KIP-862: Self-join optimization for stream-stream joins

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *"Accepted"*

Discussion thread: https://lists.apache.org/thread/g211rn7j09z1cgrspz0kstxqkkbwfpq9

Voting thread: https://lists.apache.org/thread/shxrhyxlt87rpc64d0xlbkndr6hy9oyh

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

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

Motivation

A self-join is a join whose left and right-side arguments are the same entity (a stream reading from the same topic). Although self-joins are currently supported in Streams, their implementation is inefficient as they are implemented like regular joins where a state store is created for both left and right join arguments. Since both of these arguments represent the same entity, we don't need to create two state stores (as they will contain the exact same data) but only one. This optimization is only suitable for inner joins whose join condition is on the primary key. We do not consider foreign-key joins as we would need to create a state store for both arguments in order to be able to do efficient lookups. Hence, we will handle foreign-key self joins as regular inner foreign-key joins. Moreover, we do not consider outer joins since we are focusing on primary key joins and there will always be at least one join result, the current record joining with itself.

One should consider this work as a logical plan optimization rule. If the graph contains a join whose arguments are the same entity, then we will rewrite it by applying the aforementioned self-join optimization that will create only one state store.

This optimization only applies to Stream-Stream inner joins. A Table-Table inner self-join would not yield interesting results since a row would always join with itself. Moreover, this optimization does not apply to N-way self-joins as in the current implementation, an inner join is a binary operator. What this entails is that in the beginning one self-join will be applied whose results contain records whose columns are the concatenation of the columns of the left and right join arguments. The subsequent join would then be applied on the result of the first join (the concatenated rows) and not on the original input which means it won't be a self-join anymore. In order to optimize N-way self-joins, we would need to implement a new n-ary operator which is out of the scope of this KIP.

Public Interfaces

No public interfaces will be impacted.

The config *TOPOLOGY_OPTIMIZATION_CONFIG* will be extended to accept a list of optimization rule configs in addition to the global values "all" and "none" . The new optimization configs a user can provide are:

1. merge.repartition.topics
2. reuse.ktable.source.topics
3. single.store.self.join

The config can either have the values "all", "none" or any of the three rules above. It cannot combine "all" or "none" with any other config.

Proposed Changes

The changes required to implement this proposal are:

- Identify whether the arguments to a join are the same entity

The cases we are going to support are :

```
stream1 = builder.stream(INPUT_TOPIC);
stream1.join(stream1);
```

and

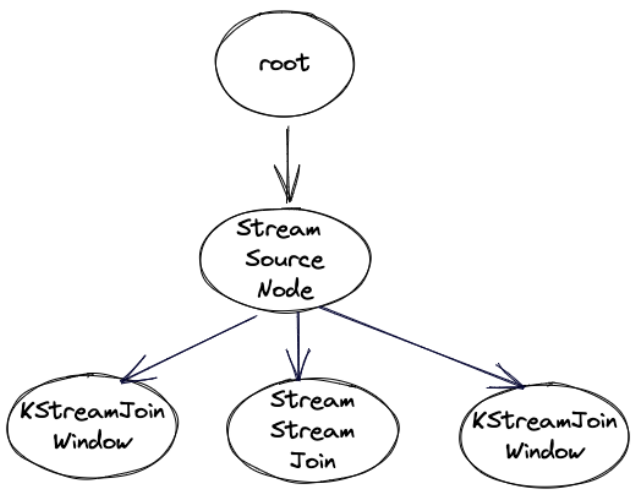
```
stream1 = builder.stream(INPUT_TOPIC);
stream2 = builder.stream(INPUT_TOPIC);
stream1.join(stream2);
```

The algorithm to identify whether two streams represent the same entity will check that the `StreamSourceNode` node has a single parent.

