Discussion: Expose State Store Names in DSL (0.10.0)

Background and Problem

Summary of today's (0.10.0.0) state stores and their name definitions.

State Store Names

In the lower-level API layer, users need to specify a name for any state stores they created. The state store is used in two ways:

1. For persistent stores (RocksDB), the local directory for any specific state store is defined as "[state.dir]/[application.id]/[task.id]/rocksdb/[store.name]", where "state.dir" is default to "/tmp/kafka-streams" (in CP as of 3.0.0: "/var/lib/kafka-streams"), and task.id is in the format of "integer_integer", where the first integer refers to "topic group id", and the second integer refers to "partition id".
2. For the store's changelog topic name, it is defined as "application.id-store.name-changelog". This is to differentiate different applications using the same state store names against a shared Kafka cluster.

Both of these two aspects are treated as internal implementation details and abstracted from users for now.

State Store in Streams DSL

In the higher-level DSL layer, we abstract further the state store names from user as well. More concretely, for certain operators the library creates the following state stores automatically which are hidden from the users:

1. When a KStream is involved in windowed joins, we require users to define a Windows spec with a name (i.e. we document it in JavaDocs as the "window name") and materialize the KStream into a state store, with store name as "window.name-this" or "window.name-other", depending if the KStream is the inner or outer relation.

2. When a KTable is involved in a stateful operation, such as joining with another KStream or KTable, or aggregation, its "source" KTable is materialized with the store name as the same as the source topic. **NOTE** that this means KTable materialization is lazy, and always on the source KTable.

   For example, in the above dummy topology:

   ```
   KTable table = builder.table("topic1");    // this table will not be materialized.
   table.to("topic2");
   ```

   Since "table" is not involved in any stateful operations throughout her life time, it will not be materialized at all. While for this example:

   ```
   KTable table1 = builder.table("topic1");   // table1 will be materialized with store name "topic1"
   KTable table2 = table1.filter(..);
   KTable table3 = table2.mapValues(..);
   KStream table4 = stream1.join(table3);
   ```

   In this case, since "table3" is involved in a stateful operation, we will materialize its source "table1", with the state store named "topic1". Then upon each incoming record from "stream1", we will logically query "table3" to find matching record by query the state store "topic1", and passing the returned value through the filter and mapValues operators before applying the join operator.

3. When an aggregation operation (either a KStream windowed / non-windowed aggregation, or a KTable aggregation) creates a new KTable, the newly created KTable will be automatically materialized since the aggregation is implemented as keeping a running result that keep getting updates. For KStream windowed aggregation, we use the window name as the state store name; for others we require from users to provide a "KTable name" for all these aggregation operators for the resulted KTable, which is then used as the state store name. To illustrate this, let's see a more complex example:

```
KTable table1 = builder.table("topic1");   // table1 will be materialized with store name "topic1"
KTable table2 = table1.filter(..);
KTable table3 = table2.mapValues(..);
KTable table4 = table3.groupBy(..).aggregate(.., "table4");       // table4 is materialized with store
name "table4"
KTable table5 = stream1.aggregateByKey(Windows("window1"), ..);   // table5 is materialized with store
name "window1"
table5.to("topic2");
```

In the above example, again "table1" is materialized with state store "topic1", and upon each incoming record from "topic1", by updating the state store we will generate a pair <old-value, new-value>. The pair will be passed through the filter and mapValue operators, and then applied with the aggregate function separated specified for "table3" to update the state store "table4".

In summary, at the DSL layer we tried to abstract the state stores as well as its names from the users as conceptual "KTable names" or "Window names". However, we still need to educate users to change these name if they change part of their topologies related to these operators but not the whole topology hence they can reuse the application.id, hence we realize it actually makes less sense to abstract these state store names. Plus, we are considering two future works which will require us to expose the state stores even from the Streams DSL layer:

1. Queryable State: users need to know which state stores are created, and their corresponding names in order to query them.
2. Reuse Changelog: because we are abstracting the state stores and their names, the changelog topics are also abstracted from users as "internal topics". This resulted in unnecessary duplicate logs in Kafka, for example in the above code, the Kafka topic "topic2" and the internal changelog topic "application.id-window1-changelog" are actually storing exactly the same data.

   NOTE not all materialized state stores will be associated with a changelog topic: for example the state store "topic1" above does not need to create a separate changelog as we can already treat the source topic "topic1" as its changelog.

Proposal

We are considering to expose the auto-created state store names in the Streams DSL, and also expose the changelog topics for those state stores (i.e. expose the principle of creating the topic name as "application.id-store.name-changelog" to users) in both Processor and Streams DSL layers. With both of these two changes, users will have a complete ideas about which state stores is in this topology and their corresponding changelogs. This would be a major API change that touches many functions of KTable and KStream.

The following proposal for API change is only to kick off some discussions:

1. Remove the window name from Windows class.
2. Always enforce KTable materialization upon creation (i.e. even for "builder.table(..).to()" we also materialize it into a state store; this will indeed incur unnecessary overhead but should be very rare, and it helps for consistency of materialization).
3. Require a String typed "state store name" for all aggregation operations that creates a KTable, as well as stream-stream join operations:
   a. There are two cases that a new KTable is created: any aggregation operations, and builder.table(). For both cases a state store name is explicitly required.
   b. For windowed stream-stream join where both streams need to be materialized, explicitly require users to specify the state store names for both.
   c. Document that the corresponding changelog topic name is "application.id-store.name-changelog".
4. [Optional] We can also allow users to override the KTable's corresponding state store changelog names for the former case for user-friendly topics naming, but reminding them that this is universal and hence may be conflicting with other applications.

With the above changes we are effectively exposing the state stores as well as their changelog topic names in the Streams DSL. **NOTE** this means that we cannot change such implementations moving forward without breaking backward compatibility. For example, for some aggregations we thought about not creating the running results store, but buffer the raw input data in the store and apply aggregations periodically; even if we do this kind of optimizations later, we should not remove the running results store as it is "exposed" to users.

This proposal is implemented in https://github.com/apache/kafka/pull/1526.