Spark

Alan Sussman, Department of Computer Science

With thanks to D. Wheeler (GMU), T. Yang (UCSB) and Apache documentation

UNIVERSITY OF MARYLAND

Announcements

- Assignment 4 due Tuesday, May 2
  - Questions?
- Project 3 grades posted
  - Ask TAs if you have questions about the grading
- 3rd quiz will be last week of class – start Tuesday or Wednesday?

Apache Spark

- Processing engine; instead of just "map" and "reduce", defines a large set of *operations* (transformations & actions)
  - Operations can be arbitrarily combined in any order
- Open source software
- Supports Java, Scala and Python
- Key construct: Resilient Distributed Dataset (RDD)

Resilient Distributed Dataset (RDD)

- An **RDD** is a fault-tolerant collection of elements that can be operated on in parallel
- RDDs represent data or transformations on data
- Two ways to create RDDs: *parallelizing* an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat
  - or by transforming other RDDs (you can stack RDDs)
- Actions can be applied to RDDs; actions force computations and return values
- Lazy evaluation: Nothing computed until an action requires it
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
  - Less suitable for applications that make asynchronous fine-grained updates to shared state

Spark example #1 (Scala)

```
// "sc" is a "Spark context" – this transforms the file into an RDD
val textFile = sc.textFile("README.md")

// Return number of items (lines) in this RDD; count() is an action
textFile.count()

// Demo filtering.  Filter is a transform.  By itself this does no real work
val linesWithSpark = textFile.filter(line => line.contains("Spark"))

// Demo chaining – how many lines contain "Spark"?  count() is an action.
textFile.filter(line => line.contains("Spark")).count()

// Length of line with most words.  Reduce is an action.
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)

// Word count – traditional map-reduce.  collect() is an action
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)

wordCounts.collect()
```

https://spark.apache.org/docs/latest/quick-start.html

Spark example #2 (Python)

