inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture 18 – MapReduce

2014-10-13

Senior Lecturer SOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

Top 10 Tech Trends ⇒

(1) Computing everywhere (2) Internet of Things (3) 3D printing (4) Analytics everywhere (5) Context rich systems (6) smart machines (7) cloud computing (8) software applications (9) web-scale IT (10) Security. Agree?

www.computerworld.com/article/2692619/gartner-lays-out-its-top-10-tech-trends-for-2015.html

Review of Last Lecture

- Warehouse Scale Computing
  - Example of parallel processing in the post-PC era
  - Servers on a rack, rack part of cluster
  - Issues to handle include load balancing, failures, power usage (sensitive to cost & energy efficiency)
  - PUE = Total building power / IT equipment power

Great Idea #4: Parallelism

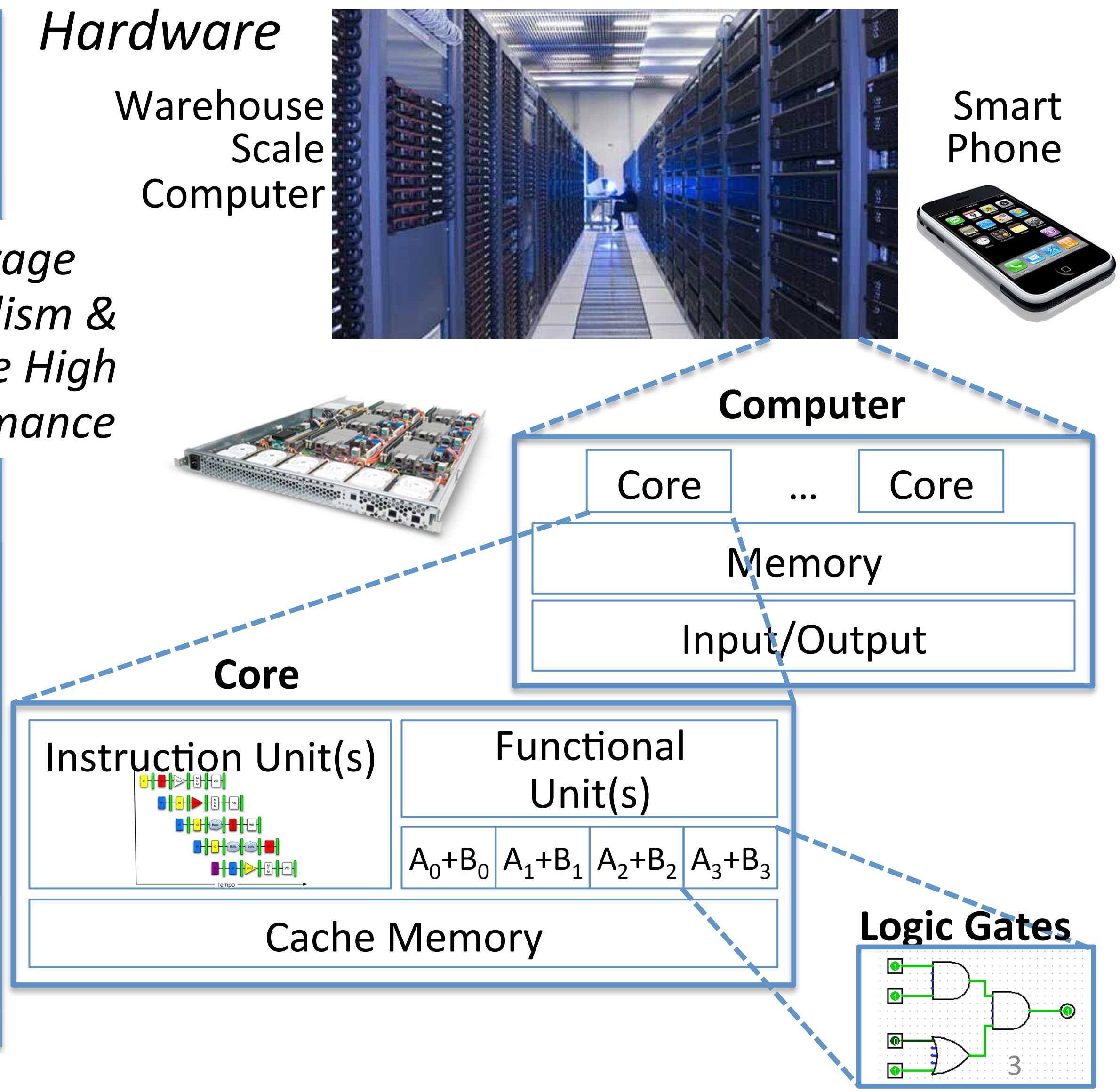
Today's Lecture

Software

- Parallel Requests
  Assigned to computer
  e.g. Search "Garcia"
- Parallel Threads
  Assigned to core
  e.g. Lookup, Ads
- Parallel Instructions
  > 1 instruction @ one time
  e.g. 5 pipelined instructions
- Parallel Data
  > 1 data item @ one time
  e.g. add of 4 pairs of words
- Hardware descriptions
  All gates functioning in
  parallel at same time

Leverage Parallelism & Achieve High Performance

Hardware

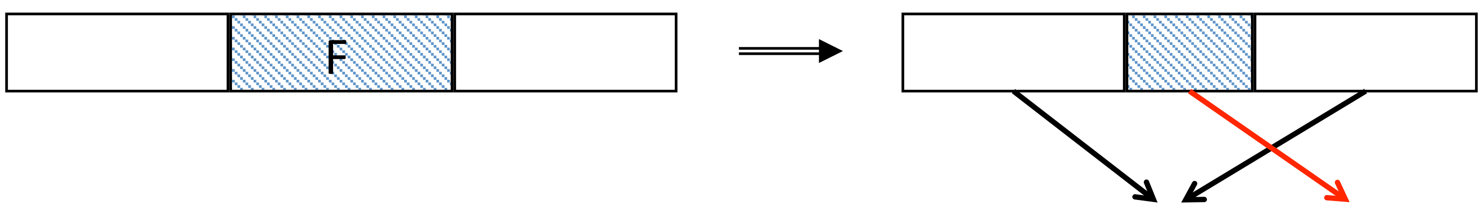


Amdahl's (Heartbreaking) Law

- Speedup due to enhancement E:

$$\text{Speedup w/E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/E}}$$

- **Example:** Suppose that enhancement E accelerates a fraction F (F<1) of the task by a factor S (S>1) and the remainder of the task is unaffected



- Exec time w/E = Exec Time w/o E × [(1-F) + F/S]

  Speedup w/E = 1 / [(1-F) + F/S]

Amdahl's Law

- Speedup $= \dfrac{1}{(1 - F) + \dfrac{F}{S}}$

Non-sped-up part

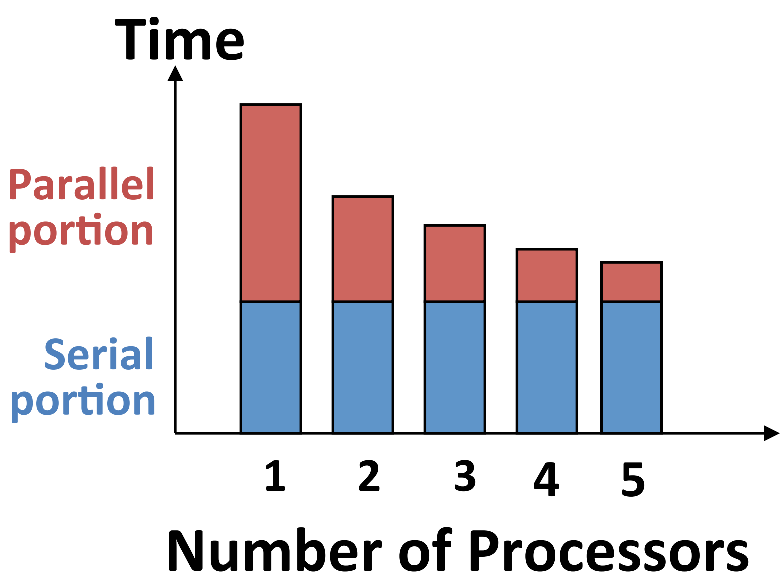
Sped-up part

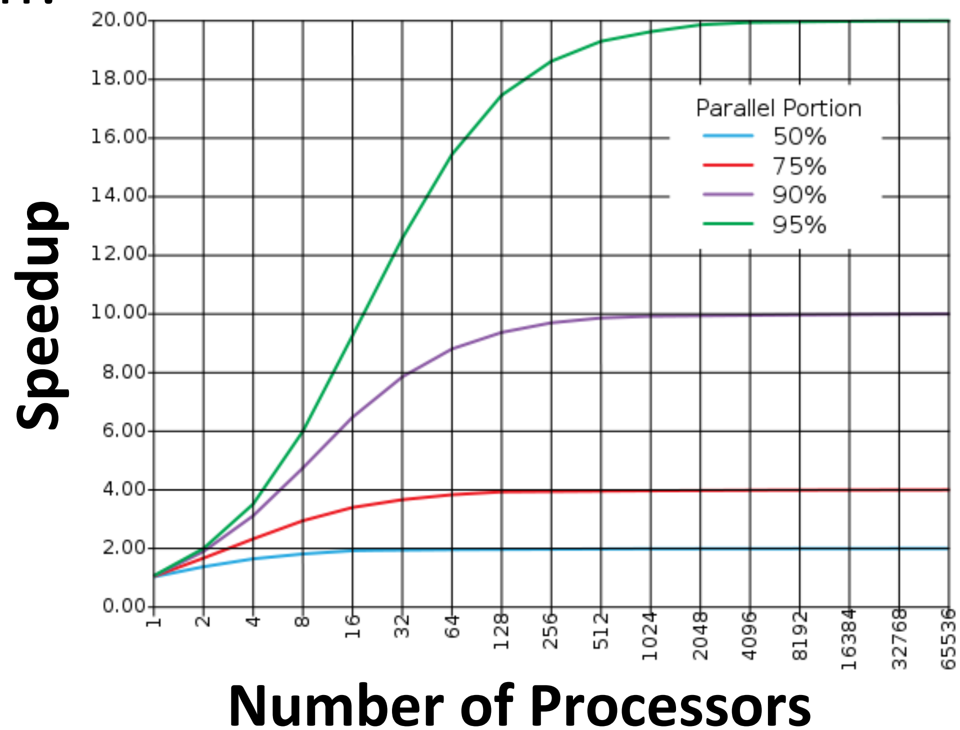
- **Example:** the execution time of 1/5 of the program can be accelerated by a factor of 10. What is the program speed-up overall?

$$\frac{1}{0.8 + \frac{0.2}{10}} = \frac{1}{0.8 + 0.02} = \color{red}{1.22}$$

Consequence of Amdahl's Law

- The amount of speedup that can be achieved through parallelism is limited by the non-parallel portion of your program!

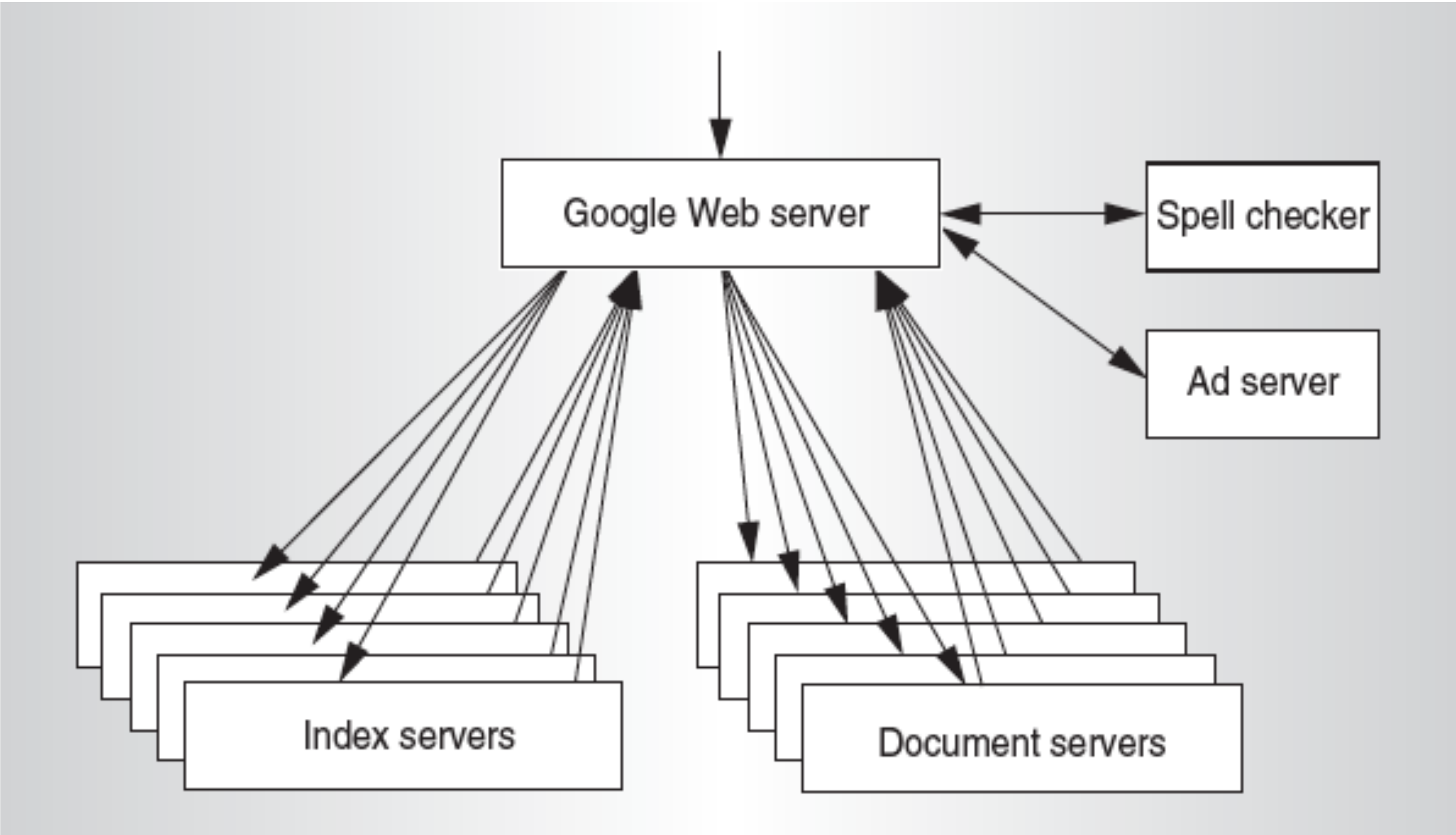




Request-Level Parallelism (RLP)

- Hundreds or thousands of requests per sec
  - Not your laptop or cell-phone, but popular Internet services like web search, social networking, …
  - Such requests are largely independent
    - Often involve read-mostly databases
    - Rarely involve strict read–write data sharing or synchronization across requests
- Computation easily partitioned within a request and across different requests

Google Query-Serving Architecture



Data-Level Parallelism (DLP)

- Two kinds:
  1) Lots of data in memory that can be operated on in parallel (e.g. adding together 2 arrays)
  2) Lots of data on many disks that can be operated on in parallel (e.g. searching for documents)

1) SIMD does Data-Level Parallelism (DLP) in memory
2) Today's lecture, Lab 6, Proj. 3 do DLP across many servers and disks using MapReduce

Administrivia … The Midterm

- Average around 10/20
  - Despite lots of partial credit
  - Regrades being processed
  - Have perspective – it's only 20 / 300 points.
  - Don't panic. Do lots of practice problems in a team. Do NOT study alone.
- Part 2 will be easier to compensate
- You can clobber Part 1 with Part 2

What is MapReduce?

- Simple data-parallel programming model designed for scalability and fault-tolerance

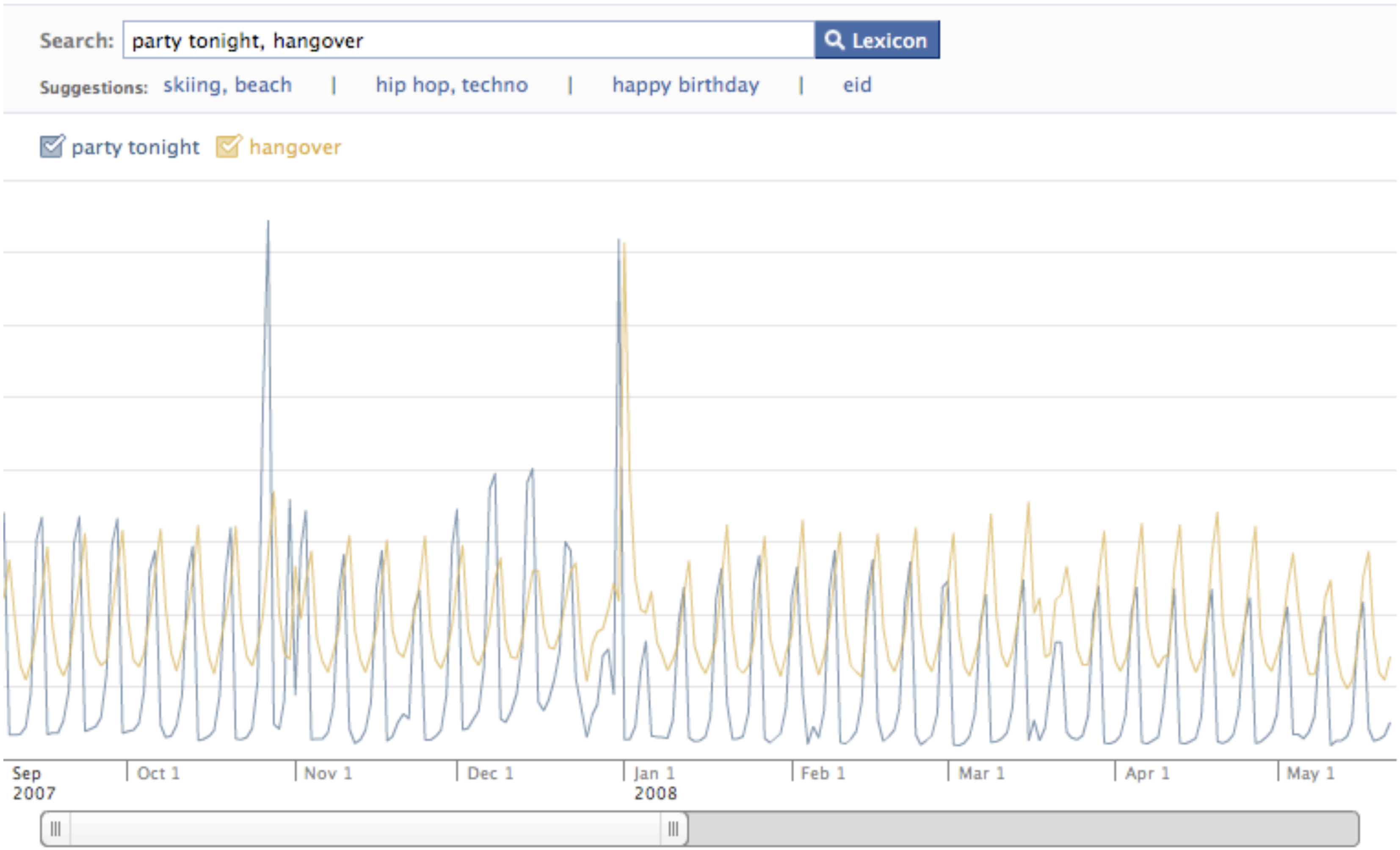
- Pioneered by Google
  - Processes > 25 petabytes of data per day

- Popularized by open-source Hadoop project
  - Used at Yahoo!, Facebook, Amazon, …

What is MapReduce used for?

- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
  - For computing multi-layer street maps
- At Yahoo!:
  - "Web map" powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

Example: Facebook Lexicon



www.facebook.com/lexicon(no longer available)

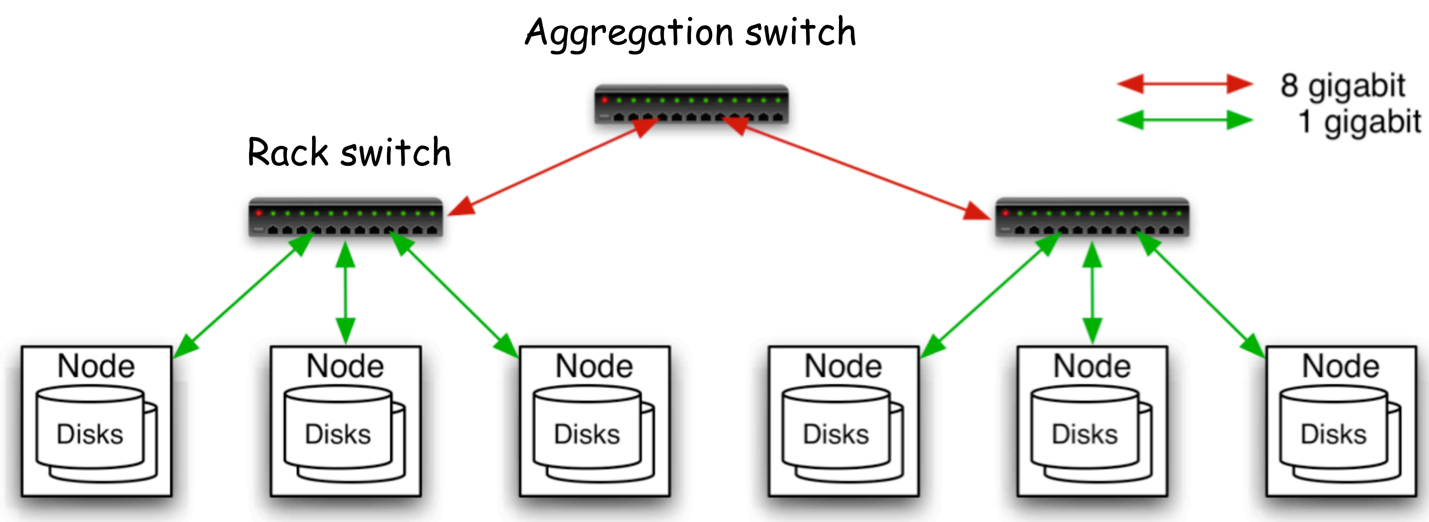
MapReduce Design Goals

1. Scalability to large data volumes:
   - 1000's of machines, 10,000's of disks

2. Cost-efficiency:
   - Commodity machines (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance via re-execution (fewer administrators)
   - Easy, fun to use (fewer programmers)

Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *6th USENIX Symposium on Operating Systems Design and Implementation, 2004*. (optional reading, linked on course homepage – a digestible CS paper at the 61C level)

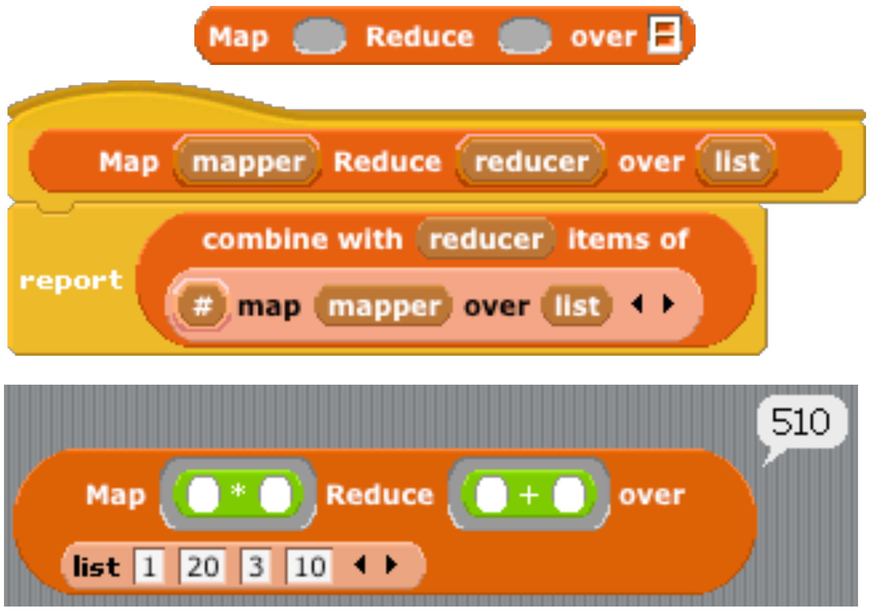
Typical Hadoop Cluster

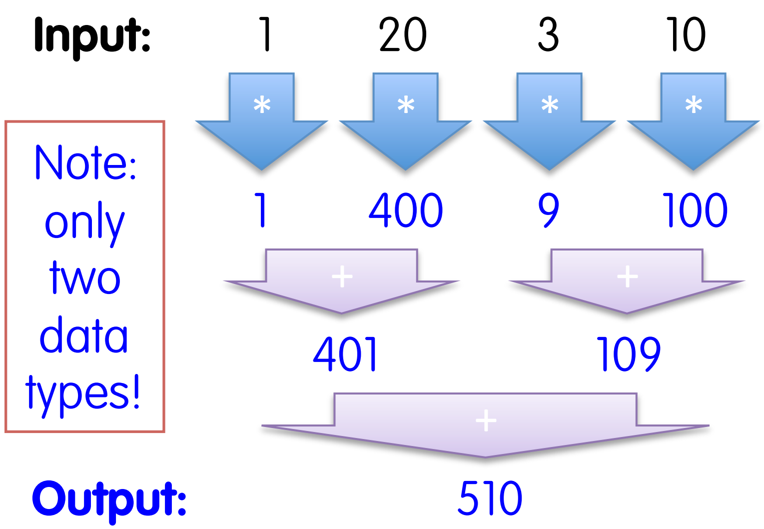


- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort):
  8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf

MapReduce in CS10 & CS61A{,S}






```
> (reduce +
    (map square '(1 20 3 10))
510
```

```
>>> from functools import reduce
>>> def plus(x,y): return x+y
>>> def square(x): return x*x
>>> reduce(plus,
        map(square, (1,20,3,10)))
510
```

MapReduce Programming Model

Input & Output: each a set of key/value pairs

Programmer specifies two functions:

```
map (in_key, in_value) →
      list(interm_key, interm_value)
```

- Processes input key/value pair
- Slices data into “shards” or “splits”; distributed to workers
- Produces set of intermediate pairs

```
reduce (interm_key, list(interm_value)) →
         list(out_value)
```

- Combines all intermediate values for a particular key
- Produces a set of merged output values (usu just one)

code.google.com/edu/parallel/mapreduce-tutorial.html

MapReduce WordCount Example

- “Mapper” nodes are responsible for the **map** function

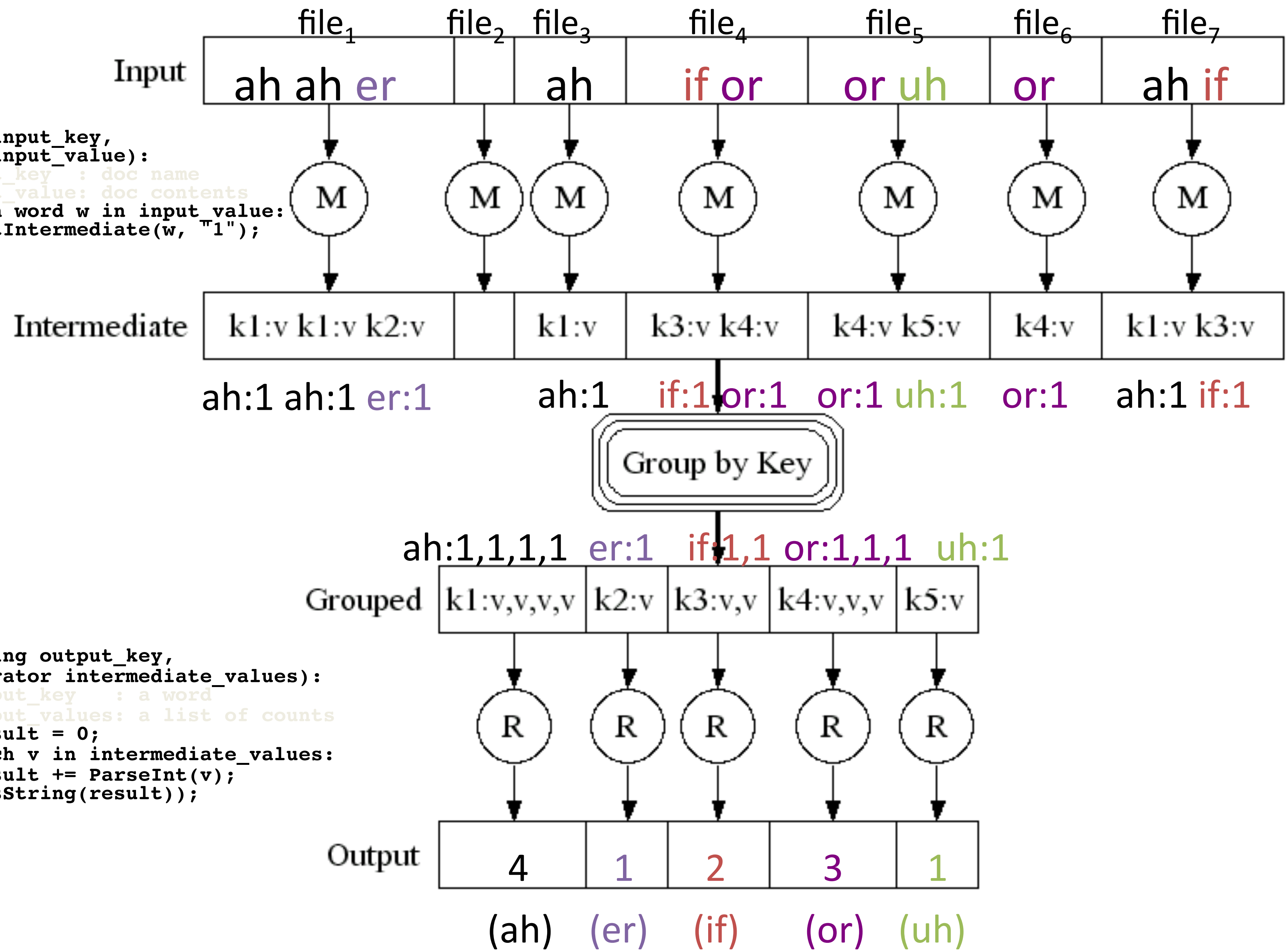
```
// "I do I learn" → ("I",1), ("do",1), ("I",1), ("learn",1)
map(String input_key,
    String input_value):
    // input_key  : document name (or line of text)
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");
```

- “Reducer” nodes are responsible for the **reduce** function

```
// ("I",[1,1]) → ("I",2)
reduce(String output_key,
       Iterator intermediate_values):
    // output_key   : a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

- Data on a distributed file system (DFS)

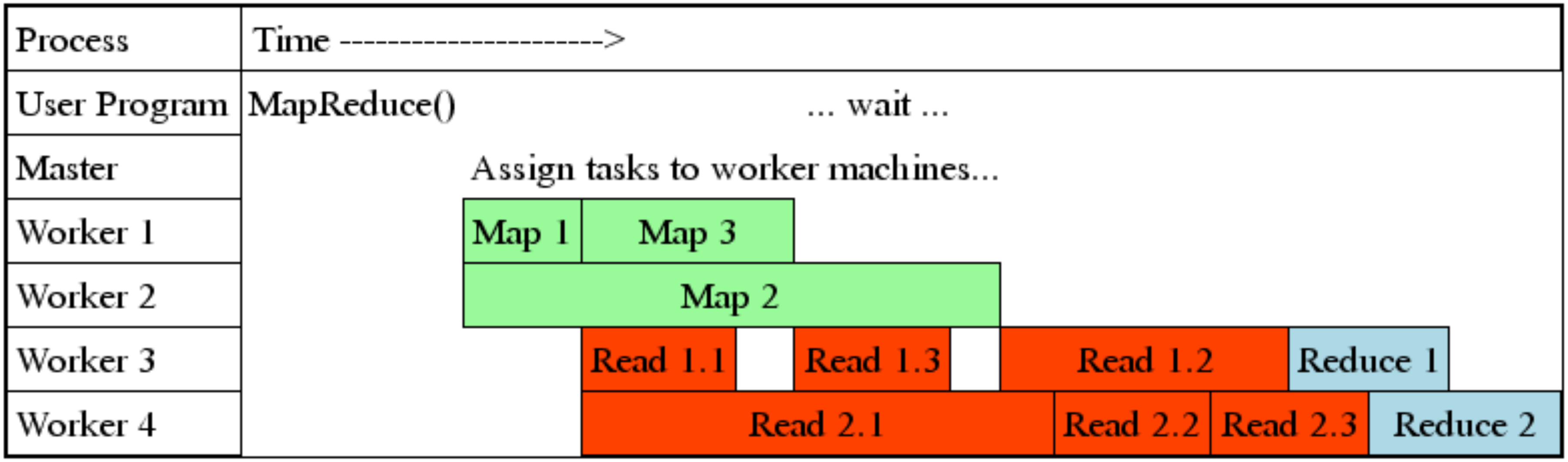
MapReduce WordCount Diagram




```
map(String input_key,
    String input_value):
    // input_key  : doc name
    // input_value: doc contents
    for each word w in input_value:
        EmitIntermediate(w, "1");
```

```
reduce(String output_key,
       Iterator intermediate_values):
    // output_key   : a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

MapReduce Processing Time Line



- Master assigns map + reduce tasks to “worker” servers
- As soon as a map task finishes, worker server can be assigned a new map or reduce task
- Data shuffle begins as soon as a given Map finishes
- Reduce task begins as soon as all data shuffles finish
- To tolerate faults, reassign task if a worker server “dies”

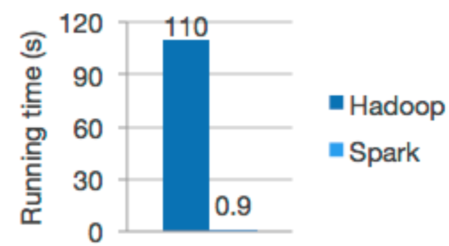
MapReduce WordCount Java code

```java
public static void main(String[] args) throws IOException {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(WCMap.class);
    conf.setCombinerClass(WCReduce.class);
    conf.setReducerClass(WCReduce.class);
    conf.setInputPath(new Path(args[0]));
    conf.setOutputPath(new Path(args[1]));
    JobClient.runJob(conf);
}

public class WCMap extends MapReduceBase implements Mapper {
    private static final IntWritable ONE = new IntWritable(1);
    public void map(WritableComparable key, Writable value,
                    OutputCollector output,
                    Reporter reporter) throws IOException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            output.collect(new Text(itr.next()), ONE);
        }
    }
}

public class WCReduce extends MapReduceBase implements Reducer {
    public void reduce(WritableComparable key, Iterator values,
                       OutputCollector output,
                       Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += ((IntWritable) values.next()).get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

spark.apache.org

Spark

- **Apache Spark™** is a fast and general engine for large-scale data processing.
- **Speed**
  - Run programs up to 100x faster than Hadoop in memory, or 10x faster on disk.
  - Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.



- **Ease of Use**
  - Write applications quickly in Java, Scala or Python.
  - Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala and Python shells.

Word Count in Spark's Python API

```python
file = spark.textFile("hdfs://...")

file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

Cf Java:

```java
public static void main(String[] args) throws IOException {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(WCMap.class);
    conf.setCombinerClass(WCReduce.class);
    conf.setReducerClass(WCReduce.class);
    conf.setInputPath(new Path(args[0]));
    conf.setOutputPath(new Path(args[1]));
    JobClient.runJob(conf);
}

public class WCMap extends MapReduceBase implements Mapper {
    private static final IntWritable ONE = new IntWritable(1);
    public void map(WritableComparable key, Writable value,
                    OutputCollector output,
                    Reporter reporter) throws IOException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            output.collect(new Text(itr.next()), ONE);
        }
    }
}

public class WCReduce extends MapReduceBase implements Reducer {
    public void reduce(WritableComparable key, Iterator values,
                       OutputCollector output,
                       Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += ((IntWritable) values.next()).get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

Peer Instruction

1. Writing & managing SETI@Home is relatively straightforward; just hand out & gather data
2. Most parallel programs that, when run on N (N big) identical supercomputer processors will yield close to N x performance increase
3. The majority of the world's computing power lives in supercomputer centers and warehouses

	123
A:	FFF
B:	FFT
B:	FTF
C:	FTT
C:	TFF
D:	TFT
D:	TTF
E:	TTT

Peer Instruction Answer

1. The heterogeneity of the machines, handling machines that fail, falsify data. FALSE
2. The combination of Amdahl's law, overhead, and load balancing take its toll. FALSE
3. Have you considered how many PCs + Smart Devices + game devices exist? Not even close. FALSE

1. Writing & managing SETI@Home is relatively straightforward; just hand out & gather data
2. Most parallel programs that, when run on N (N big) <u>identical</u> supercomputer processors will yield close to N x performance increase
3. The majority of the world's computing power lives in supercomputer centers and warehouses

	123
A:	FFF
B:	FFT
B:	FTF
C:	FTT
C:	TFF
D:	TFT
D:	TTF
E:	TTT

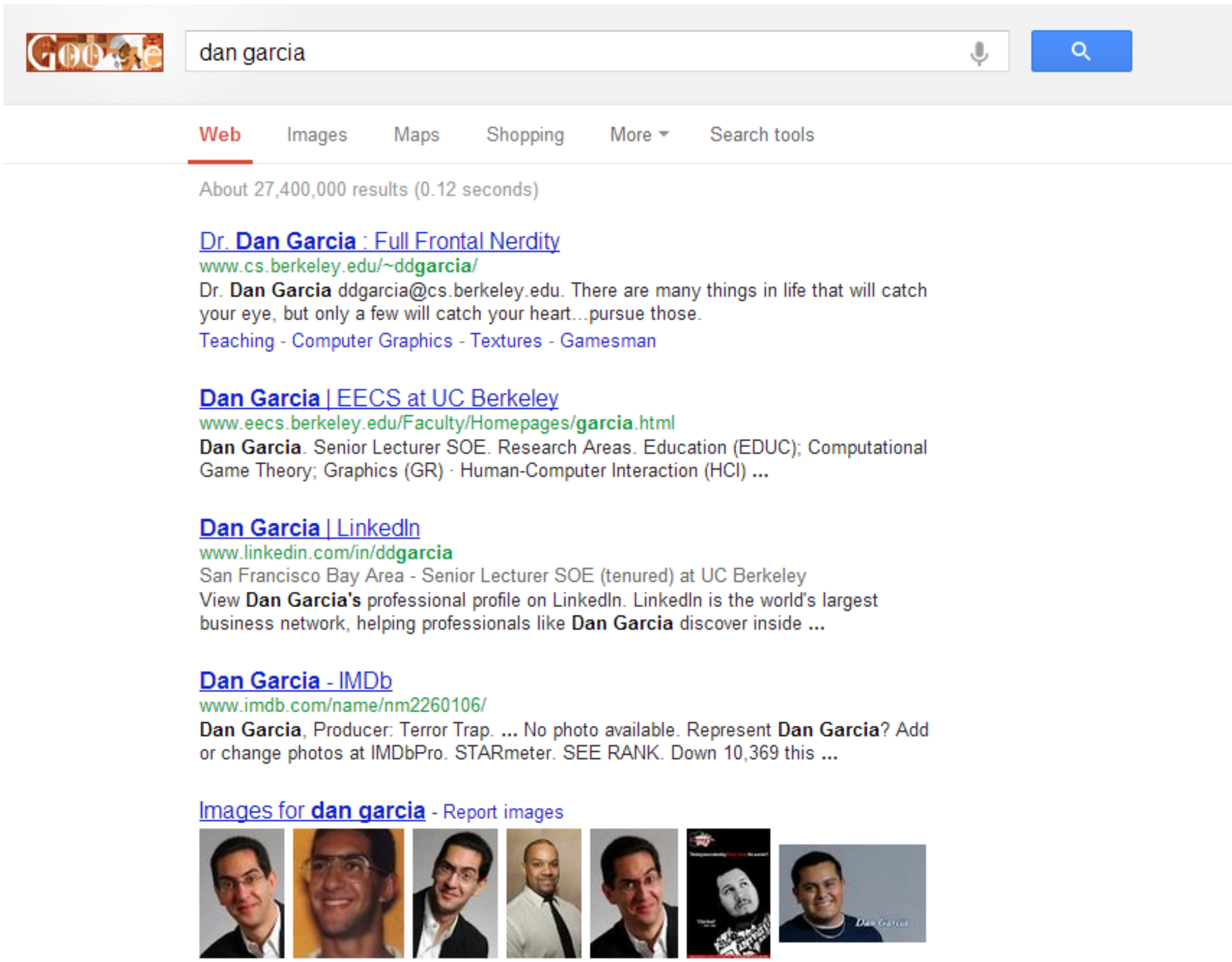
Bonus Slides

Agenda

- Amdahl's Law
- Request Level Parallelism
- MapReduce (Data Level Parallelism)
  – Background
  – Design
  – Theory
- Administrivia
- More MapReduce
  – The Combiner
  – Execution Walkthrough
  – (Bonus) Example 2: PageRank (aka How Google Search Works)

Anatomy of a Web Search

- Google "Dan Garcia"



Anatomy of a Web Search (1 of 3)

- Google “Dan Garcia”
  - Direct request to “closest” Google Warehouse Scale Computer
  - Front-end load balancer directs request to one of many arrays (cluster of servers) within WSC
  - Within array, select one of many Google Web Servers (GWS) to handle the request and compose the response pages
  - GWS communicates with Index Servers to find documents that contain the search words, “Dan”, “Garcia”, uses location of search as well
  - Return document list with associated relevance score

Anatomy of a Web Search (2 of 3)

- In parallel,
  - Ad system: run ad auction for bidders on search terms
  - Get images of various "Dan Garcia"s
- Use docids (document IDs) to access indexed documents
- Compose the page
  - Result document extracts (with keyword in context) ordered by relevance score
  - Sponsored links (along the top) and advertisements (along the sides)

Anatomy of a Web Search (3 of 3)

- Implementation strategy
  - Randomly distribute the entries
  - Make many copies of data (a.k.a. "replicas")
  - Load balance requests across replicas
- Redundant copies of indices and documents
  - Breaks up search hot spots, e.g. "WhatsApp"
  - Increases opportunities for request-level parallelism
  - Makes the system more tolerant of failures

The Combiner (Optional)

- One missing piece for our first example:
  - Many times, the output of a single mapper can be "compressed" to save on bandwidth and to distribute work (usually more map tasks than reduce tasks)
  - To implement this, we have the combiner:

```
combiner(interm_key,list(interm_val)):
    // DO WORK (usually like reducer)
    emit(interm_key2, interm_val2)
```

Our Final Execution Sequence

- Map – Apply operations to all input key, val
- Combine – Apply reducer operation, but distributed across map tasks
- Reduce – Combine all values of a key to produce desired output

Agenda

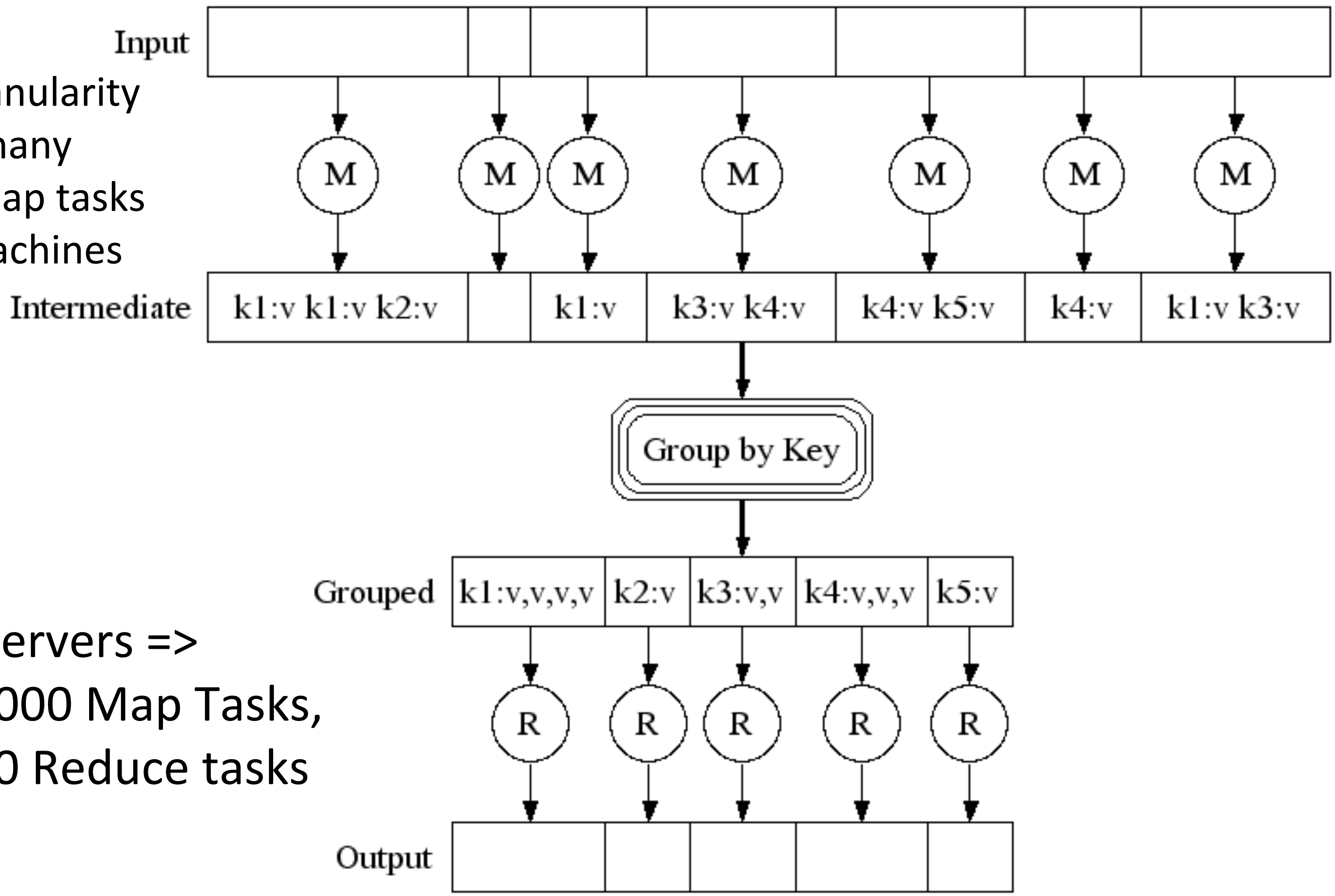
- Amdahl's Law
- Request Level Parallelism
- MapReduce (Data Level Parallelism)
  - Background
  - Design
  - Theory
- Administrivia
- More MapReduce
  - The Combiner + Example 1: Word Count
  - Execution Walkthrough
  - (Bonus) Example 2: PageRank (aka How Google Search Works)

Execution Setup

- Map invocations distributed by partitioning input data into M *splits*
  - Typically 16 MB to 64 MB per piece
- Input processed in parallel on different servers
- Reduce invocations distributed by partitioning intermediate key space into R pieces
  - e.g. hash(key) mod R
- User picks M >> # servers, R > # servers
  - Big M helps with load balancing, recovery from failure
  - One output file per R invocation, so not too many

MapReduce Execution

Fine granularity tasks: many more map tasks than machines

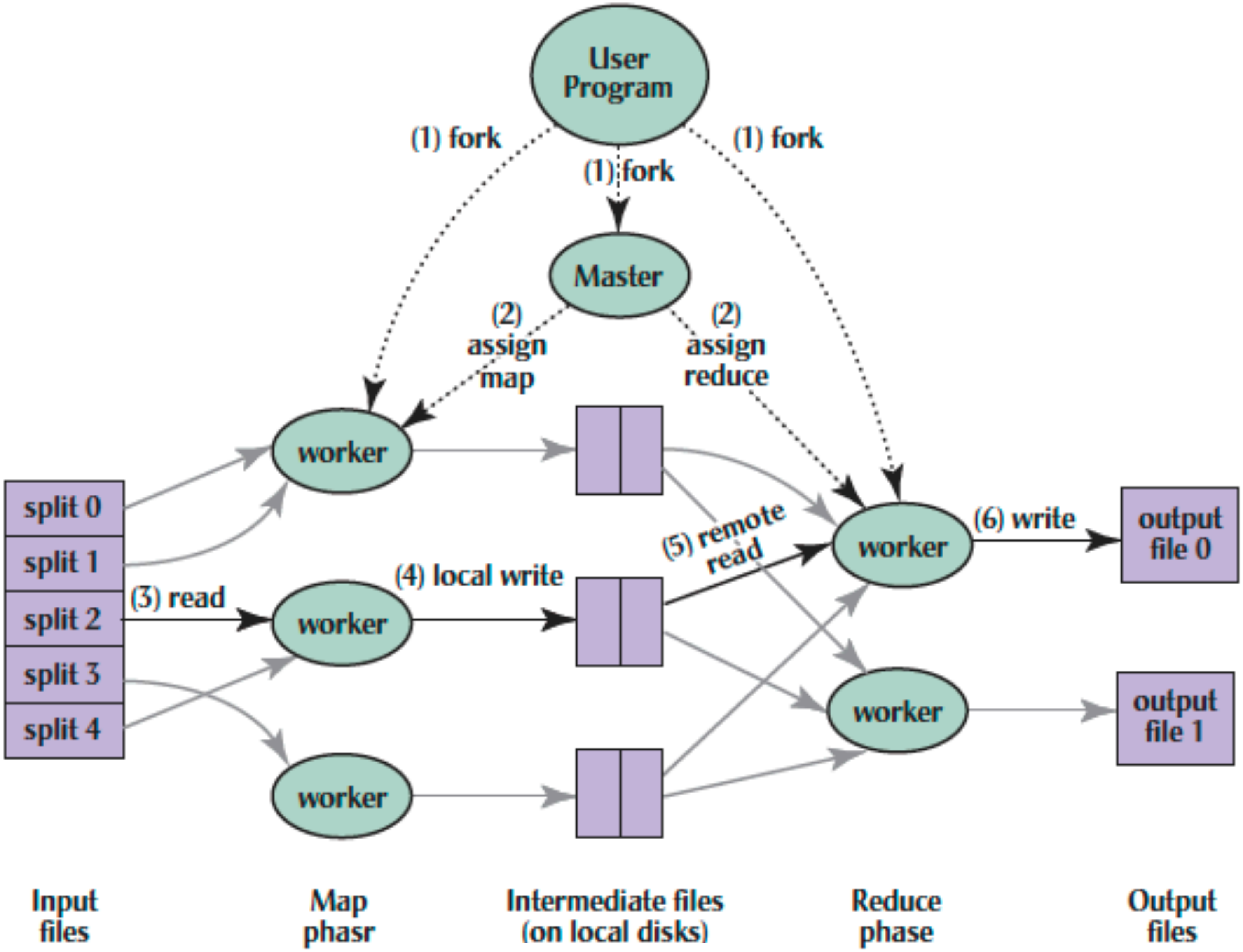


2000 servers =>

≈ 200,000 Map Tasks,

≈ 5,000 Reduce tasks

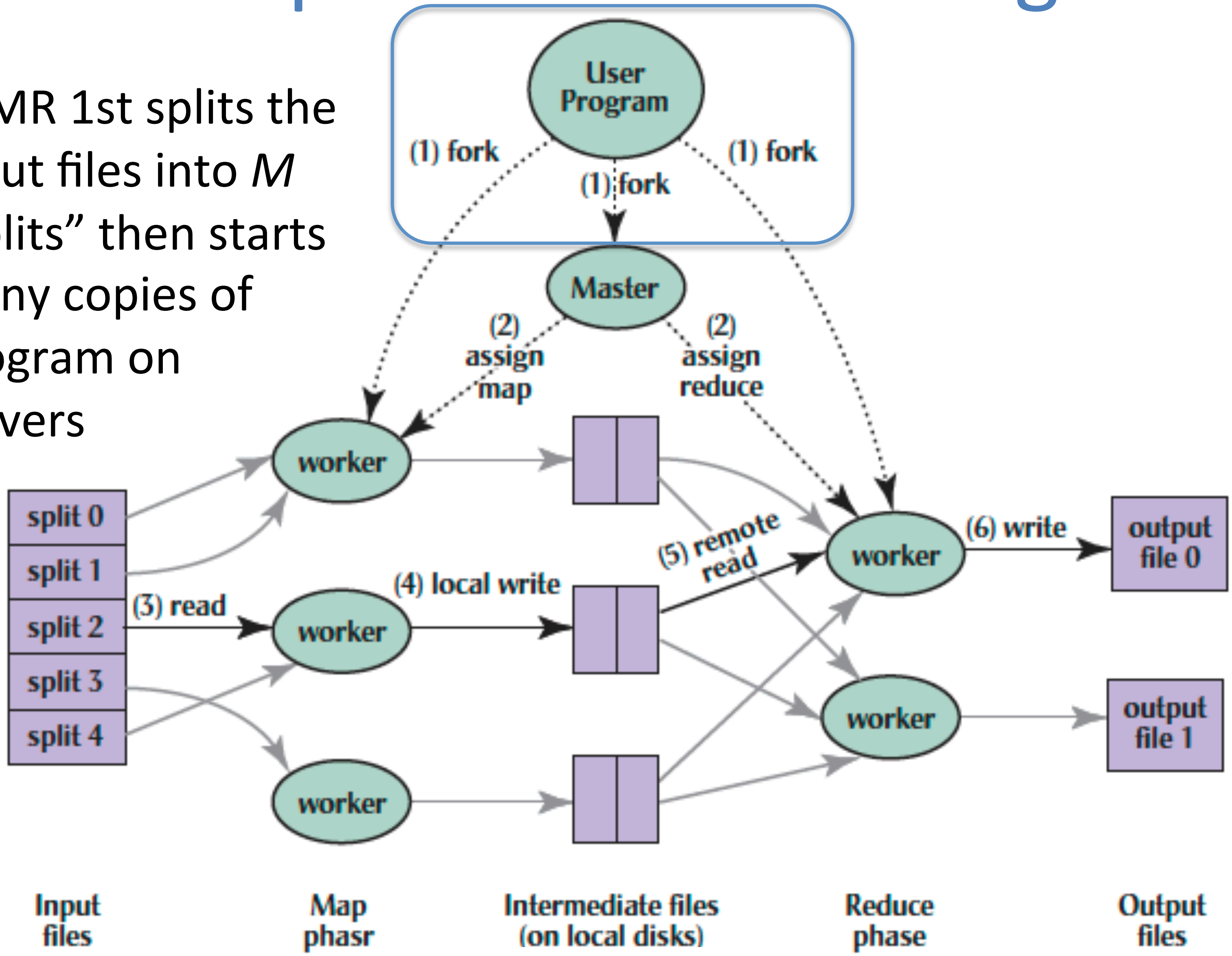
MapReduce Processing



Shuffle phase

MapReduce Processing

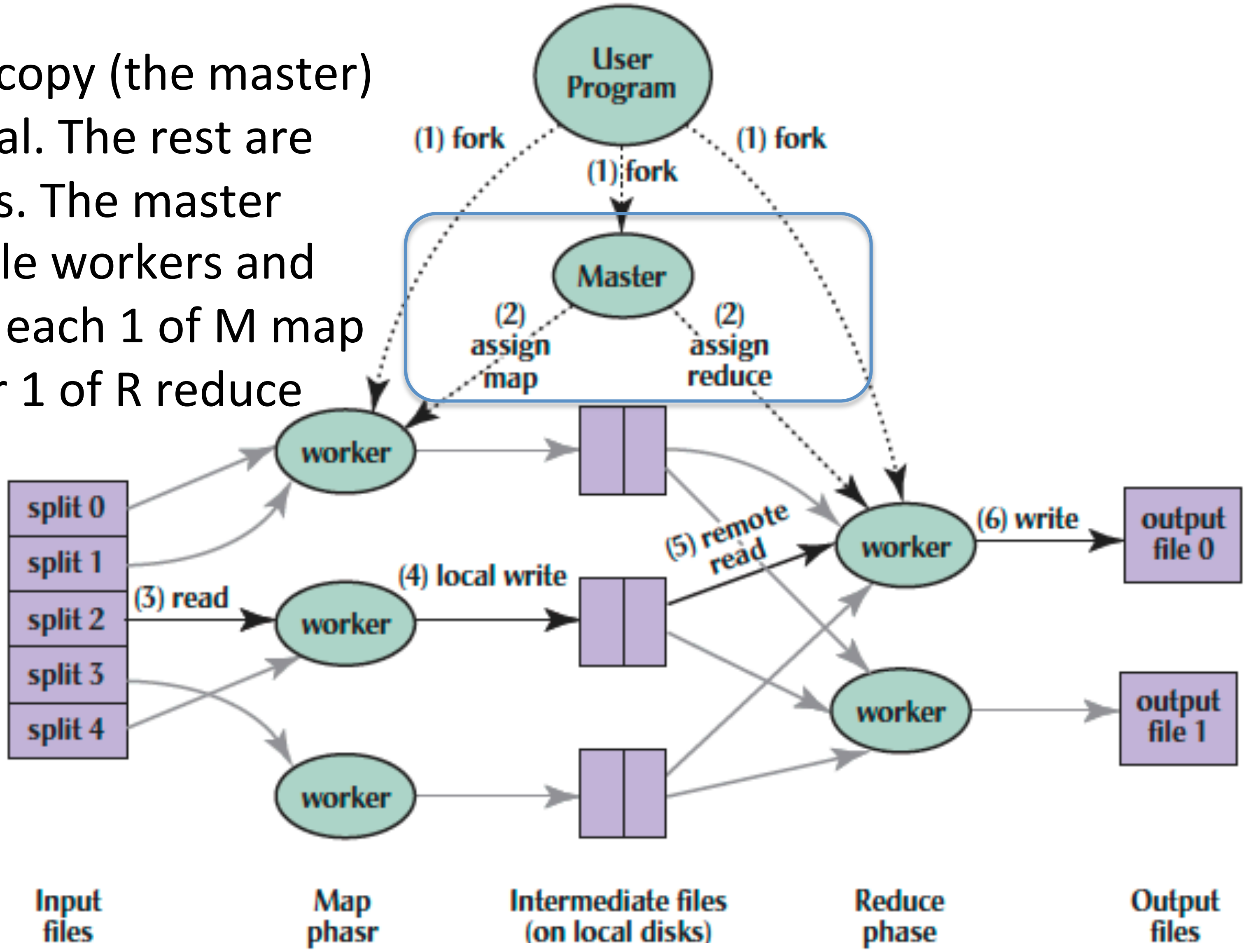
1. MR 1st splits the input files into *M* "splits" then starts many copies of program on servers



Shuffle phase

MapReduce Processing

2. One copy (the master) is special. The rest are workers. The master picks idle workers and assigns each 1 of M map tasks or 1 of R reduce tasks.

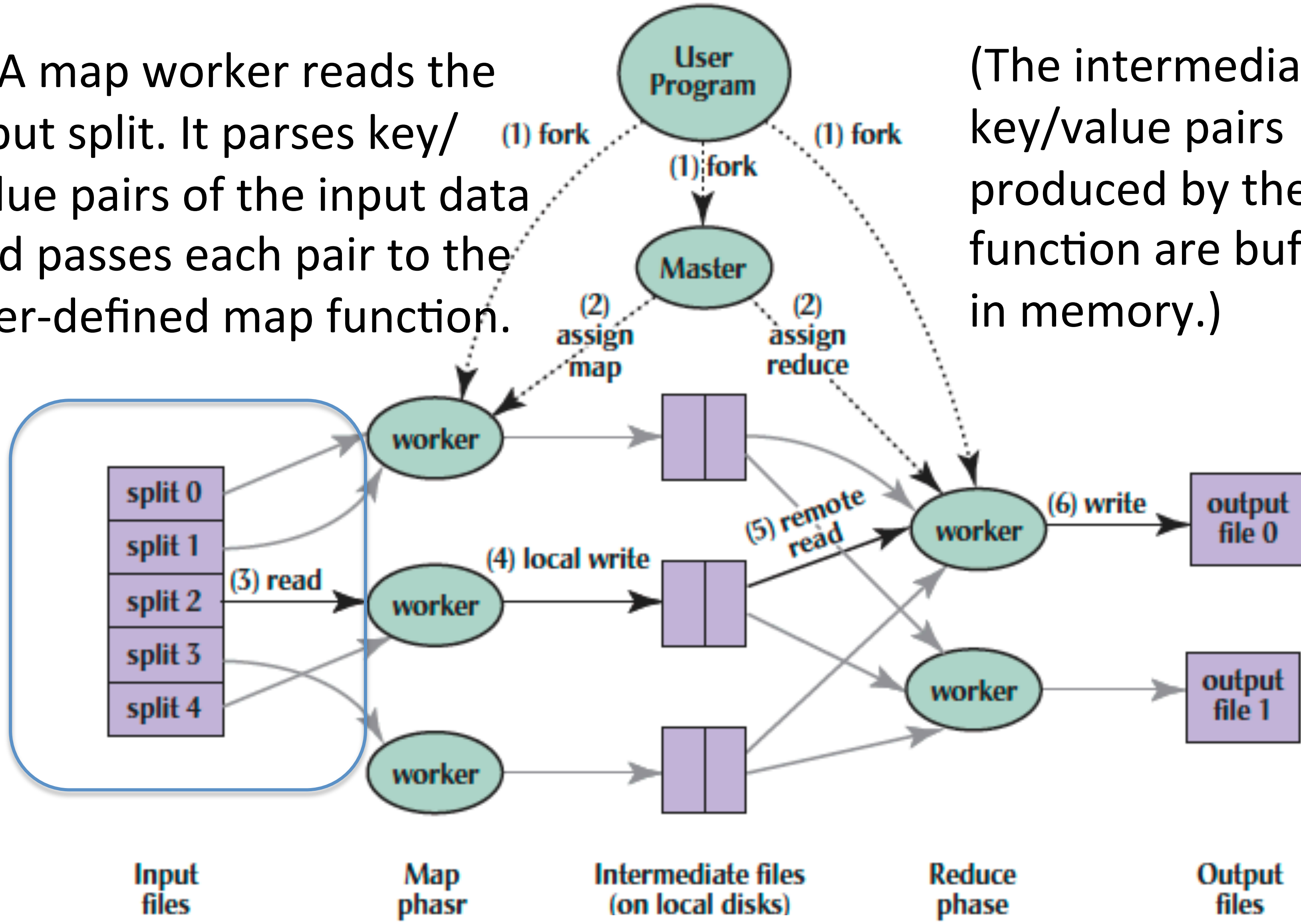


Shuffle phase

MapReduce Processing

3. A map worker reads the input split. It parses key/value pairs of the input data and passes each pair to the user-defined map function.

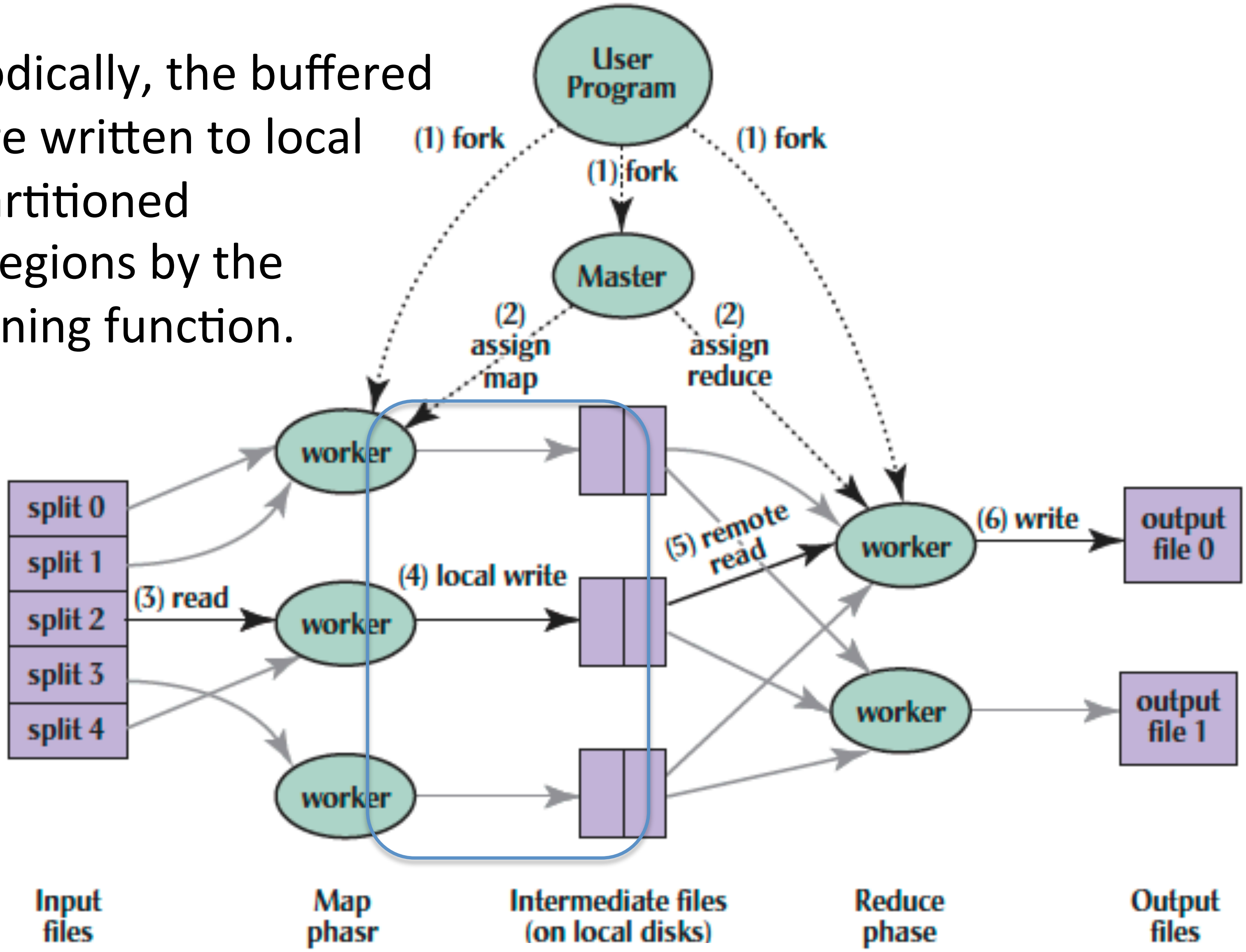
(The intermediate key/value pairs produced by the map function are buffered in memory.)



Shuffle phase

MapReduce Processing

4. Periodically, the buffered pairs are written to local disk, partitioned into *R* regions by the partitioning function.

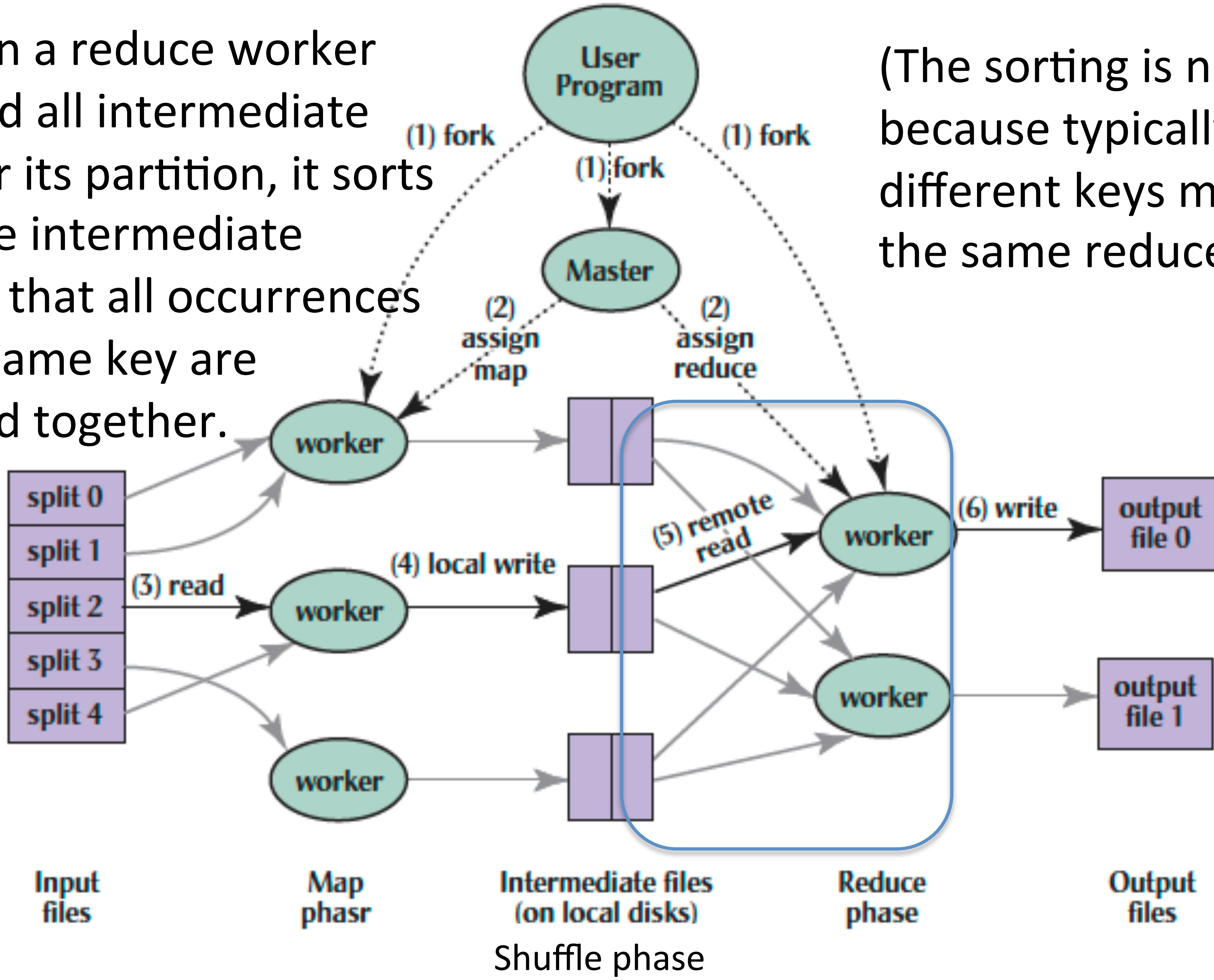


Shuffle phase

MapReduce Processing

5. When a reduce worker has read all intermediate data for its partition, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.

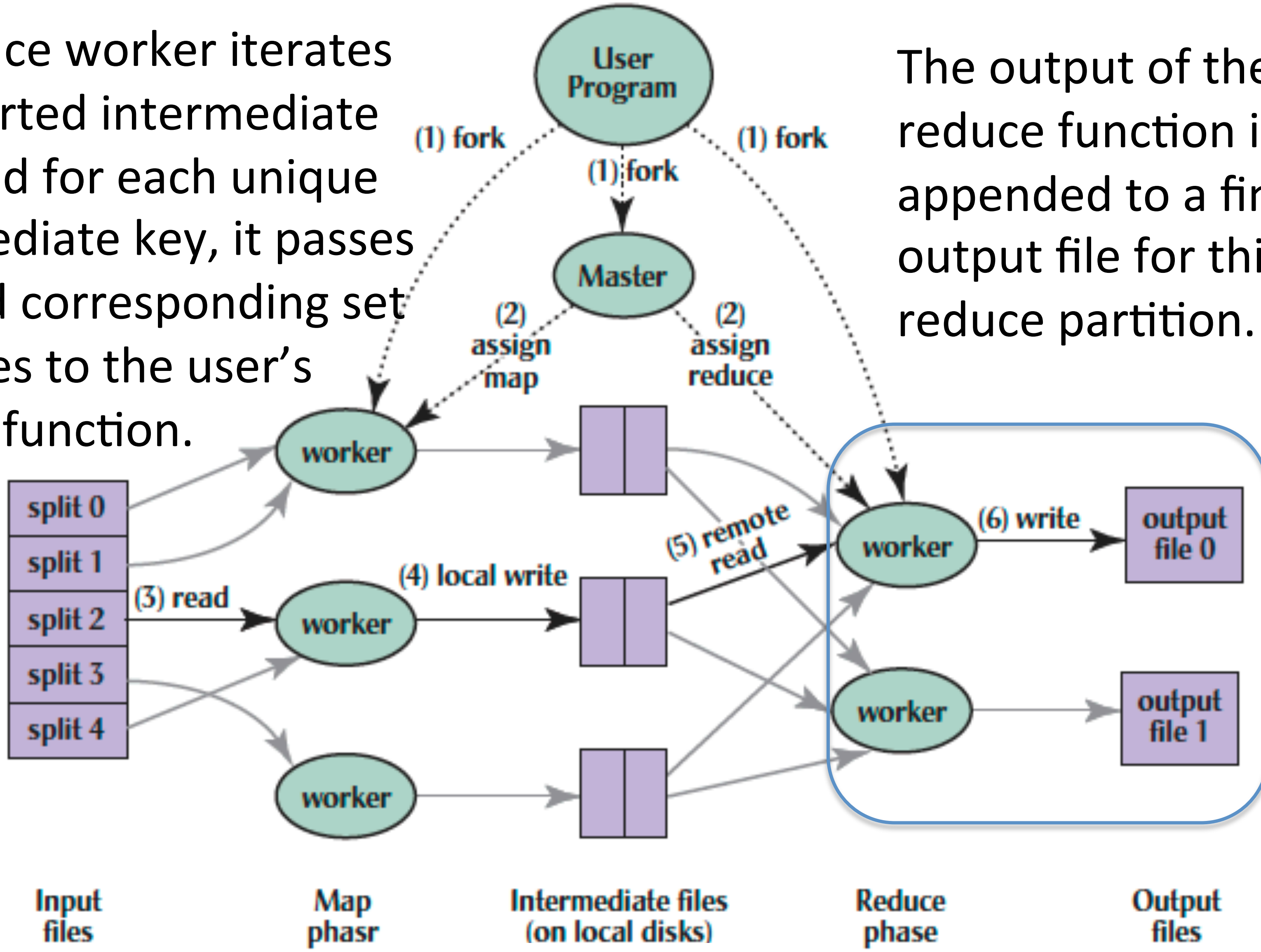
(The sorting is needed because typically many different keys map to the same reduce task)



MapReduce Processing

6. Reduce worker iterates over sorted intermediate data and for each unique intermediate key, it passes key and corresponding set of values to the user's reduce function.

The output of the reduce function is appended to a final output file for this reduce partition.

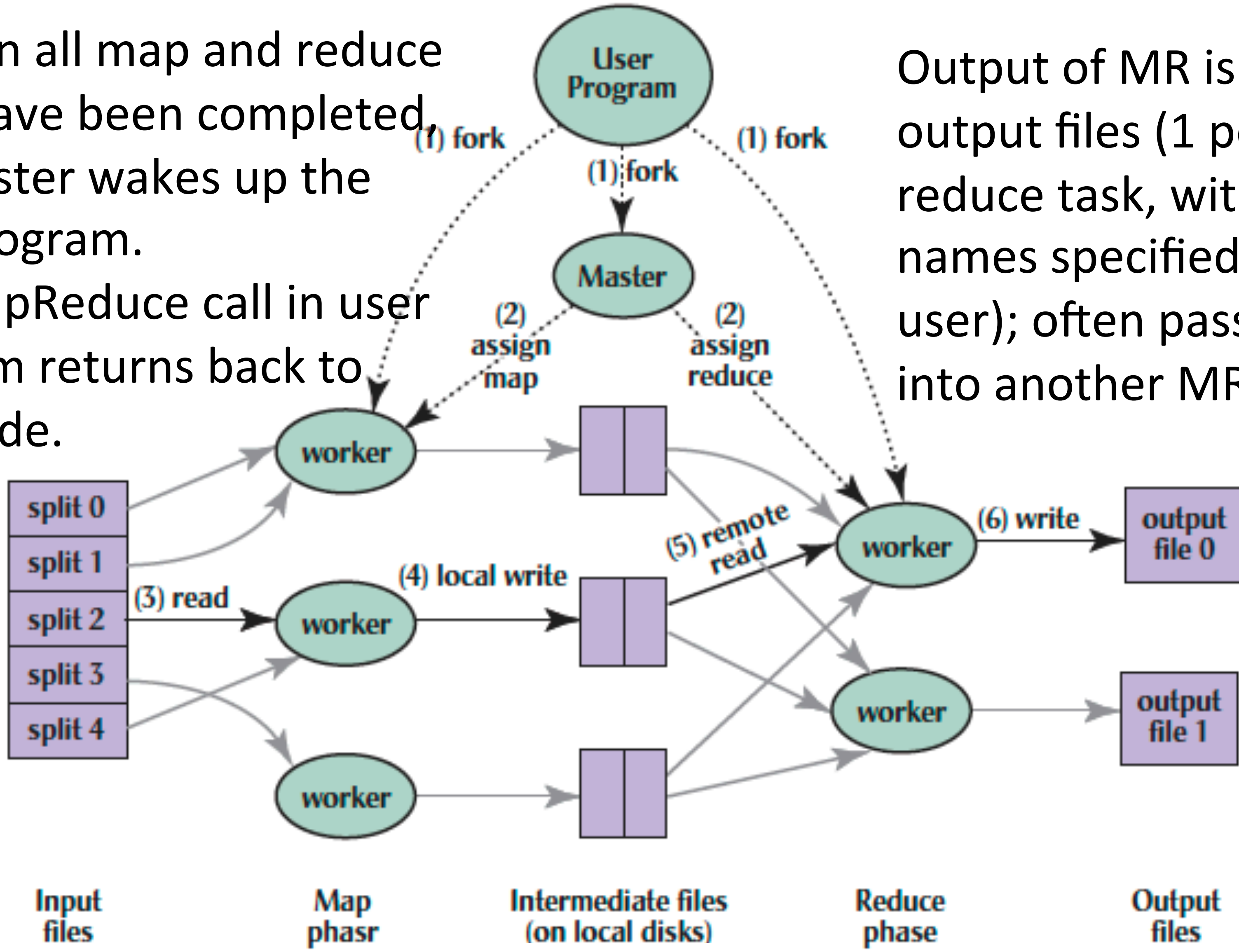


Shuffle phase

MapReduce Processing

7. When all map and reduce tasks have been completed, the master wakes up the user program.
The MapReduce call in user program returns back to user code.

Output of MR is in *R* output files (1 per reduce task, with file names specified by user); often passed into another MR job.

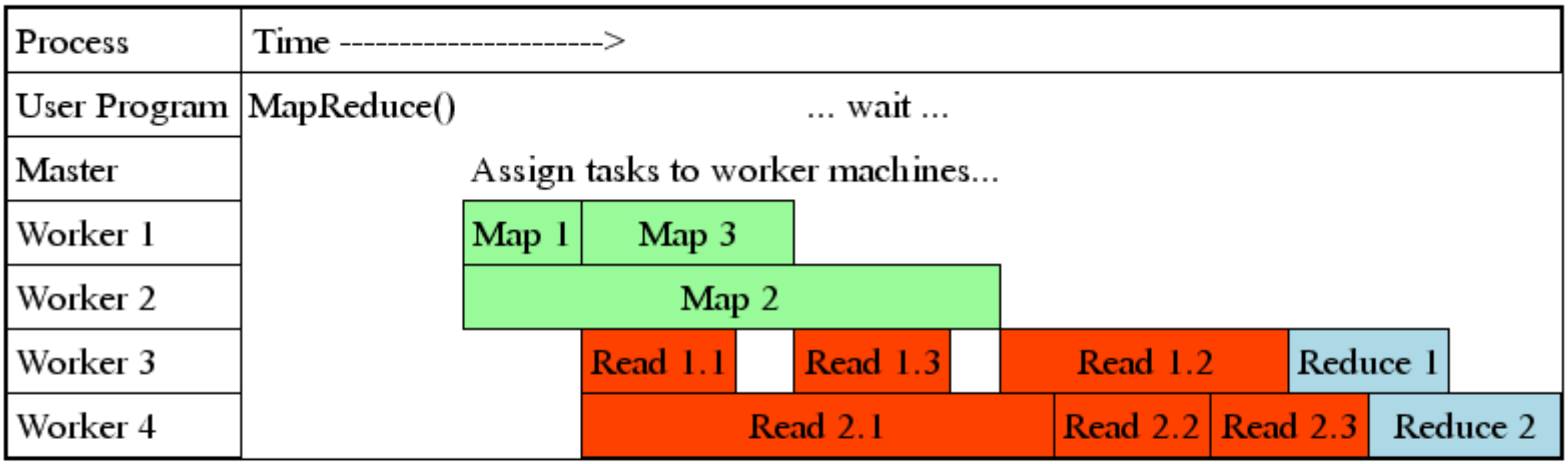


Shuffle phase

What Does the Master Do?

- For each map task and reduce task, keep track:
  - State: idle, in-progress, or completed
  - Identity of worker server (if not idle)
- For each completed map task
  - Stores location and size of R intermediate files
  - Updates files and size as corresponding map tasks complete
- Location and size are pushed incrementally to workers that have in-progress reduce tasks

MapReduce Processing Time Line



- Master assigns map + reduce tasks to “worker” servers
- As soon as a map task finishes, worker server can be assigned a new map or reduce task
- Data shuffle begins as soon as a given Map finishes
- Reduce task begins as soon as all data shuffles finish
- To tolerate faults, reassign task if a worker server “dies”

MapReduce Failure Handling

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Master failure:
  - Protocols exist to handle (master failure unlikely)
- Robust: lost 1600 of 1800 machines once, but finished fine

MapReduce Redundant Execution

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time
  - 3% more resources, large tasks 30% faster

Summary

- MapReduce Data Parallelism
  - Divide large data set into pieces for independent parallel processing
  - Combine and process intermediate results to obtain final result
- Simple to Understand
  - But we can still build complicated software
  - Chaining lets us use the MapReduce paradigm for many common graph and mathematical tasks
- MapReduce is a "Real-World" Tool
  - Worker restart, monitoring to handle failures
  - Google PageRank, Facebook Analytics

Agenda

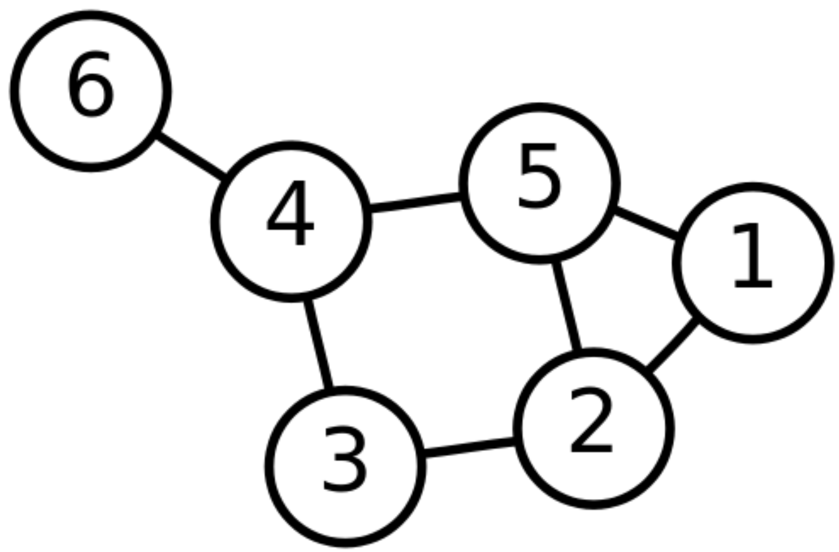
- Amdahl's Law
- Request Level Parallelism
- MapReduce (Data Level Parallelism)
  - Background
  - Design
  - Theory
- Administrivia
- More MapReduce
  - The Combiner + Example 1: Word Count
  - Execution Walkthrough
  - Example 2: PageRank (aka How Google Search Works)

PageRank: How Google Search Works

- Last time: RLP – how Google handles searching its huge index
- Now: How does Google generate that index?
- PageRank is the famous algorithm behind the "quality" of Google's results
  - Uses link structure to rank pages, instead of matching only against content (keyword)

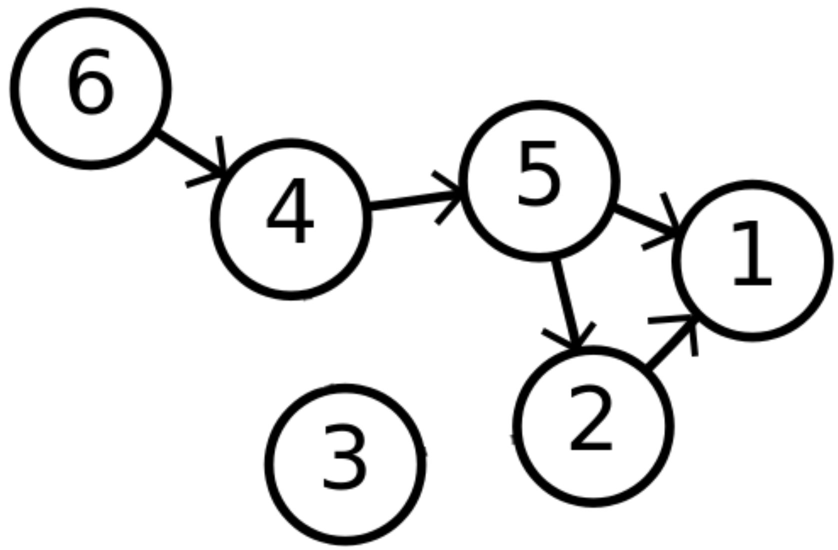
A Quick Detour to CS Theory: Graphs

- Def: A set of objects connected by links
- The "objects" are called Nodes
- The "links" are called Edges
- Nodes: {1, 2, 3, 4, 5, 6}
- Edges: {(6, 4), (4, 5), (4, 3), (3, 2), (5, 2), (5, 1), (1, 2)}



Directed Graphs

- Previously assumed that all edges in the graph were two-way
- Now we have one-way edges:
- Nodes: Same as before
- Edges: (order matters)
  - {(6, 4), (4, 5), (5, 1), (5, 2), (2, 1)}



The Theory Behind PageRank

- The Internet is really a directed graph:
  - Nodes: webpages
  - Edges: links between webpages
- Terms (Suppose we have a page A that links to page B):
  - Page A has a <u>forward-link</u> to page B
  - Page B has a <u>back-link</u> from page A

The Magic Formula

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

The Magic Formula

Node *u* is the vertex (webpage) we're interested in computing the PageRank of

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

The Magic Formula

R'(u) is the PageRank of Node *u*

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

The Magic Formula

c is a normalization factor that we can ignore for our purposes

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

The Magic Formula

E(u) is a "personalization" factor that we can ignore for our purposes

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

The Magic Formula

We sum over all backlinks of *u*: the PageRank of the website *v* linking to *u* divided by the number of forward-links that *v* has

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u)$$

But wait! This is Recursive!

- Uh oh! We have a recursive formula with no base-case
- We rely on convergence
  - Choose some initial PageRank value for each site
  - Simultaneously compute/update PageRanks
  - When our Delta is small between iterations:
    - Stop, call it "good enough"

Sounds Easy. Why MapReduce?

- Assume in the best case that we've crawled and captured the internet as a series of (url, outgoing links) pairs
- We need about 50 iterations of the PageRank algorithm for it to converge
- We quickly see that running it on one machine is not viable

Building a Web Index using PageRank

- Scrape Webpages
- Strip out content, keep only links (input is key = url, value = links on page at url)
  - This step is actually pushed into the MapReduce
- Feed into PageRank Mapreduce
- Sort Documents by PageRank
- Post-process to build the indices that our Google RLP example used

Using MapReduce to Compute PageRank, Step 1

- Map:
  - Input:
    - key = URL of website
    - val = source of website
  - Output for each outgoing link:
    - key = URL of website
    - val = outgoing link url
- Reduce:
  - Input:
    - key = URL of website
    - values = Iterable of all outgoing links from that website
  - Output:
    - key = URL of website
    - value = Starting PageRank, Outgoing links from that website

Using MapReduce to Compute PageRank, Step 2

- Map:
  - Input:
    - key = URL of website
    - val = PageRank, Outgoing links from that website
  - Output for each outgoing link:
    - key = Outgoing Link URL
    - val = Original Website URL, PageRank, # Outgoing links
- Reduce:
  - Input:
    - key = Outgoing Link URL
    - values = Iterable of all links to Outgoing Link URL
  - Output:
    - key = Outgoing Link URL
    - value = Newly computed PageRank (using the formula), Outgoing links from document @ Outgoing Link URL

Repeat this step until PageRank converges – chained MapReduce!

Using MapReduce to Compute PageRank, Step 3

- Finally, we want to sort by PageRank to get a useful index
- MapReduce's built in sorting makes this easy!

- Map:
  - Input:
    - key = Website URL
    - value = PageRank, Outgoing Links
  - Output:
    - key = PageRank
    - value = Website URL

Using MapReduce to Compute PageRank, Step 3

- Reduce:
  - In case we have duplicate PageRanks
  - Input:
    - key = PageRank
    - value = Iterable of URLs with that PageRank
  - Output (for each URL in the Iterable):
    - key = PageRank
    - value = Website URL
- Since MapReduce automatically sorts the output from the reducer and joins it together:
- We're done!

Using the PageRanked Index

- Do our usual keyword search with RLP implemented
- Take our results, sort by our pre-generated PageRank values
- Send results to user!
- PageRank is still the basis for Google Search
  - (of course, there are many proprietary enhancements in addition)

Further Reading (Optional)

- Some PageRank slides adapted from http://www.cs.toronto.edu/~jasper/PageRankForMapReduceSmall.pdf
- PageRank Paper:
  - Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.*