732A54 / TDDE31
Big Data Analytics

LINKÖPING UNIVERSITY

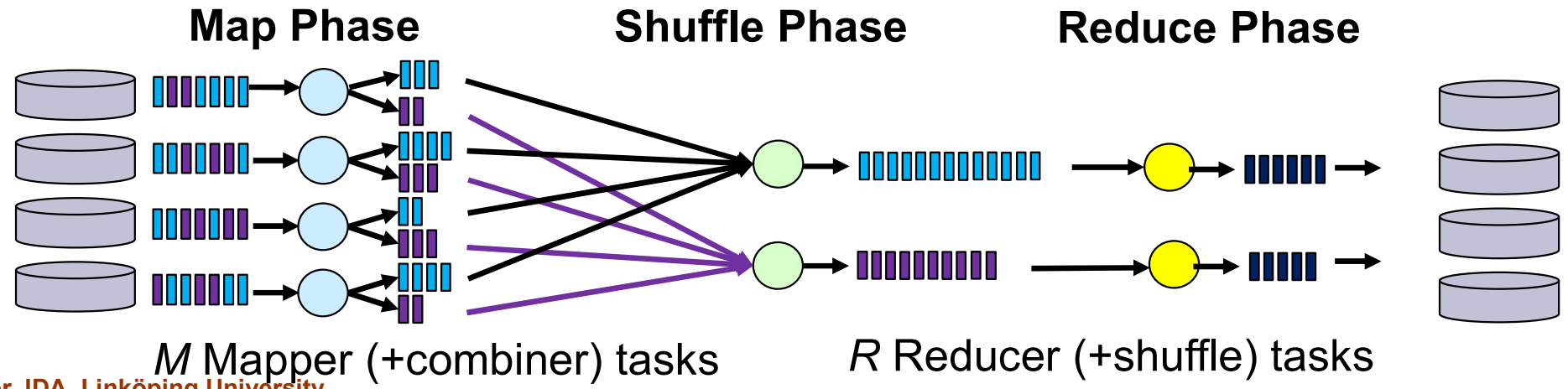
# Introduction to Spark

**Christoph Kessler**

IDA, Linköping University

# Recall: MapReduce Programming Model

- Designed to operate on LARGE distributed input data sets stored e.g. in HDFS nodes
- Abstracts from parallelism, data distribution, load balancing, data transfer, fault tolerance
- Implemented in **Hadoop** and other frameworks
- Provides a high-level parallel programming construct (= a skeleton) called **MapReduce**
  - A generalization of the data-parallel *MapReduce* skeleton of Lecture 1
  - Covers the following algorithmic design pattern:



# From MapReduce to Spark

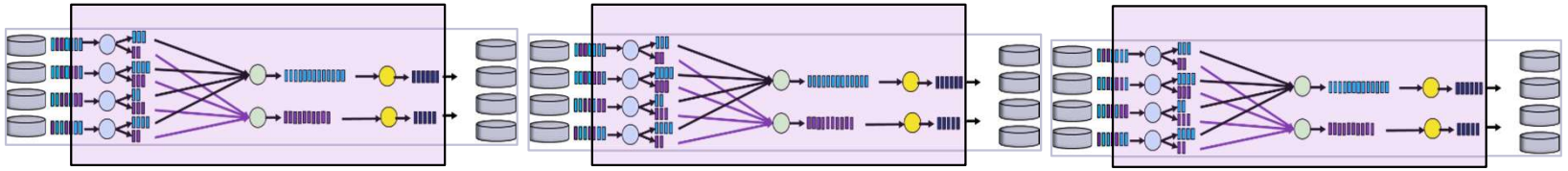
**MapReduce**

- is for large-scale computations matching the *MapReduce* pattern,
- with input, intermediate and output data stored in secondary storage

**Limitations**

By chaining multiple MapReduce steps, we can emulate *any* distributed computation.

- For complex computations composed of *multiple* MapReduce steps
  - E.g. iterative computations
    - e.g. parameter optimization by gradient search



…

→ Much unnecessary disk I/O – data for next MapReduce step could remain in main memory or even cache memory

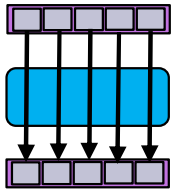
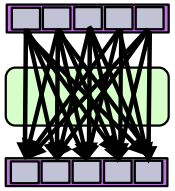
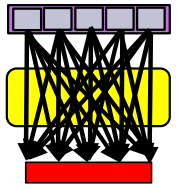
→ Data blocks used multiple times are read multiple times from disk

→ Bad data locality across subsequent Mapreduce phases

- Sharing of data only in secondary storage
  - Latency can be too long for interactive analytics
- Fault tolerance by replication of data – more I/O to store copies → slow

# Splitting the MapReduce Construct into Simpler Operations – 2 Main Categories:

- **Transformations**: Elementwise operations, fully parallelizable
  - Working on distributed data. Mostly variants of **Map**
- **Actions**: Operations with internally global dependence structure
  - Mostly variants of **Reduce** and writing back to non-distr. file / to master

| | | | |
|---|---|---|---|
| **Transformations**<br><br>Both input and output data operands are distributed | *map*(*f* : T ⇒ U)<br>*filter*(*f* : T ⇒ Bool)<br>*flatMap*(*f* : T ⇒ Seq[U])<br>*sample*(*fraction* : Float)<br>*groupByKey*()<br>*reduceByKey*(*f* : (V, V) ⇒ V)<br>*union*()<br>*join*()<br>*cogroup*()<br>*crossProduct*()<br>*mapValues*(*f* : V ⇒ W)<br>*sort*(*c* : Comparator[K])<br>*partitionBy*(*p* : Partitioner[K]) | Local dep. |  |
| | | | Element-wise dependences only, e.g. map, filter, flatMap |
| | | Global dep. |  |
| | | | Involves some shuffle and sorting across blocks, but still produces distributed output ("RDD") |
| **Actions**<br><br>Output data is not distributed | *count*()<br>*collect*()<br>*reduce*(*f* : (T, T) ⇒ T)<br>*lookup*(*k* : K)<br>*save*(*path* : String) | Global dep. |  |

**RDD transformations and actions available in Spark**. Seq[T] denotes a sequence of elements of type T.

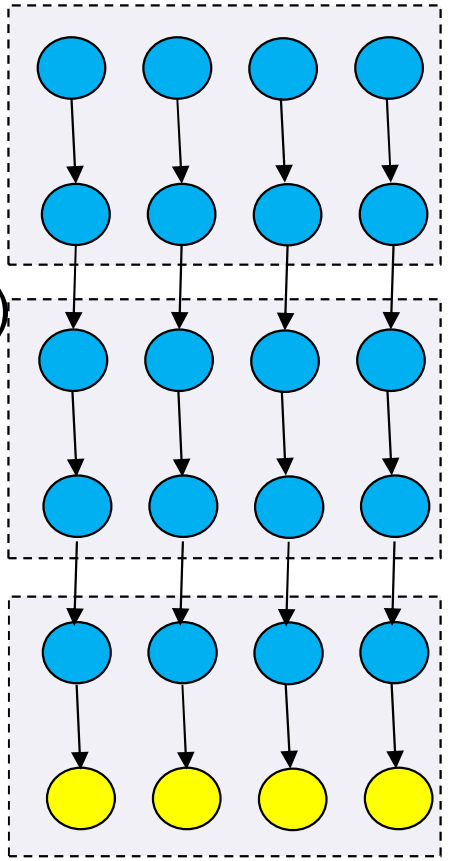
Table source: Zaharia *et al.*: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.

