Scalability and Map Reduce Programming

CS 144 Web Application

TA: Jin Wang

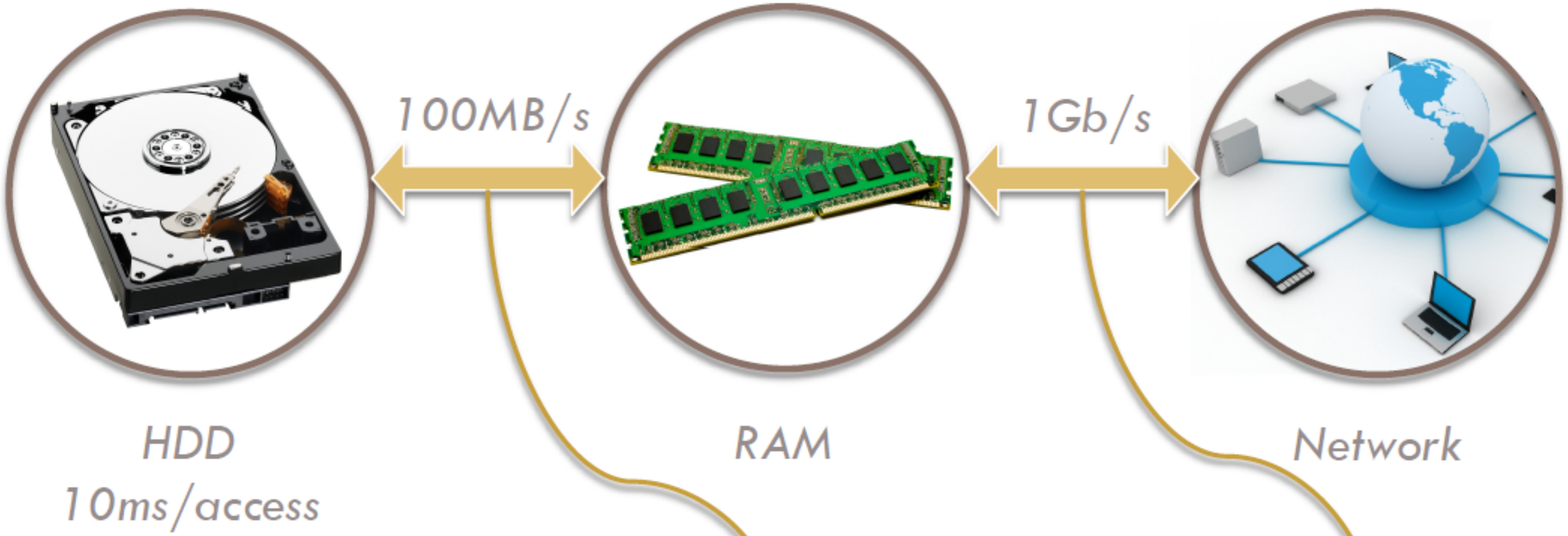
03/02/2018

Overview

- Review of foundations about scalability
- The Hadoop platform
- **Map Reduce Programming**

Scalability

How to estimate our server capabilities:

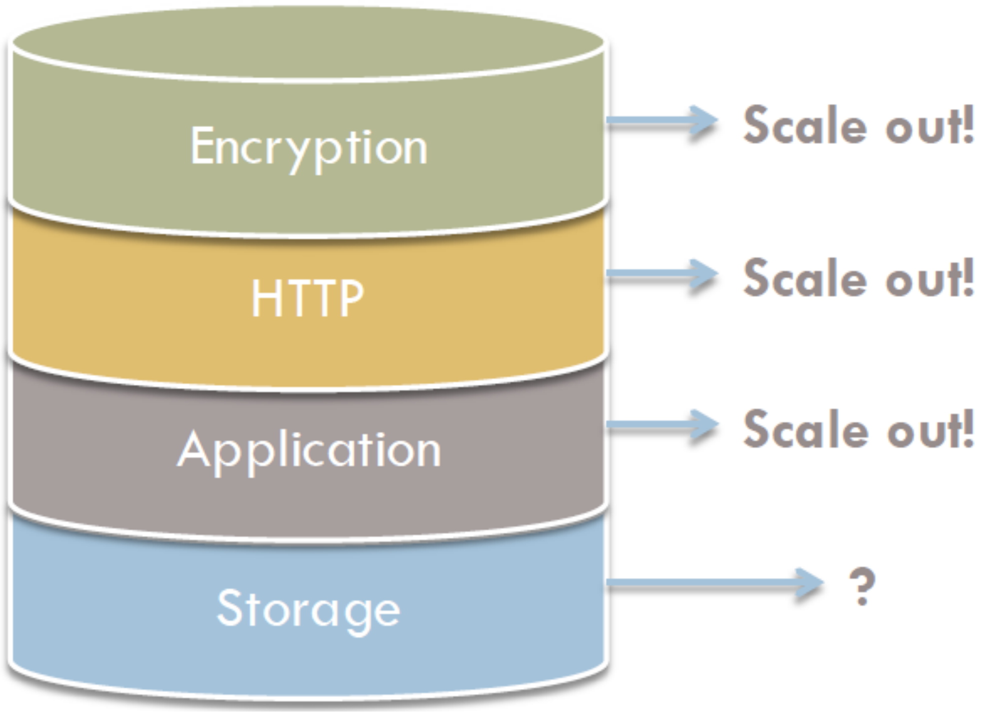


When a **static** page is 10KB	Sequential access	10,000 pages/s	Up to 10,000 pages/s
	Random access	100 pages/s	

