Apache Spark

Yogesh Simmhan

Slide Credits:

- https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf
- https://www.slideshare.net/deanchen11/scala-bay-spark-talk
- https://databricks-training.s3.amazonaws.com/slides/advanced-spark-training.pdf
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, M. Zaharia, et al., NSDI 2012
- http://spark.apache.org/docs/latest/programming-guide.html

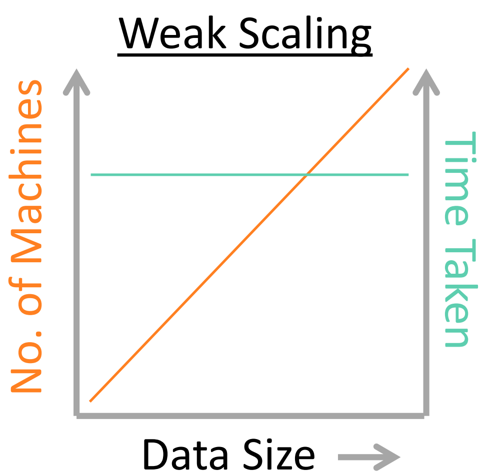
Distributed Systems

- Distributed Computing
  - Clusters of machines
  - Connected over network
- Distributed Storage
  - Disks attached to clusters of machines
  - Network Attached Storage
- *How can we make effective use of multiple machines?*

- **Commodity** clusters vs. **HPC** clusters
  - Commodity: Available off the shelf at large volumes
  - Lower Cost of Acquisition
  - Cost vs. Performance
    - Low disk bandwidth, and high network latency
    - CPU typically comparable (Xeon vs. i3/5/7)
    - Virtualization overhead on Cloud
- *How can we use many machines of modest capability?*

Scalability

- Strong vs. Weak Scaling
- **Strong Scaling**: How the performance varies with the # of processors for a *fixed total problem size*
- **Weak Scaling**: How the performance varies with the # of processors for a *fixed problem size per processor*
  - Big Data platforms are intended for "Weak Scaling"



Ease of Programming

- Programming distributed systems is difficult
  - Divide a job into multiple tasks
  - Understand dependencies between tasks: Control, Data
  - Coordinate and synchronize execution of tasks
  - Pass information between tasks
  - Avoid race conditions, deadlocks
- Parallel and distributed programming models/languages/abstractions/platforms try to make these easy
  - E.g. Assembly programming vs. C++ programming
  - E.g. C++ programming vs. Matlab programming

Availability, Failure

- Commodity clusters have lower reliability
  - Mass-produced
  - Cheaper materials
  - Smaller lifetime (~3 years)
- *How can applications easily deal with failures?*
- *How can we ensure availability in the presence of faults?*

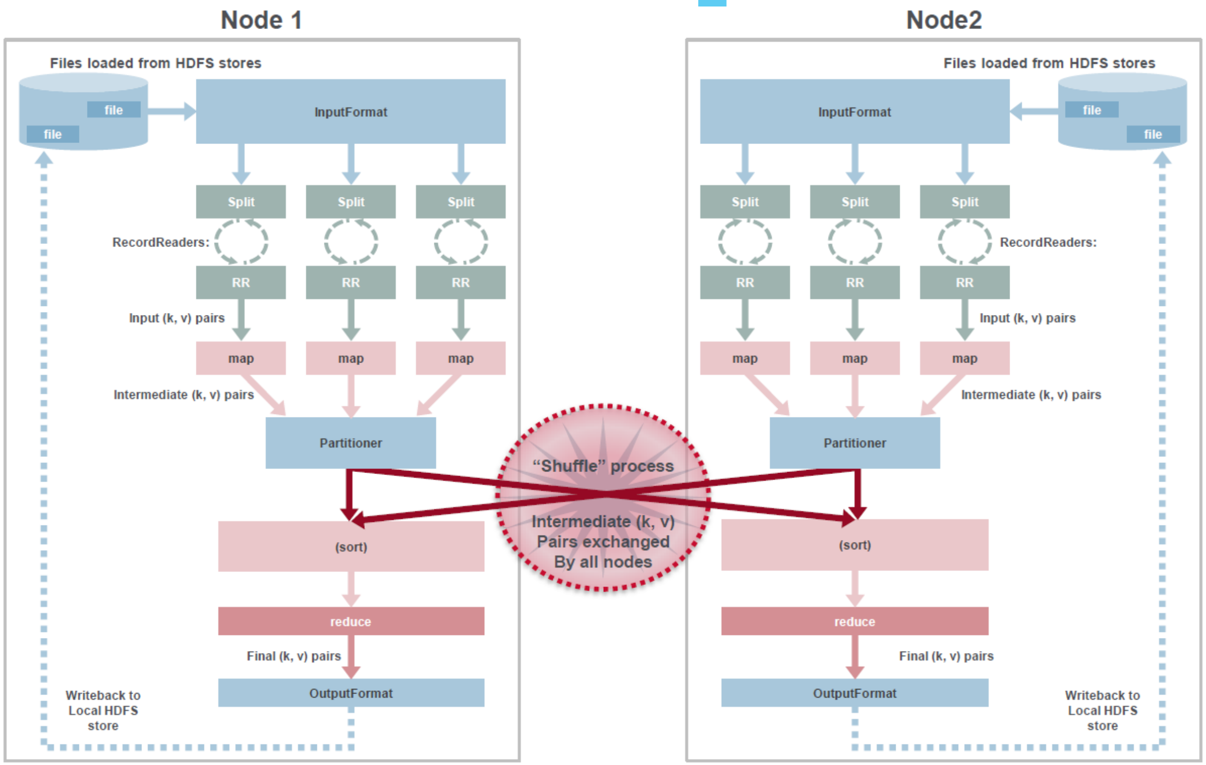
MapReduce, the *classic* Big Data platform