# Remark on data types

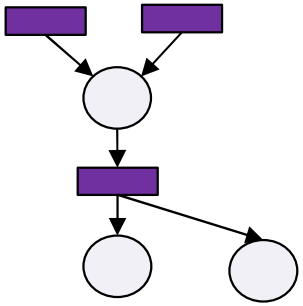
- Most transformations and actions can work on arbitrary element data types (i.e., not only on key-value pairs).
- Some *transformations* work only on **key-value pairs**, namely **groupByKey**(), **reduceByKey**(), **combineByKey**(), **aggregateByKey**().
  - These are transformations (return a RDD, are evaluated lazily) but include a shuffle-and-sort-by-key phase (as in MapReduce) → a non-local dependence pattern
  - The implementation uses internally a combiner.
- Also some *actions* work only on key-value pairs, e.g. **countByKey**

# Spark Idea: Data Flow Computing in Memory

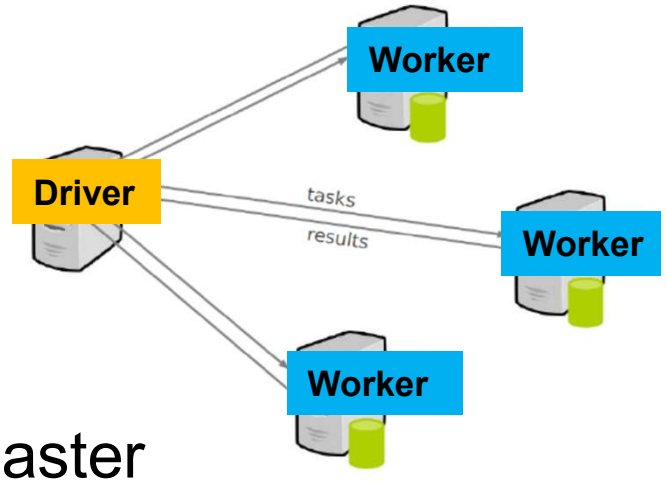
Instead of calling subsequent rigid MapReduce steps, the Spark programmer describes the overall **data flow graph** of how to compute all intermediate and final results from the initial input data



- *Lazy evaluation of transformations*
  - Transformations are just added to the graph (postponed)
  - Actions "push the button" for computing (= *materializing* the results) according to the data flow graph
- Gives more flexibility to the scheduler
  - Better data locality (esp. with local dependence patterns)
  - Keep data in memory as capacity permits, can skip unnecessary disk storage of temporary data
- No replication of data blocks for fault tolerance – in case of task failure (worker failure), **recompute** it from available, earlier computed data blocks according to the data flow graph
  - Needs a data structure for operand data that "knows" how its data blocks are to be computed: the **RDD**

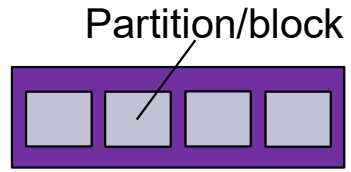


# Spark Execution Model

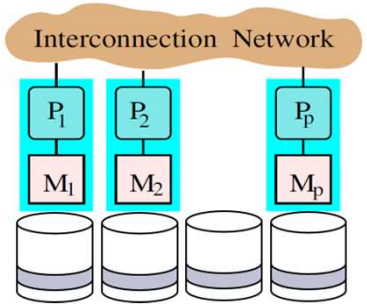



- **Driver program** (sequential) runs on host / master
- Operations on distributed data (RDDs) run on **workers**
- Collect data from workers to driver program on demand

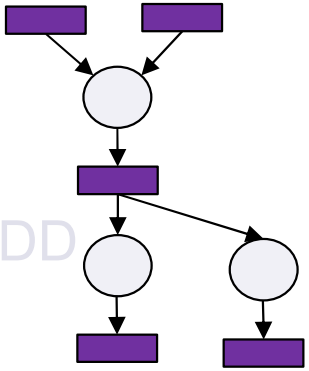
# Resilient Distributed Datasets (RDDs)



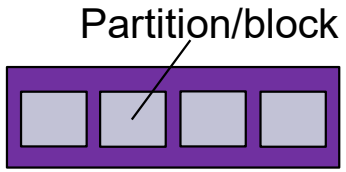

- **Containers for operand data** passed between parallel operations
  - *Read-only* (after construction) collection of data objects
  - Partitioned and distributed across workers (cluster nodes)
  - Materialized on demand from construction description
  - Can be rebuilt if a partition (data block) is lost
  - By default, cached **in main memory** –
    not persistent (in secondary storage) until written back



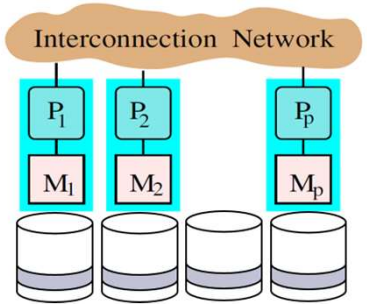

- **Construction of new RDDs:**
  - By reading in from a file e.g. in HDFS
  - By partitioning and distributing a non-distributed collection (e.g., array) previously residing on master node ("*scatter*")
  - By a *Map* operation: $A \rightarrow \text{List}(B)$
    (elementwise transformation, filtering, …) applied on another RDD
  - **Changing persistence state of a RDD:**
    - By a **caching** hint for data to be reused – if enough space in memory
    - By **materializing** (persisting, saving) to a file
      (and discarding its copy in memory)



# Resilient Distributed Datasets (RDDs)

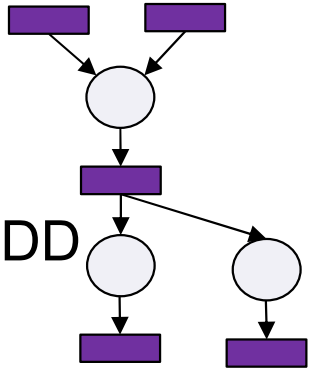



- **Containers for operand data** passed between parallel operations
  - *Read-only* (after construction) collection of data objects
  - Partitioned and distributed across workers (cluster nodes)
  - Materialized on demand from construction description
  - Can be rebuilt if a partition (data block) is lost
  - By default, cached **in main memory** –
    not persistent (in secondary storage) until written back

