionedBytesStoreSupplier}.
     * <p>
     * This store supplier can be passed into a
     * {@link #versionedKeyValueStoreBuilder(VersionedBytesStoreSupplier, Serde, Serde)}.
     *
     * @param name             name of the store (cannot be {@code null})
     * @param historyRetention length of time that old record versions are available for query
     *                         (cannot be negative). If a timestamp bound provided to
     *                         {@link VersionedKeyValueStore#get(Object, long)} is older than this
     *                         specified history retention, then the get operation will not return data.
     * @param segmentInterval  size of segments for storing old record versions (must be positive). Old record
versions
     *                         for the same key in a single segment are stored (updated and accessed) together.
     *                         The only impact of this parameter is performance. If segments are large
     *                         and a workload results in many record versions for the same key being collected
     *                         in a single segment, performance may degrade as a result. On the other hand,
     *                         reads and out-of-order writes which access older segments may slow down if
     *                         there are too many segments.
     * @return an instance of {@link VersionedBytesStoreSupplier}
     * @throws IllegalArgumentException if {@code historyRetention} or {@code segmentInterval} can't be
represented as {code long milliseconds}
     */
    public static VersionedBytesStoreSupplier persistentVersionedKeyValueStore(final String name,
                                                                                final Duration historyRetention,
                                                                                final Duration segmentInterval) {
               // ...
       }

         /**
     * Creates a {@link StoreBuilder} that can be used to build a {@link VersionedKeyValueStore}.
     *
     * @param supplier   a {@link VersionedBytesStoreSupplier} (cannot be {@code null})
     * @param keySerde   the key serde to use
     * @param valueSerde the value serde to use; if the serialized bytes is {@code null} for put operations,
     *                   it is treated as a deletion
     * @param <K>        key type
     * @param <V>        value type
     * @return an instance of a {@link StoreBuilder} that can build a {@link VersionedKeyValueStore}
     */
    public static <K, V> StoreBuilder<VersionedKeyValueStore<K, V>> versionedKeyValueStoreBuilder(final
VersionedBytesStoreSupplier supplier,
                                                                                                  final
Serde<K> keySerde,
                                                                                                  final
Serde<V> valueSerde) {
        // ...
    }
}
```

To understand the history retention and segment interval parameters for the `persistentVersionedKeyValueStore()` methods requires brief discussion of the planned RocksDB implementation for versioned stores.

RocksDB Implementation Overview

Here's a high-level overview of the RocksDB versioned store implementation (details are outside the scope of this KIP).

Each store has an associated, fixed-duration *history retention* which specifies how long old record versions should be kept for. In particular, a versioned store guarantees to return accurate results for calls to `get(key, asOfTimestamp)` where the provided timestamp bound is within history retention of the current observed stream time. (If the timestamp bound is outside the specified history retention, then a record is still returned if the latest record version for the key satisfies the timestamp bound. Otherwise, a warning is logged and null is returned.)

To achieve this, the store will consist of a "latest value store" and "segment stores." The latest record version for each key will be stored in the latest value store, while all older versions will be stored in the segment stores.

Each record version has two associated timestamps:

- a `validFrom` timestamp. This timestamp is explicitly associated with the record as part of the put() call to the store; i.e., this is the record's timestamp.
- a `validTo` timestamp. This is the timestamp of the next record (or deletion) associated with the same key, and is implicitly associated with the record. This timestamp can change as new records are inserted into the store.

The validity interval of a record is from validFrom (inclusive) to validTo (exclusive), and can change as new record versions are inserted into the store (and validTo changes as a result).

Old record versions are stored in segment stores according to their validTo timestamps. The use of segments here is analogous to that in the existing RocksDB implementation for windowed stores. Because records are stored in segments based on their validTo timestamps, this means that entire segments can be expired at a time once the records contained in the segment are no longer relevant based on the store's history retention. (A difference between the versioned store segments implementation and that of windowed stores today is that for versioned stores all segments will share the same physical RocksDB instance, in contrast to windowed stores where each segment is its own RocksDB, to allow for many more segments than windowed stores use today.)

The segment interval parameter for controlling segment size is (optionally) exposed to users in the static constructor methods above because benchmarking a prototype implementation showed that this parameter has significant effect on store performance based on workload characteristics.

VersionedBytesStoreSupplier Interface

Here's the VersionedBytesStoreSupplier interface used by the Stores.java methods above:

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

/**
 * A store supplier that can be used to create one or more versioned key-value stores,
 * specifically, {@link VersionedBytesStore} instances.
 * <p>
 * Rather than representing the returned store as a {@link VersionedKeyValueStore} of
 * type <Bytes, byte[]>, this supplier interface represents the returned store as a
 * {@link KeyValueStore} of type <Bytes, byte[]> (via {@link VersionedBytesStore}) in order to be compatible
with
 * existing DSL methods for passing key-value stores such as {@link StreamsBuilder#table(String, Materialized)}
 * and {@link KTable#filter(Predicate, Materialized)}. A {@code VersionedKeyValueStore<Bytes, byte[]>}
 * is represented as a {@code KeyValueStore KeyValueStore<Bytes, byte[]>} by interpreting the
 * value bytes as containing record timestamp information in addition to raw record values.
 */
public interface VersionedBytesStoreSupplier extends KeyValueBytesStoreSupplier {

    /**
     * Returns the history retention (in milliseconds) that stores created from this supplier will have.
     * This value is used to set compaction configs on store changelog topics (if relevant).
        *
     * @return history retention, i.e., length of time that old record versions are available for
     *         query from a versioned store
     */
    long historyRetentionMs();
}
```

