Processing Fast Data

UNIVERSITÉ CÔTE D'AZUR

Context and motivation

- So far, we have seen how to store and analyze data
  - Assume all data is available
  - Process whole dataset or restrict it manually (i.e. in algorithm)
- Fine if
  - It makes sense
  - Latency is not the main factor (e.g. a couple of minutes to get results is fine)
- But what if
  - We want the result fast
  - There is no notion of “whole” dataset, new data is created all the time

Data streams

- A stream is an unbounded sequence of elements
  - No end to a stream
  - New data becomes available over time
  - Some might become unavailable or outdated
- Elements of a stream have timestamps
  - Generation time as measured by the source
  - Arrival in the system as measured by the system
- Elements are produced by external sources
  - No control on the volume and the arrival rate
- Systems specializes in stream-processing are called Data Stream Processing (DSP) systems or Stream Processing Engines (SPE)

The 8 requirements of Real-Time Stream Processing

- *Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 requirements of real-time stream processing. SIGMOD Rec. 34, 4 (December 2005), 42–47.*
- To efficiently process data-streams, a system must have these features

1) Keep the data moving
   - Data should not be stored for processing
2) Support a high-level, stream-oriented language
   - Need for adapted primitives and operators

The 8 requirements of Real-Time Stream Processing

3) Provide resilience against streams imperfections
   - Missing or out-of-order data

4) Be deterministic
   - Outcome should be predictable and repeatable

5) Have an efficient way to store, access or modify state information
   - This information can represent results on past events
   - Allows processing new data with past information
   - Should not add latency (no complex or remote database access)

The 8 requirements of Real-Time Stream Processing

6) Ensure availability and data integrity
   - Use high-availability solutions like master with failover
   - No data should be lost

7) The system should be scalable
   - Distributed with automatic horizontal scaling

8) Process data immediately
   - Low overhead and low latency

General principles

Models

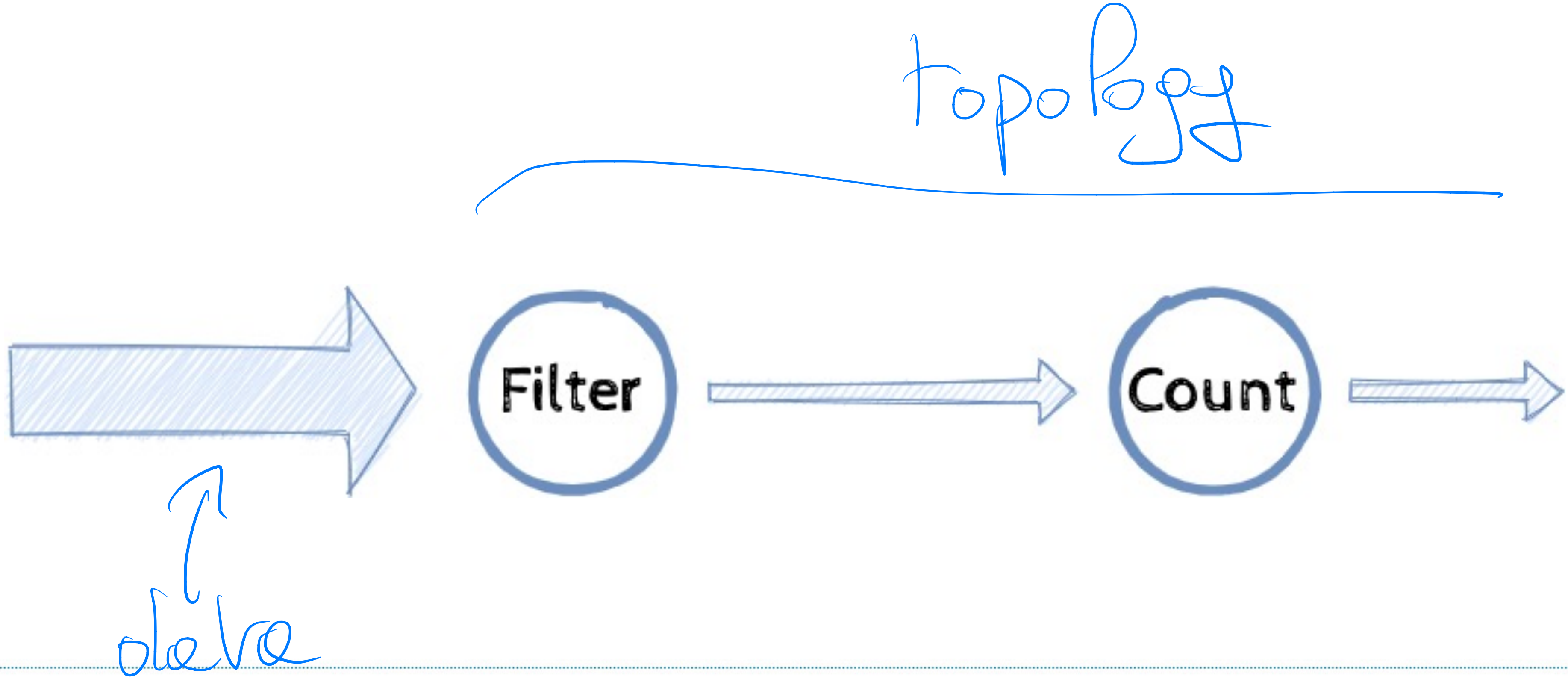
Processing models

- Batch processing
  - Wait for all data before processing
  - High latency but high efficiency
- Event-based processing
  - Process each new data as they arrive
  - Low latency but high overhead
- Micro-batching
  - Process new data in small batches
  - Try to balance latency and overhead

Programming model

- Dataflow (or datastream) programming
  - Program is a Directed (Acyclic) Graph or D(A)G
  - Usually called topology
- Each vertex is an operator
  - Data flow through edges
- Data usually have
  - Timestamp
  - Key-value format
  - Sometimes called tuples

