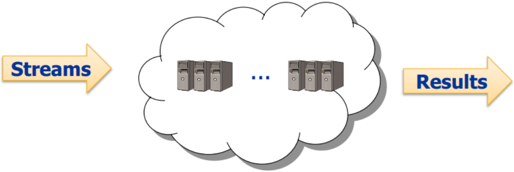
Scalable Stream Processing
Storm, SEEP and Naiad

Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology

Motivation

- Users of big data applications expect fresh results.
- New stream processing systems (SPS) are designed to scale to large numbers of machines.
- SPS design issues (reminder):
  - SPS data flow: composiation and manipulation
  - SPS runtime: parallelization, fault-tolerance, optimization



Outline

- Storm
- SEEP
- Naiad

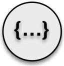
Storm

Contribution

- Storm is a real-time distributed stream data processing engine at Twitter.

Data Model (1/3)

- Tuple
  - Core unit of data.
  - Immutable set of key/value pairs.



- Stream
  - Unbounded sequence of tuples.

Data Model (2/3)

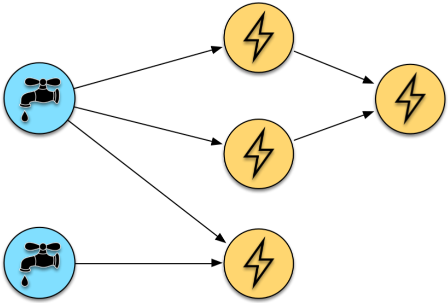
- Spouts
  - Source of streams.
  - Wraps a streaming data source and emits tuples.

- Bolts
  - Core functions of a streaming computation.
  - Receive tuples and do stuff.
  - Optionally emit additional tuples.

Data Model (3/3)

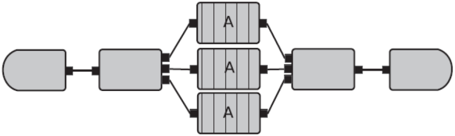
- Topology
  - DAG of spouts and bolts.
  - Data flow representation streaming computation

- Storm executes spouts and bolts as individual tasks that run in parallel on multiple machines.



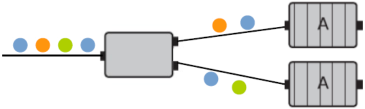
Parallelisation (1/3)

- Data parallelism



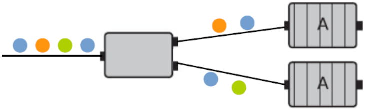
Parallelisation (2/3)

- Shuffle grouping: randomly partitions the tuples.

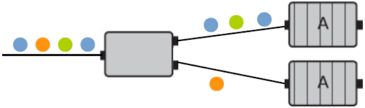


Parallelisation (2/3)

- Shuffle grouping: randomly partitions the tuples.

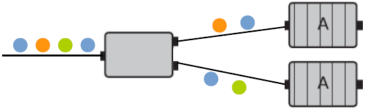


- Field grouping: hashes on a subset of the tuple attributes.

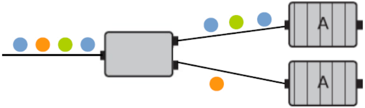


Parallelisation (2/3)

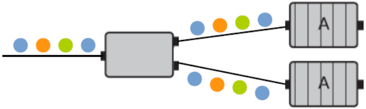
- Shuffle grouping: randomly partitions the tuples.



- Field grouping: hashes on a subset of the tuple attributes.



- All grouping: replicates the entire stream to all the consumer tasks.



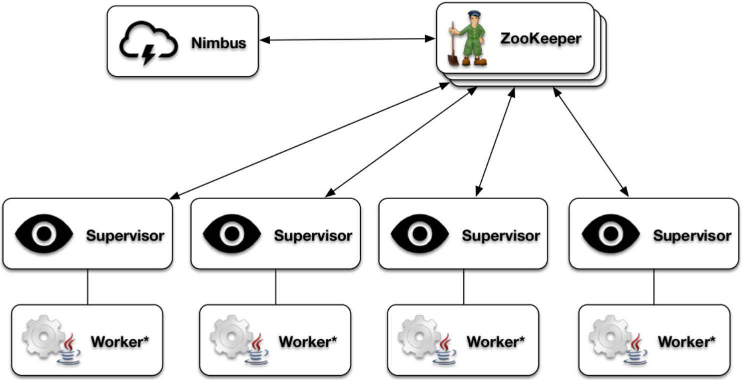
Parallelisation (3/3)

- Global grouping: sends the entire stream to a single bolt.
- Local grouping: sends tuples to the consumer bolts in the same executor.

