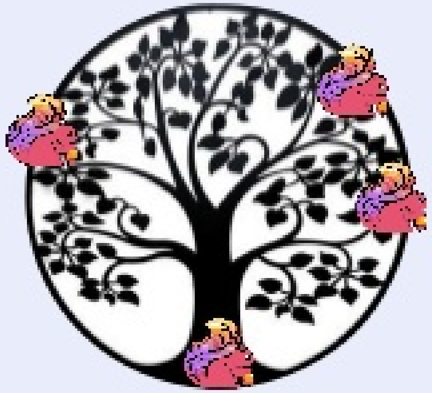
Stream Loops on Flink

Reinventing the wheel for the streaming era

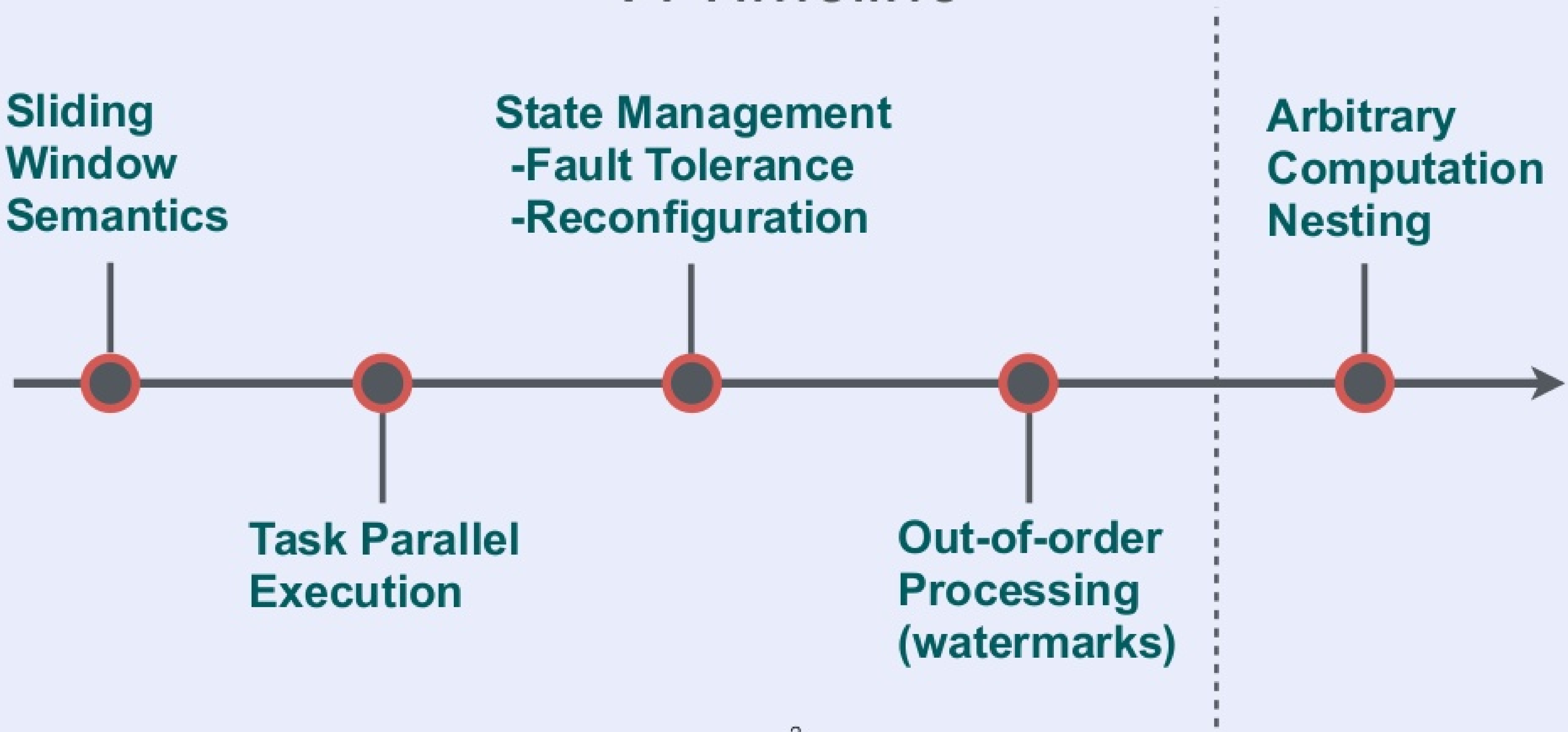


Paris Carbone

Systems Researcher@KTH/SICS <parisc@kth.se>
Committer@Flink <senorcarbone@apache.org>

Stream Compute Landmarks

A Timeline

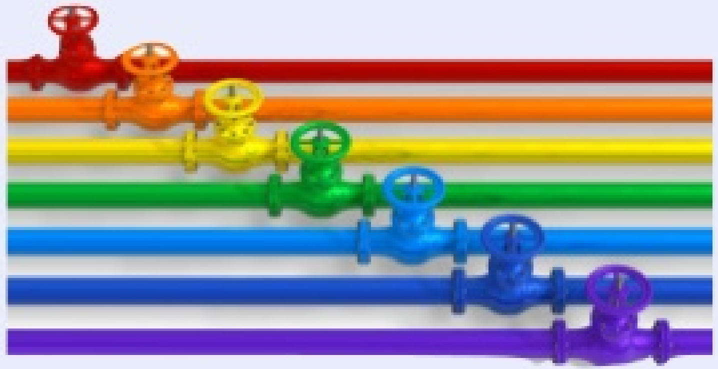


Has Streaming Subsumed Batch?

short answer: NO

Flink DataSet

Flink DataStream



single-pass

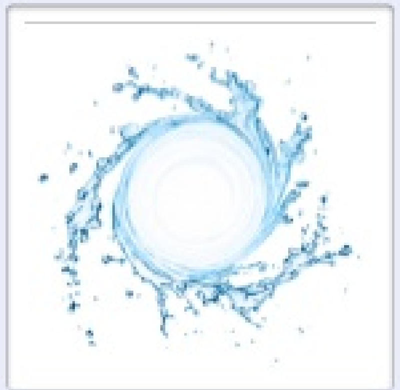
- Staged Processing
- Allows Bulk/Delta **Iterative** Computation

- Continuous Processing
- Allows correct **Acyclic** Computation

long answer: not YET

Why Iterations Matter

- Iterations are fundamental building blocks for:
  - Graph Analysis (PageRank, Conn.Comp., SSSP etc.)
  - Machine Learning (Gradient Descent, PCA etc.)
  - Transactions (e.g., optimistic concurrency control)



Iterative Processing Primitives are currently <u>not present</u> in todays' production-grade stream processing systems.

Current Challenges

- Data is born and evolves continuously as a data stream (e.g., user interactions, sensor events, server logs)

Current Challenges

- To run iterative analysis users have manually do the…
  - bucketing

Current Challenges

- To run iterative analysis users have manually do the…
  - bucketing
  - laundry (iterative job scheduling)

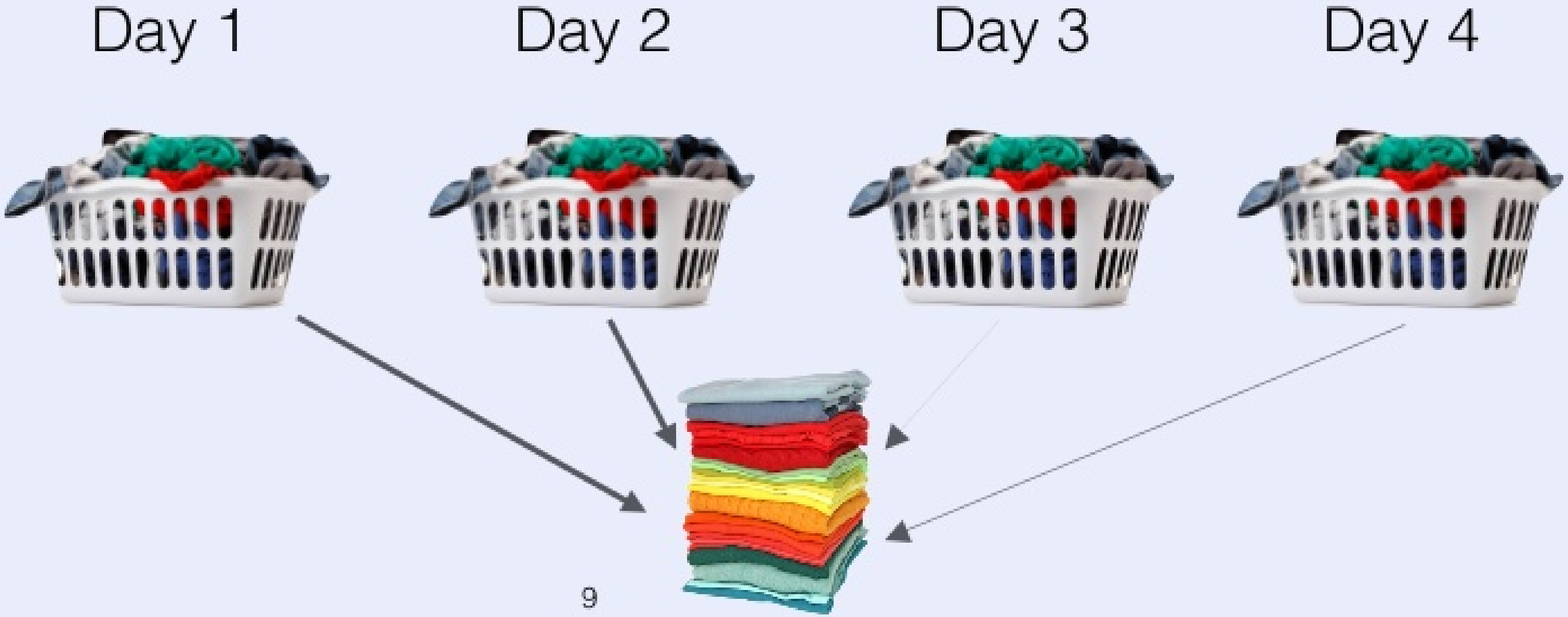
Current Challenges

- To run iterative analysis users have manually do the…
  - bucketing
  - laundry (iterative job scheduling)
  - sorting (model integration)

