Chapter 4:

Apache Spark

Lecture Notes

Winter semester 2016 / 2017

Ludwig-Maximilians-University Munich

Based on lectures by

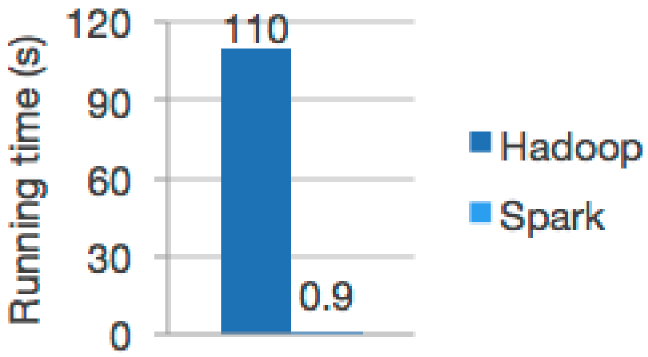
Donald Kossmann (ETH Zürich), as well as

Jure Leskovec, Anand Rajaraman, and Jeff Ullman (Stanford University)

Motivation

Spark becomes new standard for the MR applications:

- Logistic regression in Hadoop and Spark:

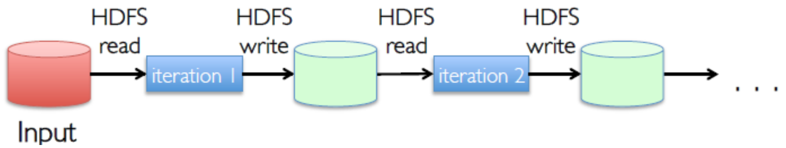


- Cloudera replaces classic MR framework with Spark
- IBM puts 3500 Researches to work on Spark related projects

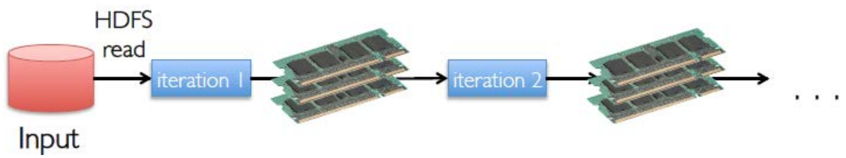
Motivation

Most of the Algorithms require a chain of MR steps:

- Tedious to program
- Writes to disk and reads from disk between steps are expensive



- Idea: Use memory instead of disk



Apache Spark

- Keeps data between operations in-memory
- Lot of convenience functions (e.g. filter, join)
- No restrictions for the operations order from the framework (not just Map->Reduce)
- Spark program is a pipeline of operations on distributed datasets (RDD)
- API: Java, Scala, Python, R

Resilient Distributed Dataset (RDD)

- Read-only collection of objects
- Partitioned across machines
- Enables operations on partitions in parallel
- Creation:
  - Parallelizing a collection
  - Data from files (e.g. HDFS)
  - As result of transformation of another RDD

```
In [25]: numbers=sc.parallelize([1,2,3,4,5,6,7,8,9,10])

In [26]: numbers
Out[26]: ParallelCollectionRDD[21] at parallelize at PythonRDD.scala:391
```

Resilient Distributed Dataset (RDD)

