CS 4440 A

Emerging Database Technologies

Lecture 14

03/13/24

So far: One query/update
One machine

Multiple query/updates
One machine

Transactions

One query/update
Multiple machines

Distributed query processing
Map-Reduce, Spark

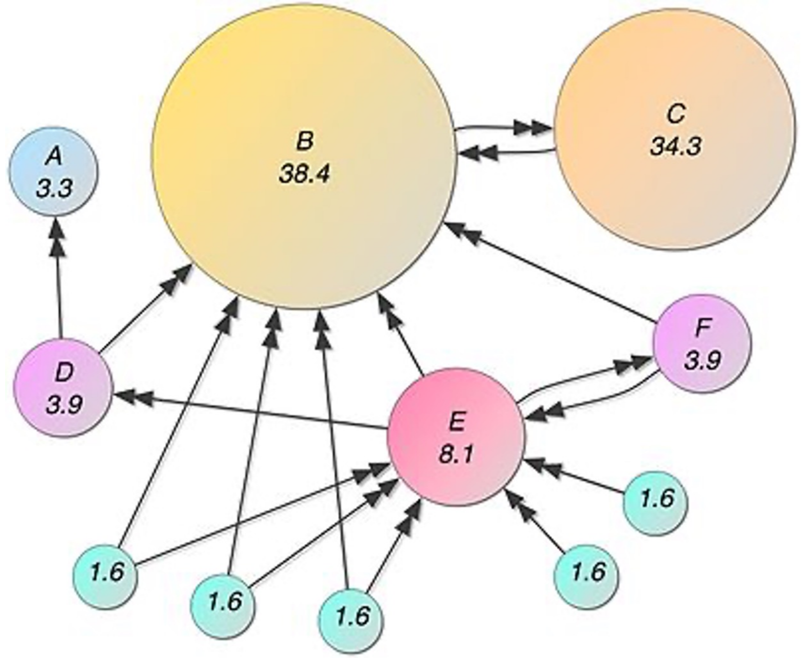
Historical Context

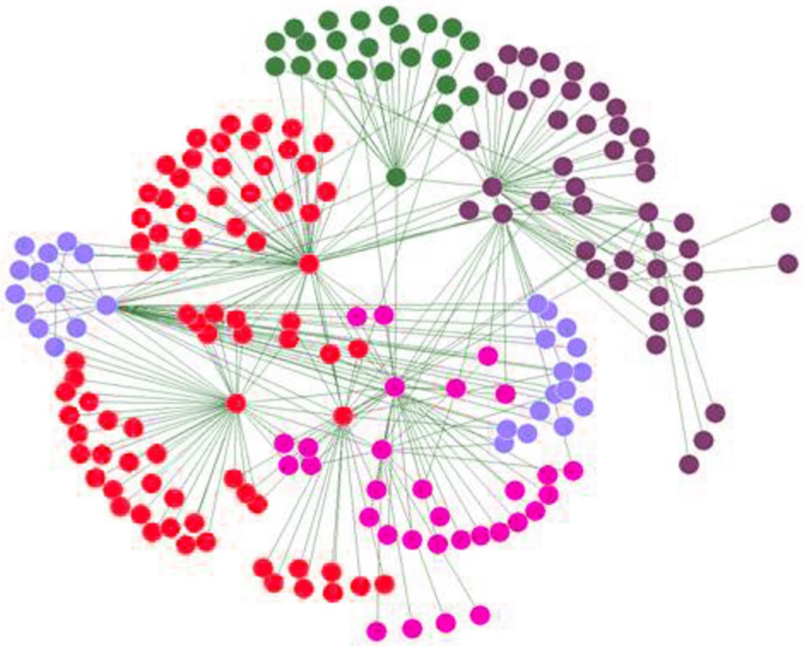
- Early 2000s, people wants to scale up systems
  - Non SQL or Non relational (nowadays, Not only SQL)
- Triggered by needs of Web 2.0 companies (e.g., Facebook, Amazon, Google)
- Trades off consistency requirements of RDBMS for speed

Image source: https://www.geeksforgeeks.org/what-is-internet-definition-uses-working-advantages-and-disadvantages/

Goal: managing large amounts of data quickly

- Ranking Web pages by importance
  - Iterated matrix-vector multiplication where dimension is many billions
- Search friends in social networks
  - Graphs with hundreds of millions of nodes and many billions of edges





Solution: horizontal scaling

- Instead of a supercomputer (aka vertical scaling), we have large collections of commodity hardware connected by Ethernet cables or inexpensive switches

Challenges

- How do you distribute computation?
- How can we 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 lose 1/day
    - With 1M machines, 1,000 machines fail every day!

A new software stack

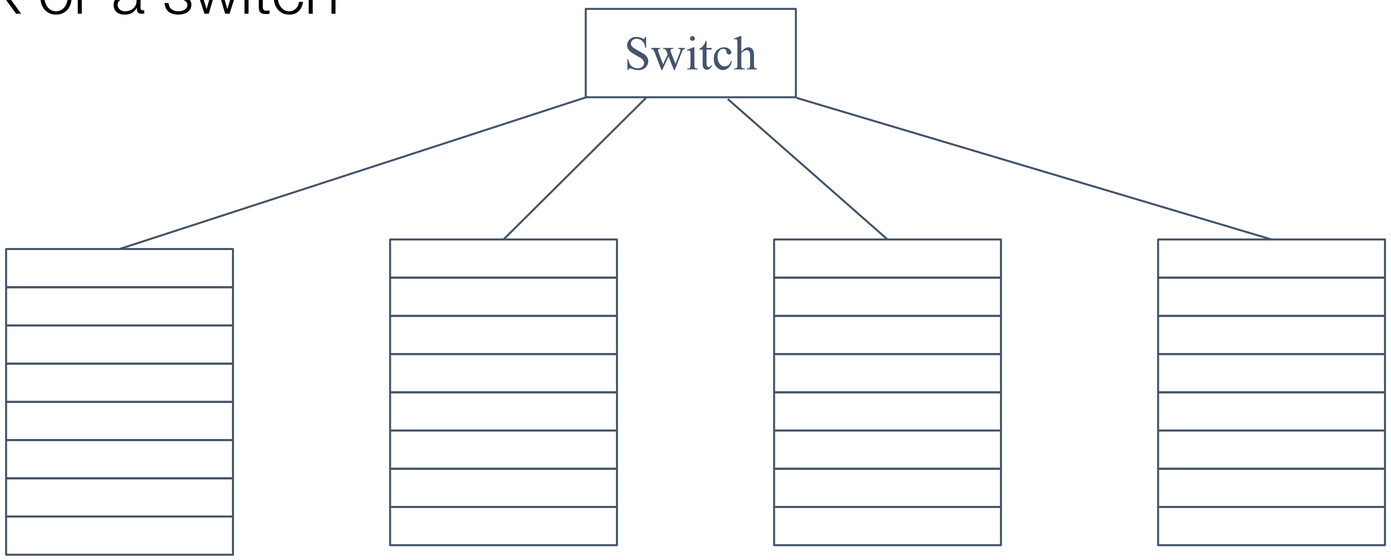
- Distributed file system
  - Large blocks, data replication, redundancy to protect against media failures

- MapReduce programming system
  - Enables common calculations on large-scale data to be performed on computing clusters efficiently
  - Tolerant to hardware failures
  - Extensions to acyclic workflows, recursive algorithms

Physical organization of compute nodes

Parallel-computing architecture

- Compute nodes are stored on racks (perhaps 8-64 on a rack)
- The nodes on a single rack are connected by a network, typically gigabit Ethernet
- There can be many racks of compute nodes connected by another level of network or a switch



Racks of compute nodes

Physical organization of compute nodes

It is a fact of life that components fail

- Loss of single node (e.g., disk crashes)
- Loss of an entire rack (e.g., network fails)

Solutions

- Files must be stored redundantly
- Computations must be divided into tasks, such that if any one task fails to execute to completion, it can be restarted without affecting other tasks