Scaling Web Applications

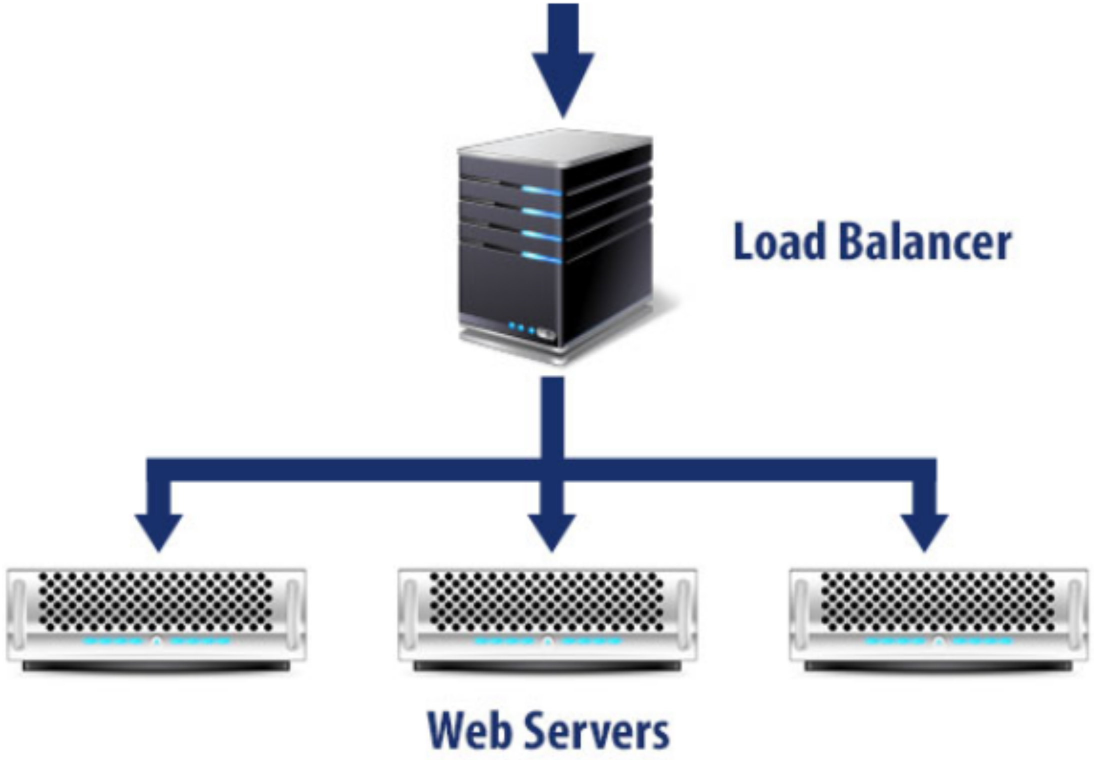
- Scale out
- Scale up

Web Server Architecture



Scaling Web Applications

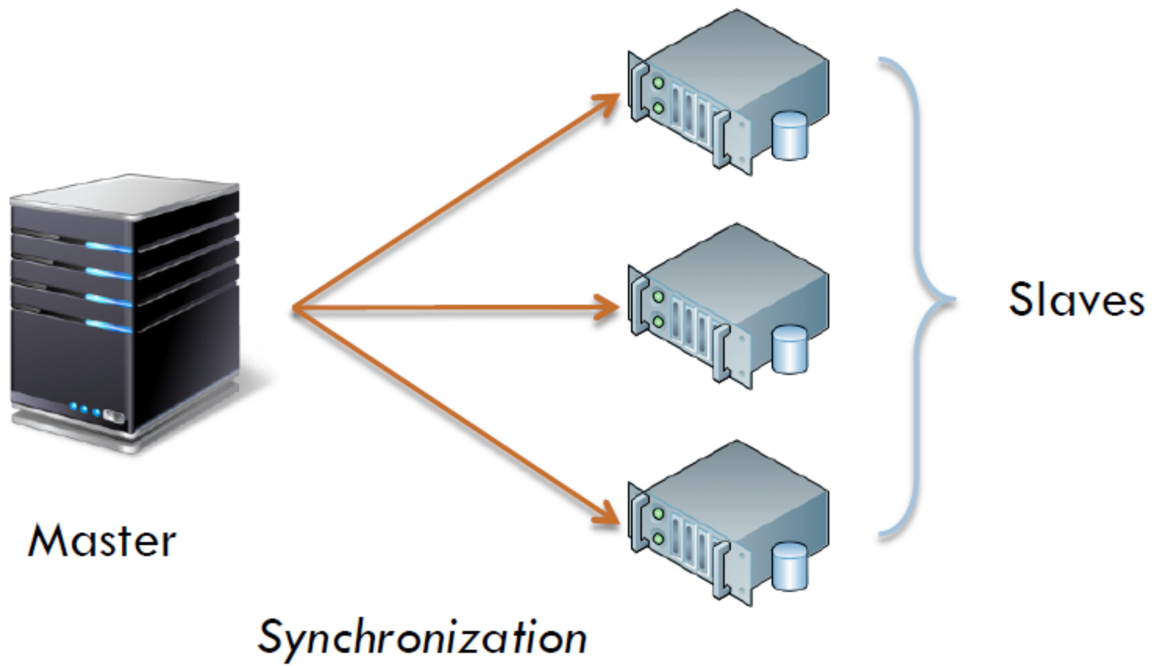
- The Load Balancer
  - TCP-NAT Request Distributor:
  DNS Round Robin, or software



