Analysis of massive data sets

http://www.fer.hr/predmet/avsp

Prof. dr. sc. Siniša Srbljić

Doc. dr. sc. Dejan Škvorc

Doc. dr. sc. Ante Đerek

Faculty of Electrical Engineering and Computing
Consumer Computing Laboratory

Analysis of Massive Data Sets: MapReduce Programming Model

Marin Šilić, PhD

Overview

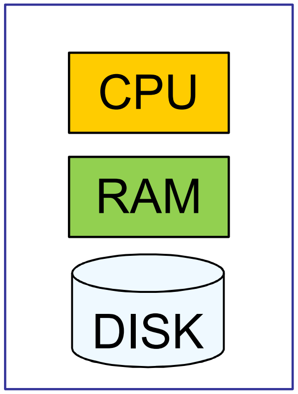
- Motivation
- Storage Infrastructure
- Programming Model: MapReduce
- MapReduce: Implementation
- MapReduce: Refinements
- Problems Suited for MapReduce
- MapReduce Other Implementations

Motivation

- **Modern Data Mining Applications**
  - Examples
    - Ranking Web pages by importance
    - Query friends networks on social networking site
  - What is in common?
    - Require processing of large amount of data
    - Data is often regular
    - Idea is to exploit parallelism

Motivation

- **Single Node Architecture**
  - Most of the computing done on a single node



All the data fits in a RAM of single node

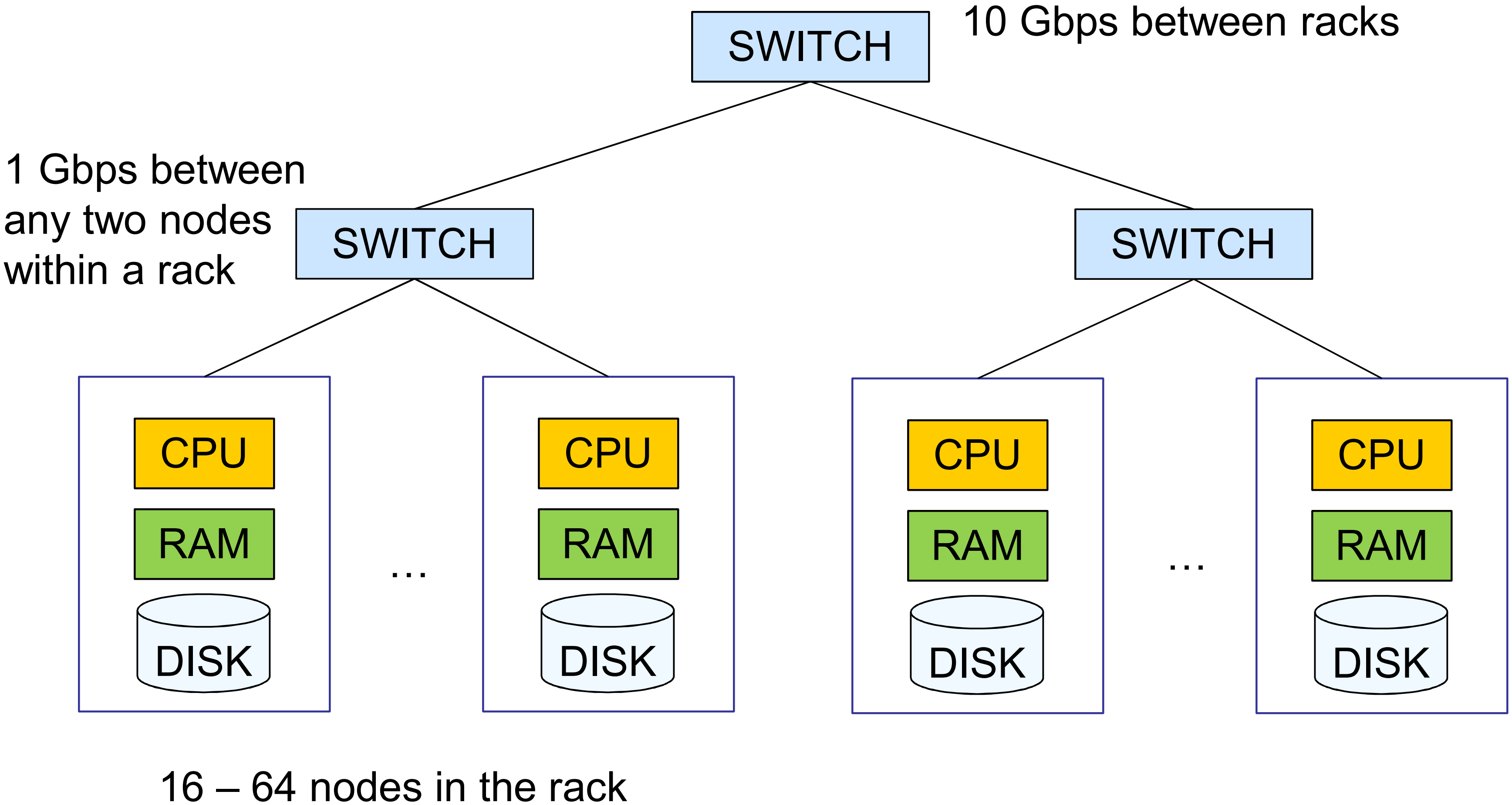
Machine Learning Statistics Data Mining

Motivation

- **Parallelism in the past**
  - Scientific application
    - Large amount of calculations
    - Done on special purpose computers
    - Multiple processors, specialized hardware etc.
- **Parallelism today**
  - Prevalence of the Web
    - Computing is done on installations of large amount *ordinary* computing nodes
