

ΚΑΤΑΝΕΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ

(Distributed Systems)

Lecture #12

MapReduce

ΤΙ ΕΙΝΑΙ ΤΟ MapReduce

- Πρόκειται για ένα (data-parallel) προγραμματιστικό μοντέλο, κατάλληλο για μαζική επεξεργασία πολύ μεγάλων όγκων δεδομένων σε φθηνούς κατανεμημένους πόρους με αυξημένη ανοχή λαθών (fault tolerance)
- Αναπτύχθηκε/προτάθηκε από την Google η οποία το χρησιμοποιεί εσωτερικά για υποστήριξη των μαζικά μεγάλου όγκου επεξεργασιών που κάνει καθημερινά (επεξεργάζεται περισσότερα από 100 PBytes τη μέρα)
- Η χρήση του στον υπόλοιπο κόσμο έγινε δημοφιλής μέσω της πρώτης αντίστοιχης ελεύθερης υλοποίησής του (Apache Hadoop project)
- Χρησιμοποιείται έντονα και από τους άλλους γνωστούς μεγάλους στο χώρο του cloud computing και γενικότερα του Internet (Yahoo!, Facebook, Amazon, ...)

ΠΟΥ ΧΡΗΣΙΜΟΠΟΙΕΙΤΑΙ

At Google:

- Index building for Google Search
- Article clustering for Google News
- Statistical machine translation

At Yahoo!:

- Index building for Yahoo! Search
- Spam detection for Yahoo! Mail

At Facebook:

- Ad optimization
- Spam detection

ΠΟΥ ΧΡΗΣΙΜΟΠΟΙΕΙΤΑΙ

Ερευνητικά:

- Analyzing Wikipedia conflicts (PARC)
Natural language processing (CMU)
- Bioinformatics (Maryland)
- Particle physics (Nebraska)
- Ocean climate simulation (Washington)
- High energy physics (CERN)
-

MapReduce & Cloud Computing



HOW MUCH DATA?

- 7 billion people
- Google processes 100 PB/day; 3 million servers
- Facebook has 300 PB + 500 TB/day; 35% of world's photos
- YouTube 1000 PB video storage; 4 billion views/day
- Twitter processes 124 billion tweets/year
- SMS messages – 6.1T per year
- US Cell Calls – 2.2T minutes per year
- US Credit cards - 1.4B Cards; 20B transactions/year

6

MapReduce & Cloud Computing

Χαρακτηριστικά του MapReduce που καθιστούν ιδανική τη χρήση του σε Cloud περιβάλλοντα και το δένουν άρρηκτα με το Cloud computing:

- Commodity nodes (cheap, but unreliable)
- Commodity network (low bandwidth)
- Automatic fault-tolerance (fewer admins)

Επεκτασιμότητα (scalability) σε μεγάλους και τεράστιους όγκους (σύνολα) δεδομένων:

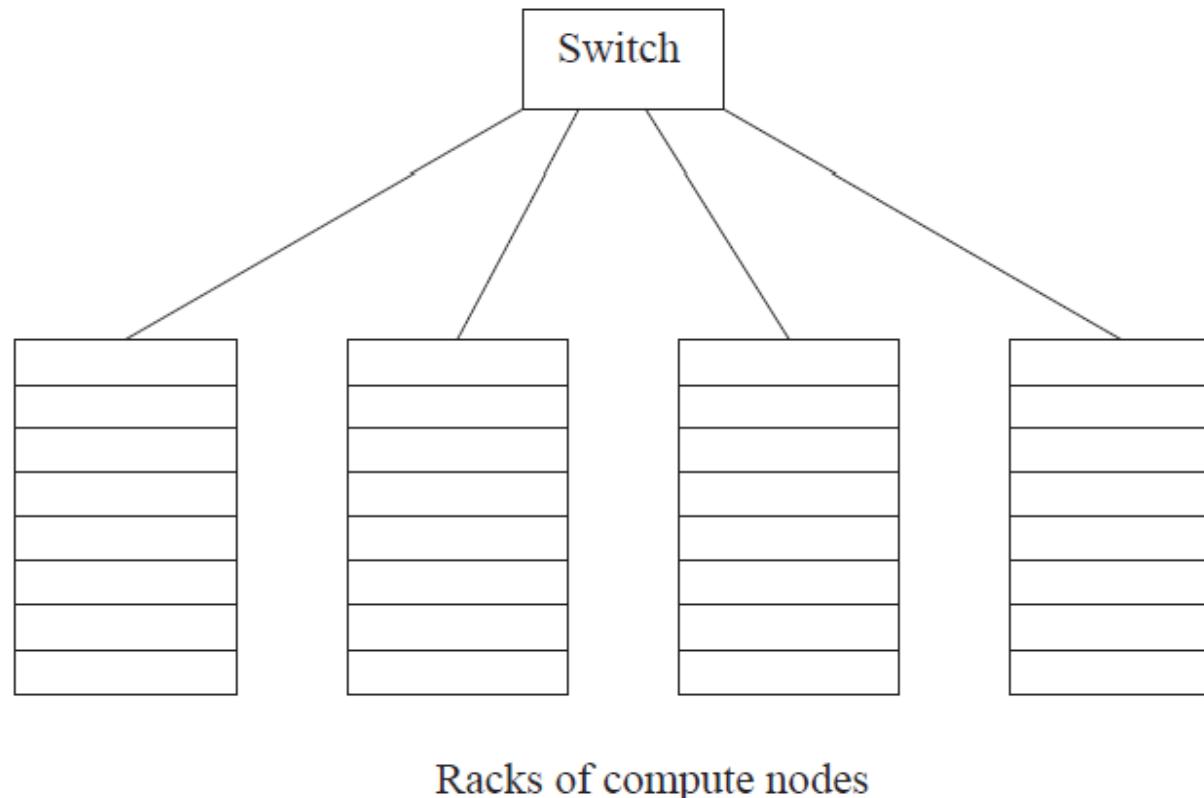
- Scan 100 TB on 1 node @ 50 MB/s = 24 days
- Scan on 1000-node cluster = 35 minutes

MapReduce & Cloud Computing

1. Components fail, and the more components, such as (cheap) compute nodes and interconnection networks, a system has, the more frequently something in the system will not be working at any given time.
 - Mean time between failures for 1 node = 3 years
 - MTBF for 1000 nodes = 1 day
 - **Solution: Restrict programming model so you can efficiently “build-in” fault-tolerance**
2. Commodity network = low bandwidth
 - **Solution: Push computation to the data**

MapReduce & Cloud Computing

Σε τέτοιου είδους συστήματα οι δύο βασικές περιπτώσεις αποτυχιών είναι η αποτυχία-απώλεια σε επίπεδο ενός node (**loss of a single node** - e.g., the disk at that node crashes) και η αποτυχία-απώλεια σε επίπεδο rack (**loss of an entire rack** - e.g., the network connecting its nodes to each other and to the outside world fails).