As mentioned in the Javadoc, the reason that this supplier extends `KeyValueBytesStoreSupplier` and therefore returns a store of type `KeyValueStore<Bytes, byte[]>` rather than a `VersionedKeyValueStore<Bytes, byte[]>` (as the name suggests) is in order to be compatible with existing DSL methods for passing key-value stores, e.g., StreamsBuilder#table() and KTable methods, which are explicitly typed to accept `Materialized<K, V, KeyValueStore<Bytes, byte[]>` . The alternative to fitting `VersionedKeyValueStore` into `KeyValueStore` in this way is to introduce new versions of all relevant StreamsBuilder and KTable methods to relax the `Materialized` type accepted by these methods. While this is possible, and we could even deprecate the existing methods in favor of the new ones introduced, this is a large surface area for public interface changes that it's best to avoid if possible.

The cost of fitting `VersionedKeyValueStore` into `KeyValueStore` as proposed is two additional layers of translation, for both DSL and PAPI users, whenever put() or get() is called on a versioned store: the record being written or read must be converted between `(keyBytes, valueBytes, timestamp)` and `(keyBytes, valueBytes + serializedTimestamp)` and back. It also means that users who wish to create their own `VersionedKeyValueStore` implementation (specifically, PAPI users who want to use the provided `Stores#versionedKeyValueStoreBuilder` method, and DSL users) also need to mimic this translation layer from `VersionedKeyValueStore` to `KeyValueStore` and back.

To alleviate this pain, we could expose an additional helper method for the conversion and/or add an additional method to `VersionedBytesStoreSupplier` which directly returns a `VersionedKeyValueStore<Bytes, byte[]>` if implemented. The latter allows us to save on the two additional layers of translation, at the expense of complicating one of the interfaces. Unless reviewers feel strongly about this (avoiding the extra translation and/or making it easier for users to create their own `VersionedKeyValueStore` implementations), I propose to leave these options out for now and we can always revisit them later.

For completeness, here's the new VersionedBytesStore interface which VersionedBytesStoreSupplier instances will return. Unless a user chooses to implement their own VersionedBytesStoreSupplier (i.e., in order to implement a custom versioned store to pass to the DSL or to the new Stores#versionedKeyValueStoreBuilder() method), then users will not need to interact with this interface.

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

/**
 * A representation of a versioned key-value store as a {@link KeyValueStore} of type <Bytes, byte[]>.
 * See {@link VersionedBytesStoreSupplier} for more.
 */
public interface VersionedBytesStore extends KeyValueStore<Bytes, byte[]>, TimestampedBytesStore {

    /**
     * The analog of {@link VersionedKeyValueStore#get(Object, long)}.
     */
    byte[] get(Bytes key, long asOfTimestamp);

        /**
     * The analog of {@link VersionedKeyValueStore#delete(Object, long)}.
     */
    byte[] delete(Bytes key, long timestamp);
}
```

Internally, this interface will be used to assist in the representation of `VersionedKeyValueStore<Bytes, byte[]>` as `KeyValueStore<Bytes, byte[]>`.

Additional Interface Changes

TopologyTestDriver

TopologyTestDriver users should be able to get (and interact with) versioned stores from their topology, similar to the existing methods for other store types:

```
public class TopologyTestDriver implements Closeable {

        // ... existing methods ...