Word Count in Storm

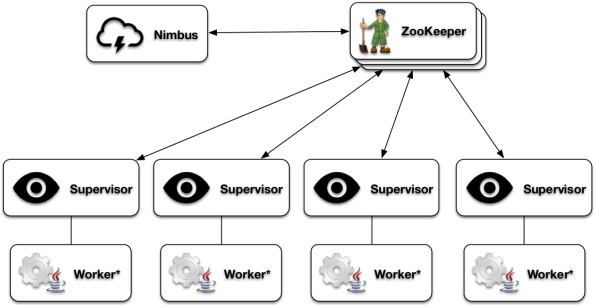
```java
public class WordCountTopology {
  public static class SplitSentence implements IRichBolt { }
  public static class WordCount extends BaseBasicBolt { }

  public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("spout", new RandomSentenceSpout(), 5);

    builder.setBolt("split", new SplitSentence(), 8)
      .shuffleGrouping("spout");
    builder.setBolt("count", new WordCount(), 12)
      .fieldsGrouping("split", new Fields("word"));

    Config conf = new Config();
    conf.setMaxTaskParallelism(3);
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("word-count", conf, builder.createTopology());

    cluster.shutdown();
  }
}
```

Storm Architecture



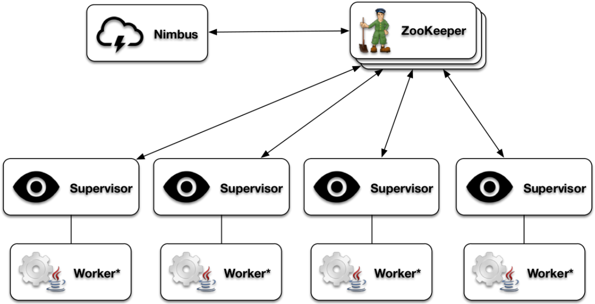
Storm Components (1/4)

- **Nimbus**
  - The master node.
  - Clients submit topologies to it.
  - Responsible for distributing and coordinating the execution of the topology.



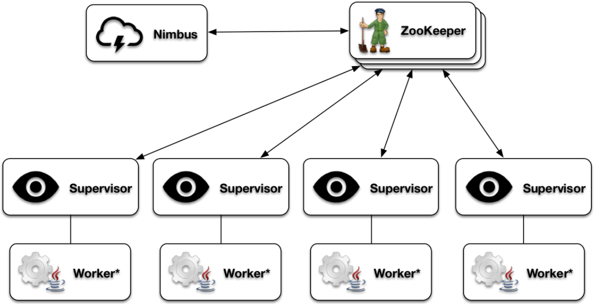
Storm Components (2/4)

- Zookeeper
  - Nimbus uses a combination of the local disk(s) and Zookeeper to store state about the topology.



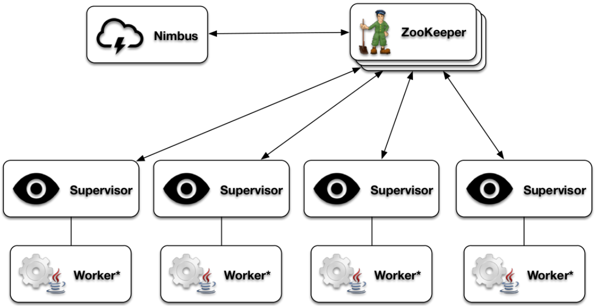
Storm Components (3/4)

- Worker nodes
  - Each worker node runs one or more worker processes.
  - Each worker process runs a JVM, in which it runs one or more executors.
  - Executors are made of one or more tasks, where the actual work for a bolt or a spout is done in the task.



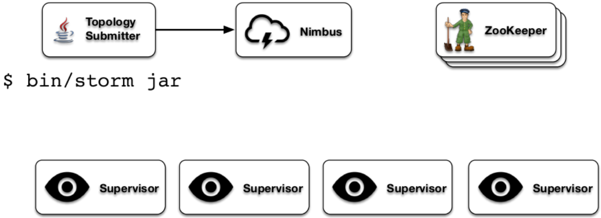
Storm Components (4/4)

- Supervisor
  - Each worker node runs a supervisor.
  - It receives assignments from Nimbus and spawns workers based on the assignment.
  - Contact Nimbus with a periodic heartbeat protocol, advertising the topologies that they are currently running, and any vacancies that are available to run more topologies.



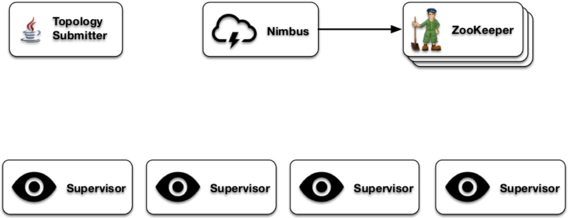
Storm Deployment (1/5)

- Topology submitter uploads topology to Nimbus.



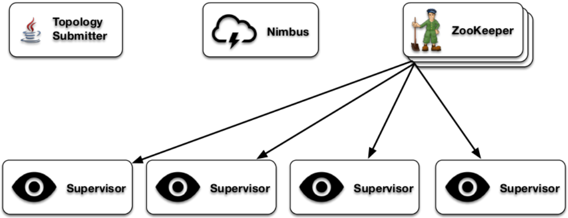
Storm Deployment (2/5)

- Nimbus calculates assignments and sends to Zookeeper.