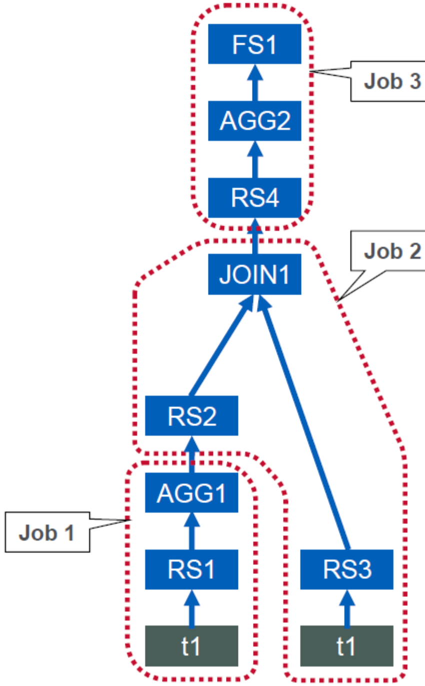
- Eased distributed data-parallel programming
  - Simple primitives: **Map, Reduce**, *Combiner, Partitioner*
  - Avoids race conditions
- Offered reliable storage (HDFS) and execution
- Scaled to large datasets

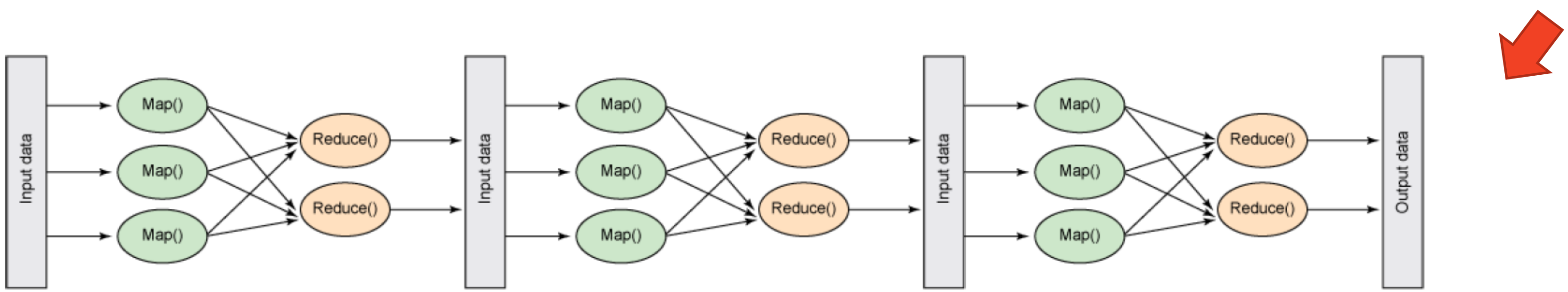
Challenges with MapReduce

- Shuffle between Map and Reduce
  - Allows results larger than RAM, enables recovery
  - But forces a global key-sort and disk writes
- Multi-stage jobs write output of Reduce/Input to Map to HDFS
  - I/O ops forced at each "stage"
  - Scheduling overheads for each MR job
- *Can we do better?*

