



NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS
SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION


A STORM ARCHITECTURE FOR FUSING IOT DATA

**A framework on top of Storm's streaming
processing system**

Zampouras A. Dimitrios

Supervisor: Associate Prof. Hadjiefthymiades Stathes

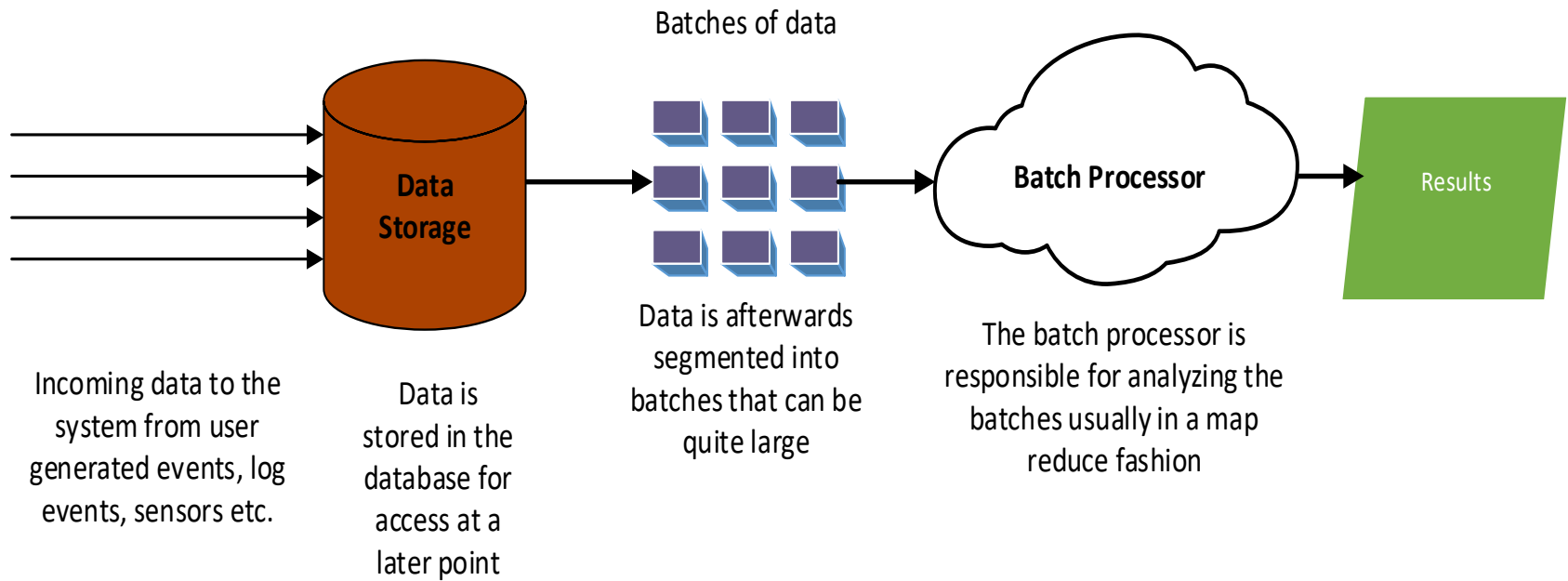
Internet of things - IOT

- By 2020, 25 billion devices will be connected to mobile networks worldwide
- 5G to support connectivity
- Huge amount of data generated  Big Data

Traditional data processing systems prove inadequate

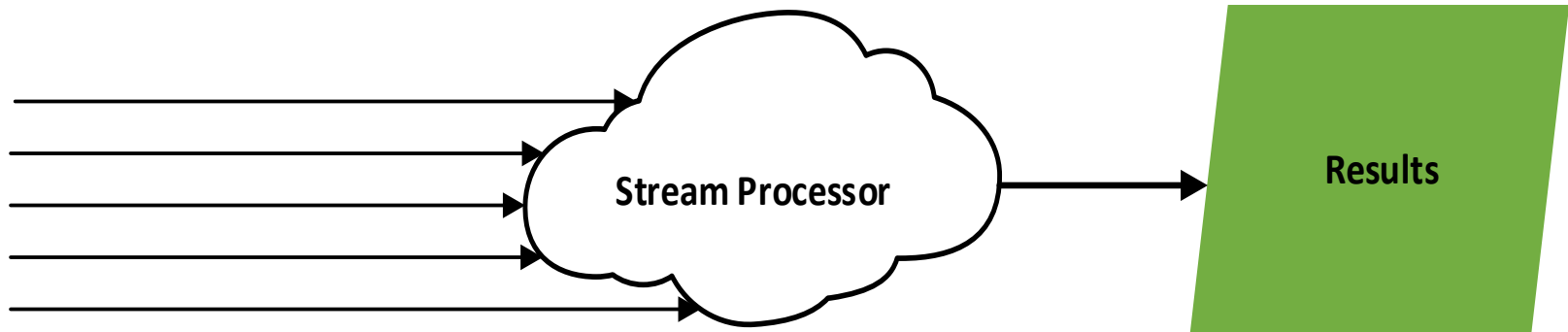
Big data processing systems (1/2)

□ Batch processing systems



Big data processing systems (2/2)

- Stream processing systems



Incoming data to the system from user generated events, log events, sensors etc.