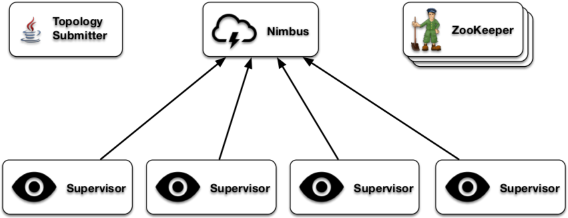
Storm Deployment (3/5)

- Supervisor nodes receive assignment information via Zookeeper watches.



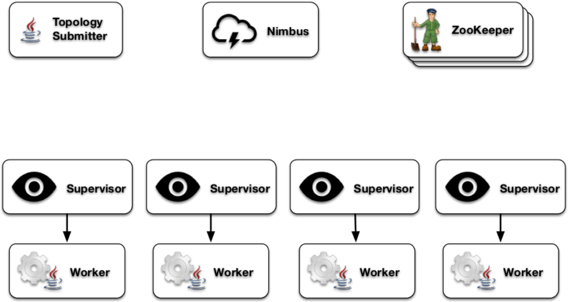
Storm Deployment (4/5)

- Supervisor nodes download topology from Nimbus.



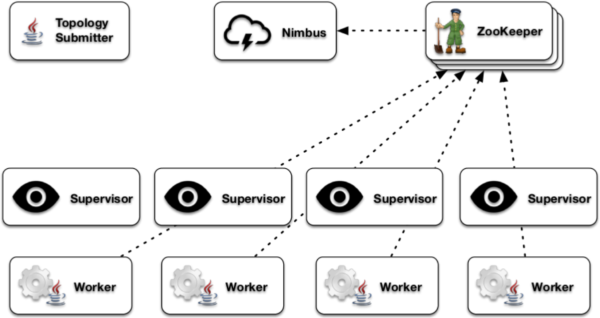
Storm Deployment (5/5)

- Supervisors spawn workers (JVM processes) to start the topology.



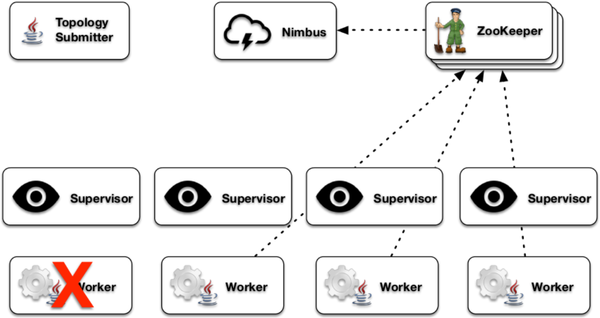
Fault Tolerance (1/4)

- Workers heartbeat back to Supervisors and Nimbus via ZooKeeper, as well as locally.



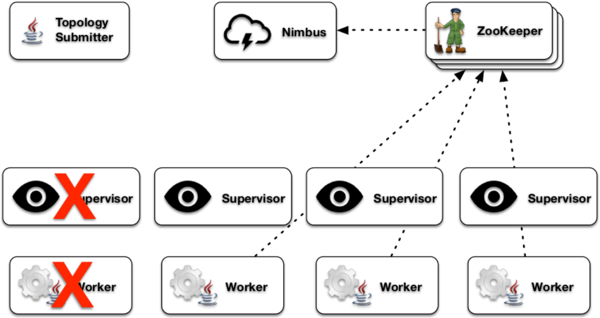
Fault Tolerance (2/4)

- If a worker dies (fails to heartbeat), the Supervisor will restart it.
- If a worker dies repeatedly, Nimbus will reassign the work to other nodes in the cluster.



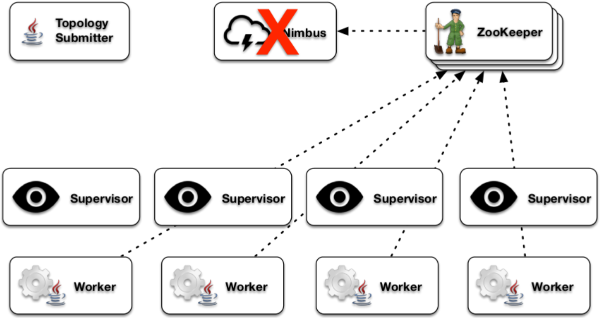
Fault Tolerance (3/4)

- If a supervisor node dies, Nimbus will reassign the work to other nodes.



Fault Tolerance (4/4)

- If Nimbus dies, topologies will continue to function normally, but won't be able to perform reassignments.



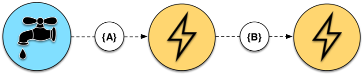
Reliable Processing (1/6)

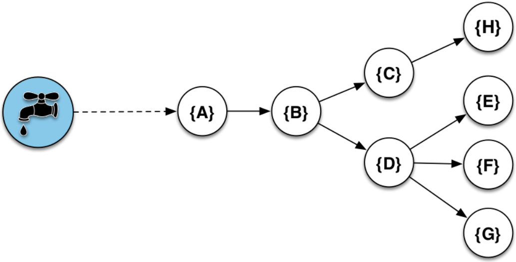
- Storm provides two types of semantic guarantees:
  - At most once: each tuple is either processed once, or dropped in the case of a failure.
  - At least once (reliable processing): it guarantees that each tuple that is input to the topology will be processed at least once.