 # Estimate π (compute-intensive task).

 # Pick random points in the unit square ((0, 0) to (1,1)),

 # See how many fall in the unit circle. The fraction should be π / 4

 # Note that "parallelize" method creates an RDD

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

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

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

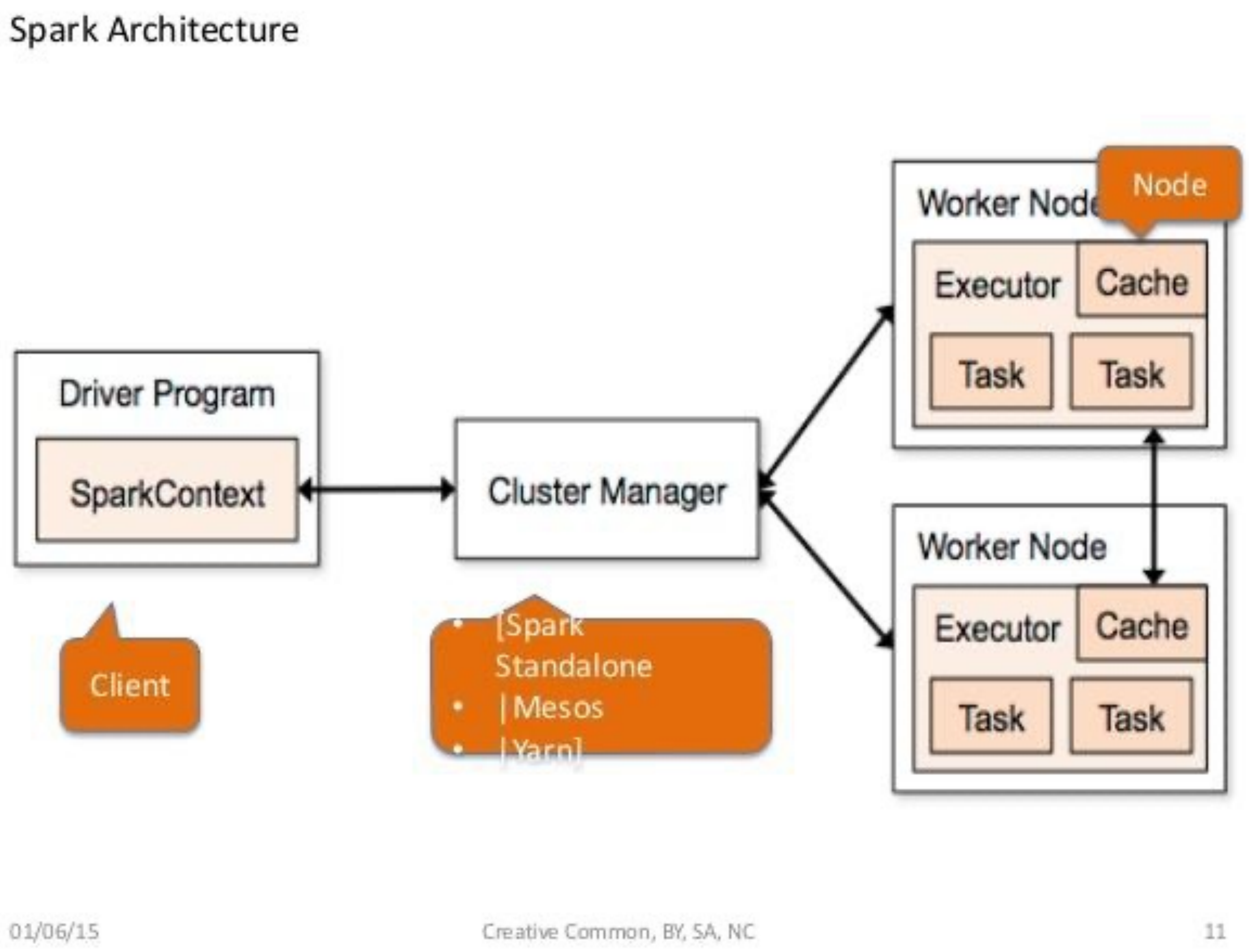
https://spark.apache.org/docs/latest/quick-start.html