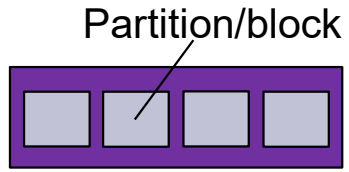



- **Construction of new RDDs:**
  - By reading in from a file e.g. in HDFS
  - By partitioning and distributing a non-distributed collection (e.g., array) previously residing on master node ("***scatter***")
  - By a *Map* operation: *A* → List(*B*)
    (elementwise transformation, filtering, …) applied on another RDD
  - **Changing persistence state of a RDD:**
    - By a **caching** hint for data to be reused – if enough space in memory
    - By **materializing** (persisting, saving) to a file
      (and discarding its copy in memory)

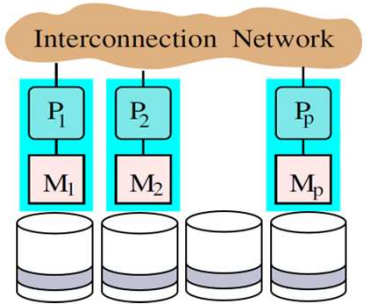
```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```



# Resilient Distributed Datasets (RDDs)

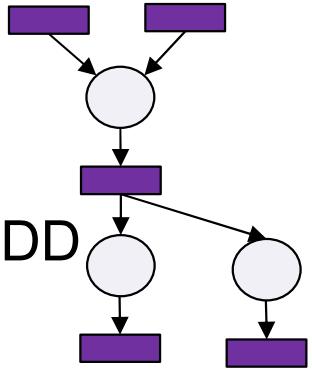



- **Containers for operand data** passed between parallel operations
  - *Read-only* (after construction) collection of data objects
  - Partitioned and distributed across workers (cluster nodes)
  - Materialized on demand from construction description
  - Can be rebuilt if a partition (data block) is lost
  - By default, cached **in main memory** – not persistent (in secondary storage) until written back




- **Construction of new RDDs:**
  - By reading in from a file e.g. in HDFS
  - By partitioning and distributing a non-distributed collection (e.g., array) previously residing on master node ("***scatter***")
  - By a *Map* operation: *A* → List(*B*) (elementwise transformation, filtering, …) applied on another RDD
  - **Changing persistence state of a RDD:**
    - By a **caching** hint for data to be reused – if enough space in memory
    - By **materializing** (persisting, saving) to a file (and discarding its copy in memory)

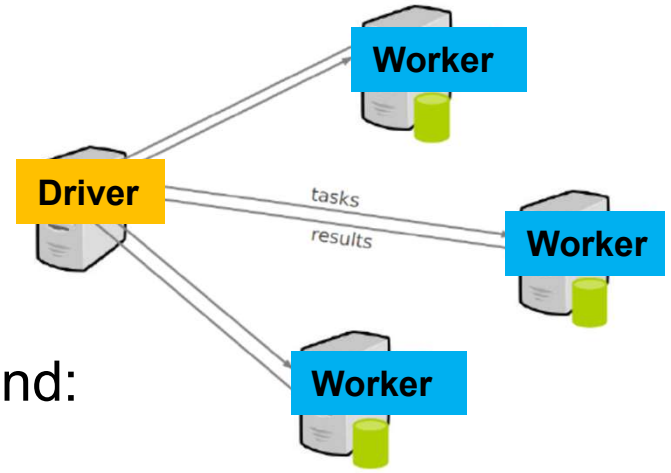
```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```



```
cachedData = distdata.cache()
```

```
distdata.saveAsTextFile(...)
```

# Actions on RDDs




**Recall: Spark execution model:**

- **Driver program** (sequential) runs on host / master
- Operations on RDDs run on **workers**
- Collect data from workers to driver program on demand:

**Parallel Collect Operations on RDDs:**

- **Reduce**
  - Combine RDD elements using an associative binary function to produce a (scalar) result at the driver program.
  - Key-value pairs to reduce over are grouped by key, as in MapReduce
- **Collect**
  - Send all elements of the RDD to the driver program (”*gather*”)
    - The reverse operation of **parallelize**
- **Foreach**
  - Pass each RDD element through a user-provided function
  - *Eager* evaluation - *Not* producing another RDD (difference from Map/Filter)
  - Might be used e.g. for copying data to another system

# Classification of RDD Operations

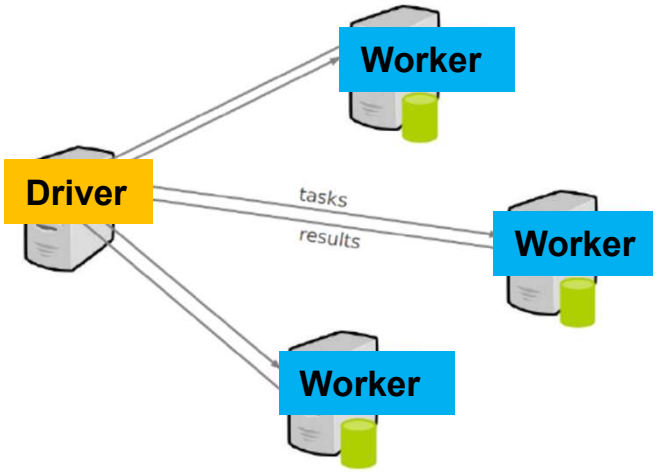
- **Transformations**: Lazy, parallelizable
  - Working on distributed data. Mostly variants of **Map**
- **Actions**: Materialization points ("push the button")
  - Mostly variants of **Reduce** and writing back to non-distr. file / master

| | | | |
|---|---|---|---|
| **Transformations**<br><br>Both input and output data operands are distributed | *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**<br><br>Output data is not distributed | *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 |

**RDD transformations and actions available in Spark**. Seq[T] denotes a sequence of elements of type T.

Table source: Zaharia *et al.*: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.

# Shared Variables




- shared = not partitioned and distributed, accessible by all workers

- **Broadcast Variables**
  - Replicated shared variables – 1 copy on each worker
  - Read-only for workers
  - For global data needed by all workers, e.g. filtering parameters, lookup table

- **Accumulator Variables**
  - Residing on driver program process
  - Workers can not read, only add their contributions using an associative operation
  - Good for implementing counters and for global sum

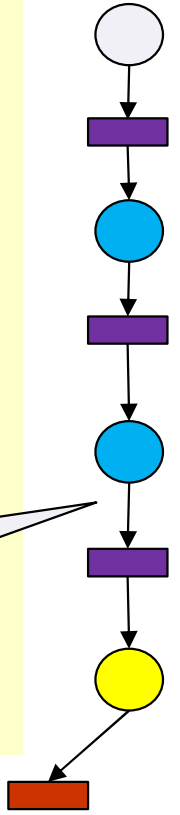
# Example: Text Search

- Count lines containing ”ERROR” in a large log file in HDFS

Python code adapted from Zaharia *et al*. 2010

```
// Create a RDD from file:
file = sc.textFile("hdfs://...")

// Filter operation to create RDD containing lines with ”ERROR”:
errs = file.filter( lambda line: line.find("ERROR”)>=0 )

// Map each line to a 1:
ones = errs.map( lambda word: (word, 1) )

// Add up the 1’s using Reduce:
count = ones.reduce( lambda x, y: x+y )
```