- In our projects, Apache Tomcat is both in the application and HTTP layers.

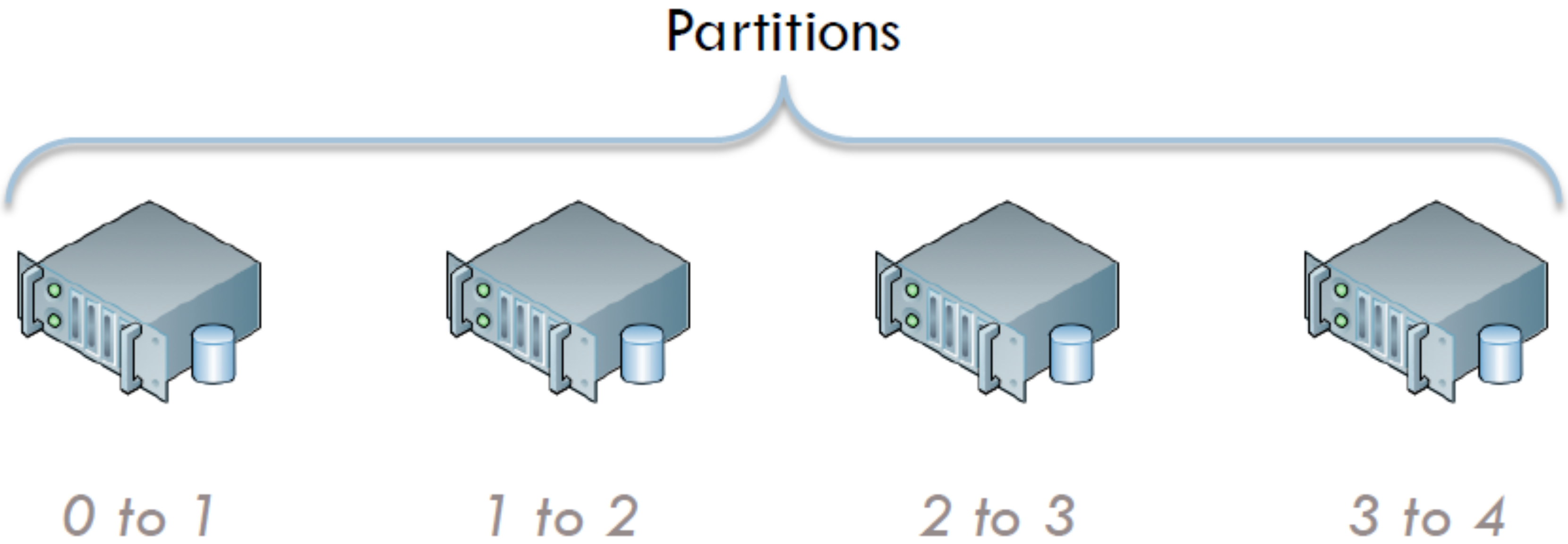
Scaling the Storage Layer

- Scenario 1: Read Only
  - Information doesn’t change. Clients only read data.
  - Use **replication**



