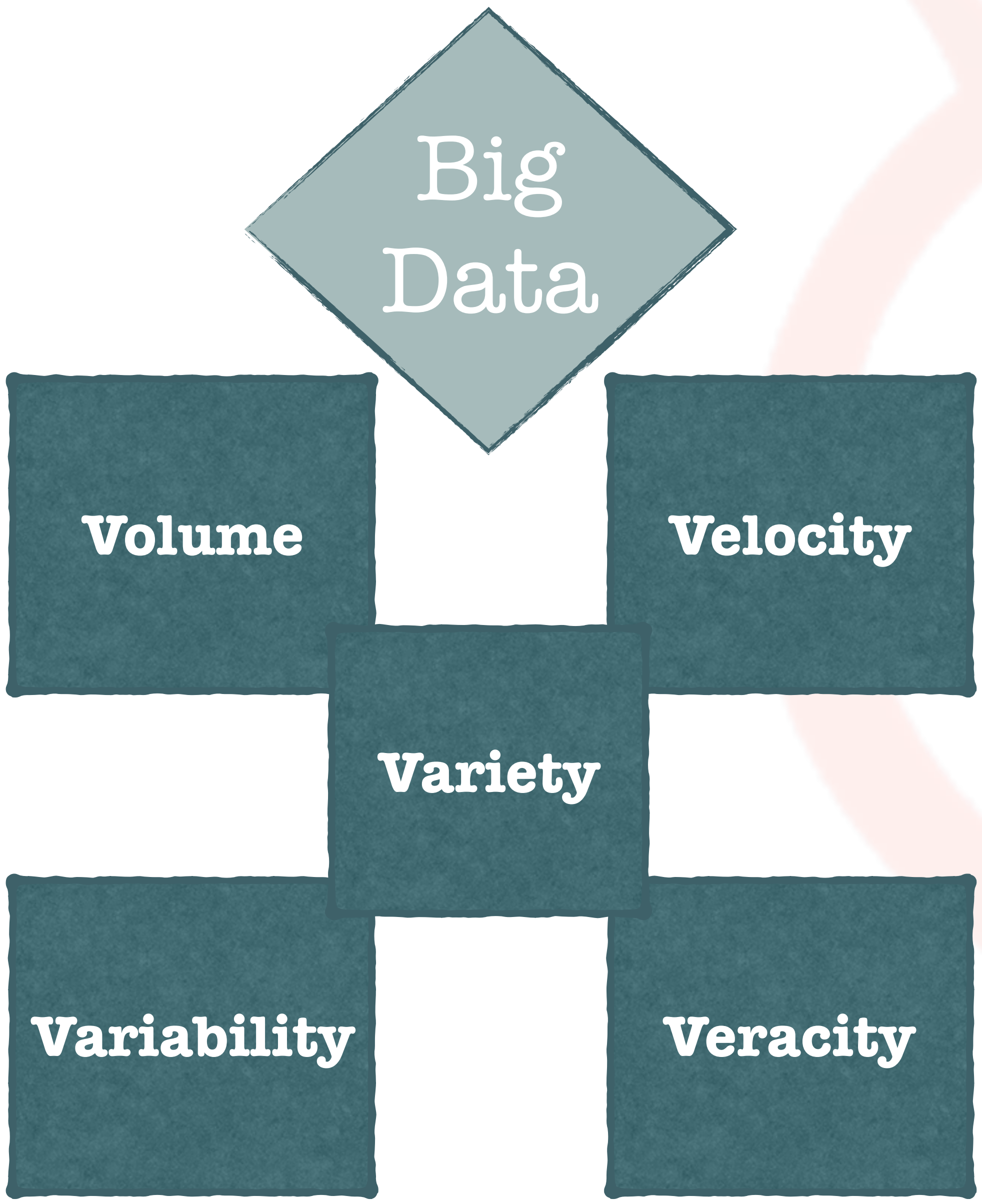
Models and Tools for Big Data Analytics in HPC

Claudia Misale

Università degli Studi di Torino
Parallel Computing Group

Big Data 5V-features



- High Volume of information generated and processed at high Velocity
- Information shape may Vary - structured or unstructured
- Information source may be Variable (e.g., images, text, …)
- Its accuracy is not guaranteed (e.g., incomplete data)

What to do with Big Data

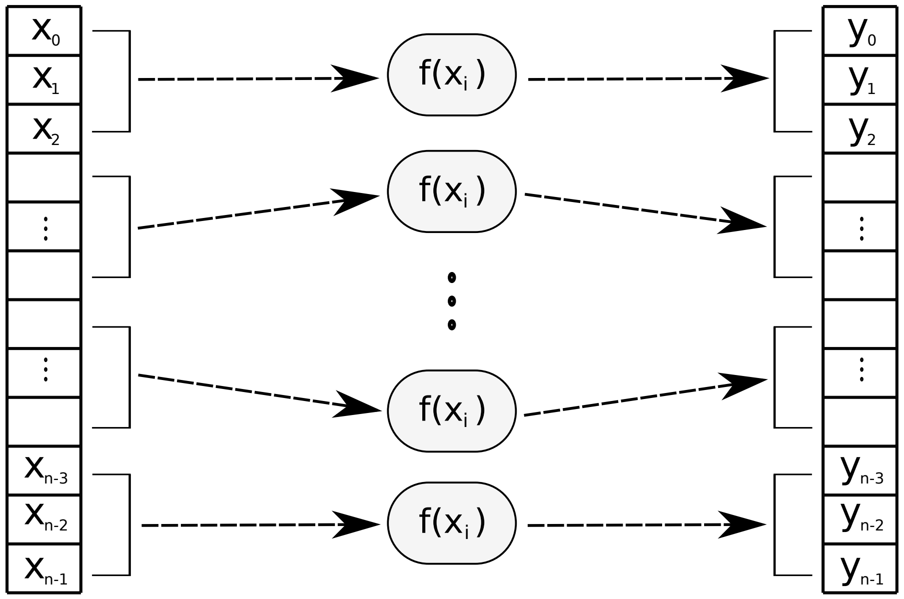
Analytics:

Extracting knowledge from (un)structured data

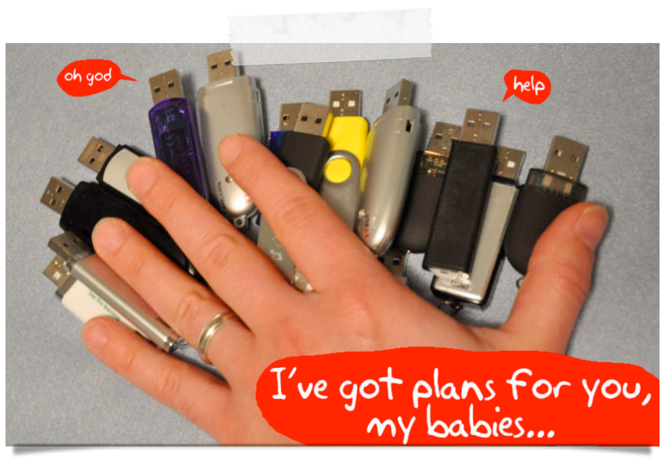
- Data retrieved in two ways
  - Statically - databases, data marts, data lake
  - Dynamically - live streams, high-frequency sources
- Two ways of processing: **Batch** and **Stream**

Batch Processing

- Process data with fixed size
- Stored on disks
- Analysis considering the whole input or part of it
- Repeatable analysis



Map



Frameworks for Batch Processing

- Apache Spark
- Apache Flink
- Google Dataflow
- Hadoop MapReduce

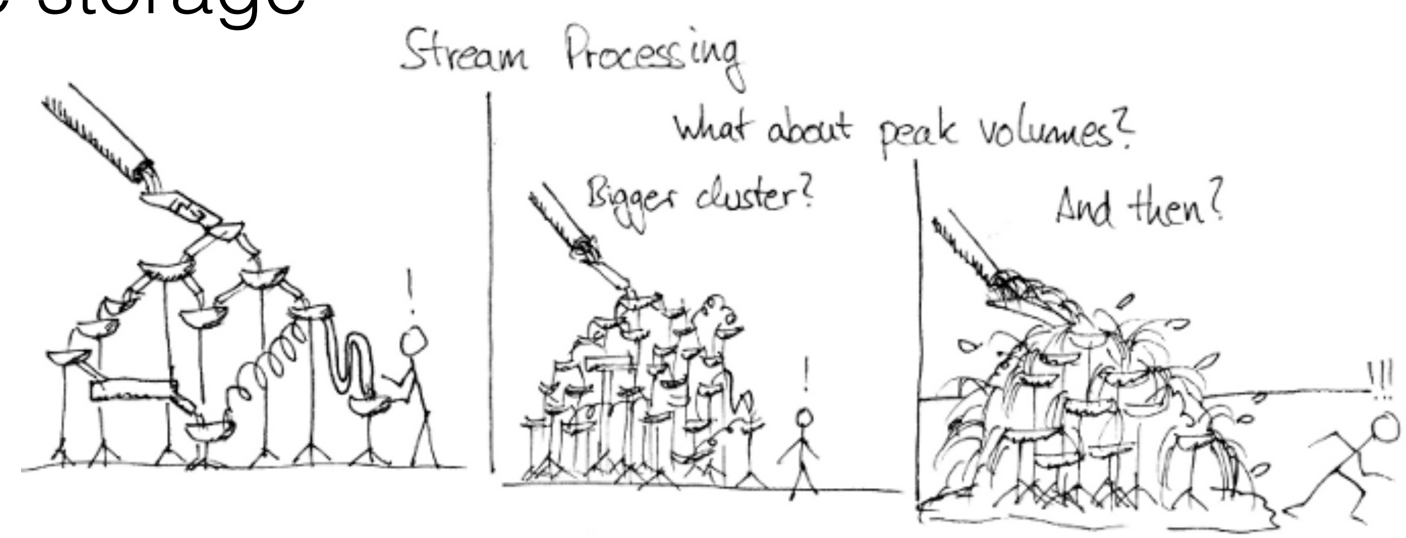
Specialised (machine learning)

- Google Tensorflow, Caffe, Torch

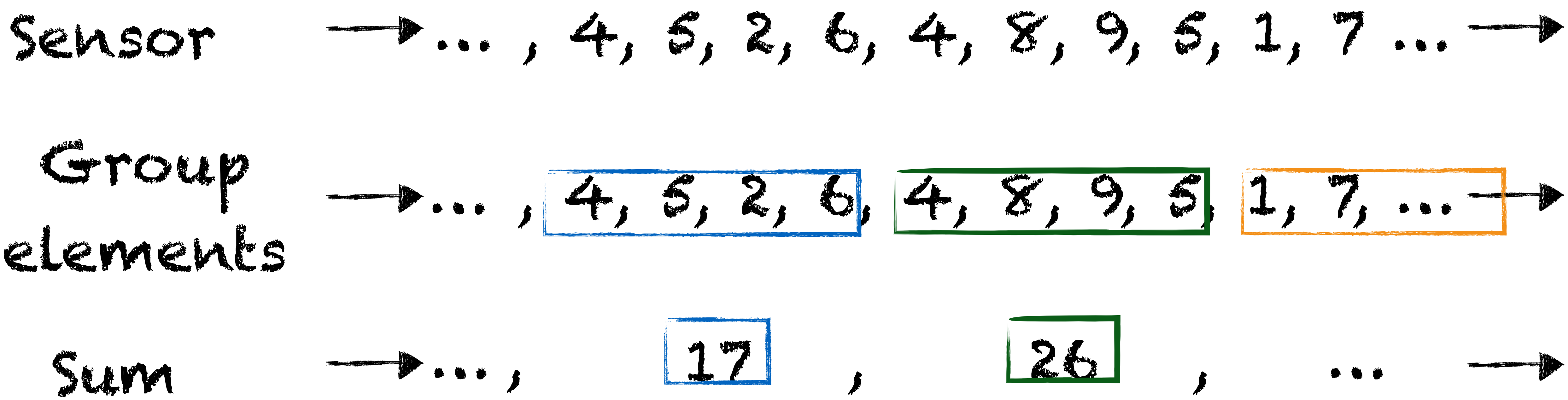
Mainly implemented in Java/Scala

Stream Processing

- Process data possibly unbounded (stream)
- Generated by a live source — Real Time processing
- Analysis considering continuous subsets of the stream — Pipeline parallelism
- Not repeatable unless infinite storage