The MR Pipeline





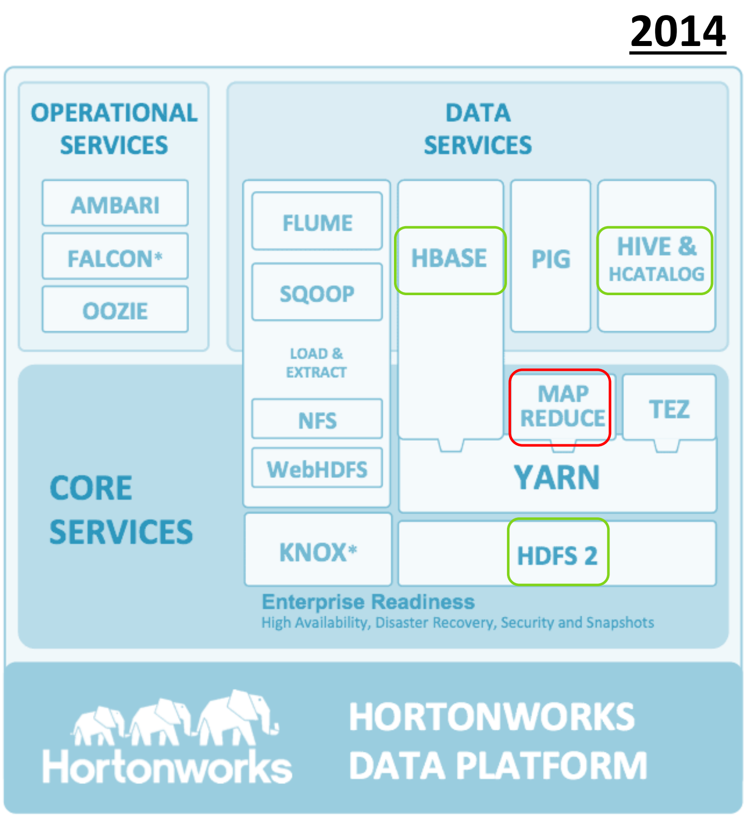


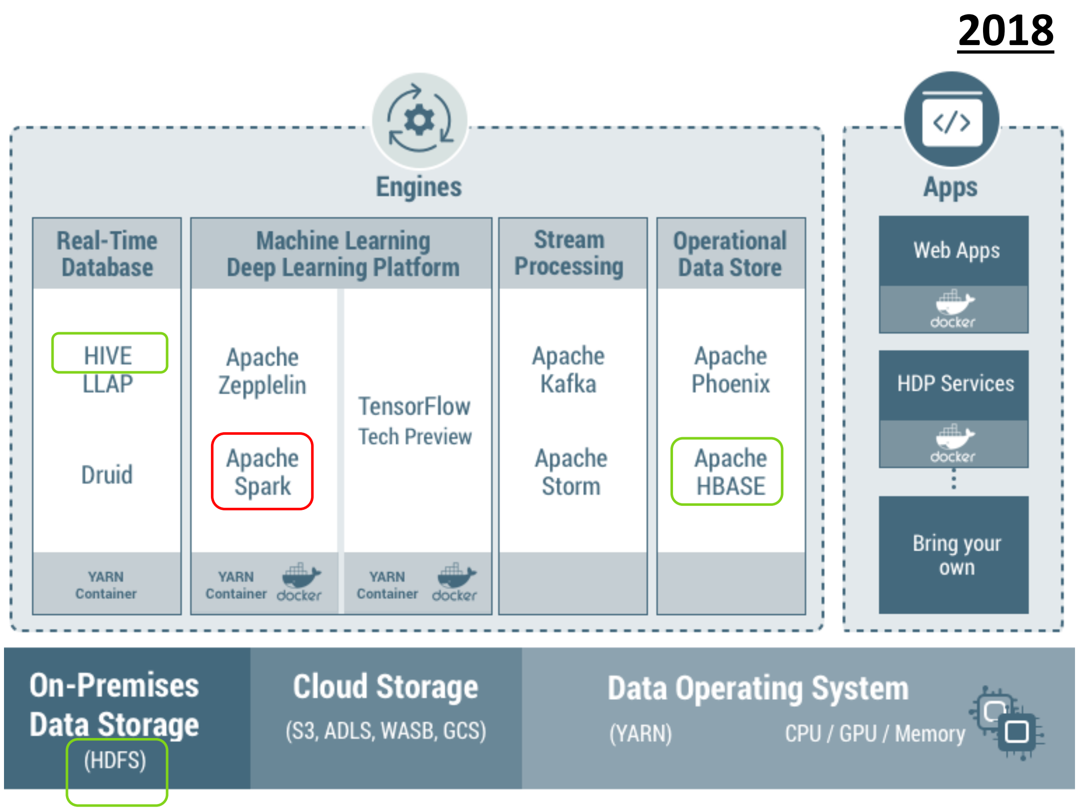
Introduction to Spark on Hadoop, Carol McDonald, MapR, https://www.slideshare.net/caroljmcdonald/introduction-to-spark-on-hadoop

Why Spark?

- Ease of language definition
  - Data typing, dataflow composition, Java/Python/Scala bindings
  - But Pig, Hive, HBase, etc. give you that
- Better performance using "In memory" compute
  - Multiple stages part of same job
  - Lazy evaluation, caching/persistence

Big Data Stack Evolution

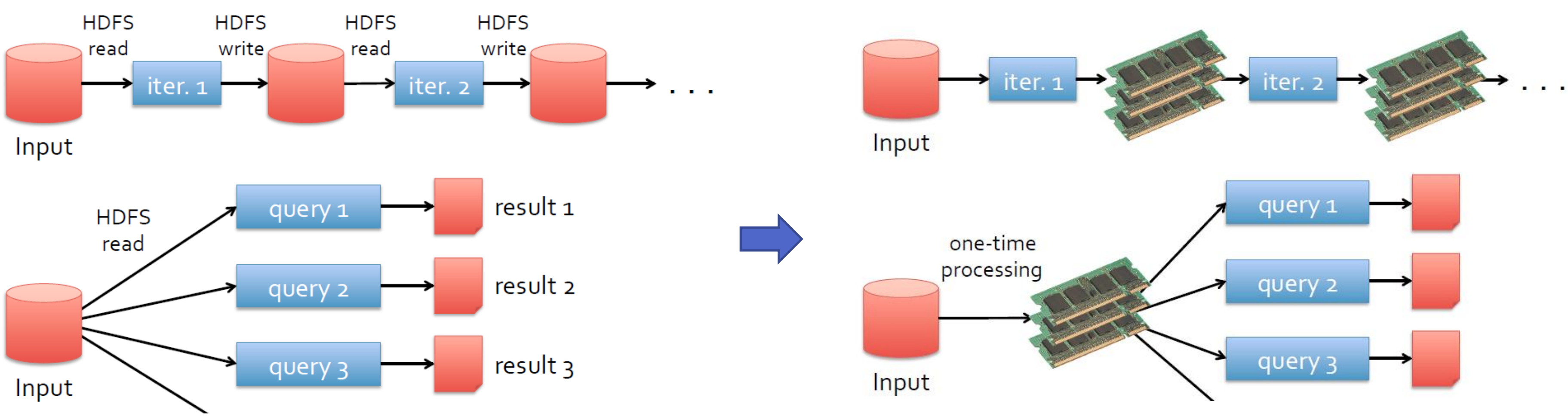




https://hortonworks.com/blog/using-hdp-hadoop-platform-service/
https://hortonworks.com/products/data-platforms/hdp/

