universidade de aveiro 

Mining Large Scale Datasets

Distributed File Systems and MapReduce

(Adapted from CS246@Starford.edu; http://www.mmds.org)

Sérgio Matos - aleixomatos@ua.pt

Motivation

- Data mining = extraction of actionable information from (usually) very large datasets
  - Descriptive methods
    Find human-interpretable patterns that describe the data.
    Example: Clustering
  - Predictive methods
    Use some variables to predict unknown or future values of other variables. Example: Recommender systems
- This course: emphasis on **algorithms that scale**
  - Algorithms
  - Computing architectures
  - Automation for handling large data

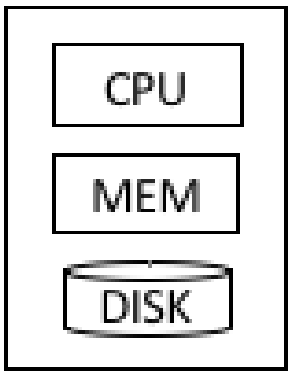
Motivation

- Different types of data
  - High dimensional
  - Graphs
  - Streams
  - Labeled data
- Different models of computation
  - MapReduce
  - Streams and online algorithms
  - Single machine in-memory

Motivation

- Massive data
- Data transfer (bandwidth) is biggest limitation
  - 1 hour just to read 1TB from SSD (10x more from HDD)
  - And we still haven't done anything with the data...
- MapReduce
  - Hide complexity from distributed programming
  - Move computation close to the data
  - Replicate data for reliability

Motivation



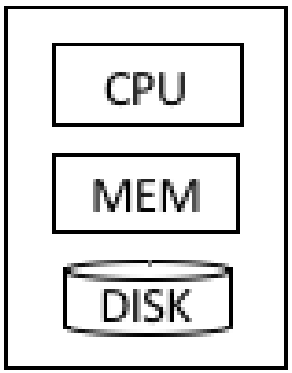
Classical machine learning / data mining

- Entire data processed in memory

  or

- Partial data read to memory
- Partial results written to disk

Motivation



Classical machine learning / data mining

- Entire data processed in memory

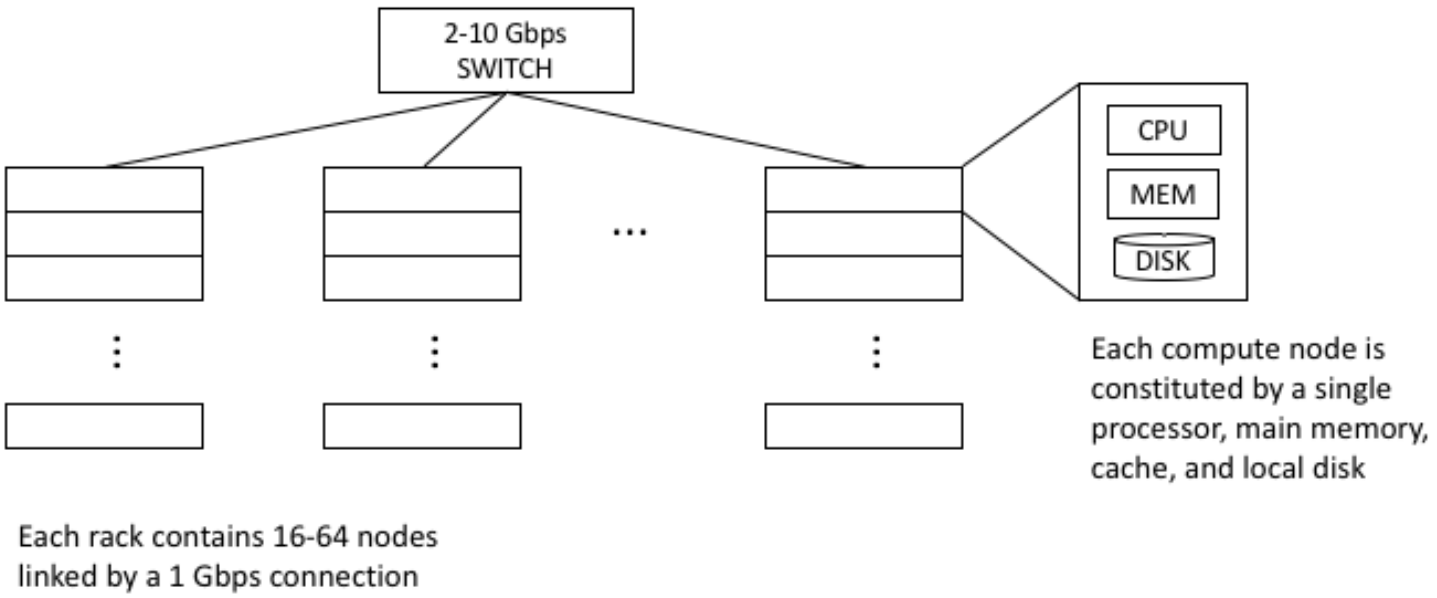
  or

- Partial data read to memory
- Partial results written to disk

What if data so big that the computation takes too long?