Stream Processing



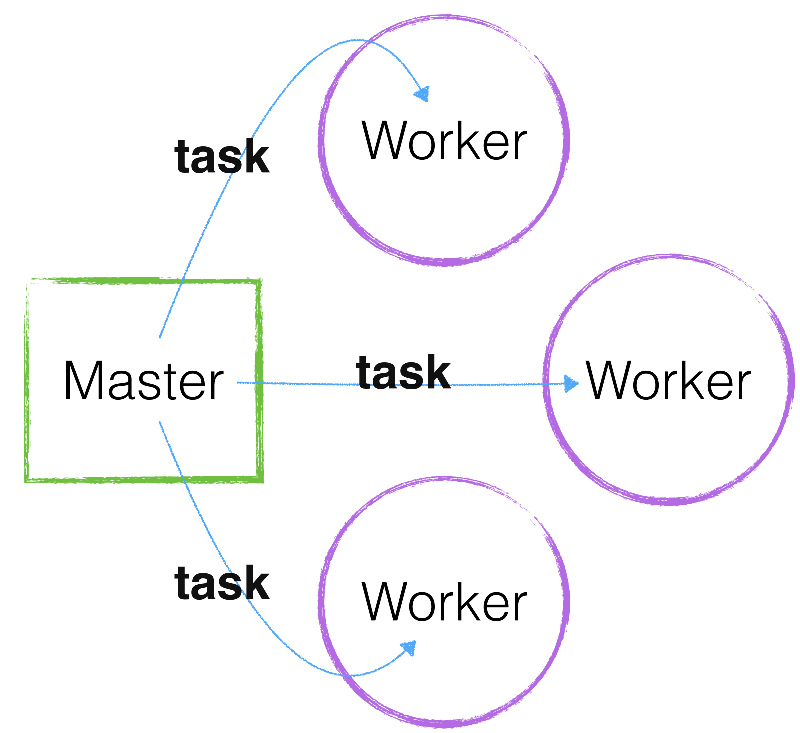
Frameworks for Stream Processing

- Apache SparkStreaming
- Apache Flink
- Apache Storm
- Apache Kafka
- Google Dataflow

All implemented in Java/Scala

Everything I do, I do it for you..

- Analytics requires tools having strong requirements
  - Programmability and ease of use
  - Able to process data from different sources
  - Performance (throughput, latency, …)
- Write on one, run on more
  - Parallelism on shared memory
  - Parallelism on distributed memory
  - Data Parallelism / Pipelining



State of the art

- There are an uncountable number of tool for Analytics
- Common aspects: dataflow model + functional style API
- Different API: Topological vs Declarative

Dataflow Model

Functional style

Declarative API

1. *Methods on objects* representing collections

2. *Functions on values*

- Both are algebras on **finite, (un)ordered datasets**

- APIs exposing a functional-like style

```
JavaRDD<String> words = textFile.flatMap(new
FlatMapFunction<String, String>() {
      public Iterable<String> call(String s) {
        return Arrays.asList(s.split(" "));
      }
    });

  JavaPairRDD<String, Integer> pairs =
    words.mapToPair(new PairFunction<String, String,
Integer>() {
      public Tuple2<String, Integer> call(String s) {
        return new Tuple2<String, Integer>(s, 1);
      }
    });

  JavaPairRDD<String, Integer> counts =
    pairs.reduceByKey(new Function2<Integer, Integer,
Integer>() {
      public Integer call(Integer a, Integer b) {
        return a + b;
      }
    });
```

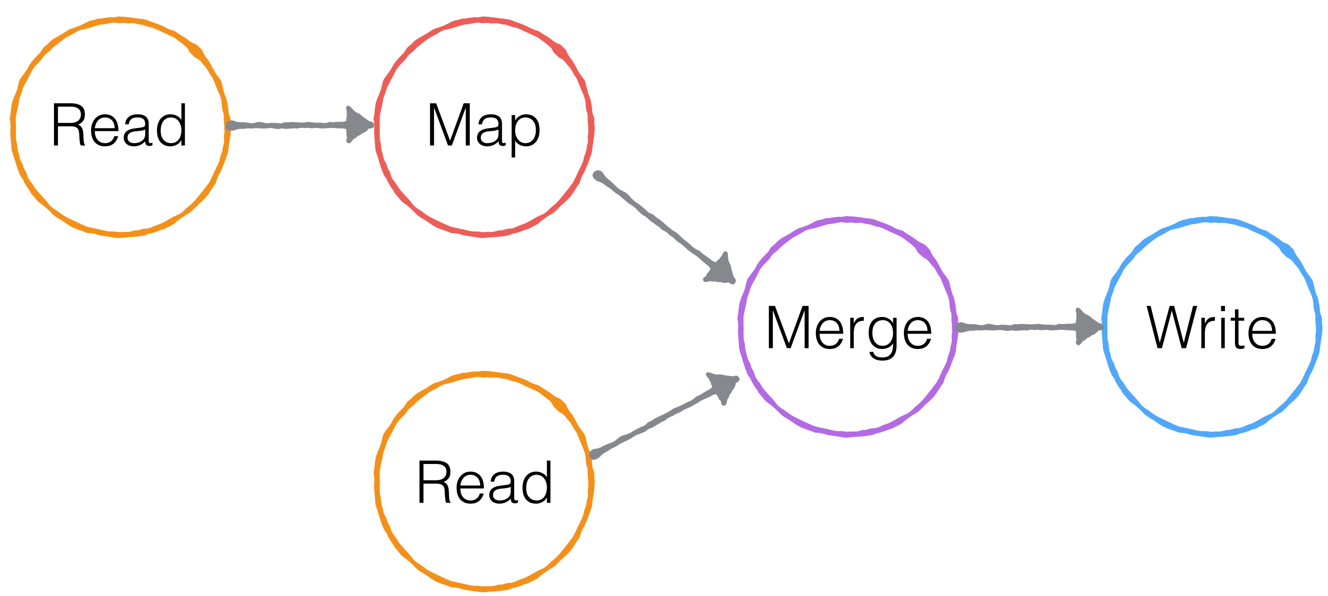
Spark code for Word Count

Topological API

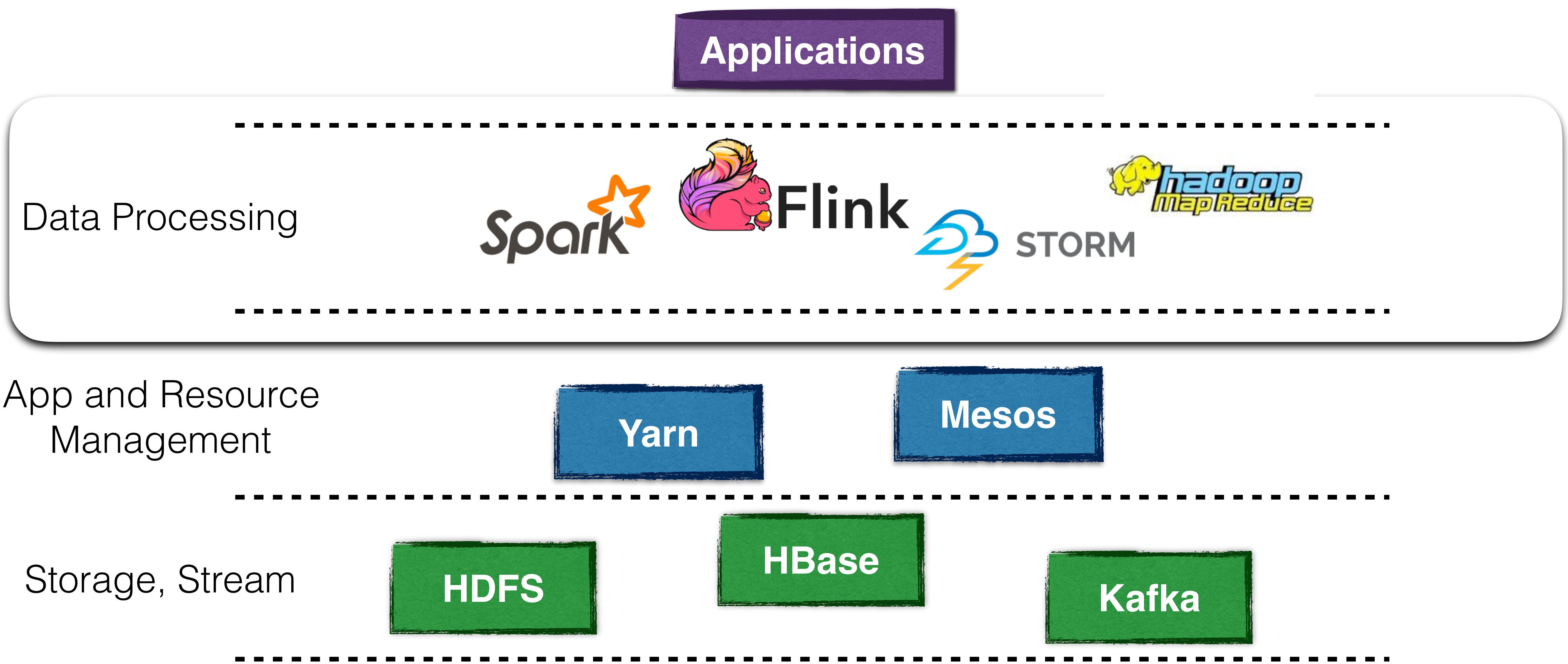
1. Programs are expressed as graphs built by explicitly connecting processing nodes — Dataflow graphs

2. Graph nodes are provided with the code defined by the user

- Model suitable for both bound and unbound data — more abstract



Analytics Frameworks Overview



Google MapReduce

Google MapReduce

- Both programming model and implementation (most famous HadoopMR)

- At very higher level is based on the composition of a Map and a Reduce function (old concept of functional languages)