Data is being processed individually, each time it enters the system

Batch & Stream processing systems

- Apache Hadoop,
 - based on the map-reduce model
 - partition data into batches
 - schedule and handle execution lifecycle of cluster
- Apache Spark,
 - Hybrid processing system(batch & stream)
 - Caching of intermediate results for faster processing
 - Almost real time

Batch & Stream processing systems

- Apache Samza,
 - real time stream processing system
 - uses Apache Kafka and Hadoop YARN
 - uses containers
- Apache Kafka Streams,
 - A library that can perform stream processing on top of Kafka queue
 - output to topics

Batch & Stream processing systems

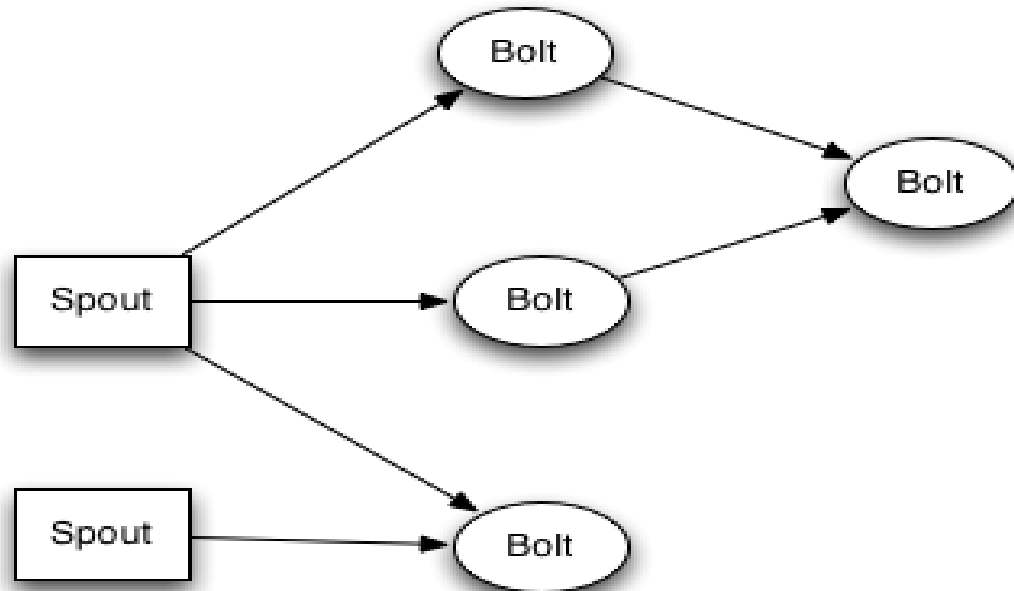
- Apache Flink,
 - supports stream and batch processing,
 - fault tolerant
 - supports exactly-once-processing

Apache Storm

- Real-time computation system
- scalable and fault tolerant
- ability to implement many components in many programming languages
- Creation of logical topologies, directed acyclic graphs(DAGs) of computation

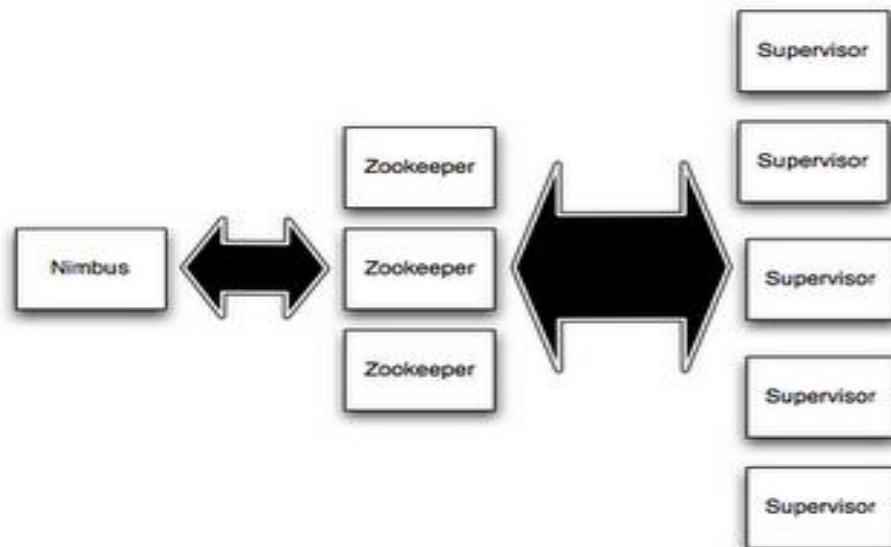
Storm core concepts - Components

- Spouts, responsible to fetch data into the system
- Bolts, the logical processing unit
- Spouts and bolts chained together form a logical computation graph



