Kafka Streams Internal Data Management

Table of Contents

Overview

Kafka Streams allows for stateful stream processing, i.e. operators that have an internal state. This internal state is managed in so-called **state stores**. A state store can be ephemeral (lost on failure) or fault-tolerant (restored after the failure). The default implementation used by Kafka Streams DSL is a fault-tolerant state store using 1. an internally created and compacted changelog topic (for fault-tolerance) and 2. one (or multiple) RocksDB instances (for cached key-value lookups). Thus, in case of starting/stopping applications and rewinding/reprocessing, this internal data needs to get managed correctly.

Current State

We first want to give an overview about the current implementation details of Kafka Streams with regard to (internally) created topics and the usage of RocksDB. We can categorize available transformations for `KStream` and `KTable` as shown below. All operators within a category use the same internal state management mechanism. Therefore, we get an overview of the state management strategy for each transformation.

- **single stream tuple-by-tuple**: those transformations do not use an internal state — however, if they change the key ("new key") data re-partitioning might be required after the transformation (i.e., writing to and reading from a topic KAFKA-3561 - Getting issue details... STATUS KAFKA-3576 - Getting issue details... STATUS)
- **aggregation and joins**: those transformations do use internal state (RocksDB plus changelog topic)
- operations marked with "+ state" allows the usage of user defined state

State management details are given below.

KStream API

`KStream` currently offers the following methods which do have different implication with regard to (internally) created topics and RocksDB usage.

single stream					**multiple streams**	
data transformation				**other**	**data transformation**	
tuple-by-tuple (KStream -> KStream)		**aggregation**			**tuple-by-tuple (i.e., joins)**	
same key	**new key**	**non-windowed (KStream -> KTable)**	**windowed (KTable -> KTable<<W,K> V>**		**non-windowed KStream-KTable**	**windowed KStream-KStream**
filter (1:[0,1])		aggregateByKey	aggregateByKey	print	join (as of 0.10.2)	join
filterNot (1:[0,1])		reduceByKey	reduceByKey	~~writeAsText~~ (deprecated as of 1.0)	leftJoin	leftJoin
	selectKey (1:1)	countByKey	countByKey	foreach		outerJoin
mapValues (1:1)	map (1:1)			through	merge (as of 1.0)	
flatMapValues (1:n)	flatMap (1:n)			to		

transformValues (1:1 + state)	transform (1:n + state)			branch		
	process (1:0 + state)			peek (as of 0.11.0)		

KTable API

`KTable` currently offers the following methods which do have different implication with regard to (internally) created topics and RocksDB usage. KAFKA-3576 - Getting issue details... STATUS

single stream				**multiple streams**
data transformation			**other**	**data transformation**
tuple-by-tuple (KTable -> KTable/KGroupedTable)		**aggregation** **(KGroupedTable -> KTable)**		**tuple-by-tuple** **(KTable-KTable joins)**
same key **(-> KTable)**	**new key** **(-> KGroupedTable)**			
filter (1:[0,1])		aggregate	~~print (deprecated as of 1.0)~~	join
filterNot (1:[0,1])		reduce	~~writeAsText (deprecated as of 1.0)~~	leftJoin
mapValues (1:1)		count	~~foreach (deprecated as of 1.0)~~	outerJoin
	groupBy (1:1) [internally simple map]		~~through (deprecated as of 1.0)~~	
			~~to (deprecated as of 1.0)~~	
			toStream	

Data Management

Overview

There are four methods to *explicitly* deal with user topics:

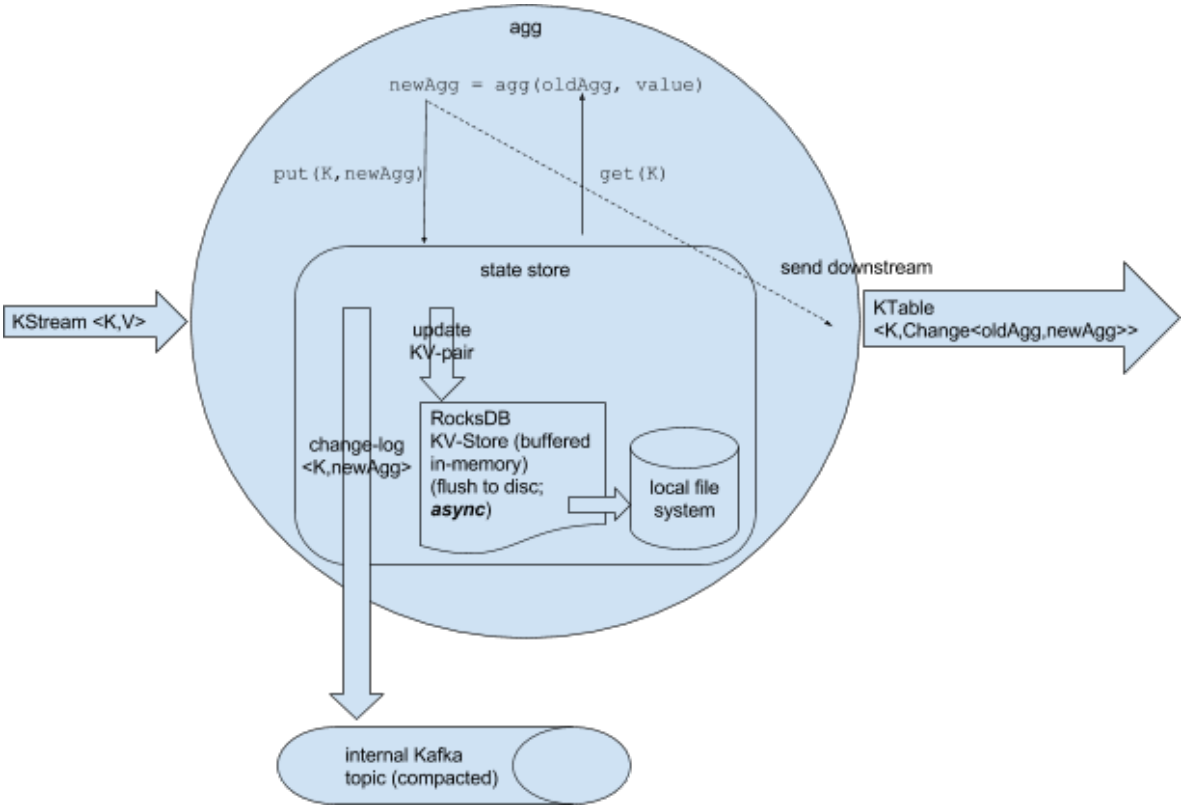
1. `KStreamBuilder#stream()` for consuming
2. `KStreamBuilder#table()` for consuming
3. `KStream/KTable#to()` for writing
4. `KStream/KTable#through()` for writing and reading again.