Day 1 Day 2 Day 3 Day 4

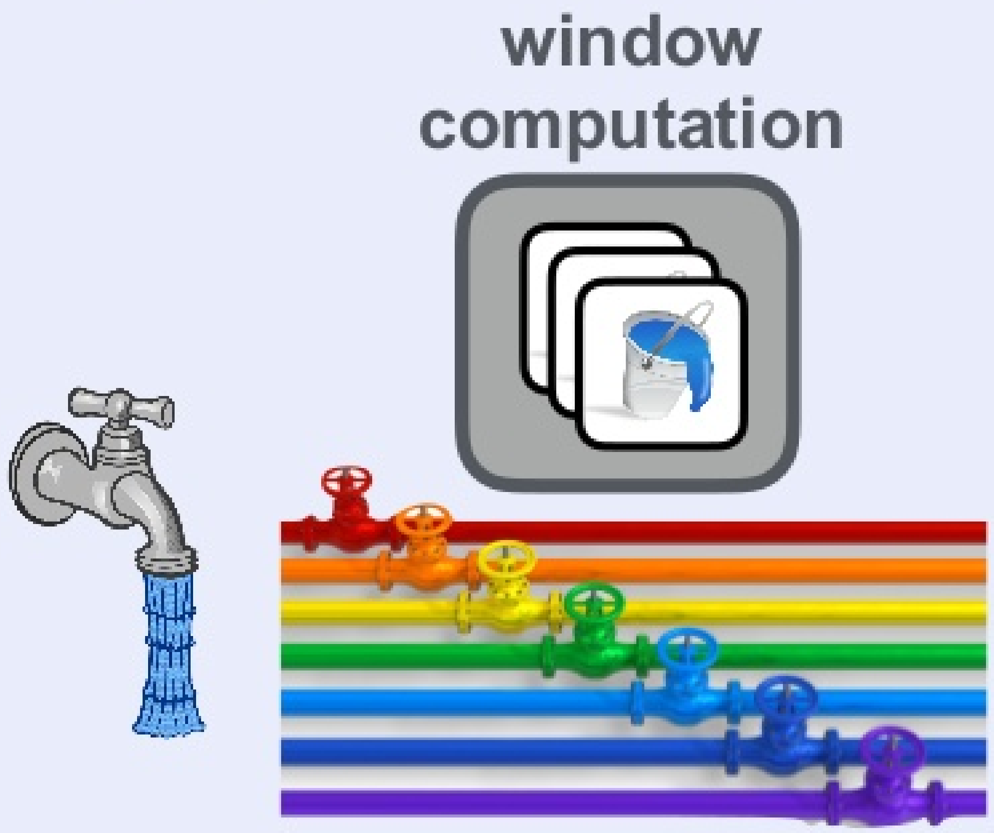
Current Challenges

- To run iterative analysis users have manually do the…
  - bucketing
  - laundry (iterative job scheduling)
  - sorting (model integration)



A Partially Solved Problem

- Most challenges mentioned are solved and automated for single-pass aggregation via stream windowing.



windows are a **natural fit** for declaring iterative computation

How would a **distributed iterative computation** look like as a **special type of window aggregation?**

Anatomy of Iterative Computation

Solution (finite state)

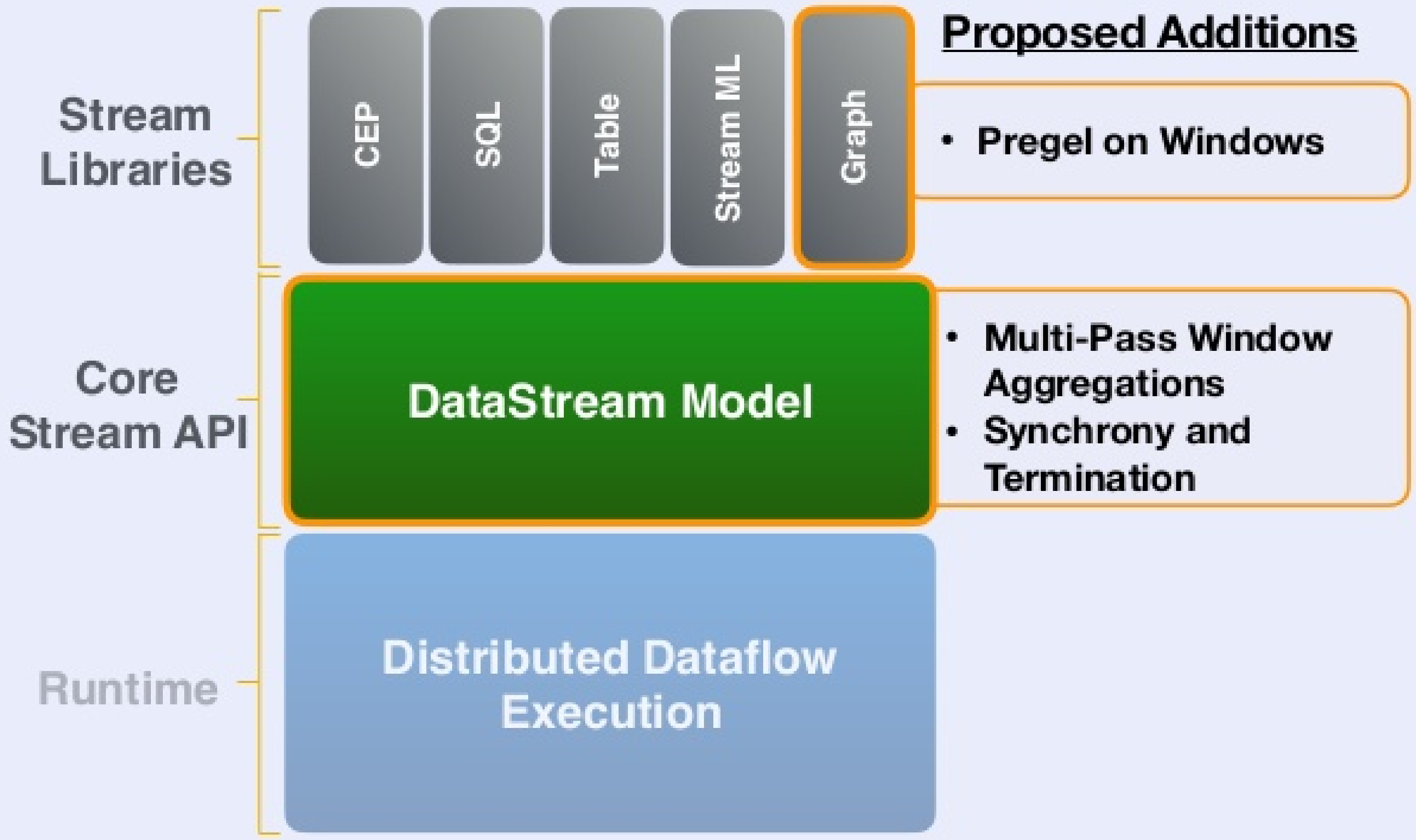
Loop Step Function (computation)

```
def iterate(c,L)(x_init)
    x = x_init
    do{
        x = L(x)
    }while(¬c(x,L(x)))
    x
```

Final Result-Solution

Termination (fixpoint/fixed)