    - The costs are greatly reduced compared with usage of special purpose parallel machine

Motivation

- **Cluster Architecture**



Motivation

- **Google Example**
  - 20+ billion websites x 20KB = 400+ TB
    - It takes ~1000 hard drives to store the Web!
  - 1 computer reads 30-35 MB/sec
    - It takes 4 months to read the Web!
  - Challenge is to do something useful with the data
  - Nowadays, a standard architecture for such computation is used
    - Cluster of commodity Linux nodes
    - Ethernet network to connect them

Motivation

- **Large-scale Computing for data mining on commodity hardware**
- **Challenges**
  i. How to distribute computing?
  ii. Make easy to write distributed programs?
  iii. Incorporate fault-tolerance
     - Machine may operate up to 1000 days
     - Suppose you have 1000 machines, expected lost is 1 per day
     - However, people estimate Google posses more than 1M machines
       - 1000 machines fail every day!

Motivation

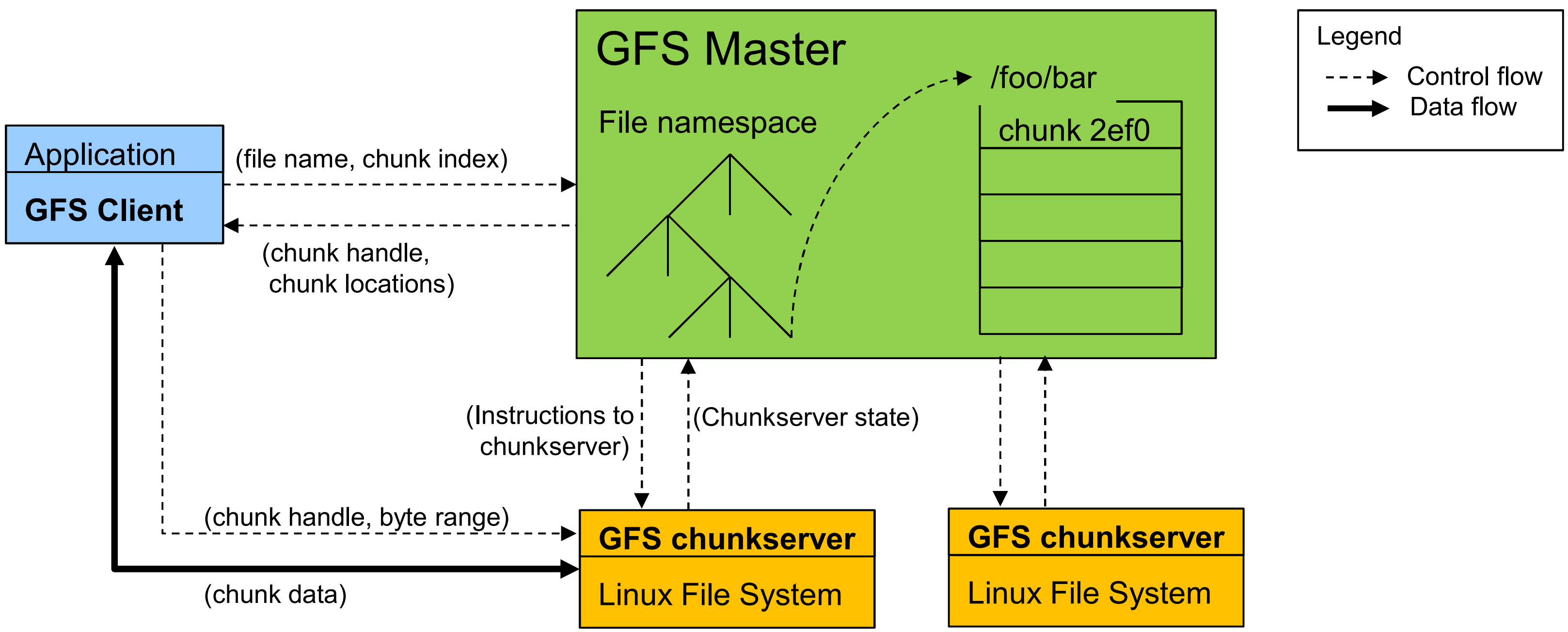
- **Key Ideas**
  - Bring the computation to the data
  - Store files multiple times for reliability
- **MapReduce addresses the challenges**
  - Google's computational and data manipulation model
  - Very convenient to handle Big Data
  - Storage infrastructure – File System
    - GFS, HDFS
  - Programming model
    - MapReduce