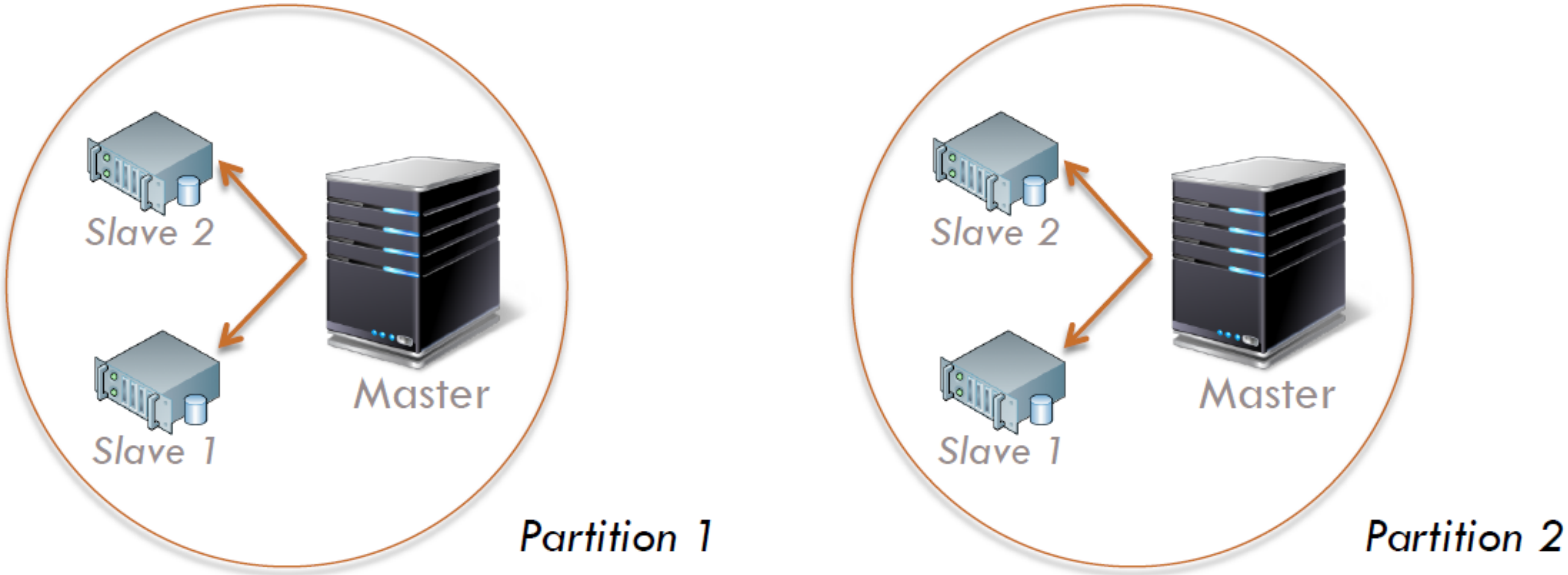
Scaling the Storage Layer

- *Scenario 2*: Local Read/Write
  - Reads and writes are scoped to individual users
  - Use **shard** or **partitioning**



Scaling the Storage Layer

- *Scenario 3*: Global Read/Write
  - Reads and writes are global, and all users can see everyone updates
  - Use **partitioning** and **replication**



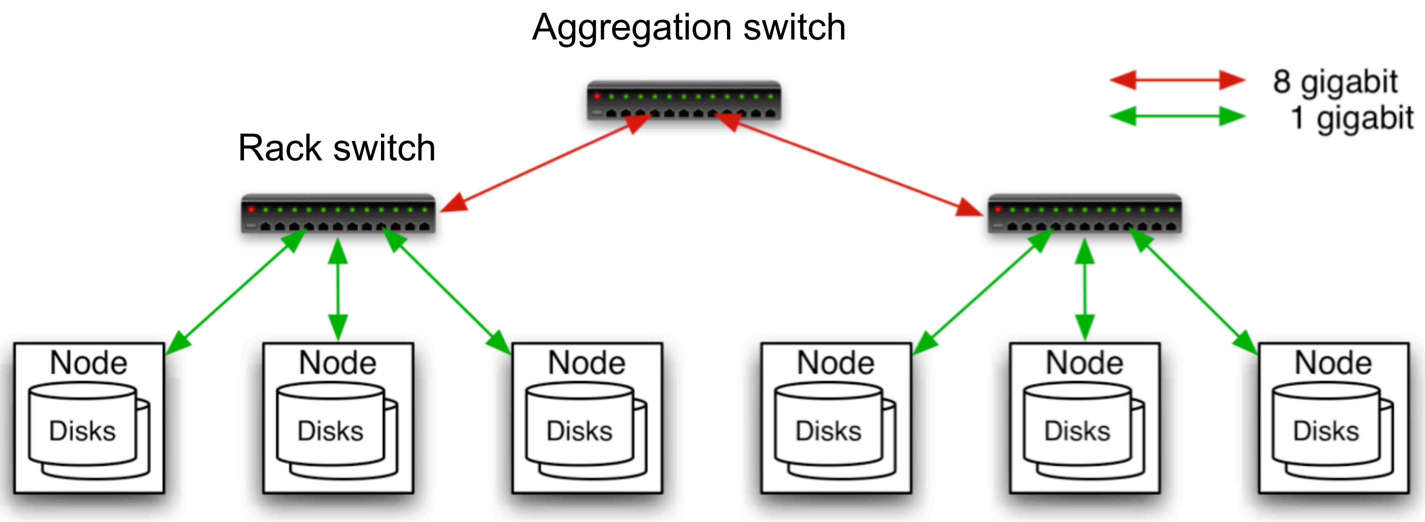
Introduction to Hadoop

- Download from hadoop.apache.org
- To install locally, unzip and set `JAVA_HOME`
- Docs: hadoop.apache.org/common/docs/current