Programming Model Extensions



based on Flink 1.6

Window Iterations

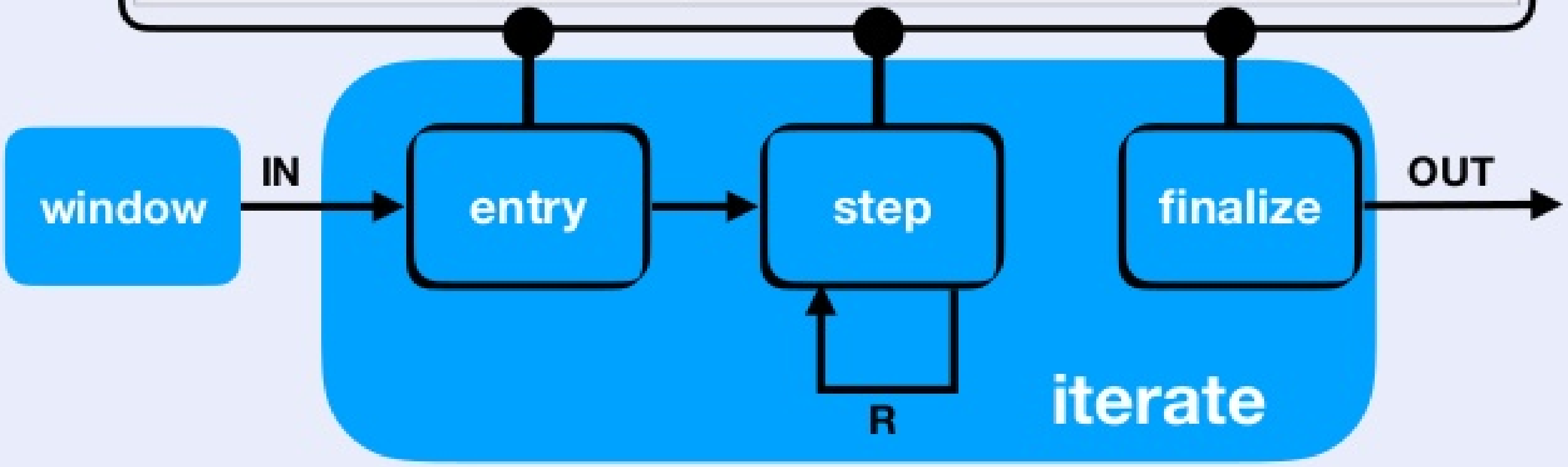
An extension of existing window aggregation to multi-pass aggregates

```
val stream: DataStream[IN] = ...
stream.keyBy(...).window(...)
   .iterate(<Termination>,<Synchrony>,<Key>,<Entry>,<Step>,<Finalize>)
```

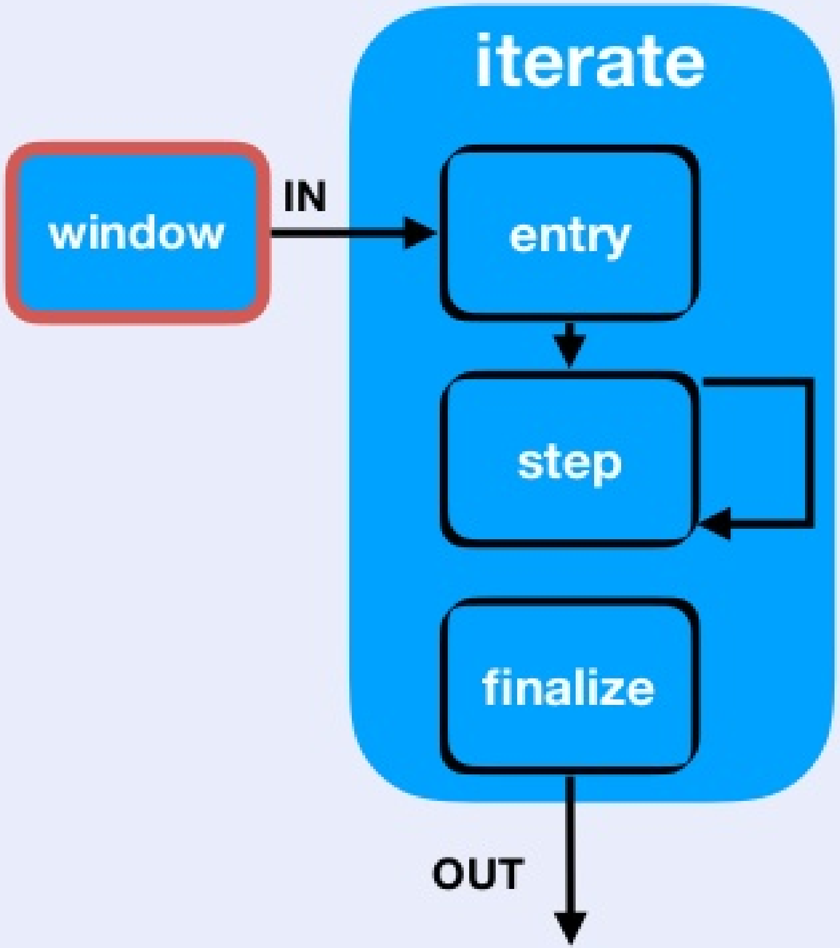
Iteration Primitive	Definition	Description
Termination	Termination.Fixpoint Termination.Fixed($n \in \mathbb{N}$)	Loop Termination Criterion
Synchronization	Synchrony.Stale($n \in \mathbb{N}$) Synchrony.Strict	Loop Synchronization Type
Loop Key	$R \rightarrow K$	Key Extractor for Structured Loop
Entry Function	$(ctx, [IN]) \rightarrow (ctx, [R])$	Initialization Logic
Step Function	$(ctx, [R]) \rightarrow (ctx, [R])$	Iteration Step Logic
Finalize Function	$(ctx) \rightarrow (ctx, [OUT])$	Finalization Logic after Termination

Loop Context

LoopContext Properties	Definition
key: K	The current key of the computation
localProgress: $\mathbb{N}$	The current superstep of the local computation
globalProgress: $\mathbb{N}$	The last parallel complete superstep
eventTime: $\mathbb{N}$	The event time (ctx) of local computation
loopState: ManagedState	keyed state, purgeable per iterative process
persistentState: ManagedState	keyed state, persistent across iterative processes



PageRank Example

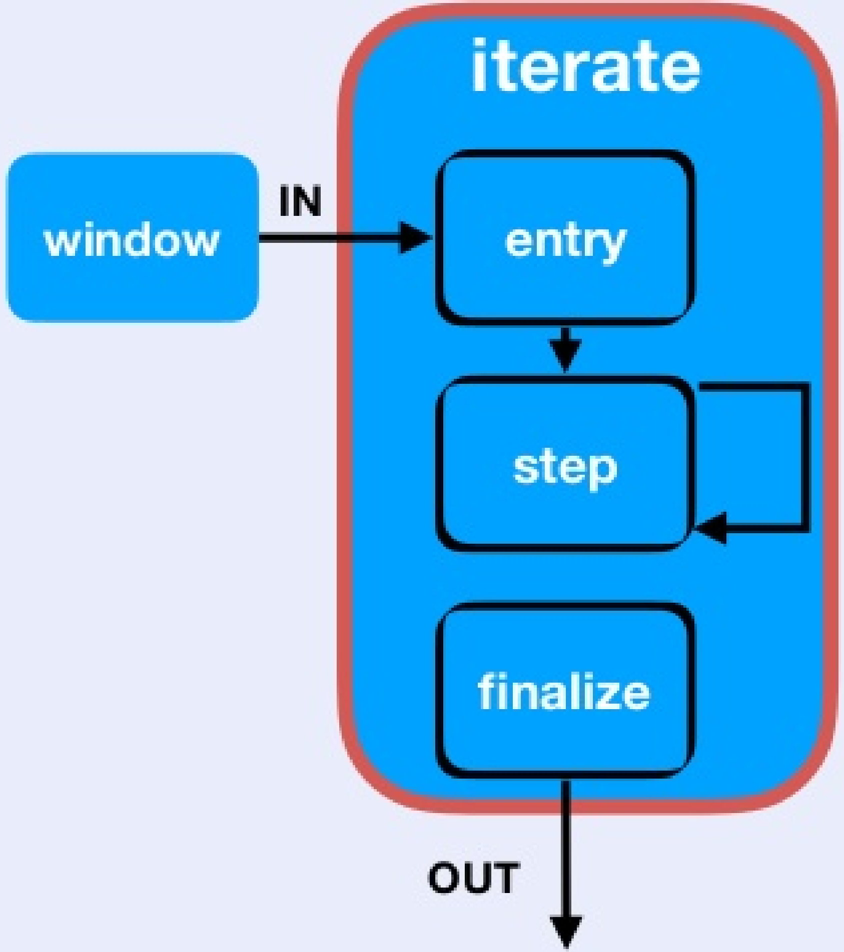



```
case class GraphEdge(from:Long, to:Long)
case class VRank(id:Long, rank:Double)

def computeRank(w:Iterable[Double]) = 0.15/w.size + 0.85 * w.sum

val input: DataStream[GraphEdge] = getEdgeStream()
input
  .flatMap(e => List(e,GraphEdge(e.to,e.from)))
  .keyBy(edge => edge.from)
  .timeWindow(30 Sec)
```

