COMP9313: Big Data Management

MapReduce

Data Structure in MapReduce

- Key-value pairs are the basic data structure in MapReduce
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be arbitrary data structures

- The design of MapReduce algorithms involves:
  - Imposing the key-value structure on arbitrary datasets
    - E.g., for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - In some algorithms, input keys are not used (e.g., wordcount), in others they uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms

Recall of Map and Reduce

- Map
  - Reads data (split in Hadoop, RDD in Spark)
  - Produces key-value pairs as intermediate outputs

- Reduce
  - Receive key-value pairs from multiple map jobs
  - aggregates the intermediate data tuples to the final output

MapReduce in Hadoop

- Data stored in HDFS (organized as blocks)
- Hadoop MapReduce Divides input into fixed-size pieces, input splits
  - Hadoop creates one map task for each split
  - Map task runs the user-defined map function for each record in the split
  - Size of a split is normally the size of a HDFS block
- Data locality optimization
  - Run the map task on a node where the input data resides in HDFS
  - This is the reason why the split size is the same as the block size
    - The largest size of the input that can be guaranteed to be stored on a single node
    - If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks

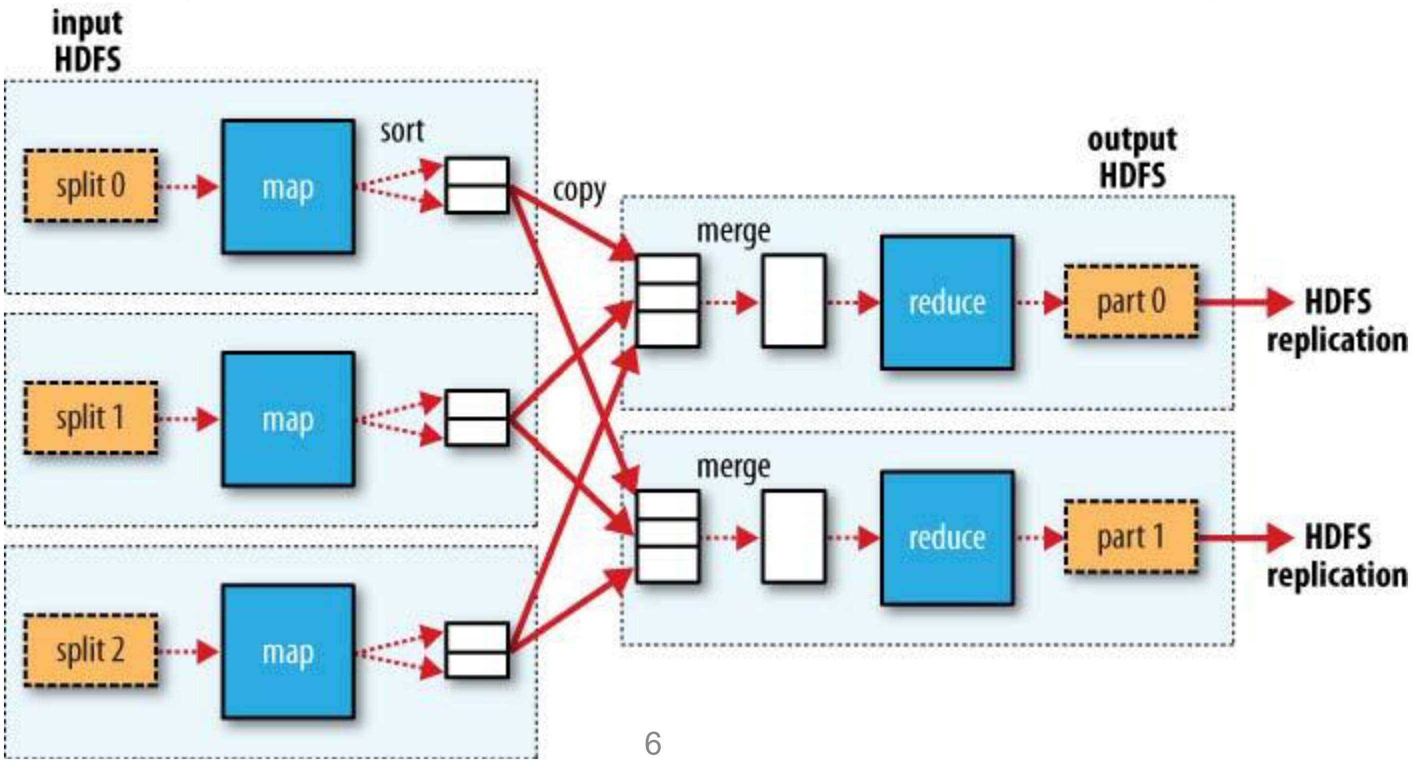
MapReduce in Hadoop

- Map tasks write their output to local disk (not to HDFS)
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - Storing it in HDFS with replication, would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on another node

- Reduce tasks don't have the advantage of data locality
  - Input to a single reduce task is normally the output from all mappers
  - Output of the reduce is stored in HDFS for reliability
  - The number of reduce tasks is not governed by the size of the input, but is specified independently

More Detailed MapReduce Dataflow

- When there are multiple reducers, the map tasks partition their output:
  - One partition for each reduce task
  - The records for every key are all in a single partition
  - Partitioning can be controlled by a user-defined partitioning function