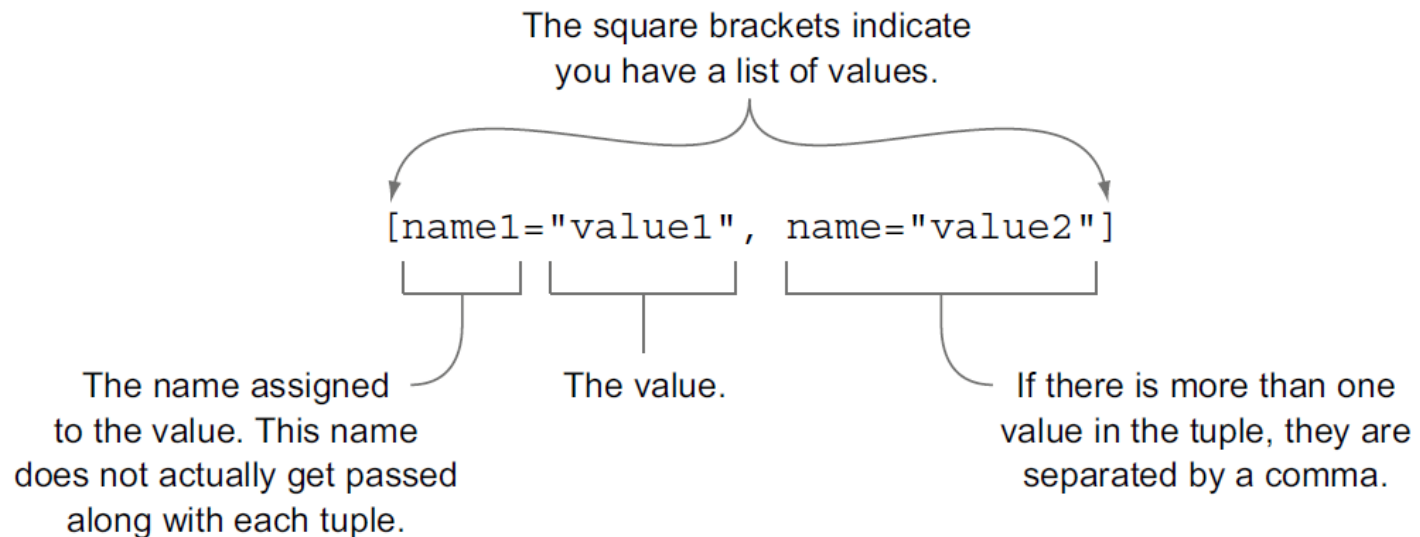
Storm core concepts - Cluster management

- Nimbus – master node responsible for assigning tasks to executors/worker nodes
- Cluster of apache zookeeper for coordination of resources
- Worker nodes run a Supervisor daemon responsible for executing tasks



Storm core concepts – Tuple

- Abstract data structure passed between spouts and bolts
- A wrapper of objects, an ordered list of values where each value has a “name”, a label



Storm core concepts – Streams

- Streams are the core abstraction in Storm
- An unbounded sequence of tuples
- Every component in Storm(spout/bolt) is responsible for creating one or more streams.
- A name needs to be given, otherwise “default”
- The declaration of a stream and its fields happen with the ***declareOutputFields*** method

Example Spout

```
public class MessageSpout implements IRichSpout {  
    private SpoutOutputCollector collector;  
  
    @Override  
    public void open (Map conf, TopologyContext context, SpoutOutputCollector collector) {  
        this.collector = collector;  
    }  
  
    @Override  
    public void nextTuple() {  
        collector.emit(new Values(sender, recipient, message));  
    }  
  
    @Override  
    public void declareOutputFields (OutputFieldsDeclarer declarer) {  
        declarer.declare(new Fields("from", "to", "message"));  
    }  
}
```

Example Bolt

```
public class LogBolt implements IRichBolt {  
    private OutputCollector collector;  
    Logger logger = Logger.getLogger(LogBolt.class);  
  
    @Override  
    public void prepare (Map conf, TopologyContext context, OutputCollector collector) {  
        this.collector = collector;  
    }  
  
    @Override  
    public void execute (Tuple tuple) {  
        logger.info("Logging call from " + tuple.getString(0) + " to " + tuple.getString(1)  
            + " message: " + tuple.getString(2));  
    }  
  
    @Override  
    public void declareOutputFields (OutputFieldsDeclarer declarer) { /*empty*/ }  
}
```

Parallelism in Storm

- Parallelism hints :

```
builder.setSpout( "word-spout", new WordSpout(), 4);
```

```
builder.setBolt( "count-bolt", new WordCount(), 10);
```

We can indicate to Storm that we would like four instances of WordSpout and ten instances of WordCount running in parallel