User topics are required to be created by the user before Kafka Streams application is started. Furthermore, an *interal topic* is created each time re-partitioning is required (via an ingested `through()` KAFKA-3561 - Getting issue details... STATUS); this happens when the key is changed before an aggregation is performed (not necessarily directly after each other):

```
// Pseudo code
KStream source = builder.stream(...)
source.someTransformation().newKey().someTransformation().someAgg(); // newKey() refers to any transformation
that changes the key
```

Here's an illustration of the above pseudo-code topology:

This implies that after “new key” there was no `to()`/`through()` performed. The aggregation itself uses a **RocksDB** instance as key-value **state store** that also persists to local disk. Flushing to disk happens *asynchronously*. Furthermore, an **internal compacted changelog topic** is created. The state store sends changes to the changelog topic in a batch, either when a default batch size has been reached or when the commit interval (see "Commits" below) is reached.



RocksDB is just used as an internal lookup table (that is able to flush to disk if the state does not fit into memory KAFKA-3776 - Getting issue details... STATUS) and the internal changelog topic is created for fault-tolerance reasons. Thus, the changelog topic is the source of truth for the state (= the log of the state), while RocksDB is used as (non-fault tolerant) cache. RocksDB cannot be used for fault-tolerance because flushing happens to local disk, and it cannot be controlled when flushing happens. RocksDB flushing is only required because state could be larger than available main-memory. Thus, the internal changelog topic is used for fault-tolerance: If a task crashes and get restarted on different machine, this internal changelog topic is used to recover the state store. Currently, the default replication factor of internal topics is 1.

There are two main differences between non-windowed and windowed aggregation with regard to key-design. For window aggregation the key is **<K,W>**, i.e., for each window a new key is used. ~~Thus, the memory usage grows over time (~~ KAFKA-4015 - Getting issue details... STATUS ~~), even if the key-space is bounded (i.e., the number of unique keys). This implies that log-compaction cannot purge any old data.~~ The second difference is about RocksDB instances: instead of using a single instance, Streams uses multiple instances (called “segments”) for different time periods. After the window retention time has passed old segments can be dropped. Thus, RocksDB memory requirement does not grow infinitely ~~(in contrast to changelog topic)~~. (KAFKA-4015 was fixed in 0.10.1 release, and windowed changelog topics don't grow unbounded as they apply an additional retention time parameter).

For **KTable** a similar behavior applies. Using *groupBy().someAgg()* results in **internal topic** and **RocksDB** creation.

For **stateful** KStream transformation (*transform*, *transformValue*, and *process*) an **explicit state store** is used. Depending on the use state store, a changelog topic might get created.

For **joins**, one or two internal **state stores (RocksDB plus internal changelog topic)** are used. Behavior is same as for aggregates. Joins can also be windowed (see window aggregates).

Related Work to state stores: KAFKA-3909 - Getting issue details... STATUS

Commits

Kafka Streams commit the current processing progress in regular intervals (parameter *commit.interval.ms*). If a commit is triggered, all state stores need to flush data to disk, i.e., all internal topics needs to get flushed to Kafka. Furthermore, all user topics get flushed, too. Finally, all current topic offsets are committed to Kafka. In case of failure and restart, the application can resume processing from its last commit point (providing at-least-once processing guarantees).

Internal Topics and State Store Names

Currently in the Streams DSL we are trying to abstract the auto generated internal topics and state store names as "KTable names" and "window names"; however, in future release all state store name will be exposed to the user. KAFKA-3870 - Getting issue details... STATUS Internal topics follow the naming convention `<application.id>-<operatorName>-<suffix>;` this naming convention might change any time in.