KIP-614: Add Prefix Scan support for State Stores

Status

Current state: *"Accepted"*

Discussion thread: here

JIRA: KAFKA-10648

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

Motivation

The idea of this KIP is to add prefix scan support to State Stores. This would be helpful for cases when the keys share common prefixes and users want to iterate over them. With prefix scan, the overhead of several I/O requests can be reduced. Currently, this is not possible and, users, despite knowing prefix patterns in the keys, would still need to make multiple calls to iterate over them.

Public Interfaces

The proposal is to add a new method to **ReadOnlyKeyValueStore** called **prefixScan**. This method takes 2 arguments, 1) the prefix to search for and 2) A serializer for the Prefix Key Type.

prefixScan

```
        /**
     * Get an iterator over keys which have the specified prefix. The type of the prefix can be different from
that of
     * the key. That's why, callers should also pass a serializer for the prefix to convert the prefix into the
     * format in which the keys are stored underneath in the stores
     * @param prefix The prefix.
     * @param prefixKeySerializer Serializer for the Prefix key type
     * @param <PS> Prefix Serializer type
     * @param <P> Prefix Type.
     * @return The iterator for keys having the specified prefix.
     */
    <PS extends Serializer<P>, P> KeyValueIterator<K, V> prefixScan(P prefix, PS prefixKeySerializer);
```

This method would also have a default implementation that throws an UnsupportedOperationException for source compatibility with existing state stores.

Proposed Changes

All the classes which implement ReadOnlyKeyValueStore and which support prefixScan would implement this method. Rest of the classes would throw use the default implementation which throws an UnsupportedOperationException.

As an example, plz review the following code for RocksDB store:

RocksDBStore(Highlighting only the changes)

RockDBStore

```
public class RocksDBStore implements KeyValueStore<Bytes, byte[]>, BatchWritingStore {

        // Current code
        // implement prefixScan
        @Override
    public <PS extends Serializer<P>, P> KeyValueIterator<Bytes, byte[]> prefixScan(P prefix, PS
prefixKeySerializer) {
        Objects.requireNonNull(prefix, "prefix cannot be null");
        Objects.requireNonNull(prefixKeySerializer, "prefixKeySerializer cannot be null");

        validateStoreOpen();
        Bytes prefixBytes = Bytes.wrap(prefixKeySerializer.serialize(null, prefix));

        final KeyValueIterator<Bytes, byte[]> rocksDbPrefixSeekIterator = dbAccessor.prefixSeek(prefixBytes);
        openIterators.add(rocksDbPrefixSeekIterator);

        return rocksDbPrefixSeekIterator;
    }

        interface RocksDBAccessor {
        // Current RocksDBAccessor code

                KeyValueIterator<Bytes, byte[]> prefixScan(final Bytes prefix);
        }

        class SingleColumnFamilyAccessor implements RocksDBAccessor {
                // Current Accessor code.

                 @Override
                public KeyValueIterator<Bytes, byte[]> prefixScan(final Bytes prefix) {
              return new RocksDBPrefixIterator(name, db.newIterator(columnFamily), openIterators, prefix);
                }
        }
}
```

RocksDBPrefixIterator

```
class RocksDBPrefixIterator extends RocksDbIterator {
    private byte[] rawPrefix;

    RocksDBPrefixIterator(final String name,
                          final RocksIterator newIterator,
                          final Set<KeyValueIterator<Bytes, byte[]>> openIterators,
                          final Bytes prefix) {
        super(name, newIterator, openIterators);
        this.rawPrefix = prefix.get();
        newIterator.seek(rawPrefix);
    }

    private boolean prefixEquals(final byte[] x, final byte[] y) {
        final int min = Math.min(x.length, y.length);
        final ByteBuffer xSlice = ByteBuffer.wrap(x, 0, min);
        final ByteBuffer ySlice = ByteBuffer.wrap(y, 0, min);
        return xSlice.equals(ySlice);
    }

    @Override
    public KeyValue<Bytes, byte[]> makeNext() {
        final KeyValue<Bytes, byte[]> next = super.makeNext();
        if (next == null) return allDone();
        else {
            if (prefixEquals(this.rawPrefix, next.key.get())) return next;
            else return allDone();
        }
    }
}
```