- Exposes a functional style declarative API

  1. *Methods on objects* representing collections

  2. *Functions on values*

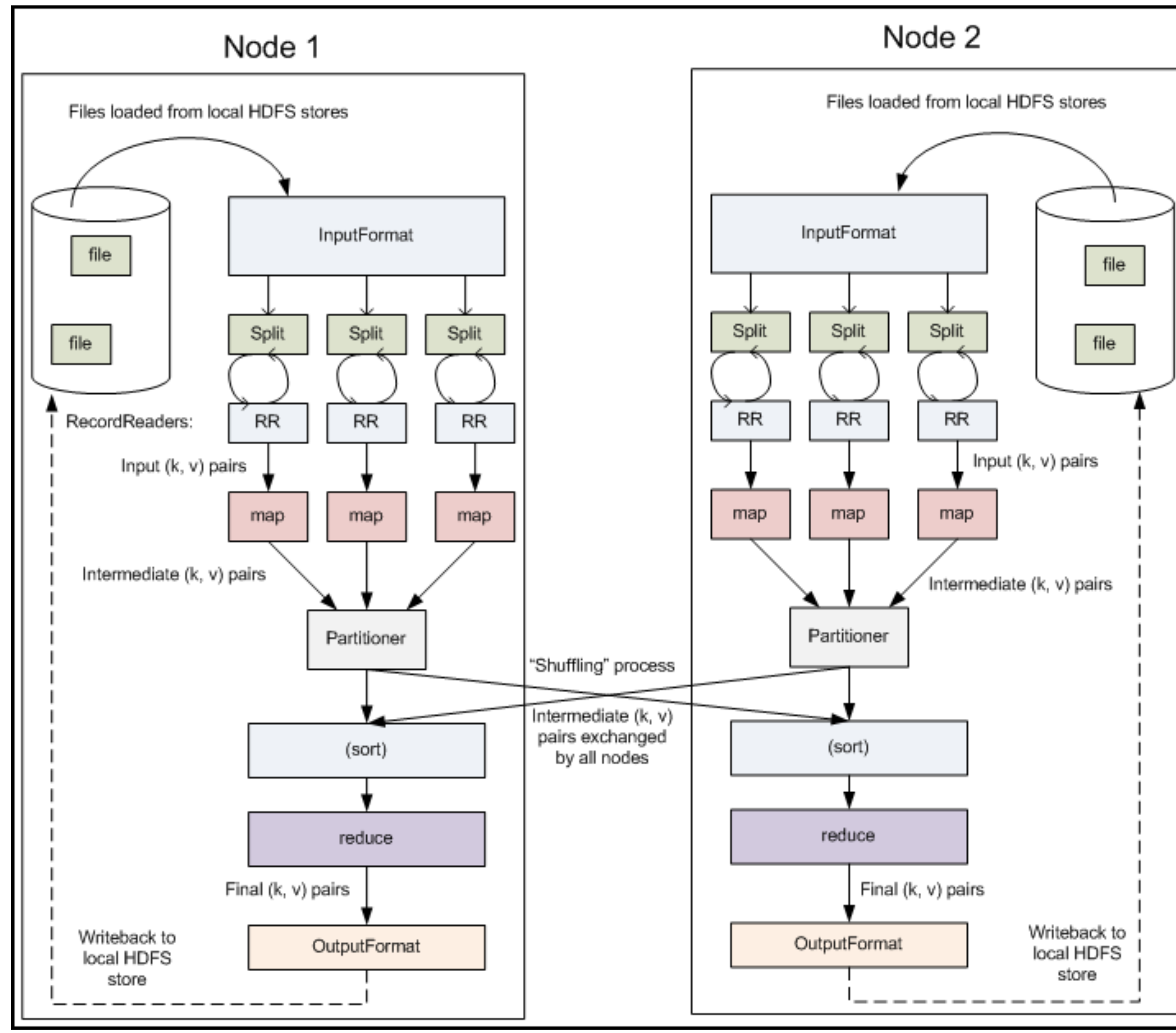
Google MapReduce Contribution to SOTA

- Not in the use of a map+reduce function
    - which is simplified by the API
- Novelty in the **key-value model** underlying the programming and execution model
- **Shuffle** phase for data repartition
    - idea of "*moving computation to data*"
        - Runtime exploits the natural data partitioning on distributed FS by forcing operations to be computed using only local data

MapReduce Execution Model

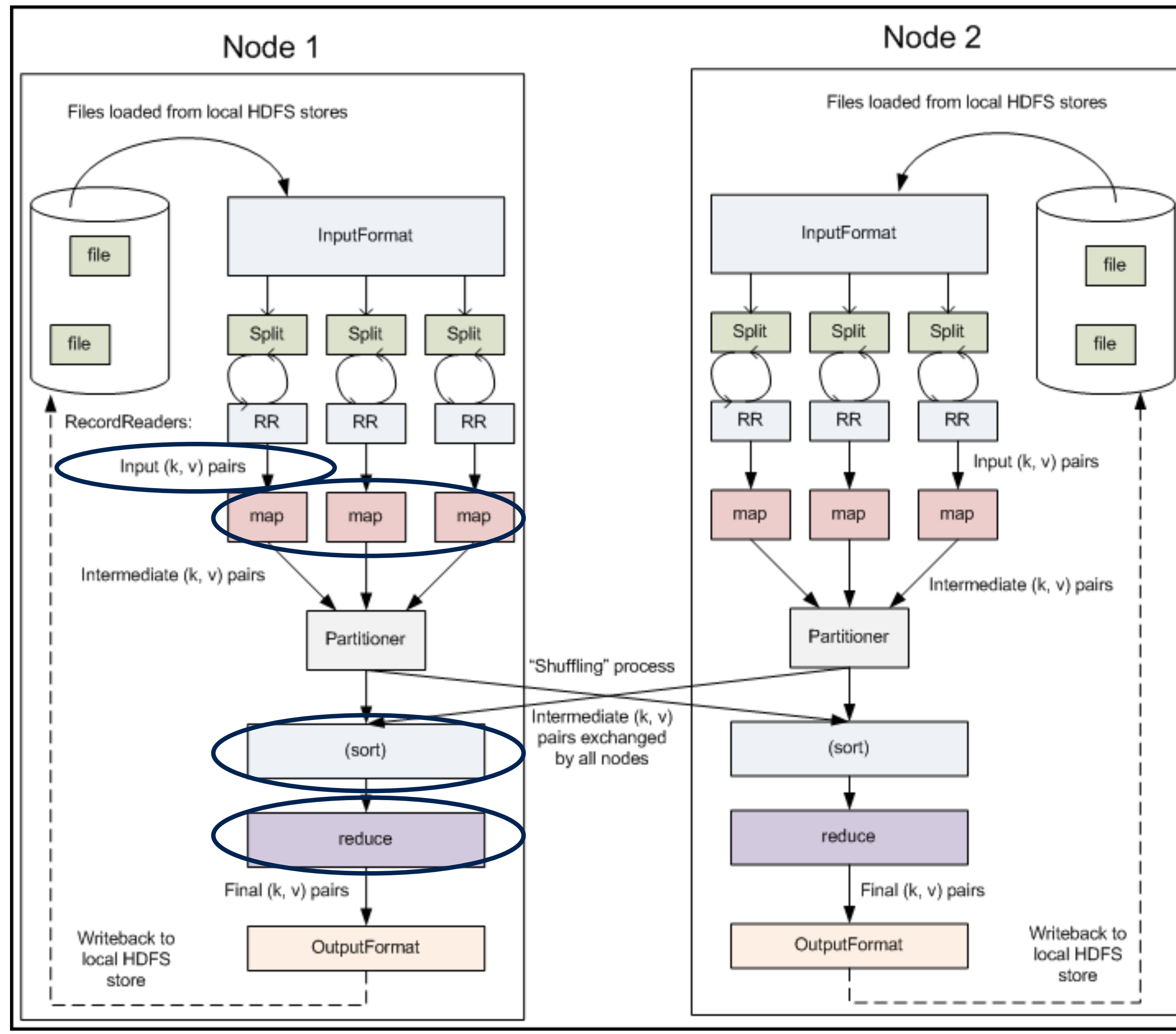
Five++ steps:

1. Input preparation
2. User defined **`map`** execution
3. Shuffle **`map`** output
4. Sort shuffled data
5. User defined **`reduce`** execution
6. Produce output



An Execution Model for Sorting

- Programming model influenced by the implementation
- Map input is KV pair
- Map output is shuffled, then ordered
- Reduce per key on ordered KV pairs



Limitations of MapReduce

- Not universal language
- MR tasks are acyclic dataflow programs
  - stateless map + stateless reduce
- Not performant on iterative computations
  - data stored on disk between map and reduce phases

Apache Spark

Apache Spark

- Batch (first) and Stream (then) processing framework
- Born to mainly implement iterative algorithms
- Declarative API
    1. *Methods on objects* representing collections
    2. *Functions on values*

Apache Spark Data Model

- Data model: **Resilient Distributed Dataset** (RDD)

  Immutable collection of objects partitioned across a cluster, that can be operated in parallel

- Two kind of operations: *Transformations* and *Actions*
  - resulting into a ***DAG*** of operations

```
JavaRDD<String> words = textFile.flatMap(new FlatMapFunction<String, String>() {
      public Iterable<String> call(String s) {
        return Arrays.asList(s.split(" "));
      }
    });
```

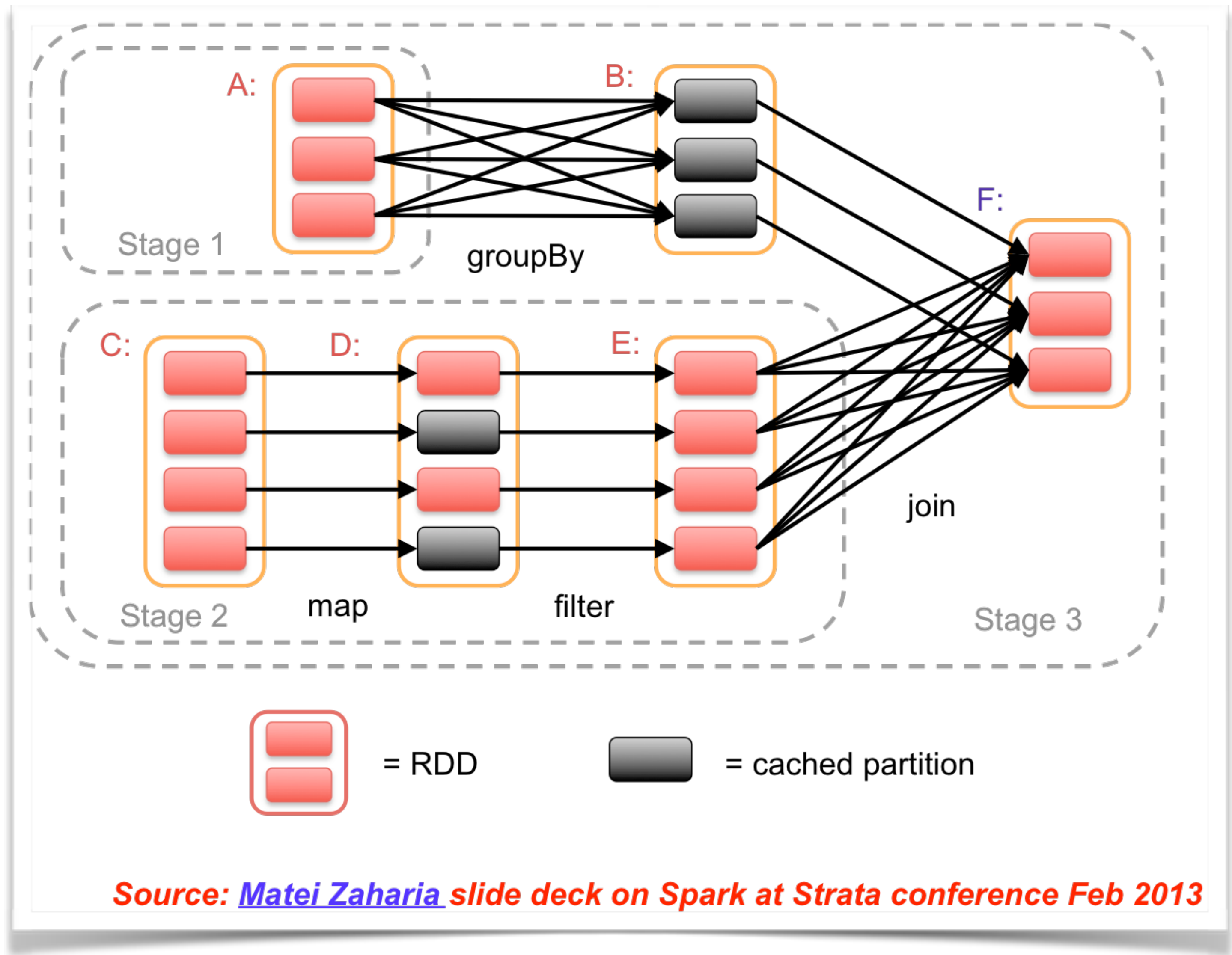
Spark RDD

- **Read only** collection partitioned across a cluster
- Created by *Transformations* triggered by *Actions*
- RDDs are **in memory** unless cached/persisted
  - more like an expression than a value
- **Lazy** and **ephemeral**: computed and materialised only by actions
- **Resilient**: lost RDD are recomputed automatically

A DAG in Spark

- Blocks = RDDs
- Arrows = dependencies
- Dependencies:
    1. **Narrow** (no shuffle)
    2. **Wide** (with shuffle)

- **Shuffle** on KV pairs (MapReduce model)
- Stages separated by wide dependencies