Large-scale file system organization

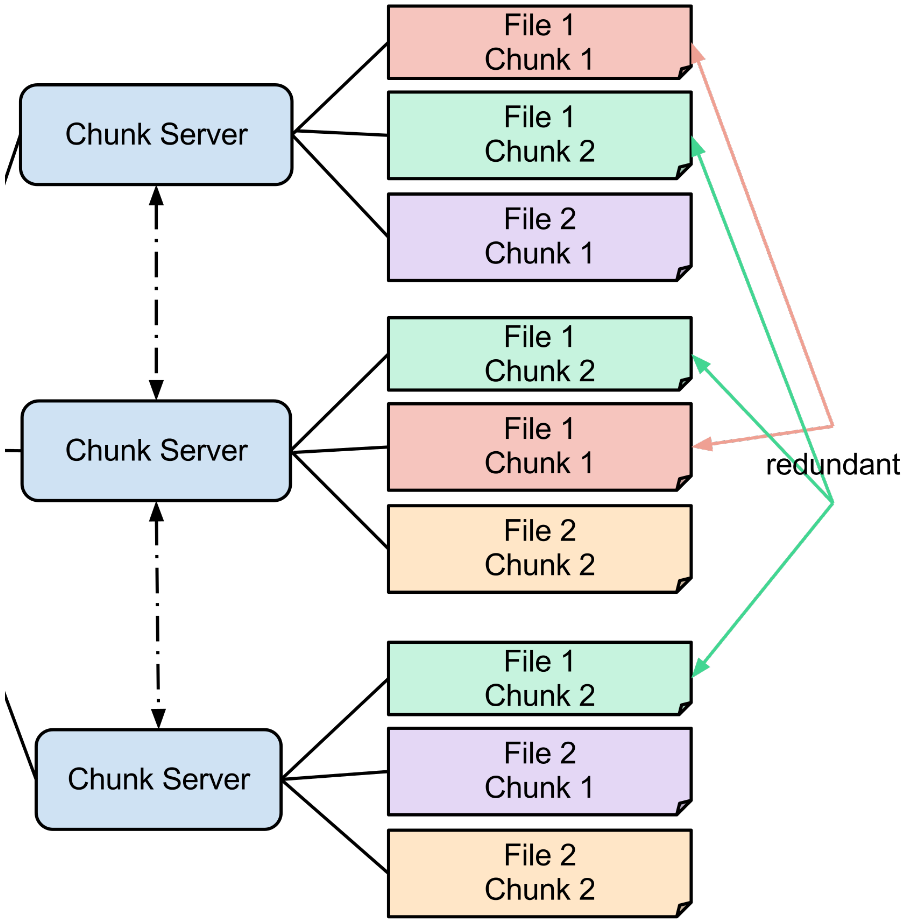
To exploit cluster computing, files must look and behave differently from conventional file systems on single computers

A Distributed File System (DFS) can be used when:

- For very large files: TBs, PBs
- Files are rarely updated and usually read or appended with data
- Mostly sequential reads

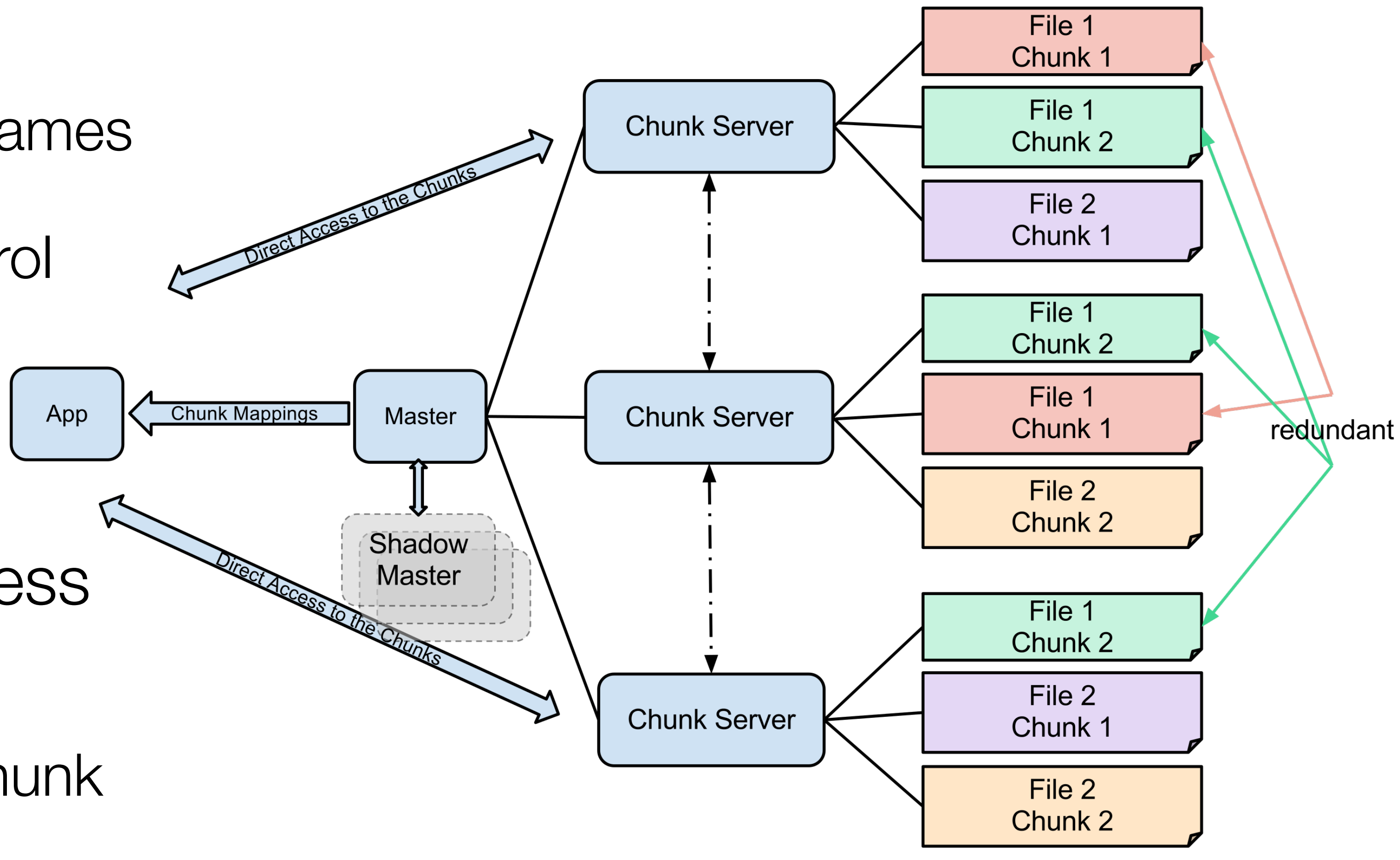
Distributed File System (DFS)

- Files are divided into chunks, which are typically 64 MBs
    - Chunks are replicated (say 3 times) at different compute nodes (called chunks ervers)
    - The compute nodes should be located on different racks
    - Chunk size and degree of replication decided by the user



Distributed File System (DFS)