- Split data
- Use multiple processors

Cluster Computing



Compute nodes are commodity hardware
$\rightarrow$reduced cost compared to special-purpose parallel machines

Cluster computing

Large-scale computing for **data mining** problems on **commodity hardware**

- Challenges:
  - How to distribute computation?
  - How to make it easy to write distributed programs?
  - Machines fail:
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to loose 1/day
    - With 1M machines 1,000 machines fail every day!

Network bottleneck

- Issue

  Copying data over a network takes time

- Idea

  Bring computation to data

  Store files multiple times for reliability

- Spark/Hadoop address these problems
  - Storage Infrastructure: Distributed File system
    Google GFS
    Hadoop HDFS
  - Programming model
    MapReduce
    Spark

Storage infrastructure

- Problem

  If nodes fail, how to store data persistently?

- Answer - **Distributed File System**
  - Provides global file namespace, redundancy, availability

- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

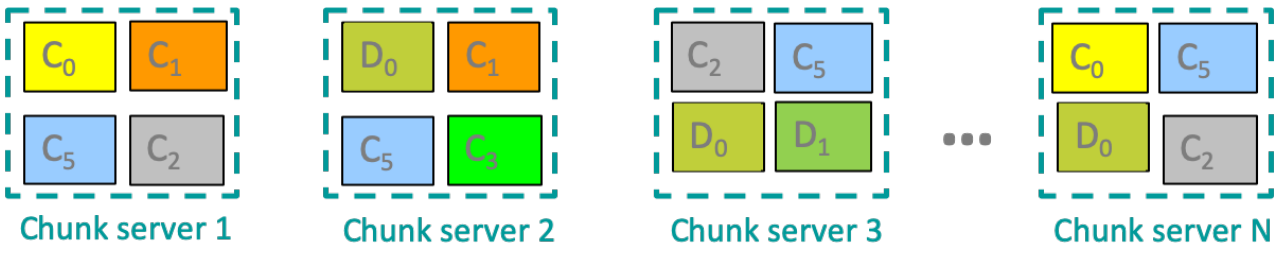
Distributed File System

- Chunk servers
  - File is split into contiguous chunks (16-64MB tipically)
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- Master Node / Name Node
  - Stores metadata about where files are stored
  - May also be replicated
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

Distributed file system

- Data kept in "chunks" spread across machines
- Each chunk replicated on different machines

  ↪ Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

Programming Model

- MapReduce is a style of programming designed for:
  - Easy parallel programming
  - Invisible management of hardware and software failures
  - Easy management of very-large-scale data
- Several implementations
  - original Google implementation just called “MapReduce”
  - Hadoop, Spark, Flink

3 steps of MapReduce

- **Map**
  - Apply a user-written Map function to each input element
  - Output of the Map function is a set of 0, 1, or more key-value pairs
- **Group by key**
  - System sorts all the key-value pairs by key, and outputs key-(list of values) pairs
- **Reduce**
  - User-written Reduce function is applied to each key-(list of values)