Storage Infrastructure

- **Distributed File System**
  - Google: GFS, Hadoop: HDFS
  - Provides global namespace
  - Specifically designed for Google’s needs
- **Usage pattern**
  - Huge files (100s of GB, up to TB)
  - Data is rarely updated in place
  - The dominant operations are
    - Appends
    - Reads

Storage Infrastructure

- **GFS Architecture**
  - GFS cluster consists of:
    - Single *master*, multiple *chunkservers* and multiple *clients*

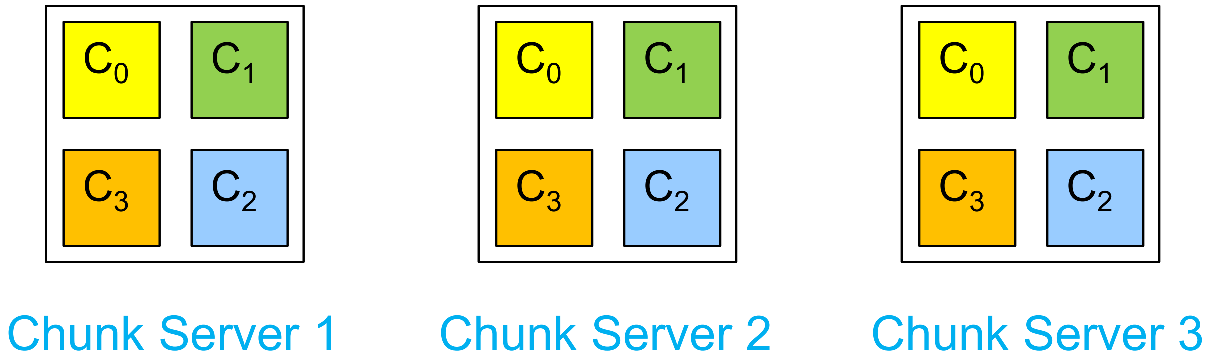


Storage Infrastructure

- **GFS Architecture**
  - Chunk Servers
    - File is split into contiguous chunks
    - Chunk size is 64MB
    - Each chunk is replicated (2x or 3x)
    - Replicas in different racks
  - Master Node
    - Stores metadata about where files are stored
    - Replicated – shadow master
  - Client Library
    - Contacts master to locate chunk servers
    - Directly accesses data from chunk servers

Storage Infrastructure

- **Reliable Distributed System**
  - Data is kept in chunks replicated on different machines
    - Easy to recover from disk or machine failure