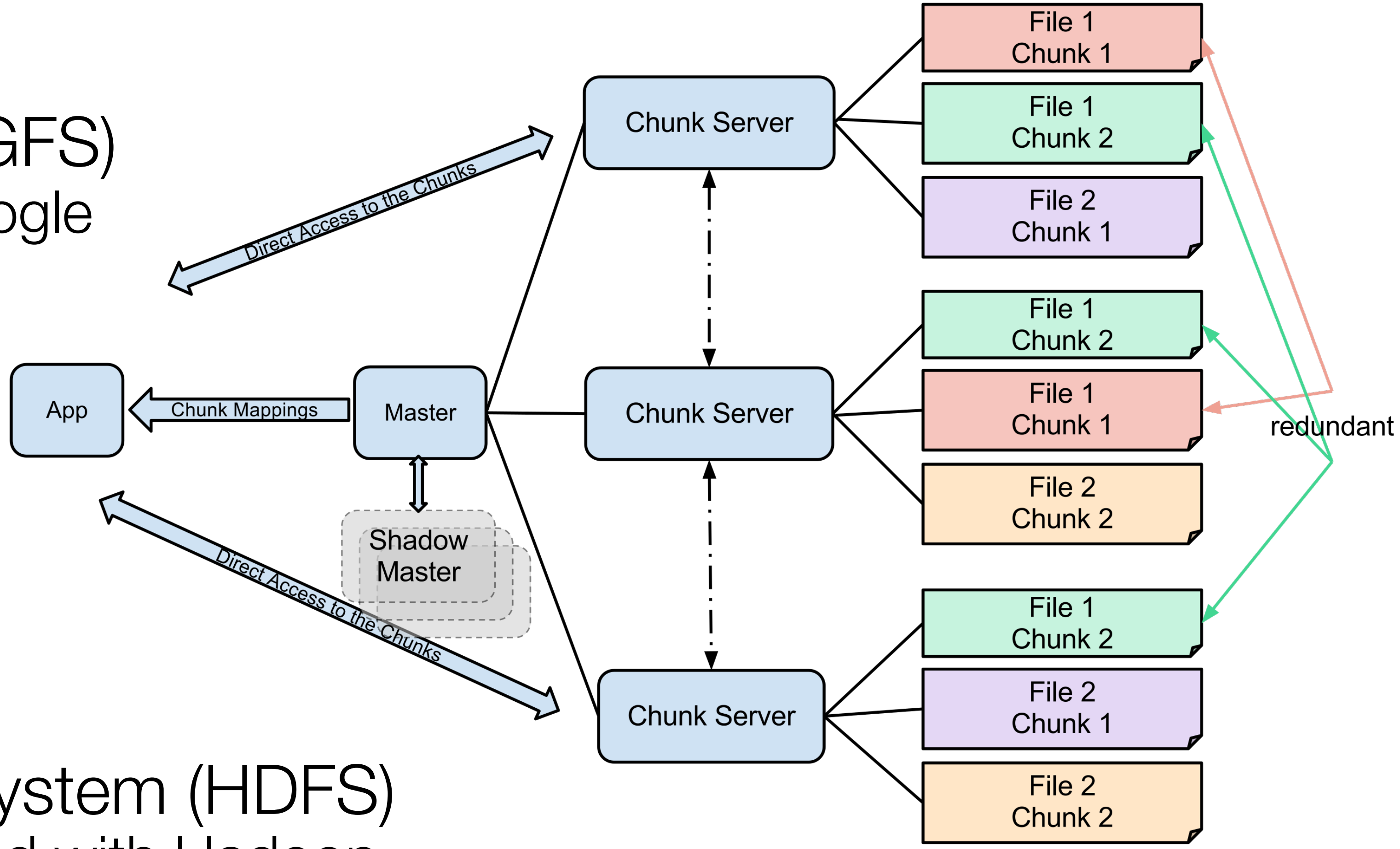
- Master node
  - Stores metadata: file names + chunk ids + chunk locations, access control
  - Master node itself is replicated
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data



DFS implementations

The Google File System (GFS)

- Previously used in Google
- Proprietary



Hadoop Distributed File System (HDFS)

- Open-source DFS used with Hadoop

Image source: https://en.wikipedia.org/wiki/Google_File_System#/media/File:GoogleFileSystemGFS.svg