PageRank Example

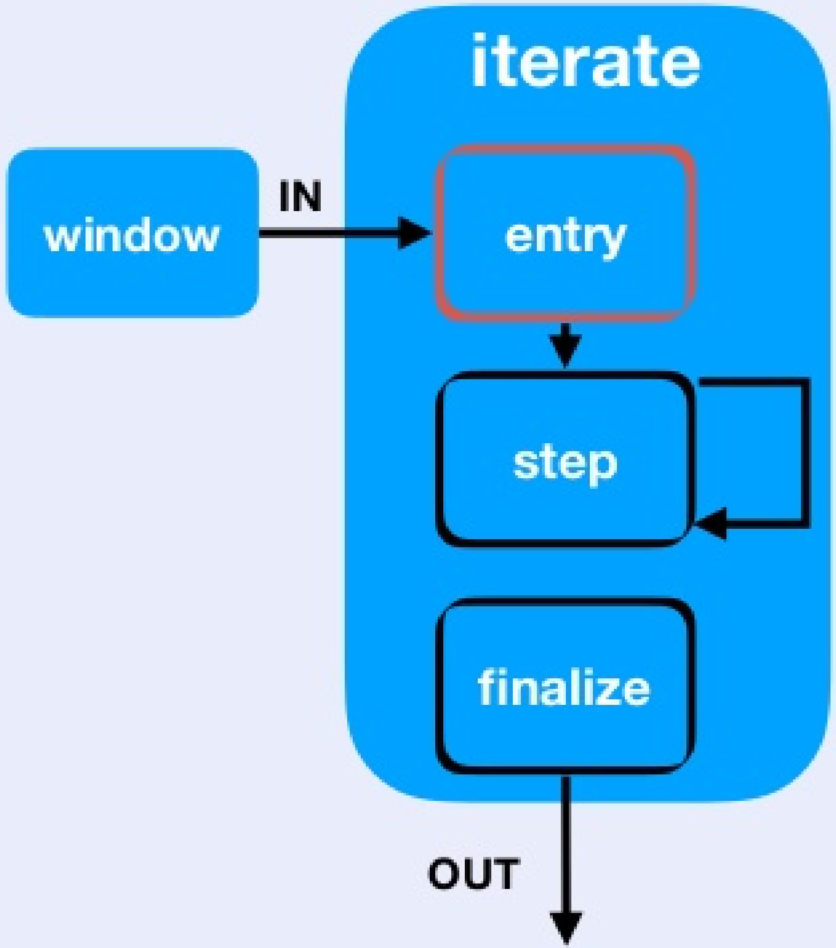



```
case class GraphEdge(from:Long, to:Long)
case class VRank(id:Long, rank:Double)

def computeRank(w:Iterable[Double]) = 0.15/w.size + 0.85 * w.sum

val input: DataStream[GraphEdge] = getEdgeStream()
input
  .flatMap(e => List(e,GraphEdge(e.to,e.from)))
  .keyBy(edge => edge.from)
  .timeWindow(30 Sec)
  .iterate(Termination.Fixed(40), Synchrony.Strict,
   (vRank => vRank.id),
```

PageRank Example

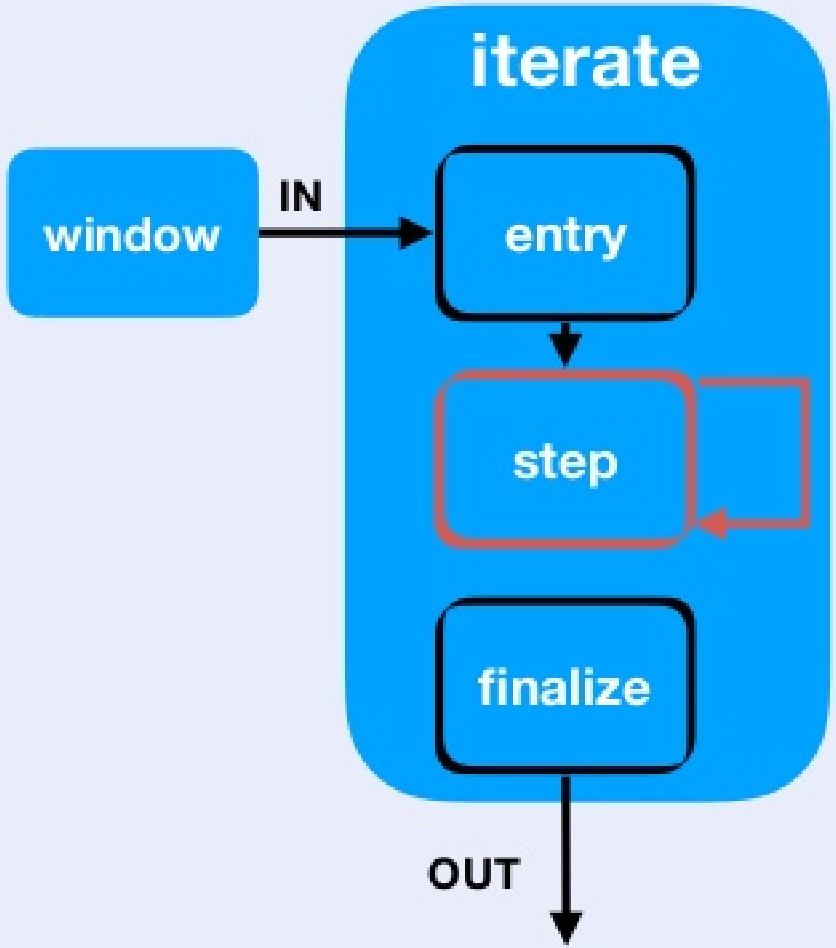



```
case class GraphEdge(from:Long, to:Long)
case class VRank(id:Long, rank:Double)

def computeRank(w:Iterable[Double]) = 0.15/w.size + 0.85 * w.sum

val input: DataStream[GraphEdge] = getEdgeStream()
input
  .flatMap(e => List(e,GraphEdge(e.to,e.from)))
  .keyBy(edge => edge.from)
  .timeWindow(30 Sec)
  .iterate(Termination.Fixed(40), Synchrony.Strict,
    (vRank => vRank.id),
    (ctx:LoopContext,input:Iterable[GraphEdge],out:Collector[VRank]) =>
      {  //ENTRY FUNCTION
         ctx.loopState("neighbors").setList(input.map(e => e.to))
         //Start from stored rank
         val rank = ctx.loopState("rank")
         rank.setValue(ctx.persistentState("rank").value)
         ctx.loopState("neighbors").foreach(n => out.collect(VRank(n,
             rank.value)))
      },
```

PageRank Example

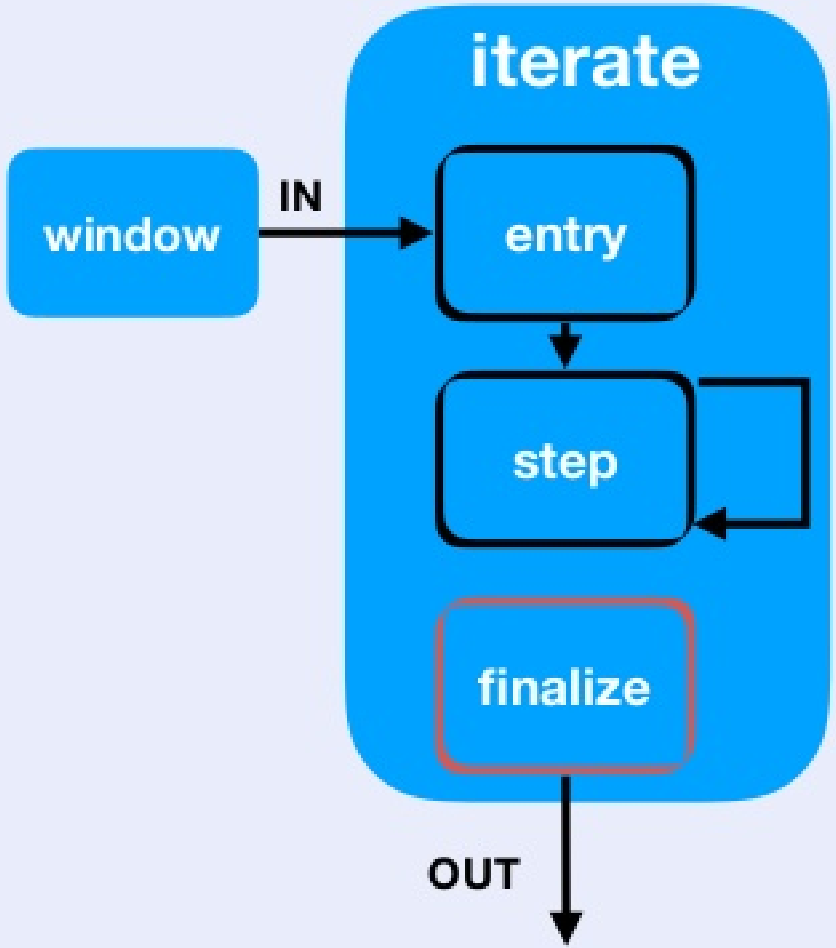



```
case class GraphEdge(from:Long, to:Long)
case class VRank(id:Long, rank:Double)

def computeRank(w:Iterable[Double]) = 0.15/w.size + 0.85 * w.sum

val input: DataStream[GraphEdge] = getEdgeStream()
input
  .flatMap(e => List(e,GraphEdge(e.to,e.from)))
  .keyBy(edge => edge.from)
  .timeWindow(30 Sec)
  .iterate(Termination.Fixed(40), Synchrony.Strict,
    (vRank => vRank.id),
    (ctx:LoopContext,input:Iterable[GraphEdge],out:Collector[VRank]) =>
      {  //ENTRY FUNCTION
         ctx.loopState("neighbors").setList(input.map(e => e.to))
         //Start from stored rank
         val rank = ctx.loopState("rank")
         rank.setValue(ctx.persistentState("rank").value)
         ctx.loopState("neighbors").foreach(n => out.collect(VRank(n,
             rank.value)))
      },
    (ctx:LoopContext,input:Iterable[VRank],out:Collector[VRank]) =>
      {  //STEP FUNCTION
         val newRank = computeRank(input.map(c => c.rank))
         ctx.loopState("rank").setValue(newRank)
         ctx.loopState("neighbors").foreach(n => out.collect(VRank(n,
             newRank)))
      },
```