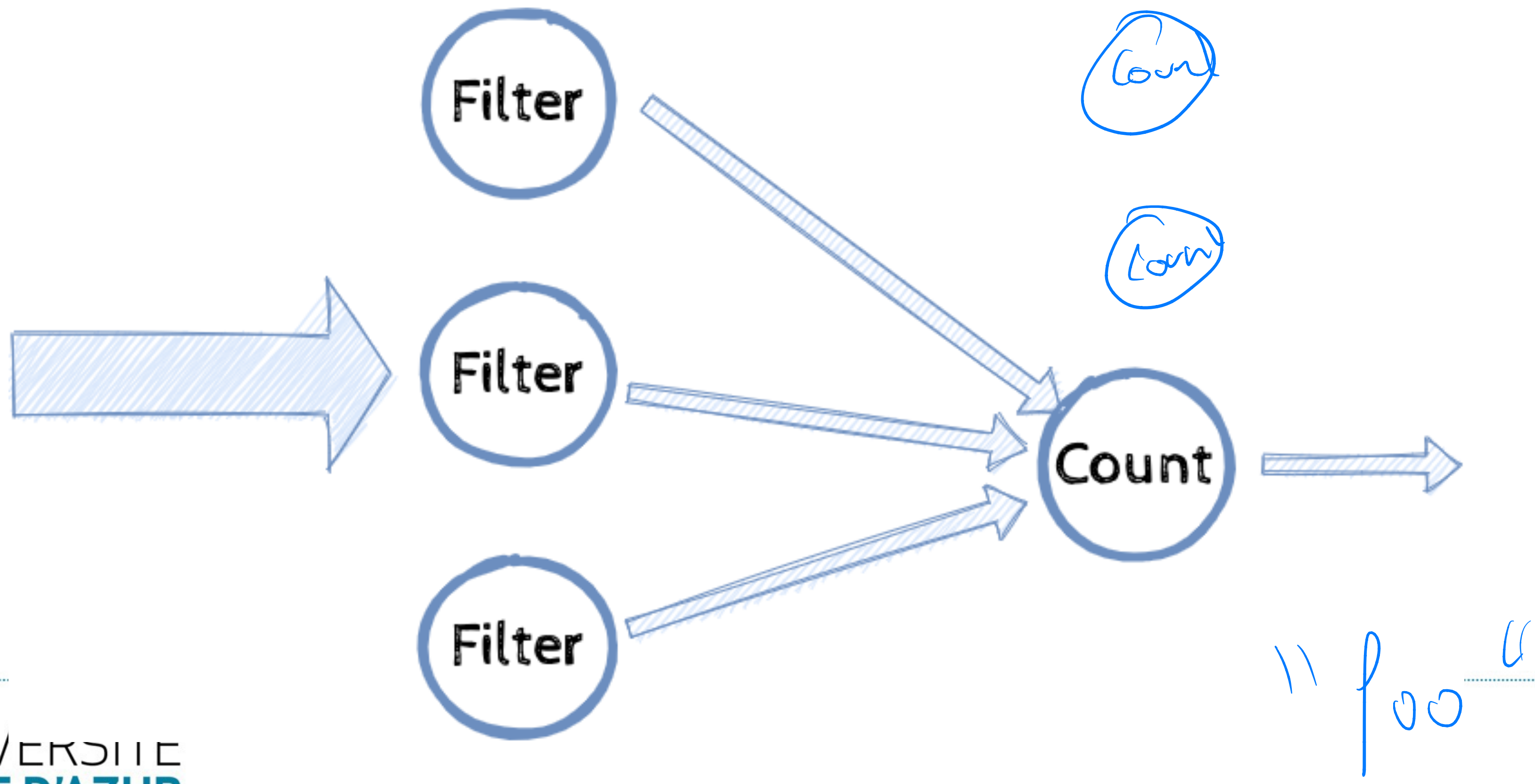
Example : counting specific words



Programming model

- Assume a shared-nothing architecture
  - Operators can have internal state
  - Work on separate streams/partitions
  - Avoid global state in the system
- Easy parallelism and distribution
  - Split data among operators
  - Scaling-out operators

Example : counting specific words



UNIVERSITÉ CÔTE D'AZUR

Execution Model

- Exactly ~~one~~ once
  - Each data is processed exactly once
  - Intuitive model, but hard to enforce
- At least once
  - All data will be processed once
  - Some might be processed multiple times

