KIP-992: Proposal to introduce IQv2 Query Types: TimestampedKeyQuery and TimestampedRangeQuery

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *Accpted*

Discussion thread:https://lists.apache.org/thread/ogo7ntmj8srdcko2h86vvd9djjsjfvcj

Vote thread: https://lists.apache.org/thread/q4kn2g6tmc837ph2zvff40pgpmgzok3d

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

In the current IQv2 code, there are noticeable differences when interfacing with `kv-store` and `ts-kv-store` . Notably, the return type `V` acts as a simple value for `kv-store` but evolves into `ValueAndTimestamp<V>` for `ts-kv-store` , which presents type safety issues in the API.

shouldHandleKeyQuery

```
public <V> void shouldHandleKeyQuery(
            final Integer key,
            final Function<V, Integer> valueExactor,
            final Integer expectedValue) {
        ...
                  final KeyQuery<Integer, V> query = KeyQuery.withKey(key);
            ...
                  final StateQueryRequest<V> request =
                inStore(STORE_NAME)
                        .withQuery(query)
                        .withPartitions(mkSet(0, 1))
                        .withPositionBound(PositionBound.at(INPUT_POSITION));
                ...
        final StateQueryResult<V> result =
                IntegrationTestUtils.iqv2WaitForResult(kafkaStreams, request);
        ...
        final QueryResult<V> queryResult = result.getOnlyPartitionResult();
        ...
        final V result1 = queryResult.getResult();
        final Integer integer = valueExactor.apply(result1);
        assertThat(integer, is(expectedValue));
       ...
    }
```

Before the introduction of `TimestampedKeyQuery` , when using `KeyQuery` , we obtained the result using the following code:

```
final V result1 = queryResult.getResult();
```

This meant that the returned result could be of two potential data types: plain `V` or `ValueAndTimestamp<V>` . This was a source of inconsistency. For instance, querying a `kv-store` with `KeyQuery` would return a `V` type, but querying a `ts-kv-store` would yield a `ValueAndTimestamp<V>` . This behavior is unintuitive and potentially confusing for developers.

To ensure consistency, we suggest that KeyQuery always return the plain V type, enhancing the predictability of the mentioned code. Likewise, RangeQuery should uniformly return the plain V KeyValueIterator.

For those requiring timestamped values from a `ts-kv-store` , we recommend introducing a new query type: `TimestampedKeyQuery` . This new query will specifically target `ts-kv-stores` and will return `ValueAndTimestamp<V>` . Furthermore, to complement this, `TimestampedRangeQuery` should be introduced to query ranges in `ts-kv-stores` , ensuring that the returned value always includes timestamps.

TimestampKeyQuery

```
public final class TimestampedKeyQuery<K, V> implements Query<ValueAndTimestamp<V>>
```

TimestampRangeQuery

```
public final class TimestampedRangeQuery<K, V> implements Query<KeyValueIterator<K, ValueAndTimestamp<V>>>
```

Why introduce `TimestampedKeyQuery` and `TimestampedRangeQuery` ? The primary motivation behind this is to ensure type safety and foster a clear distinction in our API. They bridge the difference between simple key-value stores and those integrated with timestamps, offering a more robust and intuitive querying mechanism.

Proposed Changes

Within the current IQv2 codebase, there have been distinct interactions between plain-kv-store and ts-kv-store. These differences, especially in return types, have raised concerns over type safety within the API.

To address these challenges and streamline the querying experience, we have decided to refine our approach and introduce two specialized query types: `TimestampedKeyQuery` and `TimestampedRangeQuery` .

`TimestampedKeyQuery` : This query type will consistently return `ValueAndTimestamp<V>` , ensuring that there's a clear and predictable return type associated with timestamped key-value stores.