PageRank Example




```scala
case class GraphEdge(from:Long, to:Long)
case class VRank(id:Long, rank:Double)

def computeRank(w:Iterable[Double]) = 0.15/w.size + 0.85 * w.sum

val input: DataStream[GraphEdge] = getEdgeStream()
input
  .flatMap(e => List(e,GraphEdge(e.to,e.from)))
  .keyBy(edge => edge.from)
  .timeWindow(30 Sec)
  .iterate(Termination.Fixed(40), Synchrony.Strict,
    (vRank => vRank.id),
    (ctx:LoopContext,input:Iterable[GraphEdge],out:Collector[VRank]) =>
      {  //ENTRY FUNCTION
         ctx.loopState("neighbors").setList(input.map(e => e.to))
         //Start from stored rank
         val rank = ctx.loopState("rank")
         rank.setValue(ctx.persistentState("rank").value)
         ctx.loopState("neighbors").foreach(n => out.collect(VRank(n,
              rank.value)))
      },
    (ctx:LoopContext,input:Iterable[VRank],out:Collector[VRank]) =>
      {  //STEP FUNCTION
         val newRank = computeRank(input.map(c => c.rank))
         ctx.loopState("rank").setValue(newRank)
         ctx.loopState("neighbors").foreach(n => out.collect(VRank(n,
              newRank)))
      },
    (ctx:LoopContext,out:Collector[VRank]) =>
      {  //FINALIZE FUNCTION
         //Log new rank
         val finalRank = ctx.loopState("rank").value
         ctx.persistentState("rank").setValue(finalRank)
         out.collect(VRank(ctx.key, finalRank))
      });
```

Higher-Level Abstractions

- Vertex-Centric Model (Part of experimental Gelly-Streams lib)

Example:
single source shortest paths

```
//Stream of edges with nil (empty) state and vertices with 'Long' state
val input: EdgeStream<Long,Nil> edges = getEdgeStream()
//iterations on directed graph snapshots defined over a 5min tumbling
    window
input
  .snapshot(Minutes(5), Direction.DIRECTED)
  .runFixpoint(
  (ctx:VertexContext) => //VERTEX COMPUTATION
        {  //vertex computation
           if(ctx.superstep == 0 && ctx.isSource()){
              ctx.setSnapshotState(0l)
              ctx.neighbors.foreach(vertex => vertex.send(Message(1l)))
           }
           else{
              dist = ctx.getVertexState().getOrElse(Long.max)
              newDist = (dist :: ctx.getMessages().values).min
              if(dist != newDist){
                 ctx.setSnapshotState(distance);
                 ctx.neighbors.foreach(vertex =>
                      vertex.send(Message(newDist+1)))
              }
           }
        });
    .toVertexStream(); // creates a DataStream<GraphVertex(distance)>
```

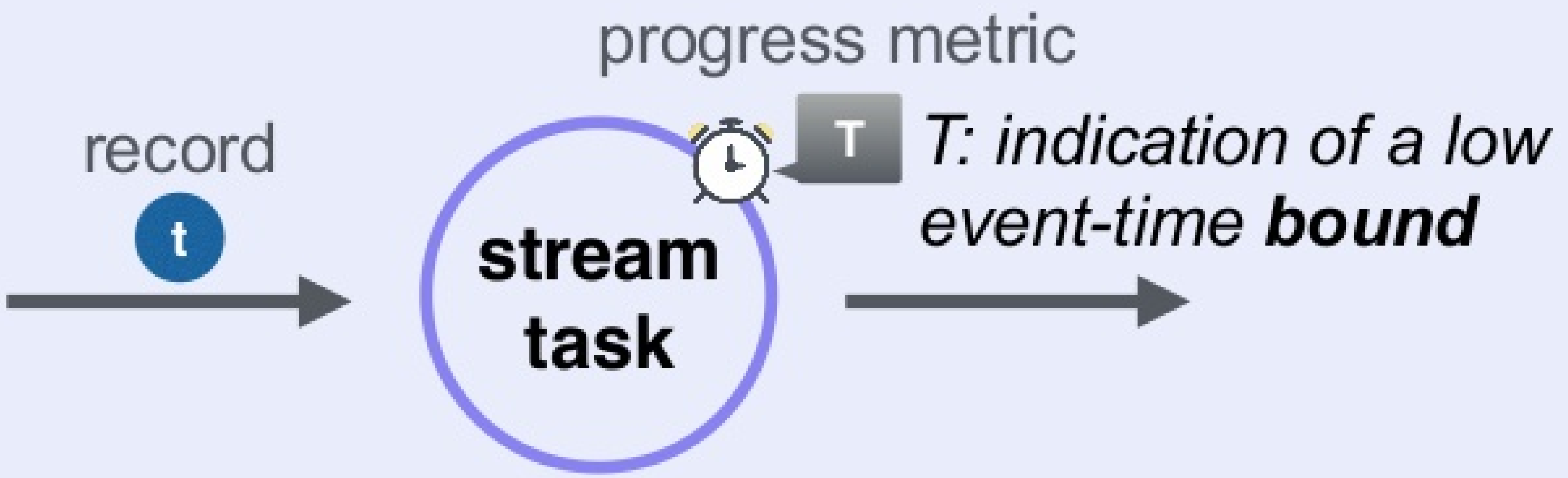
Enough about what

- We need to examine "how" to implement iterations

window computation

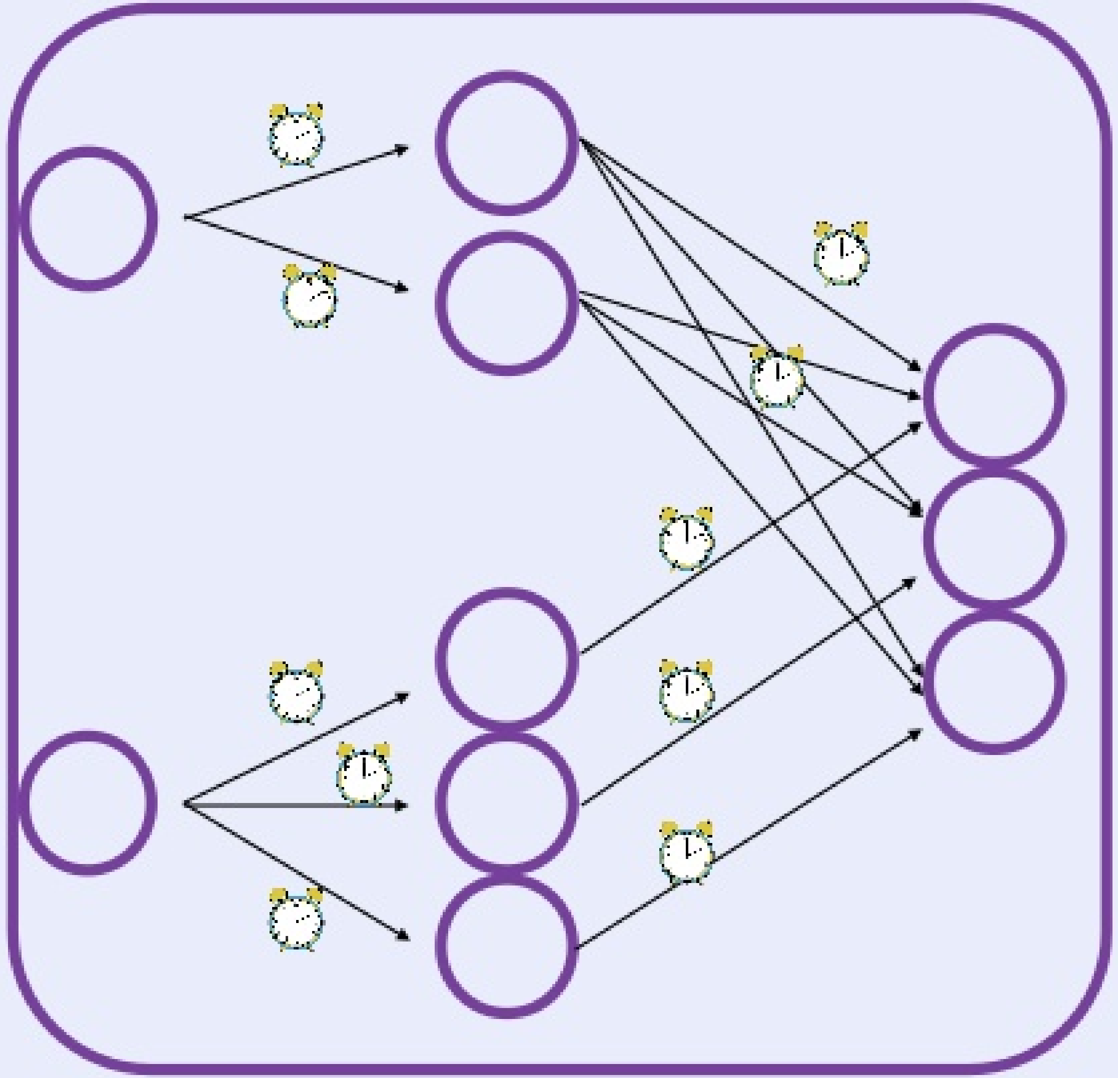
executed in **parallel** and **out-of-order** using event-time progress