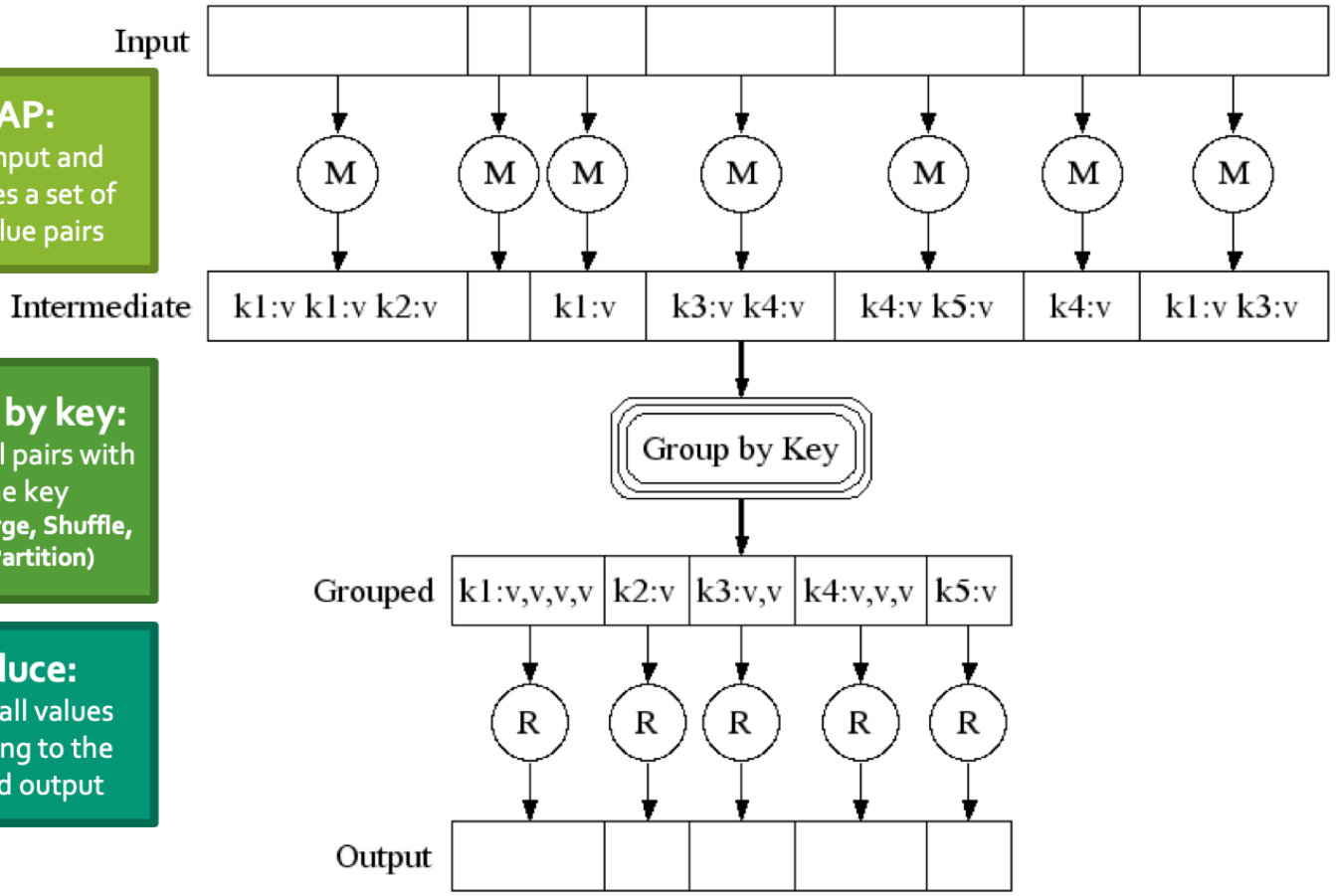
Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce

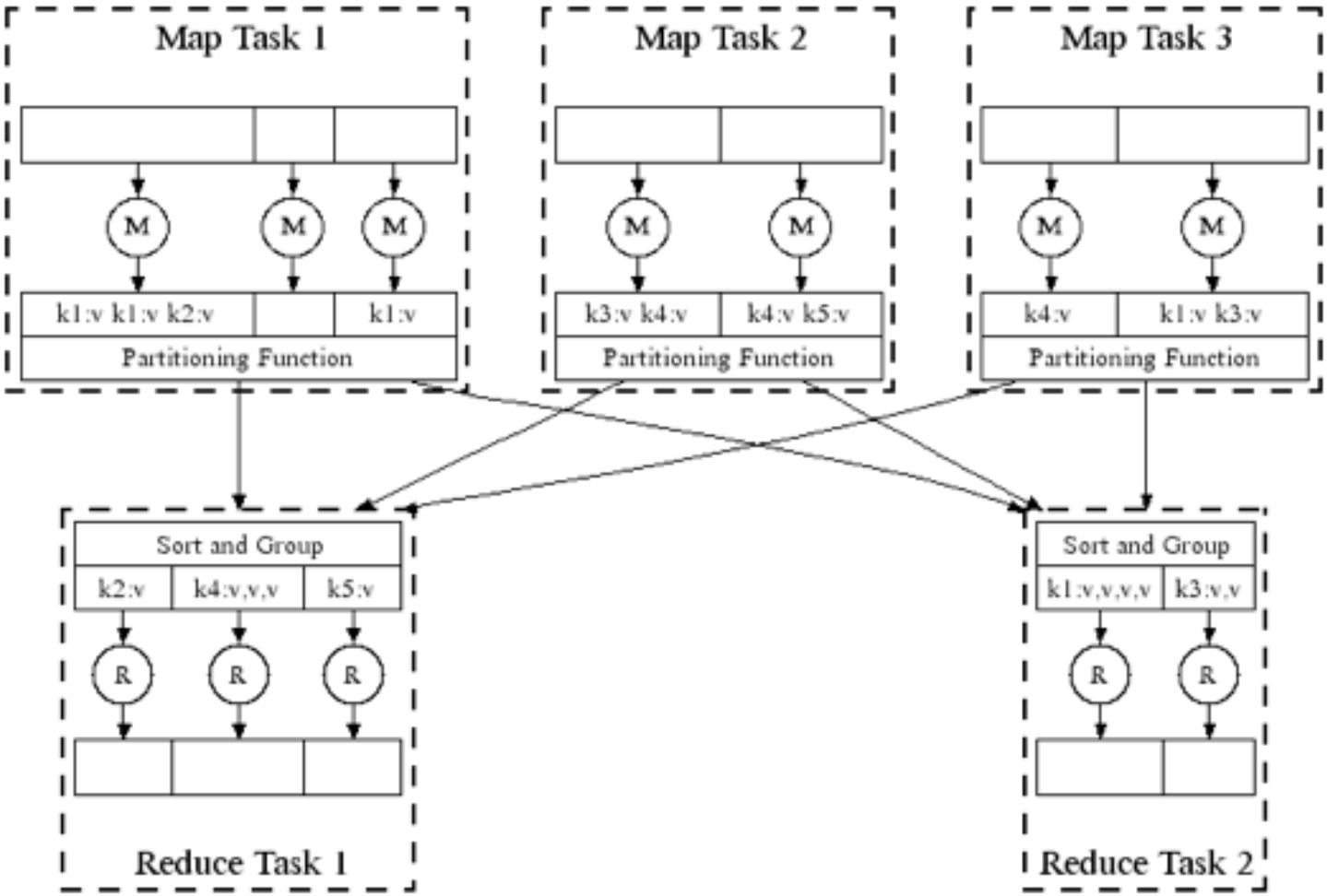
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



MapReduce in Parallel



MapReduce

MapReduce environment takes care of:

- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the group by key step
- Handling machine failures
- Managing required inter-machine communication

MapReduce Example

Application: count occurrences of distinct words in a collection of documents

- Input is a collection of documents
- The Map function uses keys of type String (words) and values of type integer
- Each Map task reads a document, breaks it into its sequence of words w1, w2, . . . , wn, and emits the sequence of key-value pairs: (w1,1), (w2,1),...,(wn,1)
- The Reduce function adds up all the values for the same key (word).

MapReduce Example

```
map(key, value):
    // key: document name; value: text of the document
    for each word w in value:
        emit(w, 1)

reduce(key, values):
    // key: a word; values: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(key, result)
```

