Distributed Systems

Spring Semester 2020

Lecture 22: Big Data Systems (Naiad)

John Liagouris
liagos@bu.edu

Why Naiad?

- Batch and stream processing
- Cyclic dataflows (even nested iterations)
- Efficient incremental computations (Differential Dataflow)
- Impressive performance results (low latency - high throughput)

PageRank

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
             .mapValues(sum => a/N + (1-a)*sum)
}
```

PageRank

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (
  // with the contribu
  val contribs = links
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
                 .mapValues(sum => a/N + (1-a)*sum)
}
```

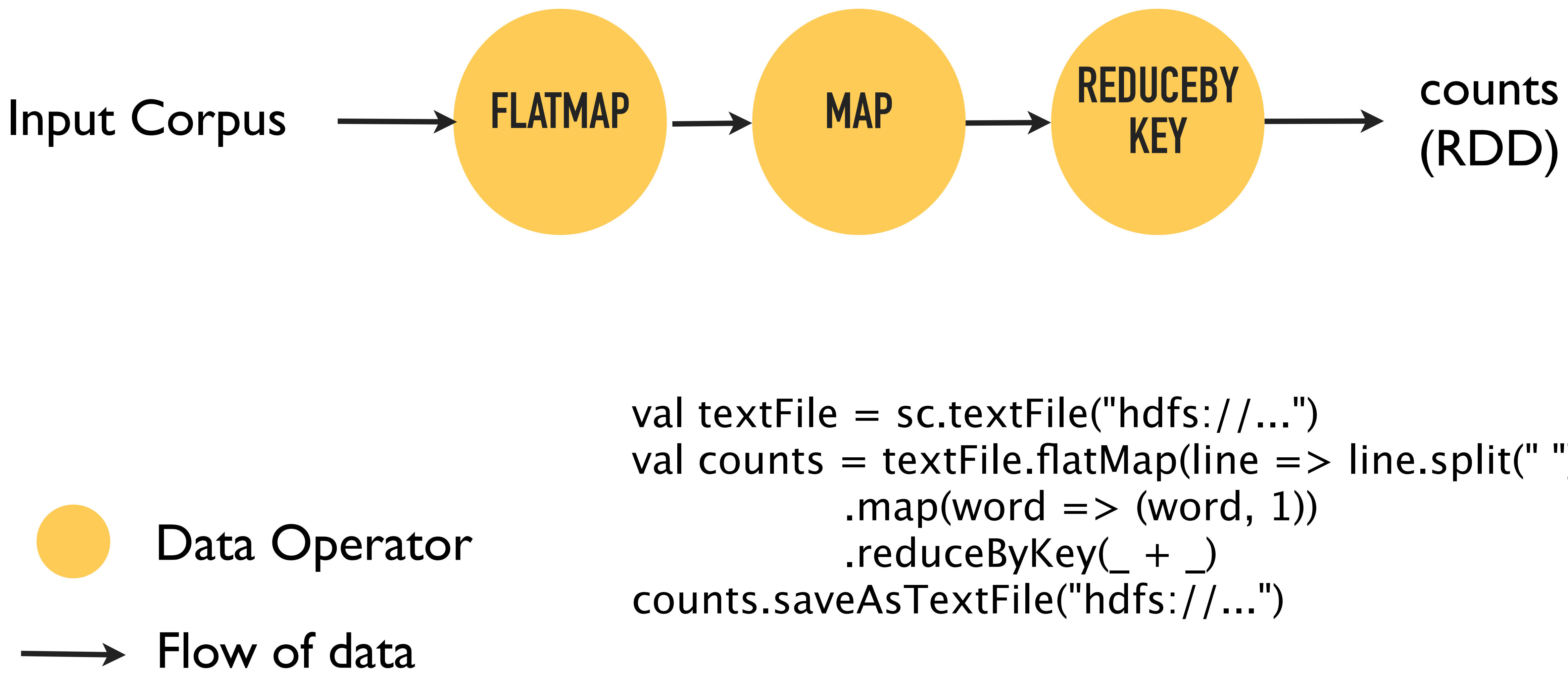
What if new pages
are added to the input file?