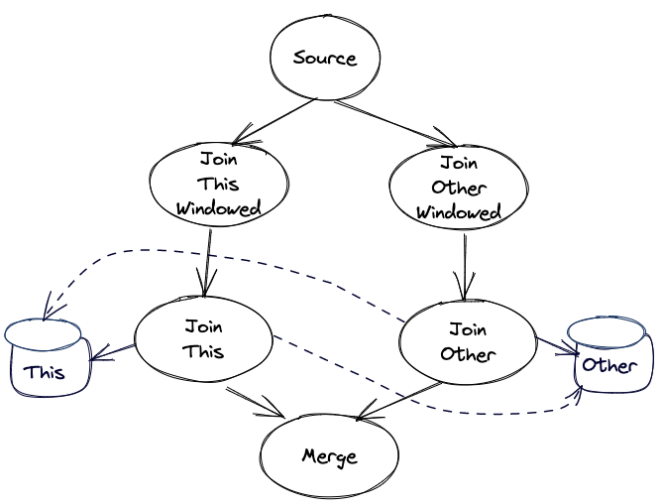
- Implement the self-join operator

The self-join operator will maintain one state store. For every new record, the operator will add it to the state store and use it to perform a lookup into the same state store to do the actual join. This means that every record will at least join with itself.

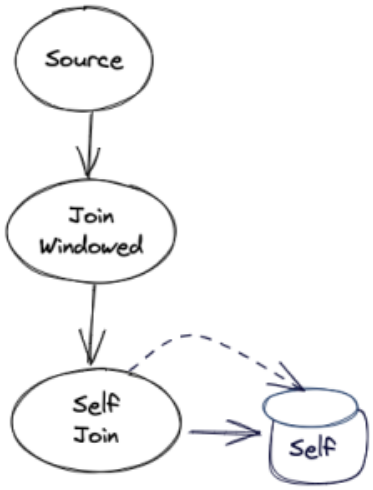
- Add a rule to the optimizer that will rewrite an inner join to a