A brief history of MapReduce and Hadoop

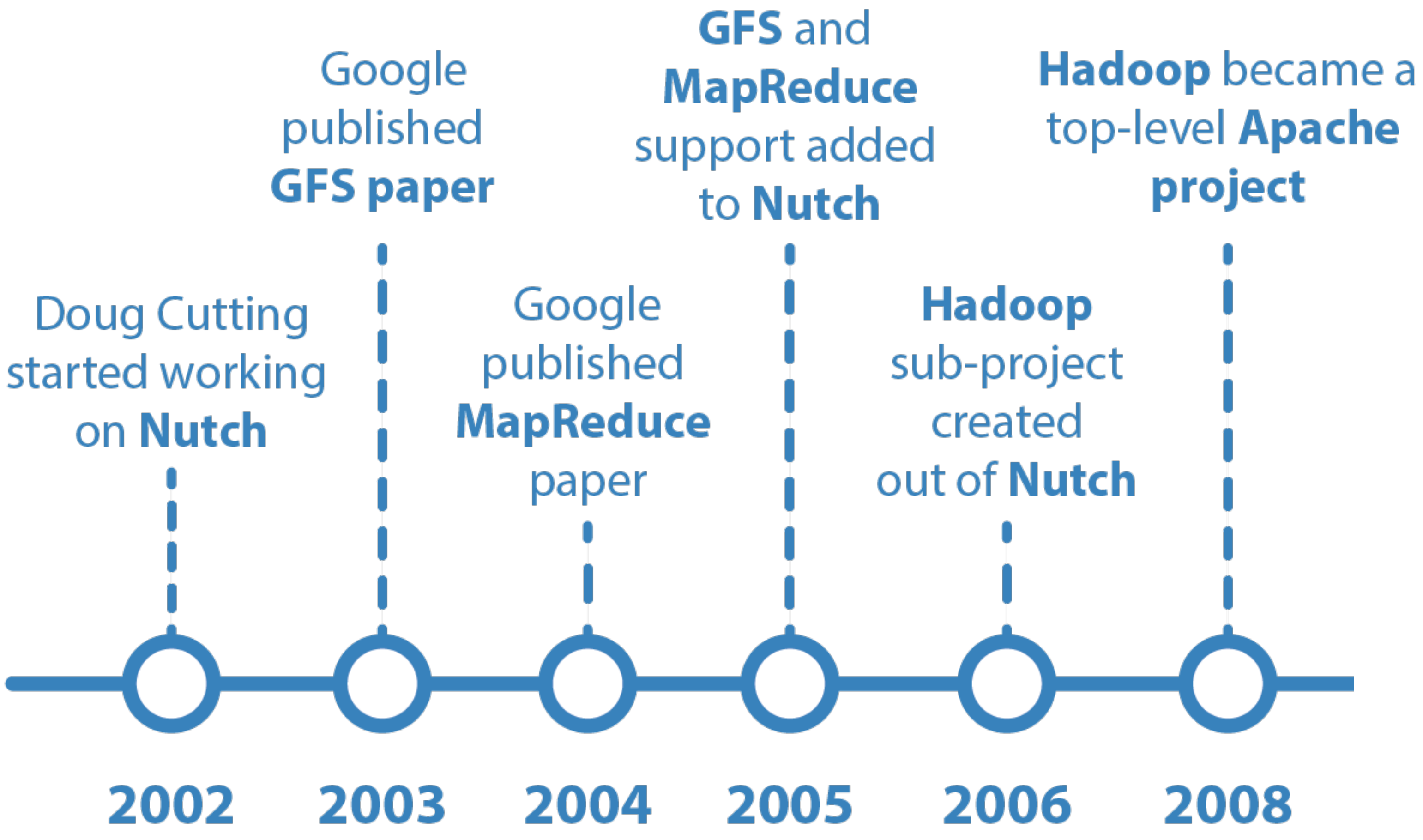


Image source: https://dzone.com/articles/lambda-architecture-with-apache-spark

MapReduce Overview

Read a lot of data

Map: extract something you care about from each record

Shuffle and Sort

Reduce: aggregate, summarize, filter, transform

Write the results

Paradigm stays the same,
Change map and reduce
functions for different problems

Slide source: Jeff Dean

Data Model

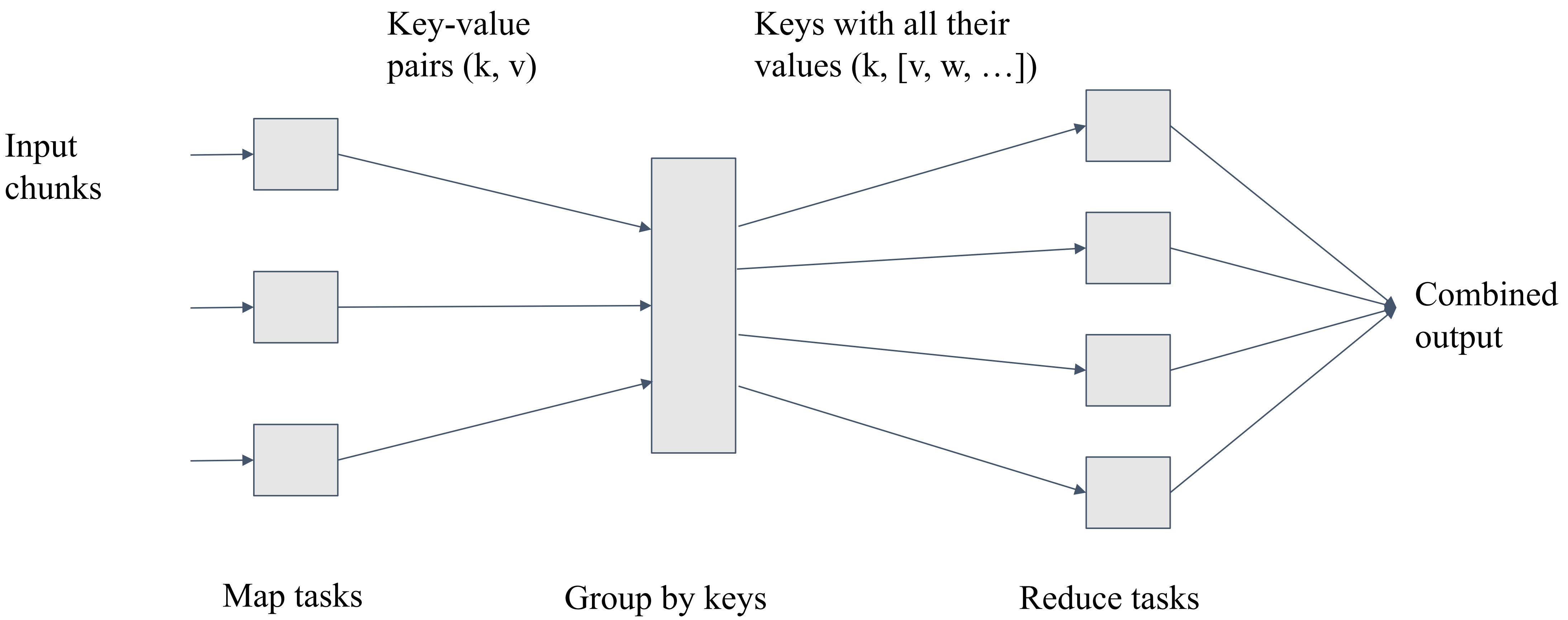
Data is stored as flat files, not relations!

A file = a bag of (key, value) pairs

A MapReduce program

- Input: a bag of (inputkey, value) pairs
- Output: a bag of (outputkey, value) pairs
  - outputkey is optional

MapReduce Overview



Example: Word counting

- Count the number of times each distinct word appears in large collection of documents

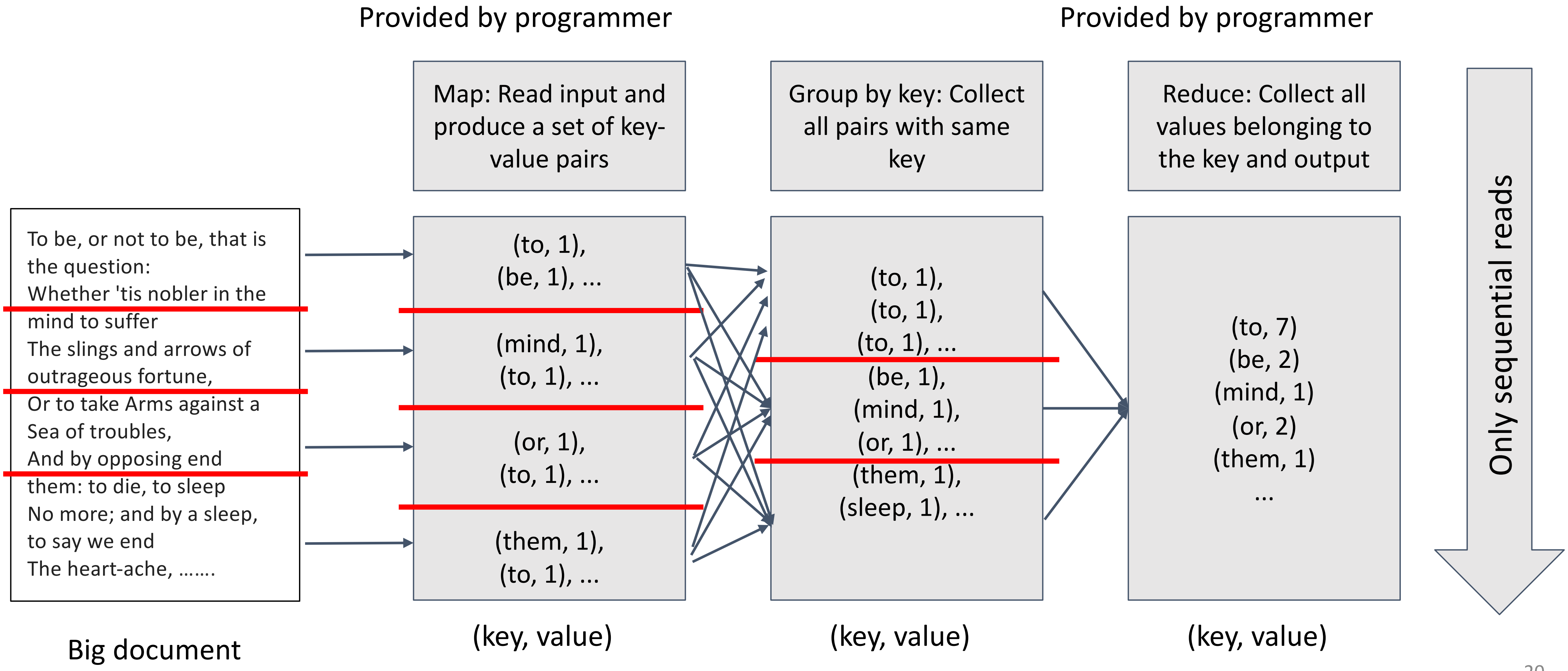
- Many applications:
  - Analyze web server logs to find popular URLs
  - Statistical machine translation (e.g., count frequency of all 5-word sequences in documents)