  - Brings the computation to the data
  - Chunk Servers are also used as computing nodes

Programming Model: MapReduce

- **Motivating example**
  - Let's imagine a huge document
  - Count the number of times each distinct word occurs
  - For example
    - Mining some web server logs, counting URLs
    - File too large to fit in memory
    - All word pairs `<word, count>` fit in memory
    - Solution on linux: `words(doc.txt) | sort | uniq -c`
      - `words` takes a file and outputs each word in a line

Programming Model: MapReduce

- **The High-level Overview of MapReduce**
  - Read the data sequentially
  - Map:
    - Identify the entities you care about
  - Group entities by key:
    - Sort and Shuffle
  - Reduce:
    - Aggregate, count, filter, transform
  - Write the results

Programming Model: MapReduce

- **Map Phase**

Input key-value pairs

`<key1, value1>`

`<key2, value2>`

`<key3, value3>`

…

`<keyN, valueN>`

Map Phase

Intermediate key-value pairs

`<key1, value1>`

`<key2, value2>`

`<key3, value3>`

…

`<keyM, valueM>`

Programming Model: MapReduce

- **Group Phase**
  - Performed by the framework itself

Intermediate key-value pairs

<key1, value1>

<key2, value2>

<key3, value3>

…

<key*M*, value*M*>

Group Phase

Key-value groups

<key1, [value1, value2, …, value*N*]>

<key2, [value1, value2, …, value*N*]>

<key3, [value1, value2, …, value*N*]>

…

<key*M*, [value1, value2, …, value*N*]>

Programming Model: MapReduce

- **Reduce Phase**

Key-value groups

<key1, [value1, value2, …, value*N*]>

<key2, [value1, value2, …, value*N*]>

<key3, [value1, value2, …, value*N*]>

…

<key*M*, [value1, value2, …, value*N*]>

Reduce Phase

Output key-value pairs

<key1, value1>

<key2, value2>

<key3, value3>

…

<key*M*, value*M*>

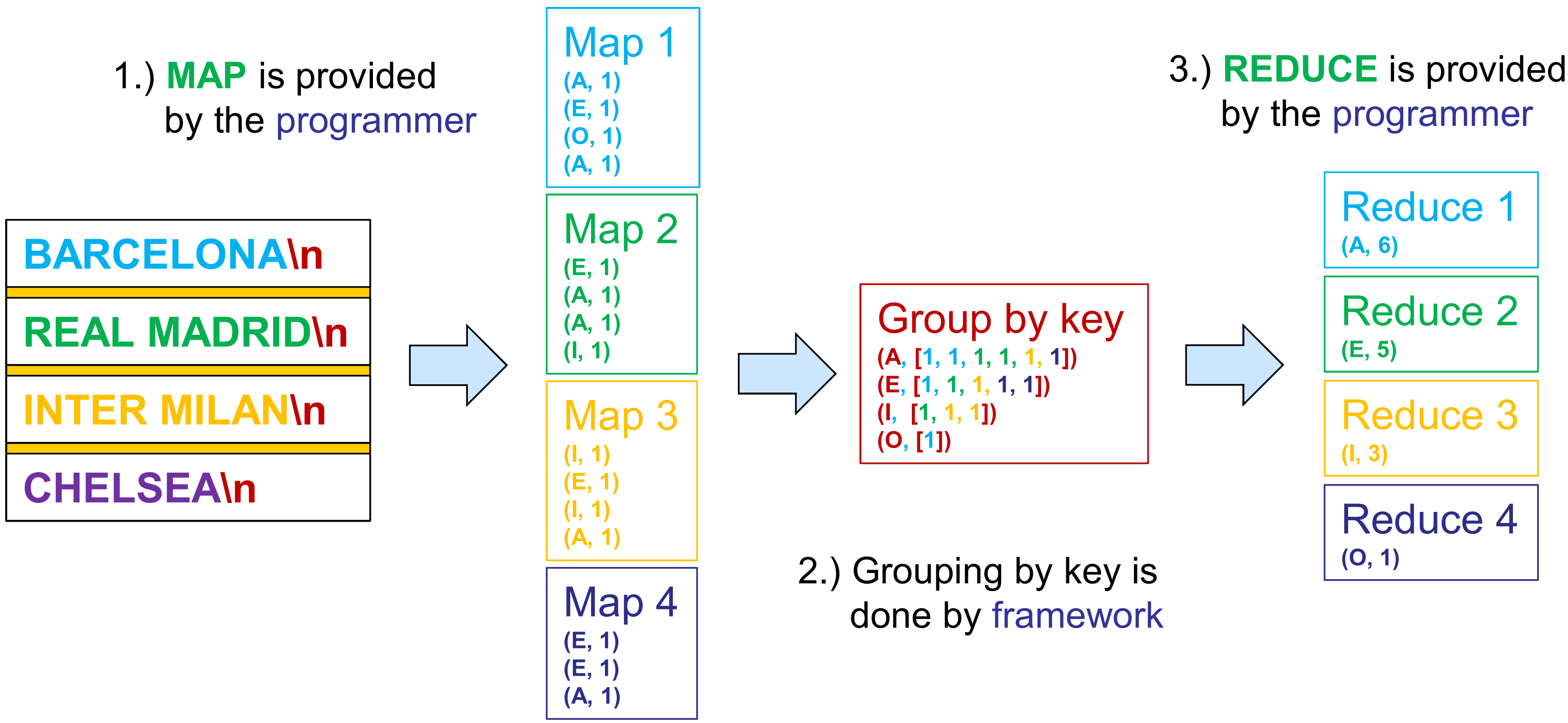
Programming Model: MapReduce

- **Programmer implements two methods:**
  - **Map(k, v) → <k’, v’>***
    - Input is a key-value pair and out put is a set of key-value pairs
      - Key might the filename and value is a line in the file
    - For each input key-value pair (k, v) there is one Map
  - **Reduce(k’ <v’>*) → <k’, v’’>***
    - All values v’ with the same key k’ are processed together
    - There is one Reduce function per each unique key k’

Programming Model: MapReduce

MapReduce Letter Counting Example

- File containing 4 string lines, count the number of each vowel letter in the file using MapReduce



