

FLIP-14

Scoped Loops and Job Termination

This document is going to be moved to the Apache Flink wiki upon approval

Status

Current state: *PENDING*

Related JIRA:

FLINK-5089 (Loops and Nesting) <https://issues.apache.org/jira/browse/FLINK-5089>

FLINK-2390 (Iteration Termination) <https://issues.apache.org/jira/browse/FLINK-2390>

Pull-Requests (in dependency order):

1. Loops + StreamScope API [prototype branch](#) (PR pending)
2. Job Termination [prototype branch](#) (PR pending)

Motivation and Scope

The current implementation of loops (or Iterations) on the streaming API of Flink does not allow for a deterministic termination of jobs and depends on user-defined timeouts to enforce termination. This impacts correctness, resulting to incomplete processing or/and inconsistent state when iterations exist in the topology. We propose a new functional, yet compositional API (i.e. nested loops) for expressing asynchronous DataStream loops with proper scoping and a distributed coordination algorithm for graph termination that exploits scoping information.

This is a proposal for a major fix in the DataStream execution and thus it should not be perceived simply as an improvement or a feature. It recommends changes to the **PublicEvolving** stream iteration API which means that these changes can already be incorporated in any upcoming Flink release if this proposal gets accepted by the community.

I - Scoped Loops

In the [current Iteration API](#) users can open and close iterations (i.e. add feedback edges) in an arbitrary manner. This means that users can nest iterations, yet close them outside outer iterations, since there is no notion of scopes. Apart from being a rather inconvenient programming paradigm, it makes things hard when we want to establish global properties, such as the state of an iterative computation and as a result of the whole dataflow graph, in a decentralized way (Termination and timeout elimination are addressed in part II of the FLIP).

In order to address iterations in a more consistent way that allows for proper scoping, we propose the following API, as a first step towards that direction.

The Loops API

A **loop** can be defined in terms of a **LoopFunction** which takes an input Data Stream and returns the *feedback* and the *output* Data Streams. Each loop has its own unique context. This API allows the system to create a separate context per loop and make operators and tasks **aware** of their current scope in the graph (i.e. their loop and outer loops in which they reside). The **LoopFunction** which wraps the logic of an iteration, looks as follows in Java:

```
@Public
public interface LoopFunction<T, R> extends Function, Serializable {
    Tuple2<DataStream<T>, DataStream<R>> loop(DataStream<T> value);
}
```

For loops with a different feedback type, we propose the **CoLoopFunction** that takes a connected stream of the input and the feedback streams instead.

```
@Public
public interface CoLoopFunction<T, F, R> extends Function, Serializable {
    Tuple2<DataStream<F>, DataStream<R>> loop(ConnectedStreams<T, F> input);
}
```

The **iterate** operation of the **DataStream** will also support these two flavors of loops as follows:

```
public <R> DataStream<R> iterate(LoopFunction<T, R> loopFun)
public <F,R> DataStream<R> iterateWithFeedback
    (CoLoopFunction<T, F, R> coLoopFun)
```

Loops can be trivially nested as shown below:

```
DataStream loop = map1.iterate(new LoopFunction<Integer, String>() {
    @Override
    public Tuple2<DataStream<Integer>, DataStream<Integer>>
        loop(DataStream<Integer> input) {
        input.map(...).iterate(new LoopFunction<Integer, Integer>() {
            @Override
            public Tuple2<DataStream<Integer>, DataStream<Integer>>
                loop(DataStream<Integer> input2) {
                DataStream<> tmp=input2.map(...).split(...);
                return new Tuple2<>(tmp.select(...), tmp.select(...));
            }
        });
        return new Tuple2<>(nestedLoop, nestedLoop);
    }
    ...
});
```

The [StreamScope](#)

Given the proposed way of defining loops, the system can now infer scopes in a structured way. A **StreamScope** is an immutable data structure, specified per loop context and it basically contains the chain of all of its outer scopes. This is very powerful, not only for distributed termination but also for future additions in the systems such as progress tracking (see [TimelyDataflow](#)) for iterative computations and stream supersteps later on. These are the properties of StreamScope :

- All operators that do not reside within a loop contain the default **StreamScope**.
- The two main operations of a StreamScope are **nest()** and **unnest()**
- Every invocation of the DataStream **iterate** operation generates a new StreamScope, **nested** on its outer scope.
- The level of each StreamScope (**getLevel()**) is the number of its outer scopes. **Fig.2** shows the assigned scope levels on each operator in an example graph.
- StreamScopes can be compared topologically (**isInnerOf(..)**, **isOuterOf()**).
- Operators who reside in the exact same loop should have **equal** StreamScope
- All outer scopes of a StreamScope can be retrieved
 - (e.g. scope.getContextForLevel(scope.getLevel()-2), or scope.unnest().unnest())

API Restrictions based on Scopes

All binary operations can and should be restricted on data streams of the same StreamScope.

II - Decentralized Job Termination

Currently, an iteration head and tail are two obscure, interconnected dataflow operators that utilize an in-memory blocking queue. A fixed timeout at the consumer side (head) terminates the head task and subsequently allows for a graceful, yet irregular, *shut-down* of the dataflow graph in topological operator order.

Problems with Existing Mechanism

- An arbitrary loop timeout does not necessarily mean that all computation in the dataflow graph has finished. With the existing mechanism it is possible to end up terminating a job non-deterministically in a non-final state when computation is still pending. This can trivially lead to inconsistent processing and boils down to **at-most-once guarantees** when loops exist in the dataflow graph. (**Note:** Setting a very large value by default does not really solve the issue. Operators can take an arbitrary amount of time to execute, i.e. due to checkpointing very large state, communicating with another service etc).
- It introduces an unnecessary lower latency cap to the termination process.
- It adds execution details (i.e. timeouts) to the user-facing API. The user should not really worry about queue staleness timeouts, especially since this is something that the runtime can instead figure out and execute in speculative manner.