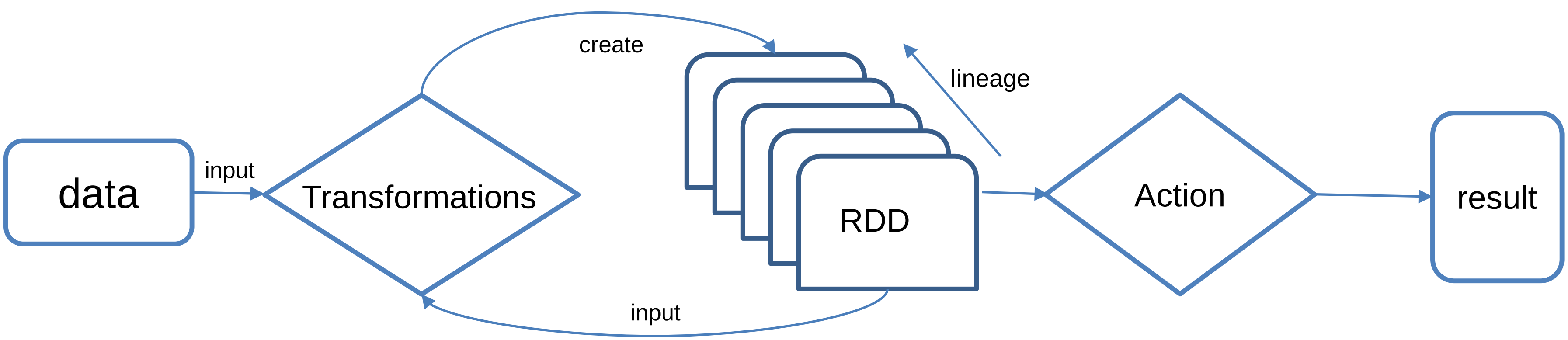
- Number of partitions determines parallelism level
- Can be cached in memory between operations
- Graph based representation (Lineage)

- Fault-Tolerant
  - In case of machine failure:
    RDD can be reconstructed

RDD Transformations and actions

Two types of operations:

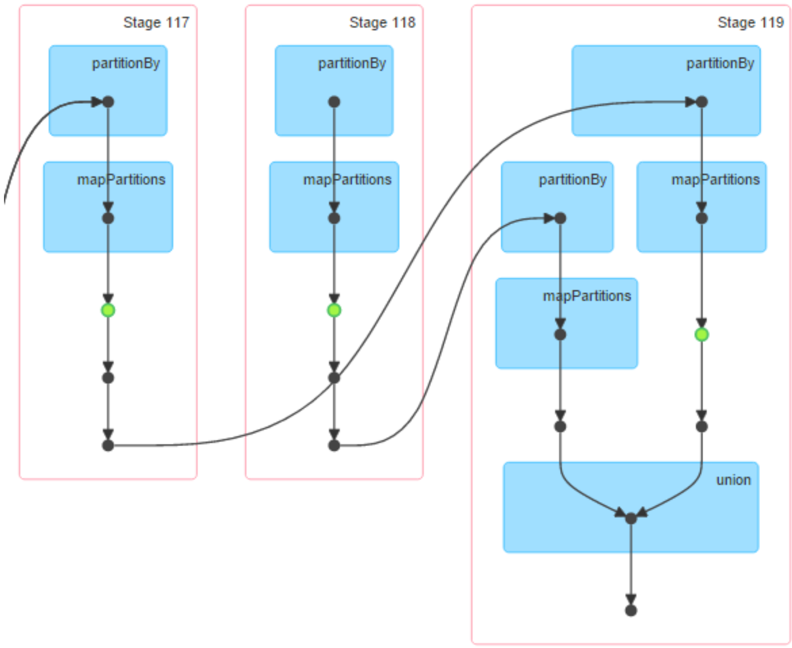
- Transformations (lazily evaluated)
- Actions (trigger transformations)



