KIP-985: Add reverseRange and reverseAll query over kv-store in IQv2

Status

Current state: *Accepted*

Discussion thread: *here [Change the link from the KIP proposal email archive to your own email thread]*

***Voting thread**: https://lists.apache.org/thread/xxyb5yyqrsdxsyxxbjhvnlxw5fl8xd0c*

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 concepts of `reverseRange` and `reverseAll` are not tied to any specific method or class. Rather, they represent functionalities we wish to achieve. Currently, with `RangeQuery`, we can use methods like `withRange()`, `withLowerBound()`, `withUpperBound()`, and `withNoBounds()`.

Utilizing these, the query results are ordered based on the serialized byte[] of the keys, not the 'logical' key order.

Take `IQv2StoreIntegrationTest` as an example: we have two partitions with four key-value pairs:

- `<0,0>` in Partition0
- `<1,1>` in Partition1
- `<2,2>` in Partition0
- `<3,3>` in Partition1

When we use `RangeQuery.withRange(1,3)`, the returned result is:

- Partition0: `[2]`
- Partition1: `[1, 3]`

To achieve the functionalities of `reverseRange` and `reverseAll`, we can introduce a method named `withDescendingKeys()` for reversed queries. For example, by using `RangeQuery.withRange(1,3).withDescendingKeys()`, the expected result would be:

- Partition0: `[2]`
- Partition1: `[3, 1]`

This means the results are in the reverse order of their keys.

To ensure that we can achieve this functionality, the keys in both `RocksDB` and `InMemoryKeyValueStore` should be sorted. We know that `RocksDB k` eys are inherently sorted. After investigation, we found that `InMemoryKeyValueStore` uses a `TreeMap`, implying its keys are also sorted. Therefore, performing the aforementioned queries is feasible.

Proposed Changes

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 `RangeQuery` 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.

```
/**
 * Interactive query for issuing range queries and scans over KeyValue stores.
 * <p>
 *  A range query retrieves a set of records, specified using an upper and/or lower bound on the keys.
 * <p>
 * A scan query retrieves all records contained in the store.
 * <p>
 */
@Evolving
public final class RangeQuery<K, V> implements Query<KeyValueIterator<K, V>> {
    ...

        /**
     * 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 returned records based on the serialized byte[] of the keys (can be
unordered, or in ascending or in descending order).
     */
    public ResultOrder resultOrder()

    /**
     * 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 RangeQuery<K, V> withDescendingKeys()

    /**
     * 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 RangeQuery<K, V> withAscendingKeys()
      ...
}
```

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
}
```

Test Plan

This time, our goal is to implement `reverseRange` and `reverseAll` functionalities. While these terms are used for clarity, in practice, they correspond to `RangeQuery.withRange().withDescendingKeys()` and `RangeQuery.withNoBounds().withDescendingKeys()`, respectively. To ensure the accurate retrieval of results for both functionalities, adjustments to `IQv2StoreIntegrationTest` are required. In our previous approach, we stored query results in a set, which doesn't maintain order. I've transitioned to using a list for storing query results, enabling us to distinguish between `rangeQuery` and `reverseQuery`. Here, `rangeQuery` refers to standard queries (those not using `withDescendingKeys()`) such as `withRange()`, `withLowerBound()`, `withUpperBound()`, and `withNoBounds()`. In contrast, `reverseQuery` denotes queries that employ the `withDescendingKeys()` method.

We've transitioned the `expectedValue` from a `Set` to a `List` and arranged the partition numbers in order. This organization assists us in predicting the results. If the partition numbers were random, predicting the outcome would be challenging. Ultimately, this enables us to obtain and store the answer in the `expectedValue`. Consequently, the results between `rangeQuery` and `reverseQuery` will differ.

IQv2StoreIntegrationTest

```
public class IQv2StoreIntegrationTest {
...
 @SuppressWarnings("unchecked")
    public <V> void shouldHandleRangeQuery(
                final Optional<Integer> lower,
        final Optional<Integer> upper,
        final boolean isKeyAscending,
        final Function<V, Integer> valueExtactor,
        final List<Integer> expectedValue) {

        final RangeQuery<Integer, V> query;

        if (isKeyAscending) {
            query = RangeQuery.withRange(lower.orElse(null), upper.orElse(null));
        } else {
            query = (RangeQuery<Integer, V>) RangeQuery.withRange(lower.orElse(null), upper.orElse(null)).
withDescendingKeys();
        }
        ...
        } else {
            final List<Integer> actualValue = new ArrayList<>();
           ...
            final List<Integer> partitions = new ArrayList<>(queryResult.keySet());
            partitions.sort(null);
            for (final int partition : partitions) {
                ...
                }
   ...
}
```

Compatibility, Deprecation, and Migration Plan

- Utilizing the existing `RangeQuery` class, we can make some modifications to realize the concepts of `reverseRange` and `reverseAll` . To reiterate, `reverseRange` and `reverseAll` are not classes or methods but merely concepts.
- Since nothing is deprecated in this KIP, users have no need to migrate unless they want to.

Rejected Alternatives

After initial plans to create a `ReverseRangeQuery` from the ground up, we opted to leverage existing code from the `RangeQuery` class following further discussions.

ReverseRangeQuery

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

    private final Optional<K> lower;
    private final Optional<K> upper;

    private ReverseRangeQuery(final Optional<K> lower, final Optional<K> upper) {
        this.lower = lower;
        this.upper = upper;
    }

    /**
     * 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> ReverseRangeQuery<K, V> withRange(final K lower, final K upper) {
        return new ReverseRangeQuery<>(Optional.ofNullable(lower), Optional.ofNullable(upper));
    }

    /**
     * Interactive range query using an upper bound to filter the keys returned.
     * If both <K,V> are null, RangQuery returns a full range scan.
     * @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> ReverseRangeQuery<K, V> withUpperBound(final K upper) {
        return new ReverseRangeQuery<>(Optional.empty(), Optional.of(upper));
    }

    /**
     * Interactive range query using a lower bound to filter the keys returned.
     * @param lower The key that specifies the lower bound of the range
     * @param <K> The key type
     * @param <V> The value type
     */
    public static <K, V> ReverseRangeQuery<K, V> withLowerBound(final K lower) {
        return new ReverseRangeQuery<>(Optional.of(lower), Optional.empty());
    }

    /**
     * Interactive scan query that returns all records in the store.
     * @param <K> The key type
     * @param <V> The value type
     */
    public static <K, V> ReverseRangeQuery<K, V> withNoBounds() {
        return new ReverseRangeQuery<>(Optional.empty(), Optional.empty());
    }

    /**
     * The lower bound of the query, if specified.
     */
    public Optional<K> getLowerBound() {
        return lower;
    }

    /**
     * The upper bound of the query, if specified
     */
    public Optional<K> getUpperBound() {
        return upper;
    }
}
```