Map and Reduce functions for word counting

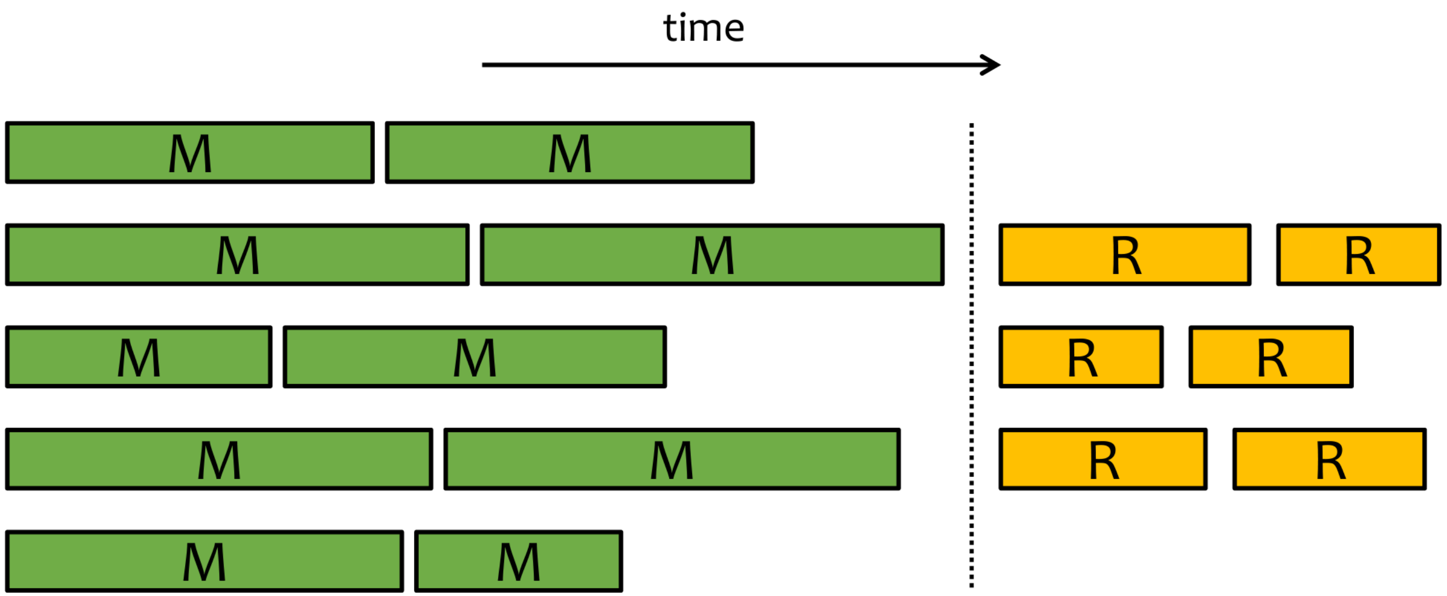
```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
     emit(w, 1)

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: word counting



MapReduce execution timeline



- When there are more tasks than workers, tasks execute in "waves"
  - Boundaries between waves are usually blurred
- Reduce tasks can't start until all map tasks are done

Fault Tolerance

- If one server fails every year… then a job with 10,000 servers will fail in less than an hour
- MapReduce handles fault tolerance by writing intermediate files to disk:
  - Mappers write file to local disk
  - Reducers read the files as input; if the server fails, the reduce task is restarted on another server

MapReduce Summary

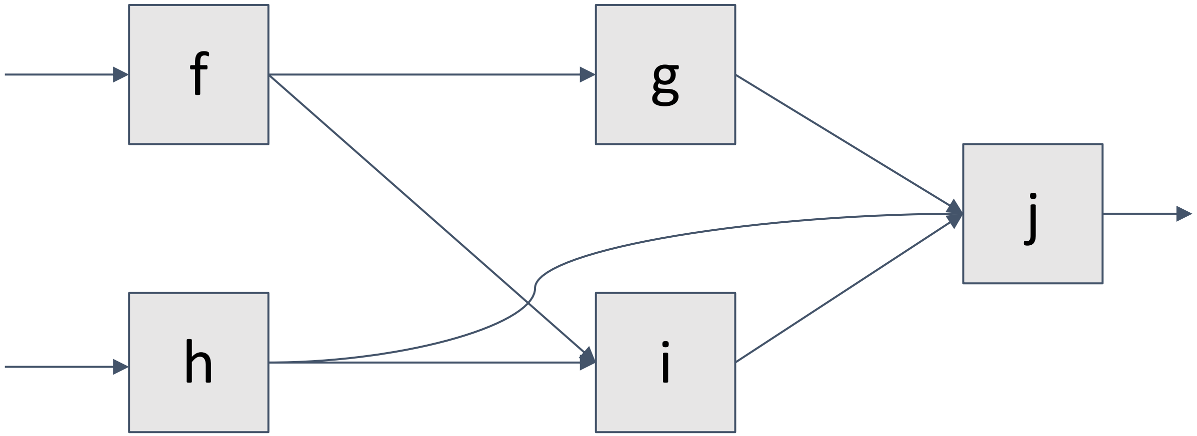
- A style of programming for managing many large-scale computations in a way that is tolerant of hardware faults
  - Just need to write two functions called *Map* and *Reduce*
  - The system manages parallel execution, coordination of tasks that execute Map or reduce, and dealing with failures

- It has several implementations, including Hadoop, Spark, Flink, and the original Google implementation just called “MapReduce”

Workflow systems

- Extends MapReduce by supporting acyclic networks of functions
  - Simple two-step workflow → any acyclic workflow of functions
  - Each function implemented by a collection of tasks
  - A master controller is responsible for dividing work among tasks

- Examples: Apache Spark and Google TensorFlow

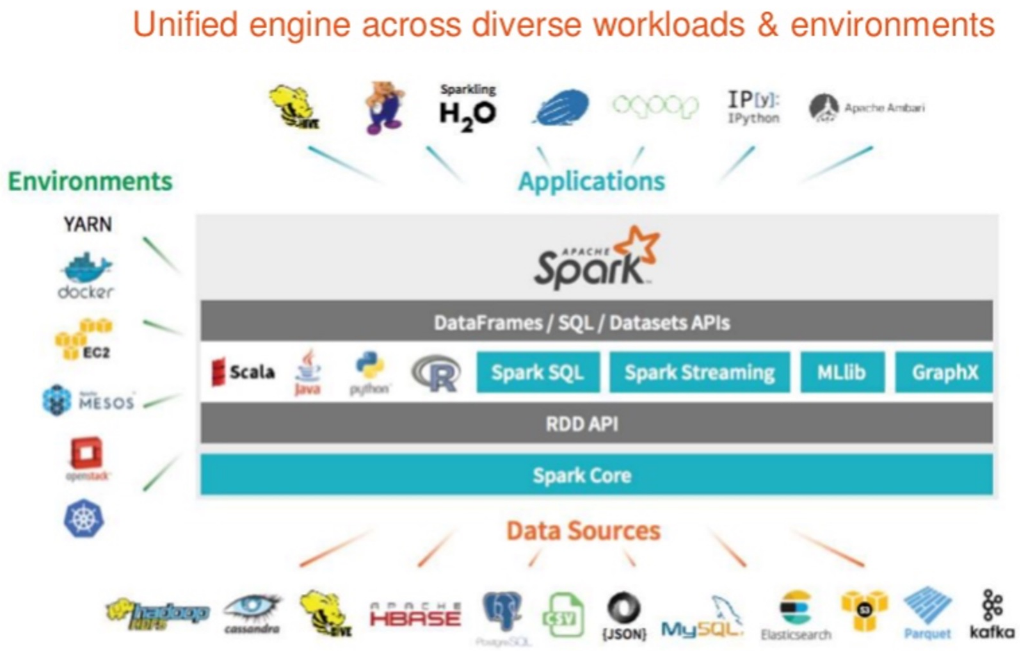


Blocking property

- Like MapReduce, workflow functions only deliver output after completion
- If task fails, no output is delivered to any successors in flow graph
- A master controller can therefore restart failed task at another compute node

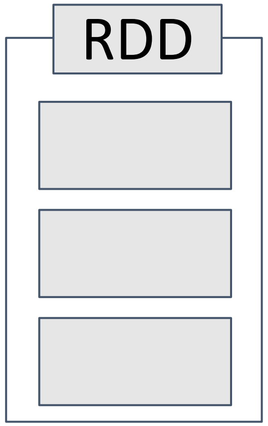
Spark: most popular workflow system

- Developed by UC Berkeley and Databricks, now maintained by Apache
- Advantages over early workflow systems
  - More efficient failure handling
  - More efficient grouping of tasks among compute nodes and scheduling function execution
  - Integration of programing language features such as looping and function libraries



Data Model: Resilient distributed dataset (RDD)

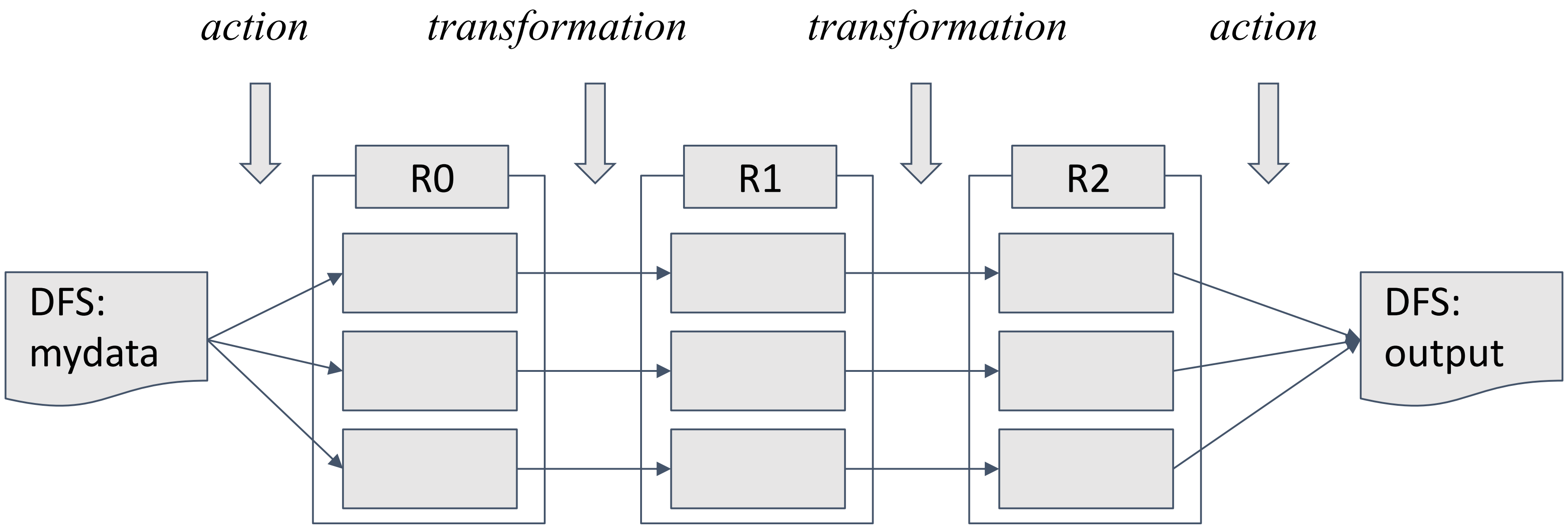
- Central data abstraction of Spark
- A file of objects of one type
  - Statically typed: RDD[T] has objects of type T
- Immutable collections of objects, together with its lineage
  - Lineage = how a dataset is computed
- Spark is resilient against loss of any or all chunks of RDD
  - If RDD in main memory is lost, can recompute lost partitions of RDD using lineage



Spark program

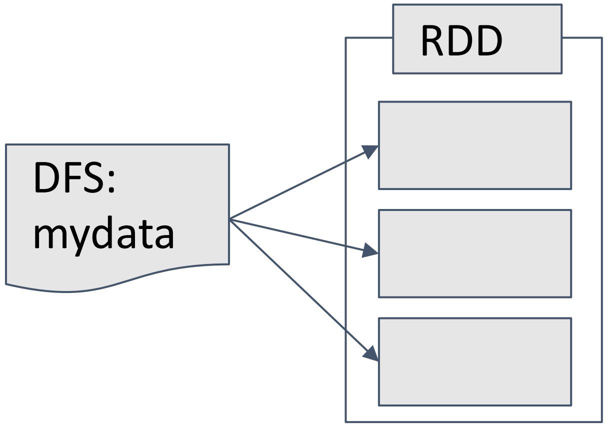
Sequence of steps of

- *Transformations*: apply some function to an RDD to produce another RDD
- *Actions:* Turn RDD into data in surrounding file system and vice versa



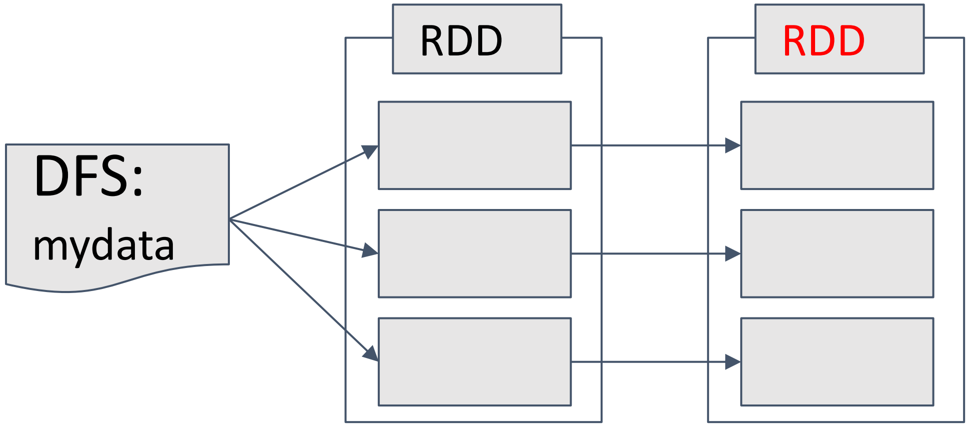
Example: average word length by letter

