Kafka Streams Discussions

This page summarizes our past feature proposals and discussions in Kafka Streams. Promoted ideas will be proposed as KIPs.

Sub-pages:

Public API Improvements

Currently, the public API of Kafka Streams is not perfect. This is a summary of knows issues, and we want to collect user feedback to improve the API.

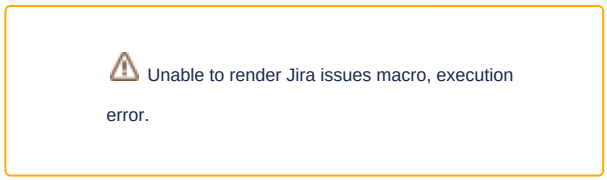
Issue	User Impact / Importance	Possible Solution	Solution User Impact
KTable API and store materialization improvements	Currently, KTable API offers some methods that confuse uses, because users think in terms of table instead of a changelog stream. Also, not all KTables are materialized and the users might want to control (ie, force) when a KTable should be materialized (for example, to allow for querying the store using interactive queries). *Importance: high*	KIP-114	medium to high • public API changes for KTable
TopologyBuilder and KStreamBuilder • leak internal methods • no clean separation of abstractions	Might be hard for users to understand concept. User might be confused by verbose API (and leaking methods) they should never see. *Importance: high*	KIP-120	medium • need to use different imports • change pattern to create topology with KStreamBuilder

Too many overloads for method of KStreamBuilder, KStream, KGroupedStream, KTable, and KGroupedTable	Many methods have more than 6 overloads and it's hard for the users to understand which one to use. Furthermore, with the verbose generics, compiler errors might be confusing and not helpful if a parameter is specified wrong (ie, I want to use overload X, does the compiler pick the correct overload? and if yes, which parameter did I get wrong? and if no, which parameter do I need to change so the compiler picks the correct overload?) As we add more feature, this is getting more severe. *Importance: high*	Change to Builder Pattern	high • need to rewrite large parts of their code
Non consistent overloads	Some APIs have non-consistent overloaded methods that might be confusing to the user (why do I need to specify this for overload A, but not for overload B? – why does overload X allow me to do this, but not overload Y) Example: • for `KGroupedStream#aggregate` required to provide argument aggValueSerde • for `KGroupedTable#aggregate` has an overload without `aggValueSerde` *Importance: medium*	Relates to "Too many overloads" – could be resolved with a clean builder abstraction.	medium • user might need to rewrite parts of the code if we deprecate some confusing overloads • user code might get cleaner
DSL limits access to records and/or record metadata	Some interfaces like `ValueJoiner` only provide the values of both records to be joined, but user might want to read the key, too. For adding the key, we loose the guarantee, that the key is not modified though. (There are more similar examples, where the key is not accessible.) Record metadata (like offset, timestamp, partition, topic) is not accessible in DSL interfaces. *Importance: low*	Change interfaces, RichFunctions, Use process /transform	low • this is more about improving the API and/or adding new features
Missing public API	Some very helpful classes, that are currently in package `internal` could get added to public API. For example, windows and some serde classes. *Importance: low*	Move classes to different package.	low • we only add new stuff
Window(s) API	• get rid of minimum retention time (that is a performance improvement that confuses many users). • remove some leaking internal APIs *Importance: low*		low
Improve Streams Config API	API is verbose and with intermixed consumer and producer configs hard to use correctly. *Importance: low*	Builder pattern	medium • users need to rewrite the config code

ProcessorContext to verbose	ProcessorContext give access to method that cannot be called. This is hard to reason about for users. *Importance: low*	Split ProcessorContext and extract RecordContext	low • most user are expected to use mainly DSL
low-level API integration into DSL	Currently, low-level API is integrated into DSL via process()/transform() and transformValues(). Those abstraction are not perfectly defined and confusing to users. *Importance: medium*	Major redesign	medium • most user are expected to use mainly DSL
Low-level API in DSL vs. "advanced DSL"	Currently, low-level API is used to empower the user to do anything within DSL. This approach is questionable to some extents. For example, if a user wants to do a stateful 1:1 transformation of records, she must implement `Transformer` interface, thus has a lot of boiler plate code to access the actual state via the context and needs to implement non related methods like punctuate(). A DSL method like `statefulMap` with interface `#map(K key, V Value, S state)` might be easier to use. The question is, if DSL can provide more DSL like methods to allow more advance computations without forcing the user to too low-level. *Importance: medium*	Major redesign	medium • it's about adding new method so existing code should not be affected
potentially non-partitioned input for stateful DSL operations	`process()`, `transform()`, and `transformValue()` all accept a stat. In order to allow for scaling, state is usually partitioned by key. However, Streams does not enforce a correct partitioning (via a call to `groupByKey`) and thus, data might not be partitioned correctly for those three operators. The use need to be aware of this, and do a manual call to `through()` right now to ensure correct partitioning. *Importance: medium*	Educate users about this issues in the docs explicitly (if users go with low level operators, they also have to take more responsibility by themselves to get it right) or allow .process() /transform() /transformValues() (that do have a state) only on `KGroupedStream`	medium • most user are expected to use mainly DSL
Simplify "message callback" use case	From mailing list: 2. For the processor api, I think this api is mostly not for end users. However this are a couple cases where it might make sense to expose it. I think users coming from Samza, or JMS's MessageListener (https://docs.oracle.com/javaee/7/api/javax/jms/MessageListener.html) understand a simple callback interface for message processing. In fact, people often ask why Kafka's consumer doesn't provide such an interface. I'd argue we do, it's KafkaStreams. The only issue is that the processor API documentation is a bit scary for a person implementing this type of api. My observation is that people using this style of API don't do a lot of cross-message operations, then just do single message operations and use a database for anything that spans messages. They also don't factor their code into many MessageListeners and compose them, they just have one listener that has the complete handling logic. Say I am a user who wants to implement a single Processor in this style. Do we have an easy way to do that today (either with the .transform/.process methods in kstreams or with the topology apis)? Is there anything we can do in the way of trivial helper code to make this better? Also, how can we explain that pattern to people? I think currently we have pretty in-depth docs on our apis but I suspect a person trying to figure out how to implement a simple callback might get a bit lost trying to figure out how to wire it up. A simple five line example in the docs would probably help a lot. Not sure if this is best addressed in this KIP or is a side comment.	Add some new methods to TopologyBuilder or add a new high level builder next to KStreamBuilder.	low • would add new API and not affect current users