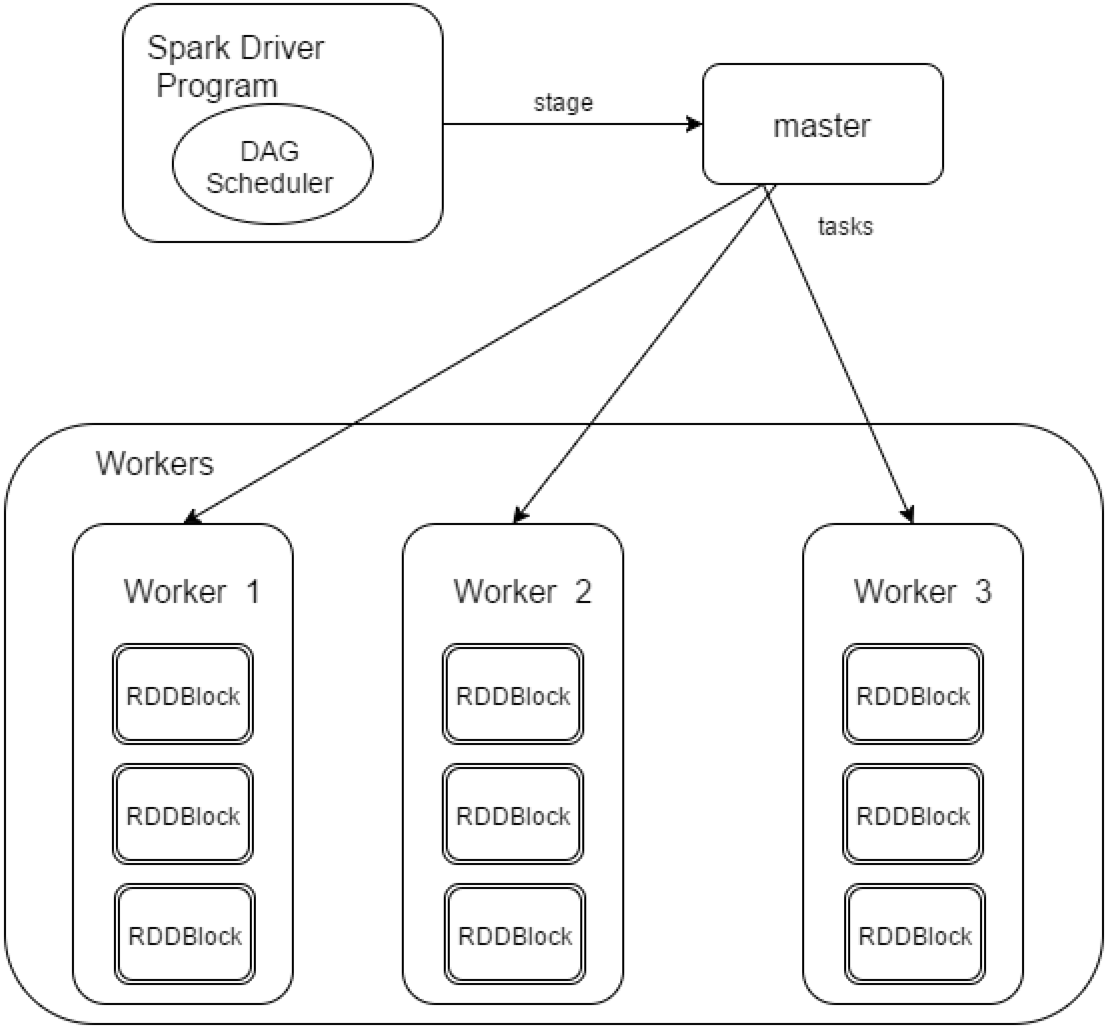
RDD Transformations and actions

Transformations

- Recipe how the new dataset is generated from the existing one
- Lazy evaluated
- Organized as Directed Acyclic Graph
- The required calculations are optimized
- DAG Scheduler defines stages for execution
- Each stage comprises tasks based on particular data partitions



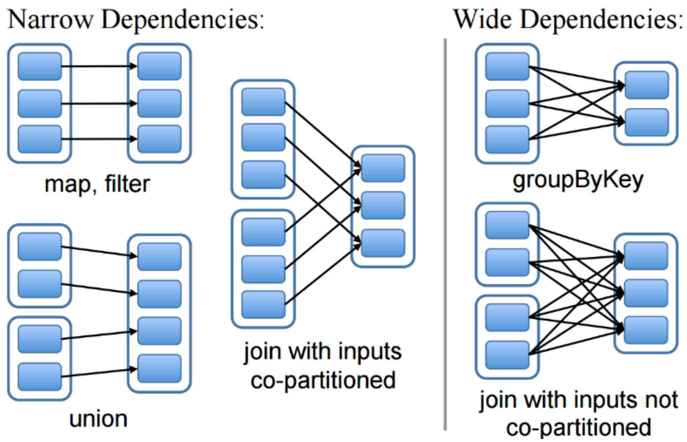
Architecture



RDD narrow and wide dependencies

- Narrow dependency
  - Each partition of the new RDD depends on partitions located on the same worker (transformation is executed locally on the workers)

- Wide dependency
  - New partition depends on partitions on several workers (shuffle necessary)