```
> avglens = sc.textFile(file)
```



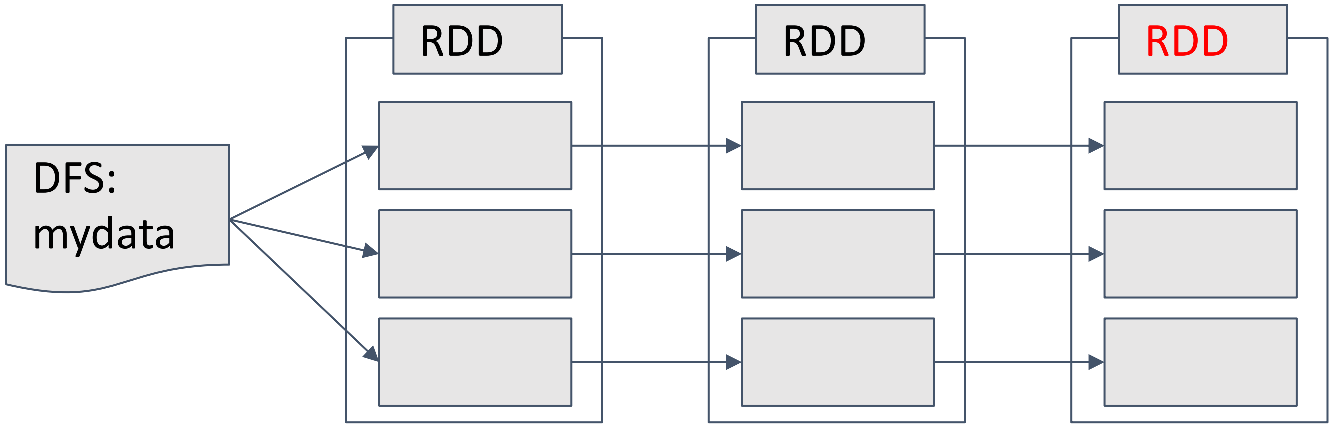
Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



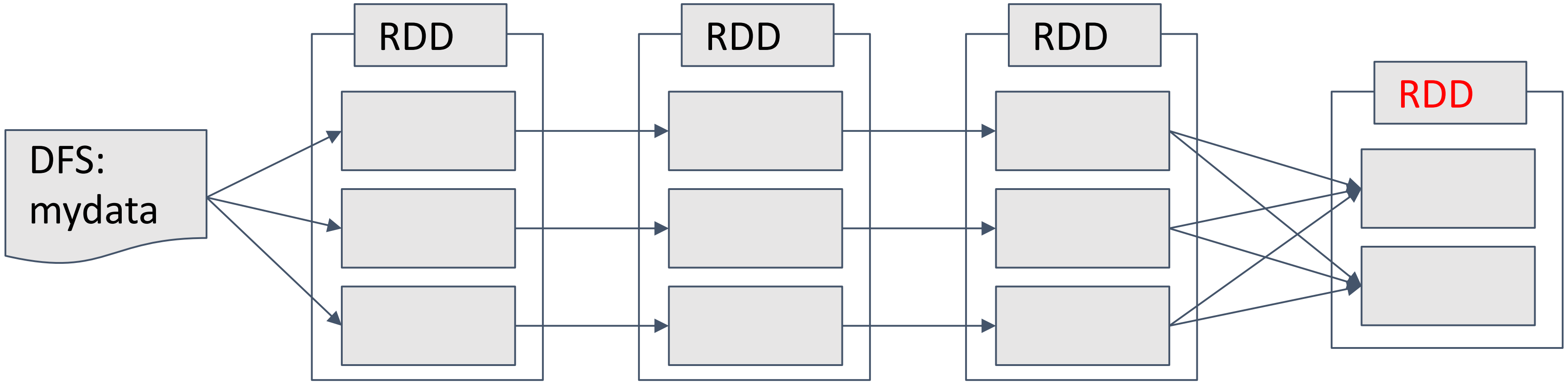
Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word)))
```



