Discussion: Non-key KTable-KTable Joins

Background and Motivation

The current (as of 0.10.0.0) semantic design of Table-Table joins can be found here. A brief summary:

- We require two joining KTables to have the same key, and hence their corresponding source Kafka topics are also partitioned on that key.
- We treat "windowed tables" (I will just call them WKTable in the following) as a special type of KTable<Windowd<K>, V>, which does not have the same key as KTable<K, V>.

However, we observed that there are at least two common cases which require non-key KTable-KTable joins, more specifically, joining two KTable streams that do not necessarily have the same key:

- The first common use case is "table join by foreign-key". Details can be found here:

  Unable to render Jira issues macro, execution error.
- The second common use case is WKTable-KTable join or WKTable-WKTable join. For example, think of "subscribing to a feed of log4j output and detect any error that is at least N standard deviations higher than the same time last week (assuming a sliding window of 1 minute every 1 second)". They are thought about in the original design (see this page for details), but decided to be discarded for the first release due to API complexity.

After 0.10.0.0, we can think of adding this support of non-keyed KTable-KTable joins to handle the above two common cases (treating KWTable just as a special key-typed KTable).

Design Proposal

This is one design proposal just for kick-offing the discussion.

Added APIs to Kafka Streams DSL

```
public interface KTable<K, V> {

    // mapper is used for mapping this table's key-value into other table's key;
    // selector is used for combining the joining tables' keys as the result table's key.

    <K1, V1, R> KTable<K2, R> join(KTable<K1, V1> other, KeyValueMapper<K, V, K1> mapper, KeyValueMapper<K, K1,
K2> selector, ValueJoiner<V, V1, R> joiner);

    <K1, V1, R> KTable<K2, R> leftJoin(KTable<K1, V1> other, KeyValueMapper<K, V, K1> mapper, KeyValueMapper<K,
K1, K2> selector, ValueJoiner<V, V1, R> joiner);

    <K1, V1, R> KTable<K2, R> outerJoin(KTable<K1, V1> other, KeyValueMapper<K, V, K1> mapper,
KeyValueMapper<K, K1, K2> selector, ValueJoiner<V, V1, R> joiner);
}
```

With the above APIs, the following use cases can be implemented:

Foreign-key Join

```
KTable<KA, VA> tableA = builder.table("topicA");

KTable<KB, VB> tableB = builder.table("topicB");    // assuming topicB's message value payload contains field
of "KA"

KTbale<KB, VR> tableR = tableB.join(tableA, (keyB, valueB) -> valueB.get("fieldKA"), (keyB, keyA) -> keyB,
(valueA, valueB) -> joined);   // join return type "VR"
```

Non-window to Window Table Join

For example, a windowed-table would "query" another non-windowed table to get augmented values.

```
KTable<KA, VA> tableA = builder.table("topicA");

KTable<Windowed<KA>, VB> tableB = stream.aggregate(...);

KTbale<Windowed<KA>, VR> tableR = tableB.join(tableA, (windowKeyB, valueB) -> windowKeyB.key, (windowKeyB,
keyA) -> windowed(combine(windowKeyB.key, keyA)), (valueA, valueB) -> joined);   // join return type "KR"
```

Window to Window Table Join

For example, comparing a windowed-table with itself (self-join) by a shifted period.

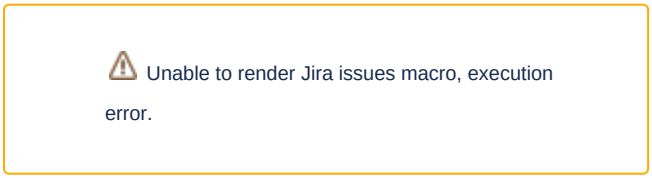
```
KTable<Windowed<KA>, VA> tableA = stream.aggregate(...);

KTbale<Windowed<KA>, VR> tableR = tableA.join(tableA, (windowKeyA, valueA) -> windowKeyA.shiftLeft(1000),
(windowKeyA, windowKeyB) -> windowKeyB, (valueA, valueB) -> joined);
```

Implementation Details

In the following section we only talk about the first case, where the mapper is for mapping the other table's key-value pair into this table's key:

1. First of all, we will repartition this KTable's stream, by key computed from the ***mapper(K, V) → K1***, so that it is co-partitioned by the same key. The co-partition topic is partitioned on the new key, but the message key and value are unchanged, and log compaction is turned off.
2. After re-partitioning the other table, materialize both streams. The other table is materialized as ***K1 → V1***, and this table is materialized as combo key **(K1, K)** with value **V** (note that we need to apply the ***mapper(K, V) → K1*** again).
3. When a record ***(k1, v1)*** is received from the other table's stream, update its materialized table, and then make a range query on the other materialized table as ***(k1, *)***, and for each matched record ***(k1, k) → v*** apply the ***joiner(v, v1) → r***, and send ***(k2, r)*** *where* ***select(k, k1) -> k2.***
4. When a record ***(k, v)*** is received from this table's repartitioned stream, update its materialized table by applying the mapper, and then make a get query on the this materialized table by mapped key ***mapper(k, v) -> k1***, and for the single matched record ***k1 → v1*** apply ***reversed joiner(v1, v) → r***, and return ***(k2, r)*** *where* ***select(k, k1) -> k2***.