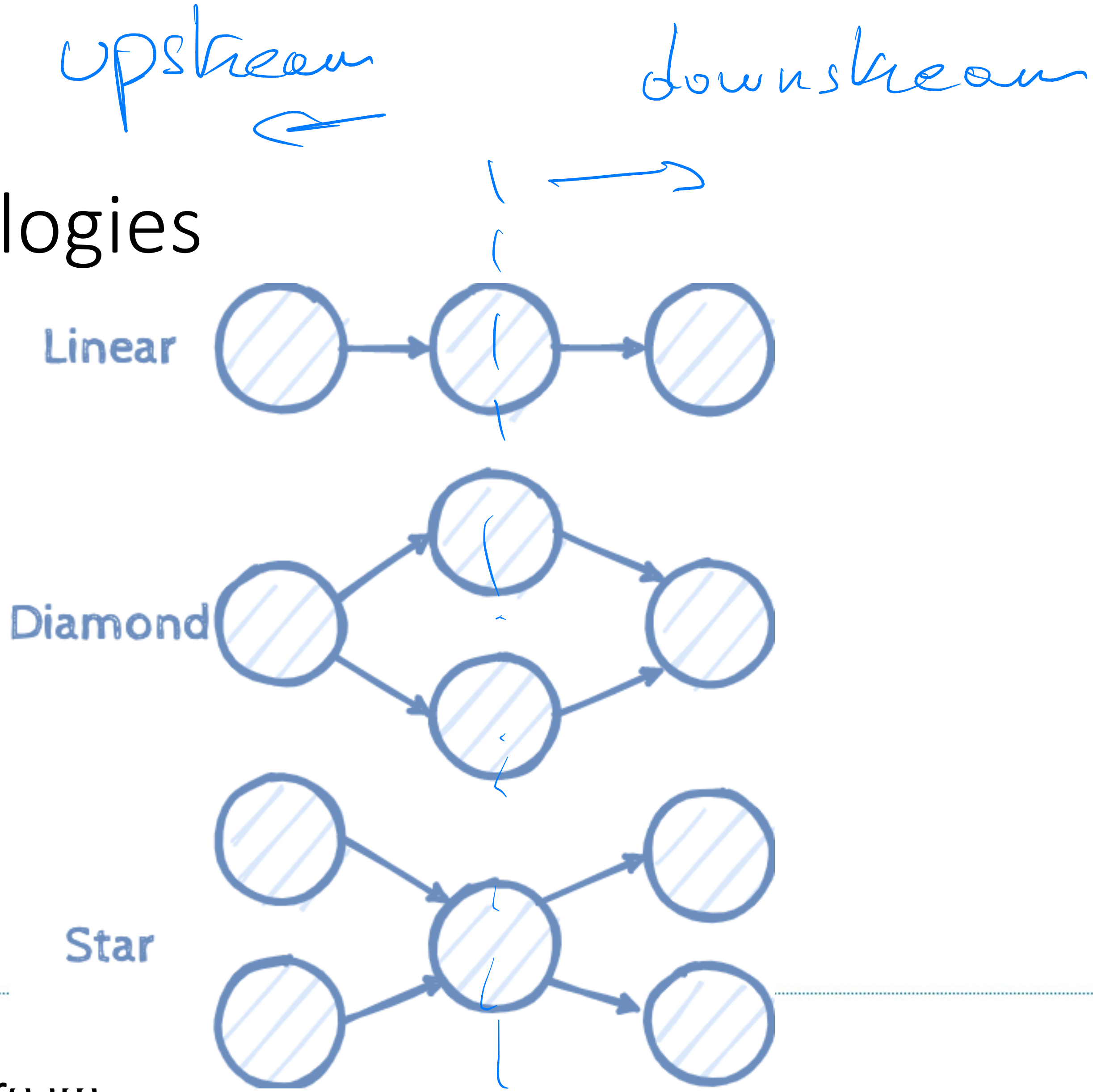
General principles

Operators and topologies

Common operators

- Some operators appear often in apps/papers
- Stateless operators
  - Map, filter,
- Stateful operators
  - Work on multiple data, including previous
  - Sum, count, min, max, average

Classical topologies



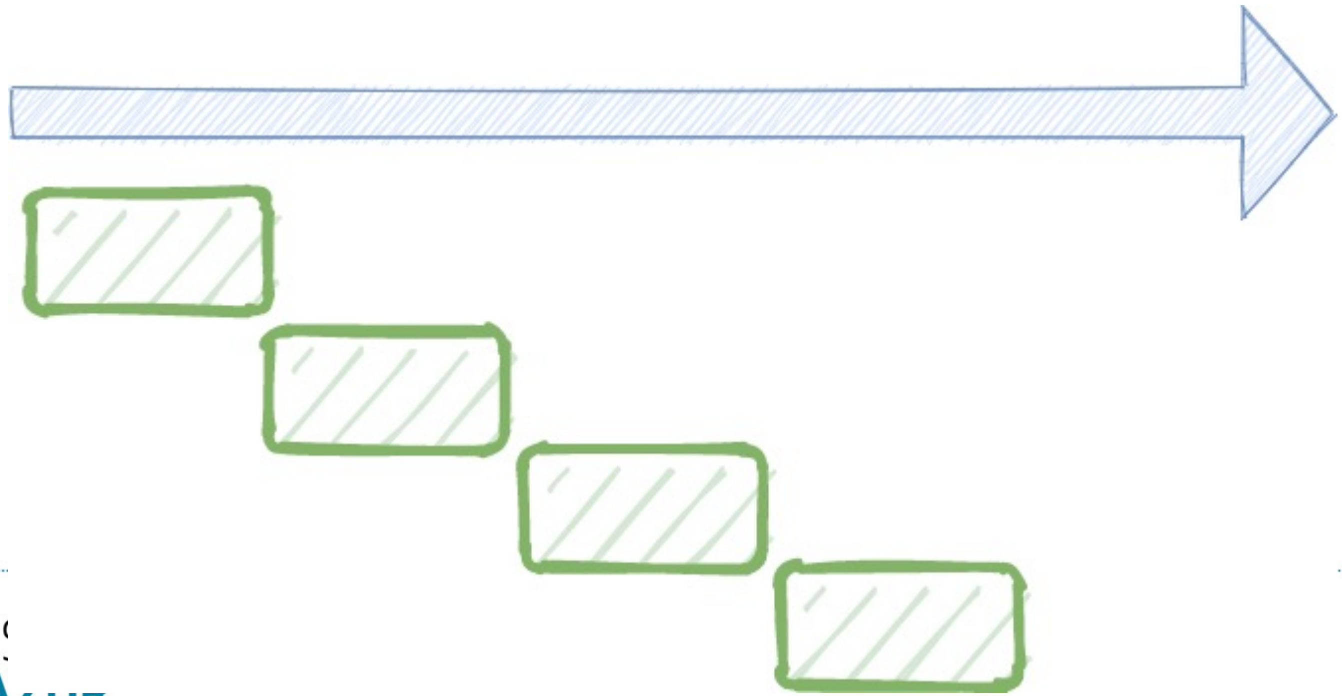
- Upstream vs Downstream

Windowing

- Sometimes need to process elements in group
  - But a datastream is unbounded!
  - Need to artificially create groups
- Window
  - A finite set of elements to be processed together
  - Can cover multiple micro-batches
- Elements can be grouped by
  - Timestamp (emission, arrival, processing…)
  - Count
  - Other criteria

