KIP-205: Add all() and range() API to ReadOnlyWindowStore

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *KAFKA-4499*

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

Motivation

Currently, both KTable and windowed-KTable stores can be queried via IQ feature. While ReadOnlyKeyValueStore(for KTable stores) provide method all() to scan the whole store (ie, returns an iterator over all stored key-value pairs), there is no similar API for ReadOnlyWindowStore (for windowed-KTable stores).

This limits the usage of a windowed store, because the user needs to know what keys are stored in order the query it.

Public Interface

The current methods supported in ReadOnlyWindowStore are the fetch methods which are relatively limited in scope when taking into account the requirement for the user to know the specific Window Keys beforehand:

Fetch methods found in ReadOnlyWindowStore

```
/**
 * Get all the key-value pairs with the given key and the time range from all
 * the existing windows.
 * <p>
 * The time range is inclusive and applies to the starting timestamp of the window.
 * For example, if we have the following windows:
 * <p>
 * <pre>
 * +-------------------------------+
 * |  key  | start time | end time |
 * +-------+------------+----------+
 * |   A   |     10     |    20    |
 * +-------+------------+----------+
 * |   A   |     15     |    25    |
 * +-------+------------+----------+
 * |   A   |     20     |    30    |
 * +-------+------------+----------+
 * |   A   |     25     |    35    |
 * +-------------------------------
 * </pre>
 * And we call {@code store.fetch("A", 10, 20)} then the results will contain the first
 * three windows from the table above, i.e., all those where 10 <= start time <= 20.
 * <p>
 * For each key, the iterator guarantees ordering of windows, starting from the oldest/earliest
 * available window to the newest/latest window.
 *
 * @return an iterator over key-value pairs {@code <timestamp, value>}
 * @throws InvalidStateStoreException if the store is not initialized
 */
WindowStoreIterator<V> fetch(K key, long timeFrom, long timeTo);

/**
 * Get all the key-value pairs in the given key range and time range from all
 * the existing windows.
 *
 * @param from      the first key in the range
 * @param to        the last key in the range
 * @param timeFrom  time range start (inclusive)
 * @param timeTo    time range end (inclusive)
 * @return an iterator over windowed key-value pairs {@code <Windowed<K>, value>}
 * @throws InvalidStateStoreException if the store is not initialized
 * @throws NullPointerException If null is used for any key.
 */

KeyValueIterator<Windowed<K>, V> fetch(K from, K to, long timeFrom, long timeTo);
```

Methods similar to the ones found in `ReadOnlyKeyValueStore` will be introduced into the `ReadOnlyWindowStore` interface. They are the following:

Additional methods to ReadOnlyWindowStore

```
    /**
     * Gets all the key-value pairs in the existing windows.
     *
     * @returns an iterator over windowed key-value pairs {@code <Windowed<K>, value>}
     * @throws InvalidStateStoreException if the store is not initialized
     */
    KeyValueIterator<Windowed<K>, V> all();

    /**
     * Gets all windows for a specific time range
     *
         * @param timeFrom the beginning of the time slot from which to search
         * @param timeTo   the end of the time slot from which to search
     * @returns an iterator over windowed key-value pairs {@code <Windowed<K>, value>}
         * @throws InvalidStateStoreException if the store is not initialized
     * @throws NullPointerException if null is used for any key
     */
    KeyValueIterator<Windowed<K>, V> fetchAll(long timeFrom, long timeTo);
```

Proposed Changes

With the need to test the implementation of these methods in mind, the ReadOnlyWindowStoreStub class will implement these additions (i.e. `@Override public Collection<Windowed<K>> range(long timeFrom, long timeTo){...}`). The corresponding test will also be modified. (e.g. `@Test public void testRange(){...}`)

Following the changes in RocksDBWindowStore, CachingWindowStore and other WindowStore API will implement these changes accordingly.

Of particular interest, however, is that in each of these classes, they require a different approach towards implementing each one. For example, take CachingWindowStore. The fetch methods are implemented using a SegmentedCacheFunction instance:

CachingWindowStore fetch() method

```
        @Override
    public synchronized WindowStoreIterator<byte[]> fetch(final Bytes key, final long timeFrom, final long
timeTo) {
        // since this function may not access the underlying inner store, we need to validate
        // if store is open outside as well.
        validateStoreOpen();
        final WindowStoreIterator<byte[]> underlyingIterator = underlying.fetch(key, timeFrom, timeTo);
        final Bytes cacheKeyFrom = cacheFunction.cacheKey(keySchema.lowerRangeFixedSize(key, timeFrom));
        final Bytes cacheKeyTo = cacheFunction.cacheKey(keySchema.upperRangeFixedSize(key, timeTo));
        final ThreadCache.MemoryLRUCacheBytesIterator cacheIterator = cache.range(name, cacheKeyFrom,
cacheKeyTo);

        final HasNextCondition hasNextCondition = keySchema.hasNextCondition(key,
                                                                             key,
                                                                             timeFrom,
                                                                             timeTo);

        final PeekingKeyValueIterator<Bytes, LRUCacheEntry> filteredCacheIterator = new FilteredCacheIterator(
            cacheIterator, hasNextCondition, cacheFunction
        );
        return new MergedSortedCacheWindowStoreIterator(filteredCacheIterator, underlyingIterator);
    }
```

Pay special attention to cacheFunction which is a class field as well as a SegmentedCacheFunction instance.

Meanwhile, in the RocksDBWindowStore class, we have the following implementation:

RocksDBWindowStore fetch() method

```
        @Override
    public WindowStoreIterator<byte[]> fetch(Bytes key, long timeFrom, long timeTo) {
        final KeyValueIterator<Bytes, byte[]> bytesIterator = bytesStore.fetch(key, timeFrom, timeTo);
        return WindowStoreIteratorWrapper.bytesIterator(bytesIterator, serdes, windowSize).valuesIterator();
    }
```

RocksDBWindowStore, having to serve a vastly different purpose from CachingWindowStore has to implement different key and value structures.

Each of these structures appear to be only different on the surface. However, the underlying principles on which both operate are basically the same. So in effect, the implementation should be relatively similar.

Compatibility, Deprecation, and Migration Plan

Rejected Alternatives