In-memory computation

- Operate on data in (distributed) memory
  - Allows many operations to be performed locally
  - Write to disk only when data sharing required across workers

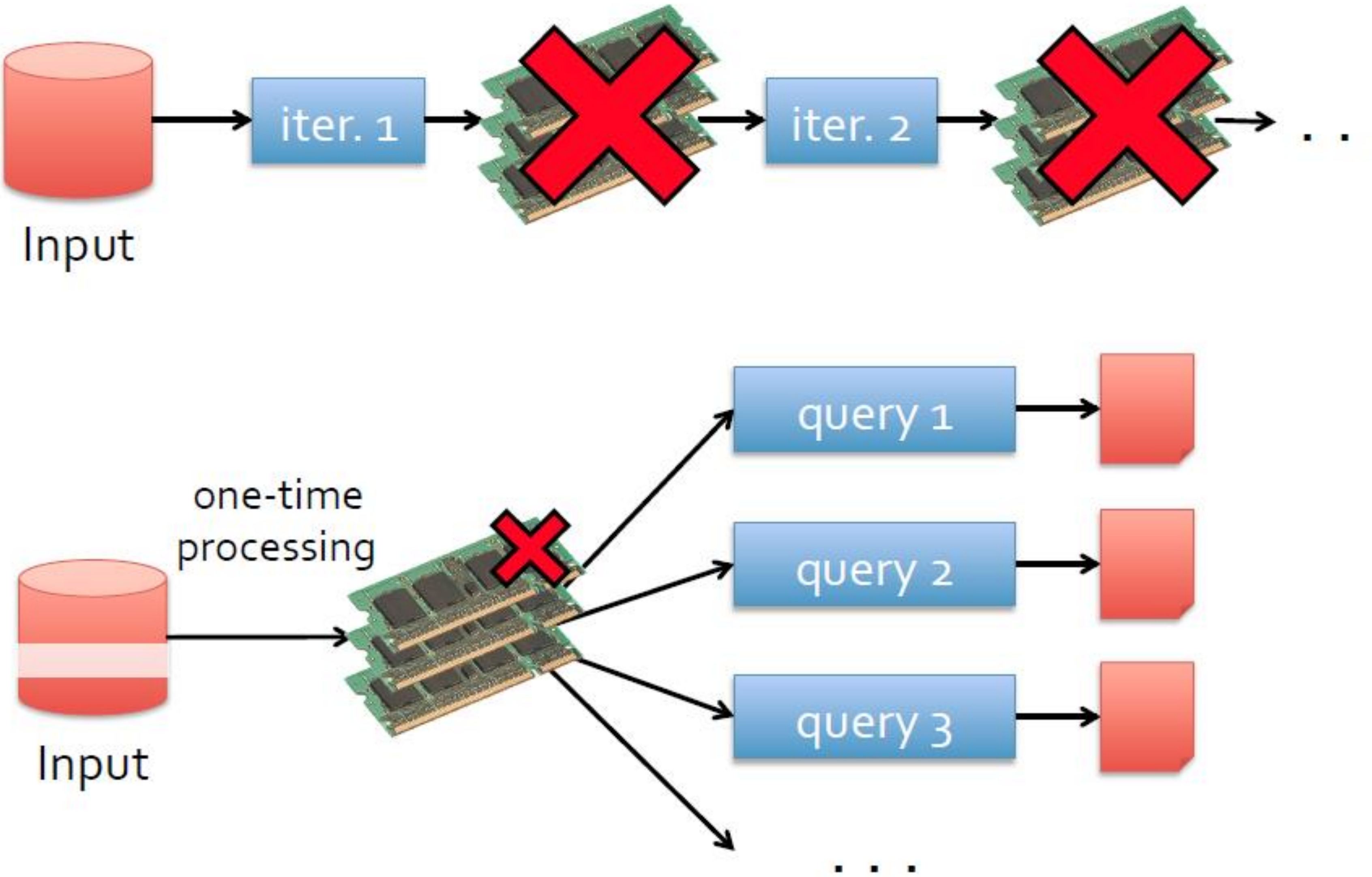


Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, M. Zaharia, et al., *NSDI* 2012

RDD: The Secret Sauce

- RDD: Resilient Distributed Dataset
  - Immutable, partitioned collection of tuples
  - Operated on by *deterministic* transformations
    - Object-oriented flavor
    - RDD.operation() → RDD
- Recovery by re-computation
  - Maintains lineage of transformations
  - Recompute missing partitions *if failure happens*
  - Not possible/not automatic in Pig
- Allows caching & persistence for reuse

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, M. Zaharia, et al., *NSDI* 2012


iter. 1
iter. 2
. . .
Input
one-time
processing
query 1
query 2
query 3
. . .
Input

RDD Operations

Allows composability into Dataflows

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

A Sample Spark Program

- Counts the number of bytes in a line, and sums the count per line
- Uses lambda expressions for compact function defn.