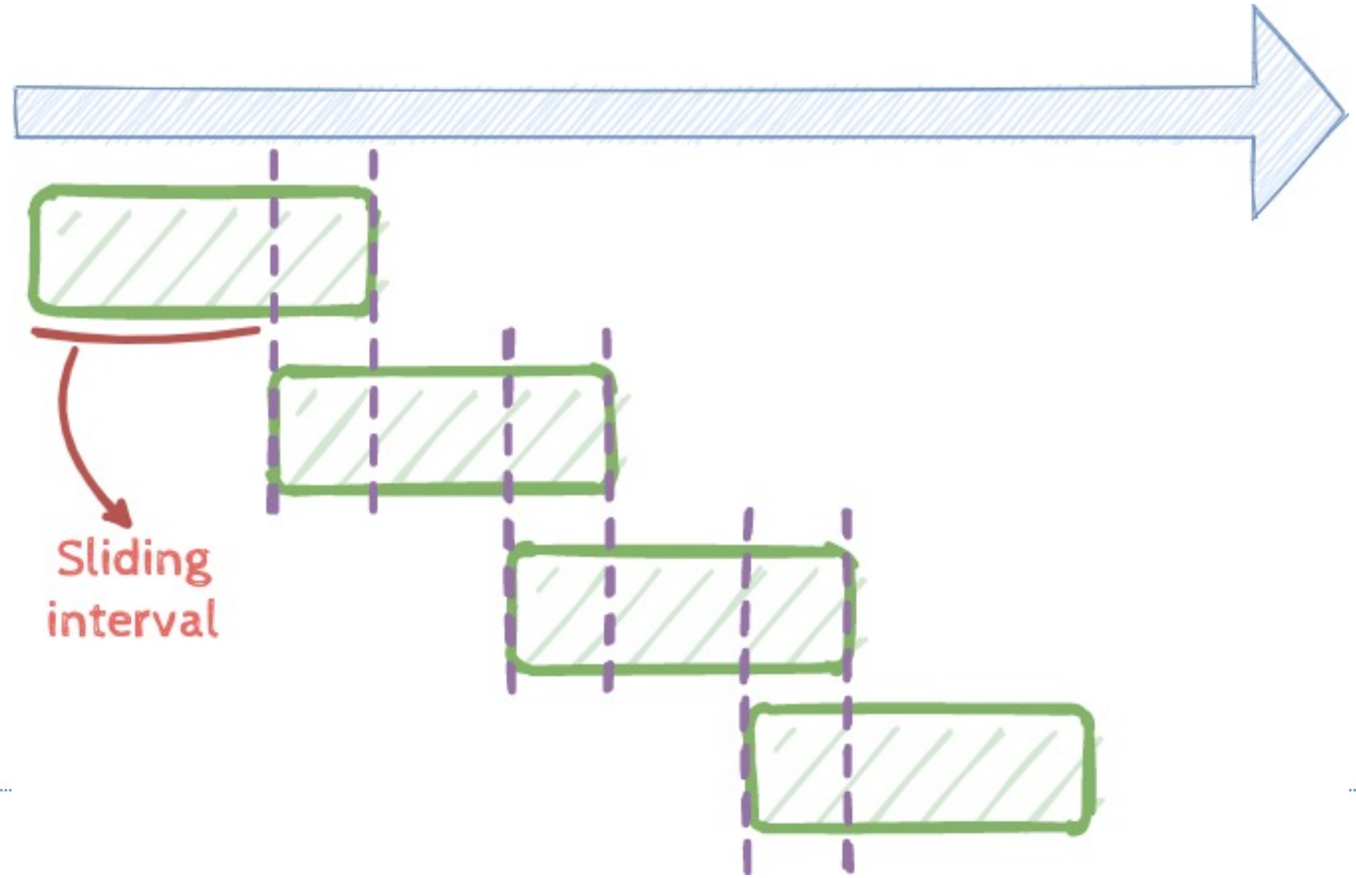
Windowing

- Tumbling
  - No common elements between successive windows



Windowing

- Sliding
  - Windows can overlap

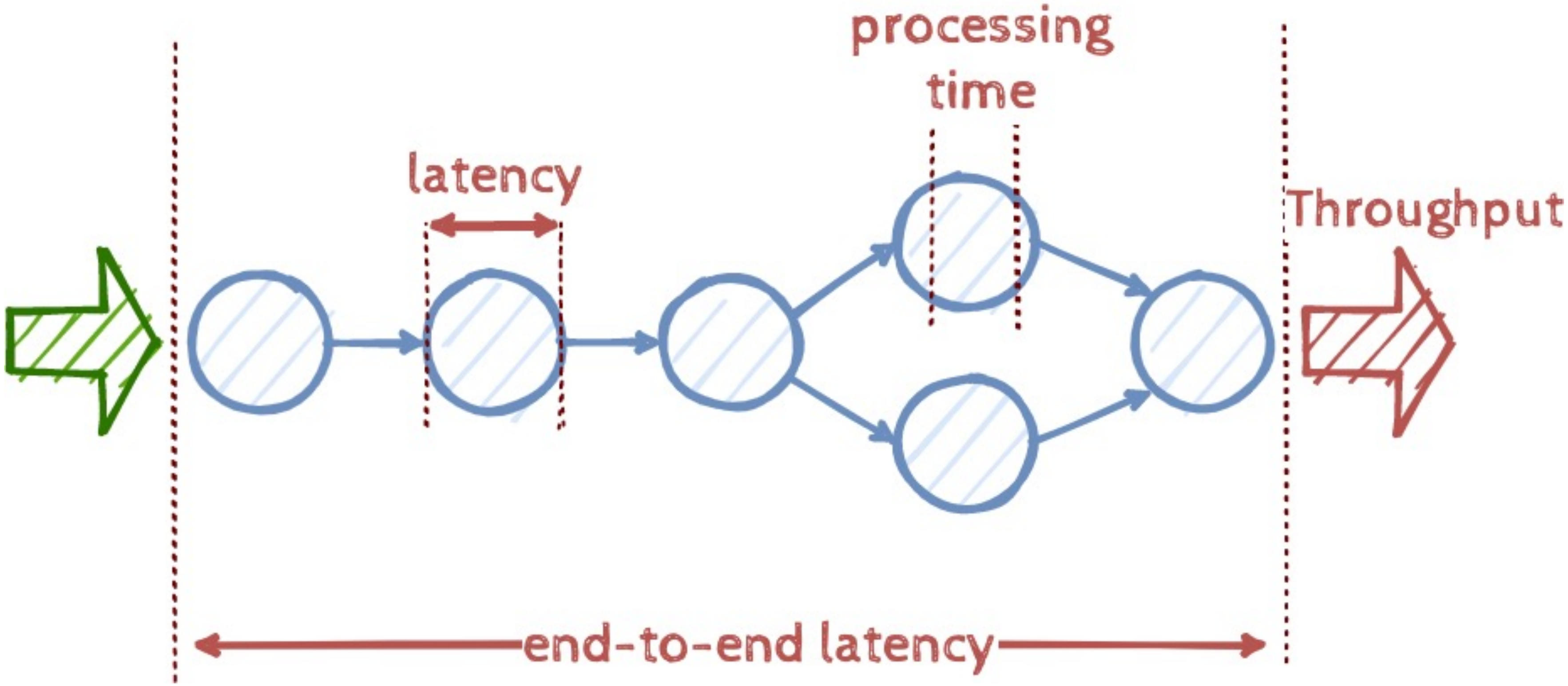


General principles

Performance

Metrics

- Processing time
  - Measured at each operator
- Latency
  - Duration between arrival of data and its processing
  - Can be measured at various points
  - End-to-end latency indicates how reactive the system is
- Throughput
  - Number of tuples processed per time-unit
- Latency and throughput can change independently
  - Example ?


processing
time
latency
Throughput
end-to-end latency

Backpressure

- Not all operator of a topology have the same speed
  - Consider slowest operator in DAG
- How to manage slow operators?
  - Put incoming tuples into a queue for buffering
  - Risk of ever-increasing queue
- Solution
  - Notify upstream to slow down
- Backpressure helps with temporary overload

Case Study : Apache Storm

Introduction and concepts

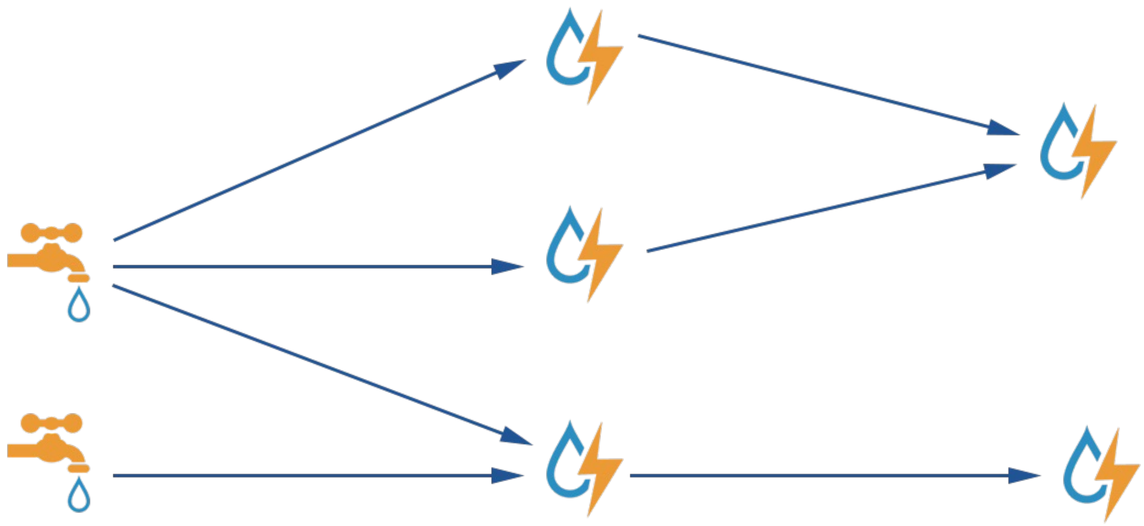
UNIVERSITÉ CÔTE D'AZUR

Introduction

- A distributed Stream Processing Engines
- Written in Clojure
  - Dialect of Lisp on the JVM
- Bought by Twitter, open sourced
- Now an Apache project

Heon

Basic concepts



- 2 basic components
  - Spouts
  - Bolts
- Data are Tuples
  - Travel on Streams
- Interconnected to form a topology

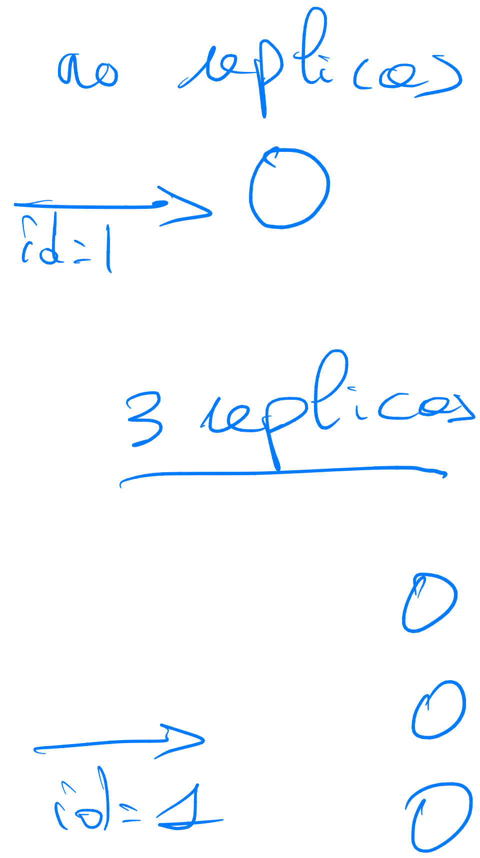
Basic concepts

- Tuples
  - Basic data element
  - Contains named fields
  - Contain integers, longs, shorts, bytes, strings, doubles, floats, booleans, byte arrays or any Serializable type
- Streams
  - Unbounded sequence of tuples
  - Streams have an id
- Spouts
  - Data source
  - Read data from external sources (file, network,...) and emit them on Streams

Basic concepts

- Bolts
  - Data processing unit
  - Receives data from Streams, treat them and send them to possibly other bolts
- Tasks
  - Replicas of Spouts and Bolts
- Topology
  - An interconnection of spouts and bolts
  - Direct Acyclic Graph
- A Storm application
  - Contains a topology
  - Runs indefinitely

Stream grouping

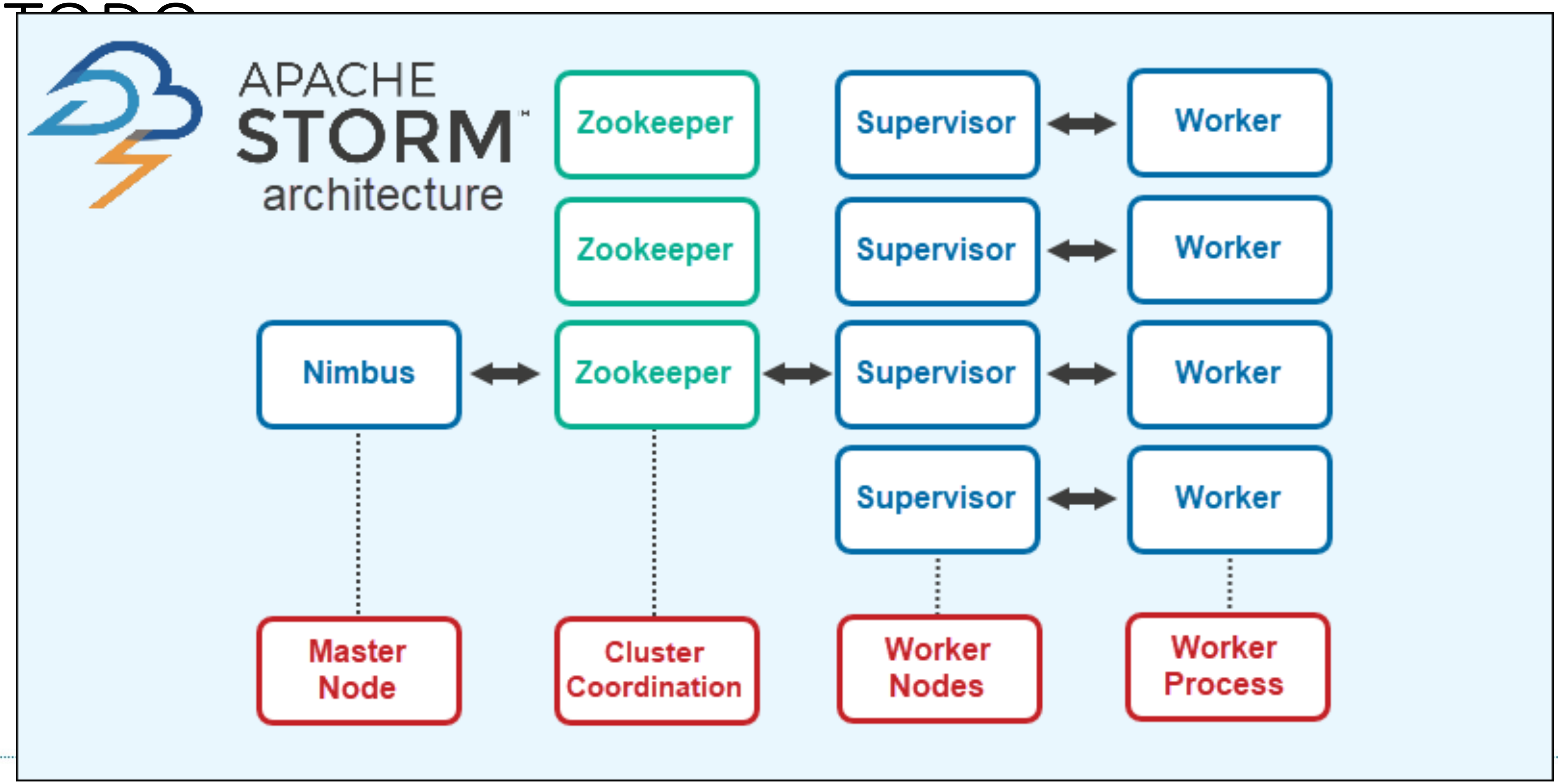


- Each bolt must specify which stream(s) it will receive
  - How is the stream partitioned if multiple replicas ?
  - Need to choose a strategy, called Stream Grouping
  - 8 of them in Storm
- Shuffle Grouping :
  - Default, random distribution, with guaranteed equal load
- Fields Grouping
  - Stream is partitioned by fields from Tuples
  - Tuples with the same value for a field are sent to the same Task
- All Grouping
  - The stream is replicated to all Tasks
- Global Grouping
  - The stream goes to the Task with the lowest id

Architecture

- Distributed architecture
- *Nimbus*
  - Controller Node
  - Manage new topologies, faut-tolerance, scheduling
- *Zookeeper*
  - Manage coordination between all components and cluster resources
  - Not a Storm specific component
- *Supervisor*
  - Provide *worker* (JVM) to *Nimbus* → Server
- *Workers*

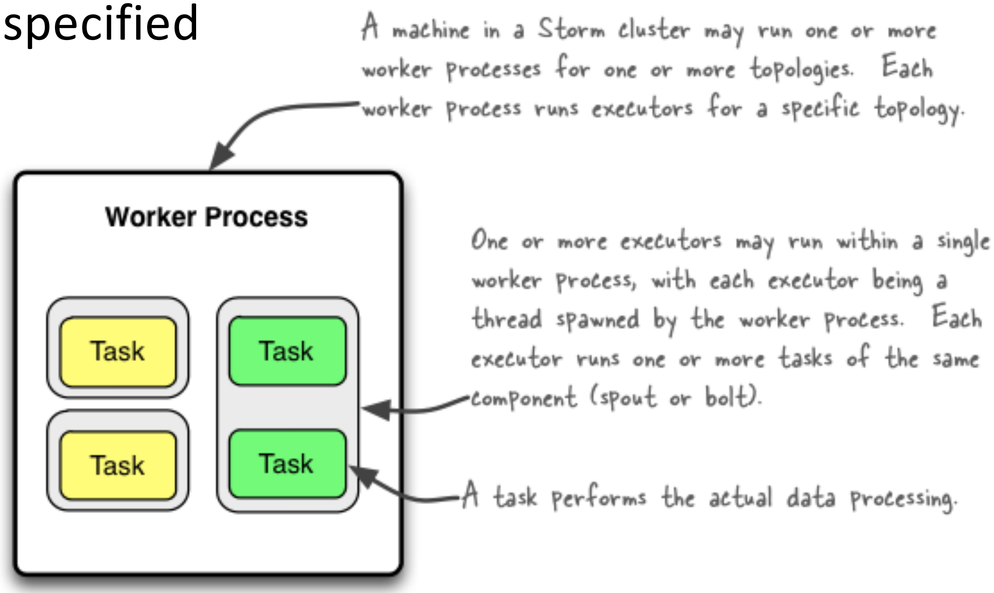
TODO



https://phoenixnap.com/kb/apache-storm

Worker

- *Worker*
  - A JVM containing *Executors* (threads)
  - Each worker is assigned a communication port (TCP), used also as ID
  - Executors process *tasks* (*Spouts and Bolts*)
  - Number of executors can be specified
    - *Parallelism hint*



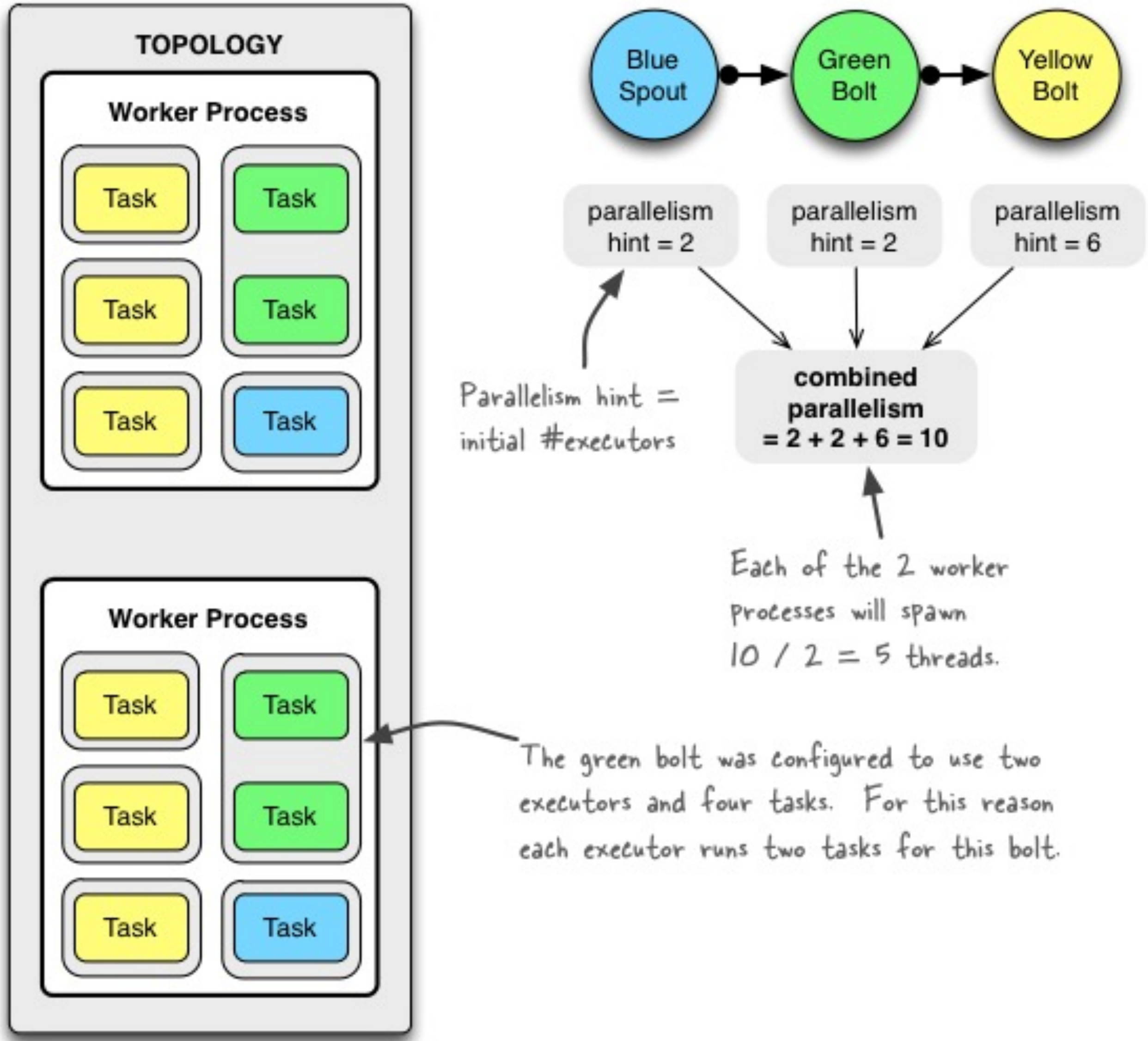
https://storm.apache.org/releases/2.6.0/Understanding-the-parallelism-of-a-Storm-topology.html

Parallelism in Storm

- A topology is ran by 3 entities
  - Worker (processes)
  - Executor (threads)
  - Tasks
- When creating a topology, it's possible to change the number of these entities
- Workers:
  - How many workers processes to create for the topology
  - *org.apache.storm.Config.setNumWorkers(…)*
- Executors
  - How many executors to spawn per Spout/Bolt
  - Parameter *parallelismHint*
- Tasks
  - How many tasks per Spout/Bolt
  - Method *setNumTasks(…)*
  - If not set, Storm will create 1 task per executor

2 executors

- Workers set to 2
- Blue Spout
  - ParallelismHint : 2
  - SetNumTasks : 1
- Green Bolt
  - ParallelismHint : 2
  - Tasks : 4
- Yellow Bolt
  - ParallelismHint : 6
  - SetNumTasks : 1



https://storm.apache.org/releases/2.6.0/Understanding-the-parallelism-of-a-Storm-topology.html

Case Study : Apache Storm

Writing topologies

UNIVERSITÉ
CÔTE D'AZUR

Writing Spouts

- A spout should
  - Extends BaseRichSpout
  - Indicates what tuples it will produce
  - Provides a mechanism to emit new tuples
- There can be many instances of the same Spout
  - Can use private attributes to save state
- Spouts are started by the Storm runtime

Tuples
{ "entier" : value

Writing Spouts

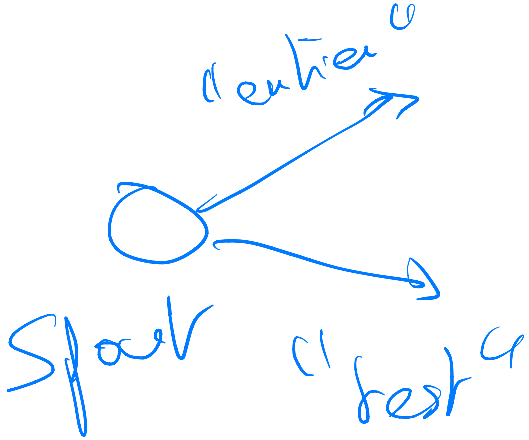
- Declaring output tuples
  - Must override *declareOutputFields*
  - Each field in the tuple has a String id, but no type is specified

```
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields(...fields:"entier"));
}
```

on the default stream

- Possible to declare multiple fields

```
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields(...fields:"word", "count"));
}
```



Writing Spouts

- A spout can declare Streams
  - Useful for partitioning output data
  - By default id is “default”

```java
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declareStream(streamId:"entier", new Fields(...fields:"entier"));
}
```

Writing Spouts

- When a task is created by Storm runtime, the open method will be called
  - Gives a reference to the SpoutOutputCollector to emit
  - Useful for instancing private variables

```
@Override
public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
  _collector = collector;
  _rand = new Random();
}
```

Writing Spouts

- Storm will request the Spout to emit new Tuples with method *nextTuple*
  - Should be non blocking
  - Will be called most of the time without pause

```
@Override
public void nextTuple() {
    Utils.sleep(millis:10);
    String[] sentences = new String[]{ "the cow jumped over the moon", "an apple a
            "four score and seven years ago", "snow white and the seven dwarfs", "i am
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
}
```

UNIVERSITÉ CÔTE D'AZUR

Send on Stream

Writing Bolts

- A Bolt should
  - Extends BaseBasicBolt
  - Indicates what tuples it will produce
  - Provides a mechanism to process incoming tuples
- There can be many instances of the same Bolt
  - Can use private attributes to save state
- Bolts are started by the Storm runtime

Writing Bolts

- Declaring output fields
  - Just like in Spouts, it can declare Fields and Streams

```
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields(...fields:"word", "count"));
}
```

Writing Bolts

- When a task is created by Storm runtime, the *prepare* method will be called
  - Does NOT give a reference to collector
  - Useful for instancing private variables

```
@Override
public void prepare(Map stormConf, TopologyContext context) {

}
```

Writing Bolts

- When a tuple is available, the Storm runtime calls *execute*
  - Pass the Tuple as argument
  - Pass the collector also
- Fields in Tuple can be accessed by String id
  - Need to know the type to call the corresponding getXXXByField

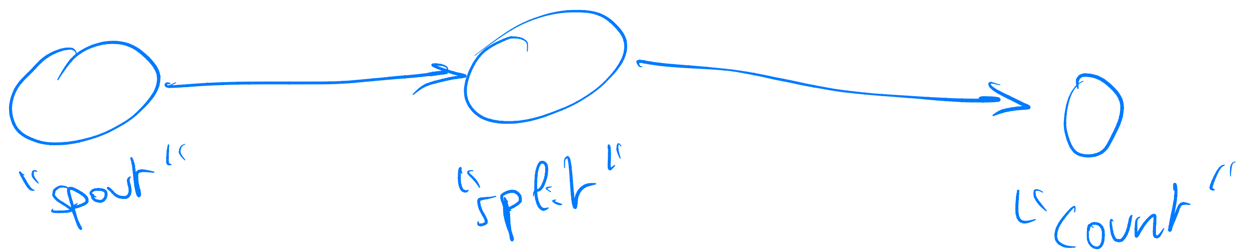
```
public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
  Integer v = tuple.getIntegerByField(field:"entier");
  basicOutputCollector.emit(new Values(v*2));

}
```

UNIVERSITÉ CÔTE D'AZUR

Writing topologies

- A topology links Spouts and Bolts with Streams
- Created using a *TopologyBuilder*
- Describe the topology
  - Each component has a String id used for linking
    - Order is destination, source
  - Must specify the type of grouping
  - Optionally specify the parallelism level and number of tasks

```java
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(id:"spout", new RandomSentenceSpout());
builder.setBolt(id:"split", new SplitSentence()).shuffleGrouping(componentId:"spout");
builder.setBolt(id:"count", new WordCount()).fieldsGrouping(componentId:"split", new Fields(...fields:"word"));
```

Wrapping-up everything

- Write startup code in a *main* method
    1. Create TopologyBuilder
    2. Describe your topology
    3. Create an instance of *org.apache.storm.Config*
    4. Write start-up/submission code
- Build with Maven

Wrapping-up everything

storm jar ...

- Execution like with Hadoop
  - Submit jar with the *storm* command
- Execution can be local or cluster-based
  - Specified in the source code

```
if (args != null && args.length > 0) {
  StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
} else {
  LocalCluster cluster = new LocalCluster();
  cluster.submitTopology(topologyName:"word-count", conf, builder.createTopology());
  Thread.sleep(10000);
  cluster.shutdown();
}
```

Windowing Support

- Storm supports both sliding and tumbling windows
- Implemented in
  - interface *IWindowedBolt*
  - Class *BaseWindowedBolt*
- When instancing Bolt, specify
  - Length of window
  - Sliding interval in number of tuples
- Length of window can be time-based of number-based
- If time-based, late tuples are logged by default