The ”**lineage**” (DFG) of RDDs leading to the result *count*



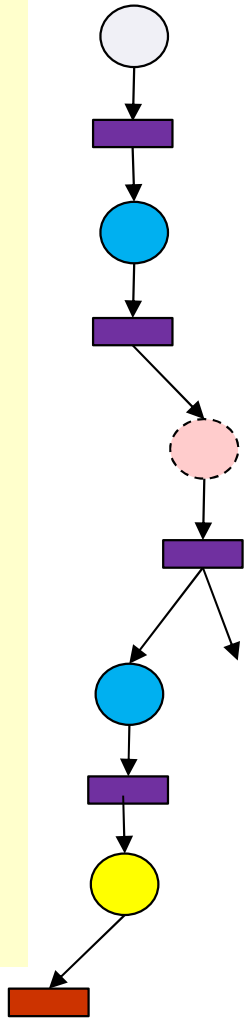
- RDDs *errs* and *ones* are lazy RDDs that are never materialized to secondary storage.
- Call to **reduce** (action) triggers computation of *ones*, which triggers computation of *errs*, which triggers reading blocks from the file.

# Example: Text Search, with reuse of *errs*

- Count lines containing ”ERROR” in a large log file in HDFS

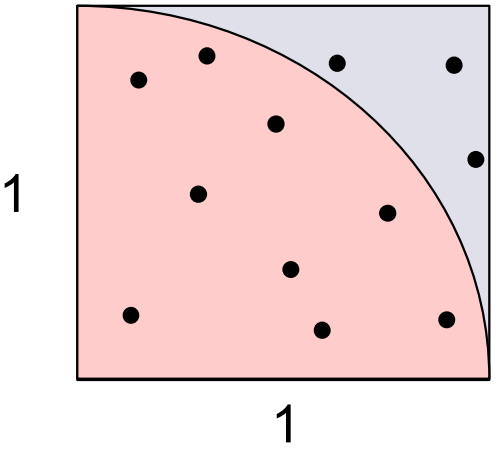
Python pseudocode

```
// Create a RDD from file:
file = sc.textFile("hdfs://...")

// Filter operation to create RDD containing lines with ”ERROR”:
errs = file.filter( lambda line: line.find("ERROR”)>=0 )

// Cache hint that errs will be reused in another operation:
cachedErrs = errs.cache();

// Map each line to a 1:
ones = cachedErrs.map( lambda word: (word, 1) )

// Add up the 1’s using Reduce:
count = ones.reduce( lambda x, y: x+y )
```



# Example: Pi Calculation

- Stochastic approximation of Pi:
  - A random point (x,y) in [0,1]x[0,1] is located within quarter unit cycle iff $x^2 + y^2 < 1$




Argument not used (index)

```
def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

count = sc.parallelize( xrange(0, NUM_SAMPLES) ) \
            .map(sample) \
            .reduce( lambda a, b: a + b)

print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Create a RDD containing all indexes 0, …, NUM_SAMPLES-1

RDD variables are implicit (operation return values)

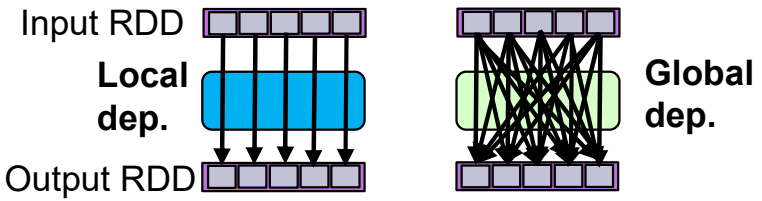
# Example: Logistic Regression

- Iterative classification algorithm to find a hyperplane that best separates 2 sets of data points
- Gradient descent method:
  - Start at a random normal-vector (hyperplane) *w*
  - In each iteration, add to *w* an error-correction term (based on the *gradient*) that is a function of *w* and the data points, to improve *w*

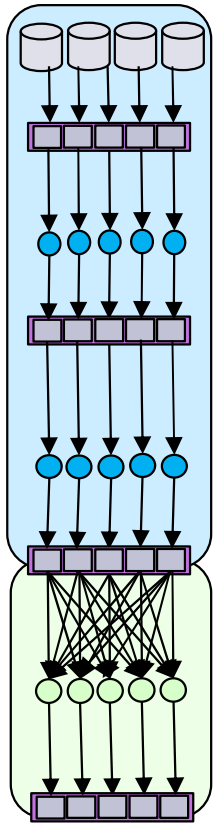
Scala pseudocode, adapted from Zaharia et al., 2010