Programming Model: MapReduce

- **The programmer needs to provide MAP and REDUCE implementation**

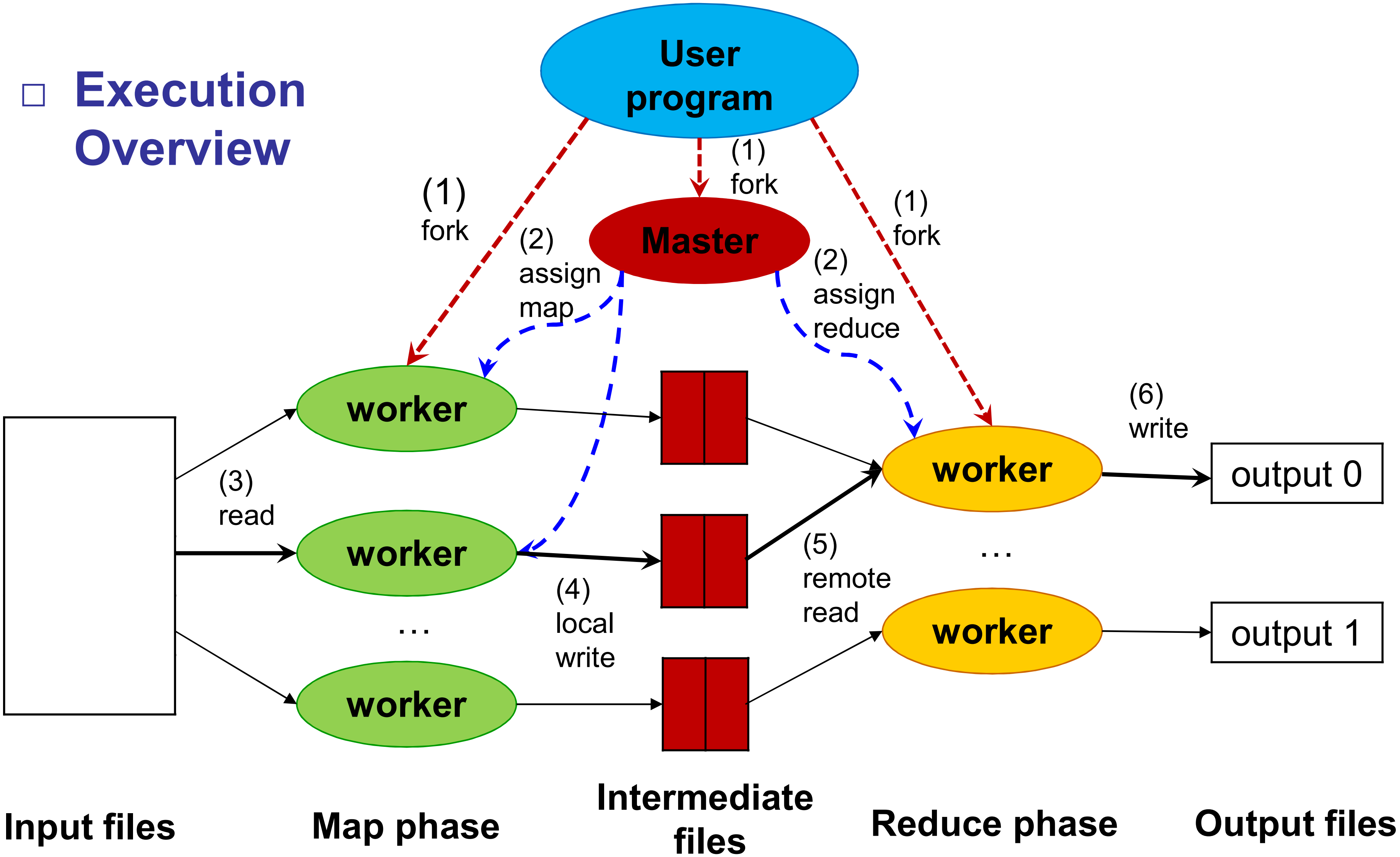
```
map (key, value):
// key: document name, value: text of the document
      for each char c in value:
            if c is vowel:
                  emit(c, 1)

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

Programming Model: MapReduce

- **MapReduce framework deals with**
  - Partitioning the input data on multiple mappers
  - Scheduling the program execution on a set of distributed nodes
  - Grouping and aggregating values by key
  - Ensuring fault tolerance
  - Handling inter-machine communication

MapReduce: Implementation

- Execution Overview



MapReduce: Implementation

- **Execution Overview**
  1) Split & Fork
     - The MapReduce library splits the input files into M pieces typically 16 – 64 MB.
     - Then, it starts a lot of copies of the program on a cluster of machines
  2) Task assignment
     - One copy is special – master, the rest are workers
     - Master chooses M map and R reduce workers

MapReduce: Implementation

- **Execution Overview**
  3) Map
     - Each Map worker reads one of the input chunks
     - Parses key-value pairs from the input
     - Passes each pair to user-defined Map function
     - Buffers the `Map` function output to local memory
  4) Local write
     - Map output is written to local disk
     - Partitioned into R regions via partitioning function
     - Location of the partitions is passed to master

MapReduce: Implementation

- **Execution Overview**
  5) Grouping and sorting
     - Masters notifies reduce worker of partitions locations
     - Reduce worker reads the intermediate data by RPC
     - It sorts the intermediate data by the intermediate key so all the data is grouped by that key
     - Typically, many different keys map to the same reduce worker
     - If the amount of intermediate data is too large, an external sort is used

MapReduce: Implementation

- **Execution Overview**
  6) Reduce
     - The reduce worker iterates over the sorted intermediate data
     - For each unique intermediate key, it passes the key and the corresponding set of intermediate values to user's defined `Reduce` function
     - The output of the `Reduce` function is appended to a final output file for this reduce partition
  7) Finalizing
     - Master wakes up the user program