PageRank

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contri                       latMap {
    (url, (l
      links.                       links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
              .mapValues(sum => a/N + (1-a)*sum)
}
```

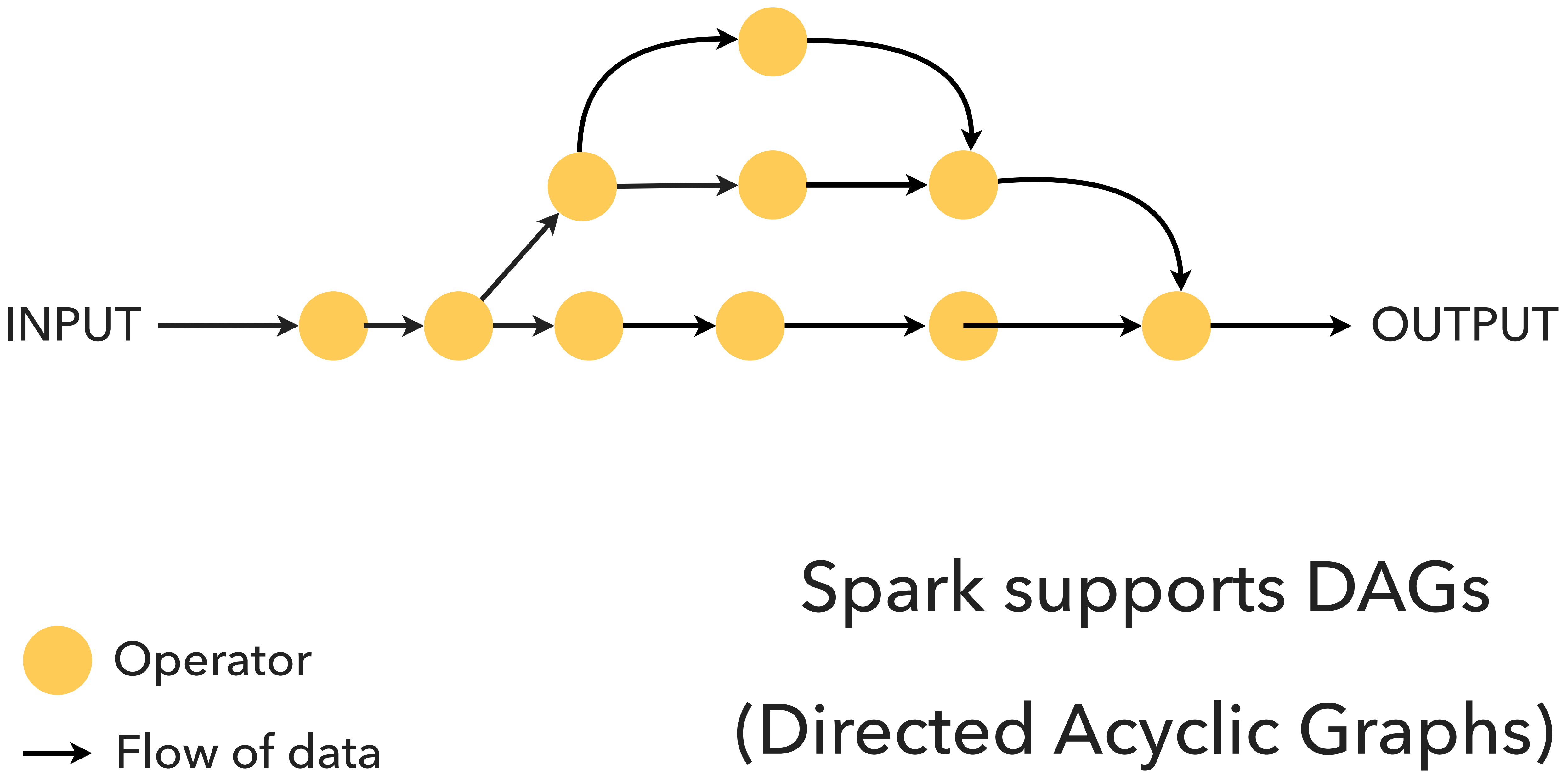
Start all over again!

Dataflow Graphs