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths =
       lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);

// Cache RDD in-memory for future use in this app
lineLengths.persist(StorageLevel.MEMORY_ONLY());
```

A Sample Spark Program

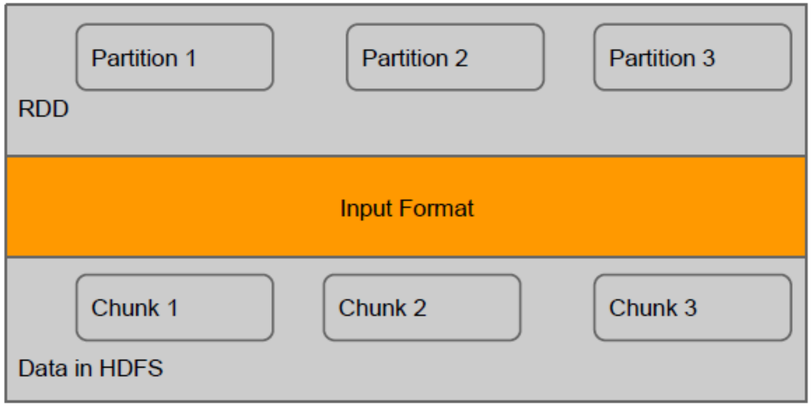
- Can pass complex functions as well

```
class GetLength implements Function<String, Integer> {
  public Integer call(String s) { return s.length(); }
}
class Sum implements Function2<Integer, Integer, Integer> {
  public Integer call(Integer a, Integer b) { return a + b; }
}

JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new GetLength());
int totalLength = lineLengths.reduce(new Sum());
```

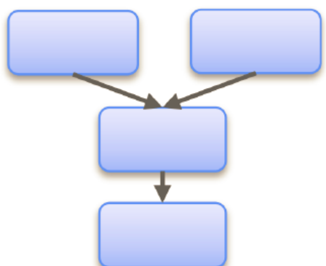
RDD Partitions

- RDD is internally a collection of partitions
  - Each partition holds a list of items
- Partitions may be present on a different machine
  - Partition is the *unit of execution*
  - Partition is the *unit of parallelism*
- They are immutable
  - Each transformation on an RDD generates a new RDD with different partitions
  - *Allows recovery of individual partitions*



Distributed Execution

RDD Objects

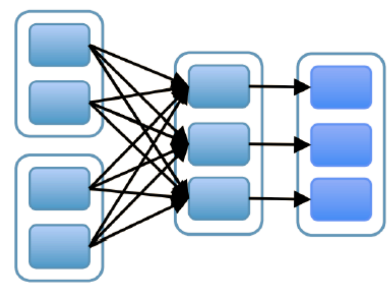


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
    .count()
```

build operator DAG

DAG

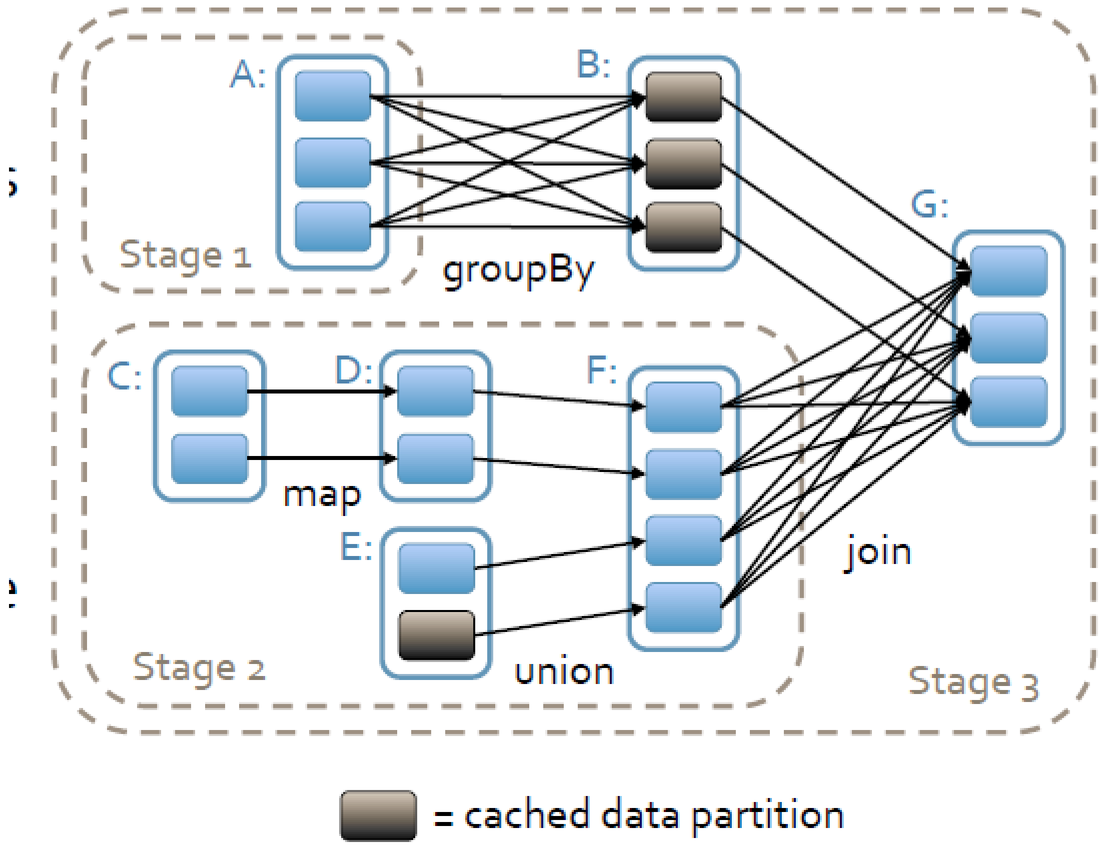
Scheduler
(DAGScheduler)



split graph into
stages of tasks

submit each
stage as ready

Distributed Execution



Creating RDD

- Load external data from distributed storage
- Create logical RDD on which you can operate
- Support for different input formats
  - HDFS files, Cassandra, Java serialized, directory, gzipped
- Can control the number of partitions in loaded RDD
  - Default depends on external DFS, e.g. 128MB on HDFS