     - At this point, `MapReduce` call returns back to the user code

MapReduce: Implementation

- **Master data structures**
  - Task status (idle, in-progress, completed)
  - Identity of worker machines
  - The locations and sizes of R intermediate file regions produced by map task
    - The master pushes these locations to reduce tasks
    - The master pings every worker periodically

MapReduce: Implementation

- **Fault Tolerance**
  - Worker failure
    - Master pings worker
    - If no response is received, the worker is marked as failed
    - Any completed map task completed by worker is reset to idle
      - The task is rescheduled and assigned to some other worker
    - Any map or reduce task in progress on a failed worker is also reset to idle and it gets rescheduled on some other worker
    - Map tasks that are completed are re-executed since the output is stored on a local disk of the failed worker
    - Completed reduce tasks do not need to be re-executed since their output is stored in a global file system

MapReduce: Implementation

- **Fault Tolerance**
  - Worker failure
    - A(Map) → B(Map)
      - All reduce workers need to get notified
      - Any reduce task that has not already read the data from worker A will read the data from worker B
  - Master failure
    - If the master dies, MapReduce task gets aborted
    - The client gets notified and can retry MapReduce operation

MapReduce: Implementation

- **Locality**
  - Input data is stored on the local disks of the machines that make up cluster
  - The master takes care of the input files locations while attempting to schedule map tasks
    - The idea is to schedule map tasks to nodes where the input data is stored
    - Failing that, it tries to schedule map tasks near the data (on the machine that is on the same network switch)
    - Large MapReduce operations on a significant fraction of the workers in a cluster
      - Most input data is read locally
      - It consumes no network bandwidth

MapReduce: Implementation

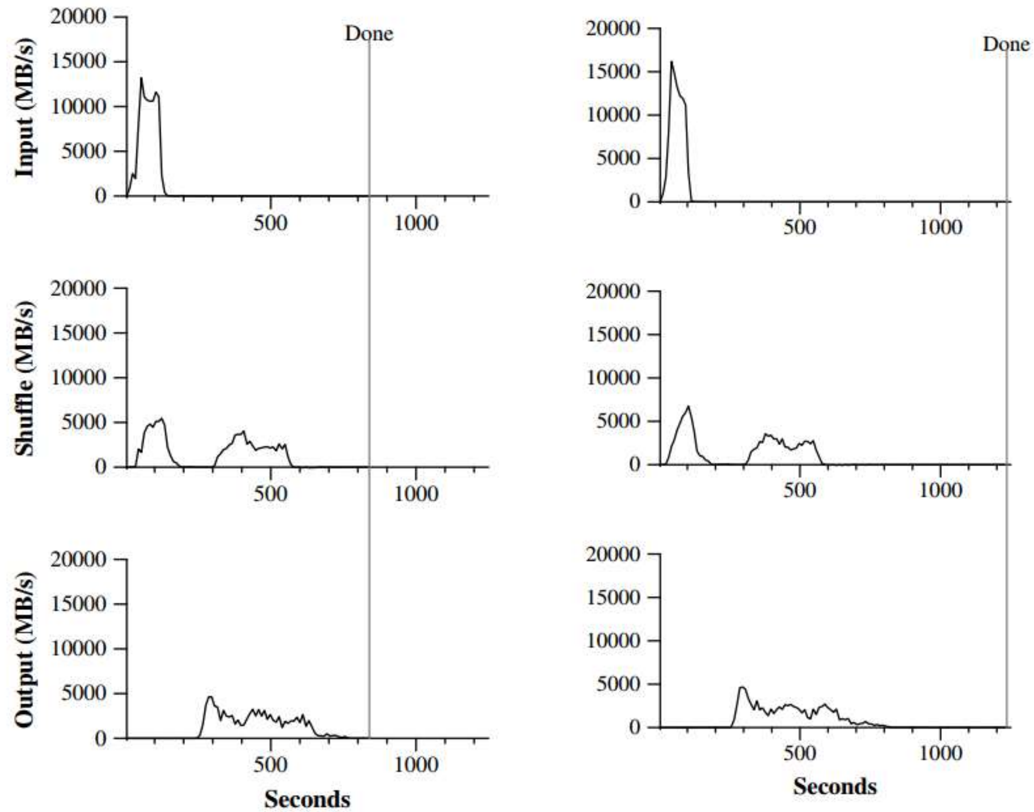
- **How many Map and Reduce tasks (granularity)?**
  - M Map and R Reduce workers
  - Ideally, M and R should be much larger than the number of workers
    - Improves dynamic load-balancing and speeds up recovery when a worker fails
  - In practice…
    - M is chosen so that each task deals with 16 – 64MB of the input data (locality!!!) *M ~ 200k*
