KIP-418: A method-chaining way to branch KStream

Status

Current state: Accepted

Discussion thread: here

Voting thread: here

JIRA: *Unable to render Jira issues macro, execution error.*

Unable to render Jira issues macro, execution error.

Pull request: PR-9107

Motivation

`KStream#branch` method uses varargs to supply predicates and returns array of streams ('Each stream in the result array corresponds position-wise (index) to the predicate in the supplied predicates').

This is poor API design that makes building branches very inconvenient because of 'impedance mismatch' between arrays and generics in Java language.

- In general, the code have poor cohesion: we need to define predicates in one place, and respective stream processors in another place of code. In case of change we must remember to edit two pieces of code.
- If the number of predicates is predefined, this method forces us to use 'magic numbers' to extract the right branch from the result (see examples here).
- If we need to build branches dynamically (e. g. one branch per enum value) we inevitably have to deal with 'generic arrays' and 'unchecked typecasts'.

Public Interfaces

In accordance with KStreams DSL Grammar, we introduce the following new elements:

- **`split`** DSLOperation
- **`BranchedKStream`** DSLObject with following DSLOperations:
  - **`branch`**
  - **`defaultBranch`**
  - **`noDefaultBranch`**
- **`Branched`** DSLParameter.

Description

1. The `split(Named named)` operation returns `BranchedKStream<K,V>`. Named parameter is needed so one can name the branch operator itself, and then all the branches might get index-suffixed names built from the branch operator name.

The overloaded parameterless alternative `split()` is also available.

2. `BranchedKStream` has the following methods:

- `BranchedKStream<K,V> branch(Predicate<? super K, ? super V> predicate, Branched<K,V> branched)` -- creates a branch for messages that match the predicate and returns `this` in order to facilitate method chaining.
- `Map<String, KStream<K,V>> defaultBranch(Branched<K,V> branched)` -- creates a default branch (for messages not intercepted by other branches) and returns the dictionary of named KStreams.
- `Map<String, KStream<K,V>> noDefaultBranch()` -- returns the dictionary of named KStreams.

Both `branch` and `defaultBranch` operations also have overloaded alternatives without the `Branched` parameter.

3. `Branched` parameter extends `NamedOperation` and has the following static methods:

- `as(String name)` -- sets the name of the branch (auto-generated by default, when `split` operation is named, then the names are index-suffixed).
- `withFunction(Function<? super KStream<K, V>, ? extends KStream<K, V>> chain)` — sets an operation with a given branch. By default, it is an `s->s` identity function. Can be complex, like `s->s.mapValues...`, a composition of functions etc.
- `withConsumer(Consumer<? super KStream<K, V>> chain)` — sets a consumer for a given branch.
- `withFunction(Function<? super KStream<K, V>, ? extends KStream<K, V>> chain, String name)` — sets both an operation and a name.
- `withConsumer(Consumer<? super KStream<K, V>> chain, String name)` — sets both a consumer and a name.

The Map returned by `defaultBranch`/`noDefaultBranch` allows us to collect all the KStream branch objects in a single scope.

How the resulting Map is formed

The keys of the Map entries are defined by the following rules:

- If `Named` parameter was provided for `split` , its value is used as a prefix for each key. By default, no prefix is used
- If a `name` is provided for the `branch`, its value is appended to the prefix to form the Map key
- If a `name` is not provided for the branch, then the key defaults to prefix + position of the branch as a decimal number, starting from "1"
- If a `name` is not provided for the `defaultBranch` call, then the key defaults to prefix + "0"

The values of the Map entries are formed as following:

- If no chain function or consumer is provided, then the value is the branch itself (which is equivalent to `ksks` identity chain function)
- If a chain function is provided and returns a non-null value for a given branch, then the value is the result returned by this function
- If a chain function returns `null` for a given branch, then the respective entry is not put to the map
- If a consumer is provided for a given branch, then the the respective entry is not put to the map

For example:

```
var result =
   source.split(Named.as("foo-"))
  .branch(predicate1, Branched.as("bar"))           // "foo-bar"
  .branch(predicate2, Branched.with(ks->ks.to("A"))  // no entry: a Consumer is provided
  .branch(predicate3, Branched.with(ks->null))       // no entry: chain function returns null
  .branch(predicate4)                                // "foo-4": name defaults to the branch position
  .defaultBranch()                                   // "foo-0": "0" is the default name for the default branch
```

Usage Examples

The following section demonstrates some standard use cases for the proposed API

Simple Example: Direct Branch Consuming

In many cases we do not need to have a single scope for all the branches, each branch being processed completely independently from others. Then we can use 'consuming' lambdas or method references in `Branched` parameter:

```
source.split()
.branch((key, value) -> value.contains("A"), Branched.with(ks->ks.to("A")))
.branch((key, value) -> value.contains("B"), Branched.with(ks->ks.to("B")))
.defaultBranch(Branched.with(ks->ks.to("C")));
```