- Three ways to write jobs:
  - Java API
  - Hadoop Streaming (for Python, Perl, etc)
  - Pipes API (C++)

Note: The following slides are borrowed from Prof. Tyson Condie

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs (Facebook):
  8-16 cores, 32 GB RAM, 8 × 1.5 TB disks, no RAID

Typical Hadoop Cluster

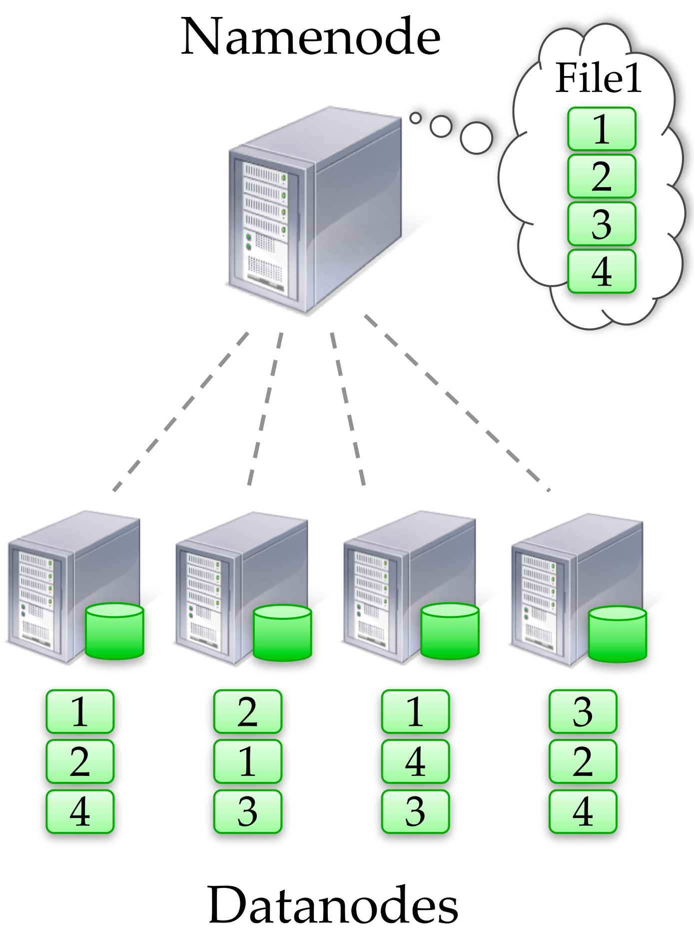
Hadoop Components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance

- MapReduce framework
  - Runs jobs submitted by users
  - Manages work distribution & fault-tolerance
  - Colocated with file system

Hadoop Distributed File System (HDFS)

- Files split into 128MB blocks
- Blocks replicated across several datanodes (often 3)
- Namenode stores metadata (file names, locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



What is MapReduce?

- Programming model for data-intensive computing on commodity clusters
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by Apache Hadoop project
  - Used by Facebook, Amazon, …

What is MapReduce Used For?

- At Google:
  - Originally: Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

MapReduce Programming Model

- Data type: key-value *records*

- Map function:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

- Reduce function:

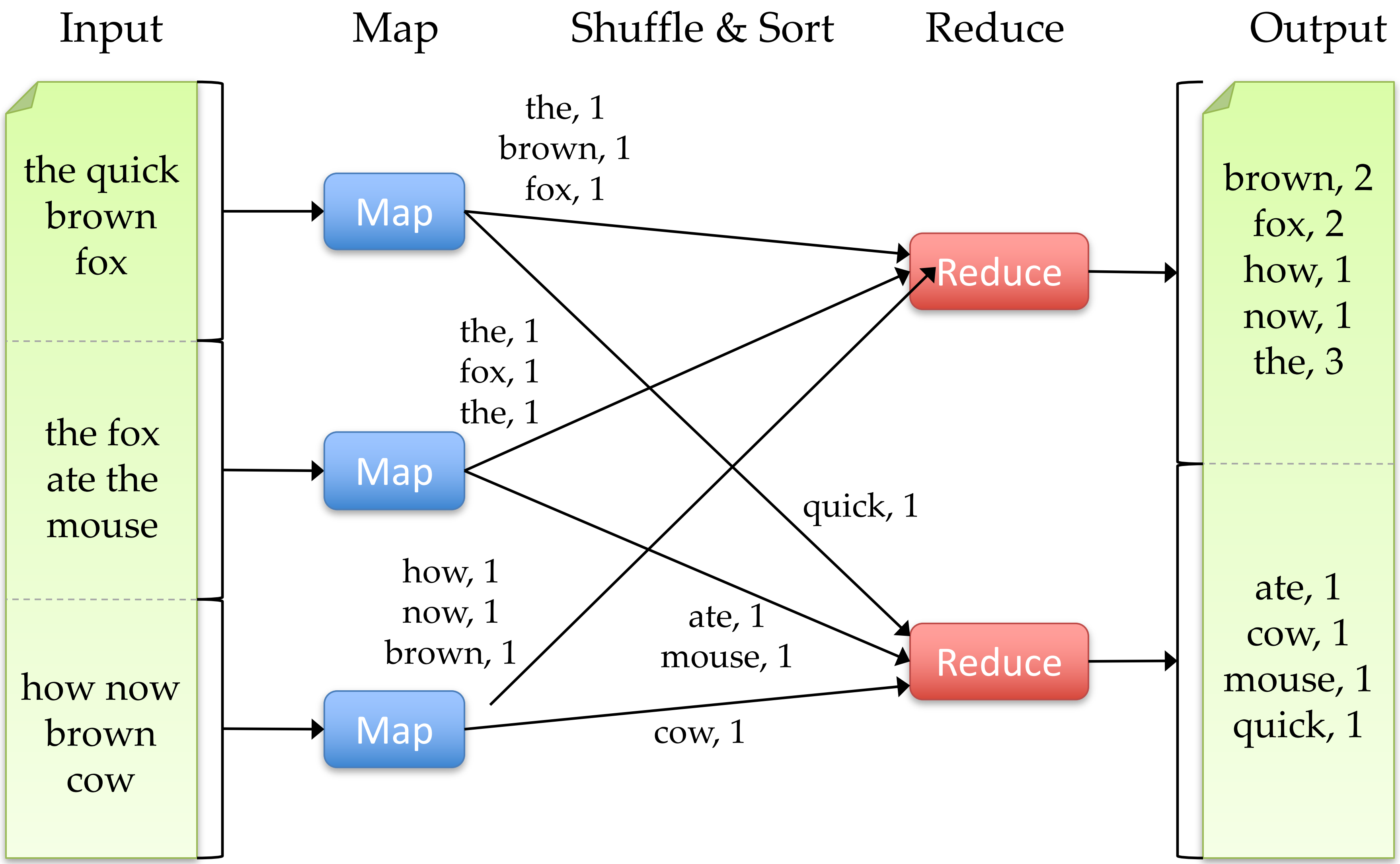
$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