MapReduce Example

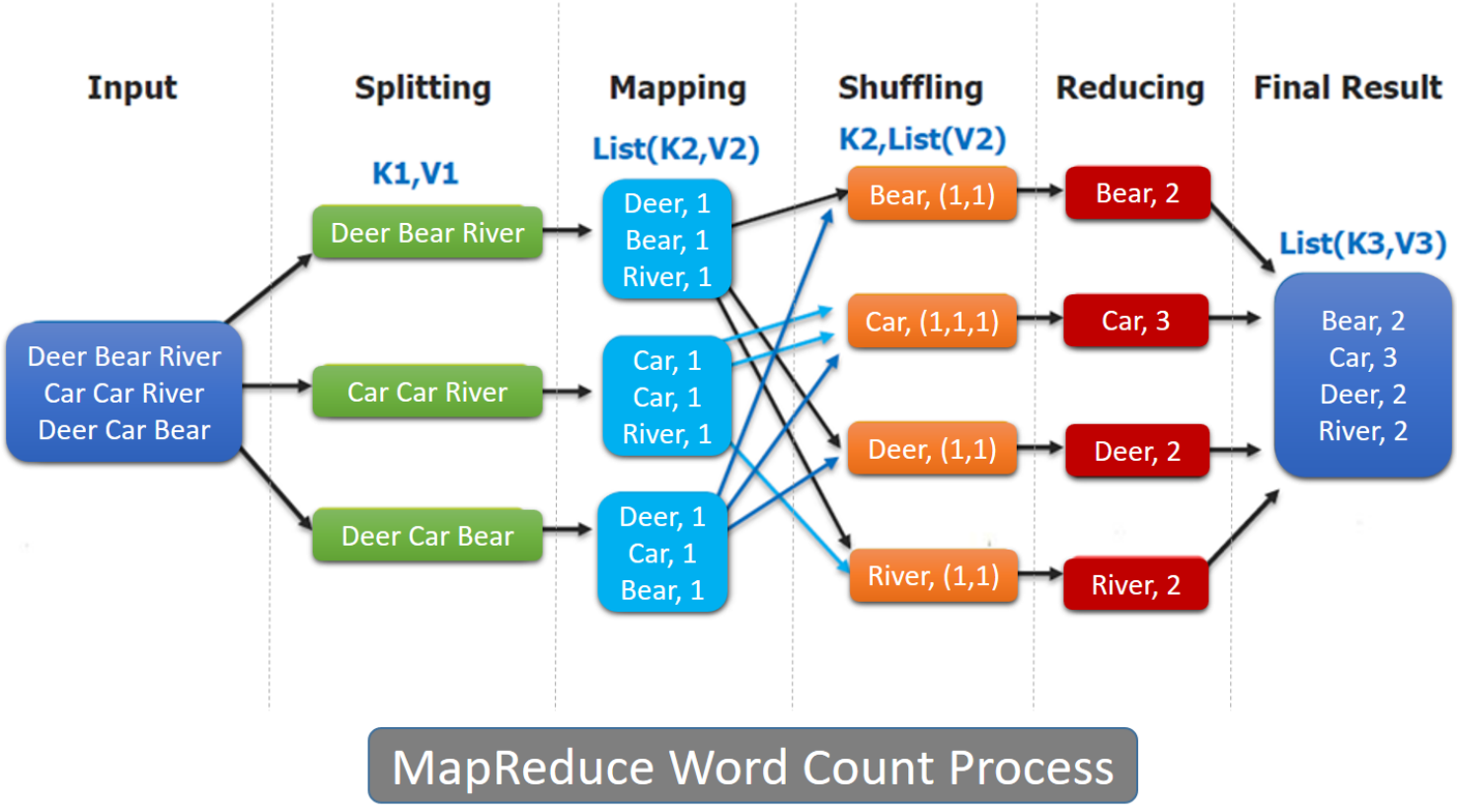


Image from: R.M.R. Pattamsetti, Distributed Computing in Java 9, Packt Publishing

Data flow

- Input and final output are stored on a distributed file system
- Scheduler tries to schedule map tasks “close” to physical storage location of input data
- Intermediate results are stored on local file system of Map and Reduce workers
- Output is often input to another MapReduce task

Master

- Master node takes care of coordination
- Task status: (idle, in-progress, completed)
- Idle tasks get scheduled as workers become available
- When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
- Master pushes this info to reducers
- Master pings workers periodically to detect failures

Handling failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- Master failure
  - MapReduce task is aborted and client is notified

Number of Map and Reduce jobs

- M map tasks, R reduce tasks
- Rule of a thumb:

  Make M much larger than the number of nodes in the cluster
- One DFS chunk per map is common

  Improves dynamic load balancing and speeds up recovery from worker failures
- Usually R is smaller than M

  Because output is spread across R files

Refinement: Combiners

- Often a Map task will produce many pairs of the form (k,v1), (k,v2), ... for the same key k