Reliable Processing (2/6)

- Bolts may emit tuples anchored to the ones they received.
  - Tuple B is a descendant of Tuple A.

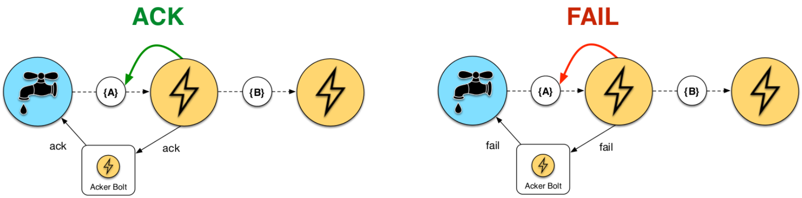
- Multiple anchorings form a Tuple tree.





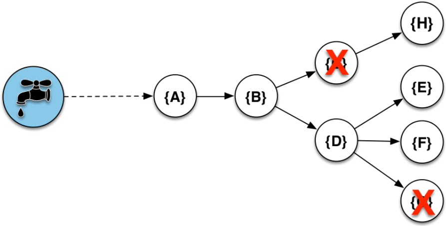
Reliable Processing (3/6)

- Bolts can acknowledge that a tuple has been processed successfully.
- Acks are delivered via a system-level bolt.
- Bolts can also fail a tuple to trigger a spout to replay the original.



Reliable Processing (4/6)

- Any failure in the tuple tree will trigger a replay of the original tuple.
- How to track a large-scale tuple tree efficiently?



Reliable Processing (5/6)

- Tuples are assigned a 64-bit message id at spout.
- Emitted tuples are assigned new message ids.
- These message ids are XORed and sent to the acker bolt along with the original tuple message id.
- When the XOR checksum goes to zero, the acker bolt sends the final ack to the spout that admitted the tuple, and the spout knows that this tuple has been fully processed.
  - $a \oplus (a \oplus b) \oplus c \oplus (b \oplus c) == 0$

Reliable Processing (6/6)

- It is possible that due to failure, some of the XOR checksum will never go to zero.
- The spout initially assigns a timeout parameter to each tuple.
- The acker bolt keeps track of this timeout parameter, and if the XOR checksum does not become zero before the timeout, the tuple is considered to have failed.
  - The data source will replay it back in the subsequent iteration.

SEEP

Contribution

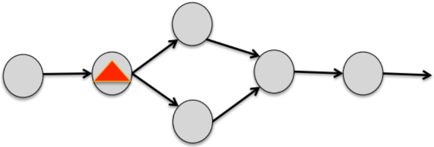
- Build a stream processing system that scale out while remaining fault tolerant when queries contain stateful operators.

Challenges

- Stateful operators
  - E.g., join or aggregation
  - Finite window of tuples: small amount of states
- Intra-query parallelism
  - Static vs. dynamic
- Fault tolerance

Core Idea

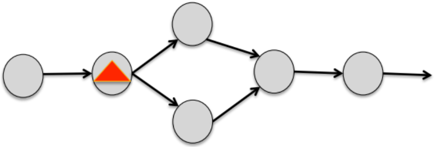
- Make operator state an external entity that can be managed by the stream processing system.



Core Idea

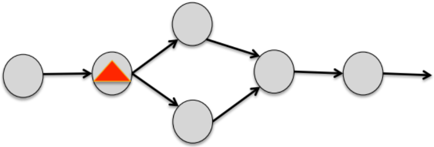
- Make operator state an external entity that can be managed by the stream processing system.

- Operators have direct access to states.

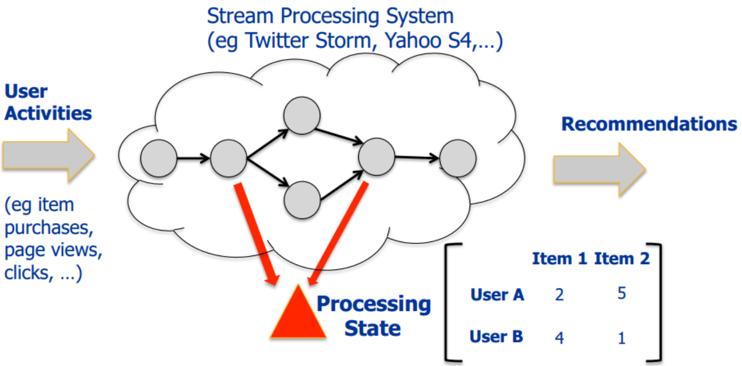


Core Idea

- Make operator state an external entity that can be managed by the stream processing system.
- Operators have direct access to states.
- The system manages states.