    - R is often constrained by the user (separate output) *R ~ 5k*
    - R is a small multiple of the number of workers *NoW ~ 2k*
  - Physical bounds
    - O(M + R) scheduling decisions, O(M*R) states

MapReduce: Implementation

- **Backup Tasks**
  - Common cause that lengthens the total execution time
  - A "straggler" – a machine that takes unusually long to complete its task
    - Bad disk, 30MB/s to 1MB/s
    - Lack of CPU, IO, bandwidth due to some other scheduled task
  - General mechanism
    - When the MapReduce operation is close to completion, the master schedules backup executions of the remaining tasks
    - The task is marked completed whenever the primary or the backup execution completes.
    - It significantly reduces the completion time

MapReduce: Implementation

- Backup Tasks
  - Sort example
    - 10^{10} 100-byte records
    - 1800 machines
    - M = 15000
    - R = 4000
  - It takes 44% longer with mechanism disabled!!!



(a) Normal execution

(b) No backup tasks

MapReduce: Refinements

- **Partitioning Function**
  - User specifies the number of reduce tasks R
  - Data gets partitioned using partitioning function on intermediate key
  - Default partitioning function: **`hash(key) mod R`**
  - However, it is not appropriate sometimes
    - For example, output keys are URL and we want all entries on the same host to end up in the same output file
    - To support such needs, user can provide special partitioning function
      - For example: **`hash(hostname(urlkey)) mod R`**

MapReduce: Refinements

- **Ordering Guarantees**
  - Within a given partition, the intermediate key/value pairs are processed in increasing key order
  - Easy to generate sorted output per partition
- **Skipping Bad Records**
  - Map and Reduce crash deterministically
    - Bug in the user code
    - Acceptable to ignore some records (statistical analysis)
  - Detect records that cause crashes and skip them

MapReduce: Refinements

- **Combiners**
  - Significant repetition in the intermediate keys produced by each map: `<k, v1>, <k, v2>,` … for the same key
    - Reduce function is commutative and associative
    - Word counting is a good example `<the, 1>`
  - MapReduce framework allows user to specify optional Combiner function
    - It performs partial merging before the data is sent over the network