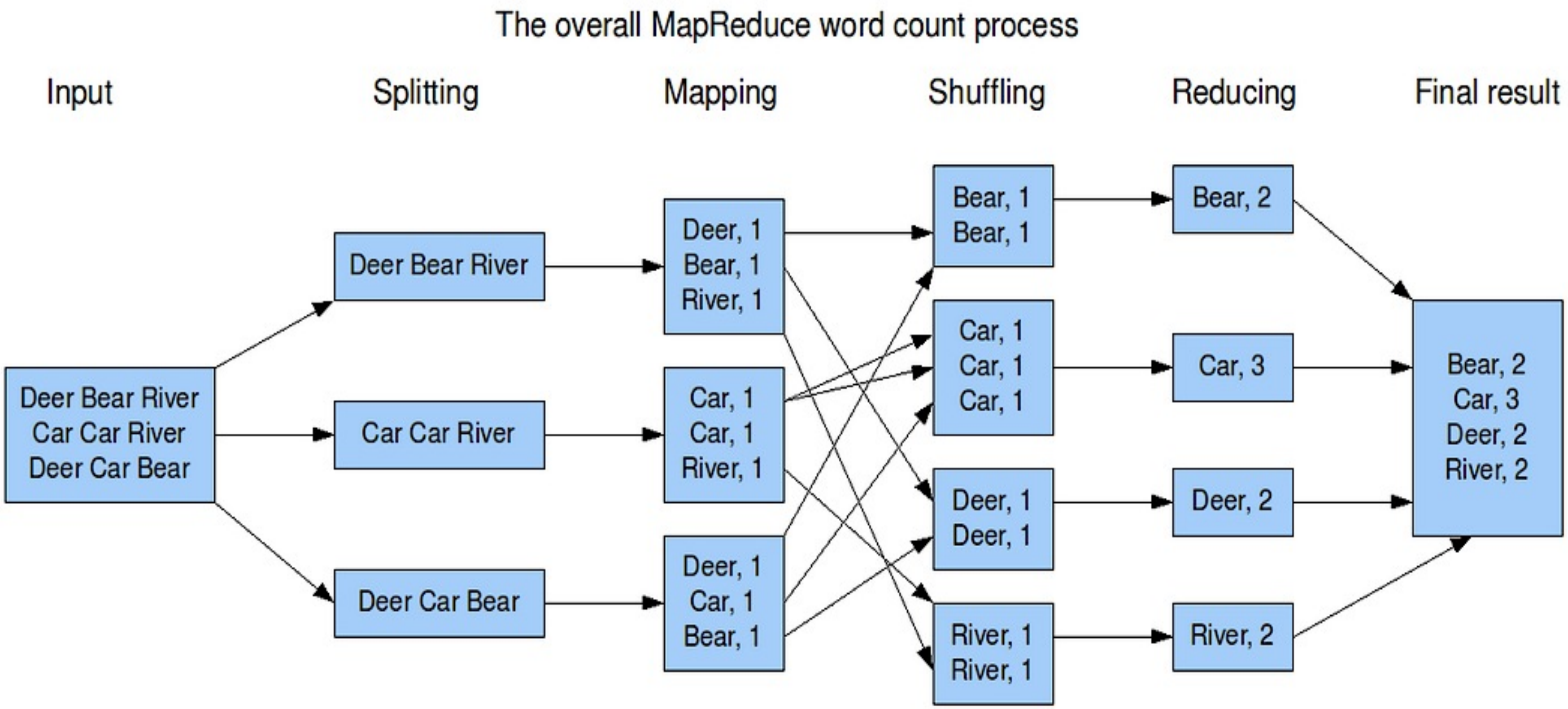
Shuffle

- Shuffling is the process of data redistribution
  - To make sure each reducer obtains all values associated with the same key.
  - It is needed for all of the operations which require grouping
    - E.g., word count, compute avg. score for each department, …
- Spark and Hadoop have different approaches implemented for handling the shuffles.

Shuffle in Hadoop (handled by framework)

- Happens between each Map and Reduce phase
- Use Shuffle and Sort mechanism
  - Results of each Mapper are sorted by the key
  - Starts as soon as each mapper finishes
- Use combiner to reduce the amount of data shuffled
  - Combiner combines key-value pairs with the same key in each par
  - This is not handled by framework!

Example of MapReduce in Hadoop



Shuffle in Spark (handled by Spark)

- Triggered by some operations
  - Distinct, join, repartition, all *By, *ByKey
