KIP-114: KTable state stores and improved semantics

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: here

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

Motivation

Kafka Streams has grown to augment KTables (e.g., changelogs) with state stores. These state stores can be seen as a way to materialize the KTables. State stores can be queried through the Interactive Queries (IQ for short) APIs, as introduced in KIP-67. This led to some confusions and inconsistencies, e.g., :

- Several KTables must be materialized, e.g., those that are created as a result of aggregation (because the aggregated data must be stored somewhere). We force users to provide a state store name because of this requirement.
- In some cases, it is not a must that the KTables are materialized, e.g., in KStreamBuilder.table().
- In some other cases, the KTables are never materialized, because the user does not have the option of materializing them, for example KTables resulting from operations like filter(). Thus user unfortunately cannot query these KTables through interactive queries. For example, the second KTable below cannot be materialized:

```
KTable table1 = builder.table(..., topic, stateStoreName); // materialized

KTable table2 = table1.filter(...); // not materialized, and not queryable by IQ
```

- KTables resulting from join() operators are not materialized, because the user does not have the option of materializing them. Thus they cannot be queried.

- Certain operations on KTables, such as print() and foreach(), are often confusing and redundant, since users can already print the values of a materialized view. Furthermore, users also have the option to use the KStream equivalent functions after converting a KTable to a KStream. So they have 3 ways of printing and iterating through a KTable.

Proposal

We propose to clean up the KTable API and make the KTable semantics clearer and consistent through API improvements and associated JavaDoc improvements.

In a nutshell the approach is as follows:

- Decouple the notion of materialization from the notion of querying. Materialization is an internal streams decision. Querying is a user-facing decision.
- We will overload each method that creates a KTable with a store name. If the user provides that name, the user can subsequently query that store with that provided name. If the user does not provide a name, or provides a null name, the user will not be able to query the KTable's stores.
- The above guarantee says that a user can query the store. It doesn't say anything about how we implement that feature internally. We could be materializing the store (e.g., backing it with a RocksDb store). Or we could be providing a read-only view of the store (e.g., by computing the result on the fly). The filter example above illustrates the two options:

KTable table2 = table1.filter() <----- user does not provide a name, no guarantee table2 is queryable

KTable table2 = table1.filter("filterStoreName")   <----- user provides a name, we guarantee table2 is queryable based on that name. Internally we could be writing each filtered value to a RocksDb store, or computing the filter result on the fly each time the store is queried.

What is in scope

The main scope of this KIP is to address the inconsistency in which KTables can be queried and which KTables cannot. As well as how a user goes about making that decision. As such, this KIP should be seen as an incremental update to the existing APIs, not a complete overhaul.

What is in scope is the exact API for addressing the above inconsistency.

What is not in scope

- Revisiting the interactive queries APIs is not in scope. Specifically, what is not in scope is re-defining the exact boundary between the DSL (i.e., the processing language) and the storage/interactive queries, and how we jump from one to the other. The boundary will remain as it is today, where to do Interactive Queries, the user needs a store name and receives a store to query based on that name. We can address that in a later KIP if required.

- What is not in scope is rethinking the DSL itself. Specifically, specifying state stores in the API can be thought of as a type of hint to the DSL to indicate that materialization is required. There could be many such hints, and perhaps they could be described with methods such as .materialize(), or .cache(), or .log(). These methods might be getting us towards a less declarative API. Either way, it is not in the scope of this KIP to undertake a complete rethink of the DSL. This KIP stays consistent with the DSL we currently have.

Interfaces / API Changes (DSL only)

3 overloads for all APIs that create KTables: with and without store name. The alternative to store name is a StateStoreSupplier.

All API that create KTables will have 3 overloaded methods, one with the store name or with a StateStoreSupplier, and one without. Note that providing a null store name is the same as using the API with no store name.

These APIs include the ones below and any of their existing overloads. We do not list the overloads here to keep the list uncluttered. Each API will have one version with no store name, and one version with a store name and one version with a StateStoreSupplier (that contains the state store name).