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

RDD Operations

- Transformations
  - From one RDD to one or more RDDs
  - Lazy evaluation…*use with care*
  - Executed in a distributed manner
- Actions
  - Perform aggregations on RDD items
  - Return single (or distributed) results to “driver” code
- RDD.collect() brings RDD partitions to single driver machine

Caution: Local Variables

- Caution: Cannot pass "local" driver variables to lambda expressions/anonymous classes....only **final**
  - Will fail when distributed

```
int counter = 0;
JavaRDD<Integer> rdd = sc.parallelize(data);

// Wrong: Don't do this!!
rdd.foreach(x -> counter += x);

println("Counter value: " + counter);
```

RDD and PairRDD

- RDD is logically a collection of items with a generic type
- PairRDD is like a "Map", where each item in collection is a <key,value> pair, each a generic type
- Transformation functions use RDD or PairRDD as input/output
- E.g. Map-Reduce

```
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);
```

Transformations

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset 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).

- **JavaRDD<R> map(Function<T,R> f)** : 1:1 mapping from input to output. Can be different types.
- **JavaRDD<T> filter(Function<T,Boolean> f)** : 1:0/1 from input to output, same type.
- **JavaRDD<U> flatMap(FlatMapFunction<T,U> f)** : 1:N mapping from input to output, different types.

Transformations

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.

- Earlier Map and Filter operate on one item at a time. No state across calls!
- **JavaRDD<U> mapPartitions(FlatMapFunc<Iterator<T>,U> f)**
- mapPartitions has access to iterator of values in entire partition, jot just a single item at a time.

Transformations

sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.

- **JavaRDD<T> sample(boolean withReplacement, double fraction)**: fraction between [0,1] without replacement, >0 with replacement
- **JavaRDD<T> union(JavaRDD<T> other)**: Items in other RDD added to this RDD. Same type. Can have duplicate items (i.e. not a 'set' union).

Transformations

intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.

- **JavaRDD<T> intersection(JavaRDD<T> other)**: Does a set intersection of the RDDs. Output *will not* have duplicates, even if inputs did.
- **JavaRDD<T> distinct()**: Returns a new RDD with unique elements, eliminating duplicates.

Transformations: **PairRDD**

groupByKey([*numTasks*])	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 `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	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.

- **JavaPairRDD<K,Iterable<V>> groupByKey()**: Groups values for each key into a single iterable.
- **JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> func)** : Merge the values for each key into a single value using an *associative* and *commutative* reduce function. Output value is of same type as input.
- For aggregate that returns a different type?
- **numPartitions** can be used to generate output RDD with different number of partitions than input RDD.

Transformations

aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	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*], [*numTasks*])	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.

- **JavaPairRDD<K,U> aggregateByKey(U zeroValue, Function2<U,V,U> seqFunc, Function2<U,U,U> combFunc)** : Aggregate the values of each key, using given combine functions and a neutral “zero value”.
  - **SeqOp** for merging a V into a U within a partition
  - **CombOp** for merging two U's, within/across partitions
- **JavaPairRDD<K,V> sortByKey(Comparator<K> comp)**: Global sort of the RDD by key
  - *Each partition* contains a sorted range, i.e., output RDD is range-partitioned.
  - Calling collect will return an ordered list of records

Transformations

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`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

- **JavaPairRDD<K, Tuple2<V,W>> join(JavaPairRDD<K,W> other, int numParts):** Matches keys in *this* and *other*. Each output pair is (k, (v1, v2)). Performs a hash join across the cluster.
- **JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)**: Cross product of values in each RDD as a pair

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.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.

RDD Persistence & Caching

- RDDs can be reused in a dataflow
  - Branch, iteration
- But it will be re-evaluated each time it is reused!
- *Explicitly persist* RDD to reuse output of a dataflow path multiple times
- Multiple storage levels for persistence
  - Disk or memory
  - Serialized or object form in memory
  - Partial spill-to-disk possible
  - *Cache* indicates “persist” to memory

RePartitioning

repartition

```
public JavaRDD<T> repartition(int numPartitions)
```

Return a new RDD that has exactly numPartitions partitions.

Can increase or decrease the level of parallelism in this RDD. Internally, this uses a shuffle to redistribute data.