  - I.e., Happens between stages
- Hash shuffle
- Sort shuffle
- Tungsten shuffle-sort
  - More on https://issues.apache.org/jira/browse/SPARK-7081

Hash Shuffle

- Data are hash partitioned on the map side
  - Hashing is much faster than sorting
- Files created to store the partitioned data portion
  - # of mappers X # of reducers
- Use consolidateFiles to reduce the # of files
  - From M * R => E*C/T * R
- Pros:
  - Fast
  - No memory overhead of sorting
- Cons:
  - Large amount of output files (when # partition is big)

Sort Shuffle

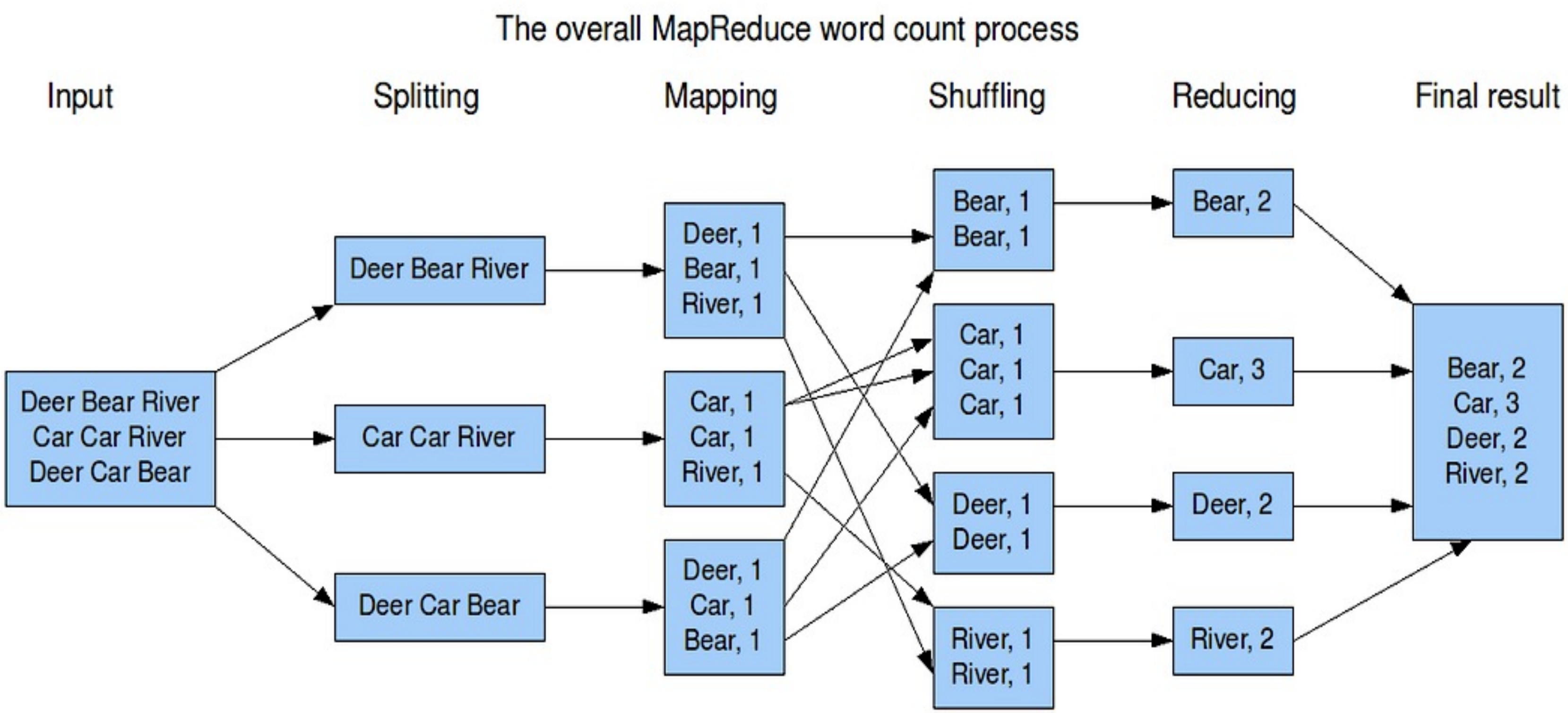
- For each mapper 2 files are created
  - Ordered (by key) data
  - Index of beginning and ending of each 'chunk'
- Merged on the fly while being read by reducers
- Default way
  - Fallback to hash shuffle if # partitions is small
- Pros
  - Smaller amount of files created
- Cons
  - Sorting is slower than hashing