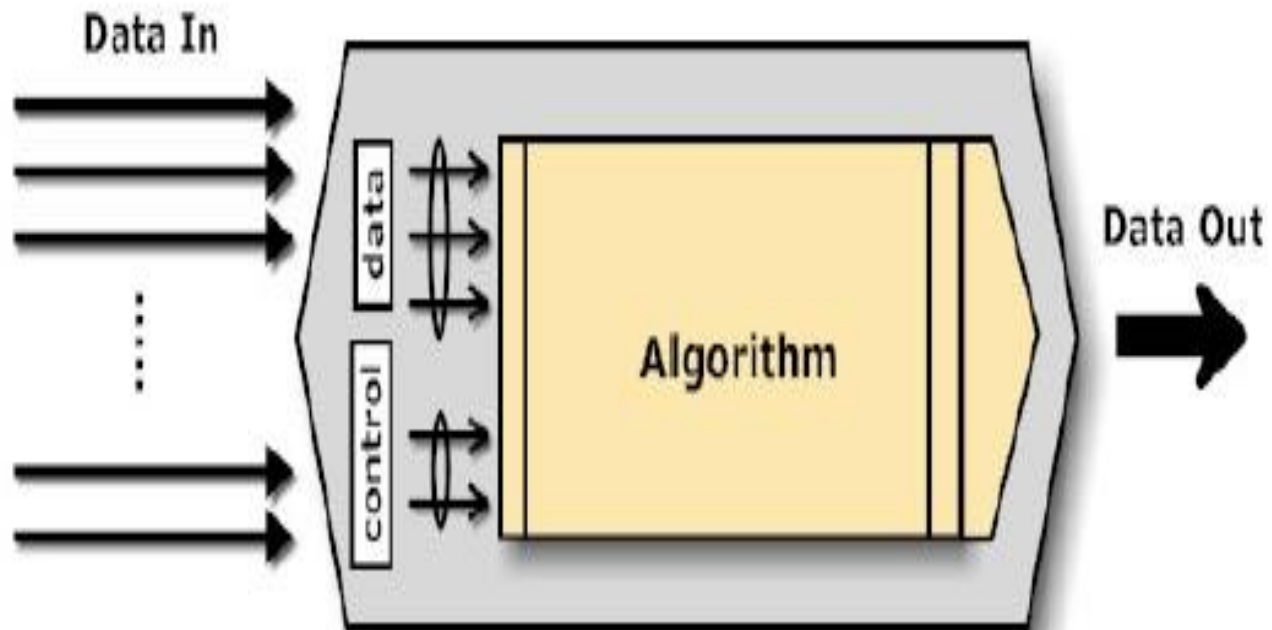
Stream Groupings

- A way to control how tuples are being sent between instances of components
- Types of grouping(most important):
 - ▣ **Shuffle grouping**, where tuples are being sent randomly to instances
 - ▣ **Global grouping**, the entire stream goes on a single bolt instance
 - ▣ **All grouping**, where the stream elements are copied and sent to each of the bolt's instances
 - ▣ **Direct grouping**, where we decide the recipient of the tuples
 - ▣ **Fields grouping**, partition the stream based on some of the fields

Fusion Box - Contextors

- Processing engine by chaining contextors
- Workflow processing, contextors create a graph of computation
- Contextors – main building blocks that encapsulate:
 - ▣ Logical processing – algorithm
 - ▣ Each contextor performs a transformation of the incoming data according to the algorithm to produce the outgoing data

Contextor



Storm as a data fusion engine

Storm is not ready to become a data fusion engine.

Modifications/Requirements:

- ▣ A way to describe topologies, a DSL language
- ▣ A framework that will handle connecting and parameterizing the interconnected components
- ▣ A complete set of algorithms that data scientists can use
- ▣ The user shouldn't need to code anything at all

Meet Apache Flux

Apache Flux is a framework for creating and deploying Storm topologies

It supports:

- Topology description and deployment via YAML files
- Definition of spouts/bolts

To achieve all that it heavily relies on reflection, which uses for creating the objects described

Essentially is really close to a bean engine

YAML topology example (1 / 3)

name: "yaml-topology"

config:

topology.workers: 1

components:

spout definitions

spouts:

- id: "spout-1"

className: "org.apache.storm.testing.TestWordSpout"

parallelism: 1

YAML topology example (2/3)

bolt definitions

bolts:

- id: "bolt-1"

className: "org.apache.storm.testing.TestWordCounter"

parallelism: 1

- id: "bolt-2"

className: "org.apache.storm.flux.wrappers.bolts.LogInfoBolt"

parallelism: 1

YAML topology example (3/3)

#stream definitions

streams:

- name: "spout-1 -> bolt-1" # name isn't used (placeholder for logging, UI, etc.)

from: "spout-1"

to: "bolt-1"

grouping:

type: FIELDS

args: ["word"]

- name: "bolt-1 --> bolt2"

from: "bolt-1"

to: "bolt-2"

grouping:

type: SHUFFLE

Java equivalent topology example

```
public static void main(String[] args) throws Exception {  
    TopologyBuilder builder = new TopologyBuilder();  
    TestWordSpout testWordSpout = new TestWordSpout();  
    builder.addSpout("spout-1",testWordSpout,1);  
    TestWordCounter testWordCounter = new TestWordCounter();  
    builder.addBolt("bolt-1", testWordCounter,1)  
        .fieldsGrouping("spout1",newFields("word"));  
    LogInfoBolt logInfoBolt = new LogInfoBolt();  
    builder.addBolt("bolt-2",logInfoBolt,1)  
        .shuffleGrouping("spout-1");  
    StormTopology topology = builder.buildTopology(...)  
    topology.run();  
}
```


Apache Flux – Properties (1 / 2)

- Component creation using classname
 - id: "zkHosts"
 - className: "org.apache.storm.kafka.ZkHosts"
- Constructor initialization
 - id: "zkHosts"
 - className: "org.apache.storm.kafka.ZkHosts"
 - constructorArgs:
 - "localhost:2181"
- Object referencing
 - id: "zkHosts"
 - className: "org.apache.storm.kafka.ZkHosts"
 - constructorArgs:
 - ref : "localhost-string" # a component with id localhost-string must be
present

Apache Flux – Properties (2/2)

□ Setting properties

- id: "spoutConfig"
className: "org.apache.storm.kafka.SpoutConfig"
properties:
 - name: "ignoreZkOffsets"
value: true
 - name: "scheme"
ref: "stringMultiScheme"

□ Calling methods

- id: "bolt-1"
className: "org.apache.storm.flux.test.TestBolt"
configMethods: # public void withFoo(String foo);
 - name: "withFoo"
args:
 - "foo"

Advancing the framework

- Flux enables us to create topologies without writing java code

However ...

- How can we encapsulate algorithms inside bolts?
- Spouts and bolts define fields and streams. How can we abstract this?

We could ...