NOTE: When sendOldValues is turned on, where a pair <old, new> is passed into the other KTable's stream, then the mapped key may be different, and hence two separate records will be sent to the re-partition topic, one for removal and one for addition. And these two records will be joined separately and independently, and hence their output ordering is arbitrary, and hence the joining results that keeps track of the changes as ***k -> {r_old, r_new}*** could be incorrect if it appends ***k -> {null, r_new}*** before ***k -> {r_old, null}*** (details can be found in



). That is why we need to have the result table key as a combination of both joining table keys, instead of just inheriting the left join table's key for normal foreign-key join.

Now the only question left is how we can do range query on combo ***(key, *).*** There are two approaches for this:

Simple approach: seek with key directly

This is just a simplified version of Option 1 below: since RocksIterator.seek() guarantees that it "position at the first key in the source that at or past target; the iterator is Valid() after this call iff the source contains // an entry that comes at or past target." (github code), we can just do a seek(key), which is supposed to locate at the first element that start with this prefix. And since the lexicographical comparator is used by default, elements with the same key prefix will be guaranteed to be exhaustively iterable until the prefix is no longer "key".

See the example code:

```
public static void main(String[] args) {

    BlockBasedTableConfig tableConfig = new BlockBasedTableConfig();
    tableConfig.setBlockCacheSize(BLOCK_CACHE_SIZE);
    tableConfig.setBlockSize(BLOCK_SIZE);

    Options options = new Options();
    options.setTableFormatConfig(tableConfig);
    options.setWriteBufferSize(WRITE_BUFFER_SIZE);
    options.setCompressionType(COMPRESSION_TYPE);
    options.setCompactionStyle(COMPACTION_STYLE);
    options.setMaxWriteBufferNumber(MAX_WRITE_BUFFERS);
    options.setCreateIfMissing(true);
    options.setErrorIfExists(false);

    WriteOptions wOptions = new WriteOptions();
    wOptions.setDisableWAL(true);

    FlushOptions fOptions = new FlushOptions();
    fOptions.setWaitForFlush(true);

    RocksDB db;
    try {
        db = RocksDB.open(options, "/tmp/MiscTest");
    } catch (RocksDBException e) {
        throw new RuntimeException("open failed, should not happen.");
    }

    String prefix1 = "alice";
    String prefix2 = "ben";
    String prefix3 = "charlie";

    Serializer<String> serializer = new StringSerializer();
    Deserializer<String> deserializer = new StringDeserializer();

    try {
        for (int i = 0; i < 1000; i++) {
            String rand = generateRandomString();
            db.put(wOptions, serializer.serialize("t", prefix1 + rand), serializer.serialize("t", rand));
            db.put(wOptions, serializer.serialize("t", prefix2 + rand), serializer.serialize("t", rand));
            db.put(wOptions, serializer.serialize("t", prefix3 + rand), serializer.serialize("t", rand));
```

```
        }
    } catch (RocksDBException e) {
        throw new RuntimeException("put failed.");
    }

    RocksIterator iter = db.newIterator();

    for (String prefix : Arrays.asList(prefix1, prefix2, prefix3)) {
        iter.seek(serializer.serialize("t", prefix));

        System.out.println("Start : " + deserializer.deserialize("t", iter.key()));

        int count = 0;
        while(iter.isValid()) {
            String key = deserializer.deserialize("t", iter.key());

            if (key.startsWith(prefix)) {
                count++;
                iter.next();
            } else {
                System.out.println("End : " + key);
                break;
            }
        }

        System.out.println("Prefix " + prefix + " : " + count);
    }

    db.close();
}
```

And one output is:

```
Start : alice12i1TJStDI
End : ben12i1TJStDI
Prefix alice : 1000
Start : ben12i1TJStDI
End : charlie12i1TJStDI
Prefix ben : 1000
Start : charlie12i1TJStDI
Prefix charlie : 1000
```

Pros: easy to implement.

Cons: may be sub-optimal in performance (see below).

Optimized approach: use RocksDB prefix seeking

RocksDB supports prefix seeking which can be treated as an optimization over option0 above with the usage of bloom filters and hashing underneath. It fits perfectly in our case, where the seeking is done by a prefix, and the iterator is placed at the first item whose key prefix matches. Background about RocksDB prefix seeking can be found here, also a very short slide-deck on example code.

Note that there is one caveat: this feature is not available in RocksDB JNI yet, I have filed a RocksDB issue (ticket), and if necessary we can contribute back to RocksDB for this feature.

Update: prefix hashing has been added to JNI: https://github.com/facebook/rocksdb/pull/1109

Pros: leverage RocksDB internal feature, which is supposed to be performant.

Cons: need to modify RocksDB code, and relying on a very new version of RocksDB to have this in JNI.