MapReduce Functions in Spark (Recall)

- Transformation
  - Narrow transformation
  - Wide transformation
- Action

- The job is a list of Transformations followed by one Action
  - Only action will trigger the 'real' execution
    - I.e., lazy evaluation

Transformation = Map? Action = Reduce?



combineByKey

- RDD([K, V]) to RDD([K, C])
  - K: key, V: value, C: combined type
- Three parameters (functions)
  - createCombiner
    - What is done to a single row when it is FIRST met?
    - V => C
  - mergeValue
    - What is done to a single row when it meets a previously reduced row?
    - C, V => C
    - In a partition
  - mergeCombiners
    - What is done to two previously reduced rows?
    - C, C => C
    - Across partitions

Example: word count

- createCombiner
  - What is done to a single row when it is FIRST met?
  - V => C
  - lambda v: v
- mergeValue
  - What is done to a single row when it meets a previously reduced row?
  - C, V => C
  - lambda c, v: c+v
- mergeCombiners
  - What is done to two previously reduced rows?
  - C, C => C
  - lambda c1, c2: c1+c2

Example 2: Compute Max by Keys

- createCombiner
  - What is done to a single row when it is FIRST met?
  - V => C
  - lambda v: v
- mergeValue
  - What is done to a single row when it meets a previously reduced row?
  - C, V => C
  - lambda c, v: max(c, v)
- mergeCombiners
  - What is done to two previously reduced rows?
  - C, C => C
  - lambda c1, c2: max(c1, c2)

Example 3: Compute Sum and Count

- createCombiner
  - V => C
  - lambda v: (v, 1)
- mergeValue
  - C, V => C
  - lambda c, v: (c[0] + v, c[1] + 1)
- mergeCombiners
  - C, C => C
  - lambda c1, c2: (c1[0] + c2[0], c1[1] + c2[1])