self-join. The graphs (logical plans) created from the DSL excerpts above are the same (after the already existing optimization `mergeDuplicateSourceNodes`) and look as follows:



The graph gets translated into the following topology (physical plan)



The self-join rewriting will take the above graph, and will translate it into the following topology instead:



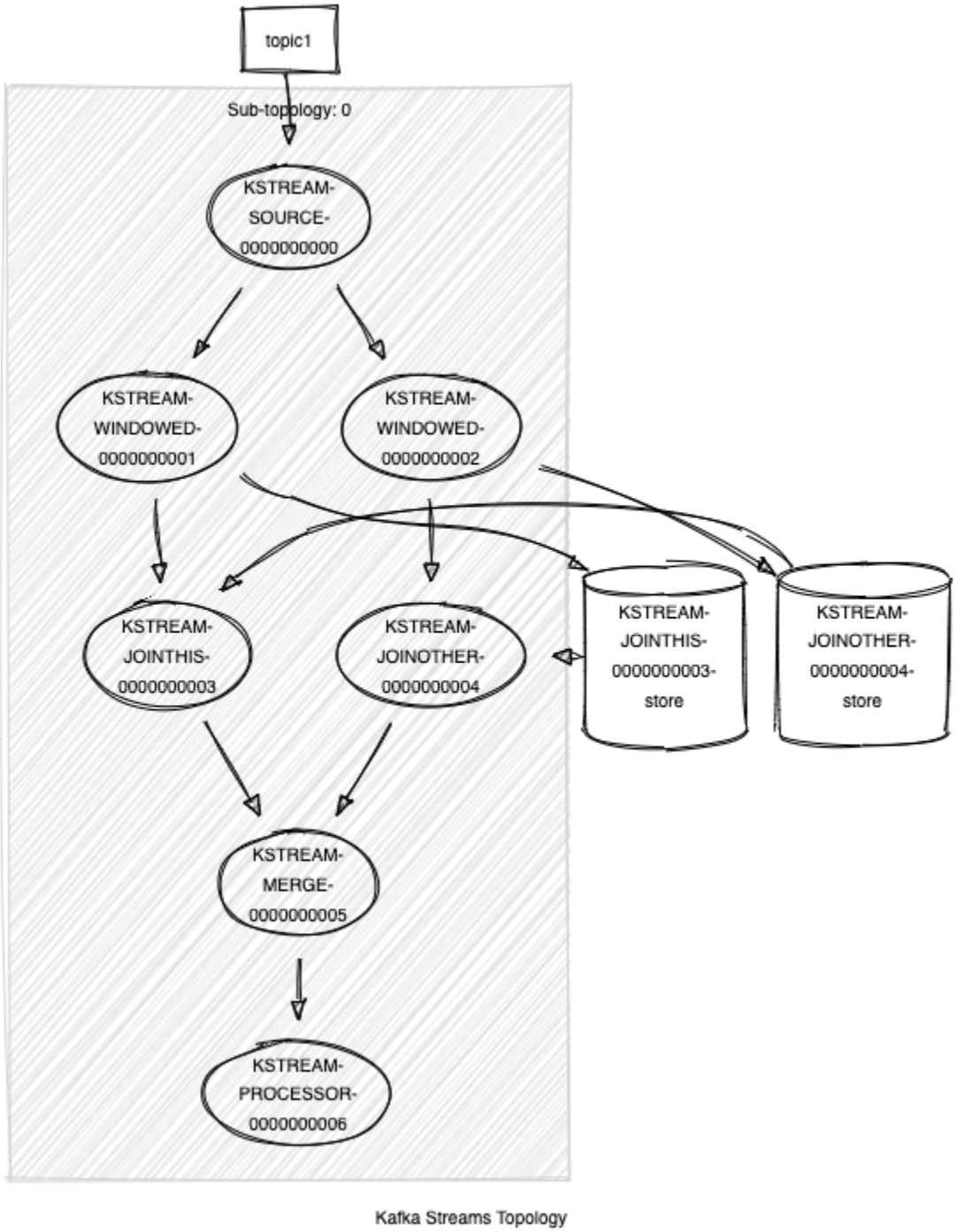
Compatibility, Deprecation, and Migration Plan