Shuffle

- internal map and reduce tasks
  to organize and aggregate data

- large costs
  - in memory data structures consume a lot of memory
    => disk I/O (shuffle spill) + garbage collection
  - many intermediate files on disk (for RDD reconstruction in case of failure) => garbage collection
  - data serialization
  - network I/O

- reduce the amount of data to be transferred in the shuffle phase by pre-aggregation

Lettercount examples

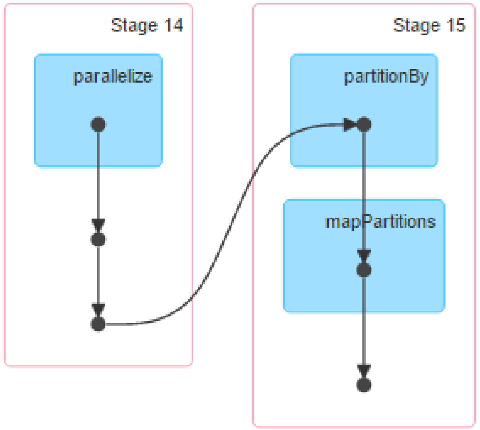

```
In [17]: data = sc.parallelize(list('dfasdfasdfasdfasdgafsgasfgasfgafgasfdgafgafafga'))

In [18]: data.map(lambda letter: (letter, 1))\
   ....: .reduceByKey(lambda f_count, s_count: f_count + s_count)\
   ....: .collect()
Out[18]: [('g', 8), ('f', 12), ('a', 13), ('s', 8), ('d', 6)]
```