Example 3: Compute Sum and Count

- data = [('A', 2.), ('A', 4.), ('A', 9.), ('B', 10.), ('B', 20.), ('Z', 3.), ('Z', 5.), ('Z', 8.)]
  - Partition 1: ('A', 2.), ('A', 4.), ('A', 9.), ('B', 10.)
  - Partition 2: ('B', 20.), ('Z', 3.), ('Z', 5.), ('Z', 8.)
- Partition 1 ('A', 2.), ('A', 4.), ('A', 9.), ('B', 10.)
  - A=2. --> createCombiner(2.) ==> accumulator[A] = (2., 1)
  - A=4. --> mergeValue(accumulator[A], 4.) ==> accumulator[A] = (2. + 4., 1 + 1) = (6., 2)
  - A=9. --> mergeValue(accumulator[A], 9.) ==> accumulator[A] = (6. + 9., 2 + 1) = (15., 3)
  - B=10. --> createCombiner(10.) ==> accumulator[B] = (10., 1)
- Partition 2 ('B', 20.), ('Z', 3.), ('Z', 5.), ('Z', 8.), ('Z', 12.)
  - B=20. --> createCombiner(20.) ==> accumulator[B] = (20., 1)
  - Z=3. --> createCombiner(3.) ==> accumulator[Z] = (3., 1)
  - Z=5. --> mergeValue(accumulator[Z], 5.) ==> accumulator[Z] = (3. + 5., 1 + 1) = (8., 2)
  - Z=8. --> mergeValue(accumulator[Z], 8.) ==> accumulator[Z] = (8. + 8., 2 + 1) = (16., 3)
- Merge partitions together
  - A ==> (15., 3)
  - B ==> mergeCombiner((10., 1), (20., 1)) ==> (10. + 20., 1 + 1) = (30., 2)
  - Z ==> (16., 3)
- Collect
  - ([A, (15., 3)], [B, (30., 2)], [Z, (16., 3)])

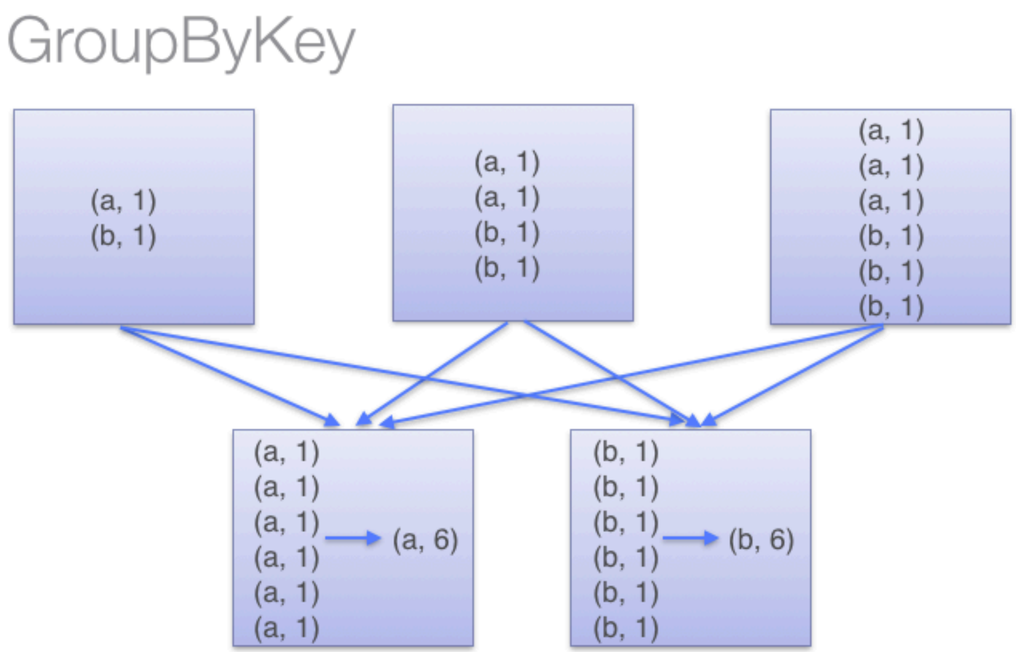
reduceByKey

- reduceByKey(func)
  - Merge the values for each key using func
  - E.g., reduceByKey(lambda x, y: x + y)