```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

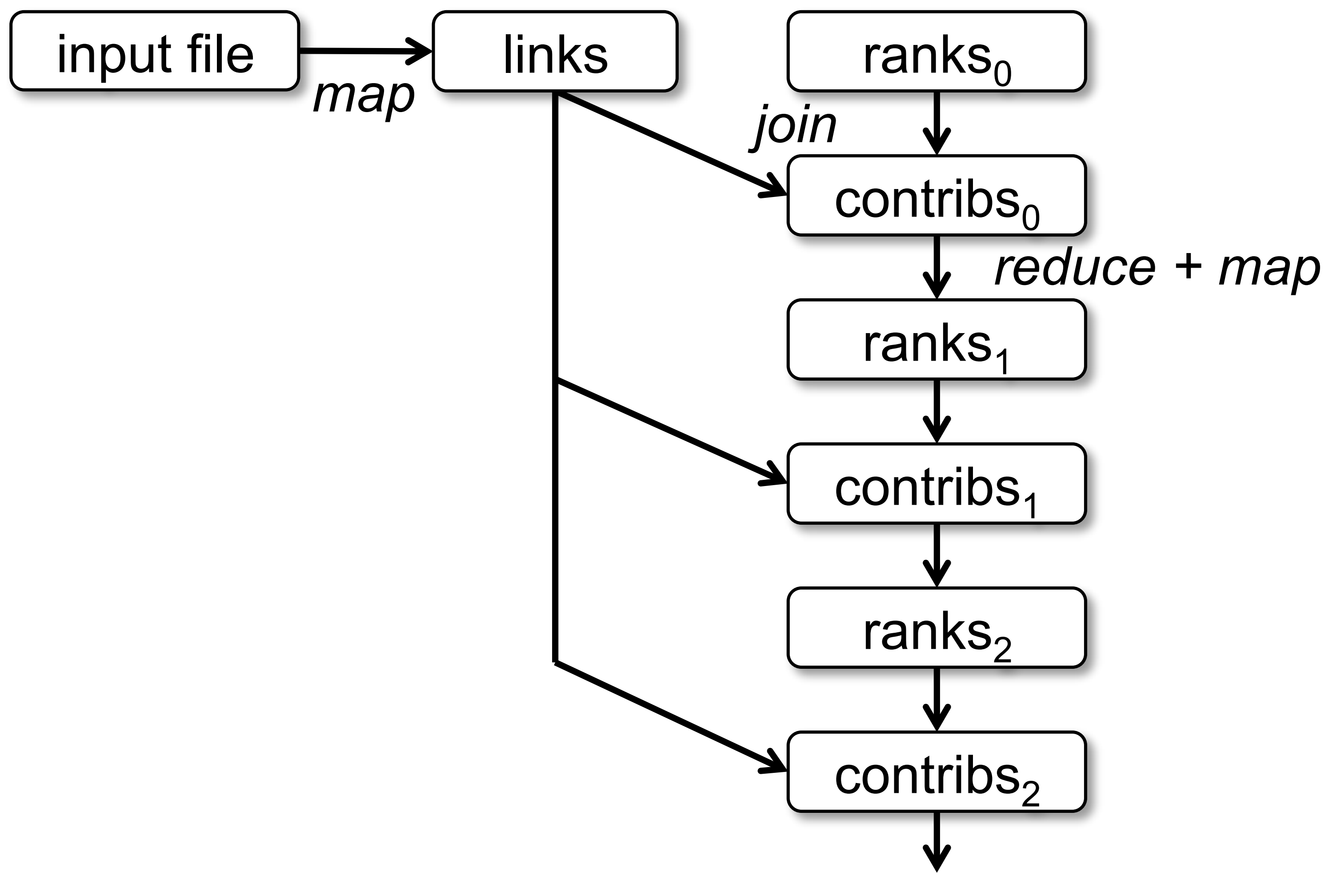
Dataflow Graphs



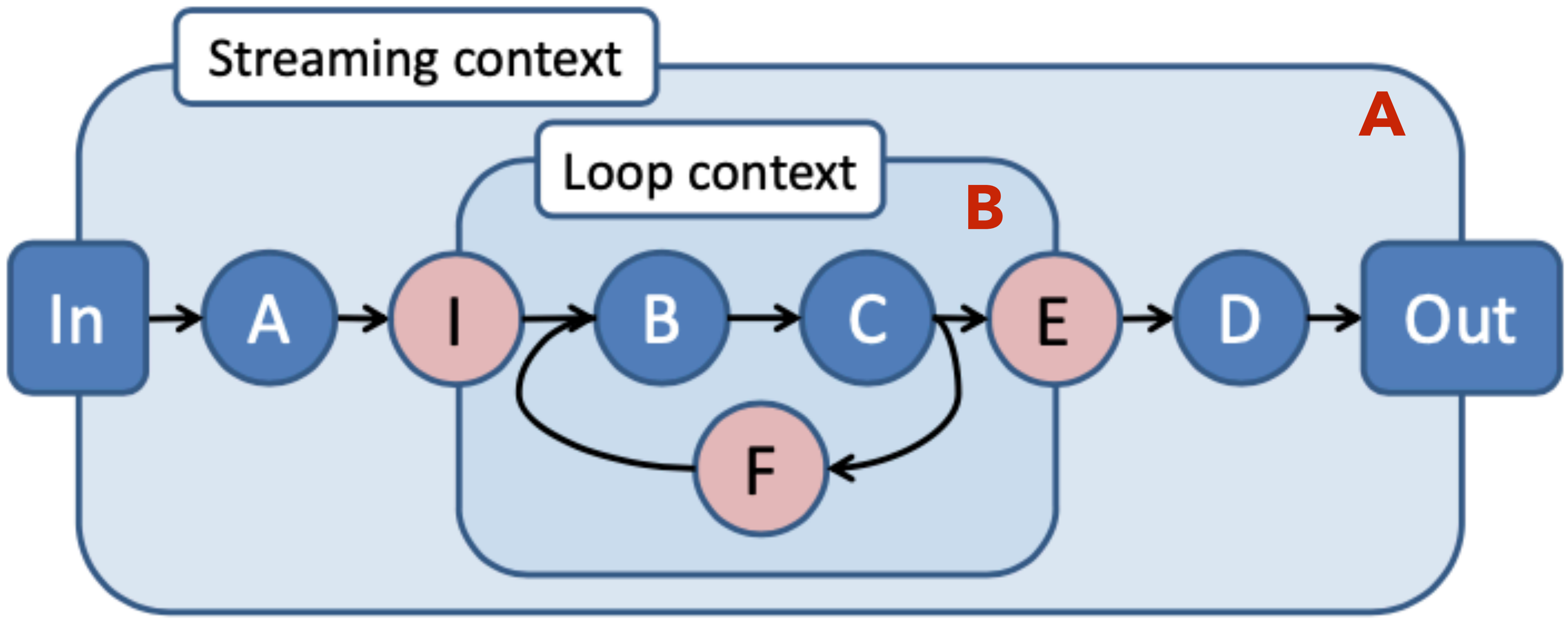
Spark supports DAGs

(Directed Acyclic Graphs)

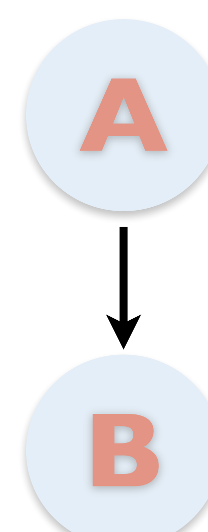