  E.g., popular words in the word count example

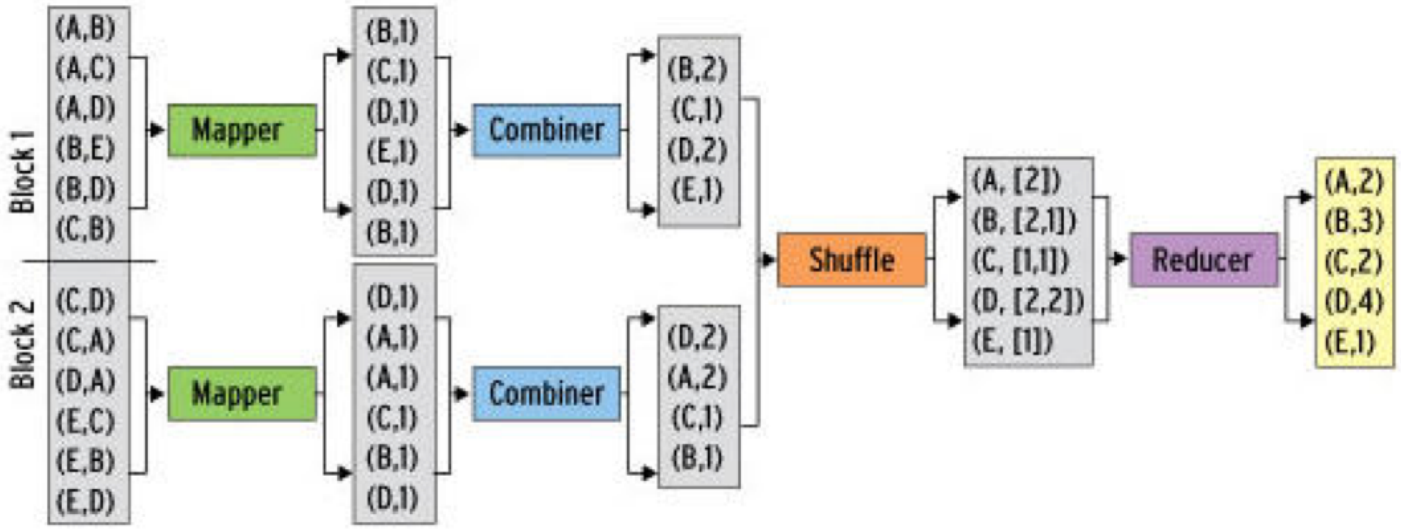
- Can save network time by pre-aggregating values in mapper

  combine(k, list(v1))

- Combiner is usually same as the reduce function

- Works only if reduce function is commutative and associative

Refinement: Combiners



Problems with MapReduce

Two major limitations of MapReduce

- Difficultly of programming directly in MR

  Many problems aren't easily described as map-reduce
- Performance bottlenecks, or batch not fitting the use cases

  Persistence to disk typically slower than in-memory work

MR doesn't compose well for large applications
Many times one needs to chain multiple map-reduce steps

Data-Flow Systems

- MapReduce uses two "ranks" of tasks
  - One for Map the second for Reduce
  - Data flows from the first rank to the second

- Data-Flow Systems generalize this in two ways
  - Allow any number of tasks/ranks
  - Allow functions other than Map and Reduce

$\hookrightarrow$ As long as data flow is in one direction only, the blocking property still applies and allows recovery of tasks rather than whole jobs

Spark

Expressive computing system, not limited to the map-reduce model

Additions to MapReduce model

- Fast data sharing
  - Avoids saving intermediate results to disk
  - Caches data for repetitive queries (e.g. for machine learning)
- General execution graphs (DAGs)
- Richer functions than just Map and Reduce

Most Popular Data-Flow System
Compatible with Hadoop

Spark

- Open source software (Apache Foundation)

  Supports Java, Scala and Python
- Key construct/idea: Resilient Distributed Dataset (RDD)
- Higher-level APIs: DataFrames, DataSets

  Introduced in more recent versions of Spark

  Different APIs for aggregate data, which allowed to introduce SQL support

Spark: RDD

- Resilient Distributed Dataset (RDD)
  - Partitioned collection of records
  - Generalizes (key-value) pairs
  - Spread across the cluster, Read-only
  - Caching dataset in memory
  - Different storage levels available
  - Fallback to disk possible