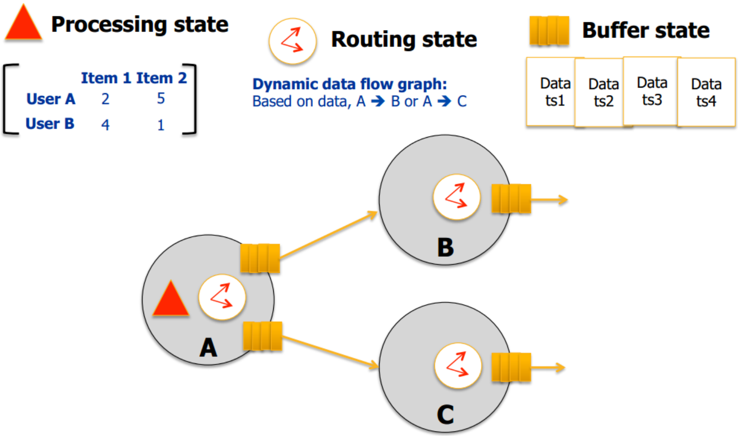


States (1/2)



Stream Processing System
(eg Twitter Storm, Yahoo S4,...)
User
Activities
(eg item
purchases,
page views,
clicks, ...)
Recommendations
Processing
State
Item 1 Item 2
User A 2 5
User B 4 1

States (2/2)



Processing state
Item 1 Item 2
User A 2 5
User B 4 1
Routing state
Dynamic data flow graph:
Based on data, A → B or A → C
Buffer state
Data ts1
Data ts2
Data ts3
Data ts4
A
B
C

Operator State Management

- On scale out: partition operator state correctly, maintaining consistency

Operator State Management

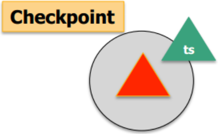
- On scale out: partition operator state correctly, maintaining consistency
- On failure recovery: restore state of failed operator

Operator State Management

- On scale out: partition operator state correctly, maintaining consistency
- On failure recovery: restore state of failed operator
- Define primitives for state management and build other mechanisms on top of them.

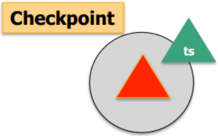
State Management Primitives

- Checkpoint
  - Makes state available to system.
  - Attaches last processed tuple timestamp.

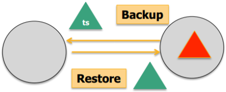


State Management Primitives

- **Checkpoint**
  - Makes state available to system.
  - Attaches last processed tuple timestamp.

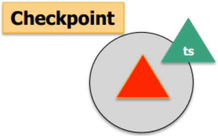


- **Backup/Restore**
  - Moves copy of state from one operator to another.

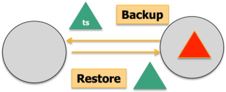


State Management Primitives

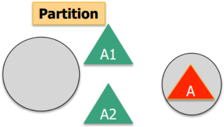
- Checkpoint
  - Makes state available to system.
  - Attaches last processed tuple timestamp.



- Backup/Restore
  - Moves copy of state from one operator to another.



- Partition
  - Splits state to scale out an operator.



State Primitives: Checkpoint

- Checkpoint state = the processing state + the buffer state

State Primitives: Checkpoint

- Checkpoint state = the processing state + the buffer state

- That routing state is not included in the state checkpoint.
  - It only changes in case of scale out or recovery.

State Primitives: Checkpoint

- Checkpoint state = the processing state + the buffer state

- That routing state is not included in the state checkpoint.
  - It only changes in case of scale out or recovery.

- The system executes checkpoint asynchronously and periodically.

State Primitives: Backup and Restore (1/2)

- The operator state (i.e., the checkpoint output) is backed up to an upstream operator.

State Primitives: Backup and Restore (1/2)

- The operator state (i.e., the checkpoint output) is backed up to an upstream operator.
- After the operator state was backed up, already processed tuples from output buffers in upstream operators can be discarded.
    - They are no longer required for failure recovery.

State Primitives: Backup and Restore (2/2)

- Backed up operator state is restored to another operator to recover a failed operator or to redistribute state across partitioned operators.

State Primitives: Backup and Restore (2/2)

- Backed up operator state is restored to another operator to recover a failed operator or to redistribute state across partitioned operators.

- After restoring the state, the system replays unprocessed tuples in the output buffer from an upstream operator to bring the operator's processing state up-to-date.

State Primitives: Partition

- Split the state of a stateful operator across the new partitioned operators when it scales out.

State Primitives: Partition

- Split the state of a stateful operator across the new partitioned operators when it scales out.
- Partitioning the key space of the tuples processed by the operator.

State Primitives: Partition

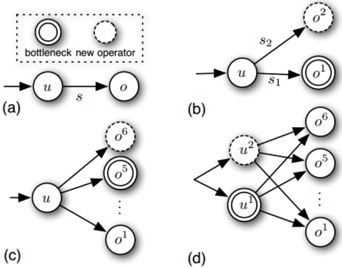
- Split the state of a stateful operator across the new partitioned operators when it scales out.
- Partitioning the key space of the tuples processed by the operator.
- The routing state of its upstream operators must also be updated to account for the new partitioned operators.

State Primitives: Partition

- Split the state of a stateful operator across the new partitioned operators when it scales out.
- Partitioning the key space of the tuples processed by the operator.
- The routing state of its upstream operators must also be updated to account for the new partitioned operators.
- The buffer state of the upstream operators is partitioned to ensure that unprocessed tuples are dispatched to the correct partition.