Example: Word Count

```
def mapper(line):
    foreach word in line.split():
        output(word, 1)


def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution



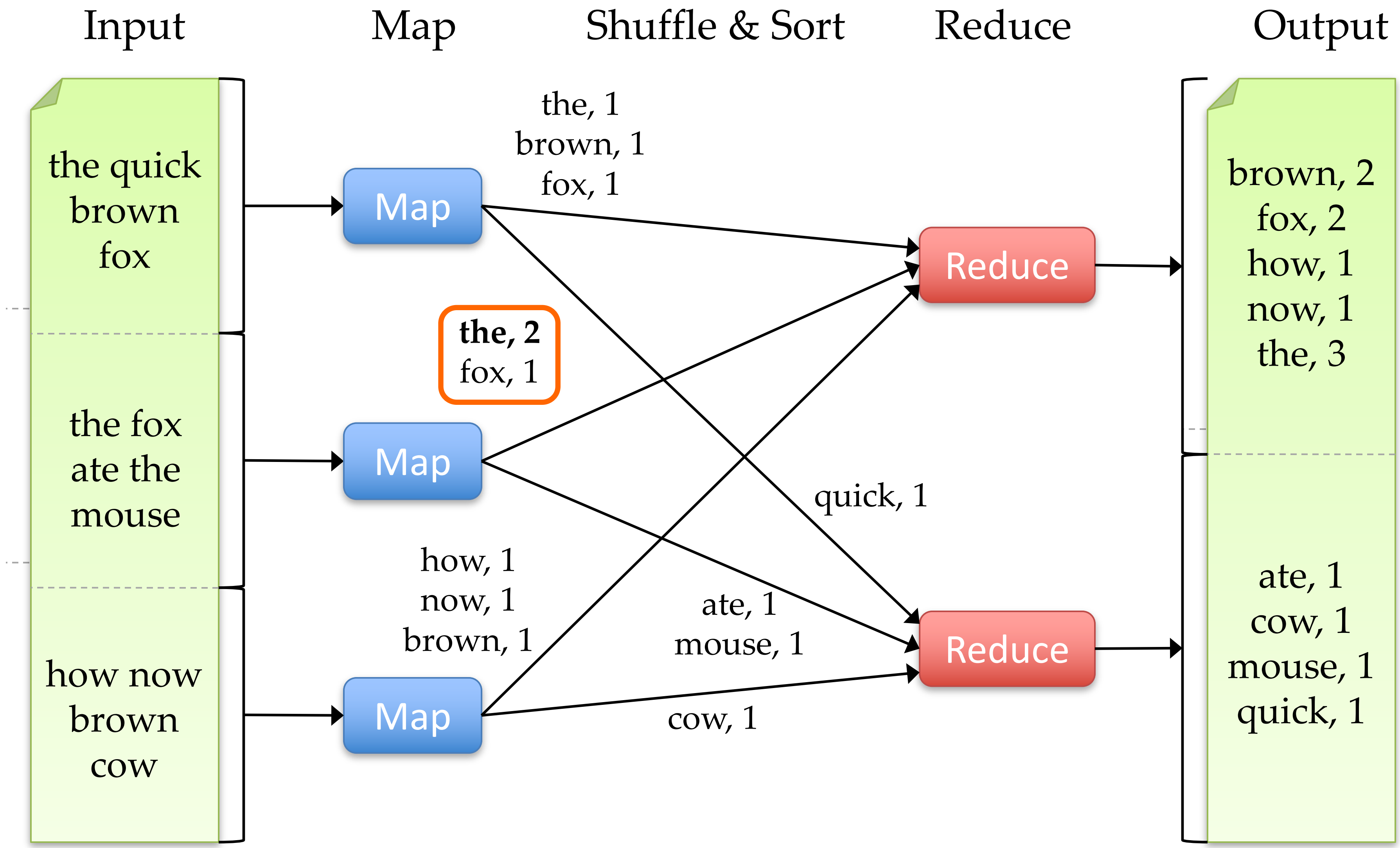
An Optimization: The Combiner

- Local reduce function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases amount of intermediate data

- Example: local counting for Word Count:

```
def combiner(key, values):
    output(key, sum(values))
```

Word Count with Combiner



MapReduce Execution Details

- Mappers preferentially scheduled on same node or same rack as their input block
  - Minimize network use to improve performance

- Mappers save outputs to local disk before serving to reducers
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes

Examples of Map Reduce Programming

1. Search

- **Input:** (lineNumber, line) records, a given pattern
- **Output:** lines matching the pattern
- **Map:**