- Make each algorithm extend the Bolt class
- Provide functions so that the user can specify fields and streams at each component

Algorithm interface

Instead of making every algorithm an extension of Bolt class we could enforce the Bolt class to hold an interface of an algorithm:

```
public interface IAlgorithm {  
    void prepare();  
    void Values execute(Tuple incomingTuple);  
}
```

Then the Bolt becomes:

Generic Bolt (1 / 3)

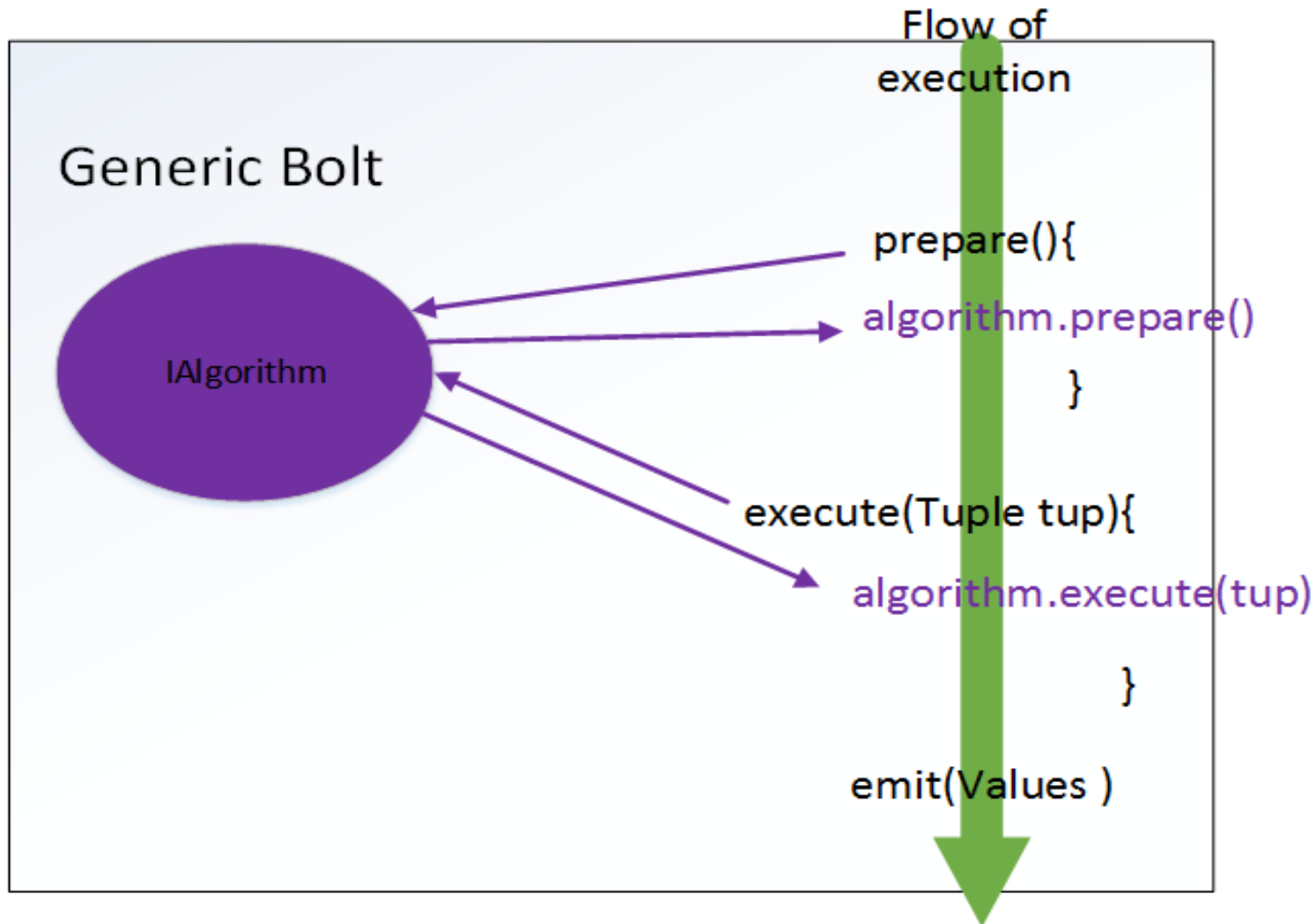
```
public class GenericBolt implements IRichBolt {  
    IAlgorithm algorithm;  
  
    public void setAlgorithm(IAlgorithm algo) {  
        this.algorithm = algo;  
    }  
  
    @Override  
    public void prepare(Map map,  
                        TopologyContext topologyContext,  
                        OutputCollector outputCollector) {  
        this.algorithm.prepare();  
    }  
}
```

Generic Bolt (2/3)

@Override

```
public void execute(Tuple tuple) {  
    values = algorithm.executeAlgorithm(tuple);  
    emit(values);  
}  
}
```

Generic Bolt (3/3)



Stream Definition (1/5)

- Whatever is declared at the `declareOutputFields` method needs to be consistent to the stream declaration in the YAML file
 - ▣ The user would have to declare the fields for each bolt/spout in the topology more than once and also declare the streams
 - ▣ But the stream declaration also happens at the stream definition section of the YAML
 - ▣ Why should we force the user to repeat the stream definition at each component?

For example....

Stream Definition (2/5)

spouts:

- id: "spout-1"

className: "org.apache.storm.testing.TestWordSpout"

configMethods:

- name: "declareStreamWithFields"

args:

- "stream-1"

- "word"

- name: "declareStreamWithFields"

args:

- "stream-2"

- "word"

parallelism: 1

We are calling method *declareStreamWithFields* to register outgoing stream and fields since Storm require that every component does so.

In a simple example we had to call twice the same method with almost the same parameters.

Stream Definition (3/5)

streams:

- name: "spout-1 --> bolt-1"

from: "spout-1"

to: "bolt-1"

grouping:

streamId: "stream-1"

type: FIELDS

args: ["word"]

- name: "bolt-1 --> bolt2"

from: "bolt-1"

to: "bolt-2"

grouping:

streamId: "stream-2"

type: SHUFFLE

The user must also define it at the stream definitions.