Spark "unfolds" loops



Dataflow Graphs in Naiad



Contexts form a hierarchy

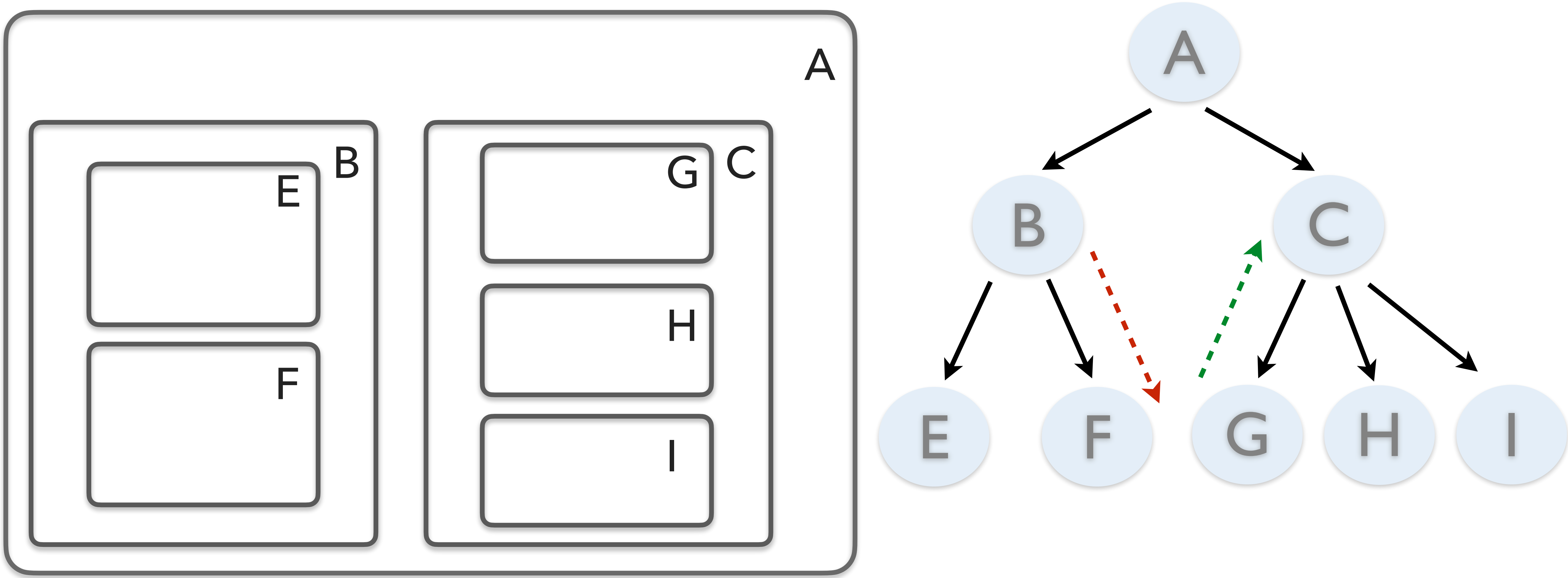


I: Ingress (enter a context)

E: Egress (exit a context)

F: Feedback (feedback loop)