Distributed File Systems

Πολλοί σημαντικοί υπολογισμοί απαιτούν για την εκτέλεσή τους λεπτά ή ακόμα και ώρες πάνω σε εκατοντάδες ή/και χιλιάδες κόμβους. Εάν έπρεπε να σταματήσουμε και να επανεκκινήσουμε τον υπολογισμό κάθε φορά που μία συνιστώσα αποτυγχάνει, τότε ο υπολογισμός είναι πιθανό να μην εκτελεστεί ποτέ επιτυχώς.

Η λύση στο πρόβλημα αυτό κινείται σε δύο βασικούς άξονες:

- **Τα αρχεία πρέπει να αποθηκεύονται πλεονασματικά** (σε πολλαπλά αντίγραφα). Αν δεν αποθηκεύσουμε πολλαπλά ένα αρχείο σε διαφορετικούς κόμβους, τότε αν ένας κόμβος πέσει, όλα τα αρχεία του κόμβου αυτού δεν θα είναι διαθέσιμα μέχρι ο κόμβος να επανέλθει ή να αντικατασταθεί. Εάν επίσης δεν έχουμε κάνει καθόλου back up τα αρχεία μας, και ο δίσκος αποτύχει, τα αρχεία θα χαθούν για πάντα.
- **Οι υπολογισμοί επίσης πρέπει να διαχωρίζονται/αποδομούνται σε** (ανεξάρτητα μεταξύ τους) tasks κατά τέτοιο τρόπο, ώστε αν ένα task αποτύχει κατά την εκτέλεσή του, να μπορεί να επανεκκινηθεί χωρίς να επηρεάζει άλλα tasks.

Distributed File Systems

Για να αξιοποιήσουμε λοιπόν τις δυνατότητες των συστημάτων συστοιχιών υπολογιστών (clusters), τα συστήματα αρχείων πρέπει να συμπεριφέρονται λίγο διαφορετικά απ' ότι τα συμβατικά συστήματα αρχείων σε μεμονωμένους υπολογιστές. Ο νέος αυτός τύπος συστήματος αρχείων λέγεται πλέον ***distributed file system / DFS*** (ένα όρος που υπάρχει και από παλαιότερα με λίγο διαφορετική ωστόσο ερμηνεία), και τυπικά χρησιμοποιείται στο ακόλουθο πλαίσιο:

- **Όταν τα αρχεία μας είναι πολύ μεγάλου / τεραστίου μεγέθους** (π.χ. terabytes etc.). Αν έχουμε μόνο μικρά αρχεία, δεν έχει νόημα-εφαρμογή η χρήση ενός DFS.
- **Όταν τα αρχεία μας δεν ενημερώνονται/τροποποιούνται συχνά.** Π.χ. σε ένα σύστημα κρατήσεων αεροπορικής εταιρείας η έννοια του DFS δεν έχει εφαρμογή, ακόμα και αν έχουμε πολύ μεγάλα σε όγκο δεδομένα, λόγω του πολύ συχνού ρυθμού αλλαγής τους.

Distributed File Systems

- ✓ Τα αρχεία διαχωρίζονται σε **αλυσίδες (chunks)** οι οποίες είναι τυπικά μεταξύ **64 και 128 Megabytes**. Τα chunks αντιγράφονται π.χ. τρεις φορές, σε τρεις διαφορετικούς κόμβους.
- ✓ Επιπρόσθετα, οι κόμβοι οι οποίοι κρατούν αντίγραφα ενός chunk, θα πρέπει να βρίσκονται **σε διαφορετικά racks**, έτσι ώστε αν χαθεί ένα rack να μην χάνονται όλα τα αντίγραφα κάποιου chunk.
- ✓ Τυπικά, τόσο το μέγεθος της αλυσίδας όσο ο βαθμός της αντιγραφής προσδιορίζεται από το χρήστη
- ✓ Για να είναι γνωστό που βρίσκονται τα chunks του κάθε αρχείου, υπάρχει ένα άλλο μικρό αρχείο (**master node or name node**) που κρατά την πληροφορία για κάθε αρχείο.
- ✓ Ο master node επίσης αντιγράφεται και ένας **κατάλογος (directory)** για το σύστημα αρχείων στο σύνολό του γνωρίζει που βρίσκονται τα αντίγραφα.
- ✓ Ο κατάλογος (directory) αυτός μπορεί επίσης να αντιγραφεί και όλοι οι συμμετέχοντες μπορούν να ξέρουν που είναι τα αντίγραφά του.

MapReduce & Cloud Computing

- ❖ Όταν προτάθηκε από την Google (2004) παρουσιάστηκαν demonstrations με λειτουργία σε clusters από 1500 έως και 4000 περίπου nodes.
- ❖ Γενικά, μπορεί να αξιοποιήσει για μαζική κατανεμημένη επεξεργασία clusters με 1000 έως και 10000 nodes (τέτοιου μεγέθους είναι τα clusters που χρησιμοποιούνται πλέον για εσωτερική υποστήριξη – μαζική κατανεμημένη επεξεργασία – από τους μεγάλους παρόχους στο χώρο του cloud computing).
- ❖ Οι δυνατότητες λειτουργίας με πολλαπλές αποτυχίες (fault tolerance) που παρέχει είναι εντυπωσιακές. Ένα cluster με 1800 nodes μπορεί να μείνει εν λειτουργίᾳ ακόμα και αν πέσουν οι 1600.

