KIP-857: Streaming recursion in Kafka Streams

Status

Current state: *Under Discussion*

Discussion thread: here

JIRA: here

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Some algorithms are best expressed *recursively*, which would involve piping the output of a pipeline back to an earlier point in the same pipeline. An example of this is graph/tree-traversal, where it can be useful to recursively traverse up a tree as new leaf nodes arrive.

This document introduces the concept of *streaming recursion*, a new DSL operator to succinctly express it, and optimizations that Kafka Streams can make to recursive algorithms that aren't currently possible.

Public Interfaces

The following new method will be introduced:

```
interface KStream<K, V> {
        KStream<K, V> recursively(UnaryOperator<KStream<K, V>> op);
}
```

Note: `UnaryOperator` is `java.util.function.UnaryOperator`

Proposed Changes

The new `recursively` method enables users to express recursive algorithms. Consider an example where we count all descendants of each node in a graph:

```
// <NodeId, ParentId>
KStream<String, String> nodes = builder.stream("nodes");

// <NodeId, ParentId>
KTable<String, String> parents = nodes.toTable();

// count descendants by recursively producing parent records
// 1L is used as a dummy value below, since we will be discarding values when we count the records by key
KTable<String, Long> descendants = nodes
        .map((child, parent) -> { KeyValue(parent, 1L) }   // emit "update" for parent of new node
        .recursively((updates) -> {                        // recursively emit "updates" for each ancestor of
the parent
                // emit a new update for the parent of this node
                 // the root node has no parent, so recursion terminates at the root node
                updates
                        .join(parents, (count, parent) -> { parent })
                        .map((child, parent) -> { KeyValue(parent, 1L) })
        })
        .groupByKey()
        .count()
```

Note: for simplicity, this example assumes that graph nodes arrive and are processed in-order; i.e. parent nodes are always processed before children.

The `recursively` method applies input records to its `op` argument. The results are then both emitted as a result of `recursively` *and also fed back in to the op* `KStream`.

Restrictions:

- `op` cannot be `UnaryOperator.identity`, or an equivalent function that simply returns its argument unmodified - this would produce an infinite recursive loop, since there's no opportunity refine the output to break out of the loop.
- `op` *MUST* "terminate"; that is, it must have some condition which eventually prevents further recursion of a record. In our example here, the terminating condition is the `join`, since the root node of our graph will have no `parent`, so the `join` will produce no output for the root node.
    - We can attempt to detect "definitely non-terminating" arguments by failing to detect operations that can cause the stream to terminate (e.g. `filter`, `join`, `flatMap`, etc.) in the process graph produced by the function.
    - We *cannot guarantee* that a function that includes terminating operations (`filter`, `join`, `flatMap`, etc.) actually terminates.

Implementation

In `KStreamImpl`, implementation is fairly simple:

1. We call `op`, passing our current `KStream` as its argument. This produces our `output KStream`.
2. We wire up the `graphNode` from the `output KStream` as a *parent* of the current `KStream`. This takes care of the recursion.
3. Finally, we return the `output KStream`. This enables users to operate on the records that are being recursively produced, as above.

Compatibility, Deprecation, and Migration Plan

- No backwards incompatible changes are introduced.

Test Plan

The following tests will be added:

- Counting descendants of graph nodes arriving in-order (as above)
- Counting descendants of graph nodes arriving in any order

Rejected Alternatives

It's currently possible to implement streaming recursion via explicit topics, albeit with a number of disadvantages:

1. The explicit topic is entirely internal to the Topology, yet **it has to be managed explicitly by the user**.
    a. This is functionally a `repartition` topic, however, because it's explicitly managed, **Streams can't automatically delete consumed records**.
    b. Consequently, to prevent the topic growing unbounded, **users would need to set retention criteria, which risks possible data loss**.
2. In scenarios where repartitioning is not required, **the explicit recursive topic adds overhead**.
3. It also adds some complexity to the user's program, making it **more difficult to reason about than it needs to be**.