Scale Out

- To scale out queries at runtime, the system partitions operators on-demand in response to bottleneck operators.
- The load of the bottlenecked operator is shared among a set of new partitioned operators.



Fault-Tolerance

- Overload and failure are handled in the same fashion.
- Operator recovery becomes a special case of scale out, in which a failed operator is scaled out.

Fault-Tolerant Scale Out Algorithm

- Two versions of operator's state that can be partitioned for scale out:
  - The current state
  - The recent state checkpoint

Fault-Tolerant Scale Out Algorithm

- Two versions of operator's state that can be partitioned for scale out:
  - The current state
  - The recent state checkpoint

- In SEEP, the system partitions the most recent state checkpoint.

Fault-Tolerant Scale Out Algorithm

- Two versions of operator's state that can be partitioned for scale out:
  - The current state
  - The recent state checkpoint

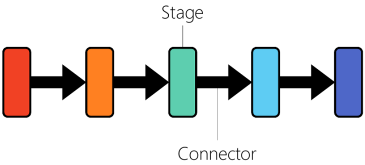
- In SEEP, the system partitions the most recent state checkpoint.

- Its benefits:
  - Avoids adding further load to the operator, which is already overloaded, by requesting it to checkpoint or partition its own state.
  - Makes the scale out process itself fault-tolerant.

Naiad

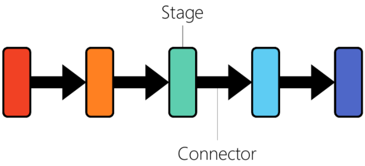
Motivation (1/2)

- Dataflow

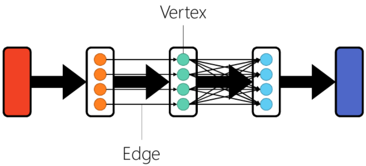


Motivation (1/2)

- Dataflow

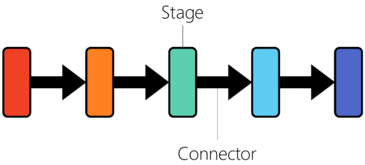


- Dataflow (parallelization)

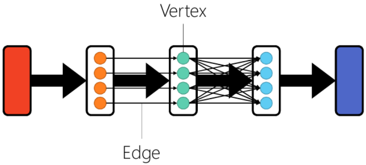


Motivation (1/2)

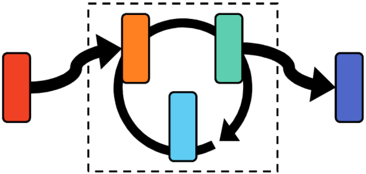
- Dataflow



- Dataflow (parallelization)

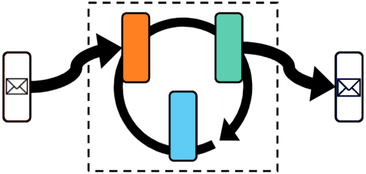


- Dataflow (iteration)



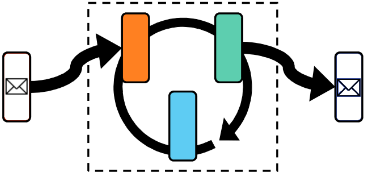
Motivation (2/2)

- Batch iteration

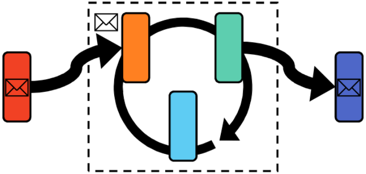


Motivation (2/2)

- Batch iteration



- Streaming iteration

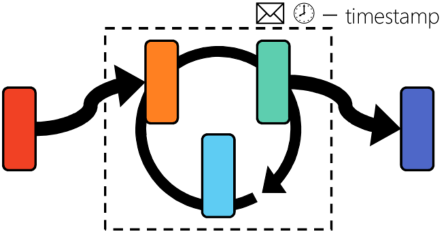


Naiad (1/2)

- Naiad is a distributed system for executing data parallel, and cyclic dataflow programs.
- It satisfies:
  - Stream processing that produces low-latency results for non-iterative algorithms,
  - Batch processing that iterates synchronously at the expense of latency.

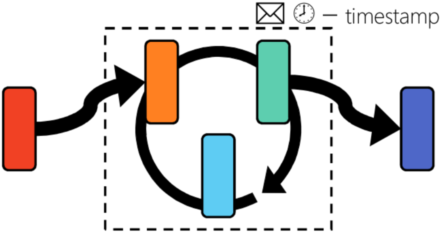
Naiad (2/2)

- Asynchronous execution: low latency of stream processors
- Fine-grained synchronous execution: high throughput of batch processors
- Support for iterative and incremental computations



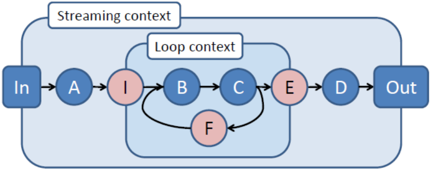
Naiad (2/2)

- Asynchronous execution: low latency of stream processors
- Fine-grained synchronous execution: high throughput of batch processors
- Support for iterative and incremental computations
- Timely dataflow: a new computation model



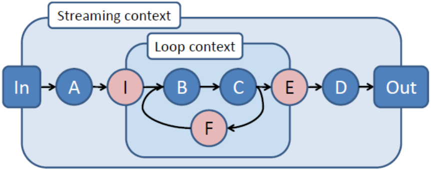
Timely Dataflow and Timestamp (1/3)

- Directed graph that may have cycles (possibly nested)
- Stateful vertices that consume and produce messages asynchronously.