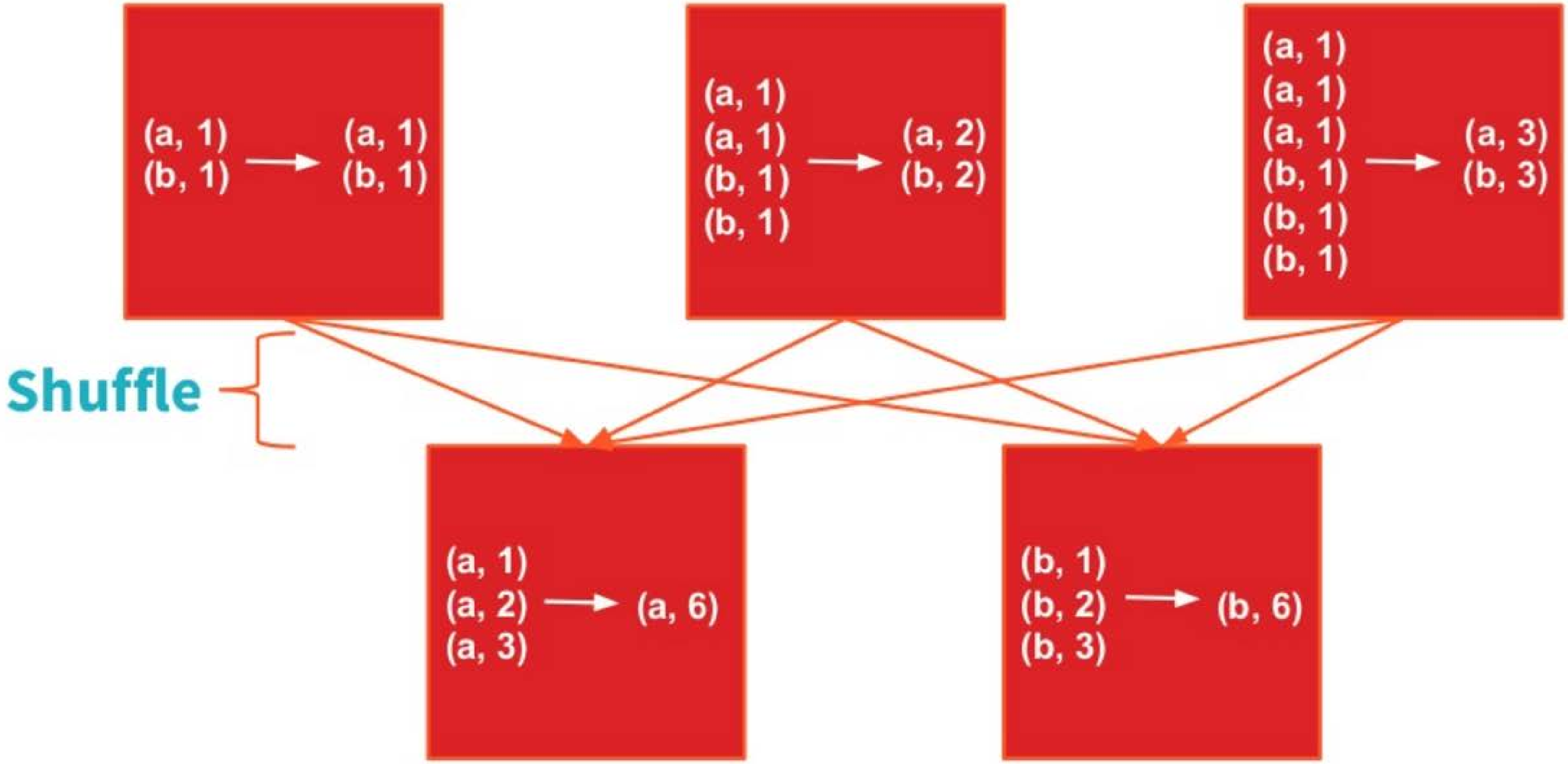
actions

transformations

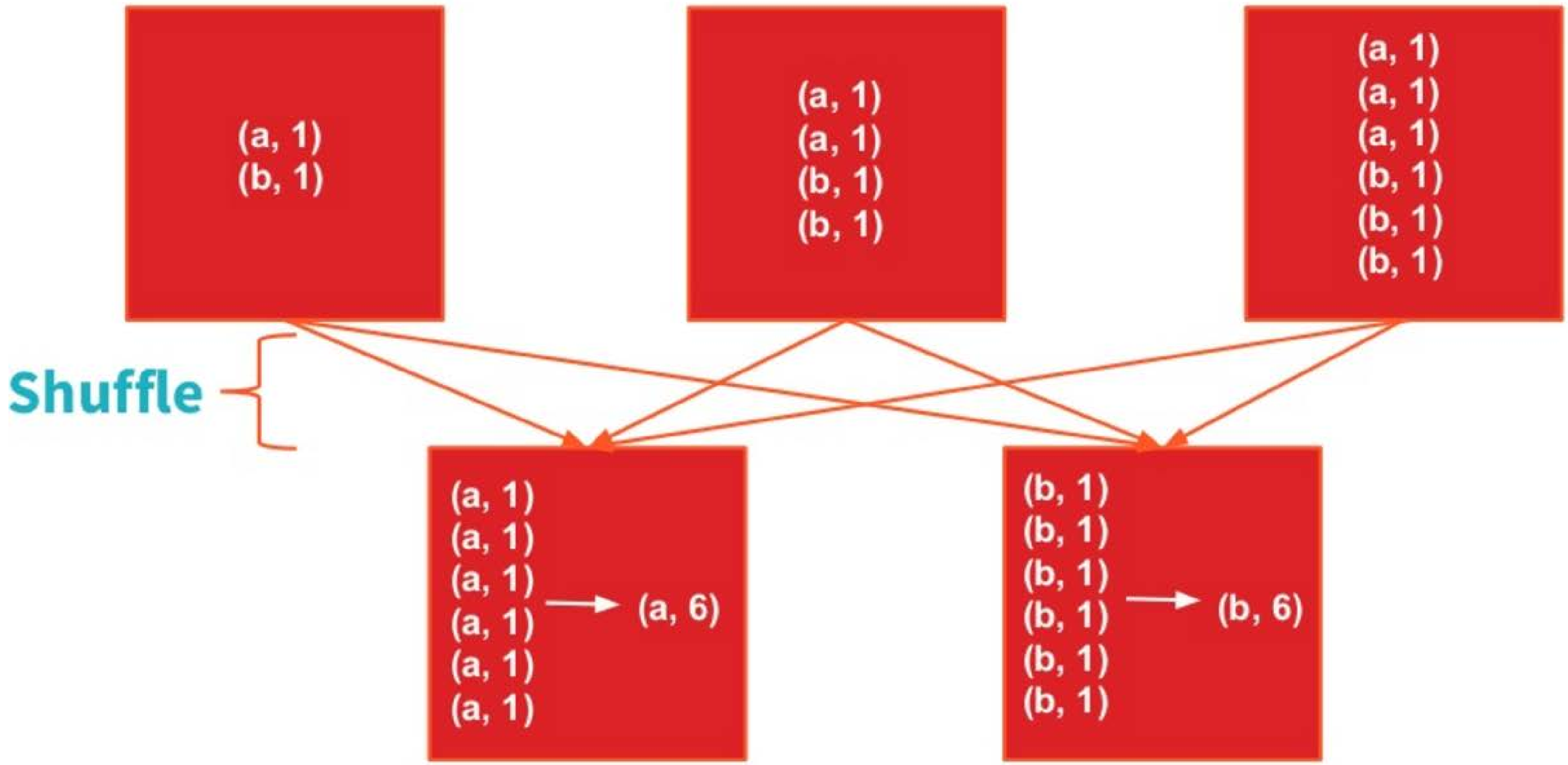
```
In [20]: data.map(lambda letter: (letter, 1))\
.groupByKey()\
.mapValues(lambda count_list:sum(count_list))\
.collect()
Out[20]: [('g', 8), ('f', 12), ('a', 13), ('s', 8), ('d', 6)]
```