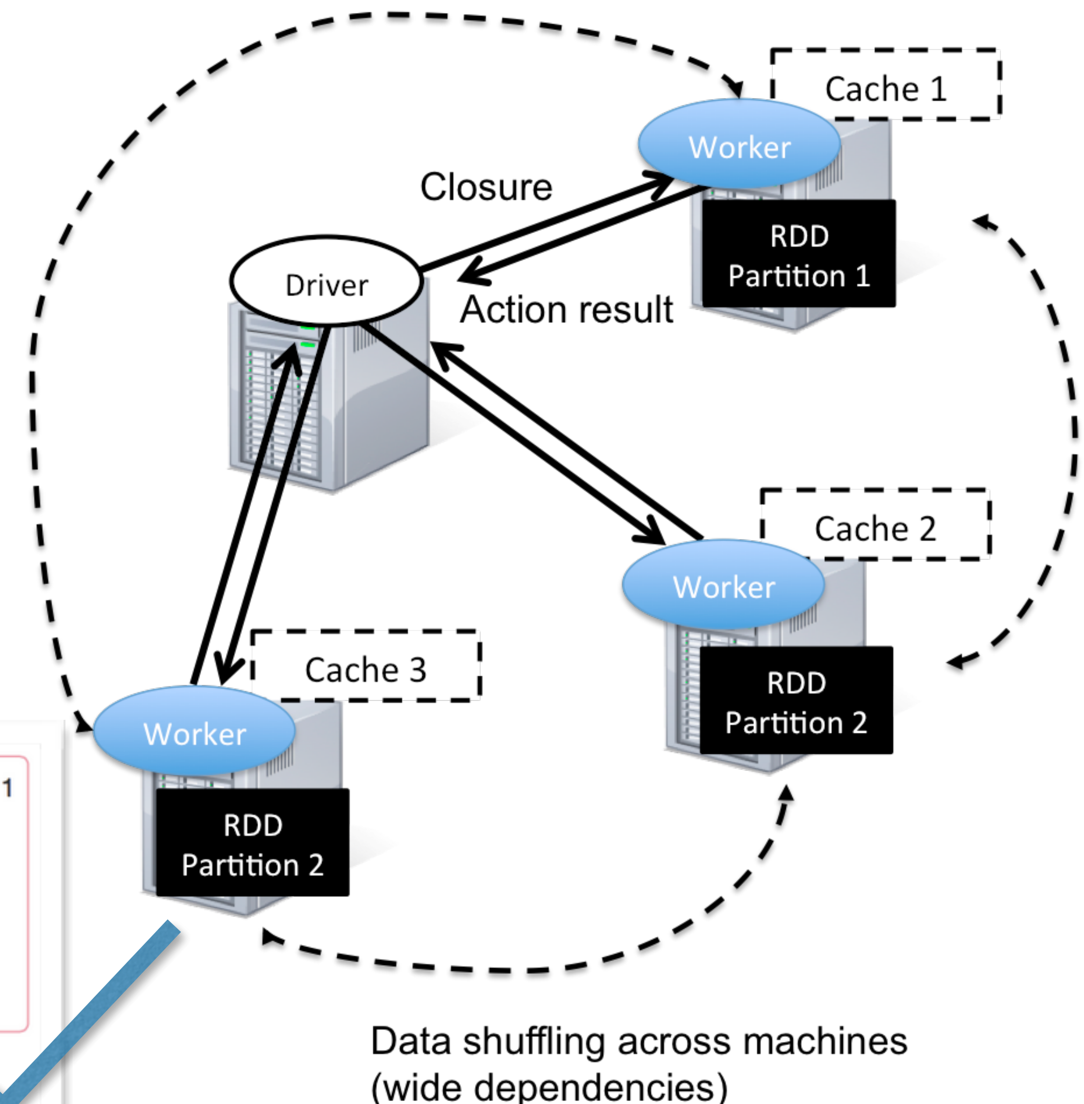


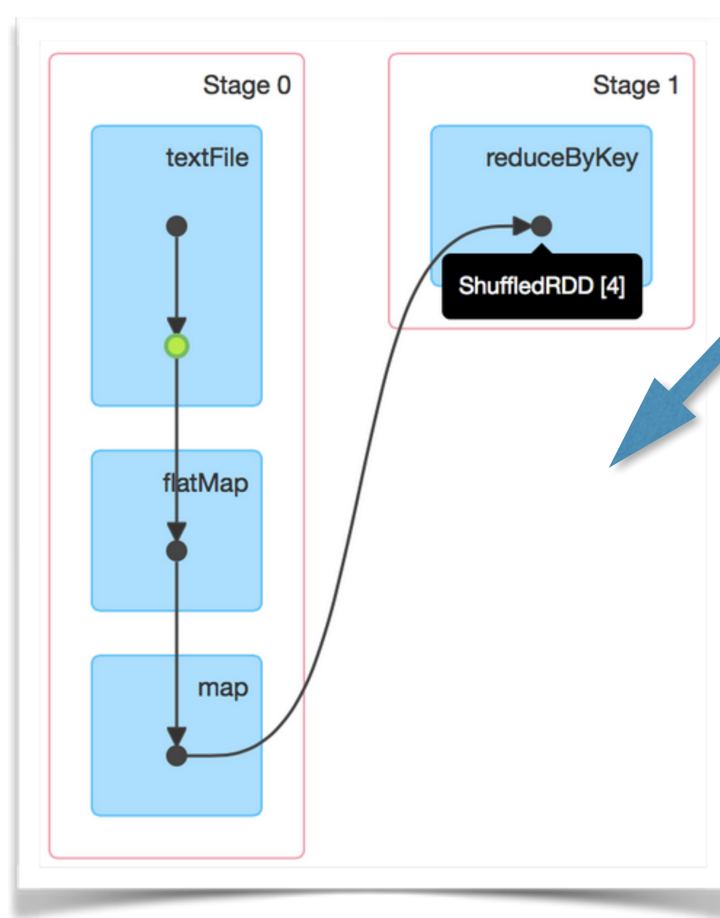
Parallel Execution Graph

Execution Model

- Master-Worker paradigm

1. Driver program
2. Worker DAG





Spark Streaming

- Built over Spark batch
- Data model: discretised stream *DStream*
    - continuous sequences of RDDs (micro-batch)
- Operations over DStream are forwarded to the underlying RDDs

Apache Flink

Apache Flink

- Framework for Batch and Stream processing
  - Runtime on stream processing (unlike Spark)
- Declarative API
  - Methods on objects representing collections

Apache Flink Data Model

- Data model: *DataStream* and *DataSet*
- ***DataStream***: abstraction representing a stream as a single object
- ***DataSet***: abstraction representing a collection as a stream with a single item
- An application results into a ***DAG*** of operations

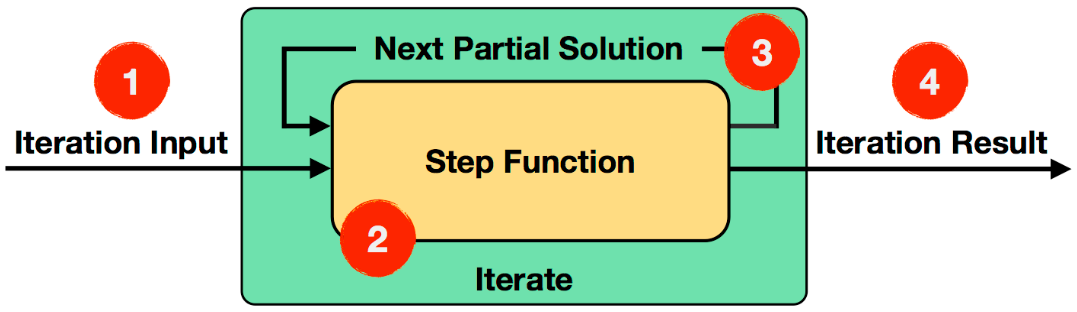
Word Count in Java

```
DataStream<Tuple2<String, Integer>> counts =
// normalize and split each line
text.map(line -> line.toLowerCase().split("\\W+"))
// convert splitted line in pairs (2-tuples) containing: (word,1)
.flatMap((String[] tokens, Collector<Tuple2<String, Integer>> out) -> {
// emit the pairs with non-zero-length words
    Arrays.stream(tokens)
        .filter(t -> t.length() > 0)
        .forEach(t -> out.collect(new Tuple2<>(t, 1)));
   })
   // group by the tuple field "0" and sum up tuple field "1"
   .keyBy(0).sum(1);
```

Application as a DAG

- Flink applications are mapped to streaming dataflow
  - Nodes are operations - arches represent streams
- Cycles are allowed with special *iteration constructs*
  - Differently from Spark - iterations are replication of DAG subgraphs

Iterative Computations



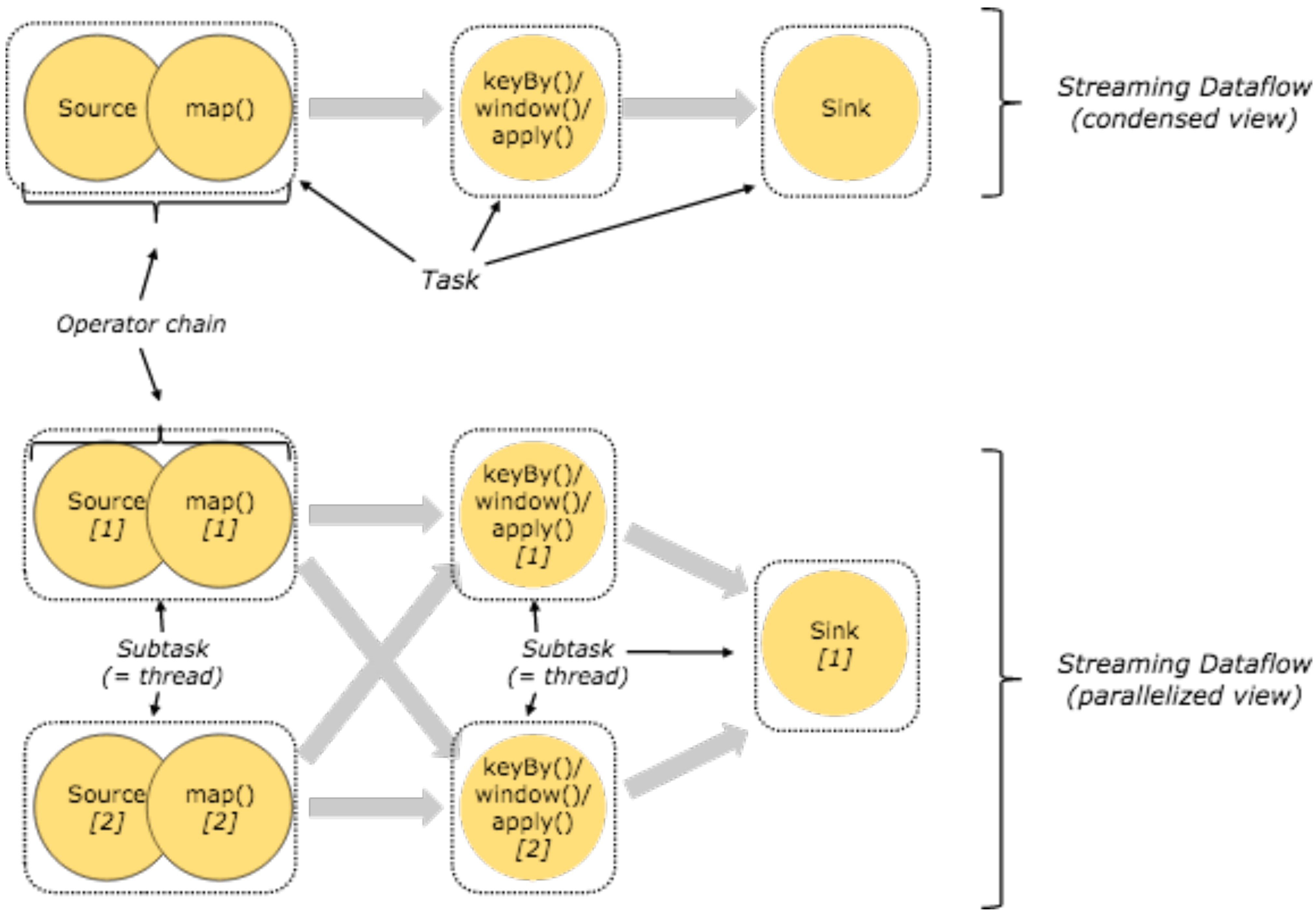
1. **Iteration Input**: Initial input for the first iteration from a data source or previous operators

