Discussion: Serialization and Deserialization Options

- Serialization Requirements
- Strong Typing in Streams DSL
- Options w. Pros / Cons
  - Current Option
  - Alternative Option
- About Open World v.s. Closed World
  - Open World APIs
  - Closed World APIs
  - Typing Needs in Practice
- Summary

There are some discussions about which serde models to use before the tech preview release, and after the release the C3 team provided some feedbacks about using the current serde models, and possible ways to improve it. This page serves as a summary of our past thoughts and discussions about pros / cons of different options for clear illustration.

Serialization Requirements

In Kafka Streams we have two scenarios where we need to materialize data: Kafka and persistent state stores. We need to IO with Kafka when:

- Creating a source stream from Kafka (deser).
- Sinking a stream to Kafka (ser).
- Using a logged state store (ser / deser for Kafka changelog)

And we need serialize / deserialize data when IOing with state stores (RocksDB); in other words, for stateless operations we will never need to materialize any data: each record will just "pass through" the operator to the downstream operators; we only need state stores for stateful operations like joins and aggregations.

For any of these case, we would need a serde for key and value separately.

Strong Typing in Streams DSL

We require strong typing in the Kafka Streams DSL, and users need to provide the corresponding serde for the data types when it is needed. More specifically, for example:

```
<K1, V1, T> KTable<K1, T> aggregate(Initializer<T> initializer,
                                    Aggregator<K1, V1, T> adder,
                                    Aggregator<K1, V1, T> substractor,
                                    KeyValueMapper<K, V, KeyValue<K1, V1>> selector,
                                    Serde<K1> keySerde,
                                    Serde<V1> valueSerde,
                                    Serde<T> aggValueSerde,
                                    String name);
```

This is the KTable aggregation APIs, where a original **KTable<K, V>** can be aggregated (using initializer / adder / substractor) by a selected key with type **K1** and value to be aggregated with type **V1** (via selector), and the aggregated value has type **T**. The resulted stream is a **KTable<K1, T>**. It will be translated into the following connected operations:

- A **select operator** that selects the aggregate key and value-to-be-aggregate as KeyValue<K1, V1>.
- A **sink operator** following the select operator to materialize the extracted KeyValue<K1, V1> into an internal topic, this is for repartitioning by the selected key.
- A **source operator** reading from this internal topic.
- An **aggregate operator** following the source operator and associated with a RockDB store<K1, T> for computing and keeping the current aggregated values.

Here we require three serdes:

- keySerde<K1> is used to materialize the selected key for possible repartitioning through Kafka, and also for storing the current aggregated values as key-value pairs in RocksDB.
- valueSerde<V1> is used to materialize the value-to-be-aggregated for possible repartitioning through Kafka.
- aggValueSerde<T> is used to store the current aggregated values as key-value paris in RocksDB.

In addition, there is a "String name" parameter, which is used as the name of the resulted **KTable<K1, V>**. Each KTable needs a name (for a source KTable, its name is the topic this stream is read from via buider.table(..)) because we need to keep a changelog Kafka topic this KTable when materializing it in RocksDB, and the topic name needs to be preserved across multiple runs for restoration, thus users need to provide this name and remember to reuse it when re-processing with a possibly modified topology.

Options w. Pros / Cons

We start with the current model for serialization, and present others that we have considered.

Current Option

As illustrated above, we require one or more serde classes corresponding to the data types on each stateful operator. Users can specify a default serde via configs, and we overload each of these stateful operators without the serde, in which case the default serde will be used; if the type does not match only a runtime ClassCastException will be thrown upon first record. Here are the known issues / observations for this approach:

1. We observed that in practice (from C3 team), although we provide overloaded function without serde, people usually just specify the serdes in each step for safety.
2. For common usage like Avro / JSON specific binding, users today need to wrap a specific serde class for each specific record class; arguably we can mitigate this problem by using some factory-like syntactic sugar like "AvroSpecificSerdeFactory.forType(Class<T>)" into the Avro serde helper class.
3. We know that due to Java type erasure, we cannot perfectly inference types from operations itself; and instead of propose a imperfect approach where "you may not need to give a serde info for some cases, and you may need to in some others", we decided to not go into that direction and bite the bullet of enforcing users to think about serdes on each stateful stage.

This option is similar to Spark Java API.