The change is backward compatible since:

- *It reuses existing topics/state stores*
- *It does not change the names of existing topics/state stores*
- *It removes from the topology the right-side state store but this state store is kept around in case users want to roll back.*
- *It does not change the internal naming of processors or graph nodes*

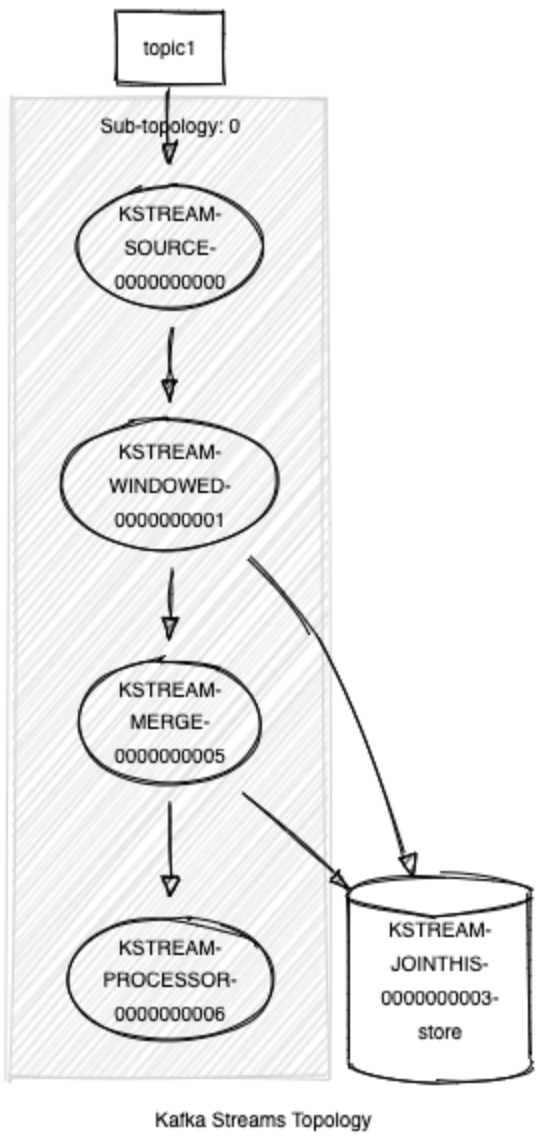
Here is an example topology of an inner join and how it gets rewritten:

```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [topic2])
      --> KSTREAM-WINDOWED-0000000001, KSTREAM-WINDOWED-0000000002
    Processor: KSTREAM-WINDOWED-0000000001 (stores: [KSTREAM-JOINTHIS-0000000003-store])
      --> KSTREAM-JOINTHIS-0000000003
      <-- KSTREAM-SOURCE-0000000000
    Processor: KSTREAM-WINDOWED-0000000002 (stores: [KSTREAM-JOINOTHER-0000000004-store])
      --> KSTREAM-JOINOTHER-0000000004
      <-- KSTREAM-SOURCE-0000000000
    Processor: KSTREAM-JOINOTHER-0000000004 (stores: [KSTREAM-JOINTHIS-0000000003-store])
      --> KSTREAM-MERGE-0000000005
      <-- KSTREAM-WINDOWED-0000000002
    Processor: KSTREAM-JOINTHIS-0000000003 (stores: [KSTREAM-JOINOTHER-0000000004-store])
      --> KSTREAM-MERGE-0000000005
      <-- KSTREAM-WINDOWED-0000000001
    Processor: KSTREAM-MERGE-0000000005 (stores: [])
      --> KSTREAM-PROCESSOR-0000000006
      <-- KSTREAM-JOINTHIS-0000000003, KSTREAM-JOINOTHER-0000000004
    Processor: KSTREAM-PROCESSOR-0000000006 (stores: [])
      --> none
      <-- KSTREAM-MERGE-0000000005
```



Self-join topology:

```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [topic1])
      --> KSTREAM-WINDOWED-0000000001
    Processor: KSTREAM-WINDOWED-0000000001 (stores: [KSTREAM-JOINTHIS-0000000003-store])
      --> KSTREAM-MERGE-0000000005
      <-- KSTREAM-SOURCE-0000000000
    Processor: KSTREAM-MERGE-0000000005 (stores: [KSTREAM-JOINTHIS-0000000003-store])
      --> KSTREAM-PROCESSOR-0000000006
      <-- KSTREAM-WINDOWED-0000000001
    Processor: KSTREAM-PROCESSOR-0000000006 (stores: [])
      --> none
      <-- KSTREAM-MERGE-0000000005
```



Kafka Streams Topology

As you can see, none of the indices or names of the process is affected.

Rejected Alternatives

Add to the DSL the operator selfJoin. We did not go with this approach as we prefer to push the complexity of the optimization to streams instead of to the user.

Pros:

1. This will make backwards-compatibility a non-issue as a user that upgrades from an older version does not have access to this DSL. If they want to use a self-join, they have to manually make the change in their code.
2. The code for a join can be chained like: `builder.stream("topic").selfJoin().map(...)...` If we don't have the operator, the code for a self-join needs to be broken into two parts: create the stream and get a reference, use the reference in a join like: `stream.join(stream)`
3. Implementation is straight-forward as we don't need to implement an optimization rule to do the rewriting

Cons:

1. The self-join is a physical plan optimization, it is not a different operator. We expose physical plan information to the user.
2. We put the burden of creating an optimal topology on the user as they need to know about the self-join operator to use it. If they don't, then their topologies will be inefficient. That's the beauty of having it as a rewriting, users will take advantage of it without even realizing it.
3. Topologies may not be created by one user only and/or topologies may involve multiple operators that do all sorts of stuff. At some point in the code, the user has two variables that they join. They may not know that these refer to the same stream to realize that they can do a self-join instead of a regular join.

Another option that combines the benefits of both, is to implement both the DSL operator and the optimization rule. The downside of this approach is it could get confusing to the users as there would be multiple ways to achieve the same thing.