Processor API "clumsy" to use	In Processor API, sources, processors, and sinks are solely connected to each other *by name* (ie, by using String). Each time, a processor or sink should be downstream to a source/processor user need to specify the corresponding name. It might be easier to allow to use the the actual "Processor Object" that should be used.
	
	Current:
	
	`builder.addSource("soureNode", "sourceTopic").addProcessor("processor", ..., "sourceNode");`
	`// builder returns TopologyBuilder to allow chaining`
	
	Basic Idea:
	
	`Source s = builder.addSource("sourceNode", "sourceTopic");`
	`Processor p = builder.addProcessor("processor", ..., s); // the source object s replaces the name "sourceTopic");`
	
	The main questions we need to consider is, if we don't limit the user, and how intuitive the API will get. It's should be a low level API and there is no need to get too close to patterns as offered in the DSL.
	
	Importance: low
		We would need to expose the concept on a "node" in `TopologyBuilder`.
		
		If we allow for this, we could actually get rid of all names (and only have them as optional parameters; if not specified, the name is generated and can be retrieved via Source#name()):
		
		`Source s = builder.addSource("sourceTopic");`
		`Processor p = builder.addProcessor(..., s); // the source object s replaces the name "sourceTopic");`
		
		We could even allow chaining (that would implicitly connect nodes):
		
		`builder.addSource("sourceTopic).addProcessor(...); // no name for neither Source no Processor and Processor consumes from Source`
			low
			
			• this would add "syntactic sugar"

Store.close() available within Processor, Transformer, and ValueTransformer	Currently, user can call `store.close()` within their user code. However, Streams will handle the stores including closing store automatically. Thus, user should not be able to call this method. For custom store, users only need to implement this method. Currently, we have a JavaDoc hint for this, but it would be better to get it into the API directly. *Importance: low*	Spilt interface into two interfaces, and hand the "limited" interface that does not offer .close() when a user retries a store from the context within an operator.	Should not affect anybody, as nobody should call .close() anyway – otherwise their code is broken in the first place.

Improve pattern to build custom stores	Building custom stores is a little hard with the current API, and we should simplify this. This requires a KIP that should cover a fix for 	Partial Redesign.	Should only affect advanced /power users.
KTable. toStream	KTable.toStream might have a confusing name. Consider KTable.getChangelog() or .toChangelog().	Needs discussion.	
Improve brachning	It's clumsy to use `branch()` as handling the returned array requires to do "index mapping" and breaks the flow.	Cf. https://issues.apache.org/jira/browse/KAFKA-5488	Low impact. We would only add a new branching API.

Many of the above issues are related to each other and/or overlap. This, also reflects in a bunch of JIRAs that are all related to API changes:

- https://issues.apache.org/jira/browse/KAFKA-4125 (Rich Functions)
- https://issues.apache.org/jira/browse/KAFKA-3455 (valid?)
- https://issues.apache.org/jira/browse/KAFKA-4713 (ProcessorContext.init)
- https://issues.apache.org/jira/browse/KAFKA-4218 (add key to ValueTransformer – ie. mapValues and transformValues)
- https://issues.apache.org/jira/browse/KAFKA-4217 (add flatTransform() and flatTransformValues() – seem invalid to me)
- https://issues.apache.org/jira/browse/KAFKA-4346 (add foreachValue to KStream)
- https://issues.apache.org/jira/browse/KAFKA-3745 (add key to ValueJoiner)
- https://issues.apache.org/jira/browse/KAFKA-4726 (add key to ValueMapper)
- https://issues.apache.org/jira/browse/KAFKA-4713 (Processors cannot call public methods on ProcessorContext from the init method)
- https://issues.apache.org/jira/browse/KAFKA-5488 (Improve branching)

Thus, to tackle this issue, it seems to be a good idea to break it down into groups of issues, and do a KIP per group to get a overall sound design.

Further Join Improvements

In order to get as close as possible to SQL-like join semantics, KStream-KStream left/outer join could be further improved. Right now, records with `null` key are dropped – however, for left/outer join this "limits" the join result unnecessarily. If we follow SQL `NULL` semantics, it holds that `NULL!=NULL`, thus we know that a `NULL`-key record will not join anyway – thus, there is not need that `null`-key records are co-located to each other and thus, we can just call `ValueJoiner` with the record and `null` as second parameter.

We can apply the same semantics to KStream-KTable left-join.

Not sure about KTable-KTable join though. IIRC, a changelog topic does not allow for `null`-keys in the first places, thus the scenario does not apply.

Not sure is this is a simple JIRA or if a KIP is required...?