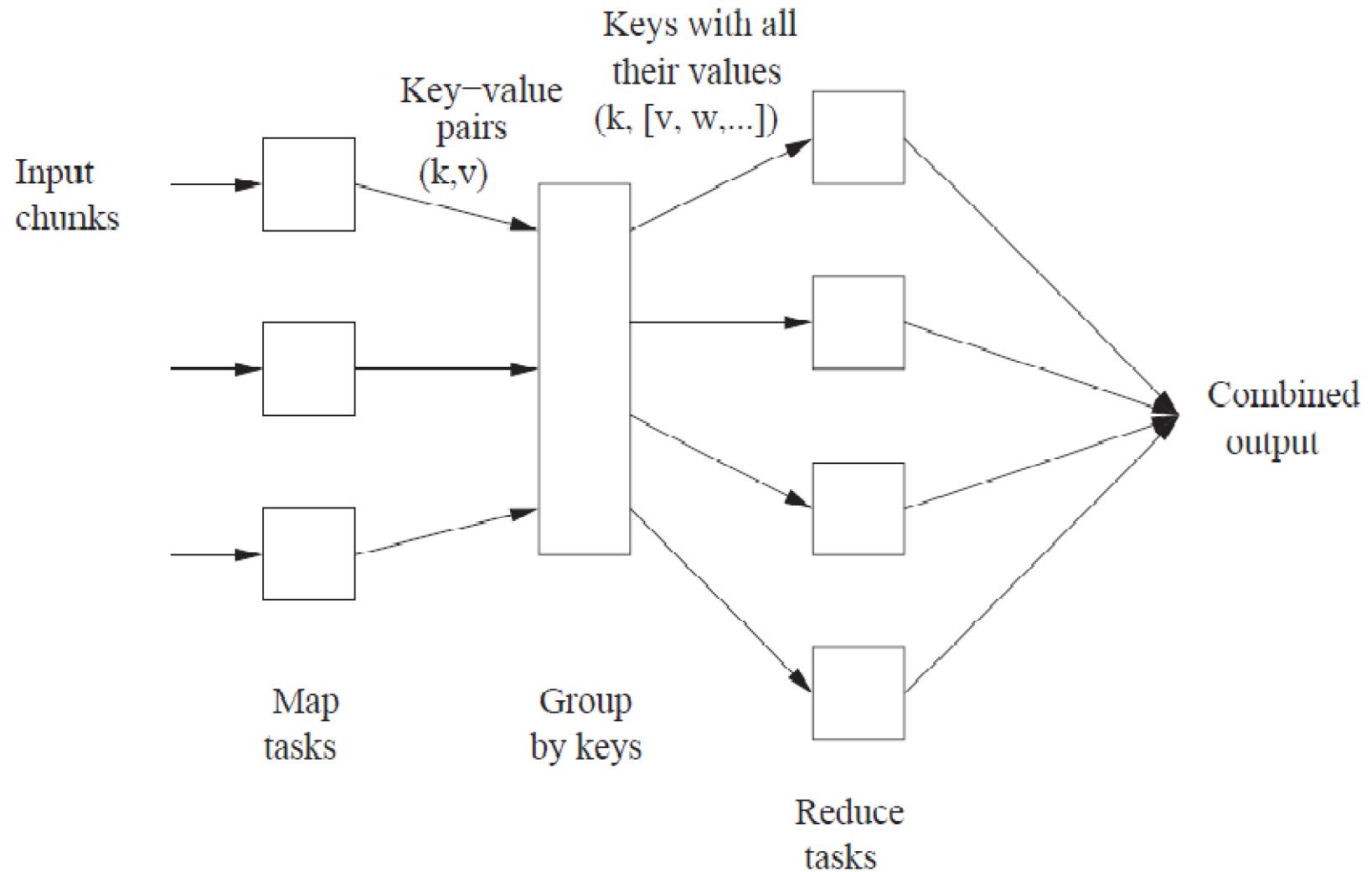
MR - Basic Operations

Συνοπτικά, ένας υπολογισμός στο MapReduce εκτελείται ως εξής:

1. Ένας αριθμός από **Map tasks** αναλαμβάνει τα καθένα έναν αριθμό από chunks από το κατανεμημένο σύστημα αρχείων. Τα Map tasks μετασχηματίζουν κάθε chunk σε μία ακολουθία από key-value pairs. Ο τρόπος με τον οποίο δημιουργούνται τα key-value pairs από τα δεδομένα εισόδου (chunks) προσδιορίζεται από τον κώδικα που γράφει ο χρήστης στη ***Map function***.
2. Τα key-value pairs του κάθε Map task συγκεντρώνονται από τον ***master controller*** και ταξινομούνται ως προς το κλειδί. Τα κλειδιά διανέμονται στη συνέχεια στα Reduce tasks, έτσι ώστε όλα τα key-value pairs με το ίδιο κλειδί να πηγαίνουν στο ίδιο Reduce task.
3. Τα **Reduce tasks** επεξεργάζονται στο τέλος ένα κλειδί τη φορά, και συνδυάζουν όλες τις τιμές που έχουν συσχετιστεί με αυτό το κλειδί με ένα συγκεκριμένο τρόπο. Ο τρόπος συνδυασμού προσδιορίζεται από τον κώδικα που γράφει ο χρήστης για στη ***Reduce function***.