2. **Step Function**: Arbitrary DAG executed in each iteration

3. **Next Partial Solution**: In each iteration, the output of the step function will be fed back into the next iteration

4. **Iteration Result**: Output to a data sink or to the following operators

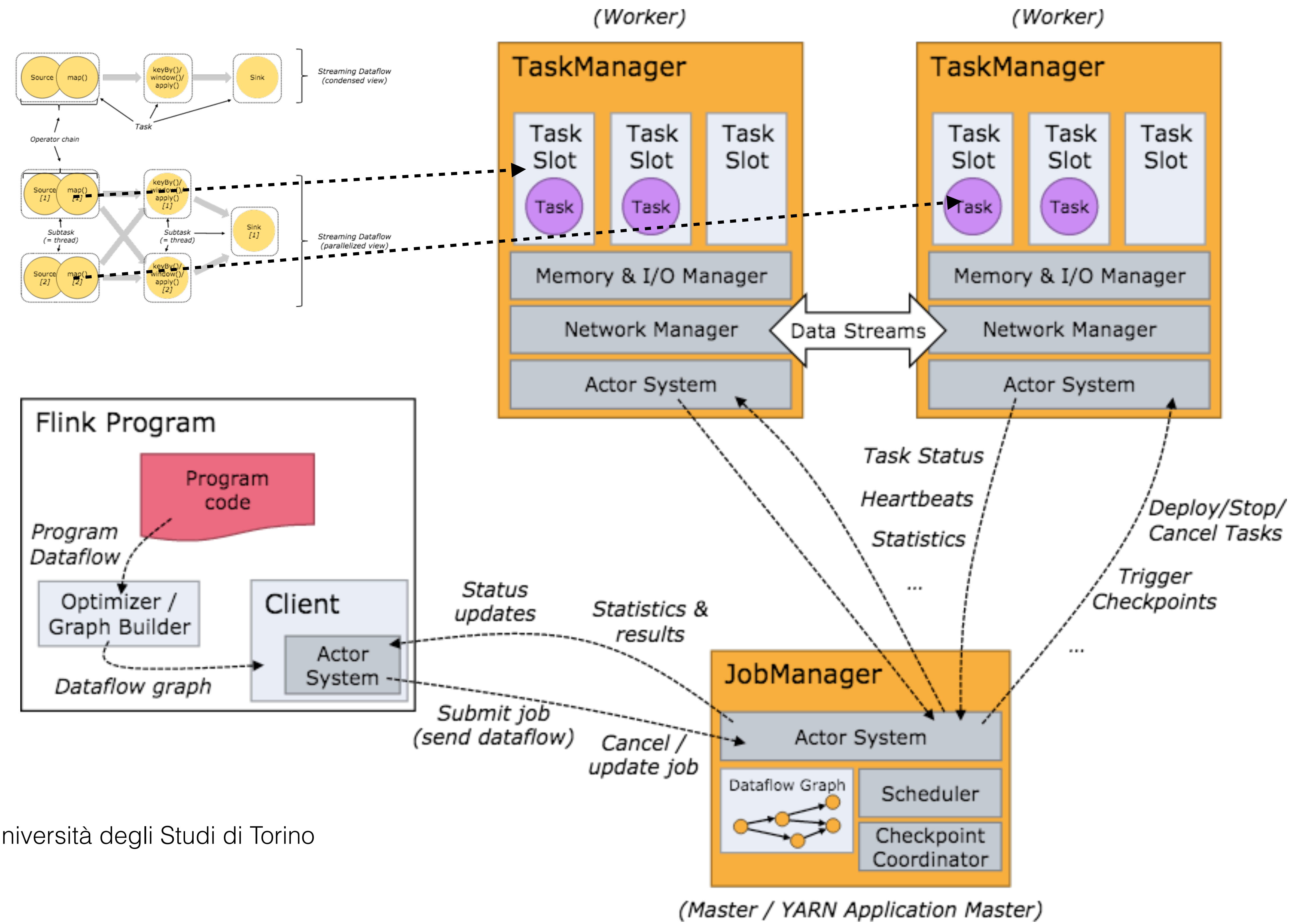
A DAG in Flink



Flink Execution Model

- Master-Worker paradigm
- The application is first transformed into a DAG (possibly optimised)
- DAG submitted to JobManager
    - Tasks divided among workers

Flink Execution Model



Apache Storm

Apache Storm

- Framework targeting only Stream processing
- Topological API
    1. Programs are expressed as graphs built by explicitly connecting processing nodes — Dataflow graphs
    2. Graph nodes are provided with the code defined by the user

Programming and Data Model

- Programming model based on three components
  - **Spout**: a source of a stream
  - **Bolt**: processing node - implements a function
  - **Topology**: composition of Spouts and Bolts
- Data model: **Tuple**
  - Lists of values of arbitrary type

Tuples Grouping

- **Grouping**: controller over tuples routing in the Topology

1. **Shuffle Grouping**: an equal number of tuples distributed across all of the workers

2. **Field Grouping**: tuples with the same field values sent to the same worker executing the bolts

3. **Global Grouping**: streams are grouped and forward to one bolt (n:1)

4. **All Grouping**: single copy of each tuple to all instances of the receiving bolt (i.e., broadcast)

Source Code Extract

```
public static class WordCount extends BaseBasicBolt {

   Map<String, Integer> counts = new HashMap<String, Integer>(); // stateful Bolt

   @Override

   public void execute(Tuple tuple, BasicOutputCollector collector) {

     String word = tuple.getString(0);

     Integer count = counts.get(word);

     if (count == null)

       count = 0;

     count++;

     counts.put(word, count);

     collector.emit(new Values(word, count));

   }

 }
```

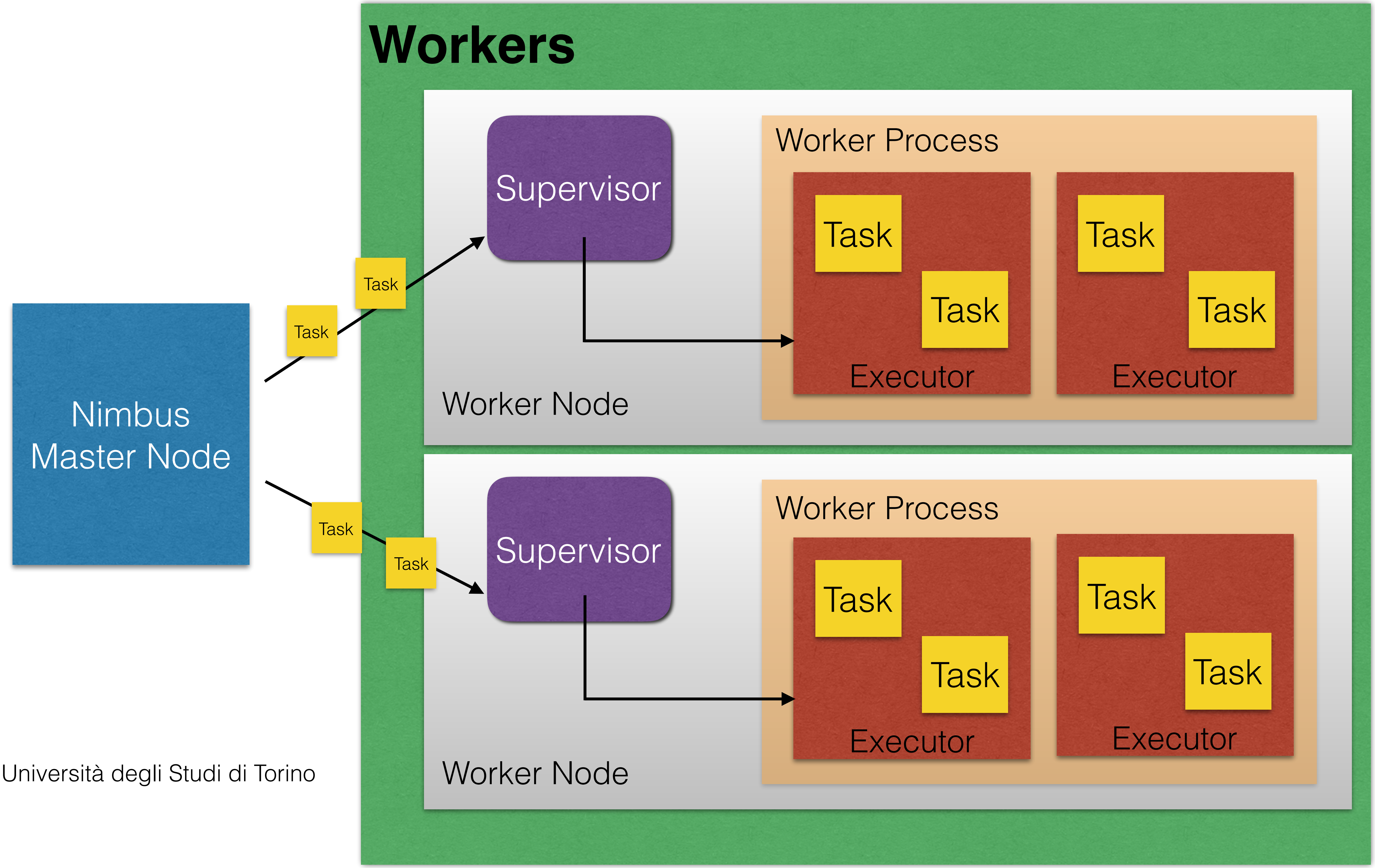
Storm Execution Model

- Master-Worker paradigm
- Workers takes Tasks in input
- **Task**: minimum logical unit of the topology
  - A Task is either the execution of a Spout or a Bolt
  - At a given time, each Spout and Bolt can have multiple instances running in multiple separate threads

Storm Execution Model

- Two types of nodes: **Nimbus** and **Supervisor**
    - **Nimbus** is the Master node
    - A **Supervisor** is a Worker Node with one or more Worker Processes
    - A **Worker Process** receives Tasks from Supervisor
    - An **Executor** is a single thread within a Worker Process
- Nimbus runs the Topology by distributing tasks to available Supervisors

Storm Execution Model



Summary

	hadoop MapReduce	Spark		
API	Declarative	Declarative	Declarative	Topological
Data Model	Key-Value Pairs	RDD DStream	DataSet DataStream	Tuple
Processing	Batch	Batch Stream	Batch Stream	Stream
Execution Model	Master Worker	Master Worker	Master Worker	Master Worker

HPC Oriented Model

PiCo — *Pi*peline *Co*mposition

- Strength:
    - Unique model for Batch and Stream processing
    - Clear denotational semantics and simple Declarative API
    - Written in C++ — homogeneous & heterogenous platforms
    - Less resources needed
- Weaknesses
    - Work in progress

PiCo Insights

- Specification and formalisation of the minimum kernel of operations needed to create a pipeline for data analytics
- Data model is also hidden to the programmer
    - model that is polymorphic w.r.t. data and processing model
- Re-use the same algorithms and pipelines on different data
- Exploits both pipelining and data parallelism