Proposal

For graphs with no loops, the termination happens in topological sequential order, from sources to sinks. Starting with the end-of-stream on all regular sources, each operator terminates in-order after all prior operators have already terminated. This is trivial when we have a DAG, but not likewise when loops exist in the pipeline. Computation might still be pending in the distributed dataflow graph, even when all regular sources are done.

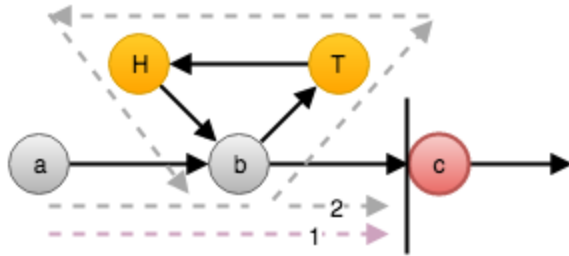
We foresee a simple and practical way to terminate any arbitrary dataflow graph with or without cycles. For a correct termination, the operators can determine in a decentralized manner that there is no pending data/computation in the pipeline. Below, we define a simple protocol that can correctly put this logic into practice.

Technique Overview

The main idea of the decentralised graph termination algorithm is that when loops are scoped, we can trivially coordinate the termination decision recursively, starting from outermost loop of the graph down to the innermost and then back. The important thing is to get knowledge of the status of all operators within a scope and its inner scopes. Thus, we propose an abortable, distributed passing-the-baton approach to traverse the graph and collect all statuses of the operators in order to determine whether termination is feasible.

Tricky Part: Termination is **staged** between sections of the dataflow graph that contain at least one loop. To enforce staging, we use a form of alignment. We basically align the batons within and out of loops. The graph in Fig.1 shows a part of the execution of the algorithm from node 'a' to node 'c'. Node 'c' will wait (align) to receive the event twice, via: $a \rightarrow b \rightarrow c$ and $a \rightarrow b \rightarrow T \rightarrow H \rightarrow b \rightarrow c$.

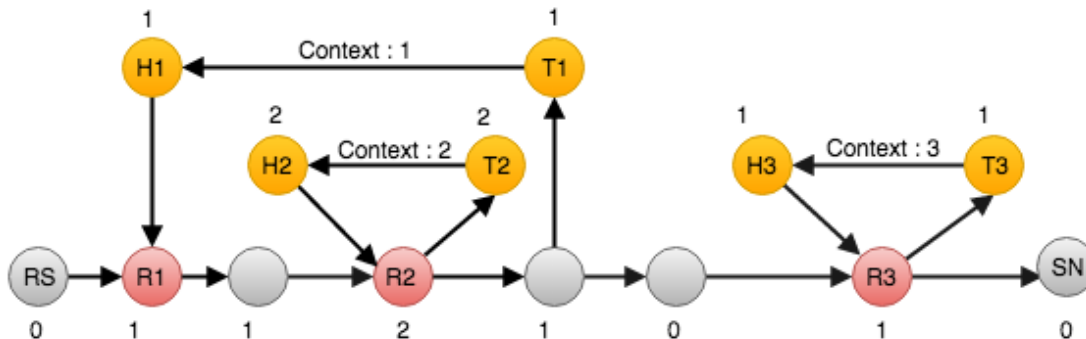
Fig.1 - Example of Alignment



In our implementation, the baton is represented as a **TerminationAttempt** event that contains the following properties:

1. The current **StreamScope** of the attempt
2. An attempt identifier (~epoch)
3. The scope level of the last visited operator (to decide on alignment)
4. The last **completed** scope level
5. The **collective status** of the operators visited in the current attempt

Fig.2 - A dataflow graph example that contains nested loops



The Algorithm

In Fig. 2 we depict an example throughout all phases to use as a reference. The protocol works as follows:

Termination Protocol : Once all regular stream sources have reached the “end-of-stream” the system enters into a “Attempting” phase, upon which the following abortable protocol becomes active:

- 1.1. The regular stream sources broadcast a TerminationAttempt event to downstream operators
- 1.2. Three types of operators might receive this event
 - Type 1 : Head Operators
 - Type 2 : Operators with Feedback link, (Red nodes in Fig. 2)
 - Type 3 : Other operators(Grey nodes in Fig. 2)
- 1.3. Each of the operators handles the TerminationAttempt as follows:
 - 1.3.1. **Type 3:** Adds operator status (BUSY | IDLE) to the event and broadcast it.
 - 1.3.2. **Type 2:** If the event is not associated with a StreamScope, then operator will associate its outermost non-terminated StreamScope to this event. Add operator status to the event and broadcast it.
 - 1.3.3. **Type 1:** if the operator’s StreamScope is not the one in the event then it will just add its status to the event and forward it downstream. Otherwise, it will go into a local speculation mode on which the following protocol is activated :
- 1.4. The operator will enter an ATTEMPTING state, broadcast a TerminationAttempt downstream and wait to receive this event back via the feedback loop.
 - 1.4.1. After receiving the TerminationAttempt back it does the following:
 - 1.4.1.1. If the operator did not receive any data streams in between and the TerminationAttempt is **collectively** marked as IDLE, it will
 - Mark its StreamScope as **complete** in this event and flag this context as terminated
 - Terminate and forward this event which will cause all operators to terminate up until reaching another Type 2 operator and goes again to 1.3.2
 - 1.4.1.2. Otherwise, the Operator will repeat protocol from step 1.4 with an increased attempt id