More Complex Example: Merging Branches

In other cases we want to combine branches again after splitting. The map returned by `defaultBranch` /`noDefaultBranch` methods provides access to the branches in the same scope:

```
Map<String, KStream<String, String>> branches = source.split()
  .branch((key, value) -> value == null,
          Branched.with(s->s.mapValues(v->"NULL"), "null")
  .defaultBranch(
          Branched.as("non-null"));

branches.get("non-null")
 .merge(branches.get("null"));
```

Dynamic Branching

There is also a case when one might need to create branches dynamically, e. g. one per enum value. This can be implemented the following way:

```
BranchedKStream branched = stream.split();
for (RecordType recordType : RecordType.values())
  branched.branch((k, v) -> v.getRecType() == recordType,
    Branched.with(recordType::processRecords));
```

This is why 'starting' `split()` operation is necessary and it is better to have it rather than add new `branch` method to `KStream` directly.

Otherwise we should treat the first iteration separately, and the code for dynamic branching becomes cluttered:

```
RecordType[] recordTypes = RecordType.values();
if (recordTypes.length != 0) {
  BranchedKStream branched = stream.
    branch((k, v) -> v.getRecType() == recordTypes[0],
       Branched.with(recordType::processRecords));

  for (int i = 1; i < recordTypes.length; i++)
    branched.branch((k, v) -> v.getRecType() == recordTypes[i],
      Branched.with(recordType::processRecords));
}
```

Proposed Changes

1. Add the following methods to `KStream`:

```
BranchedKStream<K,V> split();
BranchedKStream<K,V> split(Named n);
```

2. Deprecate the existing `KStream#branch` method.

3. Add and implement the following Branched class:

```
class Branched<K, V> implements Named<Branched<K,V>> {
    static Branched<K, V> as(String name);
    static Branched<K, V> withFunction(Function<? super KStream<K, V>, ? extends KStream<K, V>> chain);
    static Branched<K, V> withConsumer(Consumer<? super KStream<K, V>> chain);
    static Branched<K, V> withFunction(Function<? super KStream<K, V>, ? extends KStream<K, V>> chain, String
name);
    static Branched<K, V> withConsumer(Consumer<? super KStream<K, V>> chain, String name);
}
```

Add and implement the following BranchedKStream interface:

```
interface BranchedKStream<K, V> {
    BranchedKStream<K, V> branch(Predicate<? super K, ? super V> predicate);
    BranchedKStream<K, V> branch(Predicate<? super K, ? super V> predicate, Branched<K, V> branched);
    Map<String, KStream<K, V>> defaultBranch(Branched<K, V> branched);
    Map<String, KStream<K, V>> defaultBranch();
    Map<String, KStream<K, V>> noDefaultBranch();
}
```

(See https://github.com/apache/kafka/pull/6512 for a very rough draft).

Compatibility, Deprecation, and Migration Plan

The proposed change is backwards compatible.

The old KStreams#branch method should be **deprecated**.

Rejected Alternatives

1. A KStreamsBrancher class that works the same way, but does not require KStream interface modification:

```
new KafkaStreamsBrancher<String, String>()
   .branch((key, value) -> value.contains("A"), ks->ks.to("A"))
   .branch((key, value) -> value.contains("B"), ks->ks.to("B"))
   //default branch should not necessarily be defined in the end!
   .defaultBranch(ks->ks.to("C"))
   .onTopOf(builder.stream("source"));
```

Rejected because of violation of method-chaining (new auxiliary object is needed).

2.

```
source
   .split()
   .branch((key, value) -> value.contains("A"), ks->ks.to("A"))
   .branch((key, value) -> value.contains("B"), ks->ks.to("B"))
   .defaultBranch(ks->ks.to("C"));
```

Here the new KStream#branch() method returns KBranchedStream<K, V> object, which, in turn, contains `branch` and `defaultBranch` methods. This is critical that KStream consumers in .branch methods should be invoked immediately during the `branch` methods invocation. This is necessary for the case when we need to gather the streams that were defined in separate scopes back into one scope using auxiliary object:

```
@Setter
class CouponIssuer{
   private KStream<....> coffePurchases;
   private KStream<....> electronicsPurchases;

   KStream<...> coupons(){
       return coffePurchases.join(electronicsPurchases...)...
  }
}

CouponIssuer couponIssuer = new CouponIssuer();

transactionStream.branch()
     .branch(predicate1, couponIssuer::setCoffePurchases)
     .branch(predicate2, couponIssuer::setElectronicsPurchases);

KStream<..> coupons = couponIssuer.coupons();
```

This was rejected because of the difficulty of having branches in the same scope.