Shuffle reduceByKey

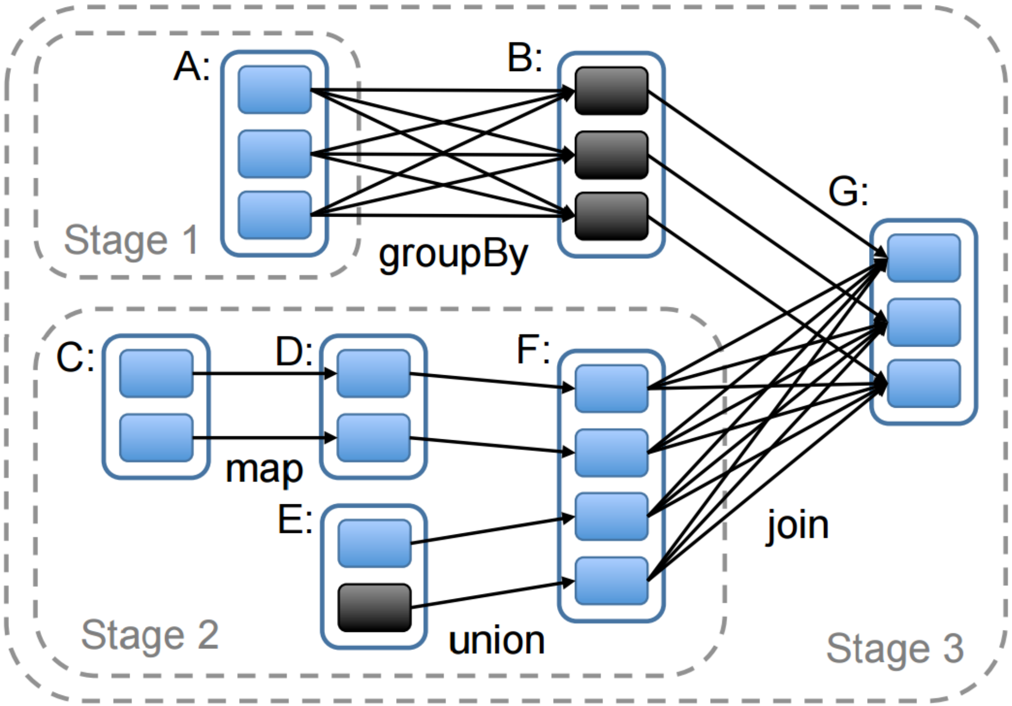


Shuffle groupByKey



RDD Persistence

Precomputed RDDs are reused



B was computed and is reused, stage 1 is skipped

RDD Persistence

- Computed RDD are held in memory as deserialized Java objects
- Old data partitions are dropped in least-recently-used fashion to free memory. Discarded RDD is recomputed if it is needed again.
- To advise Spark to keep RDD in memory call cache() or persist() operations on it