```
if(line matches pattern):
   output(line)
```

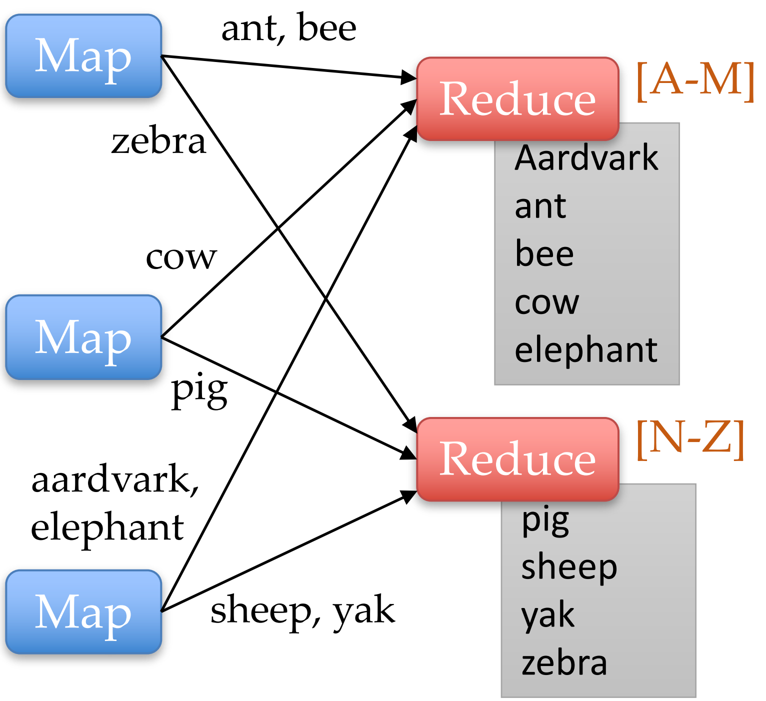
- **Reduce:** identity function
  - Alternative: no reducer (map-only job)

2. Sort

- **Input:** (key, value) records
- **Output:** same records, sorted by key

- **Map:** identity function
- **Reduce:** identify function

- **Trick:** Pick partitioning function p such that $k_1 < k_2 \Rightarrow p(k_1) < p(k_2)$



3. Inverted Index

- **Input:** (filename, text) records
- **Output:** list of files containing each word

- **Map:**

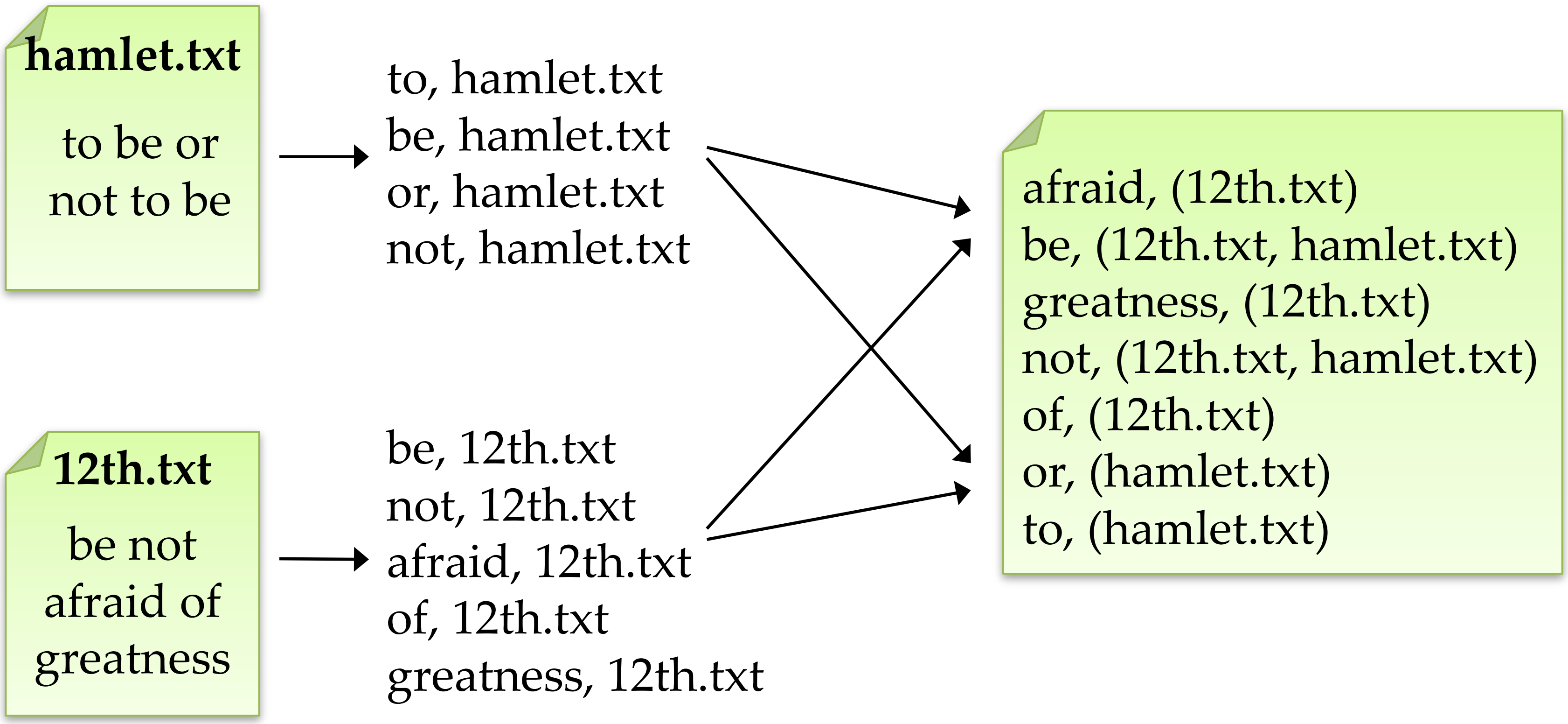
```
foreach word in text.split():
   output(word, filename)
```

- **Combine:** uniquify filenames for each word

- **Reduce:**

```
def reduce(word, filenames):
   output(word, sort(filenames))
```

Inverted Index Example



4. Most Popular Words

- **Input:** (filename, text) records
- **Output:** the 100 words occurring in most files

- Two-stage solution:
  - **Job 1:**
    - Create inverted index, giving (word, list(file)) records
  - **Job 2:**
    - Map each (word, list(file)) to (count, word)
    - Sort these records by count as in sort job

5. Numerical Integration

- **Input:** (start, end) records for sub-ranges to integrate
  - Can implement using custom InputFormat
- **Output:** integral of *f*(*x*) over entire range

- **Map:**

```
def map(start, end):
   sum = 0
   for(x = start; x < end; x += step):
      sum += f(x) * step
   output("", sum)
```

- **Reduce:**

```
def reduce(key, values):
    output(key, sum(values))
```

Hints for Map Reduce Programming

- Decide the way to partition the original data
- Avoid skewness
- Try to reduce the workload of network transmission