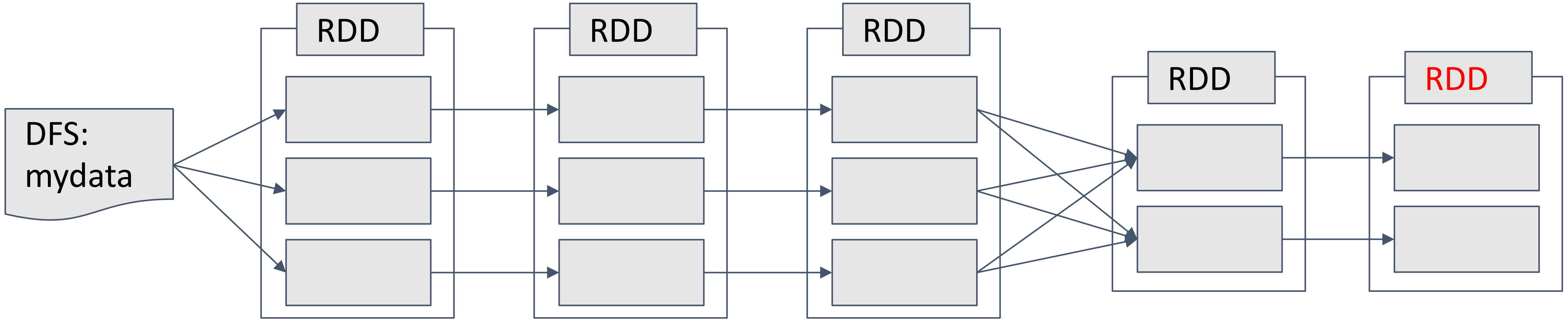
Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey()
```



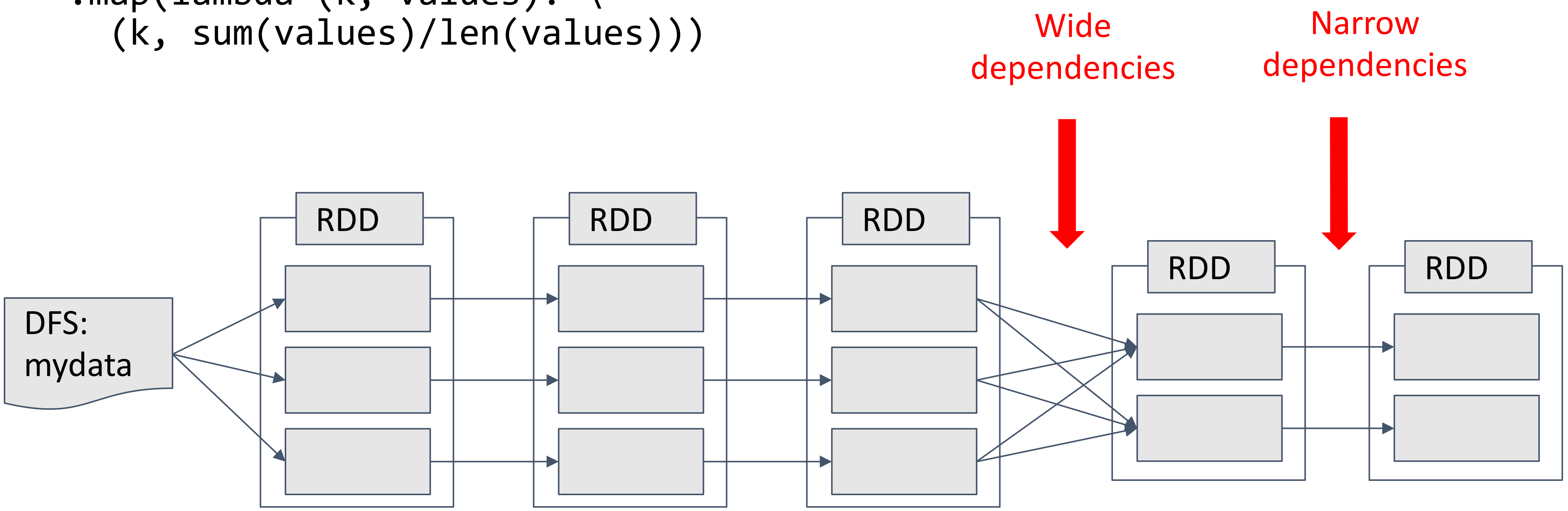
Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
      (k, sum(values)/len(values)))
```



Example: average word length by letter

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
      (k, sum(values)/len(values)))
```



Map

- Transformation that takes function as parameter and applies it to every element of RDD
- Returns a new RDD where each input element is transformed into exactly one output element.
- Not exactly the same as Map of MapReduce
  - In MapReduce, a Map function is applied to a key-value pair and produces a set of key-value pairs
  - In Spark, a Map function can apply to any object type, but produces exactly one object

```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split()) \
   .map(lambda word: (word[0], len(word))) \
   ...
```

Flatmap

- Transformation analogous to MapReduce Map, but no restriction on the type
- In comparison to a Spark Map, each object maps to a list of 0 or more objects
- All the lists are then “flattened” into a single RDD of objects

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    ...
```

Filter

- Transformation that takes a predicate that applies to the RDD object type and returns elements that satisfy predicate

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .filter(lambda word: word not in stoplist)) \
    ...
```

Reduce

- An action (not transformation) that returns a value instead of an RDD
- Takes parameter that is a function of type (V, V) => V
  - When applied to RDD, the function is repeatedly applied on pairs of elements to produce a single one
  - Function can be associative and commutative (e.g., addition), but this is not required

```
> totlen = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: len(word)) \
    .reduce(lambda a, b: a + b)
```

Other examples of actions

- Actions are operations that trigger the execution of the Spark computation and return results to the driver program or write data to external storage systems

```
# Collect RDD elements to the drive program
collected_data = rdd.collect()

# Count the number of elements in the RDD
count = rdd.count()

# Get the first three elements of the RDD
element = rdd.take(3)

# Save RDD elements to a text file
rdd.saveAsTextFile(”output_folder”)
```

Relational database operations

- Some Spark operations behave like relational algebra operations on relations that are represented by RDD’s

Join

- Takes two RDD's of type key-value pair where the key types are the same
- For each pair (*k*, *x*) and (*k*, *y*), produce (*k*, (*x*, *y*))
- Output RDD consists of all such objects

```
> x = sc.parallelize([("a", 1), ("b", 4)])
> y = sc.parallelize([("a", 2), ("a", 3)])
> x.join(y).collect()
[('a', (1, 2)), ('a', (1, 3))]
```

GroupByKey

- Takes RDD of key-value pairs, produces a set of key-value pairs
  - The value type for the output is a list of values of the input type
- Sorts input RDD by key
- For each key k produces the pair (k, [v_1, v_2, …, v_n]) for v_i's associated with k

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    ...
```

Exercise

- There are many other transformations and actions supported by Spark
- For example, *reduceByKey(func)* is like *groupByKey()*, but also applies a reduce function *func* of the form (V, V) => V on the values

- Problem: implement word count using reduceByKey()

```
> avglens = sc.textFile(file) \
   .flatMap(lambda line: line.split()) \
   ...
```

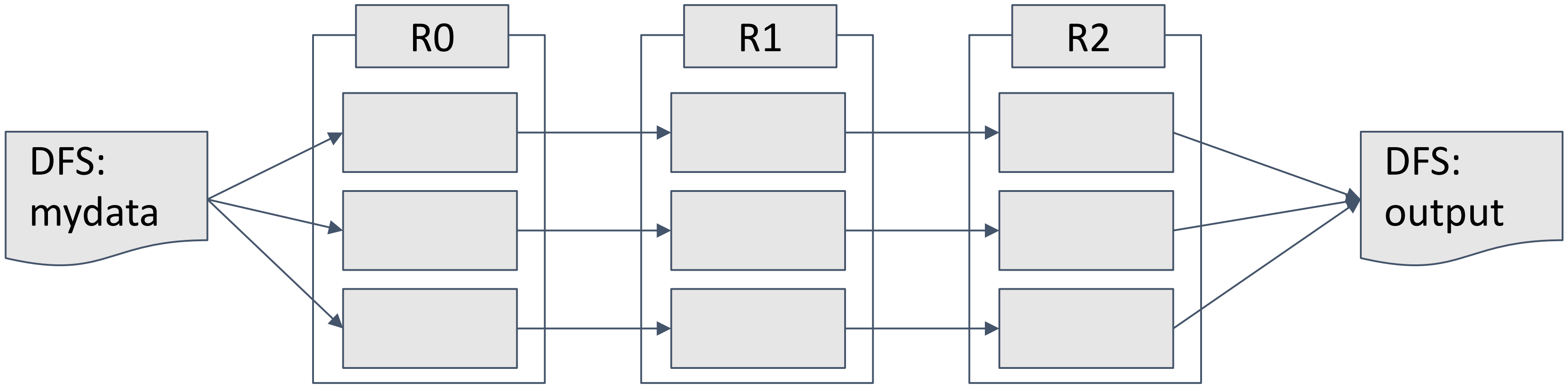
Spark implementation

- Similar to MapReduce,
  - RDD is divided into chunks, which are given to different compute nodes
  - Transformation on RDD can be performed in parallel on each of the chunks