RDD Persistence

- RDD can be persisted differently by passing argument to persist function (in python persisted objects are always serialized):
  - As deserialized java objects in memory (default)
  - As deserialized java objects in memory and on disk
  - Serialized java objects in memory
  - Serialized java objects in memory and on disk
  - Serialized on disk
  - Off Heap

RDD Persistence

- Off heap RDD persistence:
  - RDDs are persisted outside of Java Heap
  - Reduces the JVM Garbage Collection pauses
- Tachyon
  - Memory-centric distributed storage system
  - Lineage function
  - Enables data sharing between different jobs
  - Data is safe even if computation crashes

Shared variables

- The driver program passes the functions to the cluster

- If passed function uses variables defined in driver program, these are copied to each worker

```
In [201]: a = 3

In [202]: numbers = sc.parallelize([1,2,3,4])

In [203]: numbers.map(lambda n: n + a).collect()
Out[203]: [4, 5, 6, 7]
```

- Updates on these variables are not allowed

```
In [195]: a=[3]

In [196]: numbers=sc.parallelize([1,2,3,4])

In [197]: numbers.foreach(lambda n: a.append(n))

In [198]: a
Out[198]: [3]
```

Shared variables

- The necessary common data is broadcasted within each stage
- Within the stage the data is serialized and is desirialized before each task
- Broadcast variables are used to avoid multiple broadcasting and de/serialization
- Broadcast variable is shipped once and is cached deserialized
- Broadcast variable should not be modified, but can be recreated

Example broadcast variable:

```
In [219]: dict = {'dog' : 'hund', 'he' : 'er', 'weather' : 'wetter', 'is' : 'ist', 'good' : 'gut'}

In [220]: broadcasted_dict = sc.broadcast(dict)

In [221]: data = sc.parallelize( ['weather', 'is', 'good'] )

In [222]: data.map(lambda word: broadcasted_dict.value[word]).collect()
Out[222]: ['wetter', 'ist', 'gut']

In [223]: dict['good']='sehr gut'

In [224]: data.map(lambda word: broadcasted_dict.value[word]).collect()
Out[224]: ['wetter', 'ist', 'gut']

In [225]: broadcasted_dict = sc.broadcast(dict)

In [226]: data.map(lambda word: broadcasted_dict.value[word]).collect()
Out[226]: ['wetter', 'ist', 'sehr gut']
```

Shared variables

- Accumulators are only updatable shared variables in Spark
- Associative add operation on accumulator is allowed
- Own add operation for new types are allowed
- Tasks can update Accumulator value, but only driver program can read it
- Accumulator update is applied when action is executed
- Task updates accumulator each time action is called
- Restarted tasks update accumulator only once

Accumulator example:

```
In [257]: accum = sc.accumulator(0)

In [258]: data = sc.parallelize([1,2,3,4])

In [259]: def add_to_acc(acc, to_add ):
     acc.add(to_add)
     return to_add
   .....:

In [260]: res = data.map(lambda n: add_to_acc(accum,n))

In [261]: accum.value
Out[261]: 0

In [262]: res.collect()
Out[262]: [1, 2, 3, 4]

In [263]: accum.value
Out[263]: 10

In [264]: res.count()
Out[264]: 4

In [265]: accum.value
Out[265]: 20
```

Other relevant spark projects

- Spark streaming
  - Objects from stream are processed in small groups (batches)
  - Similar to batch processing
- Spark SQL
  - Processing of structured data (SchemaRDD)
  - Data is stored in columns and is analyzed in SQL manner
  - Data is still RDD and can be processed by other Spark frameworks
  - JDBC/ODBC interface

Other relevant spark projects

- GraphX
  - Distributed computations on Graphs

- Machine Learning Libraries
  - Mlib
  - H20 (Sparkling water)
  - Keystone ML

Sources

- http://www.datacenterknowledge.com/archives/2015/09/09/cloudera-aims-to-replace-mapreduce-with-spark-as-default-hadoop-framework/
- http://spark.apache.org/images/logistic-regression.png
- https://www-03.ibm.com/press/us/en/pressrelease/47107.wss
- Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.
- http://de.slideshare.net/databricks/strata-sj-everyday-im-shuffling-tips-for-writing-better-spark-programs?related=1