Grouping by Key

- ✓ Αφού όλα τα Map tasks έχουν ολοκληρωθεί επιτυχώς, τα key-value pairs ομαδοποιούνται με βάση το κλειδί, και οι τιμές που συσχετίζονται με κάθε κλειδί τοποθετούνται σε μια αντίστοιχη λίστα.
- ✓ Η ομαδοποίηση γίνεται από το σύστημα, ανεξάρτητα από τι κάνουν τα Map και Reduce tasks. Ο master controller επίσης γνωρίζει πόσα Reduce tasks θα σχηματιστούν στη συνέχεια, ας πούμε **r** τέτοια tasks.
- ✓ Η τιμή του **r** προσδιορίζεται τυπικά από το χρήστη. Στη συνέχεια ο master controller εφαρμόζει μια hash function πάνω στα κλειδιά και παράγει για κάθε κλειδί ένα bucket number από το **0 ως το r - 1**.
- ✓ Έτσι κάθε κλειδί που προκύπτει ως output από ένα Map task κατακερματίζεται και το key-value pair του τοποθετείται σε ένα από **r** τυπικά αρχεία.
- ✓ Κάθε ένα από αυτά τα αρχεία προορίζεται για ένα αντίστοιχο Reduce task.



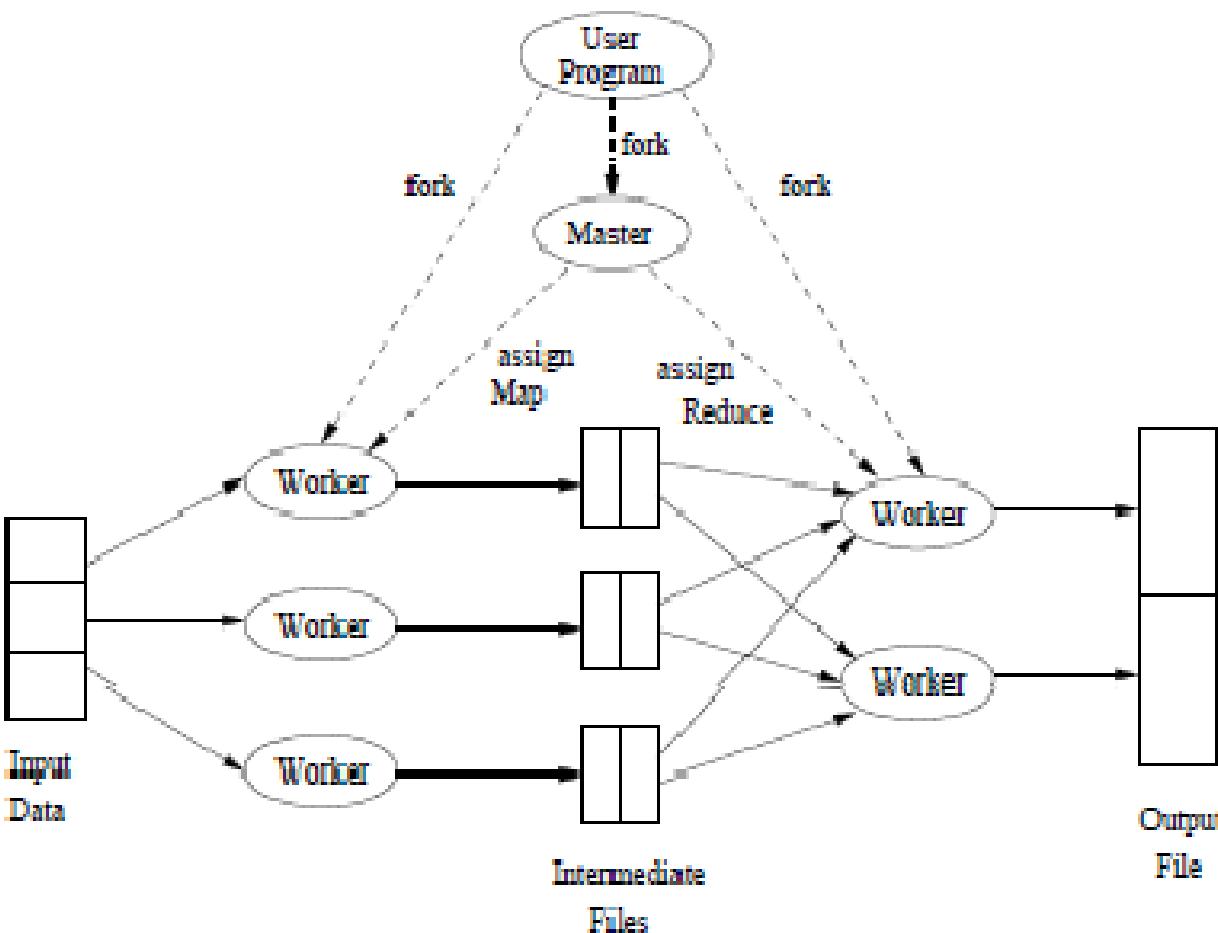


Figure 2.3: Overview of the execution of a MapReduce program

of Reduce Tasks

Εάν θέλουμε το μέγιστο δυνατό παραλληλισμό, τότε θα μπορούσαμε να χρησιμοποιήσουμε **ένα Reduce task για την εκτέλεση ενός reducer**, δηλαδή ενός μεμονωμένου κλειδιού και της σχετιζόμενης λίστας τιμών του. Επιπρόσθετα, θα μπορούσαμε να εκτελέσουμε **κάθε Reduce task σε ξεχωριστό κόμβο**, έτσι ώστε όλα να μπορούν να εκτελεστούν πραγματικά παράλληλα.

Ωστόσο οι παραπάνω επιλογές δεν είναι πάντα οι καλύτερες:

- Ένα πρόβλημα έχει να κάνει με το overhead που πληρώνουμε για κάθε task που δημιουργείται, με αποτέλεσμα, να προτιμάμε να διατηρήσουμε τον αριθμό Reduce tasks μικρότερο από τον αριθμό των διαφορετικών κλειδιών.
- Επιπρόσθετα, θυμηθείτε ότι κάθε Map task δημιουργεί ένα αρχείο για κάθε Reduce task, και όλα αυτά τα αρχεία διακινούνται στη συνέχεια στο δίκτυο.
- Συχνά επίσης υπάρχουν/δημιουργούνται πολύ περισσότερα κλειδιά από τον αριθμό των κόμβων που έχουμε στη διάθεσή μας, ούτως ώστε δεν μπορούμε να περιμένουμε αποδοτικότερη λειτουργία με τόσο μεγάλο αριθμό από Reduce tasks.