In KTable.java overload each of the following APIs by adding store name and StateStoreSupplier (when they don't exist already):

- `KTable<K, V> filter(final Predicate<? super K, ? super V> predicate);`
- `KTable<K, V> filterNot(final Predicate<? super K, ? super V> predicate);`
- `<VR> KTable<K, VR> mapValues(final ValueMapper<? super V, ? extends VR> mapper);`
- `KTable<K, V> through(final String topic)`
- `<VO, VR> KTable<K, VR> join(final KTable<K, VO> other, final ValueJoiner<? super V, ? super VO, ? extends VR> joiner);`
- `<VO, VR> KTable<K, VR> leftJoin(final KTable<K, VO> other,  final ValueJoiner<? super V, ? super VO, ? extends VR> joiner);`
- `<VO, VR> KTable<K, VR> outerJoin(final KTable<K, VO> other, final ValueJoiner<? super V, ? super VO, ? extends VR> joiner);`

In KStreamBuilder.java overload each of the following APIs by adding store name and StateStoreSupplier (when they don't exist already):

- public <K, V> KTable<K, V> table(final String topic);
- public <K, V> GlobalKTable<K, V> globalTable(final String topic)
- public <K, V> GlobalKTable<K, V> globalTable(final Serde<K> keySerde, final Serde<V> valSerde, final String topic)

In KGroupedTable.java overload each of the following APIs by adding store name and StateStoreSupplier (when they don't exist already):

- `KTable<K, Long> count(final String storeName);`
- `KTable<K, V> reduce(final Reducer<V> adder, final Reducer<V> subtractor);`
- `<VR> KTable<K, VR> aggregate(final Initializer<VR> initializer, final Aggregator<? super K, ? super V, VR> adder, final Aggregator<? super K, ? super V, VR> subtractor)`

In KGroupedStream.java overload each of the following APIs by adding store name and StateStoreSupplier (when they don't exist already):

- `KTable<K, Long> count(final String storeName);`
- `KTable<K, V> reduce(final Reducer<V> reducer);`
- `<VR> KTable<K, VR> aggregate(final Initializer<VR> initializer, final Aggregator<? super K, ? super V, VR> aggregator, final Serde<VR> aggValueSerde);`

Remove unnecessary methods

- Depreciate the following KTable methods: print(), writeAsText(), foreach() and any of their overloads.

Interfaces / API Changes (PAPI only)

During implementation it became clear that a minor adjustment was needed and org.apache.kafka.streams.processor.addGlobalStore should take a

- `"final StateStoreSupplier<KeyValueStore>" type as the first argument, not a StateStore. Note that one can obtain a StateStore by calling the .get() method on a StateStoreSupplier.`

Misc API cleanup

During implementation it became apparent that some APIs needed minor renaming or cleanup. That is listed here:

- KTable.getStoreName() -> Ktable.queryableStoreName(). Reason is that in Kafka we don't usually use the "get" prefix.

Implementation plan

One implementation detail that is important is how the Kafka Streams internals decides whether to materialize a KTable. Note that the above APIs provide a way for Interactive Queries to query a state store. They do not dictate whether the state store itself if a real, materialized one, or a view on top of a real, materialized store. Going back to the first example in the motivation:

```
KTable table1 = builder.table(..., topic, stateStoreName); // materialized
```

```
KTable table2 = table1.filter(final String stateStoreName2)
```

The store with name "stateStoreName2" could be a view on top of the store with name "stateStoreName", in which case each time we query stateStoreName2, we would, on the fly, compute the result of the filter on values actually stored in the first store. Alternatively, "stateStoreName2" could in itself be materialised, i.e., contain all the results obtained from the filtering. Materialising all Ktables with a state store name could be expensive, however it is a straightforward to implement and could be a good V1. A subsequent JIRA could do an optimization in V2.

Compatibility, Deprecation, and Migration Plan

- Will depreciate the KTable methods print(), writeAsText(), foreach() and any of their overloads.
- Code will be backwards compatible.
- PAPI code that uses org.apache.kafka.streams.processor.addGlobalStore will need to use new API signature.

Rejected Alternatives

Change the name of KTable. We discussed potentially changing the name of a KTable to something like KChangelogStream, but this doesn’t solve the main problem this KIP addresses, which is API inconsistency.

Use .materialize(String storeName) instead of overloads. In this proposal, we remove all state store parameters in KTable methods, and add a .materialize() method on the KTable. This gets us to a place where we are forced to make the DSL less declarative (see “What is not in scope” above) and forces us to have an orthogonal discussion to this KIP.

Have the KTable be the materialized view. In this alternative, the KTable would no longer be a changelog stream, it would be a materialized view. So we would collapse two things (the existing KTable and state store) into one abstraction. All KTable methods would need to take a state store name.

- There are some performance implications of doing this, e.g., each KTable would now always be materialized and that is expensive. However, we could consider mitigating the performance penalty by introducing “virtual” stores, where the data is not written to a topic, but is computed on the fly. For example, the KTable obtained from filter() would not have a changelog topic, but could still have a virtual store queryable through Interactive Queries.
- It is not clear that collapsing 2 abstractions helps. In particular, a KTable models a changelog. That itself is a useful abstraction. A state store is a materialized view. That’s a distinct abstraction with parallels in the database world.