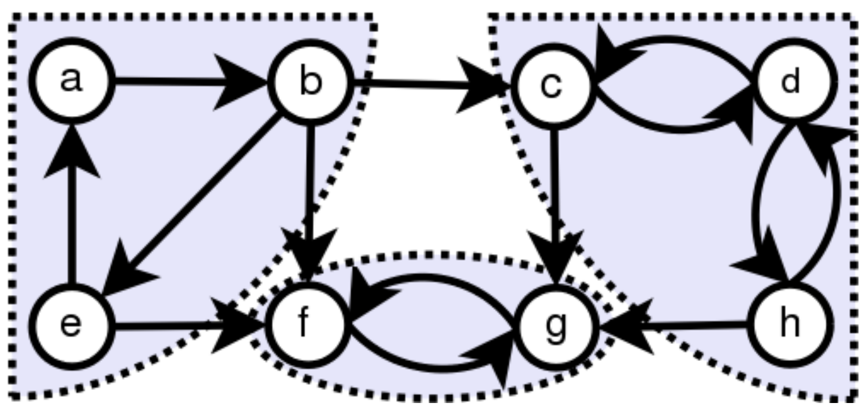
Context Hierarchy

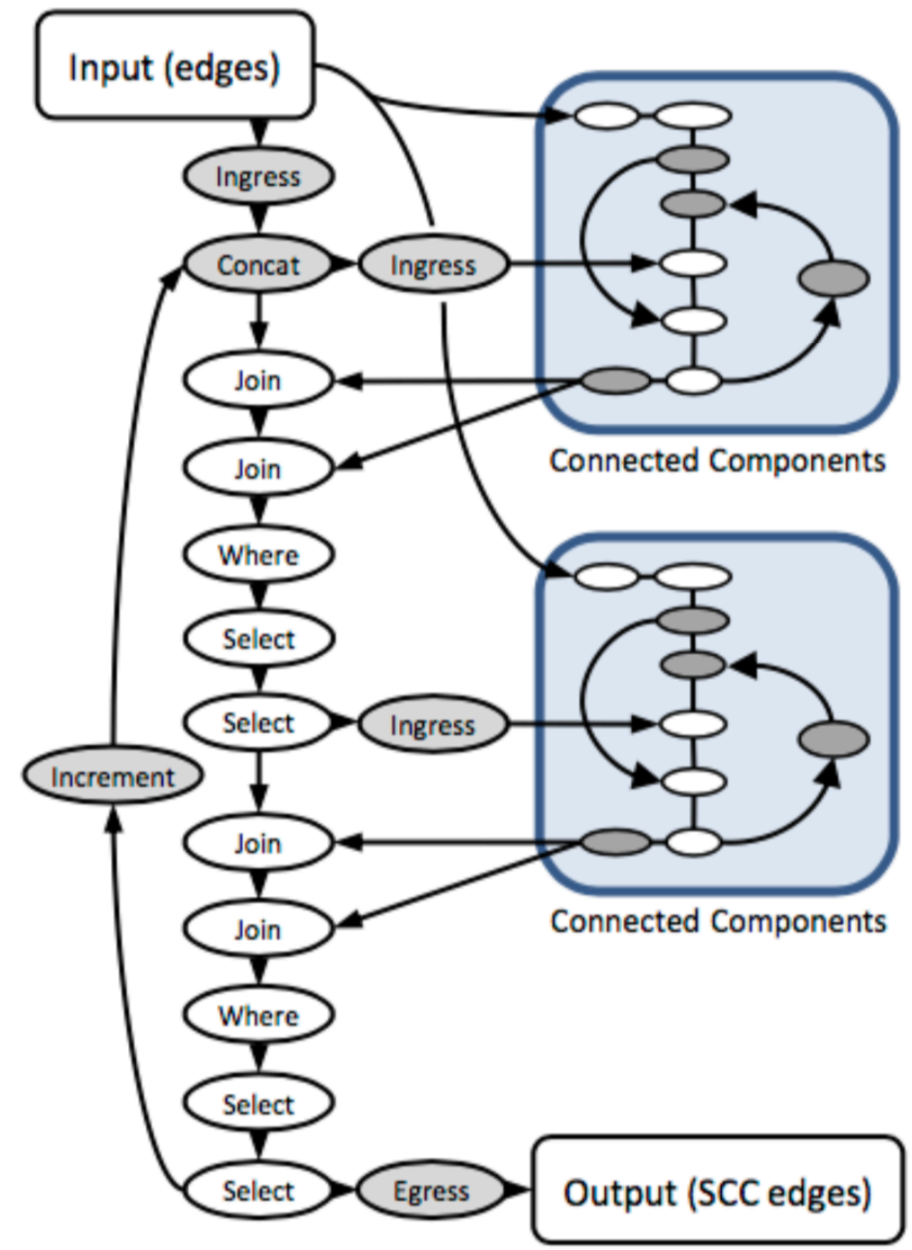


A record going through an ingress (resp. egress) node in the data flow graph is going “down” (resp. “up”) in the context tree