TimestampKeyQuery

```
@Evolving
public final class TimestampedKeyQuery<K, V> implements Query<ValueAndTimestamp<V>> {

...

    /**
     * Creates a query that will retrieve the record identified by {@code key} if it exists
     * (or {@code null} otherwise).
     * @param key The key to retrieve
     * @param <K> The type of the key
     * @param <V> The type of the value that will be retrieved
     */
    public static <K, V> TimestampedKeyQuery<K, V> withKey(final K key)

    /**
     * Specifies that the cache should be skipped during query evaluation. This means, that the query will
always
     * get forwarded to the underlying store.
     */
    public TimestampedKeyQuery<K, V> skipCache()

    /**
     * The key that was specified for this query.
     */
    public K key()

    /**
     * The flag whether to skip the cache or not during query evaluation.
     */
    public boolean isSkipCache()
```

`TimestampedRangeQuery` : Tailored for ranges with timestamps, this query will return a `KeyValueIterator<K, ValueAndTimestamp<V>>`

According to KIP-968, this KIP introduces the public enum **ResultOrder** to determine whether keys are sorted in ascending or descending or unordered order. Order is based on the serialized byte[] of the keys, not the 'logical' key order. employs the `withDescendingKeys() and withAscendingKeys()` methods to specify that the keys should be ordered in descending or ascending or unordered sequence, and the resultOrder() method to retrieve the value of enum value in **ResultOrder**. I've incorporated these variables and methods into the `TimestampedRangeQuery` class and modified some method inputs. As a result, we can now use `withDescendingKeys()` to obtain results in reverse order and use withAscendingKeys to obtain the result in ascending order.

TimestampedRangeQuery

```
@Evolving
public final class TimestampedRangeQuery<K, V> implements Query<KeyValueIterator<K, ValueAndTimestamp<V>>> {

    ...

    /**
     * Interactive range query using a lower and upper bound to filter the keys returned.
     * @param lower The key that specifies the lower bound of the range
     * @param upper The key that specifies the upper bound of the range
     * @param <K> The key type
     * @param <V> The value type
     */
    public static <K, V> TimestampedRangeQuery<K, V> withRange(final K lower, final K upper)


        /**
     * Determines if the serialized byte[] of the keys in ascending or descending or unordered order.
     * Order is based on the serialized byte[] of the keys, not the 'logical' key order.
     * @return return the order of return records base on the serialized byte[] of the keys (can be unordered,
or in ascending, or in descending order).
     */
    public ResultOrder resultOrder() {
        return order;
    }

    /**
     * Set the query to return the serialized byte[] of the keys in descending order.
     * Order is based on the serialized byte[] of the keys, not the 'logical' key order.
     * @return a new RangeQuery instance with descending flag set.
     */
    public TimestampedRangeQuery<K, V> withDescendingKeys() {
        return new TimestampedRangeQuery<>(this.lower, this.upper, ResultOrder.DESCENDING);
    }

    /**
     * Set the query to return the serialized byte[] of the keys in Ascending order.
     * Order is based on the serialized byte[] of the keys, not the 'logical' key order.
     * @return a new RangeQuery instance with ascending flag set.
     */
    public TimestampedRangeQuery<K, V> withAscendingKeys() {
        return new TimestampedRangeQuery<>(this.lower, this.upper, ResultOrder.ASCENDING);
    }

}
```

According to KIP-968, we introduce a public enum ResultOrder.

ResultOrder enum
It helps with specifying the order of the returned results by the query.

ResultOrder

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

public enum ResultOrder {
    ANY,
    ASCENDING,
    DESCENDING
}
```

Compatibility, Deprecation, and Migration Plan

- Changing the semantics of existing `KeyQuery` and `RangeQuery` is a breaking change. However, both classes are marked as `@Evolving`` and thus a breaking change in a minor release is allowed without a deprecation period. Given that IQv2 is not yet widely adopted, we believe it’s cleaner to make this breaking change right away.
- Adding new query types does not imply any compatibility concerns.

Test Plan

To ensure the robustness and accuracy of our new query types, `TimestampedKeyQuery` and `TimestampedRangeQuery` , it's essential to have thorough test coverage. With that in mind, we propose the creation of two specific test methods:

`shouldHandleTimestampedKeyQuery` : This test method will validate the functionality of `TimestampedKeyQuery` , ensuring it consistently returns `ValueAndTimestamp<V>` as expected.

`shouldHandleTimestampedRangeQuery` : This method is tailored to verify the `TimestampedRangeQuery` , ensuring that it correctly returns a `KeyValueIterator<K, ValueAndTimestamp<V>>` .

We will focus on conducting a detailed test for `shouldHandleTimestampedRangeQuery` .

Rejected Alternatives

The alternative would be to deprecate the existing `KeyQuery` and `RangeQuery` and add new query types that always return plain value. However, it seems to introduce unnecessary “deprecation noise”, and it would be hard to find good names for these newly added query types. Making a semantics change on the existing queries allows us to keep the existing names.