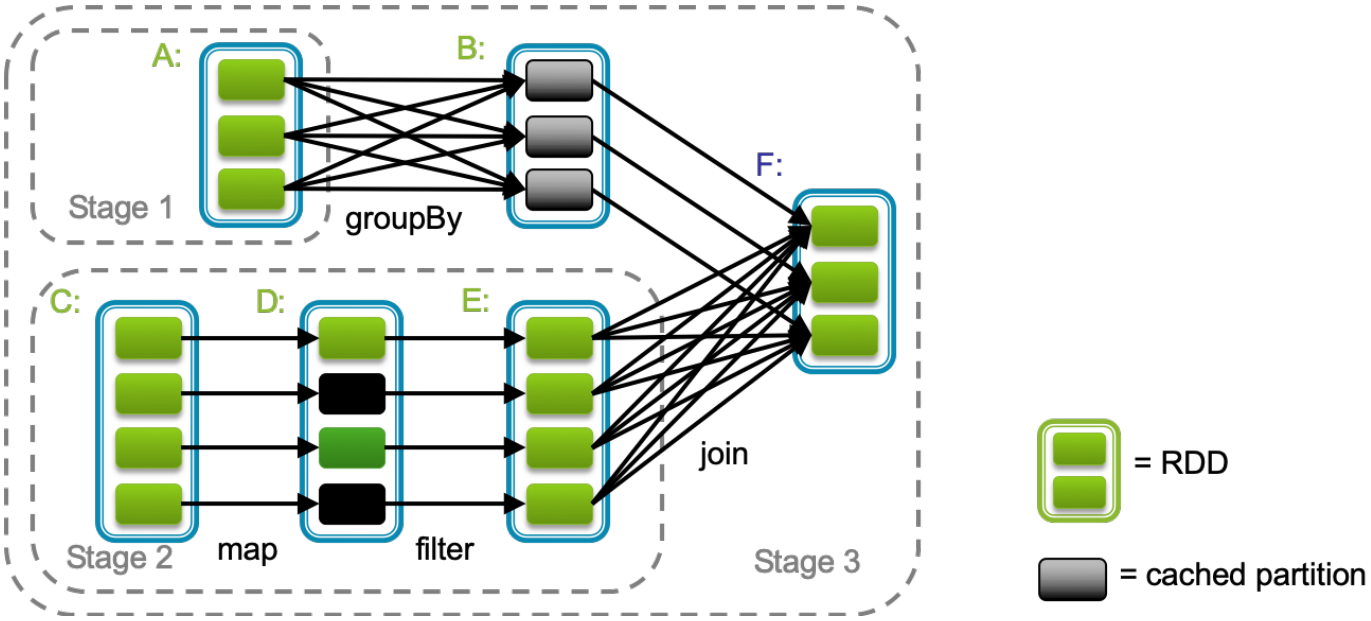
$\hookrightarrow$ RDDs can be created from Hadoop, or by transforming other RDDs (RDDs can be stacked)

$\hookrightarrow$ RDDs are best suited for applications that apply the same operation to all elements of a dataset

Spark RDD Operations

- Transformations build RDDs through deterministic operations on other RDDs
  - Include map, filter, join, union, intersection, distinct
  - Lazy evaluation: Nothing computed until an action requires it
- Actions to return value or export data
  - Include count, collect, reduce, save
  - Actions can be applied to RDDs; actions force calculations and return values

Spark Task Scheduler



Supports general task graphs
Pipelines functions where possible
Cache-aware data reuse locality
Partitioning-aware to avoid shuffles

Spark RDD

- Useful RDD Actions
  - take(n) – return the first n elements in the RDD as an array
  - collect() – return all elements of the RDD as an array. Use with caution
  - count() – return the number of elements in the RDD as an int
  - saveAsTextFile('path/to/dir') – save the RDD to files in a directory. Will create the directory if it does not exist and will fail if it does
  - foreach(func) – execute the function against every element in the RDD, but doe not keep any results.

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

map()

Apply an operation to every element of an RDD and return a new RDD that contains the results

```
>>> data = sc.textFile('path/to/file')
>>> data.take(3)
['Apple,Amy', 'Butter,Bob', 'Cheese,Chucky']

>>> data.map(lambda line: line.split(',')).take(3)
[['Apple', 'Amy'], ['Butter', 'Bob'], ['Cheese', 'Chucky']]
```

flatMap()

Apply an operation to every element of an RDD and then flattens the result

```
>>> data = sc.textFile('path/to/file')
>>> data.take(3)
['Apple,Amy', 'Butter,Bob', 'Cheese,Chucky']

>>> data.flatMap(lambda line: line.split(',')).take(6)
['Apple','Amy', 'Butter', 'Bob', 'Cheese', 'Chucky']
```

mapValues()

Apply an operation to the value of every element of a pair RDD, without changing the key
(only works with pair RDDs)

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[('Apple', 'Amy'), ('Butter', 'Bob'), ('Cheese', 'Chucky')]

>>> data.mapValues(lambda name: name.lower()).take(3)
[('Apple', 'amy'), ('Butter', 'bob'), ('Cheese', 'chucky')]
```

flatMapValues()

Iterates over each value of a pair RDD, and for each element produce a key/value entry with the old key.

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[('Apple', 'Amy'), ('Butter', 'Bob'), ('Cheese', 'Chucky')]

>>> data.flatMapValues(lambda name: name.lower()).take(3)
[('Apple', 'a'), ('Apple', 'm'), ('Apple', 'y')]
```

filter()