- createCombiner
  - lambda v: v
- mergeValue
  - func
- mergeCombiners
  - func

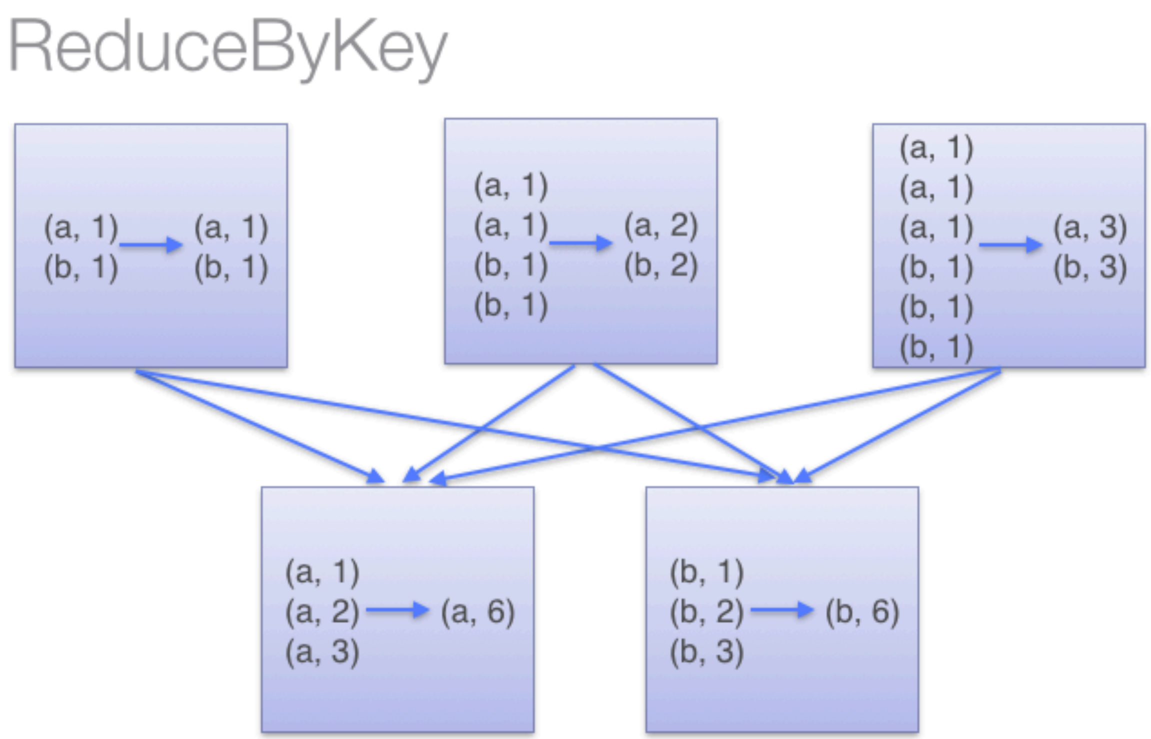
groupByKey

- groupByKey()
  - Group the values for each key in the RDD into a single sequence.
  - Data shuffle according to the key value in another RDD



reduceByKey

- Combines before shuffling
- Avoid using groupByKey



The Efficiency of MapReduce in Spark

- Number of transformations
  - Each transformation involves a linearly scan of the dataset (RDD)
- Size of transformations
  - Smaller input size => less cost on linearly scan
- Shuffles
  - data transferring between partitions is costly
    - especially in a cluster!
      - Disk I/O
      - Data serialization and deserialization
      - Network I/O

Number of Transformations (and Shuffles)

rdd = sc.parallelize(data)

- data: (id, score) pairs

- Bad design

  maxByKey = rdd.combineByKey(…)

  sumByKey = rdd.combineByKey(…)

  sumMaxRdd = maxByKey.join(sumByKey)

- Good design

  sumMaxRdd = rdd.combineByKey(…)

Size of Transformations

rdd = sc.parallelize(data)

- data: (word, 1) pairs

- Bad design

  countRdd = rdd.reduceByKey(…)

  fileteredRdd = countRdd.filter(…)

- Good design

  fileteredRdd = countRdd.filter(…)

  countRdd = fileteredRdd.reduceByKey(…)

Partition

rdd = sc.parallelize(data)

- data: (word, 1) pairs

- Bad design

  countRdd = rdd.reduceByKey(…)

  countBy2ndCharRdd = countRdd.map(…).reduceByKey(…)

- Good design

  paritionedRdd = data.partitionBy(…)

  countBy2ndCharRdd = paritionedRdd.map(…).reduceByKey(…)