```
// Read points from a text file and cache them:
points = sc.textFile(...).map(parsePoint).cache()
// Initialize w to random D-dimensional vector:
w = Vector.random(D)
// Run multiple iterations to update w:
for (i <- 1 to NUMBER_OF_ITERATIONS) {
    grad = sc.accumulator( new Vector(D) )
    for (p <- points) {  // Runs in parallel:
        val s = (1/(1+exp(-p.y*(w dot p.x)))-1) * p.y
        grad += s * p.x   // remotely add contribution to gradient value
    }
    w -= grad.value      // correction of w
}
```

# Spark Execution Model




- Depending on the kind of operations, the data dependences between RDDs in the lineage graph can be **local** (elementwise) or **global** (shuffle-like)
- When a user (program) runs an *action* on an RDD, the Spark scheduler builds a DAG (directed acyclic graph) of *stages* from the RDD lineage graph (data flow graph, task graph).
- A **stage** contains a contiguous subDAG of as many as possible operations with *local* (element-wise) dependencies between RDDs
  - The boundary of a stage is thus defined by
    - Operations with global dependencies
    - Already computed (materialized) RDD partitions.
- Execution of the operations within a stage is **pipelined**
  - intermediate results forwarded in memory
- The scheduler launches **tasks** to workers (cluster nodes) to compute missing partitions from each stage until it computes the target RDD.
- Tasks are assigned to nodes based on data locality.
  - If a task needs a partition that is available in the memory of a node, the task is sent to that node.



# Spark Performance

Results from original paper on Spark 2010:

- Spark can outperform Hadoop by 10x in iterative machine learning jobs
- Interactive query of a 39GB data set in < 1s

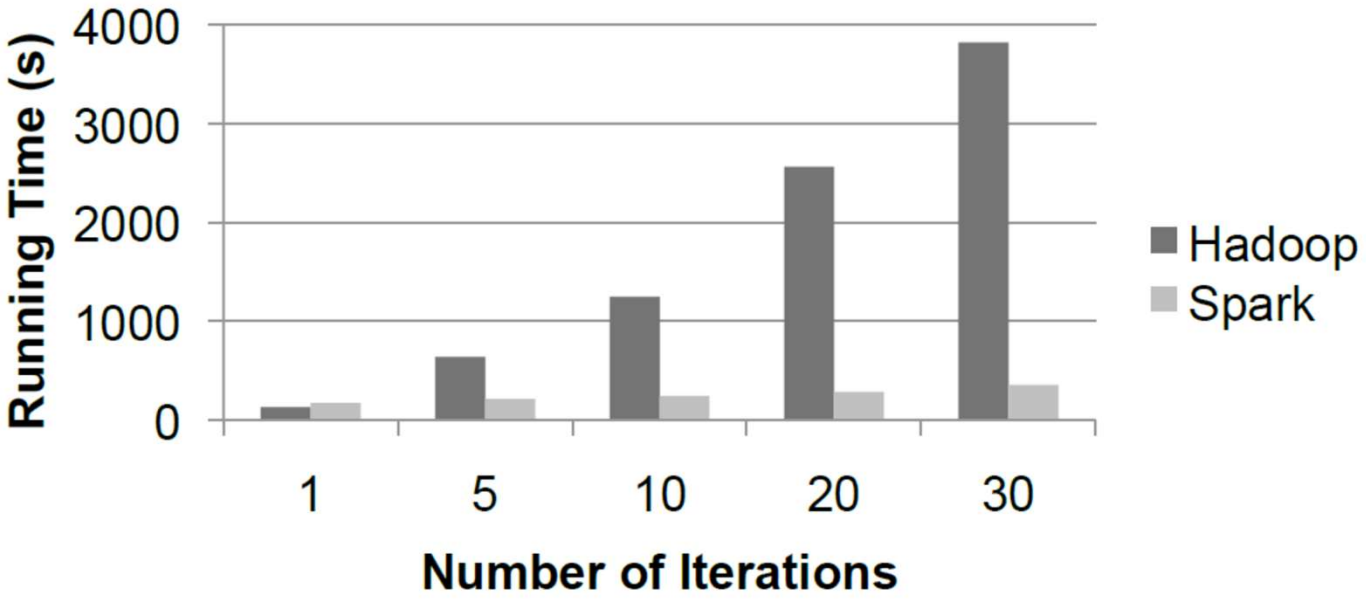



Figure 2: Logistic regression performance in Hadoop and Spark.

Image source:
M. Zaharia *et al.*,
2010. © ACM

# Using Spark

- Spark can run atop HDFS, but other implementations also exist
- Language bindings exist for Scala, Java, Python (PySpark)