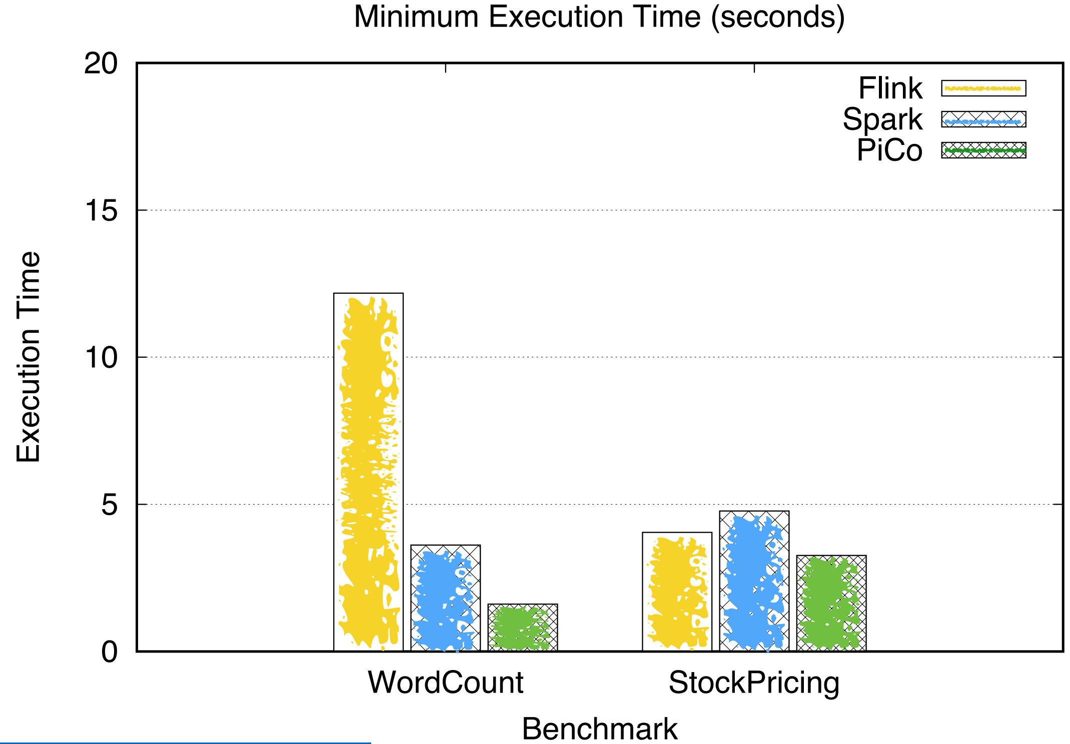
Word Count in PiCo

```
static auto tokenizer = [](std::string& in, FlatMapCollector<KV>& collector) {
    std::string::size_type i = 0, j;
    while((j = in.find_first_of(' ', i)) != std::string::npos) {
        collector.add(KV(in.substr(i, j - i), 1));
        i = j + 1;
    }
    if(i < in.size())
        collector.add(KV(in.substr(i, in.size() - i), 1));
};

Pipe countWords;
countWords
    .add(FlatMap<std::string, KV>(tokenizer)) //
    .add(PReduce<KV>([&](KV& v1, KV& v2) {return v1+v2;}));

ReadFromFile reader;
WriteToDisk<KV> writer([&](KV in) {return in.to_string();});

    /* compose the pipeline */
Pipe p2;
p2.add(reader).to(countWords).add(writer);

    /* execute the pipeline */
p2.run();
```

Preliminary Results

Scalability	Word Count	Stock Pricing
Flink	6.58	9.02
Spark	5.80	6.91
PiCo	13.60	6.69

Maximum Scalability



RAM	Word Count	Stock Pricing
Flink	3538,94 MB	3460,30 MB
Spark	1494,22 MB	1494,22 MB
Pico	157,29 MB	78,64 MB

Memory Footprint

Next (today) Lesson

- Focus on Spark:

    1. More on the programming model (batch and stream)

    2. More on the execution model

    3. Examples

Apache Spark Data Model

- Data model: **Resilient Distributed Dataset** (RDD)

  Immutable collection of objects partitioned across a cluster, that can be operated in parallel

- Two kind of operations: *Transformations* and *Actions*
  - resulting into a ***DAG*** of operations

Spark RDD

- **Read only** collection partitioned across a cluster
- Created by *Transformations* triggered by *Actions*
- RDDs are **in memory** unless cached/persisted
  - more like an expression than a value
- **Lazy** and **ephemeral**: computed and materialised only by actions
- **Resilient**: lost RDD are recomputed automatically

Spark RDD

1. RDD is **created** originally from **external data sources** (e.g. HDFS, Local file ... etc)

2. RDD undergoes a sequence of **Transformations** (e.g. map, flatMap, filter, groupBy, join), each provide a different RDD that feed into the next transformation

3. Finally the last step is an **Action** (e.g. count, collect, save, take), which convert the last RDD into an output to external data sources

Transformations and Actions

Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V,V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T],RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)],RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)],RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T],RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T,T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

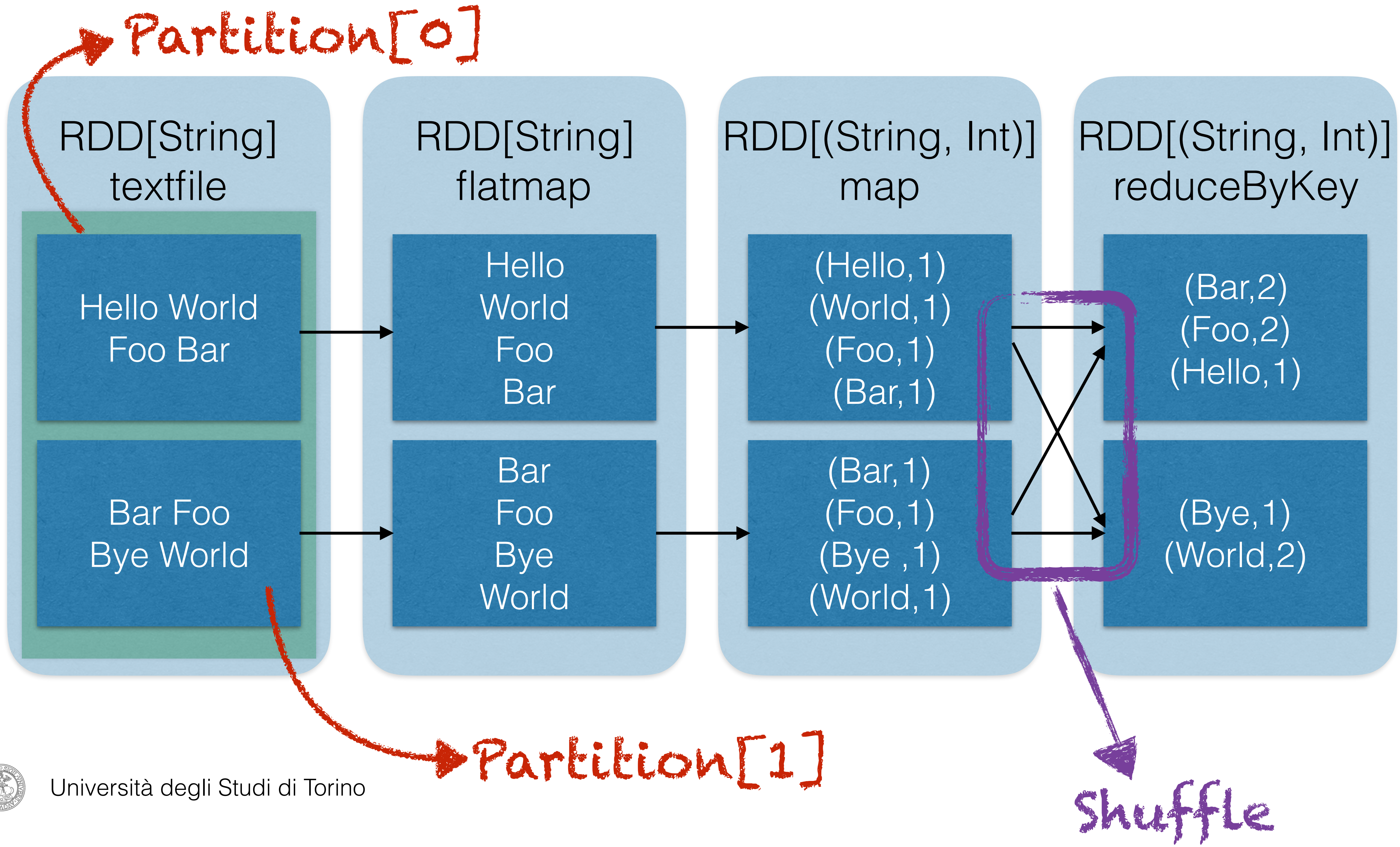
Batch Programming

- Computations on **finite data** (offline computing)
- RDDs are created by
    1. **parallelising** an existing collection
    2. **referencing** a dataset in an external storage system (HDFS)
- Transformations on RDDs create new RDDs
- Actions materialise RDDs or results

Word Count

```
SparkConf conf = new
SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);

JavaRDD<String> lines = sc.textFile("data.txt");

JavaPairRDD<String, Integer> pairs =
         lines.mapToPair(s -> new Tuple2(s, 1));

JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) ->
a + b);

List<Tuple2<String, Integer>> output = counts.collect();
    for (Tuple2<?,?> tuple : output) {
      System.out.println(tuple._1() + ": " + tuple._2());
    }
```

Word Count


Partition[0]
RDD[String]
textfile
Hello World
Foo Bar
Bar Foo
Bye World
RDD[String]
flatmap
Hello
World
Foo
Bar
Bar
Foo
Bye
World
RDD[(String, Int)]
map
(Hello,1)
(World,1)
(Foo,1)
(Bar,1)
(Bar,1)
(Foo,1)
(Bye ,1)
(World,1)
RDD[(String, Int)]
reduceByKey
(Bar,2)
(Foo,2)
(Hello,1)
(Bye,1)
(World,2)
Partition[1]
Shuffle

Shuffle Operations

- Data redistribution so that data are grouped differently across partitions
- Typically involves **copying data** among executors
- Expensive operation:
  - disk I/O
  - serialisation
  - network I/O