of Reduce Tasks

Ένα επιπλέον πρόβλημα έχει να κάνει με το γεγονός ότι συχνά υπάρχουν σημαντικές διαφορές στα μεγέθη των λιστών τιμών του κάθε κλειδιού, με αποτέλεσμα, ο κάθε reducer μην τελειώνει στον ίδιο χρόνο με οποιονδήποτε άλλον. Εάν φτιάξουμε για κάθε reducer ξεχωριστό Reduce task, τότε αναμένεται τα διάφορα Reduce Tasks να εμφανίσουν σημαντικές αποκλίσεις μεταξύ τους ως προς το χρόνο ολοκλήρωσης:

- Μπορούμε να ελαττώσουμε το μέγεθος της απόκλισης, χρησιμοποιώντας **λιγότερα Reduce tasks από reducers**. Εάν επιπλέον τα κλειδιά διανέμονται τυχαία στα Reduce tasks, μπορούμε να αναμένουμε ότι τελικά θα υπάρχει μια εξισορρόπηση στο χρόνο ολοκλήρωσης όλων.
- Μπορούμε να ελαττώσουμε ακόμα περισσότερο το μέγεθος της απόκλισης, χρησιμοποιώντας **περισσότερα Reduce tasks από υπολογιστικούς κόμβους**. Με αυτόν τον τρόπο, μεγάλα σε χρόνο Reduce tasks μπορεί να απασχολούν έναν κόμβο σχεδόν πλήρως ενώ άλλα μικρότερα Reduce tasks να τρέχουν μαζί σε έναν άλλο κόμβο.

MapReduce Programming Model

$\text{list}\langle T_{\text{in}} \rangle \rightarrow \text{list}\langle T_{\text{out}} \rangle$

- Data type: key-value *records*

$\text{list}\langle (K_{\text{in}}, V_{\text{in}}) \rangle \rightarrow \text{list}\langle (K_{\text{out}}, V_{\text{out}}) \rangle$

MapReduce Programming Model

Map function:

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

Reduce function:

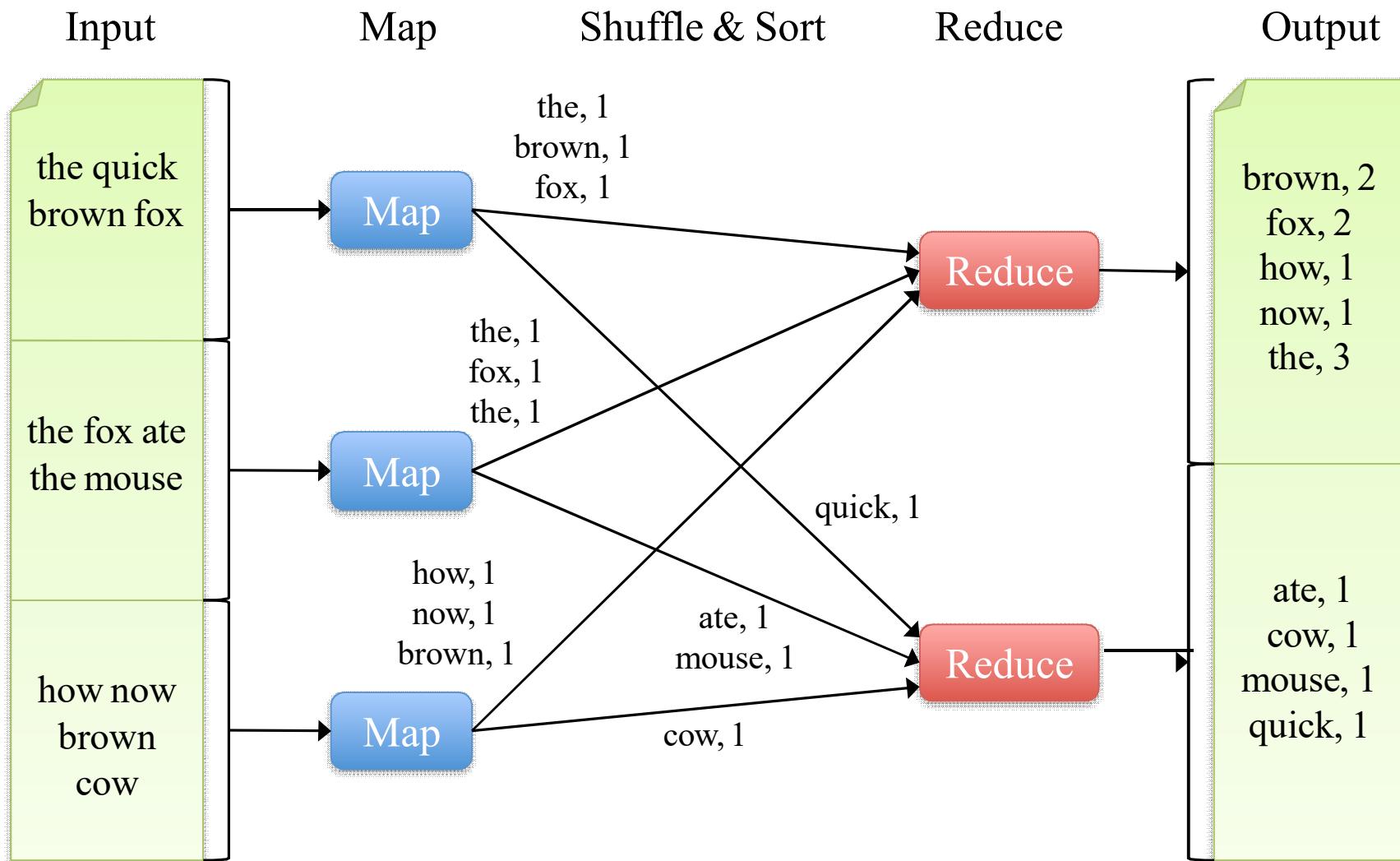
$$(K_{inter}, \text{list} <V_{inter}>) \rightarrow \text{list} <(K_{out}, V_{out})>$$