Graph Processing in Naiad

Strongly Connected Components





Credits: Frank McSherry

Logical Timestamps

$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}},\ \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$

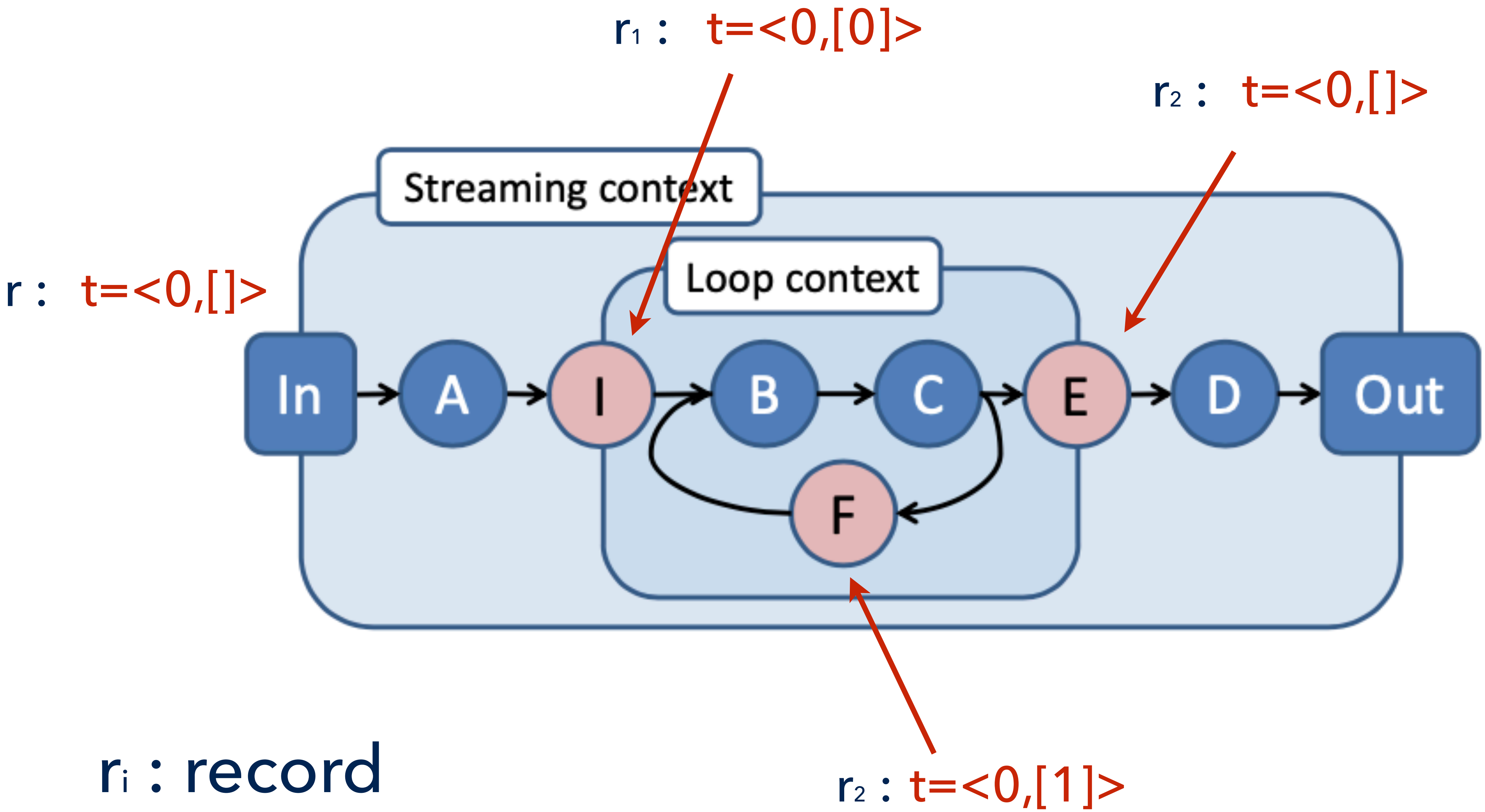
- Used to track progress of the computation
- Epoch generated by the source (“round of data”)
- Each loop counter counts the number of times a record has gone through a specific loop
- Maximum number of loop counters depends on the depth of context tree — in practice 1 or 2

Actions on Timestamps

Vertex	Input timestamp	Output timestamp
Ingress		$,0\rangle)$
Egress		$\rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$

Actions on Timestamps



r_i : record

Progress Tracking

- Protocol to know when to deliver notifications to vertices
- Notification for a timestamp t: vertex won't see any more records with this timestamp in its input(s)
- Intuition: deliver notification for t when it is impossible for predecessors to generate a timestamp earlier than t
- Distributed protocol — requires broadcasts