  - Some minor restrictions for Python
- Spark Context object
  - The main entry point to Spark functionality
  - Represents connection to a Spark cluster
  - PySpark context *sc* is up and running from start
  - Create your own Spark context object for stand-alone applications
    - sc = **new** pyspark.SparkContext( *master*, *applName*, [*sparkHome*], […] )

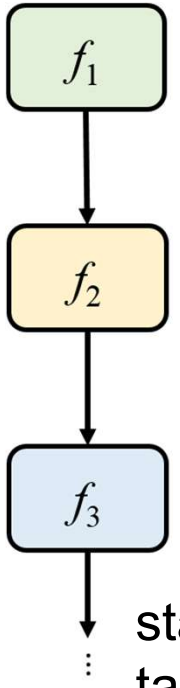
local
local[*k*]
spark://host:port
mesos://host:port

# Spark Streaming

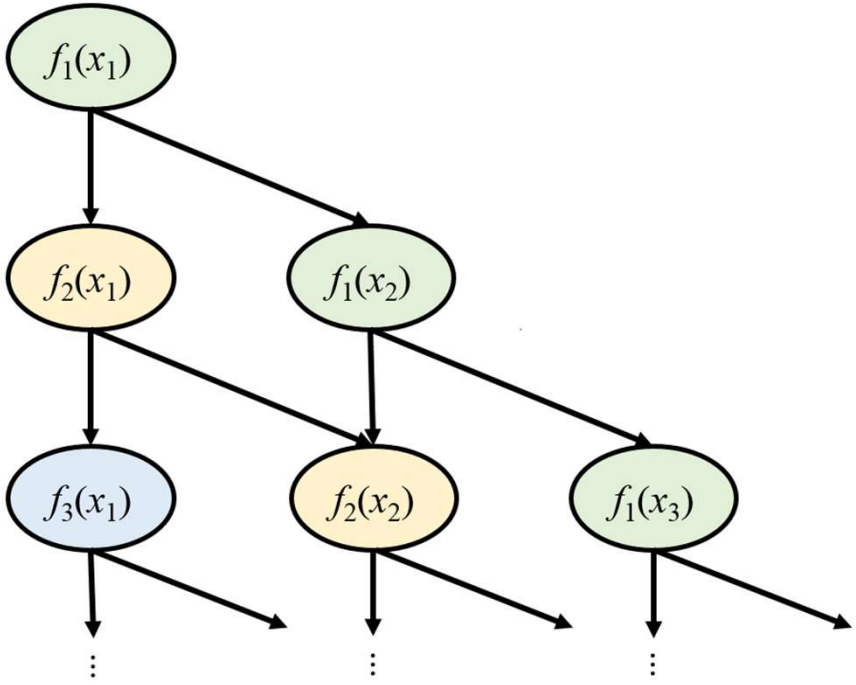
# Pipelining (Pattern)

applies a sequence of dependent computations/tasks ($f_1$, $f_2$, ..., $f_k$) elementwise to data sequence $\boldsymbol{x} = (x_1, x_2, x_3, \ldots, x_n)$

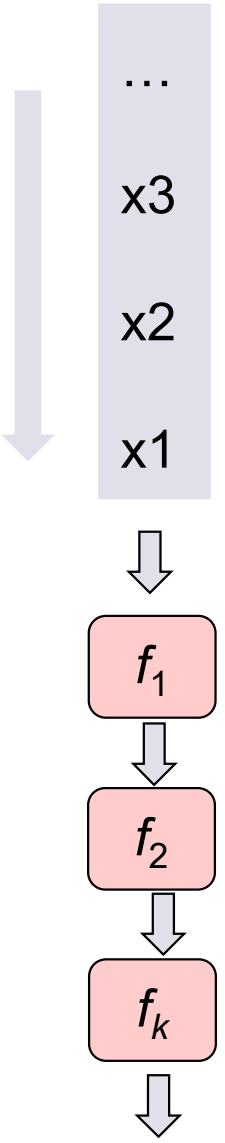
- For fixed $x_j$, must compute $f_i(x_j)$ before $f_{i+1}(x_j)$
- … and $f_i(x_j)$ before $f_i(x_{j+1})$ if the tasks $f_i$ have a *run-time state*




stage task dependence graph



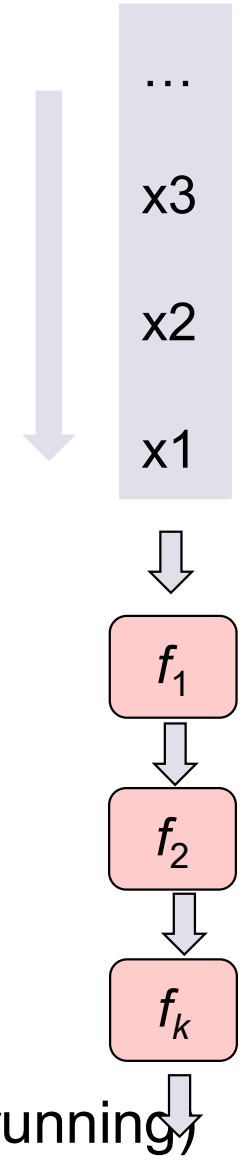

pipeline task instance dependence graph



# Pipelining (Pattern)

applies a sequence of dependent computations/tasks ($f_1$, $f_2$, ..., $f_k$) elementwise to data sequence $\boldsymbol{x} = (x_1, x_2, x_3, \ldots, x_n)$

- For fixed $x_j$, must compute $f_i(x_j)$ before $f_{i+1}(x_j)$
- … and $f_i(x_j)$ before $f_i(x_{j+1})$ if the tasks $f_i$ have a *run-time state*




**Parallelizability:** Overlap execution of all $f_i$ for $k$ subsequent $x_j$

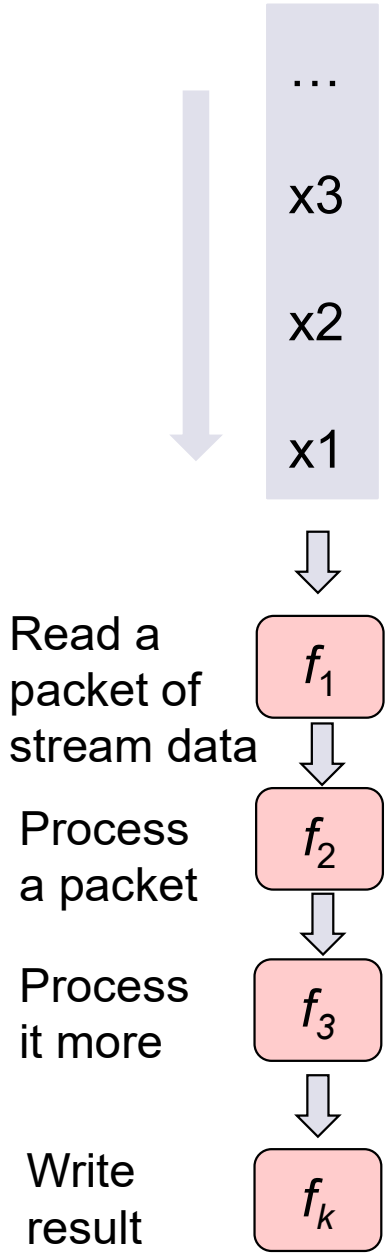
- *time=1: compute* $f_1(x_1)$
- *time=2: compute* $f_1(x_2)$ *and* $f_2(x_1)$
- *time=3: compute* $f_1(x_3)$ *and* $f_2(x_2)$ *and* $f_3(x_1)$
- *...*
- Total time: $O\left((n+k)\max_i(time(f_i))\right)$ with $k$ processors
- Still, requires good mapping of the tasks $f_i$ to the processors for even load balancing – often, static mapping (done before running)

**Notation** with higher-order function:

- $(y_1,\ldots,y_n) =$ ***pipe*** $(f_1, \ldots, f_k)\,(x_1,\ldots,x_n)$