         /**
     * Get the {@link VersionedKeyValueStore} with the given name.
     * The store can be a "regular" or global store.
     * <p>
     * This is often useful in test cases to pre-populate the store before the test case instructs the topology
to
     * {@link TestInputTopic#pipeInput(TestRecord) process an input message}, and/or to check the store
afterward.
     *
     * @param name the name of the store
     * @return the key value store, or {@code null} if no {@link VersionedKeyValueStore} has been registered
with the given name
     * @see #getAllStateStores()
     * @see #getStateStore(String)
     * @see #getKeyValueStore(String)
     * @see #getTimestampedKeyValueStore(String)
     * @see #getWindowStore(String)
     * @see #getTimestampedWindowStore(String)
     * @see #getSessionStore(String)
     */
    public <K, V> VersionedKeyValueStore<K, V> getVersionedKeyValueStore(final String name) {
        // ...
    }
}
```

ValueAndTimestamp

Another additional interface change needed as part of this proposal is to add the following static constructor to `ValueAndTimestamp` :

```
public final class ValueAndTimestamp<V> {

        // ... existing methods ...

    /**
     * Create a new {@link ValueAndTimestamp} instance. The provided {@code value} may be {@code null}.
     *
     * @param value      the value
     * @param timestamp  the timestamp
     * @param <V> the type of the value
     * @return a new {@link ValueAndTimestamp} instance
     */
    public static <V> ValueAndTimestamp<V> makeAllowNullable(
        final V value, final long timestamp) {
        // ...
    }
}
```

The reason this addition is needed is an implementation detail. The existing DSL processor implementation represents all source table state stores as timestamped key-value stores (link) which means that, unless we want to lift this restriction and change a significant amount of code internally, then versioned key-value stores will have to fit the `TimestampedKeyValueStore` interface internally. `TimestampedKeyValueStore` represents inserting a new record to the store as calling `put(K key, ValueAndTimestamp<V> v)`. In order to allow inserting tombstones into versioned stores, `ValueAndTimestamp` therefore needs to allow null values.

Even though this is a public interface change (by virtue of `ValueAndTimestamp` being public), the usage of `ValueAndTimestamp` instances with null values will be purely internal. In other words, this change is not strictly needed as a public interface change and could also be achieved through refactoring. That said, the cost of introducing this new method seems low and I'd like to propose it.

Compatibility, Deprecation, and Migration Plan

This KIP introduces a new type of store without deprecating any existing interfaces. Unless a user explicitly updates their application code to use the new store, this KIP will have no effect on their applications (versioned stores are not used anywhere by default).

The RocksDB format used for versioned stores is not compatible with the existing format for non-versioned stores.

However, RocksDB-based versioned and non-versioned stores will use the same changelog topic format, though their changelog topic configurations will differ. Specifically, the changelog bytes format for RocksDB-based versioned and non-versioned stores is the same, but changelog topics for versioned stores need `min.compaction.lag.ms` set to a value suitable for the desired history retention of the versioned store. The RocksDB versioned store implementation will set `min.compaction.lag.ms` equal to history retention plus 24 hours, where the purpose of this additional buffer is to account for the broker's usage of wall clock time in topic compactions (analogous to the extra 24 hours changelog retention for windowed stores today). Changelog topic configs will be set only on changelog topic creation, and will not be verified if the changelog topic already exists.

In light of the above, users can use the following manual procedure to update an existing application with a non-versioned store to use a versioned store instead:

1. Stop the application
2. Delete all local state (for the store being updated) from all instances
3. Update the changelog topic configurations to set `min.compaction.lag.ms` to a value suitable for the desired history retention (e.g., history retention plus some buffer to account for broker wall clock time usage in topic cleanup)
4. Update the application code to use a versioned store
5. Restart the app.

There are no plans to support a non-manual upgrade procedure or a live migration procedure at this time. In the future, it could be nice to make versioned stores the default since a non-versioned store is simply a special case of a versioned store (with history retention 0) but that's far out of scope for this KIP.

Test Plan

The RocksDB-based versioned store implementation will be tested with the Processor API: put, get, and timestamp-based get methods will have their results validated.

The manual procedure described above for updating an application using a non-versioned store to use a versioned store will be tested as well.

Rejected Alternatives

Versioned Store Interface

History retention and get(key, `asOfTimestamp`)