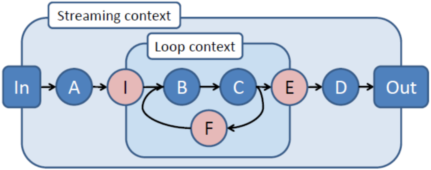
Timely Dataflow and Timestamp (1/3)

- Directed graph that may have cycles (possibly nested)
- Stateful vertices that consume and produce messages asynchronously.
- Structured loops allow feedback in the dataflow to implement iteration.



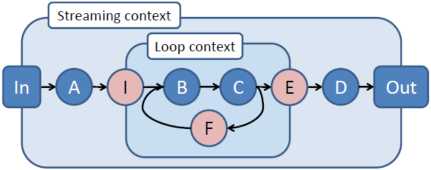
Timely Dataflow and Timestamp (1/3)

- Directed graph that may have cycles (possibly nested)
- Stateful vertices that consume and produce messages asynchronously.
- Structured loops allow feedback in the dataflow to implement iteration.
- Explicit notifications for synchronous processing to indicate all records for a given round of input or loop iteration have been received.



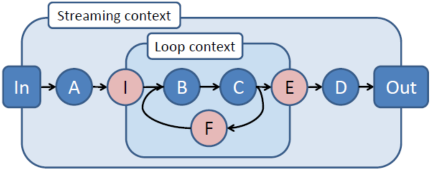
Timely Dataflow and Timestamp (2/3)

- Specified input and output vertices
- Timestamped messages passed along edges.
  - Timestamp : $(\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$



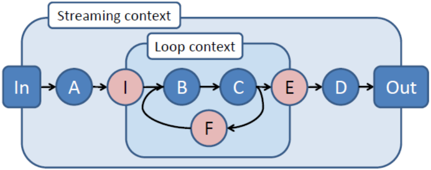
Timely Dataflow and Timestamp (2/3)

- Specified input and output vertices
- Timestamped messages passed along edges.
  - Timestamp : $(\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$
- Epoch: each record at input is labeled with epoch number to distinguish between different batches of data.



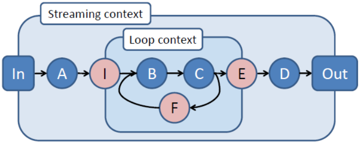
Timely Dataflow and Timestamp (2/3)

- Specified input and output vertices
- Timestamped messages passed along edges.
  - Timestamp : $(\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$
- Epoch: each record at input is labeled with epoch number to distinguish between different batches of data.
- Loop counters: a timestamp has $k \geq 0$ loop counters, where k is the depth of nesting.



Timely Dataflow and Timestamp (3/3)

- Timestamp : $(\underbrace{e \in \mathbb{N}}_{\text{epoch}}, \underbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}_{\text{loop counters}})$
- Passing ingress (I) vertex: $(e, \langle c_1, \ldots, c_k \rangle) \rightarrow (e, \langle c_1, \ldots, c_k, 0 \rangle)$
- Passing egress (E) vertex: $(e, \langle c_1, \ldots, c_k \rangle) \rightarrow (e, \langle c_1, \ldots, c_{k-1} \rangle)$
- Passing feedback (F) vertex: $(e, \langle c_1, \ldots, c_k \rangle) \rightarrow (e, \langle c_1, \ldots, c_k + 1 \rangle)$
- Timestamp ordering: $t_1 = (e_1, \vec{x_1})$ and $t_2 = (e_2, \vec{x_2})$, $t_1 \leq t_2 \iff e_1 \leq e_2 \wedge \vec{x_1} \leq \vec{x_2}$

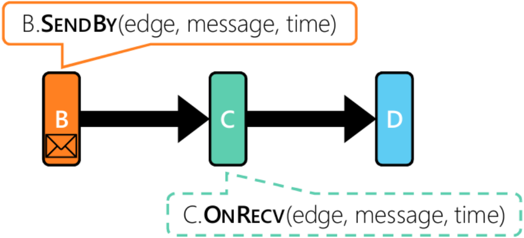


Vertex Computation (1/3)

- Timely dataflow vertex: a possibly stateful object that sends and receives messages and requests and receives notifications.