How Event-Time Progress Works



- **Every record carries a timestamp t**
- **t ≥ T**
- **T** can only **increment (monotonicity)**

How Event-Time Progress Works



- **low watermarks** propagate progress metrics along the computational graph
- **progress metric = minimum** watermark across channels
- works correctly as long as monotonicity is maintained

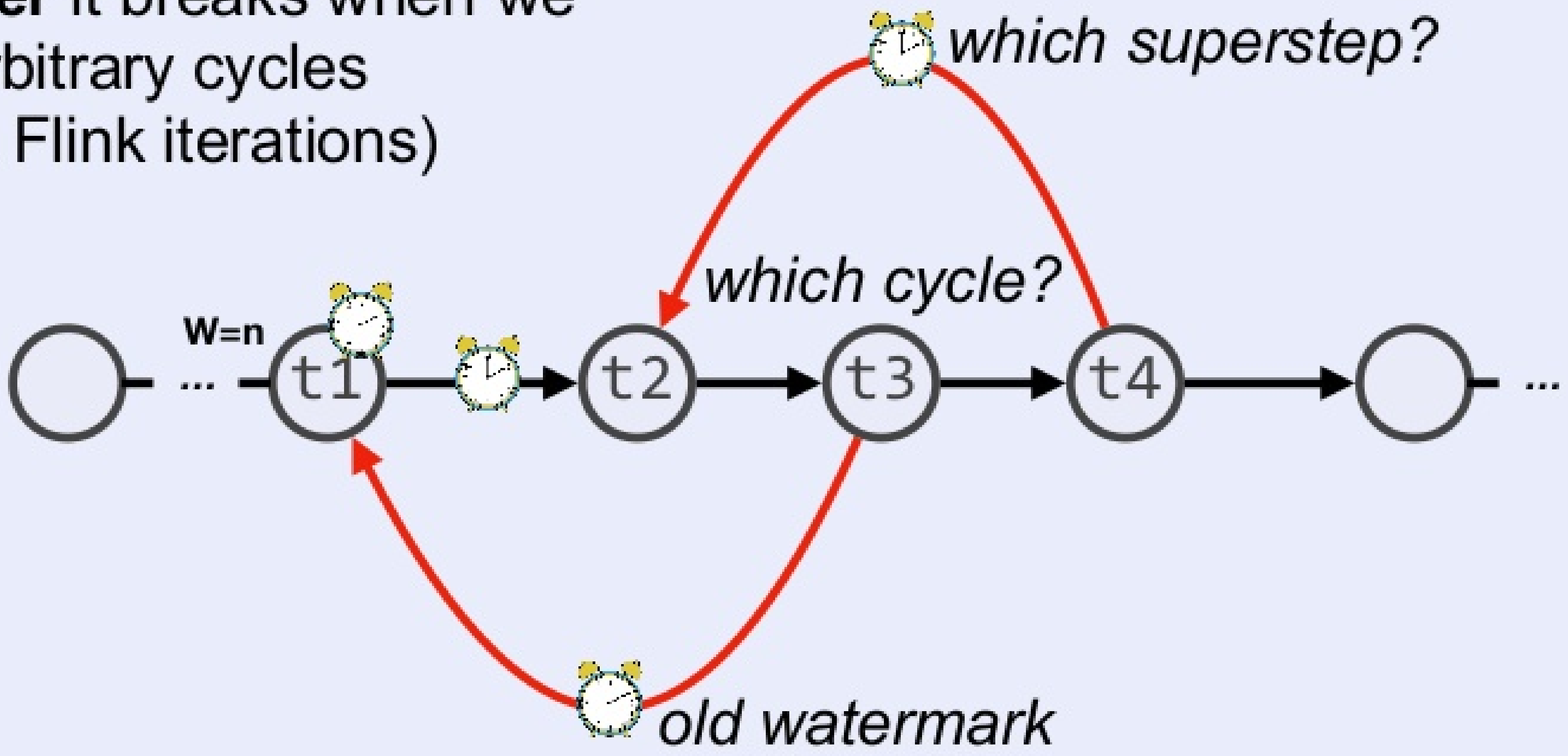
Intuition

Iteration Superstep progress bears similarities to **Event-Time** progress

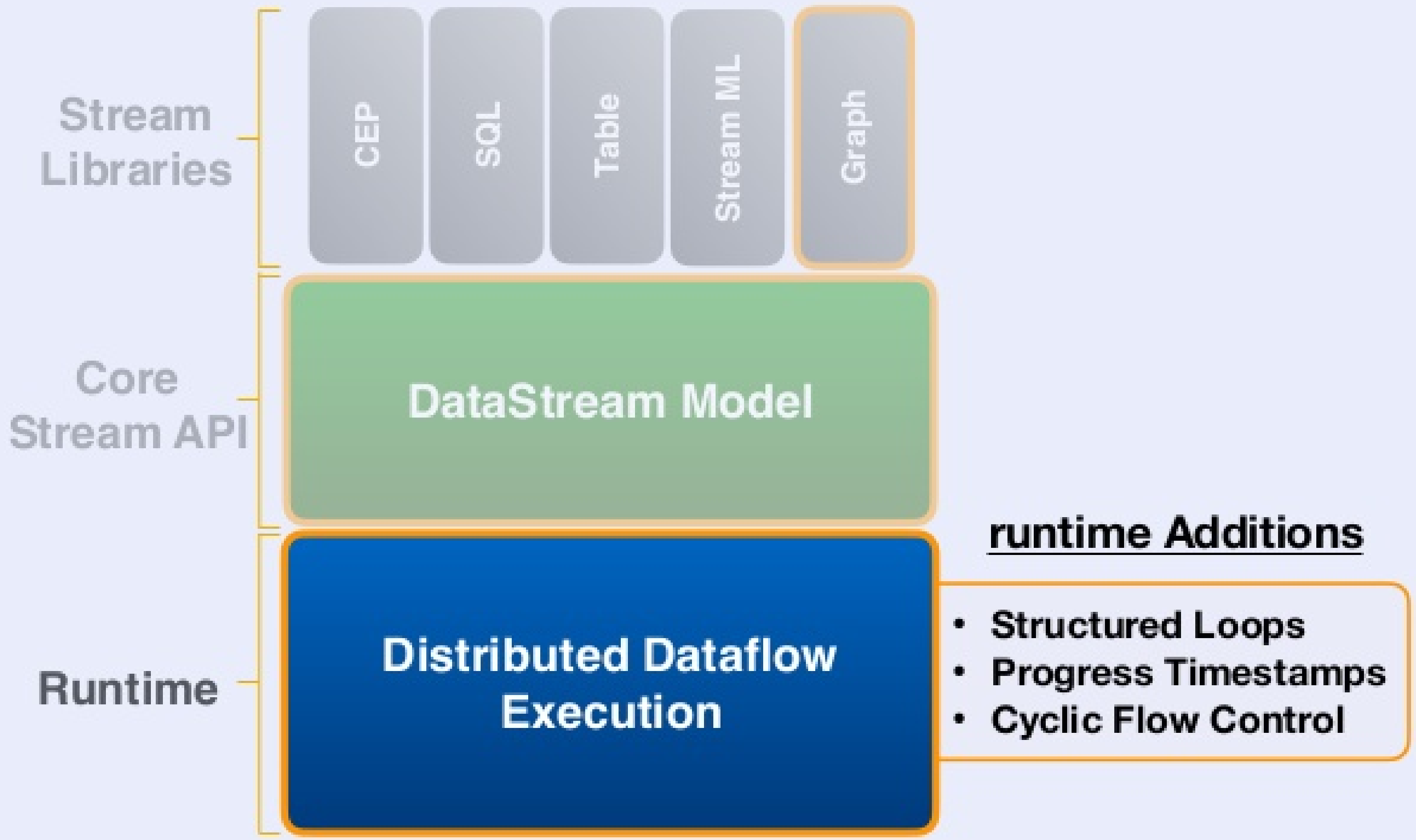
1. Can be used to run computation for out-of-order streams.
2. Need for a decentralised mechanism (no coordinator/scheduler).
3. Only relevant to respective parts of the graph (i.e., sources cannot make use of sink progress)

Watermarks and Unstructured Loops

- **Low watermarking** is a very powerful mechanism to measure and propagate progress of a **single progress metric**.
- **However** it breaks when we have arbitrary cycles (current Flink iterations)