Low-level API

$$this.\textsc{SendBy}(e : \text{Edge},\ m : \text{Message},\ t : \text{Timestamp})$$
$$this.\textsc{NotifyAt}(t : \text{Timestamp}).$$

- SENDBY: vertex sends data message to another vertex
- NOTIFYAT: vertex requires notification at timestamp t

Low-level API

$this.\text{SENDBY}(e : \text{Edge},\ m : \text{Message},\ t : \text{Timestamp})$
$this.\text{NOTIFYAT}(t : \text{Timestamp}).$

vertex u

Edge: (u,v)

the actual data

- SENDBY: vertex sends data message to a vertex
- NOTIFYAT: vertex requires notification for timestamp t

Low-level API

$$v.\textsc{OnRecv}(e : \text{Edge},\ m : \text{Message},\ t : \text{Timestamp})$$
$$v.\textsc{OnNotify}(t : \text{Timestamp}).$$

- ONRECEIVE: vertex v receives data message from a vertex
- ONNOTIFY: vertex v receives a notification for timestamp t

Low-level API

$$v.\textsc{OnRecv}(e : \text{Edge},\ m : \text{Message},\ t : \text{Timestamp})$$
$$v.\textsc{OnNotify}(t : \text{Timestamp}).$$

vertex v

Edge: (u,v)

- ONRECEIVE: vertex v receives data message from a vertex
- ONNOTIFY: vertex v receives a notification for timestamp t

Progress Tracking

$$\text{Pointstamp} : (t \in \text{Timestamp}, \overbrace{l \in \text{Edge} \cup \text{Vertex}}^{\text{location}})$$

- Each event (notification or data message) has a timestamp t and a location ℓ — "pointstamp"
- Each time an event happens its pointstamp becomes "active"
- For an active pointstamp *p*, the system maintains:
  - An <u>occurrence count (OC)</u>: how many active events of the same type (message or notification) have the same pointstamp *p*
  - A <u>precursors count (PC)</u>: how many active events "could result in" an event with this pointstamp *p*

Progress Tracking

$this.\textsc{SendBy}(e : \text{Edge},\ m : \text{Message},\ t : \text{Timestamp})$
$this.\textsc{NotifyAt}(t : \text{Timestamp}).$

$v.\textsc{OnRecv}(e : \text{Edge},\ m : \text{Message},\ t : \text{Timestamp})$
$v.\textsc{OnNotify}(t : \text{Timestamp}).$

- SENDBY and NOTIFYAT generate "active" events — increase counters
- ONRECEIVE and ONNOTIFY consume "active" events — decrease counters

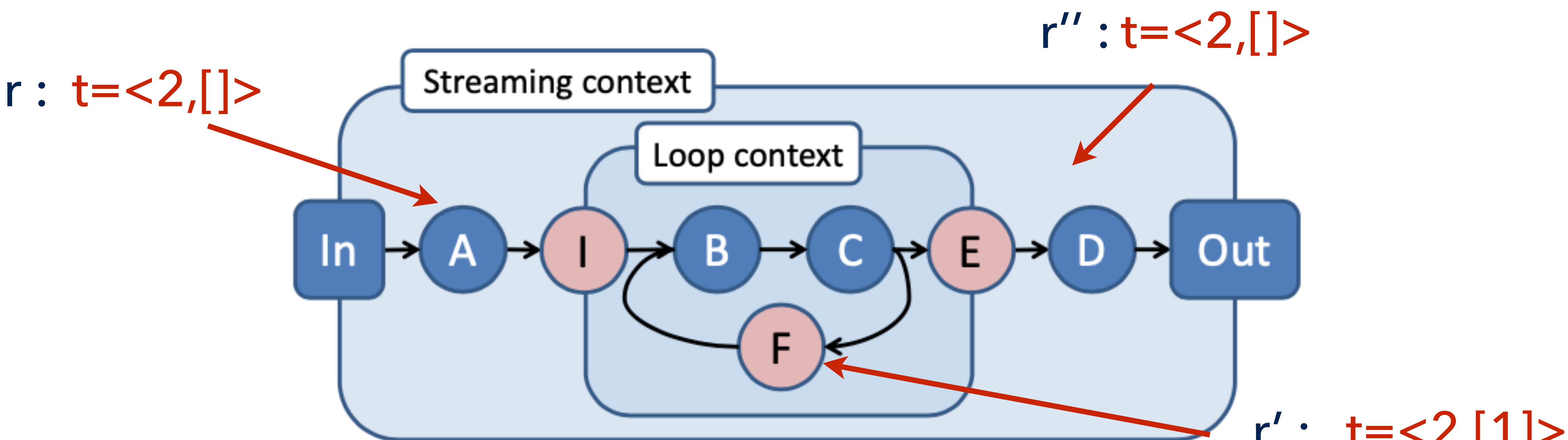
Progress Tracking

- A pointstamp *p* leaves the set of "active" pointstamps when its occurrence count becomes zero
- When a pointstamp *p* leaves the active set, the system decreases the precursor count for all active pointsamps *p* could result in
- Frontier: the set of active pointstamps with precursor counters equal to zero
- How do check if a pointstamp "could result in" another pointstamp?