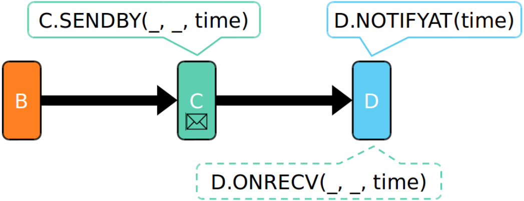
Vertex Computation (2/3)

- Message exchange is completely asynchronous.
  - `u.SendBy(e: Edge, m: Message, t: Timestamp)`
    Sending a message by `u`.
  - `v.OnRecv(e: Edge, m: Message, t: Timestamp)`
    Message is delivered to `v`.



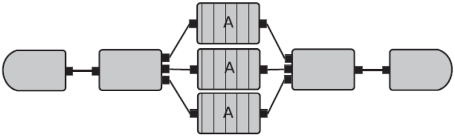
Vertex Computation (3/3)

- Notification delivery is synchronous.
  - `v.NotifyAt(t: Timestamp)`
    Requesting a notification by `v`.
  - `v.OnNotify(t: Timestamp)`
    A notification is delivered to `v`, after all messages with timestamp `t'` $\leq$ `t` have been delivereded.



Parallelisation

- Data parallelism



Word Count in Naiad (1/2)

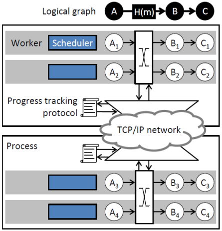
```
public static class ExtensionMethods {
  public static Stream<Pair<TRecord, Int64>, Epoch> StrCount<TRecord>(Stream<TRecord, Epoch> stream) {
    return stream.NewUnaryStage((i, s) => new CountVertex<TRecord>(i, s), ...);
  }

  internal class CountVertex<TRecord> : UnaryVertex<TRecord, Pair<TRecord, Int64>, Epoch> {
    private Dictionary<TRecord, Int64> Counts = new Dictionary<TRecord, long>();
    private HashSet<TRecord> Changed = new HashSet<TRecord>();

    public override void OnReceive(Message<TRecord, Epoch> message) {
      this.NotifyAt(message.time);

      for (int i = 0; i < message.length; i++) {
        var data = message.payload[i];
        if (!this.Counts.ContainsKey(data))
          this.Counts[data] = 0;

        this.Counts[data] += 1;
        this.Changed.Add(data);
     }
    }

    // once all records of an epoch are received, we should send the counts along.
    public override void OnNotify(Epoch time) {
      var output = this.Output.GetBufferForTime(time);
      foreach (var record in this.Changed)
        output.Send(new Pair<TRecord, Int64>(record, this.Counts[record]));

      // reset observed records
      this.Changed.Clear();
    }
```

Word Count in Naiad (2/2)

```
public class WordCount {
  public void Execute(string[] args) {
    // the first thing to do is to allocate a computation from args.
    using (var computation = NewComputation.FromArgs(ref args)) {
      // 1. Make a new data source, to which we will supply strings.
      var source = new BatchedDataSource<string>();

      // 2. Attach source, and apply count extension method.
      var counts = computation.NewInput(source).StrCount();

      // 3. Subscribe to the resulting stream with a callback to print the outputs.
      counts.Subscribe(list => { foreach (var element in list) Console.WriteLine(element); });

      // activate the execution of this graph (no new stages allowed).
      computation.Activate();

      if (computation.Configuration.ProcessID == 0) {
        // read lines of input and hand them to the input, until an empty line appears.
        for (var line = Console.ReadLine(); line.Length > 0; line = Console.ReadLine())
          source.OnNext(line.Split());
      }

      source.OnCompleted();
      computation.Join();
    }
  }
}
```

Naiad Architecture

- Workers: the smallest unit of computation (a single thread).
- Processes: a larger unit of computation (a single OS process).
    - It can contain one or more workers.
    - A machine may host one or more processes.
- Lock-free queue for data exchange between workers in the same process, and TCP connection between two different processes.



Fault Tolerance (1/2)

- Each stateful vertex implements a CHECKPOINT and RESTORE interface.

- Each vertex may either:
  - Log data as computation proceeds,
  - or write a full checkpoint when requested (potentially more compact).

Fault Tolerance (2/2)

- In periodic checkpoints:
  - All processes first pause worker and message delivery threads
  - Flush message queues by delivering outstanding `OnRecv` events
  - Invoke CHECKPOINT on each stateful vertex.
- The system then resumes worker and message delivery threads and flushes buffered messages.
- To recover from a failed process, all live processes revert to the last durable checkpoint, and the vertices from the failed process are reassigned to the remaining processes.

Summary

Summary

- Storm
  - Tuple and stream
  - Spout, bolt, and topology
  - Nimbus, worker, supervisor, and zookeeper
  - Reliable processing: xored ids

Summary

- SEEP
  - Make operator state an external entity
  - Primitives for state management: checkpoint, backup/restore, partition

Summary

- Naiad
  - Timely dataflow
  - Asynchronous and fine-grained synchronous
  - Timestamp messages, epoch, and loop counters
  - Streaming context and loop context
  - Workers and processes

Questions?

Acknowledgements

Some slides and pictures were derived from Peter Pietzuch (Imperial College) and Derek G. Murray (Google) slides.