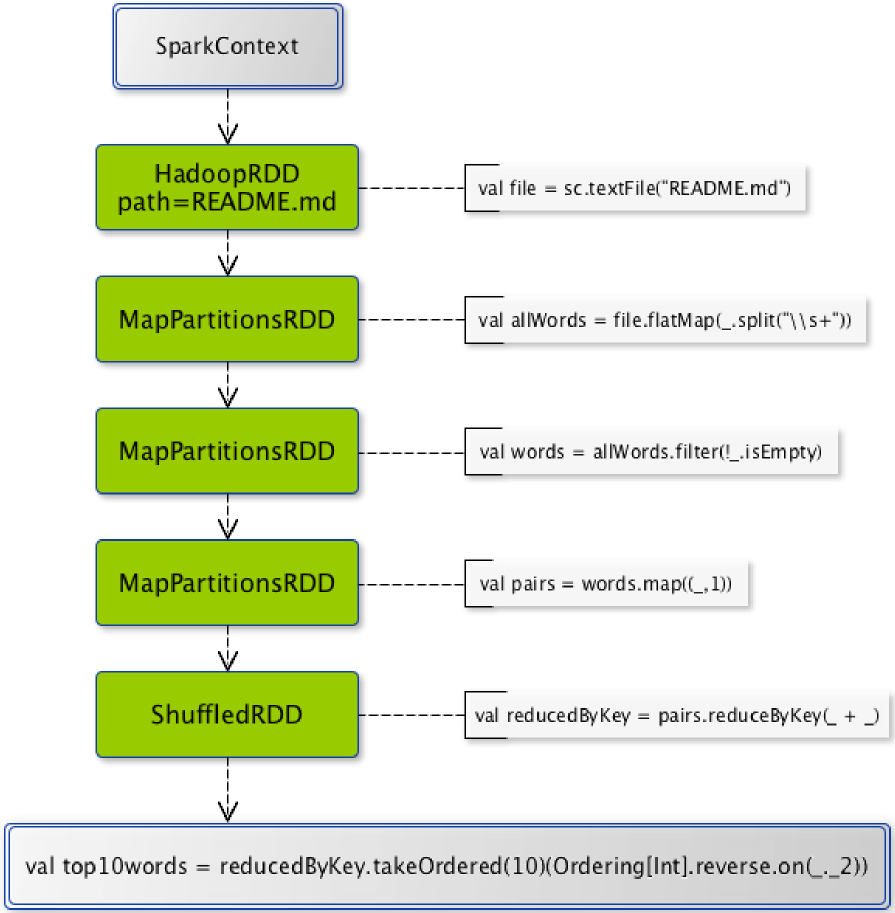
If you are decreasing the number of partitions in this RDD, consider using `coalesce`, which can avoid performing a shuffle.

coalesce

```
public JavaRDD<T> coalesce(int numPartitions,
                           boolean shuffle)
```

Return a new RDD that is reduced into `numPartitions` partitions.

From DAG to RDD lineage



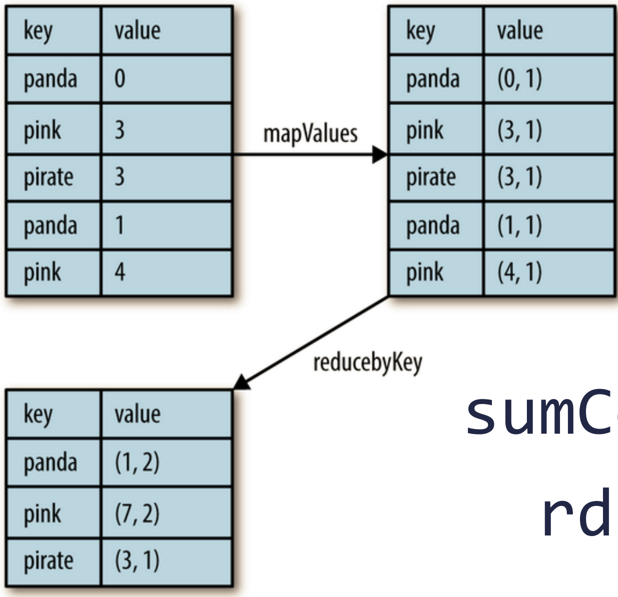
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd-transformations.html

Samples: Word Count

```
rdd = sc.textFile(“hdfs://...");
words = rdd.flatMap(x -> x.split(" "));
result = words.map(x->(x,1)).
  reduceByKey((x, y): x + y);
```


https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/ch04.html

Samples: Per-key average




```
sumCount =
  rdd.mapValues(x -> (x,1)).
    reduceByKey((x, y) ->
      (x[0]+y[0], x[1]+y[1]))
```


https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/ch04.html

PageRank

- Centrality measure of web page quality based on the web structure
  - How important is this vertex in the graph?
- Random walk
  - Web surfer visits a page, randomly clicks a link on that page, and does this repeatedly.
  - How frequently would each page appear in this surfing?
- Intuition
  - Expect high-quality pages to contain “endorsements” from many other pages thru hyperlinks
  - Expect if a high-quality page links to another page, then the second page is likely to be high quality too