Stream Definition (4/5)

streams:

- name: "spout-1 --> bolt-1"

from: "spout-1"

to: "bolt-1"

grouping:

streamId: "stream-1"

type: FIELDS

args: ["word"]

- name: "bolt-1 --> bolt2"

from: "bolt-1"

to: "bolt-2"

grouping:

streamId: "stream-2"

type: SHUFFLE

The user must also define it at the stream definitions.

Too complex and repetitive.



Stream Definition (5/5)

- Essentially each bolt in the topology graph implements a transformation of the incoming fields
- The algorithm is the main component that dictates the transformation

```
public interface FieldTransformer {  
    //applies a transformation to the incoming fields  
    //returning the new fields  
    public String[] transformFields( String incomingFields);  
}
```

Stream Definition $((5+1)/5)$

- Enable flux to connect the outgoing fields of the topology by requesting each algorithm to supply its outgoing fields
- This is done by traversing the topology graph by doing a Depth First Search
- Now the user does not need to specify the outgoing fields at each step, the framework takes care of it

Spout Definition(1 / 6)

- Implement Mqtt and Kafka consumers in a similar fashion
- Things in common with every message queue consumer
 - A connecting host,
 - Port,
 - The message topic,
 - And a message scheme, so that we can interpret the incoming messages

Spout Definition(2/6)

mqttconfig:

- id: "mqtt-config"
className: "flux.model.extended.MqttSpoutConfigDef"
brokerUrl: "tcp://localhost:1883"
topic: "health_monitor/blood_pressure"
clientId: "health_monitor"
regex: ", "

spouts:

- id: "blood-spout"
className: "consumers.MqttConsumerSpout"
constructorArgs:
 - ref: "mqtt-config"

All the parameters are passed via a configuration class

Spout Definition(3/6)

kafkaconfig:

- id: "kafka-config"
- className: "flux.model.extended.KafkaSpoutConfigDef"
- regex: ","
- zkHosts: "localhost:2181"
- topic: "health"
- zkRoot: "/health"
- clientId: "storm-consumer"

spouts:

- id: "kafka-spout"
- className: "consumers.FusionKafkaSpout"
- constructorArgs:
 - ref: "kafka-config"

All the parameters are passed via a configuration class

Spout Definition(4/6)

- id: "keyValueSchemeasMultiScheme"
 className: "org.apache.storm.kafka.KeyValueSchemeAsMultiScheme"
 constructorArgs:
 - ref: "fusionScheme"

- id: "zkHosts"
 className: "org.apache.storm.kafka.ZkHosts"
 constructorArgs:
 - "localhost:2181"

Spout Definition(5/6)

```
- id: "spoutConfig"  
  className: "org.apache.storm.kafka.SpoutConfig"  
  constructorArgs:  
    - ref: "zkHosts" # brokerHosts  
    - "health" # topic  
    - "/health" # zkRoot  
    - "storm-consumer" # id  
  properties:  
    - name: "bufferSizeBytes"  
      value: 4194304  
    - name: "fetchSizeBytes"  
      value: 4194304  
    - name: "scheme"  
      ref: "keyValueSchemeasMultiScheme"  
  spouts:  
    - id: "kafka-spout"  
      className: "org.apache.storm.kafka.KafkaSpout"  
      constructorArgs:  
        - ref: "spoutConfig"
```

Spout Definition(6/6)

- How is a spout going to resolve the fields and the primitive types?
 - Define each class type so that the primitives get resolved

mqttconfig:

```
- id: "mqtt-config"  
  className: "flux.model.extended.MqttSpoutConfigDef"  
  brokerUrl: "tcp://localhost:1883"  
  topic: "health_monitor/blood_pressure"  
  clientId: "hello"  
  regex: ", "  
  fields:  
    - "id"  
    - "value"  
    - "timestamp"  
  classes:  
    - "java.lang.String"  
    - "java.lang.Double"  
    - "java.lang.Long"
```

Mapping of classes happens at the topology creation.

We need the primitive types to be resolved to their actual type so that we can rely on reflection on the next algorithms(if ever is going to be needed).

Algorithms

- Shewhart, Cusum, Bayesian network
- Stream Merging algorithms
- Many more can be implemented
- Utility algorithms like threshold, median, max, min, field filter etc...
- Goal is to provide a complete working set

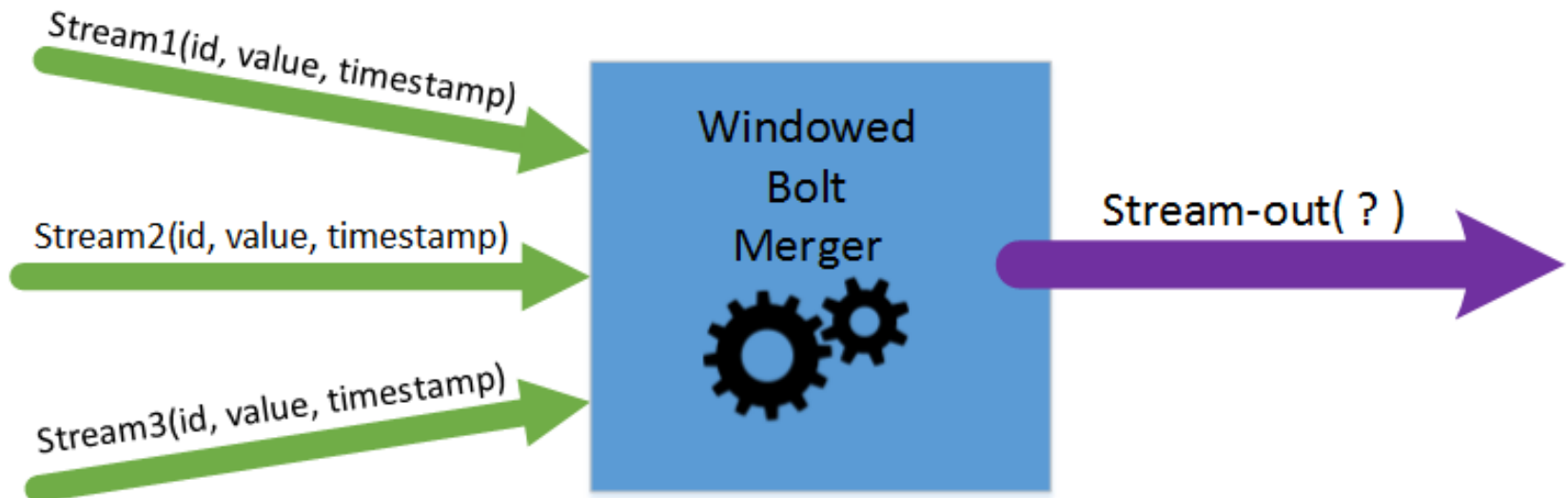
Stream merging(1 / 2)