Path Summaries

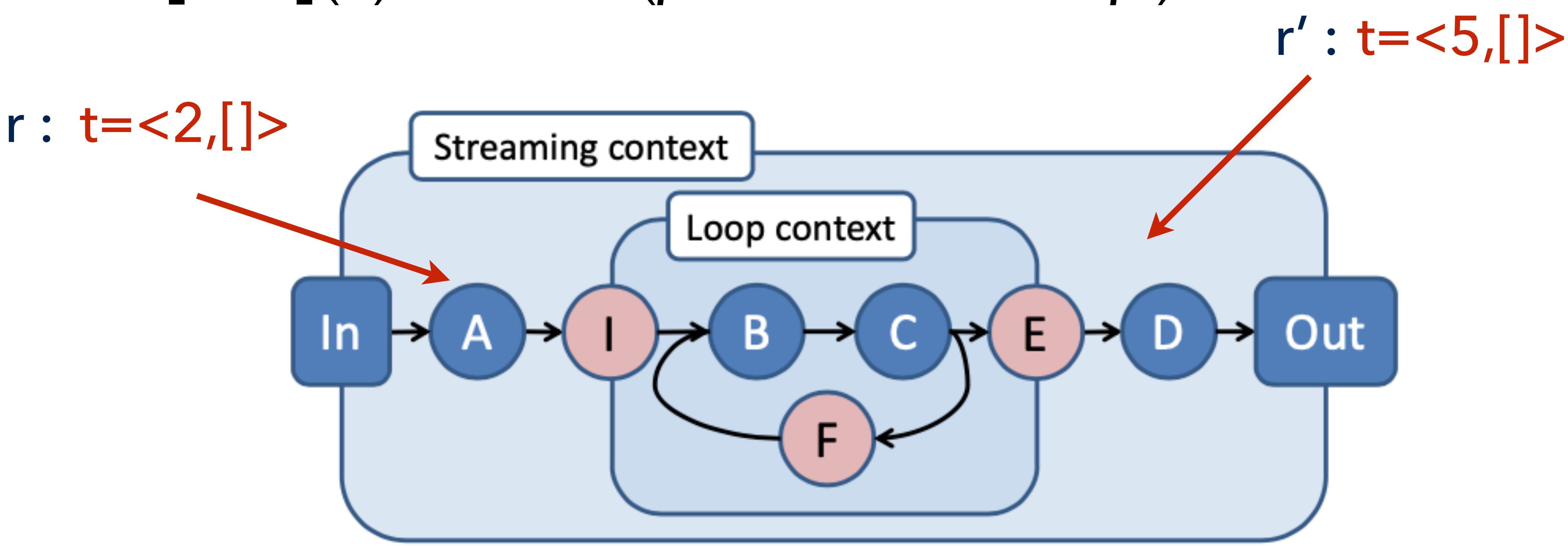
- Functions that describe what should happen to an active event if it goes through a path in the dataflow graph
- Path summary for the path A → B → C → B → C → D: IFE
- For two pointsamps $p=(t,\ell)$ and $p'=(t',\ell')$:

  p “could result in” p' iff PS[ℓ,ℓ'](t) <= t’



Path Summaries

- PS for the path A $\rightarrow$ B $\rightarrow$C $\rightarrow$B $\rightarrow$C $\rightarrow$D: IFE
  - Record r generated by A $\rightarrow$ p = (<2,[]>, A)
  - Record r′ generated by D $\rightarrow$ p' = (<5,[]>,D)
  - PS[A,D](2) = 2 < 5 (p "can result in" p')



Fault-tolerance

- Synchronous checkpoints: Stop execution → take checkpoint → resume
  - User observes latency spikes at steady state
  - If failure happens, system "rolls back" to the last checkpoint
- Alternative approach would be to log all events to disk before sending them - Write-ahead logging
  - Much higher latency per-record - expensive I/Os
- Lineage-based fault-tolerance (Spark) difficult to implement in Naiad
  - Need to keep track of lineage information per record — too many dependencies to maintain

References

- Lineage Stash: https://cs-people.bu.edu/liagos/material/sosp19.pdf
  - Lineage-based fault-tolerance for record-at-a-time execution
- Timely Dataflow: https://github.com/TimelyDataflow/timely-dataflow
  - Rust prototype of Naiad
- Differentiasl Dataflow: https://github.com/TimelyDataflow/differential-dataflow
  - Incremental computation on Timely Dataflow
- Materialize Inc: https://materialize.io
  - Streaming Data Warehouse on Timely/Differential Dataflow