Parallelized Collections

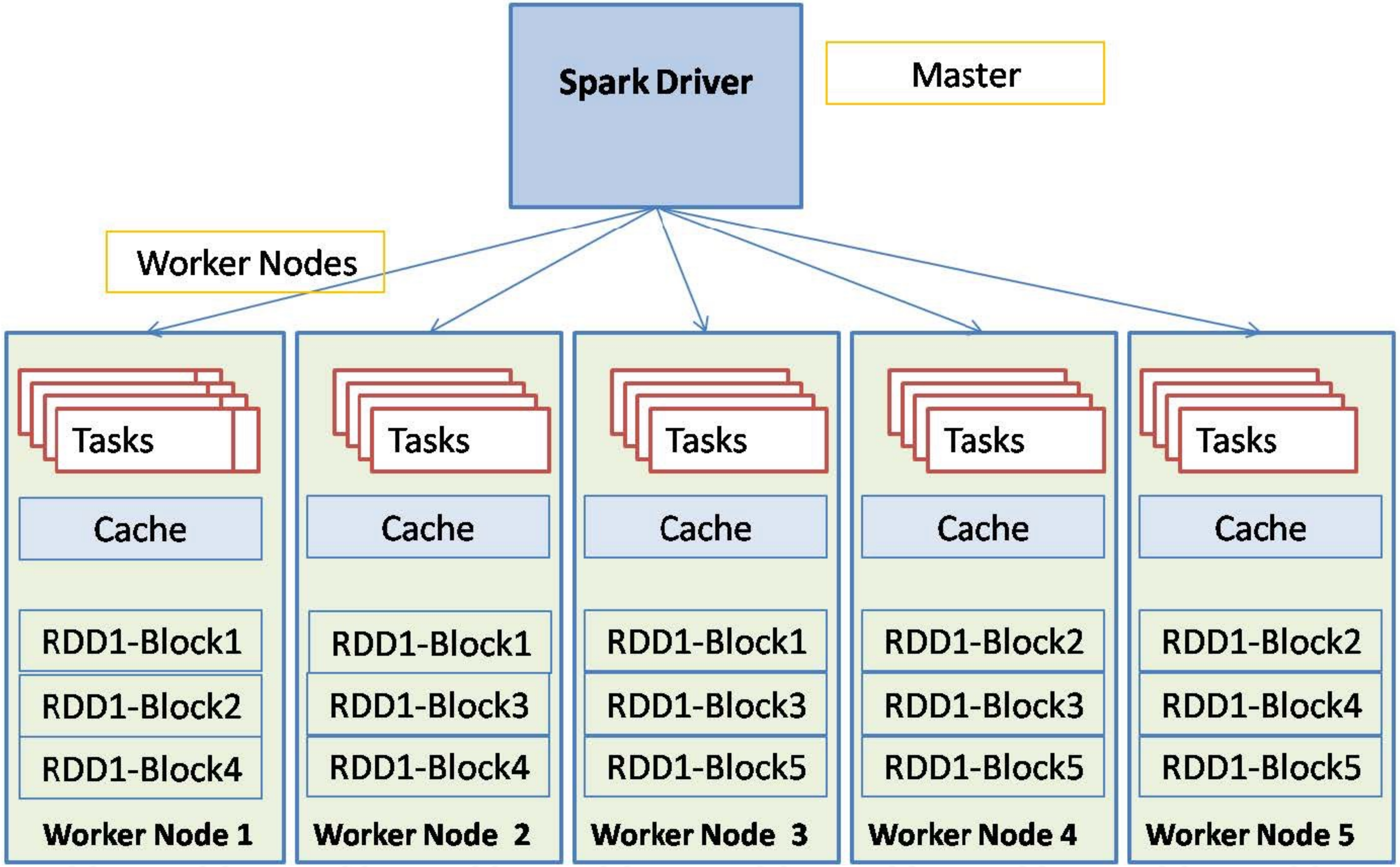
- Parallelized collections are created by calling SparkContext's *parallelize* method on an existing iterable or collection in your driver program, for Python (similar idea for Scala or Java)

- Important parameter for parallel collections is the number of *partitions* to cut the dataset into
  - Spark will run one task for each partition of the cluster
  - Typically you want 2-4 partitions for each CPU/core in your cluster
  - Spark tries to set the number of partitions automatically based on your cluster
  - But you can also set it manually by passing it as a second parameter to *parallelize*