# Streaming

- **Streaming** applies pipelining to processing of large (possibly, infinite) data *streams* from or to memory, network or devices, usually partitioned in fixed-sized data packets,
  - in order to **overlap** the processing of each packet of data **in time** with access of *subsequent* units of data and/or processing of preceding packets of data.
- Examples
  - Video streaming from network to display
  - Surveillance camera, face recognition
  - Network data processing e.g. deep packet inspection



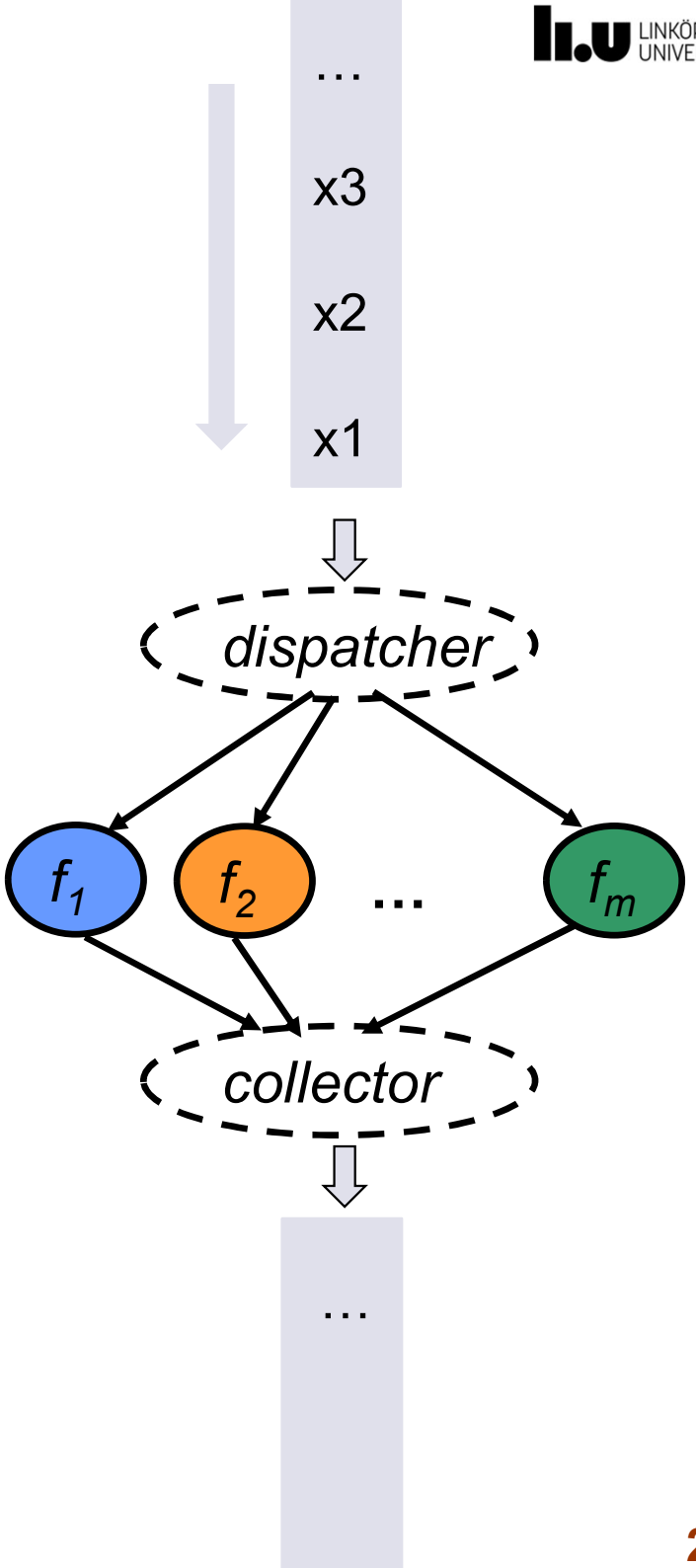
# Stream Farming

Combining streaming and task farming patterns

Independent streaming subcomputations $f_1, f_2, \ldots, f_m$ on each data packet

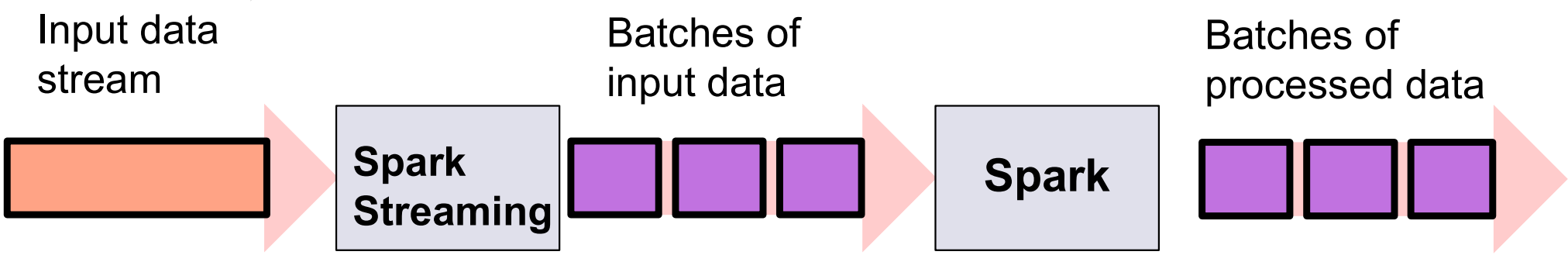
Speed up the pipeline by parallel processing of subsequent data packets

In most cases, the original order of packets must be kept after processing



# Spark Streaming

- Extension of the core Spark API
  for scalable, high-throughput, fault-tolerant stream processing of live data streams.
- **Discretized stream** or **DStream**
  - High-level abstraction representing a continuous stream of data.
  - Internally: A continuous series of RDDs



# Transformations on DStreams

- **map**( *func* ), **flatMap**( *func* ), **filter**( *func* ) – return a new DStream with **map** etc. applied to all its elements

- **repartition**(), **union**( other_stream )
- **count**() – returns a new DStream of single-element RDDs containing the number of elements in each RDD of the source DStream
- **reduce**( func ), **reduceByKey**() – aggregate each RDD of the source Dstream and return a new Dstream of single-element RDDs
- **join** ( other_stream ) – joins 2 streams of (K,V) and (K,W) pairs to a stream of (K,(V,W)) pairs
- **transform**( func ) – apply arbitrary RDD-to-RDD function to each RDD in the source DStream
- …

# Spark Streaming Example

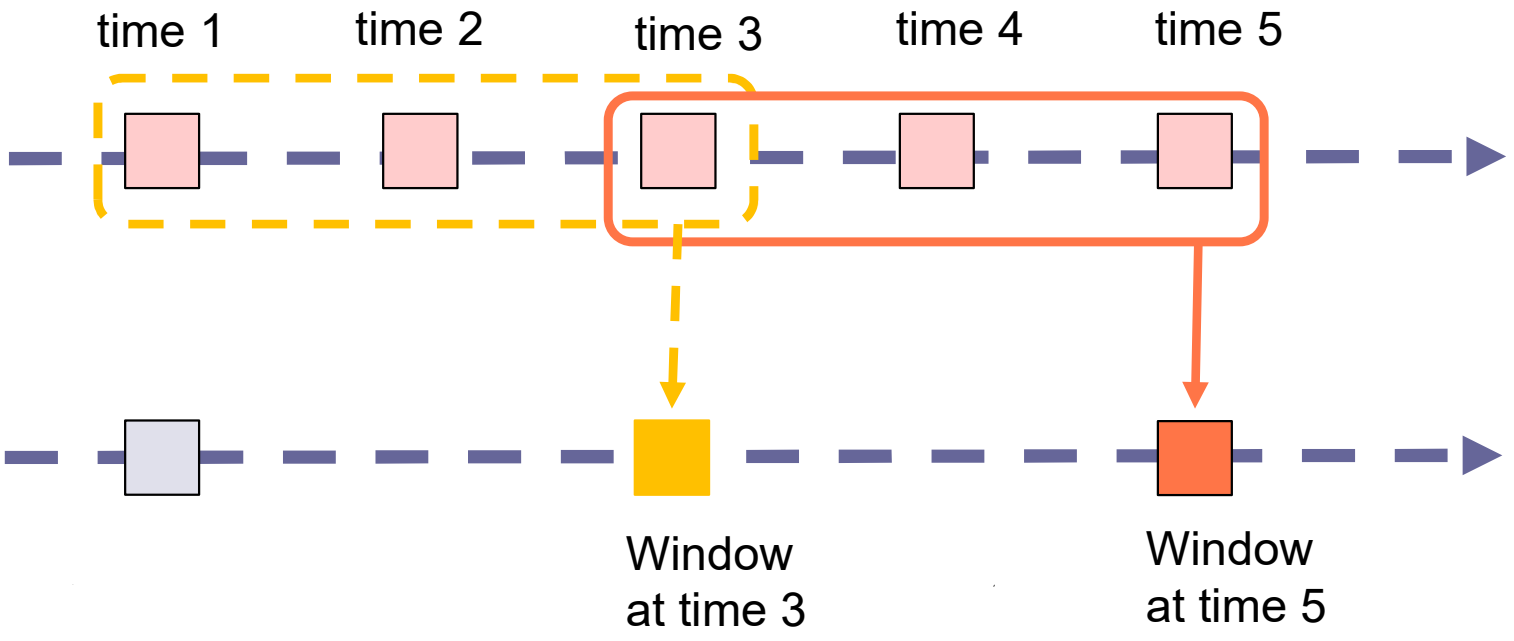
```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working threads and batch interval of 1 second:
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)