In the event that `get(key, asOfTimestamp)` is called with a timestamp bound older than the specified history retention, instead of returning null (and logging a warning) as proposed above, other design options include (1) throwing an exception or (2) updating the return type from `VersionedRecord<V>` to `Optional<VersionedRecord<V>>` and returning an empty optional to indicate that the timestamp bound was invalid. The first option is not very user-friendly. The second option complicates the interface and diverges the return types of `get(key)` and `get(key, asOfTimestamp)` .

Regarding the edge case where `get(key, asOfTimestamp)` is called with a timestamp bound older than the specified history retention but the latest record version for the key satisfies the timestamp bound, the proposal above says that the latest record version should be returned in this case, rather than rejecting the timestamped query and returning null. Returning the record is preferable because its existence (as the latest value for the key) is guaranteed in the store, and is accessible from `get(key)` anyway. The alternative of returning null, i.e., strict enforcement of the store's history retention, is not very user-friendly as users would then have to determine whether to call `get(key)` or `get(key, timestamp)` to account for this edge case.

ValueAndTimestamp as return type of get(key, asOfTimestamp) / Additional return timestamps from get(key, `asOfTimestamp`)

The proposed return type from `get(key, asOfTimestamp)` of `VersionedRecord<V>` returns the record value and timestamp (i.e., validFrom timestamp) found for the given key (and timestamp bound). In some situations, it may be useful for users to additionally have the validTo timestamp associated with the record. In order to allow for this possibility in the future, the return type of `get(key, asOfTimestamp)` is a new type, `VersionedRecord` , rather than the existing `ValueAndTimestamp<V>` type, even though the two are largely the same today. We considered keeping the interface simple by not introducing a new type, but felt that the flexibility of evolving this type in the future was worth the addition of a new class. However, we will not add additional return timestamps at this time. They can be added once we have more confidence that they will be useful for users.

Return null with timestamp from get()

In the event that `get(key)` or `get(key, asOfTimestamp)` finds that the latest record version associated with a particular key (and possible timestamp bound) is a tombstone, rather than returning null the versioned store could instead return a non-null `VersionedRecord` with null value (and relevant timestamp). This would allow users to distinguish between the key not being found in the store at all (null `VersionedRecord`) versus the key being found with a tombstone for the latest record (non-null `VersionedRecord` with null value). This proposal was rejected since the use cases for making such a distinction are limited.

Grace period separately configurable from history retention

"History retention" and "grace period" control how far back in time (relative to the current observed stream time) old reads and writes, respectively, will be accepted by the store. In the proposal above, users specify a single value which is used for both parameters, though in the future we could add an additional option for users to specify the two separately. (Today, users specify an explicit value for history retention, and grace period is automatically set to the same value. There are no compatibility concerns with introducing a new option for grace period in the future.)

Support for Upgrades

Additional support for upgrading a non-versioned store to a versioned store beyond the manual steps above were rejected on the basis of complexity. Automatic upgrades are too complex, and it's not clear that additional tooling for manual upgrades would be valuable to users at this time. It's better to get the new versioned interfaces out sooner in order to let them bake/iterate, rather than block on additional complexity for introducing the first version at this time.

Versioned Windowed Stores

This KIP only proposes to introduce versioned stores for key-value stores and not windowed (or session) stores since use cases for versioned windowed stores are limited. By not introducing versioned windowed stores, there is also the potential to unify the underlying implementations of versioned stores and windowed stores in the future by leveraging the shared validFrom/validTo timestamp abstraction. Any such unification is a long way off and not in scope for this KIP.

Additional Scope

As noted above, there are a number of additional features needed to complete the story around versioned stores. While these are important, they are all deferred to follow-up KIPs. Example features include providing an in-memory implementation of the versioned store interface, supporting interactive queries, new metrics/monitoring specific to versioned stores, updating DSL processors to use versioned stores (if provided) to address gaps in join semantics noted in the motivation section, and potentially adding additional methods to the versioned store interface.