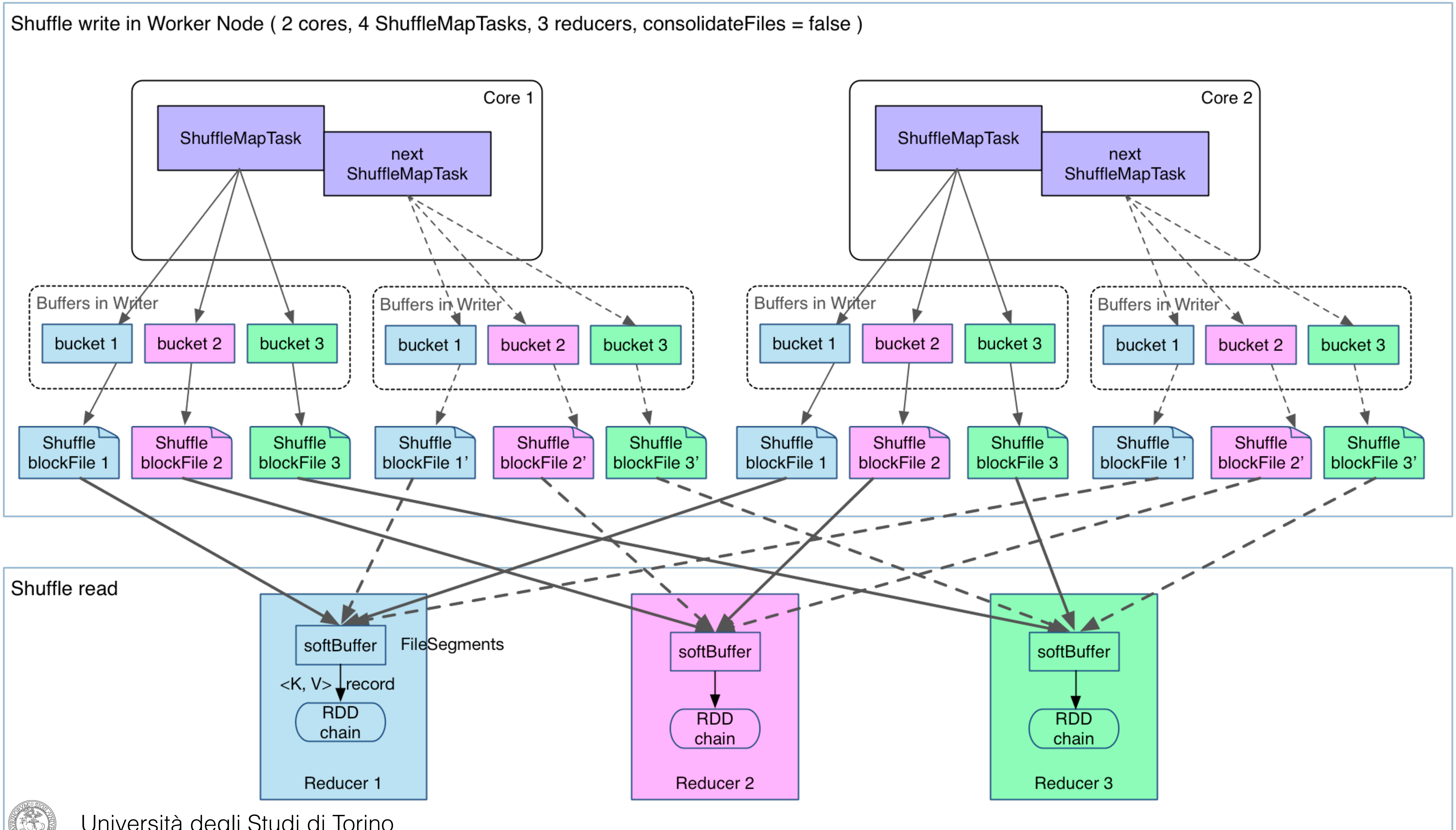
Shuffle Operations

- Map task and Reduce task - from MapReduce
- Map task:
    - results from map are in memory until they can fit
    - results are sorted by key (stable sorting per partition) and written to disk

Shuffle Operations

- Map task and Reduce task - from MapReduce
- Reduce task:
  - read file for specific partitions
  - perform the reduce operator to data

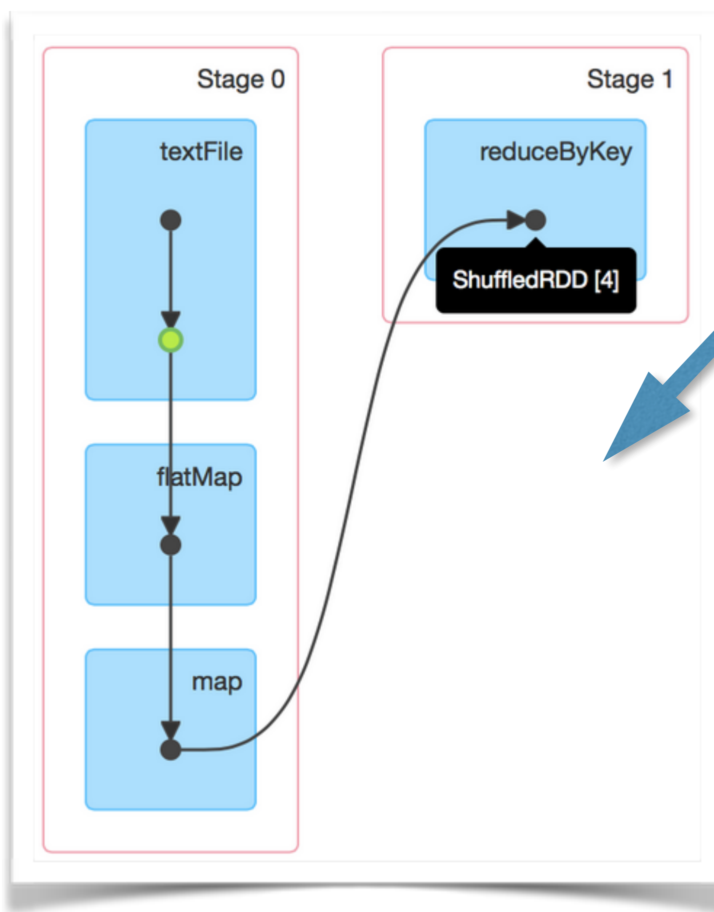
Shuffle Operations

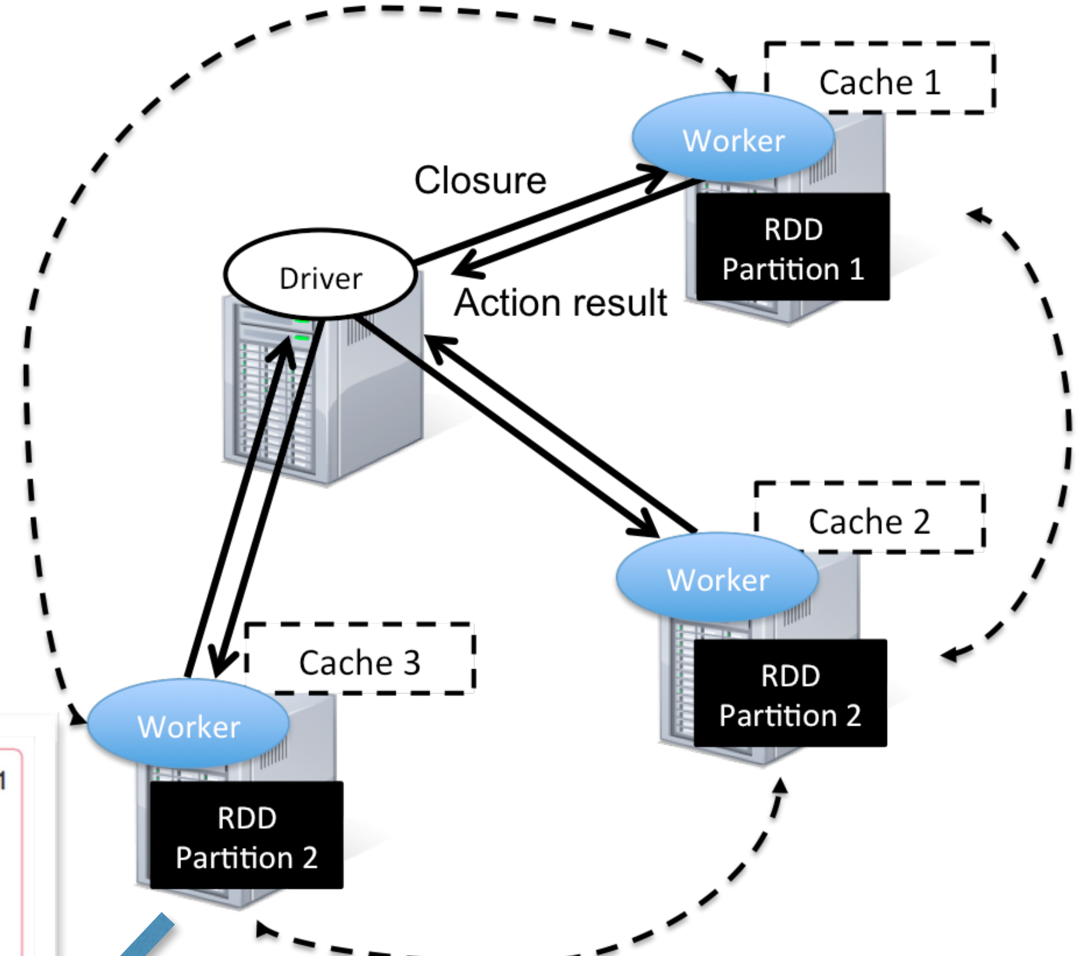


Execution Model

- Master-Worker paradigm

1. Driver program
2. Worker DAG





Data shuffling across machines
(wide dependencies)

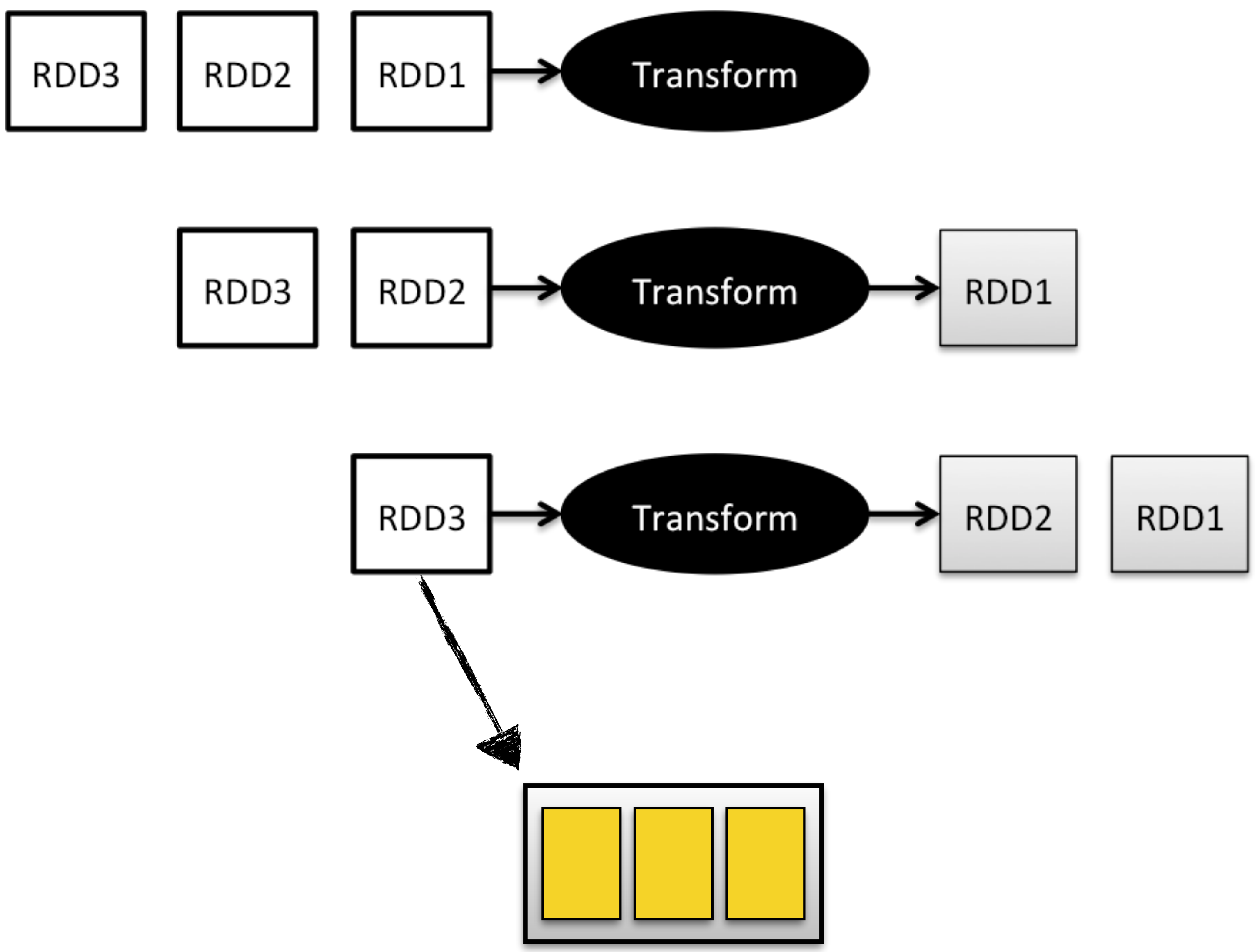
Spark Streaming

- Real-time processing of streams of data
- Data model: Discretised Stream **DStream** which is basically a sequence of RDD (**micro-batch**)
  - Each RDD contains data associated with a time interval