This is a new iterator extending the current RocksDbIterator. It invokes the built-in seek() method in Java Rocks-DB which is a wrapper over the Prefix Seek apis exposed. More information can be found here:

https://github.com/facebook/rocksdb/wiki/Prefix-Seek

Similar to the implementation for RocksDB, we would implement the prefix scan for InMemoryKeyValueStore as well.

Prefix Key Serializer

One thing which should be highlighted about the prefixScan method is the **prefixKeySerializer** argument. This is needed because the type of the prefix could be different from the type of the actual key. For example, the key in the store could be of type UUID like 123e4567-e89b-12d3-a456-426614174000. The user wants to get all keys which have the prefix 123e4567 which could be represented as a String/byte array but not a UUID. But, since all the keys are serialized in the form of byte arrays if we can serialize the prefix key, then it would be possible to do a prefix scan over the byte array key space. The argument **prefixKeySerializer** is provided precisely for this purpose.

Here is a test case which highlights the above point:

PrefixKeySerializer Usage Example

```
@Test
    public void shouldReturnUUIDsWithStringPrefix() {
        final List<KeyValue<Bytes, byte[]>> entries = new ArrayList<>();
        Serializer<UUID> uuidSerializer = Serdes.UUID().serializer();
        UUID uuid1 = UUID.randomUUID();
        UUID uuid2 = UUID.randomUUID();
        String prefix = uuid1.toString().substring(0, 4);
        entries.add(new KeyValue<>(
                new Bytes(uuidSerializer.serialize(null, uuid1)),
                stringSerializer.serialize(null, "a")));

        entries.add(new KeyValue<>(
                new Bytes(uuidSerializer.serialize(null, uuid2)),
                stringSerializer.serialize(null, "b")));


        rocksDBStore.init(context, rocksDBStore);
        rocksDBStore.putAll(entries);
        rocksDBStore.flush();

        final KeyValueIterator<Bytes, byte[]> keysWithPrefix = rocksDBStore.prefixScan(prefix,
stringSerializer);
        String[] valuesWithPrefix = new String[1];
        int numberOfKeysReturned = 0;

        while (keysWithPrefix.hasNext()) {
            KeyValue<Bytes, byte[]> next = keysWithPrefix.next();
            valuesWithPrefix[numberOfKeysReturned++] = new String(next.value);
        }

        assertEquals(1, numberOfKeysReturned);
        assertEquals(valuesWithPrefix[0], "a");
    }
```

Compatibility, Deprecation, and Migration Plan

Since the proposal adds the prefixScan method at ReadOnlyKeyValueStore interface level, we will need to either implement the method for those stores which support prefixScan. For source compatibility purposes, rest of the stores would use the default implementation which throws UnsupportedOperationException

Rejected Alternatives

The first alternative to add a separate interface called PrefixSeekableStore and have only relevant stores implement it was rejected. This is because it was not compatible with the way users of Kafka Streams define state stores in the application.

Current Implementation Details

Basic implementation described above can be found here:

https://github.com/confluentinc/kafka/pull/242.

Other considerations

While having discussions with the Kafka Streams core members on the Slack channel, one of the points brought up by Sophie Blee-Goldman was the performance of Prefix Seek API of Rocks DB Java. There was an active issue on the Facebook RocksDB side which I had mentioned to them on this issue here: https://github.com/facebook/rocksdb/issues/6004 which got closed recently. I had created a JIRA on Kafka to track the activity of this issue and if at all it makes sense to integrate these changes back. Here's the JIRA issue for reference :