PageRank, recursively

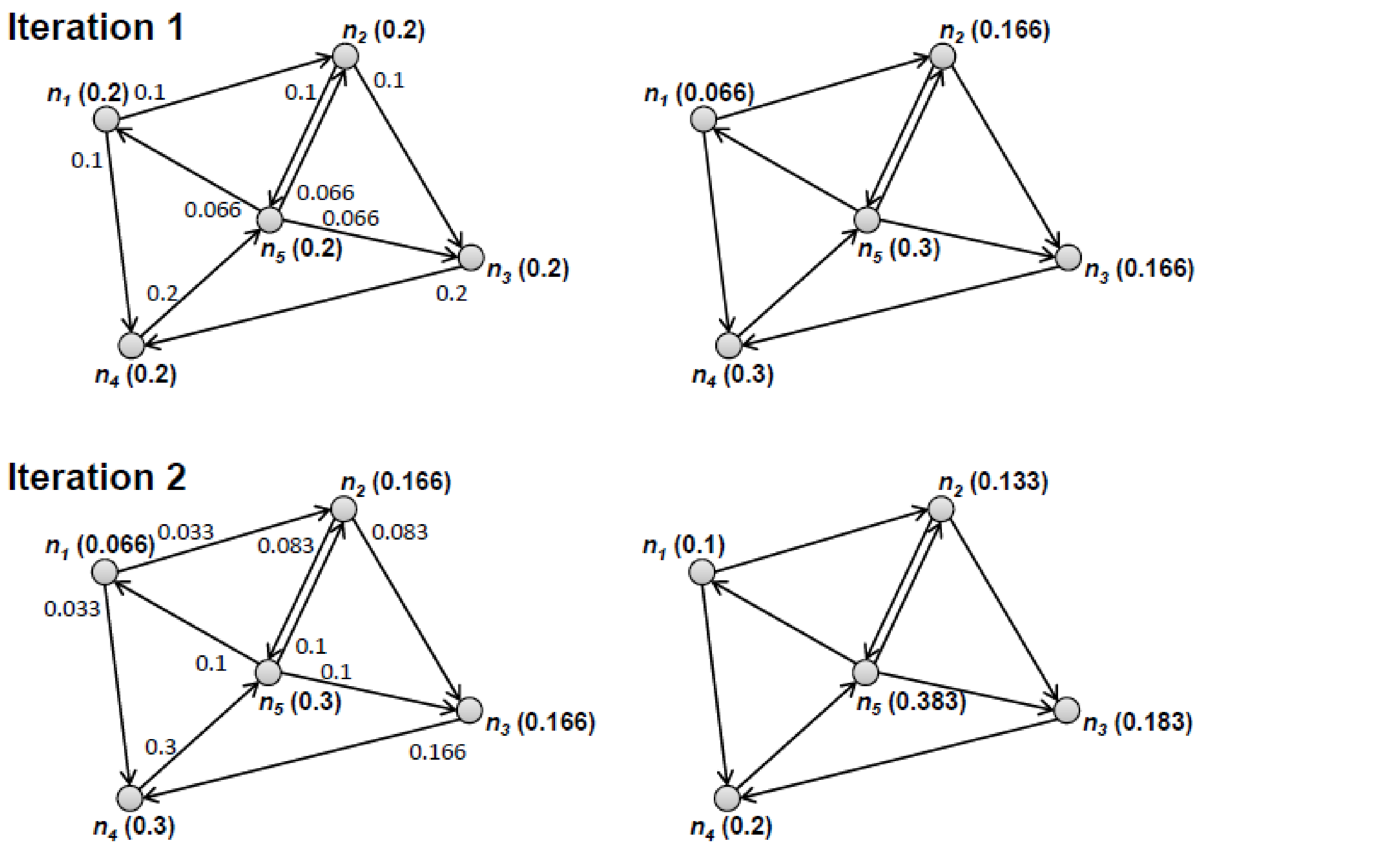
$$P(n) = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

- P(n) is PageRank for webpage/URL ‘n’
  - Probability that you’re in vertex ‘n’
- |G| is number of URLs (vertices) in graph
- α is probability of random jump
- L(n) is set of vertices that link to ‘n’
- C(m) is out-degree of ‘m’

PageRank Iterations

α=0

Initialize P(n)=1/|G|



Lin, Fig 5.7

Samples: PageRank

```
// URL neighbor URL
JavaRDD<String> lines =
spark.read().textFile(args[0]).javaRDD();
// Loads all URLs from input file and initialize their
neighbors.
JavaPairRDD<String, Iterable<String>> links =
lines.mapToPair(s -> {
      String[] parts = SPACES.split(s);
      return new Tuple2<>(parts[0], parts[1]);
    }).distinct().groupByKey().cache();

// Loads all URLs with other URL(s) link to from input
file and initialize ranks of them to one.
JavaPairRDD<String, Double> ranks = links.mapValues(rs
-> 1.0);
```

```java
// Calculates and updates URL ranks continuously using PageRank algorithm.
for (int current = 0; current < Integer.parseInt(args[1]); current++) {
  // Calculates URL contributions to the rank of other URLs.
  JavaPairRDD<String, Double> contribs = links.join(ranks).values()
        .flatMapToPair(s -> { // _1 = adj list, _2 = ranks
          int urlCount = Iterables.size(s._1());
          List<Tuple2<String, Double>> results = new ArrayList<>();
          for (String n : s._1) { // Send rank value to neighbor
            results.add(new Tuple2<>(n, s._2() / urlCount));
          }
          return results.iterator();
        });

  // Re-calculates URL ranks based on neighbor contributions.
  ranks = contribs.reduceByKey(new Sum()).mapValues(sum -> 0.15 + sum * 0.85);
}
 // Collects all URL ranks and dump them to console.
    List<Tuple2<String, Double>> output = ranks.collect();
    for (Tuple2<?,?> tuple : output) {
      System.out.println(tuple._1() + " has rank: " + tuple._2() + ".");
}
```

More on Spark