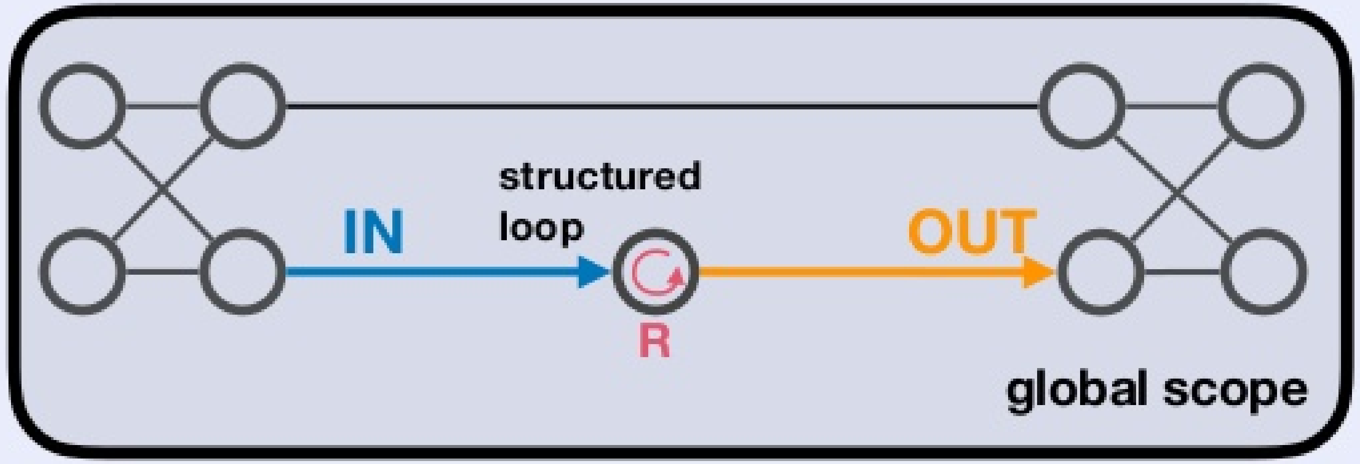


Proposed Extensions



based on Flink 1.6

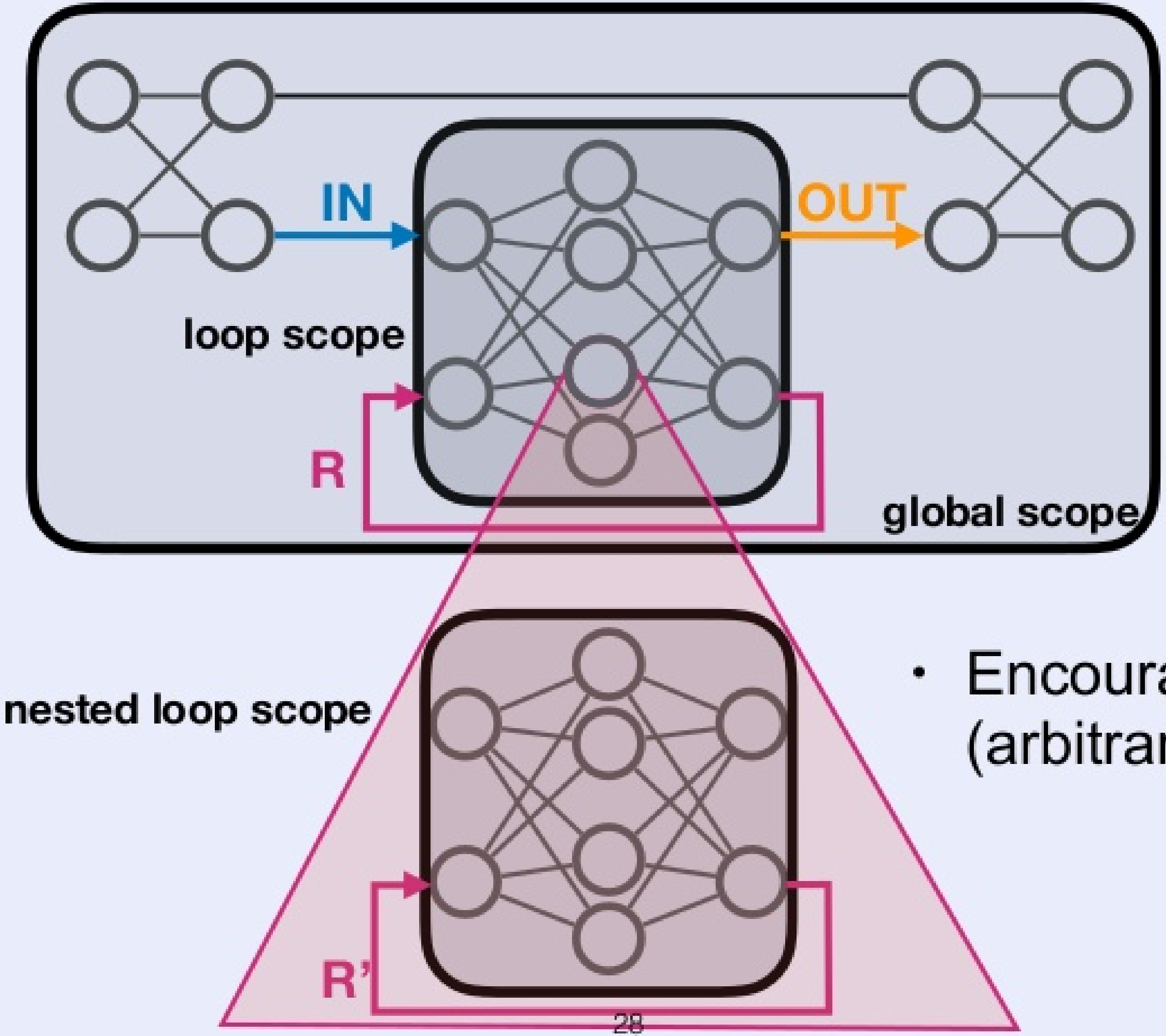
Structured Loops



- Structure loops are *low level* stream graph primitives.
- They can be seen as '**Iterative Operators**'

- Introduce the notion of "structured programming" for data streaming.
- Each **structured loop** has a **scope**.
- Each **scope** operates on its own **progress metric**.
- **Global scope** operates on **event-time progress** (**no changes made**).

Structured Loops Composition



- Encourages compositionally (arbitrary nesting)

Using Progress Timestamps

$$P = [\overbrace{p^n,}^{p^T} \underbrace{p^{n-1}, p^{n-2}, \ldots, p^0}_{P^{ctx}}]$$

Operation	**Implem.**	**Example**
progress: $\mathbb{N}^n \rightarrow \mathbb{N}$	head($\mathtt{P}$)	$\mathtt{p}_n$
unnest: $\mathbb{N}^n \rightarrow \mathbb{N}^{n-1}$	tail($\mathtt{P}$)	$[\mathtt{p}_{n-1}, \ldots, \mathtt{p}_1]$
nest: $\mathbb{N}^n \rightarrow \mathbb{N}^{n+1}$	$\mathtt{0} :: \mathtt{P}$	$[\mathtt{0}, \mathtt{p}_n, \ldots, \mathtt{t}_1]$
incr: $\mathbb{N}^n \rightarrow \mathbb{N}^n$	*head*($\mathtt{P}$)+1 :: *tail*($\mathtt{P}$)	$[\mathtt{p}_n+1, \ldots, \mathtt{p}_1]$

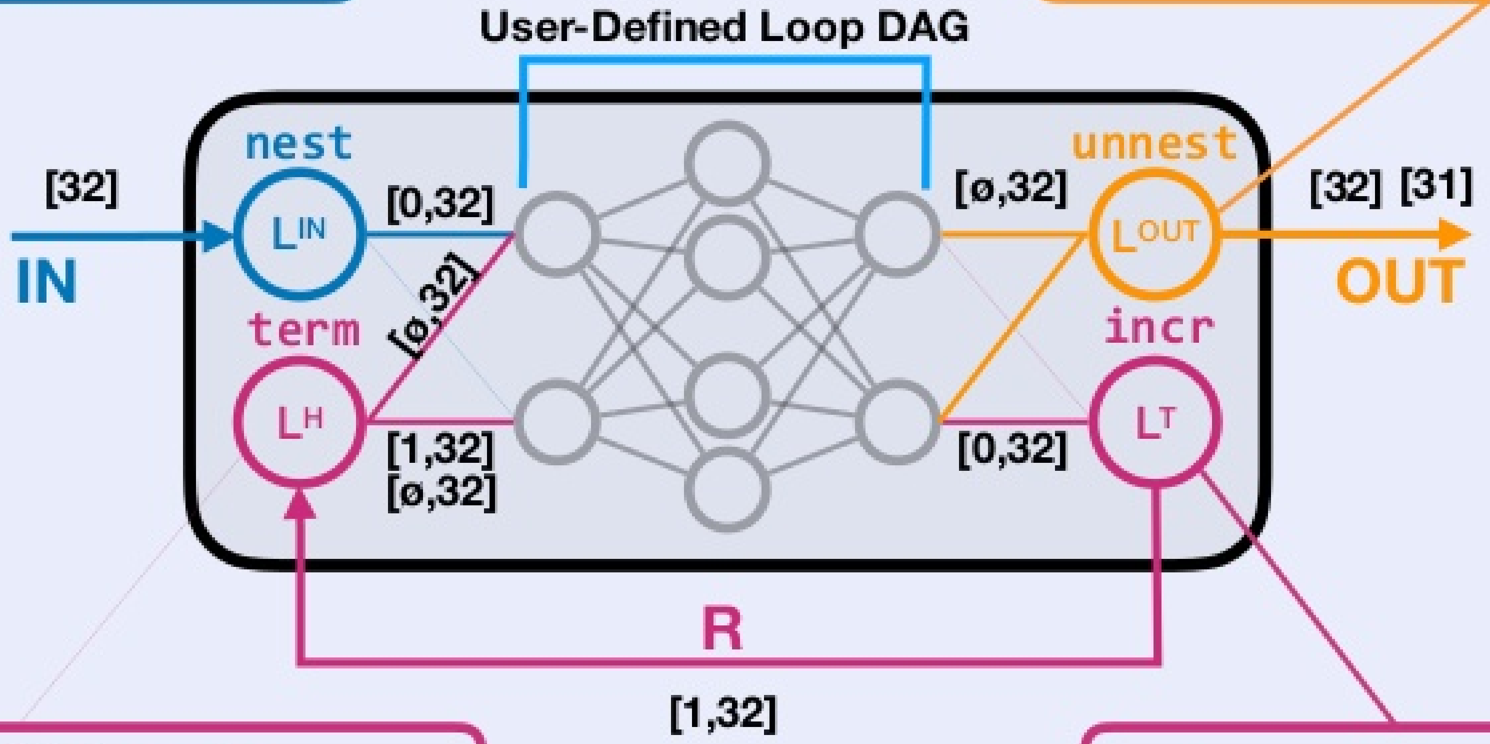
- Nests **progress timestamps**
- (e.g., last superstep p^T per window P^{ctx})

- How do we guarantee monotonic progress metrics?