Discretised Stream

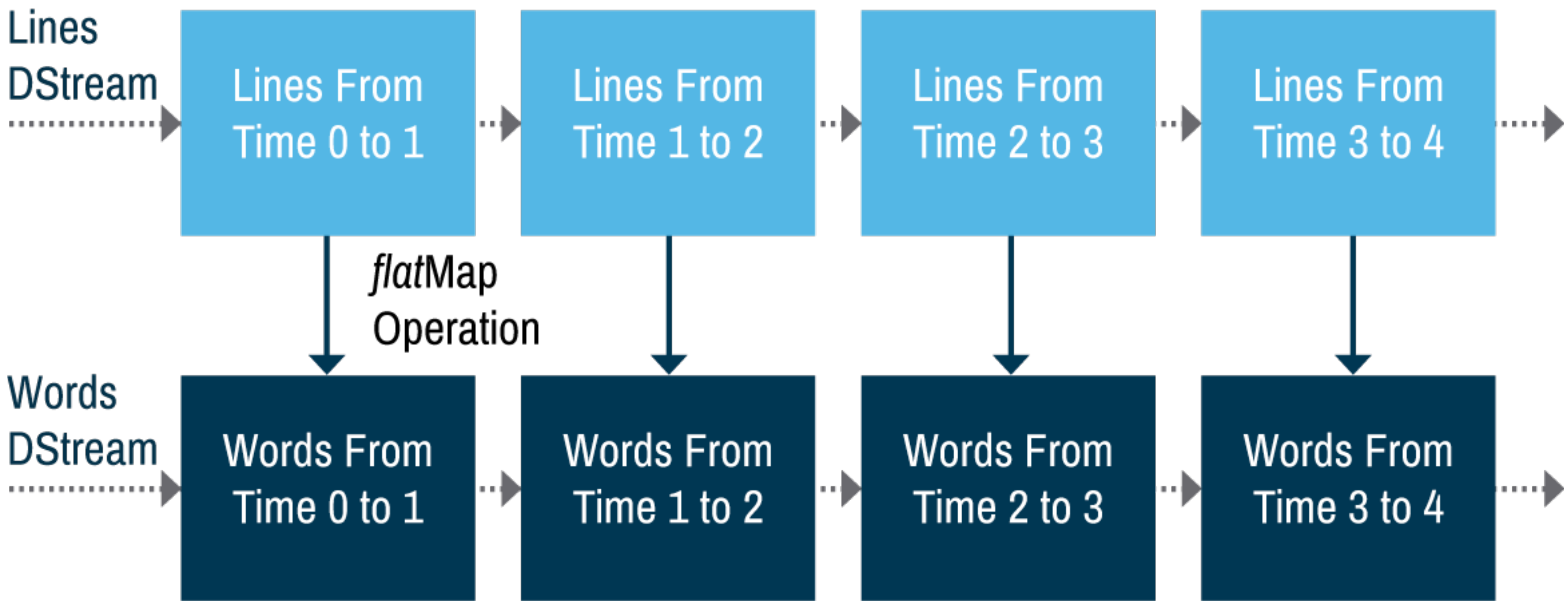
- Transformations on DStreams are forwarded to each micro-batch
  - Each transformation produces an output RDD
  - Resulting DStream is another sequence of RDDs that defines an output DStream

DStream Transformations



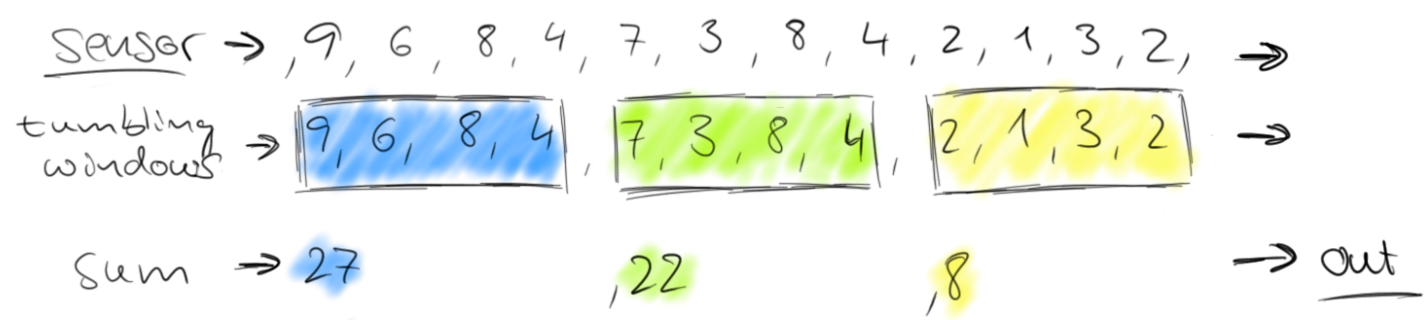
- Transform (RDD -> RDD)
  - Map
  - FlatMap
  - Filter
  - Count
  - CountByValue
  - GroupByKey
  - Reduce
  - ReduceByKey
  - Join
  - Cogroup
  - Transform
  - UpdateStateByKey

FlatMap to a DStream

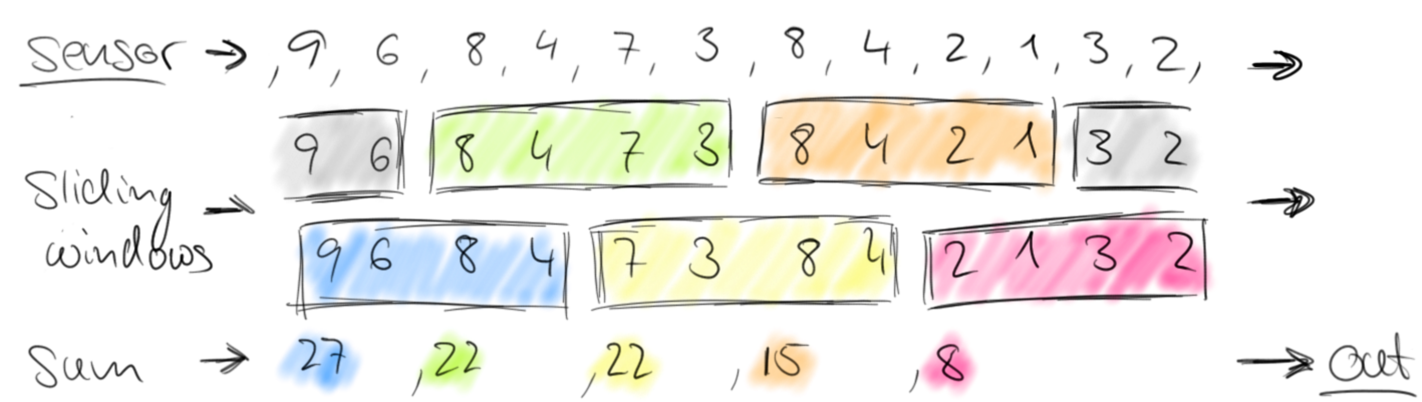


Partition the Stream with Windowing

- Tumbling Windows
  - Fixed size and consecutive



- Sliding Windows
  - Fixed size and overlapping

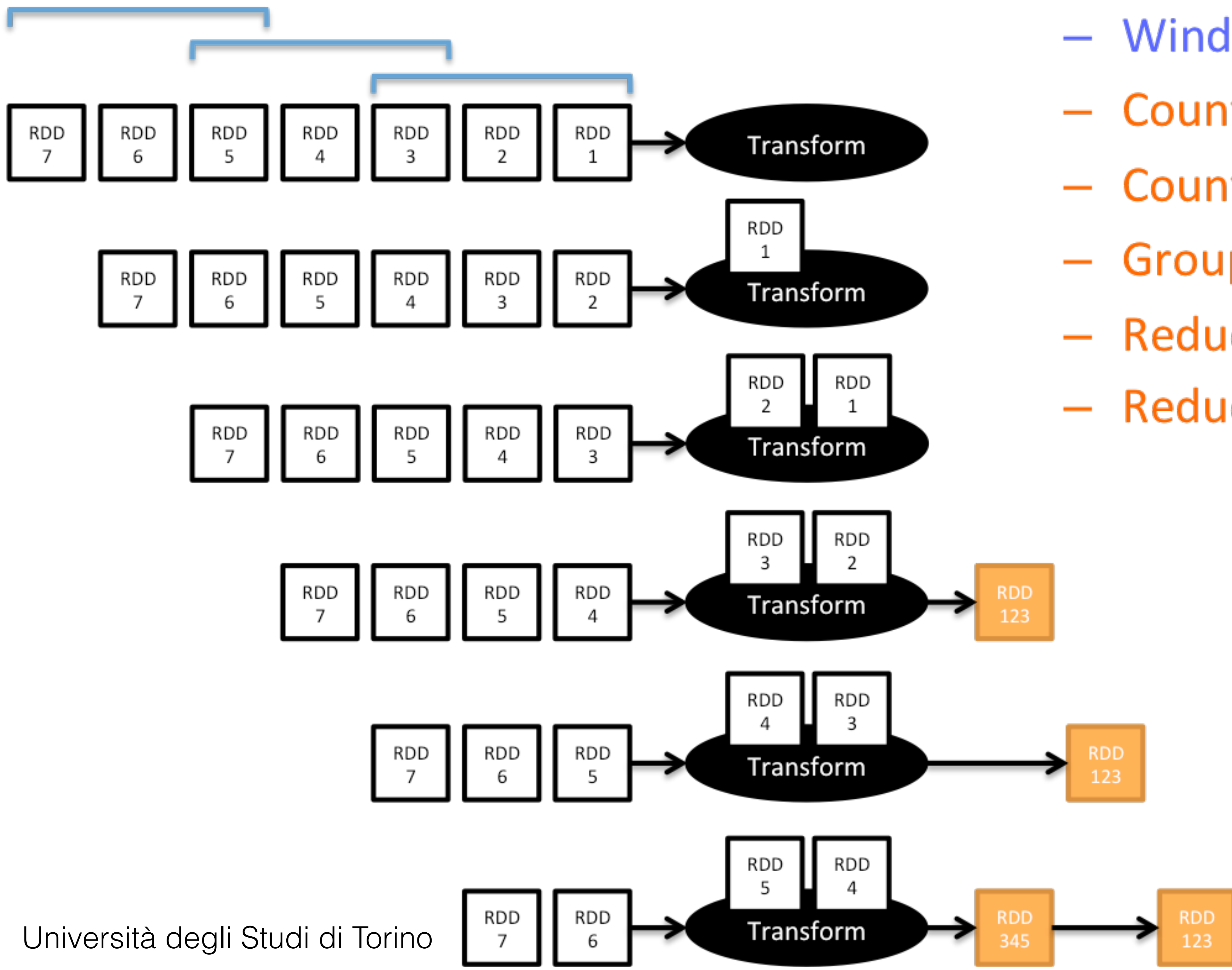


Windowing in Spark

- **Window length**: how many *consecutive* RDDs will be combined for performing the transformation
- **Slide interval**: how many RDD will be *skipped* before the next transformation executes

Windowing a DStream

windowDuration = 3, slideDuration = 2



- Window transformation
  - Window
  - CountByWindow
  - CountByValueAndWindow
  - GroupByKeyAndWindow
  - ReduceByWindow
  - ReduceByKeyAndWindow

Spark Streaming Example

```
// Create a DStream that will connect to hostname:port, like localhost:9999

JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost",
9999);

// Split each line into words

JavaDStream<String> words = lines.flatMap(

  new FlatMapFunction<String, String>() {

    @Override public Iterator<String> call(String x) {

      return Arrays.asList(x.split(" ")).iterator();

    }

  });
```

Spark Streaming Example

```
// Count each word in each batch
JavaPairDStream<String, Integer> pairs = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    @Override public Tuple2<String, Integer> call(String s) {
      return new Tuple2<>(s, 1);
    }});
JavaPairDStream<String, Integer> wordCounts = pairs.reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    @Override public Integer call(Integer i1, Integer i2) {
      return i1 + i2;
    }
  });
// Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.print();
```

Bound Operators on Unbound Data

- countByWindow
- reduceByWindow
- reduceByKeyAndWindow
- countByValueAndWindow
- Join: RDD generated by stream1 will be joined with the RDD generated by stream2