Alternative Option

We have once discussed and rejected another option, as to "register serdes for each class" at the top of the program via code, and only enforcing users to provide class object during the stateful operation. More specifically, users can still provide a default serde through configs, and they can "override" the serde for specific classes before defining the topology in the code, as:

```
builder.register(String.class, new StringSerde());     // primitive types


builder.register(myListType, new MyListSerde());       // parameterized types, talked later

builder.register(Profile.class, new AvroSpecificSerde());  // Avro specific record types
```

For Avro / JSON specific record types, in order to use a general serde class for all types we need to overload the serialization interfaces as the following so we can use the class object to wrap the output data:

```
Deserializer<K> {

    K deserialize(String topic, byte[] bytes);   // this is the existing API

    K1 deserialize(String topic, byte[] bytes, Class<K1> clazz);    // the added API
}
```

And for parameterized types, we use a specialized type class (similar ideas can be found in JSON libs as well: class)

```
Type myListType = type(List.class, String.class);  // List<String>, where Type is extending Class
```

And then in the stateful operation, we do not need to provide the serde classes, but just the class objects instead:

```
KTable<String, Raw> table1 = builder.table("topic", String.class, Raw.class);

KTable<String, ProfileByRegion> table2 = table1.aggregate(..., String.class, Profile.class, ProfileByRegion.
class);
```

The rationale is that for common practice where we use Avro / JSON throughout the topology, we only need to register a few serdes (for example, using Kryo as the default ones in config, and override all the AvroSpecificRecords for AvroSpecificSerde). This option is used in Flink Java API.

Here are the known issues / observations for this approach:

1. Although it has the benefits of "only specifying the serde classes at the top instead of throughout the topology", it may have the risk for complex topologies, users tend to forget and mess with the class serde mapping; i.e. this is a question of "whether it is worthwhile to just enforce it throughout the code for safety".
2. Asking users to learn a new "Type" class for parameterized types could be a programming burden.
3. Not clear if extending the serialization interface with this new overloaded function would be clean from any misuse side effects.

About Open World v.s. Closed World

Open World APIs

Current Kafka Streams high-level DSL is considered "Open World", where arguments of the API functions are generic typed with the provided (de-) serializers to read / write their objects into byte arrays.

The serdes are provided when we 1) read data from Kafka, 2) write data to Kafka, 3) materialize data to some persistent storage (like RocksDB), and it is always provided dynamically in the DSL.

Note that we will only materialize KTables that are read directly from Kafka topic, i.e.:

```
// we will materialize the table read from "topic", and apply the filter / mapValues logic on
// that materialized data before the aggregation processor below

builder.table("topic", keyDeserializer, valueDeserializer).filter(...).mapValues(...).aggregate()
```

We do not need to consider case 3) for KTables, but for KStream we still need to require users to provide the serde libraries for KStream when it is going to be materialized.

Its pros are:

1) Better programmability: users just need to use whatever class objects / primitives they want.

2) Type-safty: it includes compilation-time type checking.

3) Flexibility to extend to other programming interfaces in the future.

Its cons are:

1) Users have to define a class for each intermediate results throughout the topology, and also a ser-de factory for any of them that are going to be written to disk files / sockets (i.e. Kafka or RocksDB).

2) There will be some overhead in creating the transient objects at each stage of the topology (i.e. objects that are created between the consecutive transformation stages).

Closed World APIs

Another popular API category is "Closed World", where the data types are pre-defined as a compound class like JSON (e.g. Storm uses "Tuple", Spark uses "DataFrame", Hyracks uses "Frame", etc), which are essentially a map representation of the underlying byte buffer, and getters / setters are then by field-name and translated into byte-buffer reads / writes.

Its pros are:

1) Users do not need to define these classes and their serdes specifically for each of these classes.

2) Possibly reduce GC on "transient" JVM objects CPU overheads of ser-de / hashing if the library can mutate the incoming record bytes.

Its cons are:

1) Lost of type-safty, since it is always translating from a "Tuple" to another "Tuple".

2) Users have to define the "schema" of the transformed tuple dynamically that could be possibly registered in a schema registry service.

Examples of Closed-World APIs include: Storm, Spark, Hyracks, Giraph, etc.

Typing Needs in Practice

I think that for most stream processing applications, users would likely fall into one of the following categories:

1. Their data types are pure strings with some JSON format throughout the pipeline. Cases fall into this category are that a) stream source data comes from some files / logs from other systems whose format they cannot control, b) they use string just for ease of programming that involves other external systems.

2. Their data are encoded with Avro / Thrift / ProtocolBuffer throughout the pipeline. Cases fall into this category are that a) they have a centralized schema management for the whole organization that everyone depends on, b) their stream source data comes from some online services that generating messages at their own format.

3. Their source data are encoded differently for each data source, or all their data types are primitive. Cases fall into this category are that a) streaming source data are quite heterogeneous with their own SerDe factories and they do not have a centralized data schema support, b) just for demos?

4. They want strong typing in their application for better programmability in terms of functional programming / oo programming, and hence would rather want to define their own class types for all possible intermediate values rather than using a generic class as in case 1) or 2).

For the above three cases, type-safty will be mostly helpful for case 3) and 4), and possibly also in case 1) and 2) for strong typing as well (think of SpecificRecord in Avro).

Summary

In summary, many of these pros / cons considerata are really programmability / user-facing issues that personal taste may play a big role here. And hence we thought that whichever option we chose, we will not be able to make everyone happy (not sure if it is a half-half split though). So before we collect more feedbacks that brings in factors that we have not thought about before, we would like to keep the situation as-is and work on improving the current options for better programmability, but keep this in mind so that we can always come back and revisit this decision before we remove the "API unstable" tag for Kafka Streams.