# Create a DStream that will connect to TCP hostname:port, like localhost:9999, as source:
lines = ssc.socketTextStream("localhost", 9999)

# Split each line into words:
words = lines.flatMap( lambda line: line.split(" ") )

# Count each word in each batch:
pairs = words.map( lambda word: (word, 1) )
wordCounts = pairs.reduceByKey( lambda x, y: x + y )

# Print the first ten elements of each RDD generated in this DStream to the console:
wordCounts.pprint()

ssc.start()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```

Run on local host, alt. cluster name

DStream of lines

# Spark Streaming: Windowing

- Can define a sliding window over a source DStream




**Window length** (here 3)
**Slide length** (here 2)
→ Overlap size (here 1)

Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.

```
# Example:  Reduce last 30 seconds of data, every 10 seconds:
windowedWordCounts =    \
  pairs.reduceByKeyAndWindow( lambda x, y: x + y, lambda x, y: x - y, 30, 10 )
```

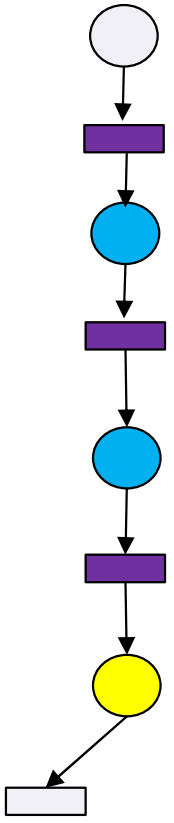
# APPENDIX

# Questions for Reflection

- Why can MapReduce emulate any distributed computation?
- For a Spark program consisting of 2 subsequent Map computations, show how Spark execution differs from Hadoop/MapReduce execution.
- Given is a file containing just integer numbers.
  Write a Spark program that adds them up.
- Write a wordcount program for Spark.
  - Solution proposal (from spark.apache.org):

```
text_file = sc.textFile("hdfs://...")

counts = text_file.flatMap( lambda line: line.split(" ") ) \
                  .map( lambda word: (word, 1) ) \
                  .reduceByKey( lambda a, b: a + b )

counts.saveAsTextFile("hdfs://...")
```

  - Note – there exist many variants for formulating this.
- Modify the wordcount program by only considering words with at least 4 characters.



# Map vs. FlatMap in Spark

**map**: transformation
RDD<T1> → RDD<T2>,
produces exactly one output element per input element.

- If T1 is List<T'>, only one output element per input list will be computed (usually also a list).
  I.e., the multidimensional structure of the RDD is preserved.

**flatmap**:
RDD<List<T1>> → RDD<T2>
= map + flatten: produce 0, 1 or several basic output elements per input element (which could be a list/array/struct) in the target RDD.

**Here (wordcount):** The input textfile (its elements are lines) is map'ed with the split function as operator. As a single line may contain multiple words, the result of each operator application (one per line) is a list of words (hence, overall an RDD of lists). Here, we are only interested in a single RDD of all words, without the line structure: the flatmap concatenates all words of all lists into one flat target RDD of words.

# Does Spark have a Combiner (as in MapReduce)?

- **reduceByKey** performs a full reduction by key *including* a combiner step, while **reduce** does not use a separate combiner step.
  - Input RDD must contain key-value pairs.
    - Whereas ordinary **reduce** works on "flat" RDDs of arbitrary element type.
  - The combiner step in **reduceByKey** counts as a *transformation,* not an action like **reduce**: it generates a RDD (of key-value pairs)
  - **reduceByKey** has a global dependence pattern (involves a shuffle-and-sort) but is still evaluated lazily
  - **reduceByKey** is a specialization of **aggregateByKey**
    - **aggregateByKey** takes 2 user functions: one that is applied to each block in the combiner step (sequentially) and one that is applied to reduce globally over the results of each block (in parallel).
      **reduceByKey** uses the same associative and commutative function in both steps.

- **combineByKey**() is a combiner working sequentially on each partition of a RDD, locally reducing it, producing a new RDD.
  - It is a *transformation* (evaluated lazily)
  - The input and output element types need not match.
  - The user function for **combining** must be associative only.
    - Always processed sequentially for each block.
    - But for **reduce**, the user function must be both associative and commutative.

# Transformations

Source: spark.apache.org

LINKÖPING UNIVERSITY

| Transformation | Meaning |
|---|---|
| **map**(*func*) | Returns a new RDD formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Returns a new RDD formed by selecting those elements of the source on which *func* returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). |
| **mapPartitions**(*func*) | Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> → Iterator<U> when running on an RDD of type T. |
| **mapPartitionsWithIndex** (*func*) | Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>)→Iterator<U> when running on an RDD of type T. |
| **sample** (*withReplacement*, *fraction*, *seed*) | Samples a fraction *fraction* of the data, with or without replacement, using a given random number generator seed. |
| **union**(*otherDataset*) | Returns a new dataset that contains the union of the elements in the source dataset and the argument. |

| Transformation | Meaning |
|---|---|
| **intersection**(*otherDataset*) | Return a new RDD that contains the intersection of elements in the source dataset and the argument. |
| **distinct**([*numPartitions*])) | Return a new dataset that contains the distinct elements of the source dataset. |
| **groupByKey** ([*numPartitions*]) | When called on a dataset of (K,V) pairs, returns a dataset of (K, Iterable<V>) pairs. If using grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. One can pass an optional *numPartitions* argument to set a different number of tasks. |
| **reduceByKey**(*func*, [*numPartitions*]) | When called on a dataset of (K,V) pairs, returns a dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V)→V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |

also: **combineByKey**

| Transformation | Meaning |
|---|---|
| **aggregateByKey**( *zeroValue*) (*seqOp*, *combOp*, [*numPartitions*]) | When called on a dataset of (K,V) pairs, returns a dataset of (K,U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |
| **sortByKey**([*ascending*], [*numPartitions*]) | When called on a dataset of (K,V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument. |
| **join**(*otherDataset*, [*numPartitions*]) | When called on datasets of type (K,V) and (K,W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. |
| **cogroup**(*otherDataset*, [*numPartitions*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith. |

| Transformation | Meaning |
|---|---|
| **cartesian**(*otherDataset*) | When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). |
| **pipe**(*command*, *[envVars]*) | Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to that process's stdin, and lines output to its stdout are returned as an RDD of strings. |
| **coalesce**(*numPartitions*) | Decreases the number of partitions in the RDD to *numPartitions*. Useful for running operations more efficiently after filtering down a large dataset. |
| **repartition**(*numPartitions*) | Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. |
| **repartitionAndSortWithinPartitions**(*partitioner*) | Repartitions the RDD according to the given *partitioner* and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery. |

# Actions

| Action | Meaning |
|---|---|
| **reduce**(*func*) | Aggregates the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| **collect**() | Returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. |
| **count**() | Returns the number of elements in the dataset. |
| **first**() | Returns the first element of the dataset (similar to take(1)). |
| **take**(*n*) | Returns an array with the first *n* elements of the dataset. |
| **takeSample**(*withReplacement*, *num*, [*seed*]) | Returns an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed. |
| **takeOrdered**(*n*, *[ordering]*) | Returns the first *n* elements of the RDD using either their natural order or a custom comparator. |

| Action | Meaning |
|---|---|
| **saveAsTextFile**(*path*) | Write the elements of the RDD as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other supported file system. Spark will call toString on each element to convert it to a line of text in the file. |
| **saveAsSequenceFile**(*path*) (Java and Scala) | Write the elements of the RDD as a SequenceFile in a given path in the local filesystem, HDFS or any other supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |
| **saveAsObjectFile**(*path*) (Java and Scala) | Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile(). |
| **countByKey**() | Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. |
| **foreach**(*func*) | Runs a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. |

# References


- M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, I. Stoica: Spark: Cluster Computing with Working Sets. Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (*HotCloud'10*), 2010, ACM.
  - See also: M. Zaharia *et al.*: Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56-65, Nov. 2016.
- Apache Spark: http://spark.apache.org
- A. Nandi: *Spark for Python Developers*. Packt Publishing, 2015.