- Two key improvements
  - Lazy evaluation of RDD's
  - Lineage for RDD's

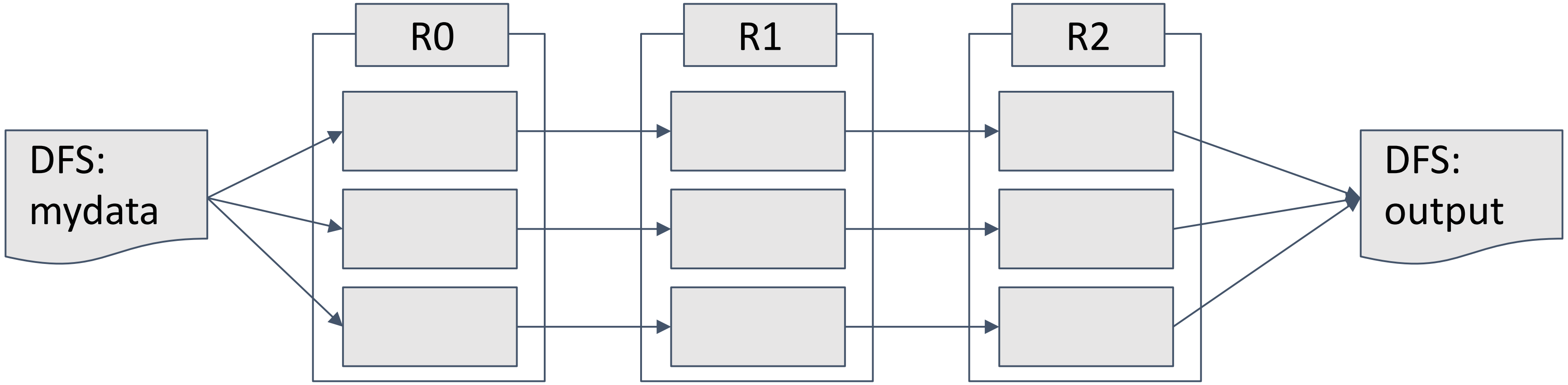
Lazy evaluation

- Spark does not actually apply transformations to RDD's until it is required to do so (e.g., storing RDD to file system or returning a result to application)
- As a result, many RDD's are not constructed all at once
  - A created RDD chunk can be used at the same compute node to apply another transformation
  - Benefit: This RDD is never stored on disk and never transmitted to another compute node

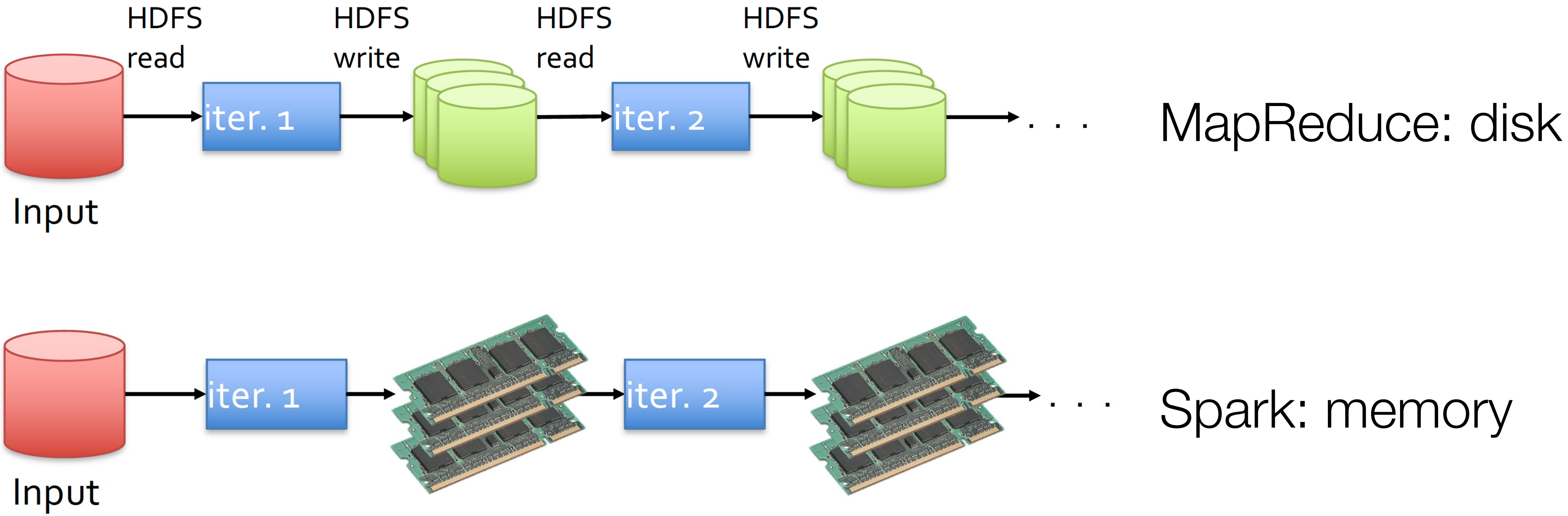


Lazy evaluation example

- Count words in document that are not stop words
  - Apply Flatmap to input RDD R_0 to create (w, 1) pairs
  - Apply Filter to each chunk R_1 of resulting RDD to produce R_2
  - If R_2 is stored in DFS (action), it triggers the transformation in R_1 and R_2



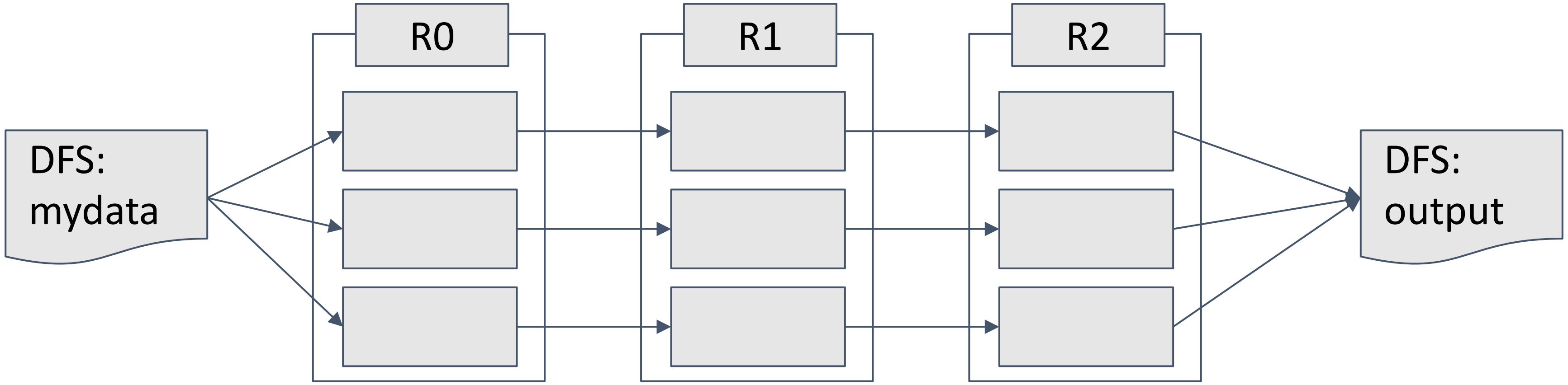
Data Sharing in MapReduce vs Spark



This is why Spark is significantly faster for iterative algorithms

Resilience of RDD's

- Spark records the *lineage* of every RDD, which can be used to re-create any RDD
    - If R_2 is lost, reconstruct from R_1
    - If R_1 is lost, reconstruct from R_0
    - If R_0 is lost, reconstruct from file system



Why not store intermediate values (like MapReduce)?

- Trading off complex recovery for greater speed when things go right is generally good
- The faster Spark runs, the less chance there is a node failure
- Also storing intermediate values requires redundant file storage for a long period

Spark programming guide and paper

- To learn more about writing Spark applications, please read the Spark programming guide: https://spark.apache.org/docs/latest/rdd-programming-guide.html

- We will read more technical details of Spark in this paper: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf