CS 6400 A

Database Systems Concepts and Design

Lecture 18

10/30/24

So far: One query/update
One machine

Multiple query/updates
One machine

Transactions

One query/update
Multiple machines

Distributed query processing
Map-Reduce, Spark

Agenda

1. Distributed File System
2. Map Reduce
3. 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)

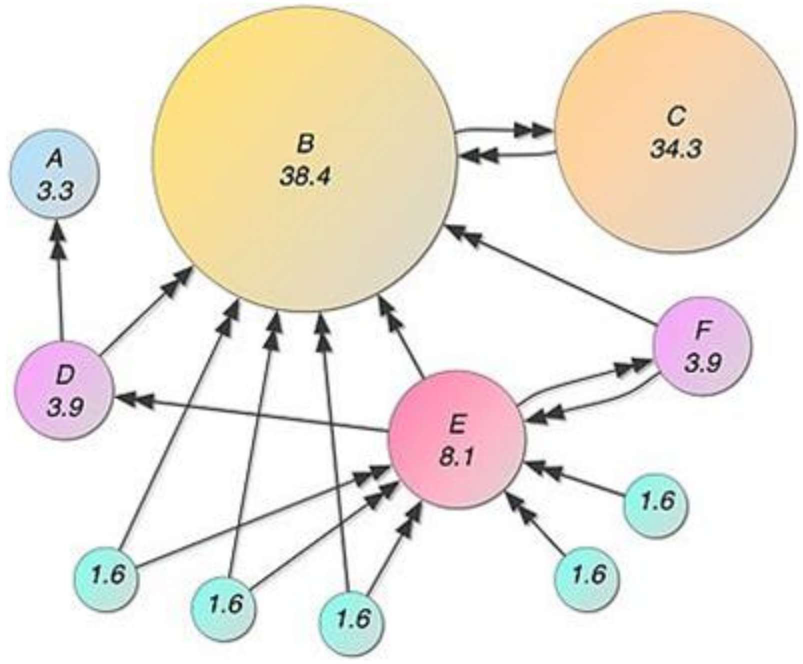
Trades off consistency requirements of RDBMS for speed

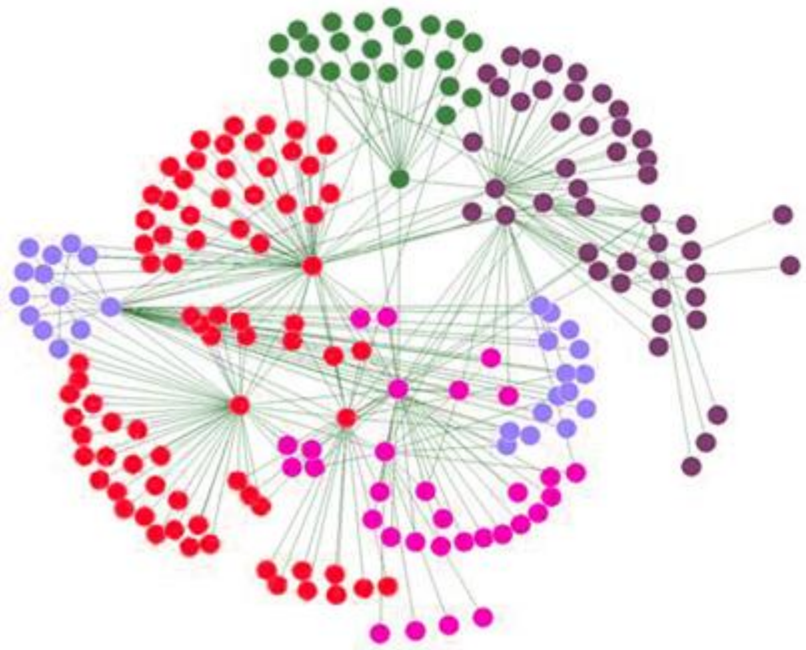
Image source: https://www.geeksforgeeks.org/what-is-internet-definition-uses-working-advantages-and-disadvantages/

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





Horizontal vs Vertical Scaling

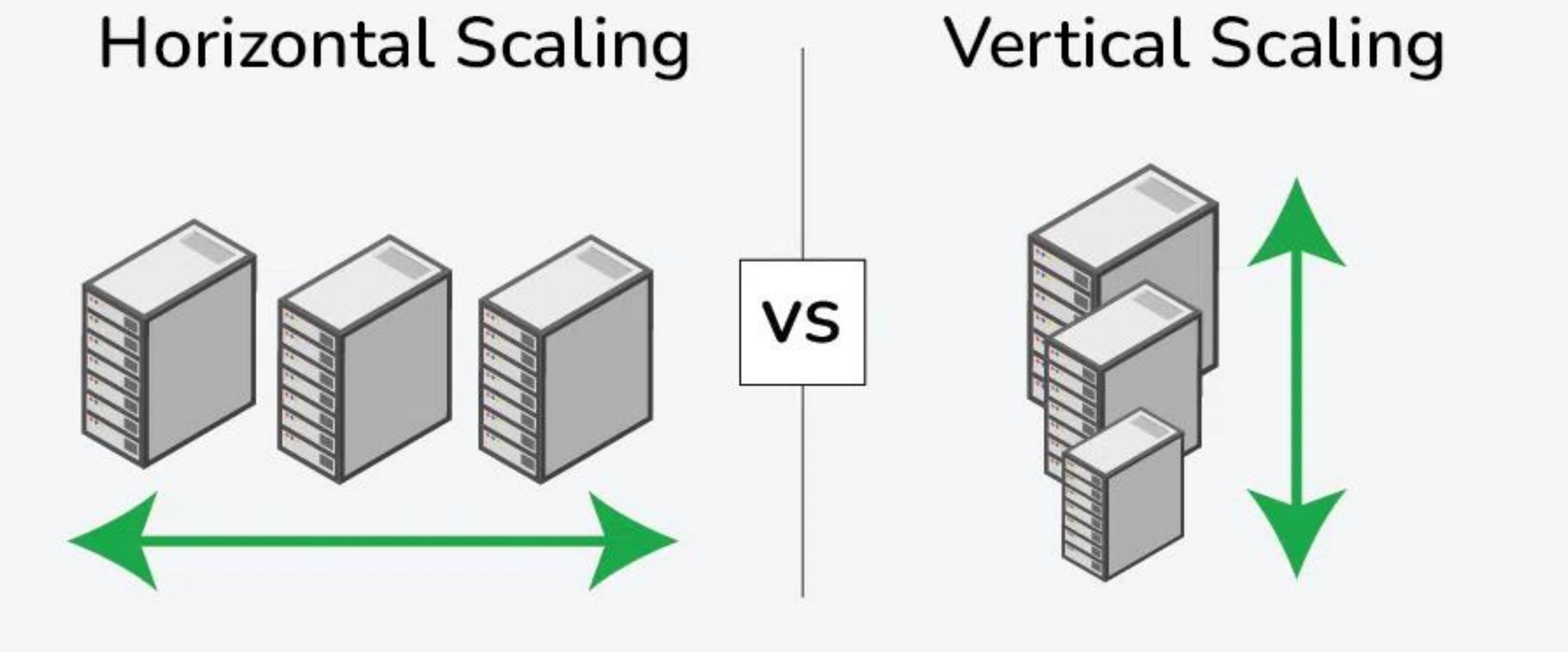


Image source: https://www.geeksforgeeks.org/system-design-horizontal-and-vertical-scaling/

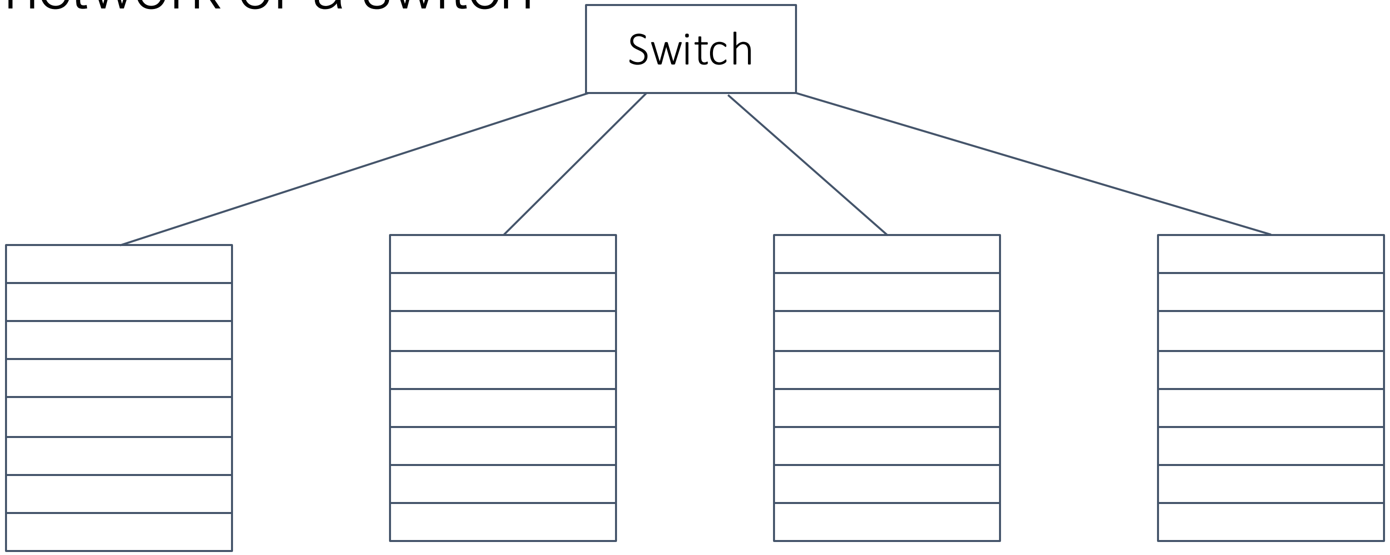
Horizontal scaling

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

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

New Challenges

How do you distribute computation?

How can we make it easy to write distributed programs?

It is a fact of life that components 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!

Need solutions for recovering data and computation during failure

A new software stack

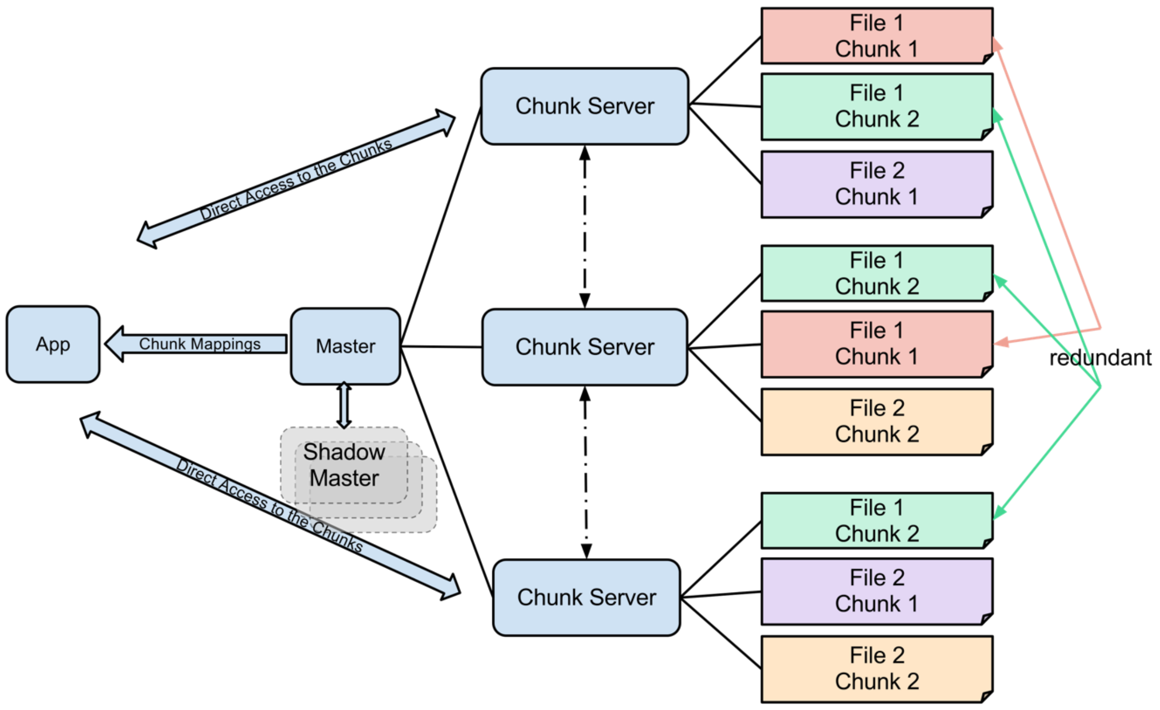
Distributed file system

- Example: Google File System
- Large blocks and data replication to protect against media failures

Programming abstraction

- Example: Map Reduce
- Enables common calculations on large-scale data to be performed on computing clusters efficiently
- Tolerant to hardware failures

1. Distributed File System



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

SOSP'03

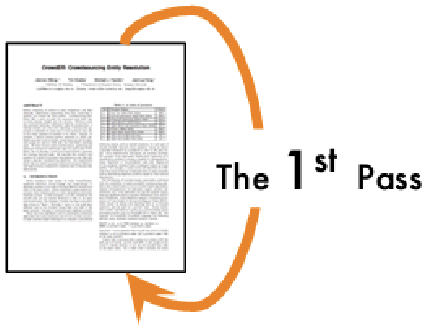
How to read a paper in depth

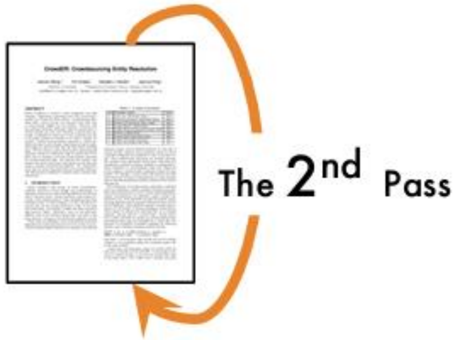
The "three-pass” approach [1]

first pass: a quick scan

second pass: with greater care, but ignore the details

third pass: re-implementing the paper





[1] S. Keshav. How to read a paper? http://blizzard.cs.uwaterloo.ca/keshav/home/Papers/data/07/paper-reading.pdf

The first pass: a quick scan

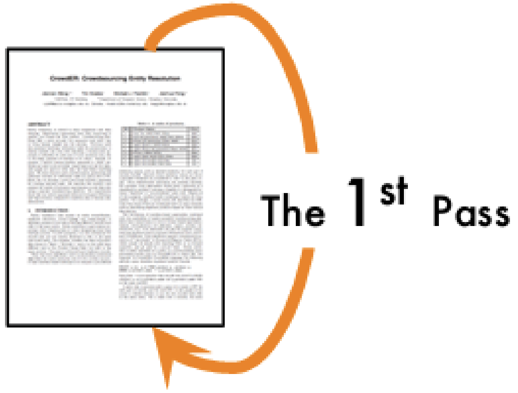
Goal: get bird's-eye view of the paper (5~10 min)

What to read:

- Title, abstract, introduction and conclusion
- Section and sub-section headings
- Main figures
- Scan of bibliography

You should be able to answer:

- What type of paper is this?
- What are the main contributions?



The second pass: grasp the content

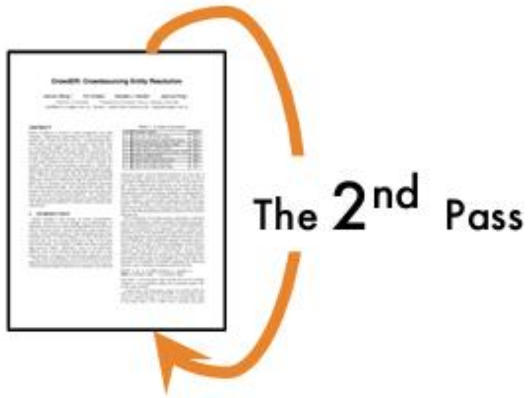
Goal: get a good understanding of the ”meat” of the paper

How to read:

- Look carefully at figures, diagrams and examples
- Take notes of questions, unread references etc.
- Ignore proofs, appendix, extensions etc.

You should be able to:

- Summarize main thrusts of the paper, with supporting evidence, to someone else



The third pass: all about the details

Goal: think about what you would have done if you were to re-implement such an idea

How to read:

- Challenge every assumption
- Compare your version with the actual paper
  - Often leads to questions like: why not do it this way?

You should be able to:

- Identify hidden assumptions/potential design flaws
- Get ideas for future work

Let’s try the first pass!

1. **Category**: What type of paper is this? A measurement paper? An analysis of an existing system? A description of a research prototype?
2. **Context**: Which other papers is it related to?
3. **Correctness**: Do the assumptions appear to be valid?
4. **Contributions**: What are the paper’s main contributions?
5. **Clarity**: Is the paper well written?

Tips for research paper presentations

Always start with the first pass to get a general impression

- You should be able to give high-level answers to questions like “what problem the paper is trying to solve”, “why does it matter”, and “why is the problem challenging” after this pass

Do a second pass to understand the main technical contributions

- We have prepared a detailed reading guide for each paper that tells you which sections to focus on versus which sections to skip

No need to do a third pass

Large-scale file system organization

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

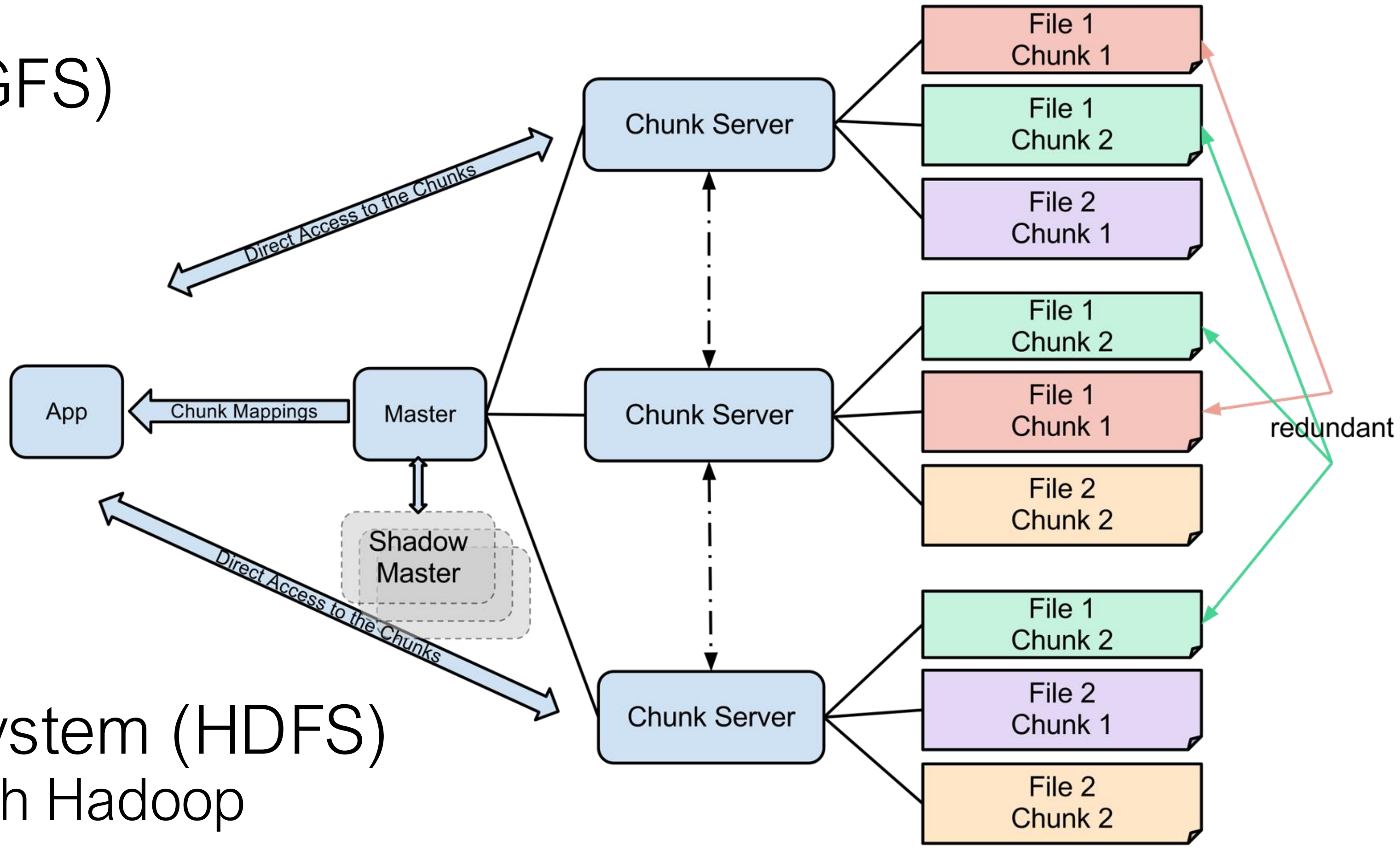
A Distributed File System (DFS) can be used when:

- For very large files: TBs, PBs
- Files are rarely updated and usually read or appended with data
- Mostly sequential reads
- Not useful for OLTP

Distributed File System implementations

The Google File System (GFS)

- Previously used in Google
- Proprietary



Hadoop Distributed File System (HDFS)

- Open-source DFS used with Hadoop

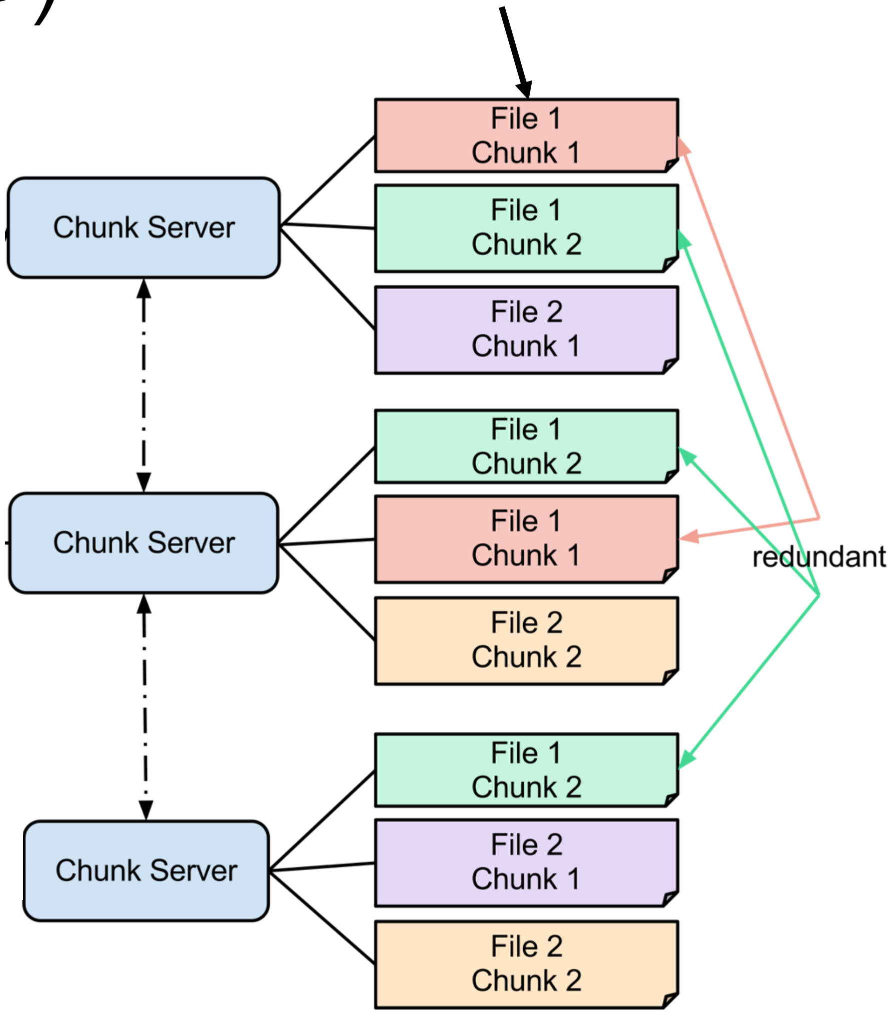
Image source: https://en.wikipedia.org/wiki/Google_File_System#/media/File:GoogleFileSystemGFS.svg

The Google File System (GFS)

Files are divided into chunks, which are typically 64 MBs

- Chunks are replicated (say 3 times) at different compute nodes (called chunks servers)
- The compute nodes should be located on different racks
- Chunk size and degree of replication decided by the user

assigned an immutable and globally unique 64 bit chunk handle

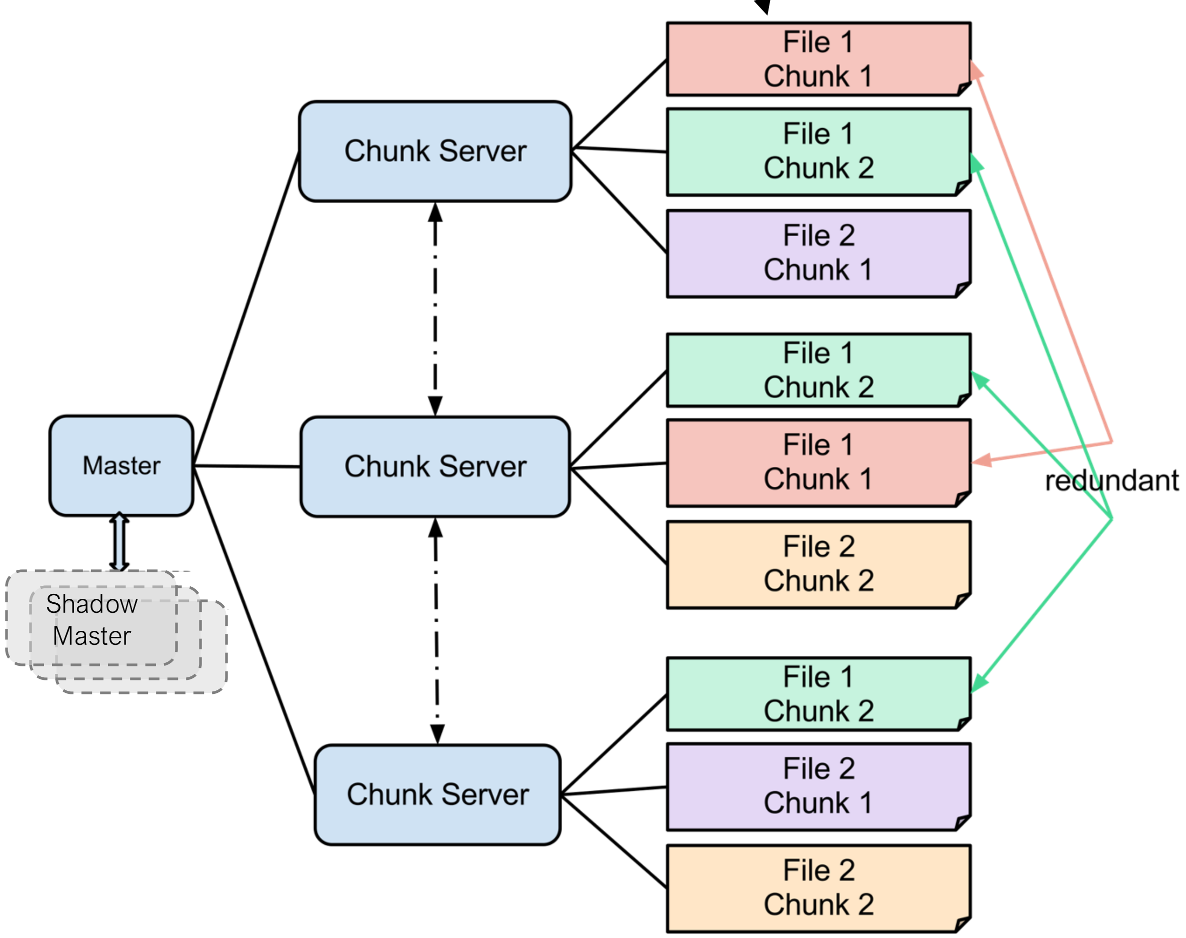


The Google File System (GFS)

Master assigns an immutable and globally unique 64 bit chunk handle

Master node

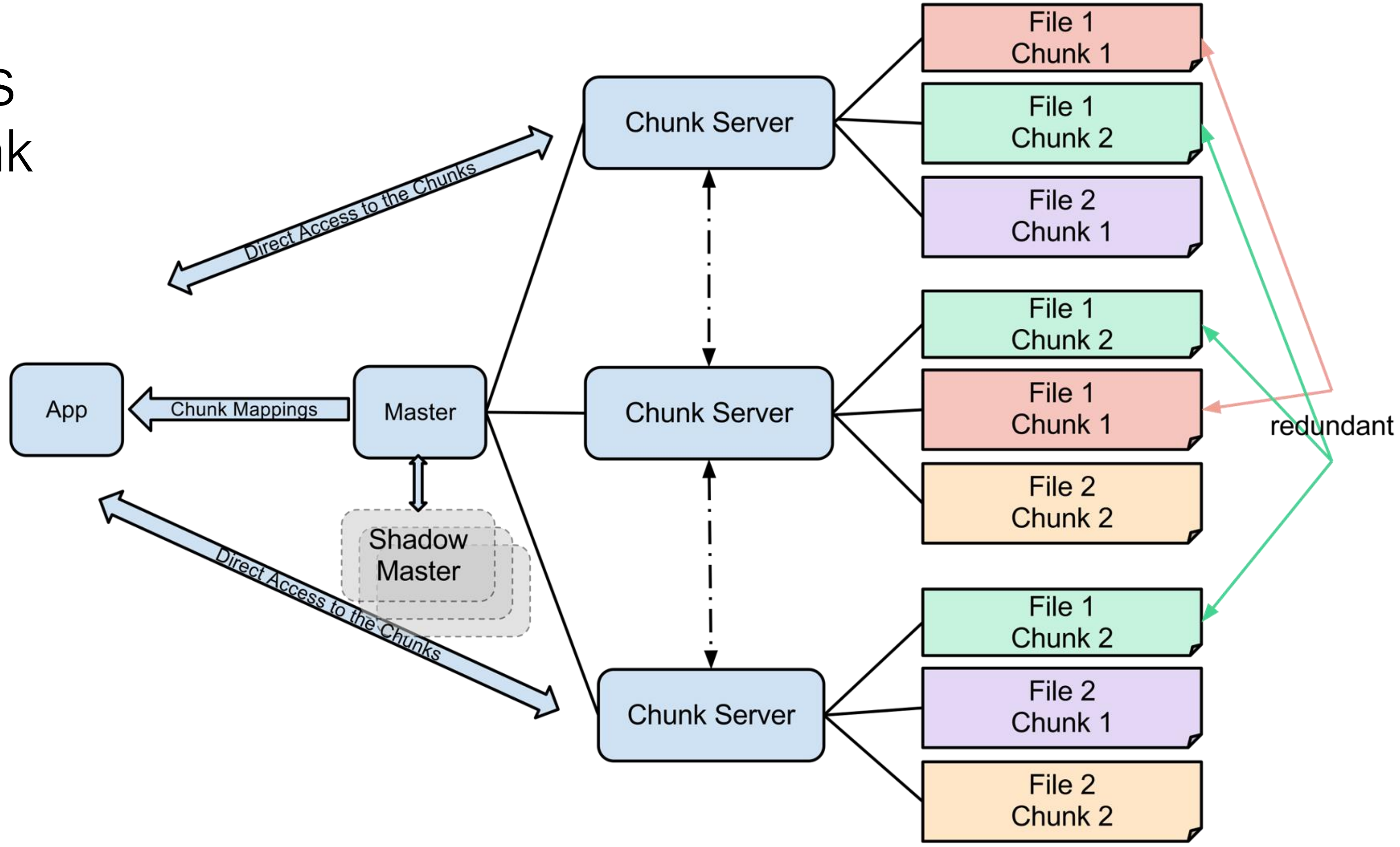
- A single master node for the cluster; master node itself is replicated
- Stores metadata (in memory): file names + chunk ids + chunk locations, access control
- Master keeps an operations log with checkpointing, similar to the recovery log
- Master keeps in sync with chunk servers using regular heartbeat messages



The Google File System (GFS)

Client library for file access

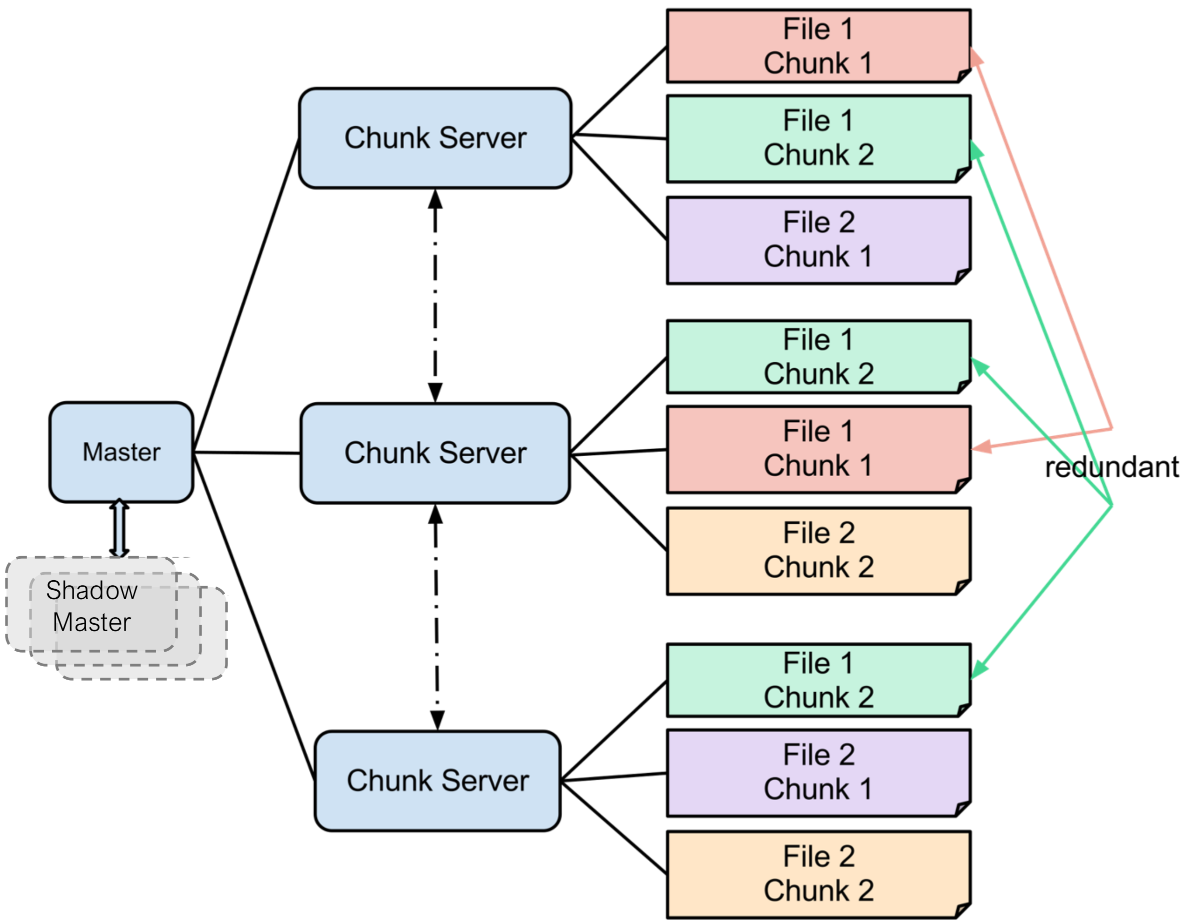
- Talks to master to find chunk servers
- Connects directly to chunk servers to access data



The Google File System (GFS)

Q: What's the benefit of having large chunk sizes (64MB vs file block sizes)

- Master node could become a bottleneck with large number of small files
- Target workload has many sequential reads
- Reduce network overhead



2. MapReduce

A brief history of MapReduce and Hadoop

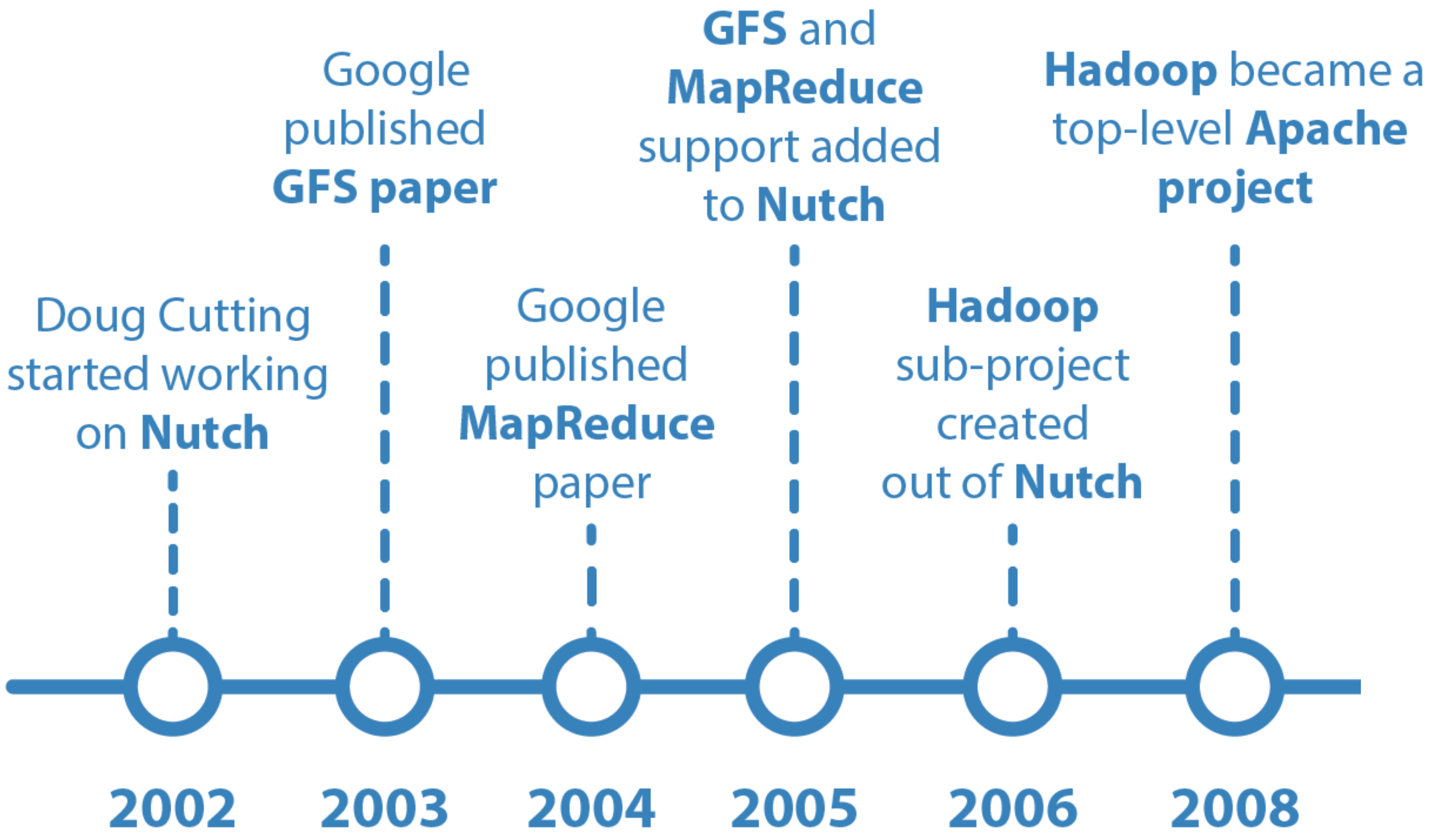


Image source: https://dzone.com/articles/lambda-architecture-with-apache-spark

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

Intermediate data is written to local disk

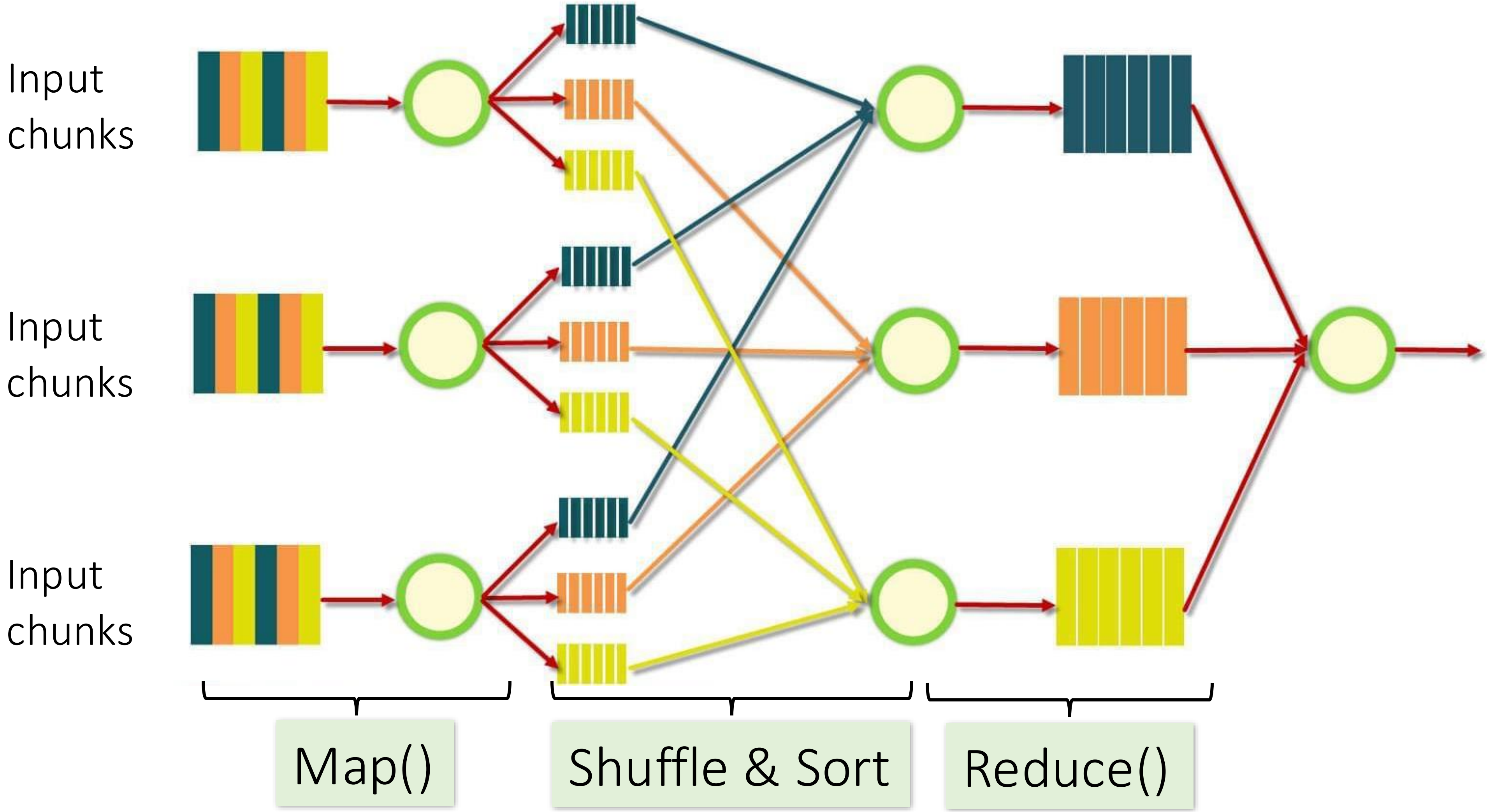


Image source: https://developerzen.com/introduction-to-mapreduce-for-.net-developers/

Example: Word counting

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

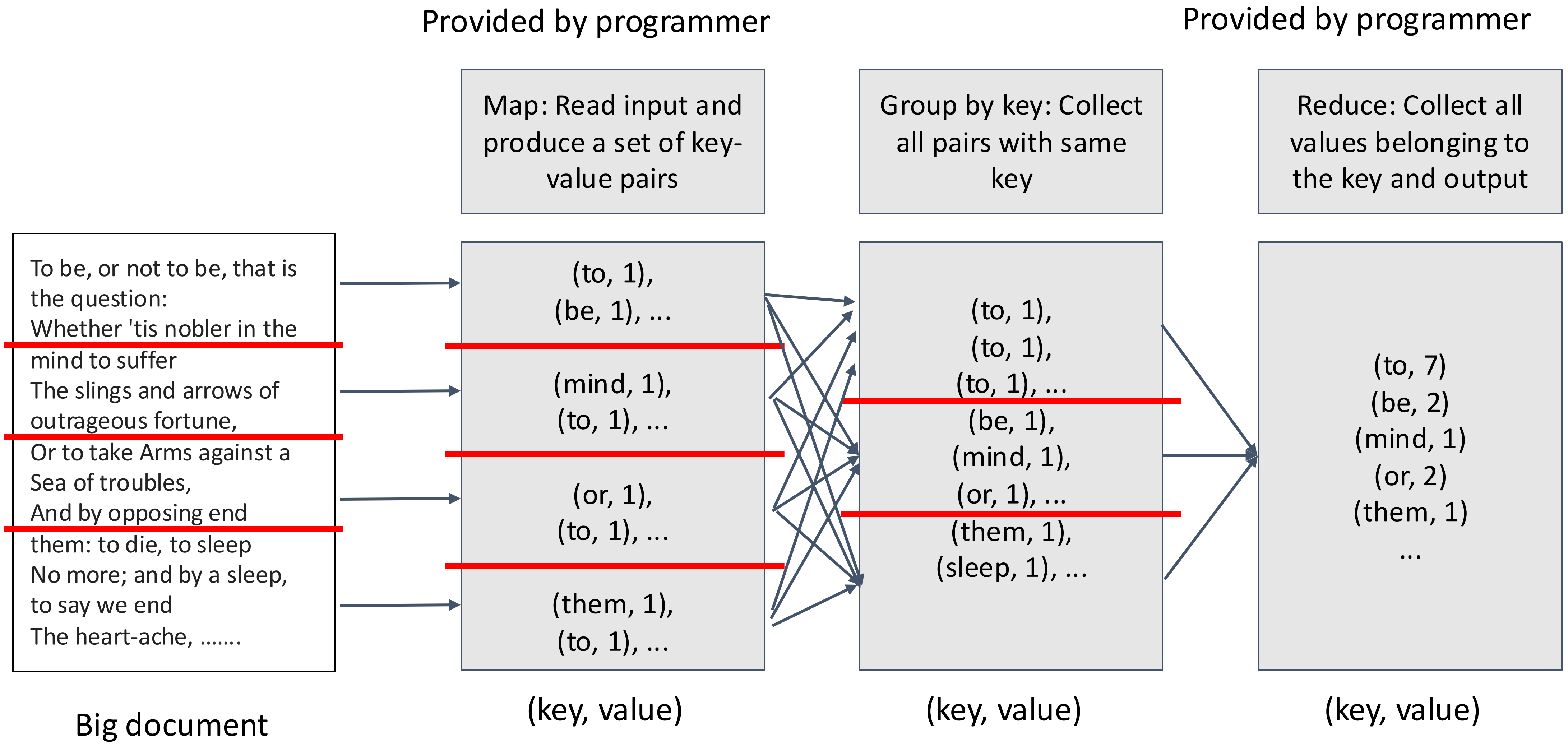
- Many applications:
    - Analyze web server logs to find popular URLs
    - Statistical machine translation (e.g., count frequency of all 5-word sequences in documents)

Map and Reduce functions for word counting

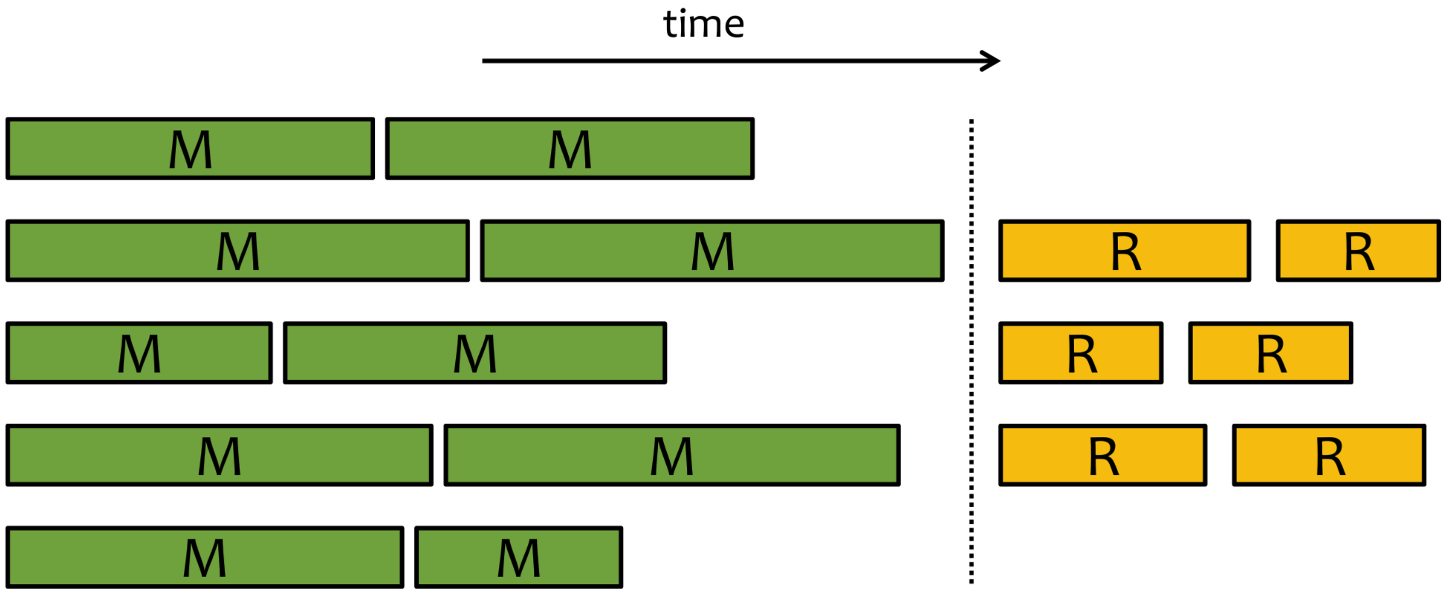
```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
     emit(w, 1)

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

Coding is simple. Do not need to worry about scaling and failure.

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

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

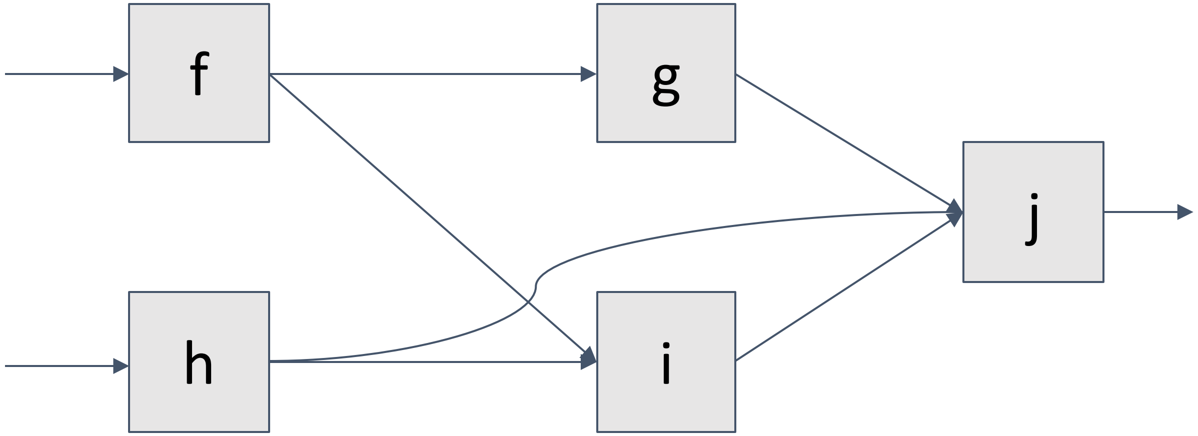
- It has several implementations, including Hadoop, Spark, Flink, and the original Google implementation just called “MapReduce”

3. Spark

Workflow systems

- Extends MapReduce by supporting acyclic networks of functions
  - Simple two-step workflow → any acyclic (DAG) 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

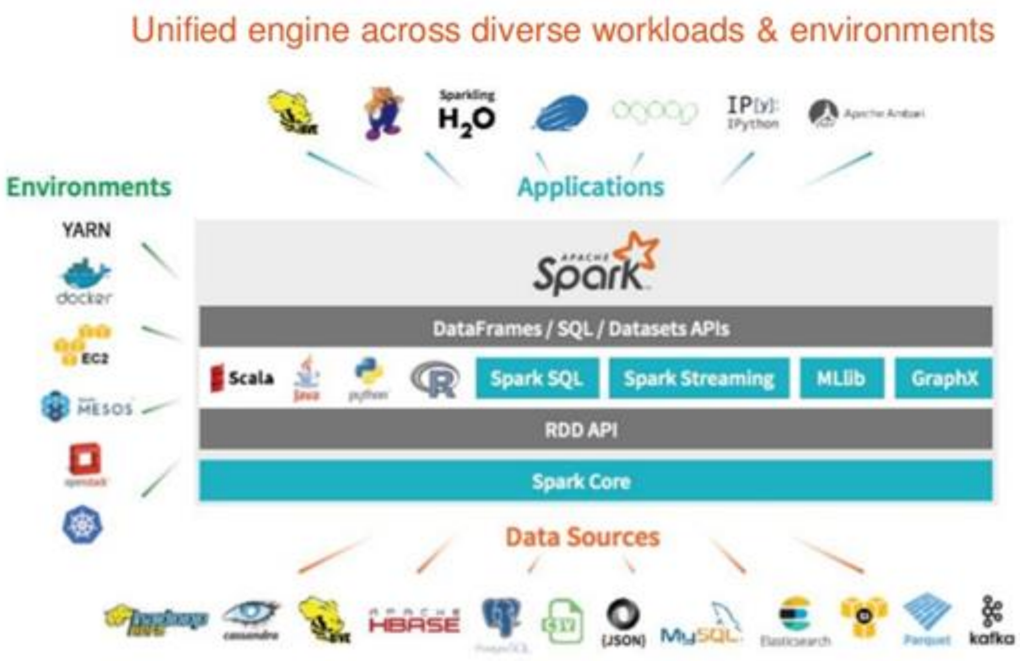


Blocking property

- 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



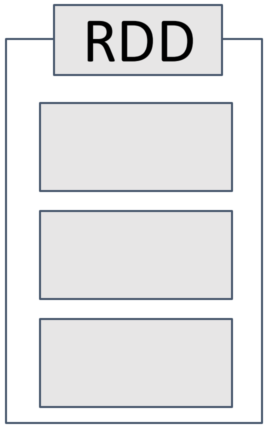
Data Model: Resilient distributed dataset (RDD)

- 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

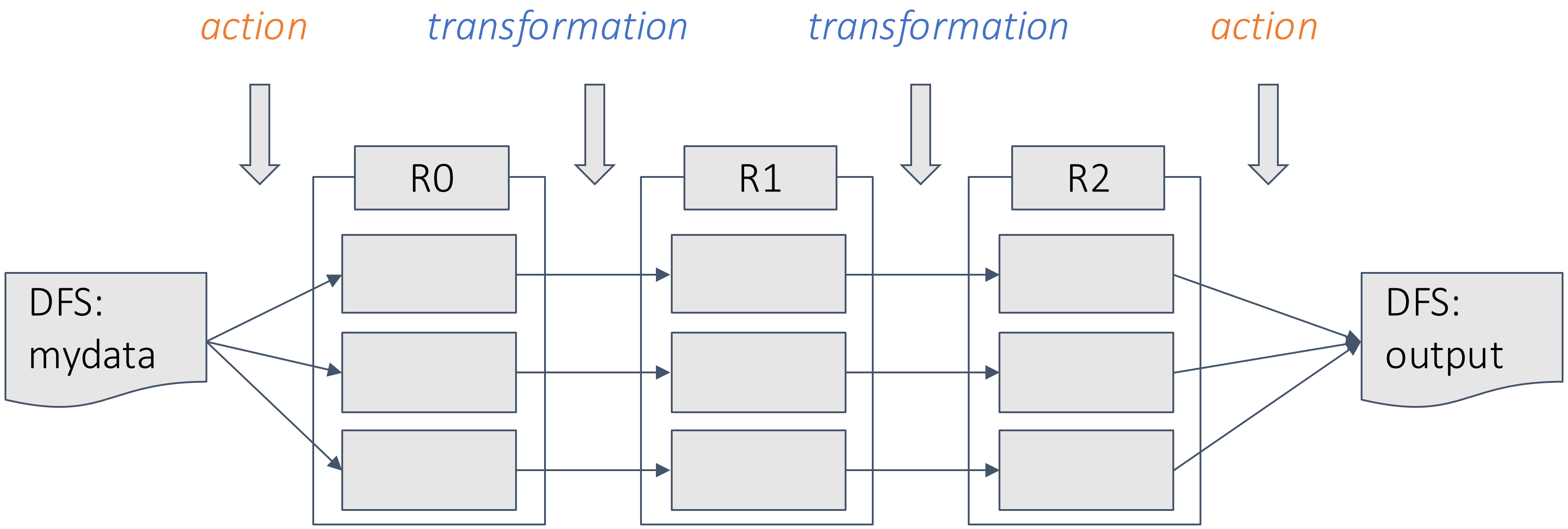
- Spark is resilient against loss of any or all chunks of RDD
  - If RDD in main memory is lost, can recompute lost partitions of RDD using lineage



Spark program

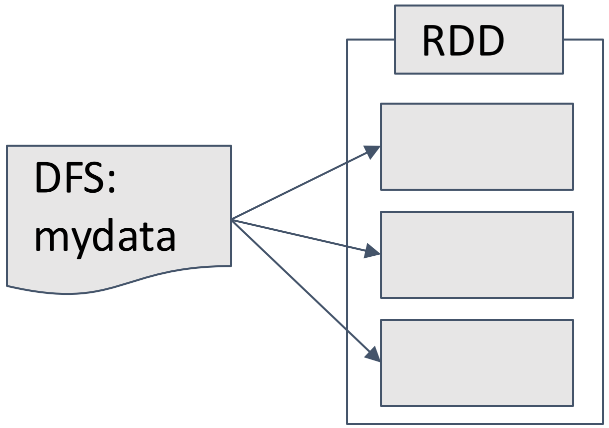
Sequence of steps of

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



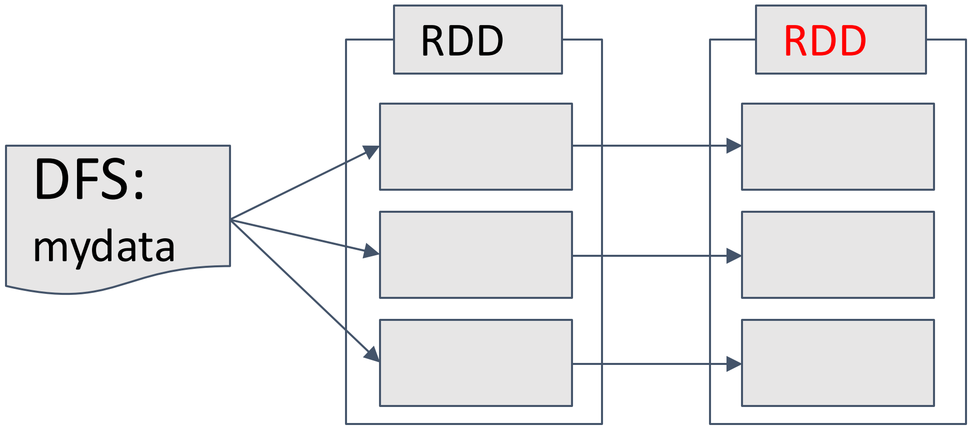
Example: average word length by letter

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



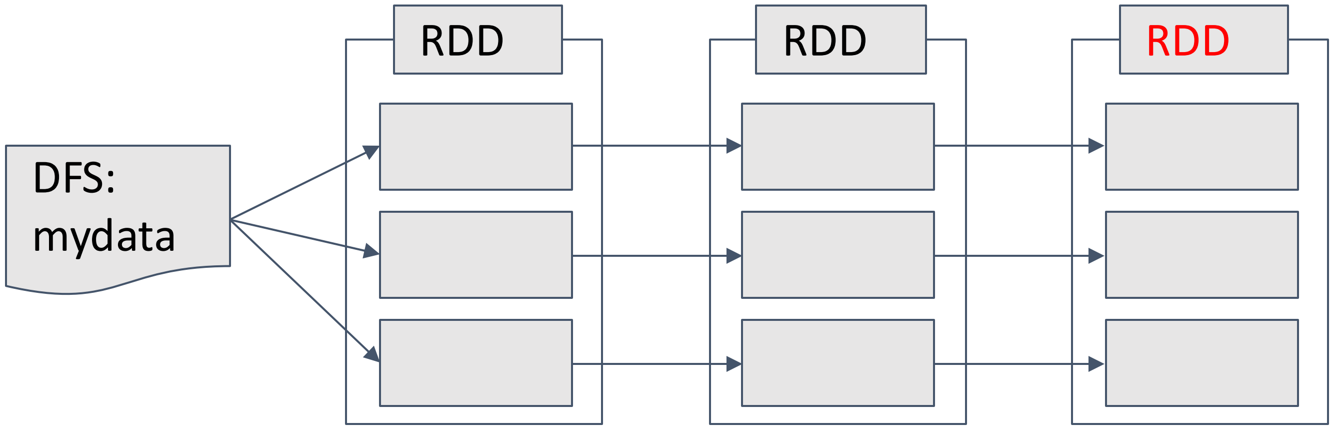
Example: average word length by letter

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



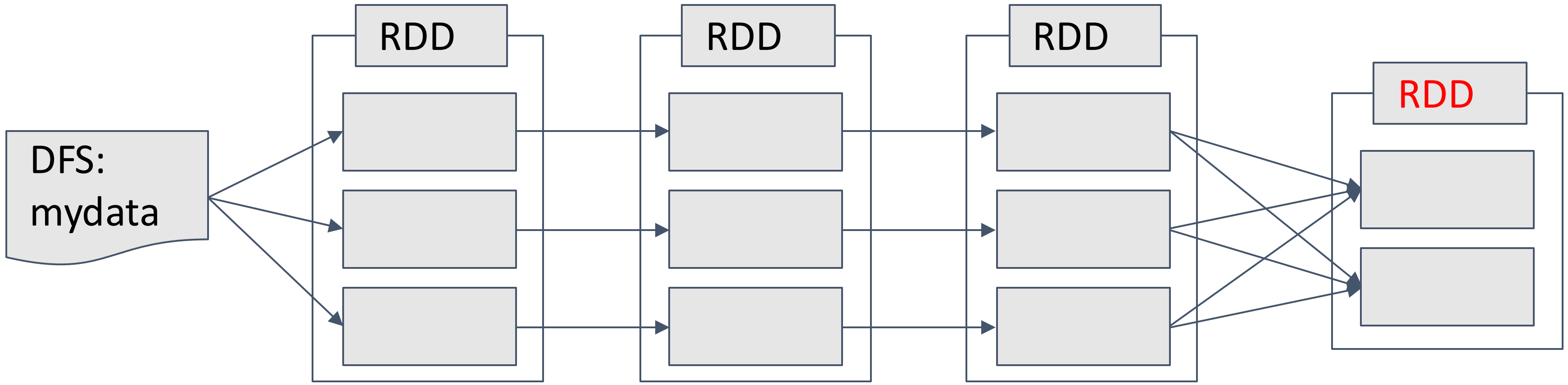
Example: average word length by letter

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



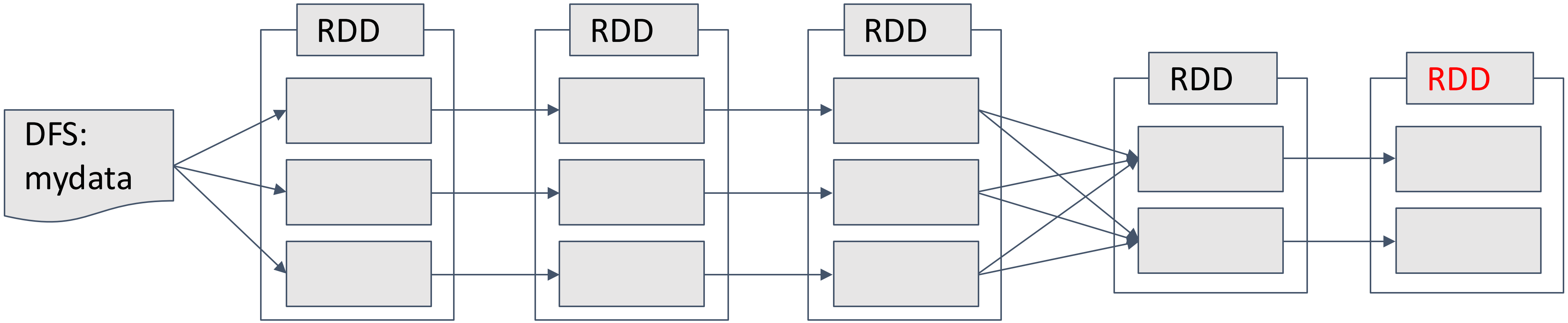
Example: average word length by letter

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



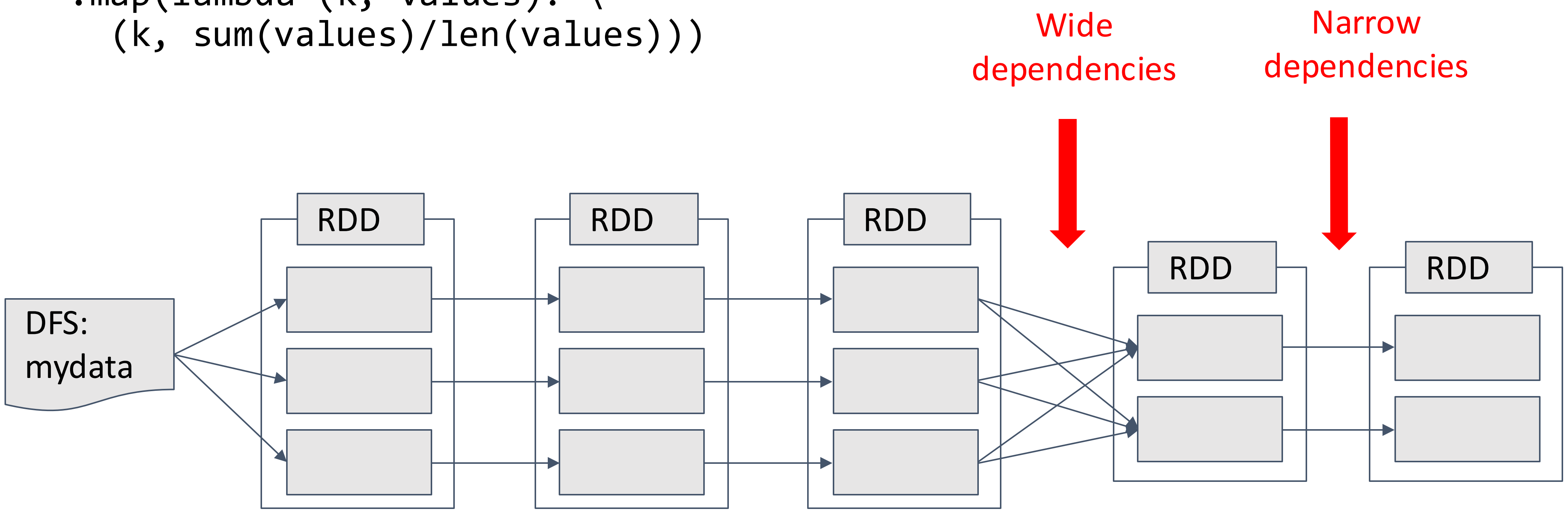
Example: average word length by letter

```
> 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)))
```



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)

```
val data = sc.textFile("input.txt")      // No execution yet
    .map(line => line.split(" "))        // Not executed
    .filter(words => words.length > 2)   // Still not executed
    .count()                             // Now it executes everything
```

Lazy evaluation

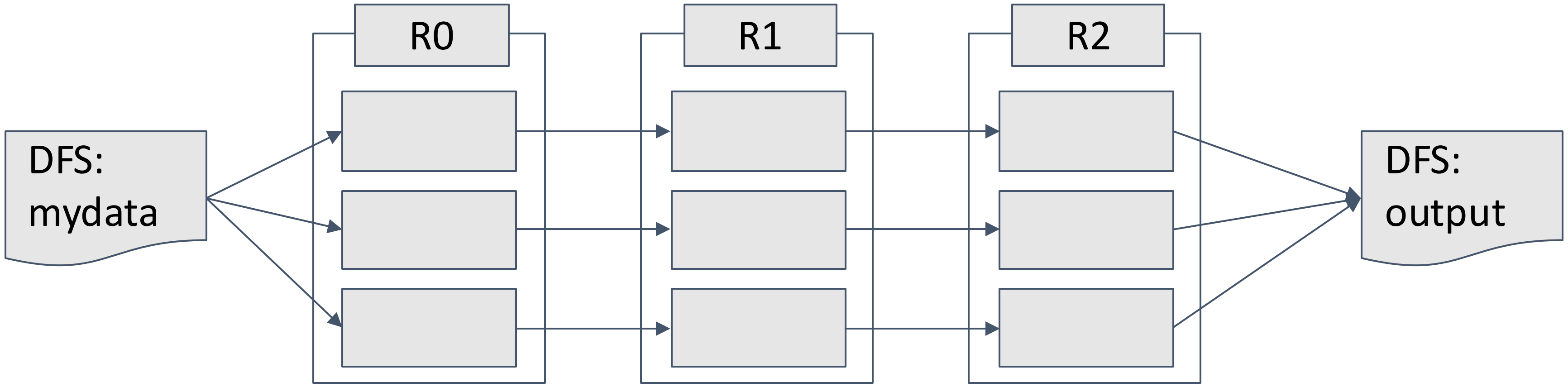
Spark does not actually apply transformations to RDD’s until it is required to do so (e.g., storing RDD to file system or returning a result to application)

Potential Benefits:

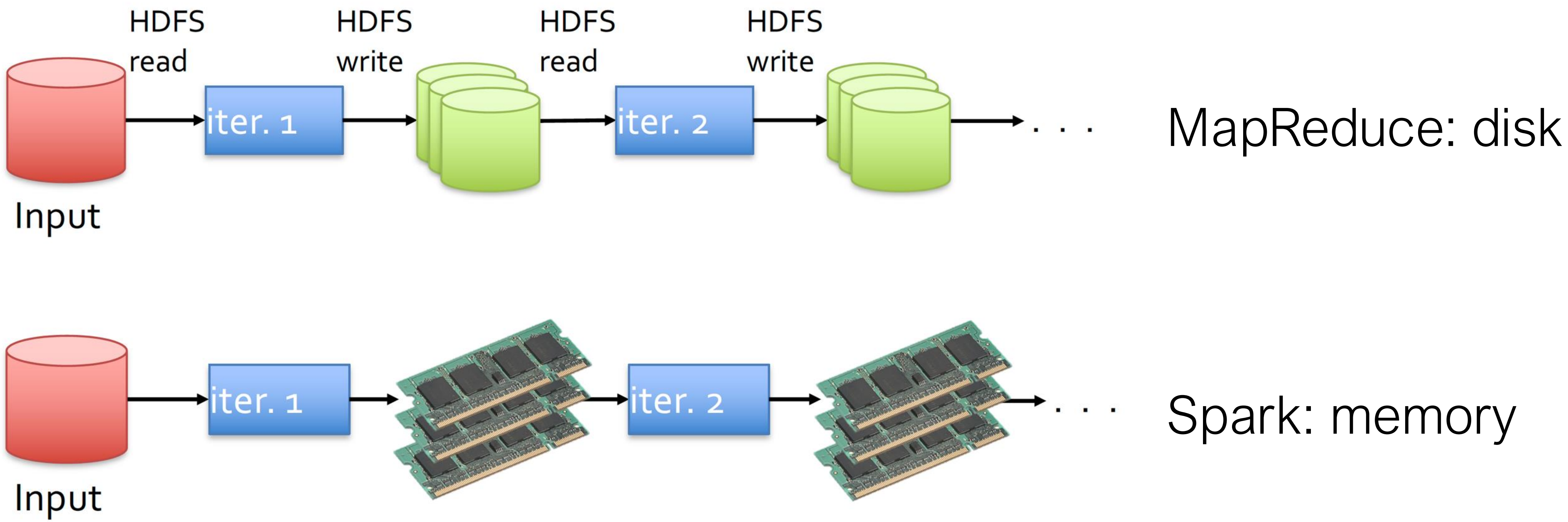
- Spark can analyze entire chain of operations and combining multiple operations to reduce unnecessary compiuations
- No immediate computation/memory usage; resources allocated only when needed
- Optimizes data shuffling and stages

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



Data Sharing in MapReduce vs Spark



This is why Spark is significantly faster for iterative algorithms

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

Mapper1 output → Disk (3x replicated)
Mapper2 output → Disk (3x replicated)
Mapper3 output → Disk (3x replicated)

RDD1 → Memory → RDD2 → Memory → RDD3
(Only checkpoint/persist if specified)

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