- Storm provides windowed bolts that can merge unbound streaming data into finite sets
 - ▣ Sliding window
 - ▣ Tumbling window
- One important aspect is that we can merge those streams based on a timestamp field that each element carries.

This significantly changes the output fields when a topology contains such an algorithm(why?)

Stream merging(2/2)

- When merging streams into finite sets a significant transformation happens.
- The previous scheme with the use of transform fields could only support minor changes (add one fields, remove one etc.)



Challenge

- Each algorithm can be chained with any one*
- Each algorithm applies a transformation to the fields
- Some algorithms apply a complete overhaul
- The next algorithm must know how to access the fields

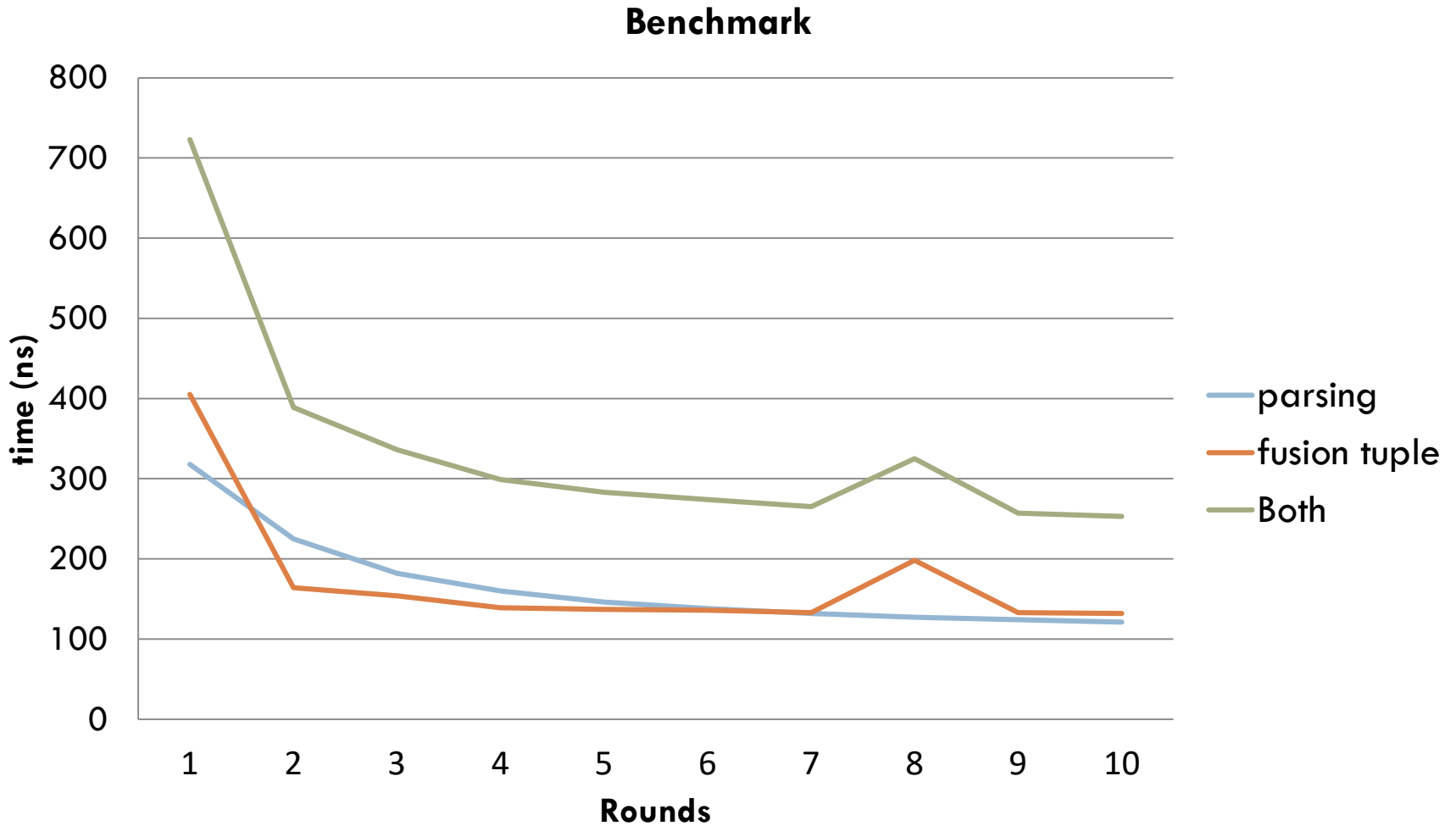
“What kind of structuring should we impose in order for each algorithm to be able to understand the incoming data?”

Meet the Fusion Tuples approach

- A fusion tuple is a data structure that can hold the maximum amount of information
- A map of streams to the fields, along with some metadata(field names, position and class)
- A merging algorithm is the most “severe” transformation that can happen to the fields

```
public class FusionTuple {  
    Map<String, List<Values>> valueMap;  
    Map<String, List<Meta>> metaMap;  
}
```


But comes with a cost



Improvise, adapt, overcome

- Supporting the fusion tuple scheme is costly, especially when used to transfer a simple array of values
- Serialization adds more computational cost
- Whoops, some grouping techniques cannot be used

Could we use the fusion tuple scheme only when needed?

Solution no. 1

- We could divide the algorithms to families according to what data structure they emit

Families of algorithms

Simple transformers
Emit plus one field, one field less etc.

Median

Min/Max

Threshold

CUSUM

Shewhart

Classification Algorithms

⋮

Simple-Stream

Window transformers

Emit multiple values, often a map structure would be helpful

Stream merger

Sliding window algorithms

Tumbling window algorithms

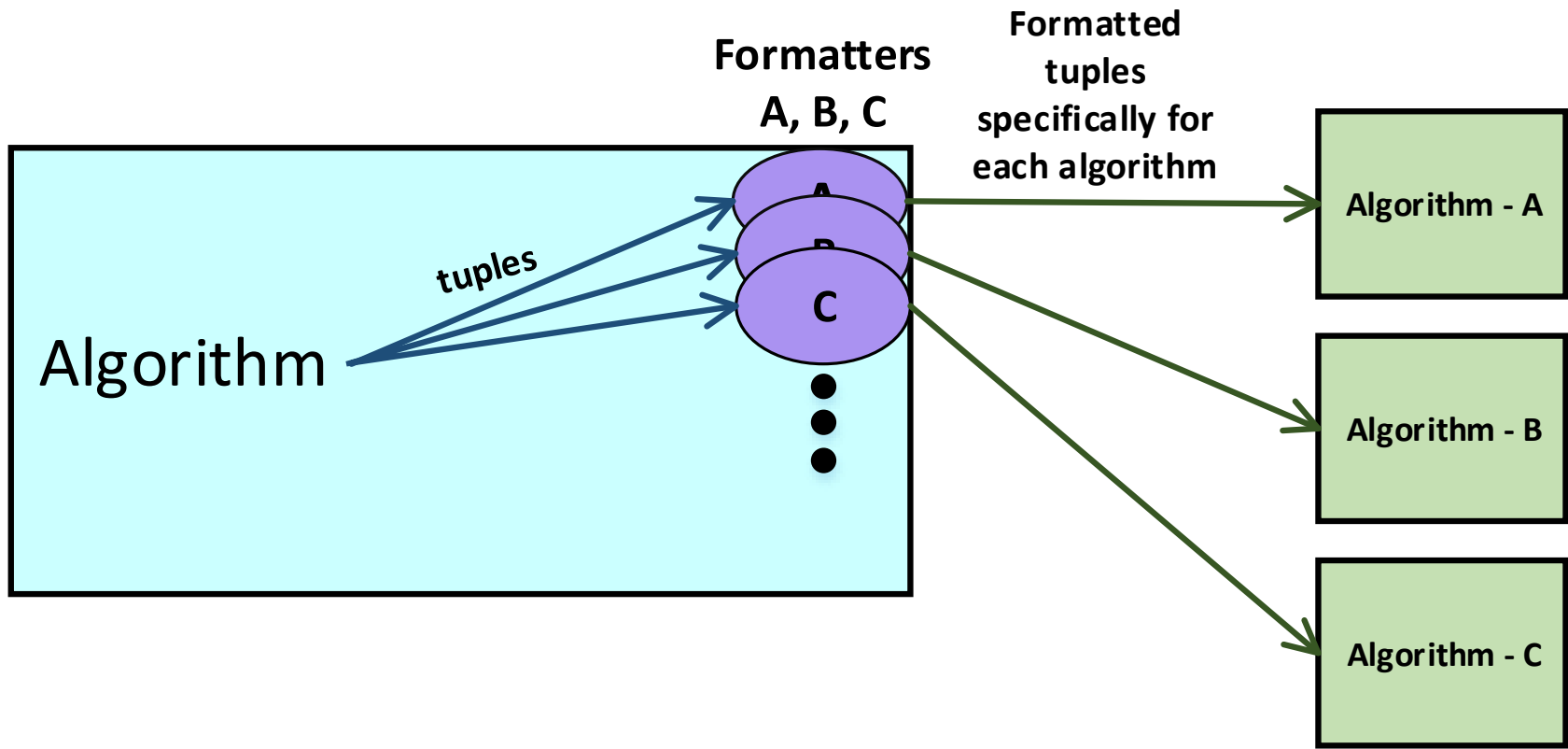
⋮

Window-Stream

Solution no. 2

- When chaining two algorithms we have to prototype their communication.
- The second algorithm expects incoming messages to uphold a certain structure
- What if each algorithm was coupled with a “formatter”?
- The algorithm could “lend” the formatter to anyone who would like to “speak” to them.

Solution no. 2 cont.




Conclusions

- No approach can come without cons, or pros for that matter
- More algorithms and scenarios that we would like to offer are needed
- At any time when we might have to change our approach when a new algorithm is presented

Questions?





“This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.”

- Winston Churchill