Return a new RDD that contains only the elements that pass a filter operation

```
>>> import re
>>> data = sc.textFile('path/to/file')
>>> data.take(3)
['Apple,Amy', 'Butter,Bob', 'Cheese,Chucky']

>>> data.filter(lambda line: re.match(r'^[AEIOU]', line)).take(3
['Apple,Amy', 'Egg,Edward', 'Oxtail,Oscar']
```

groupByKey()

Combine elements of an RDD by key and return the result in a new RDD.

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[('Apple', 'Amy'), ('Butter', 'Bob'), ('Cheese', 'Chucky')]

>>> data.groupByKey().take(1)
[('Apple', <pyspark.resultiterable.ResultIterable object>)]

>>> pair = data.groupByKey().take(1)
>>> print ''%s:%s'' % (pair[0], '',''.join([n for n in pair[1]]))
Apple:Amy,Adam,Alex
```

reduceByKey()

Combine elements of an RDD by key and then apply a reduce operation to pairs of keys until only a single key remains. Return the result in a new RDD.

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[('Apple', 'Amy'), ('Butter', 'Bob'), ('Cheese', 'Chucky')]

>>> data.reduceByKey(lambda v1, v2: v1 + ''':''' + v2).take(1)
[('Apple', 'Amy:Alex:Adam')]
```

sortBy()

Sort an RDD according to a sorting function and return the results in a new RDD.

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[('Apple', 'Amy'), ('Butter', 'Bob'), ('Cheese', 'Chucky')]

>>> data.sortBy(lambda pair: pair[1]).take(3)
[('Avocado', 'Adam'), ('Anchovie', 'Alex'), ('Apple', 'Amy')]
```

sortByKey()

Sort an RDD according to the natural ordering of the keys and return the results in a new RDD.

```
>>> data = sc.textFile('path/to/file')
>>> data = data.map(lambda line: line.split(','))
>>> data = data.map(lambda pair: (pair[0], pair[1]))
>>> data.take(3)
[('Apple', 'Amy'), ('Butter', 'Bob'), ('Cheese', 'Chucky')]

>>> data.sortByKey().take(3)
[('Apple', 'Amy'), ('Anchovie', 'Alex'), ('Avocado', 'Adam')]
```

subtract()

Return a new RDD that contains all the elements from the original RDD that do not appear in a target RDD.

```
>>> data1 = sc.textFile('path/to/file1')
>>> data1.take(3)
['Apple,Amy', 'Butter,Bob', 'Cheese,Chucky']

>>> data2 = sc.textFile('path/to/file2')
>>> data2.take(3)
['Wendy', 'McDonald,Ronald', 'Cheese,Chucky']

>>> data1.subtract(data2).take(3)
['Apple,Amy', 'Butter,Bob', 'Dinkel,Dieter']
```

join()

Return a new RDD that contains all the elements from the original RDD joined (inner join) with elements from the target RDD.

```
>>> data1 = sc.textFile('path/to/file1').map( ... ).map( ... )
>>> data1.take(3)
[('Apple', 'Amy'), ('Butter', 'Bob'), ('Cheese', 'Chucky')]

>>> data2 = sc.textFile('path/to/file2').map( ... ).map( ... )
>>> data2.take(3)
[('Doughboy', 'Pilsbury'), ('McDonald', 'Ronald'), ('Cheese', 'C

>>> data1.join(data2).collect()
[('Cheese', ('Chucky', 'Chucky'))]

>>> data1.fullOuterJoin(data2).take(2)
[('Apple',('Amy', None)), ('Cheese', ('Chucky', 'Chucky'))]
```

Spark wordcount

```
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://...")
```

Spark: DataFrame and DataSet

- DataFrame
  - Unlike an RDD, data organized into named columns
    e.g. a table in a relational database
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset
  - Extension of DataFrame API which provides type-safe, object-oriented programming interface (compile-time error detection)

Both built on Spark SQL engine
Both can be converted back to an RDD

Spark libraries

- Spark SQL
- Spark Streaming

  Stream processing of live datastreams
- MLlib

  Scalable machine learning
- GraphX

  Graph manipulation

  Extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge

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 perform 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 (higher-level APIs)
- Data processing: Spark is more general