Spark Architecture

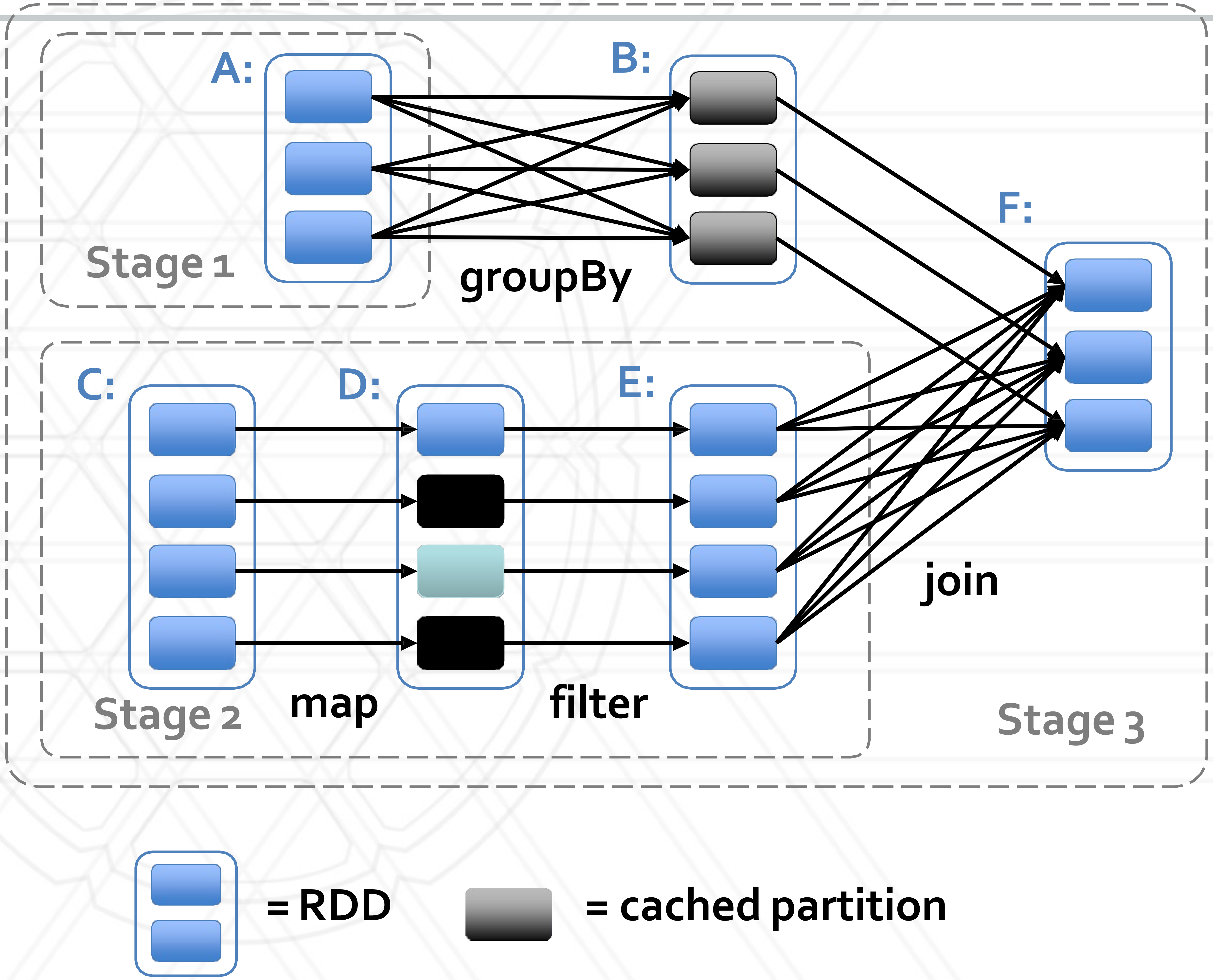


Spark Components



Under the Hood

- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles



Spark transformations

- **map**(func): Return a new distributed dataset formed by passing each element of the source through a function func
- **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)
- **filter**(func): Return a new dataset formed by selecting those elements of the source on which func returns true
- **union**(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
- **intersection**(otherDataset): Return a new RDD that contains the intersection of elements in the source dataset and the argument.

https://spark.apache.org/docs/latest/rdd-programming-guide.html

Spark transformations

- **distinct**([numTasks])): Return a new dataset that contains the distinct elements of the source dataset
- **join**(otherDataset, [numTasks]): 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.

Spark transformations

- **groupByKey**([numPartitions]): When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
  - **Note:** If you are 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
  - **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You 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.

Spark transformations

- **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 from 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.
- And a lot more …

Sample Spark Actions

- **reduce**(func): Aggregate 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**(): Return 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**(): Return the number of elements in the dataset
- **take**(n): Ret … t.

Actions **cause calculations to be performed;**
transformations **just set things up (lazy evaluation)**

…dd-programming-guide.html

Spark – RDD Persistence

- RDDs automatically recover from node failure, with no assistance from user
- You can explicitly persist (cache) an RDD
- When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it)
- Allows future actions to be much faster (often >10x).
- Mark RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes.
- Cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it
- Can choose storage level (MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, etc.)
  - Default is MEMORY_ONLY
- Can manually call unpersist()
  - Otherwise cache is managed by Spark with an LRU policy

Spark Example #3 (Python)

```
# Logistic Regression - iterative machine learning algorithm
# Find best hyperplane that separates two sets of points in a
# multi-dimensional feature space.  Applies MapReduce operation
# repeatedly to the same dataset, so it benefits greatly
# from caching the input in RAM

points = spark.textFile(...).map(parsePoint).cache()

w = numpy.random.ranf(size = D) # current separating plane

for i in range(ITERATIONS):

    gradient = points.map(

        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x

    ).reduce(lambda a, b: a + b)

    w -= gradient

print "Final separating plane: %s" % w
```

https://spark.apache.org/docs/latest/quick-start.html

Spark Example #3 (Scala)

```
// Same thing in Scala
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: " + w)
```

https://spark.apache.org/docs/latest/quick-start.html

Spark Example #3 (Java)

```
// Same thing in Java

class ComputeGradient extends Function<DataPoint, Vector> {
  private Vector w;

  ComputeGradient(Vector w) { this.w = w; }
  public Vector call(DataPoint p) {
    return p.x.times(p.y * (1 / (1 + Math.exp(w.dot(p.x))) - 1));
  }

}

JavaRDD<DataPoint> points = spark.textFile(...).map(new ParsePoint()).cache();
Vector w = Vector.random(D); // current separating plane

for (int i = 0; i < ITERATIONS; i++) {
  Vector gradient = points.map(new ComputeGradient(w)).reduce(new AddVectors());
  w = w.subtract(gradient);
}

System.out.println("Final separating plane: " + w);
```

https://spark.apache.org/docs/latest/quick-start.html

Broadcast variables

- Allow keeping a read-only variable cached on each machine in the cluster, instead of shipping with tasks
  - e.g., to give every node a copy of a large input dataset
  - Can use efficient broadcast algorithms to reduce communication costs
- Broadcast variable created and used like this:

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>

>>> broadcastVar.value
[1, 2, 3]
```

Accumulators

- Variables that are only "added" to through an associative and commutative operation
  - so can be efficiently supported in parallel
- Can be used to implement counters (as in MapReduce) or sums
- Spark natively supports accumulators of numeric types, and programmers can add support for new types
- Example of accumulator used to sum elements in an array:

```
>>> accum = sc.accumulator(0)
>>> accum
Accumulator<id=0, value=0>

>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

>>> accum.value
10
```

Shuffle Operations

- Spark's mechanism for re-distributing data so that it is grouped differently across partitions
- Typically involves copying data across executors and machines, making shuffle a complex and costly operation
- Examples where it is needed include *reduceByKey, groupByKey, join, repartition*
- Expensive because requires disk I/O, network I/O and data serialization
  - Can use a lot of heap memory for in-memory data structures to organize records (before or after data transfers)
  - Can also generate a lot of intermediate files on disks, which are preserved until the corresponding RDDs are no longer used and are garbage collected
    - so the shuffle files don't need to be re-created if the lineage is re-computed (e.g., because of a node failure)

Apache Spark: Libraries "on top" of core that come with it

- Spark SQL – for structured data processing
  - *Datasets* for distributed data collections, *DataFrames* for *Datasets* organized into named columns (tables!)
- Spark Structured Streaming – stream processing of live datastreams
- MLlib - machine learning library - DataFrame-based API is now primary API (means no new features for RDDs)
- GraphX – graph manipulation
  - extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge.
- SparkR (R on Spark) – lightweight frontend to use Spark from R (distributed DataFrame operations on large datasets)

Gray sort competition: Winner Spark-based (previously Hadoop)

	Hadoop MR Record	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**

Spark-based System 3x faster with 1/10 # of nodes

Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)

http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

Spark vs. Hadoop MapReduce

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to do well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: Spark is easier to program
- Data processing: Spark more general

"Spark vs. Hadoop MapReduce" by Saggi Neumann (March 2023)
https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

For more information

- More Spark examples at http://spark.apache.org/examples.html
- Spark (and Hadoop) Coursera tutorial
  - https://www.coursera.org/learn/introduction-to-big-data-with-spark-hadoop
- "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing" by Matei Zaharia et. al
  - http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf
- Other Spark papers listed at
  - https://spark.apache.org/research.html

UNIVERSITY OF
MARYLAND