Structured Loop Embeddings

- Initializes Scope's Progress Metric for both records and watermarks (`nest`)

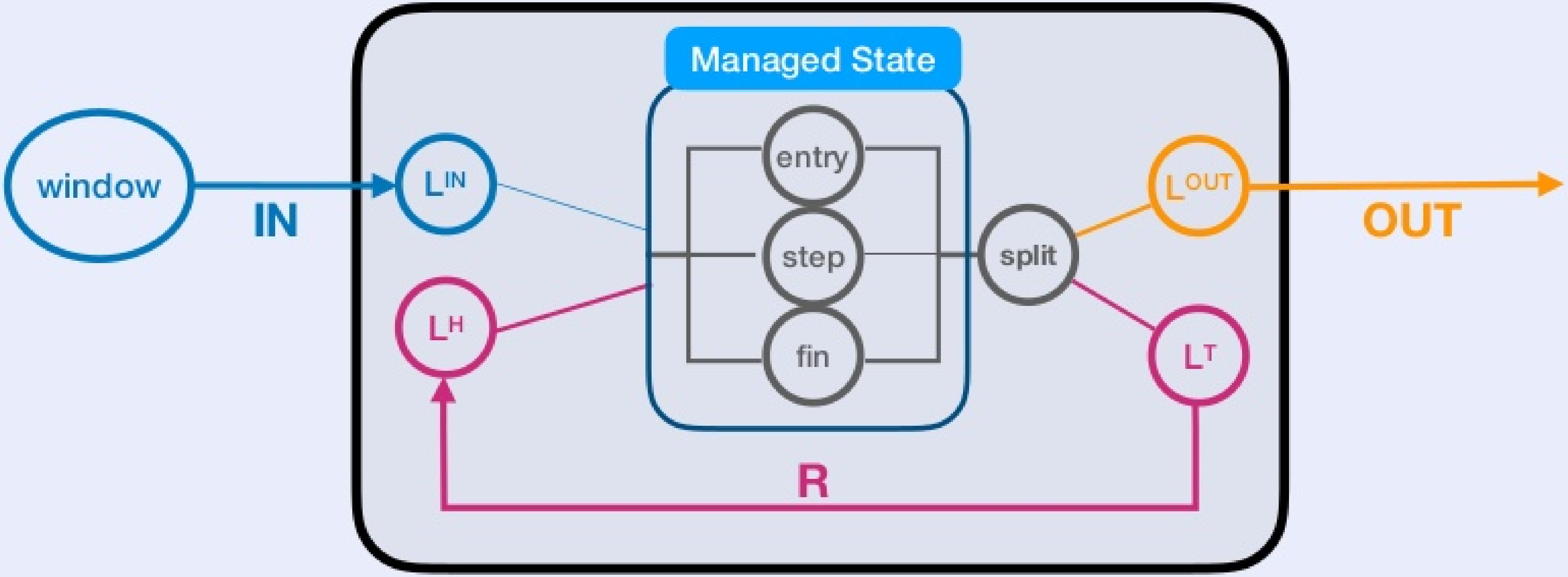
- Triggers final output of an iterative operator
- Removes inner progress metric (`unnest`)



- Forwards Records and Watermarks
- Evaluates and Disseminates Termination (ø)

- Increments Progress Metric (`incr`)

Example: Window Iteration

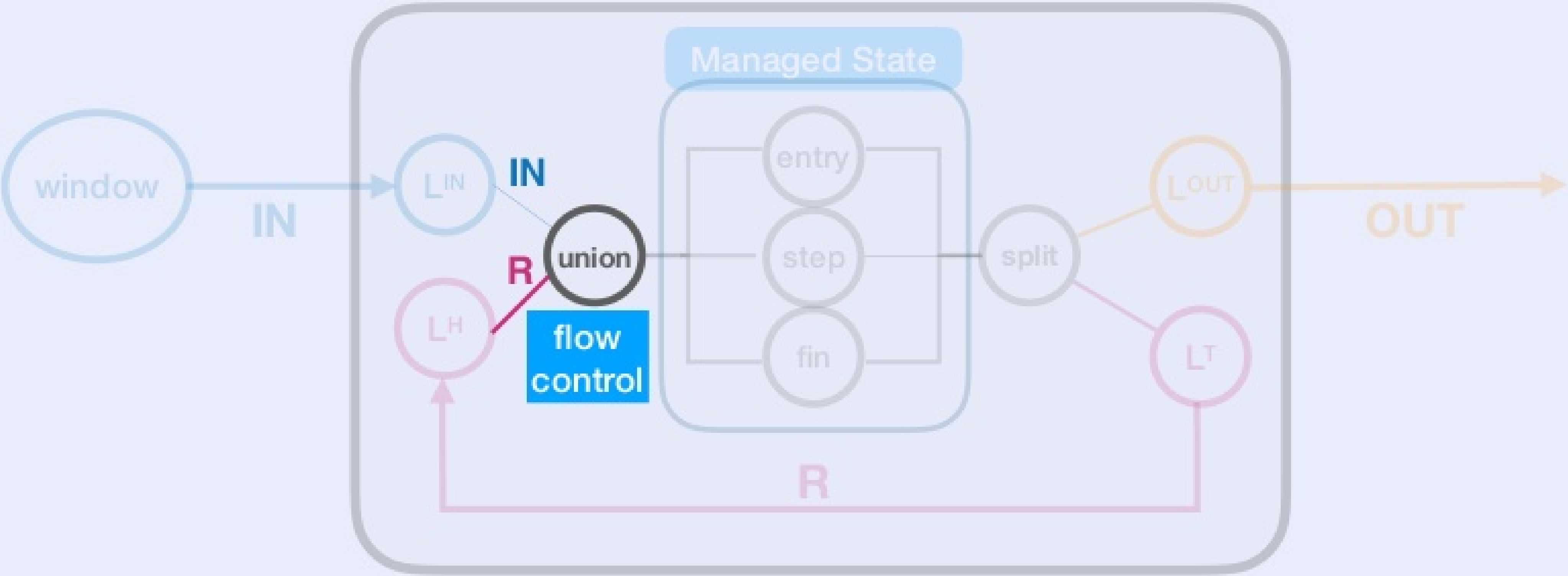


Further Challenges

what keeps us busy

- **Cyclic Flow Control**
  - **avoid deadlocks**
  - **encourage iteration completion**
- **State Management Integration**
  - **one more level of indirection (mapvalue per key)**

Cyclic Flow Control



- (Consumed Buffers < Threshold): Round-Robin between **IN** and **R**
- (Consumed Buffers >= Threshold): Exclusive Ingestion of **R**

FLIP-15

- Introduces **Scoping** and **Nesting** (for Structured Loops)
- Addresses **Flow Control** Alternatives
- Discusses Computation-agnostic Loop Termination

https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=66853132

Research Team and Sponsors

- **Marius Melzer (TUDresden)**
- **Asterios Katsifodimos (TUDelft)**
- **Vasiliki Kalavri (ETH)**
- **Seif Haridi (KTH)**
- **Pramod Bhatotia (Un.Edinburgh)**

STREAMLINE.

Swedish Foundation for Strategic Research

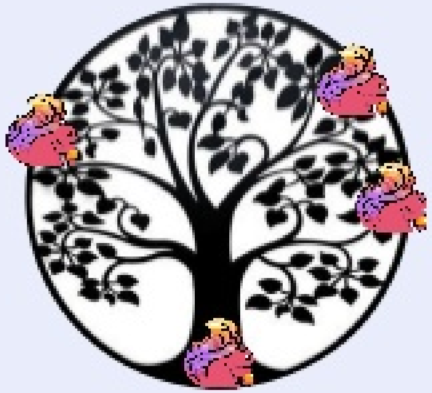
References

- **Scalable and Reliable Data Stream Processing - Paris Carbone (2018)**
- Inflight Progress Tracking for Stream Processing Systems with Cyclic Dataflows - Marius Melzer (2017)
- Naiad: a timely dataflow system - Murray, McSherry et al. (2013)
- The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing - Tyler Akidau et al. (2015)
- Out-of-order processing: a new architecture for high-performance stream systems - Li, Maier et al. (2008)
- Flexible time management in data stream systems - Srivastava, Widom (2004)

@FF2018-Berlin

Stream Loops on Flink

Reinventing the wheel for the streaming era



Paris Carbone

Systems Researcher@KTH/SICS <parisc@kth.se>
Committer@Flink <senorcarbone@apache.org>