Example: Word Count

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

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

Example: Word Count

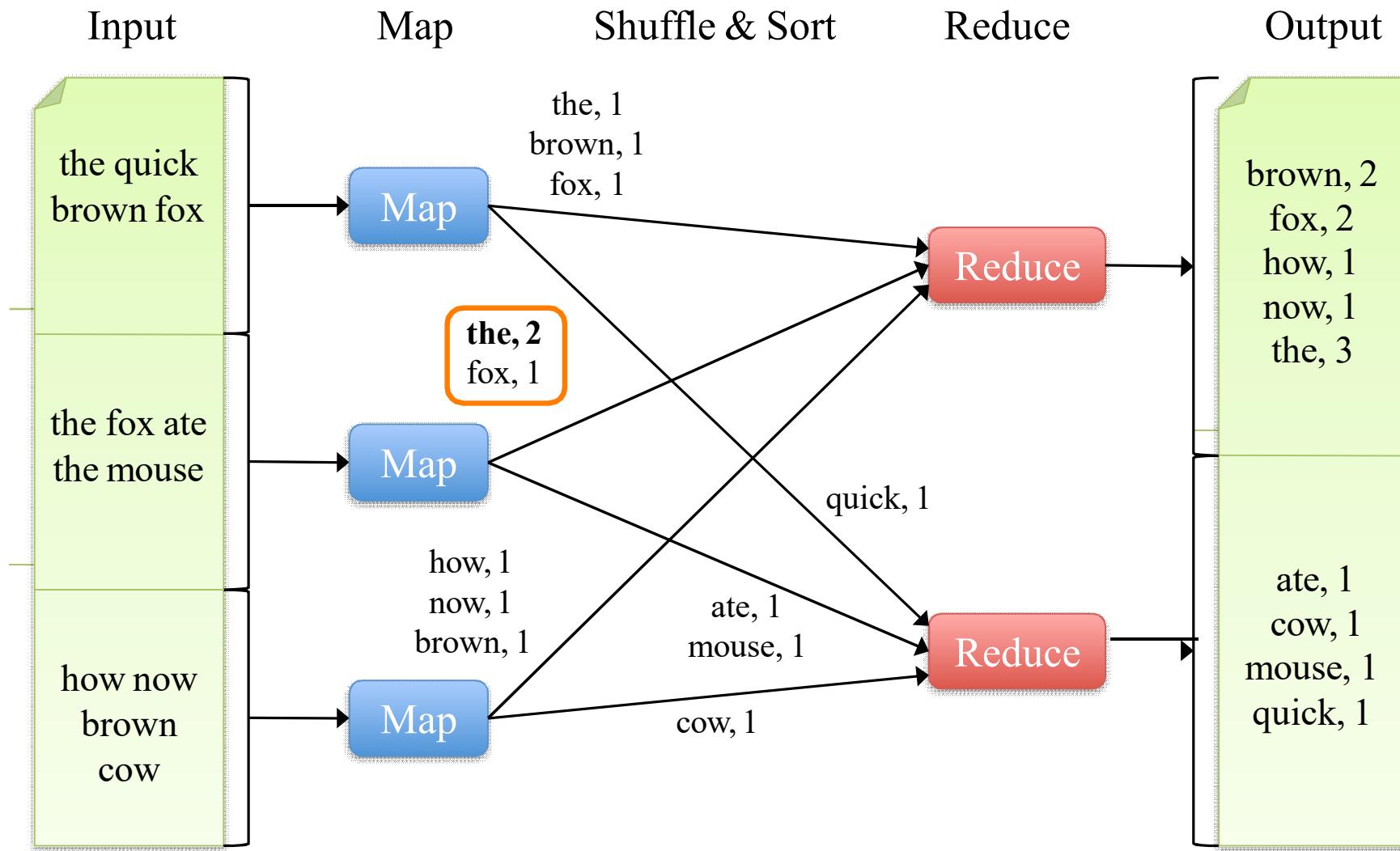


Optimization: Combiner

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

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

Example: Word Count + Combiner



MapReduce Execution Details

- Data stored on compute nodes
- 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
 - Efficient recovery when a reducer crashes
 - Allows more flexible mapping to reducers

Fault Tolerance in MapReduce

1. If a task crashes:

- Retry on another node
 - OK for a map because it had no dependencies
 - OK for reduce because map outputs are on disk
- If the same task repeatedly fails, fail the job or ignore that input block

➤ Note: For the fault tolerance to work, *user tasks must be idempotent and side-effect-free*

Fault Tolerance in MapReduce

2. If a node crashes:

- Relaunch its current tasks on other nodes
- Relaunch any maps the node previously ran
 - Necessary because their output files were lost along with the crashed node

Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):
 - Launch second copy of task on another node
 - Take the output of whichever copy finishes first, and kill the other one
- Critical for performance in large clusters (many possible causes of stragglers)

Takeaways

- By providing a restricted programming model, MapReduce can control job execution in useful ways:
 - Parallelization into tasks
 - Placement of computation near data
 - Load balancing
 - Recovery from failures & stragglers

1. Sort

- **Input:** (key, value) records
 - **Output:** same records, sorted by key
 - **Map:** identity function
 - **Reduce:** identify function
 - **Trick:** Pick partitioning function p such that $k_1 < k_2 \Rightarrow p(k_1) < p(k_2)$
-
- ```
graph LR; Map1[Map] -- "ant, bee" --> R1[Reduce A-M]; Map1 -- "zebra" --> R1; Map1 -- "cow" --> R1; Map2[Map] -- "pig" --> R2[Reduce N-Z]; Map3[Map] -- "aardvark, elephant" --> R1; Map4[Map] -- "sheep, yak" --> R2;
```

## 2. Search

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

```
if (line matches pattern):
 output(filename, line)
```
- **Reduce:** identity function
  - Alternative: no reducer (map-only job)

### 3. Inverted Index

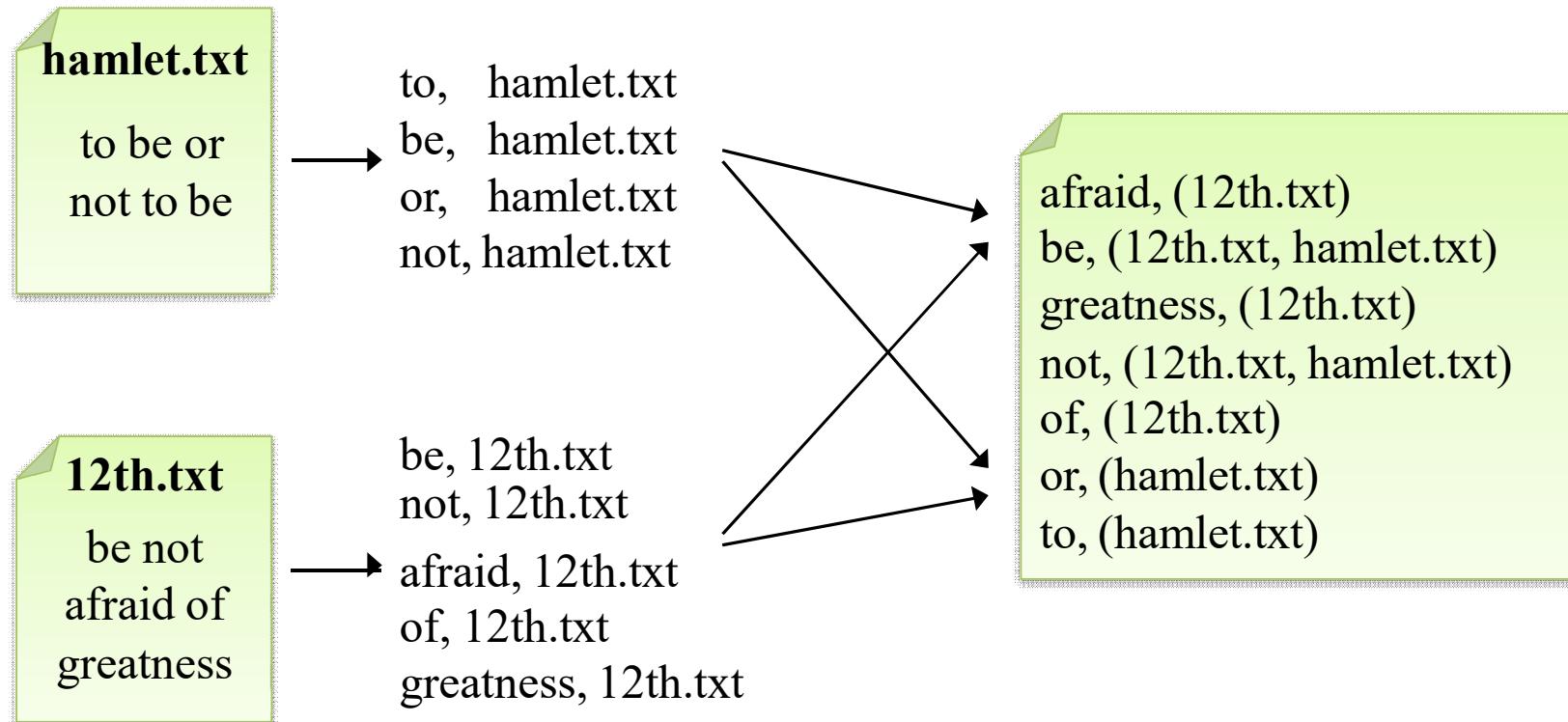
- **Input:** (filename, text) records
- **Output:** list of files containing each word
- **Map:**

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

- **Combine:** remove duplicates
- **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

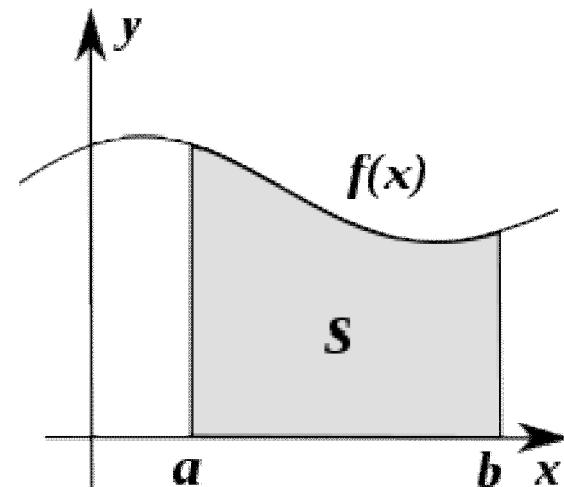
# 5. Numerical Integration

- **Input:** (start, end) records for sub-ranges to integrate\*
- **Output:** integral of  $f(x)$  over entire range
- **Map:**

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

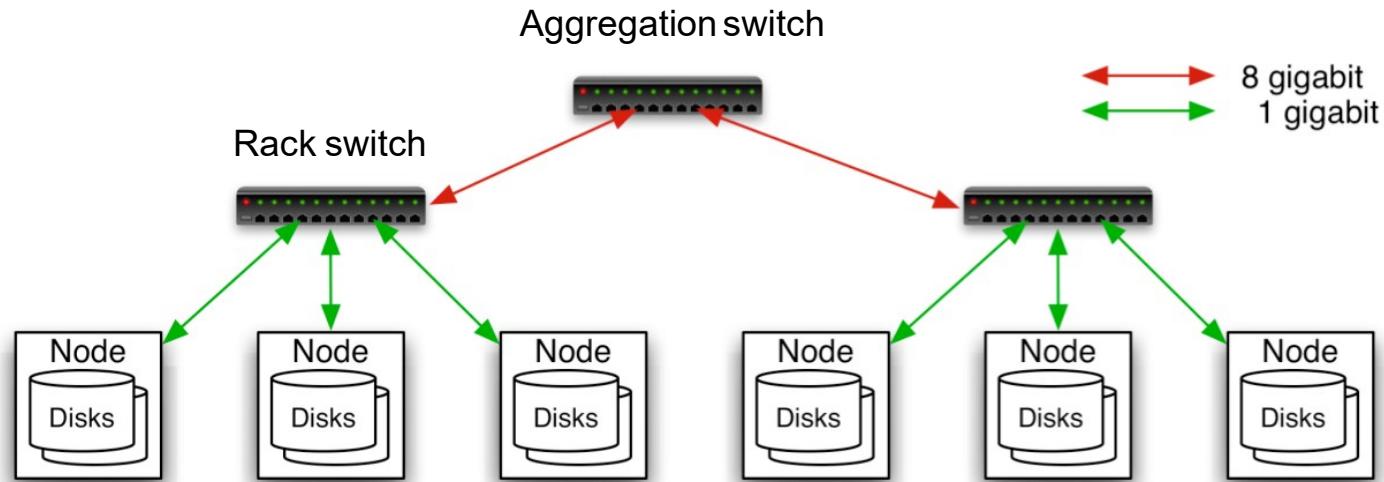
- **Reduce:**

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



\*Can implement using custom InputFormat

# Typical Hadoop cluster



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

# Typical Hadoop Cluster



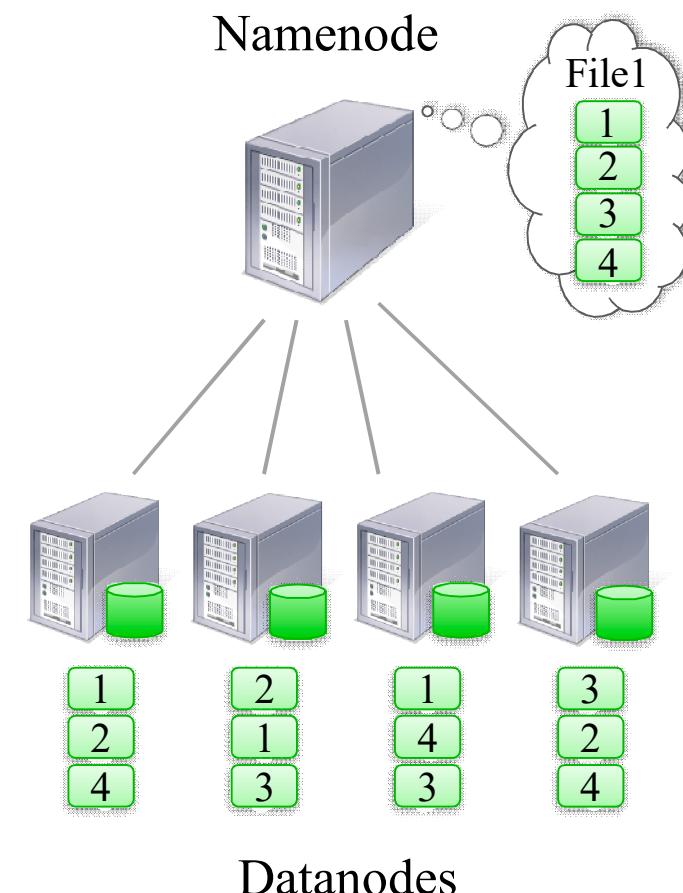
# Hadoop Components

- MapReduce
  - Runs jobs submitted by users
  - Manages work distribution & fault-tolerance
- Distributed File System (HDFS)
  - Runs on same machines!
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance



# Distributed File System

- 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



# Hadoop

- Download from [hadoop.apache.org](http://hadoop.apache.org)
- To install locally, unzip and set `JAVA_HOME`
- Docs: [hadoop.apache.org/common/docs/current](http://hadoop.apache.org/common/docs/current)
- Three ways to write jobs:
  - Java API
  - Hadoop Streaming (for Python, Perl, etc)
  - Pipes API (C++)

# Word Count in Java

```
public static class MapClass extends MapReduceBase
 implements Mapper<LongWritable, Text, Text, IntWritable> {

 private final static IntWritable ONE = new IntWritable(1);

 public void map(LongWritable key, Text value,
 OutputCollector<Text, IntWritable> output,
 Reporter reporter) throws IOException {
 String line = value.toString();
 StringTokenizer itr = new StringTokenizer(line);
 while (itr.hasMoreTokens()) {
 output.collect(new Text(itr.nextToken()), ONE);
 }
 }
}
```

# Word Count in Java

```
public static class Reduce extends MapReduceBase
 implements Reducer<Text, IntWritable, Text, IntWritable> {

 public void reduce(Text key, Iterator<IntWritable> values,
 OutputCollector<Text, IntWritable> output,
 Reporter reporter) throws IOException {
 int sum = 0;
 while (values.hasNext()) {
 sum += values.next().get();
 }
 output.collect(key, new IntWritable(sum));
 }
}
```

# Word Count in Java

```
public static void main(String[] args) throws Exception {
 JobConf conf = new JobConf(WordCount.class);
 conf.setJobName("wordcount");

 conf.setMapperClass(MapClass.class);
 conf.setCombinerClass(Reduce.class);
 conf.setReducerClass(Reduce.class);

 FileInputFormat.setInputPaths(conf, args[0]);
 FileOutputFormat.setOutputPath(conf, new Path(args[1]));

 conf.setOutputKeyClass(Text.class); // out keys are words (strings)
 conf.setOutputValueClass(IntWritable.class); // values are counts

 JobClient.runJob(conf);
}
```

# Word Count in Python with Hadoop Streaming

**Mapper.py:**

```
import sys
for line in sys.stdin:
 for word in line.split():
 print(word.lower() + "\t" + 1)
```

**Reducer.py**

```
: import sys
counts = {}
for line in sys.stdin:
 word, count = line.split("\t")
 dict[word] = dict.get(word, 0) + int(count)
for word, count in counts:
 print(word.lower() + "\t" + 1)
```

# Summary

- MapReduce's data-parallel programming model hides complexity of distribution and fault tolerance
- Principal philosophies:
  - *Make it scale*, so you can throw hardware at problems
  - *Make it cheap*, saving hardware, programmer and administration costs (but necessitating fault tolerance)
- MapReduce is not suitable for all problems, new programming models and frameworks still being created (Dryad, Pregel, Spark, etc.)