    - Combiner function is executed on machines that perform map task
    - Typically, the same code is used both for combiner and reduce functions

MapReduce: Refinements

- **IO Format Types**
  - MapReduce library provides support for reading input data in several formats
    - For example, “text” mode treats each line as a key-value pair
    - Another common supported format stores a sequence of key-value pairs sorted by key
    - Each input format knows how to split input data meaningfully
    - Users can provide new additional input types by implementing interface `reader`
  - In a similar way, a set of output formats is supported
    - It is easy for user to add support for new types

Problems Suited for MapReduce

- **Count of URL Access Frequency**
  - The Map function processes logs of web page requests
    - Outputs `(URL, 1)`
  - Reduce adds together all values for the same URL
    - Emits `(URL, total count)`
- **Reverse Web-Link Graph**
  - Map: `(target, source)` for each link to a target URL found in a page named source
  - Reduce: concatenates all source URLs associated with the target URL, `(target, list(source))`

Problems Suited for MapReduce

- **Distributed Grep**
  - Map: emit a line if it matches a pattern
  - Reduce: copy the supplied data to output
- **Distributed Sort**
  - Map: extract the key from each record
    - Emits (key, record)
  - Reduce – emit all pairs unchanged
    - Ordering guarantee and partitioning function

Problems Suited for MapReduce

- **Term-vector per Host**
  - A term vector summarizes the most important words
    - A list of `<word, frequency>` pairs
  - Map function emits a `<hostname, term vector>` pair for each input document
    - Hostname is extracted from the document URL
  - Reduce function is passed all term vectors for a given host
    - It adds these term vectors together
    - It throws away infrequent terms
    - Emits final `<hostname, term vector>`

Problems Suited for MapReduce

- **Inverted Index**
  - The Map parses each document
    - It emits a sequence of `<word, document ID>`
  - The Reduce function accepts all pairs for a given word
    - It sorts document IDs
    - It emits `<word, list(document ID)>`
  - The set of output pairs forms a simple inverted index
    - The solution can be easily enhanced to keep track of word positions

Problems Suited for MapReduce

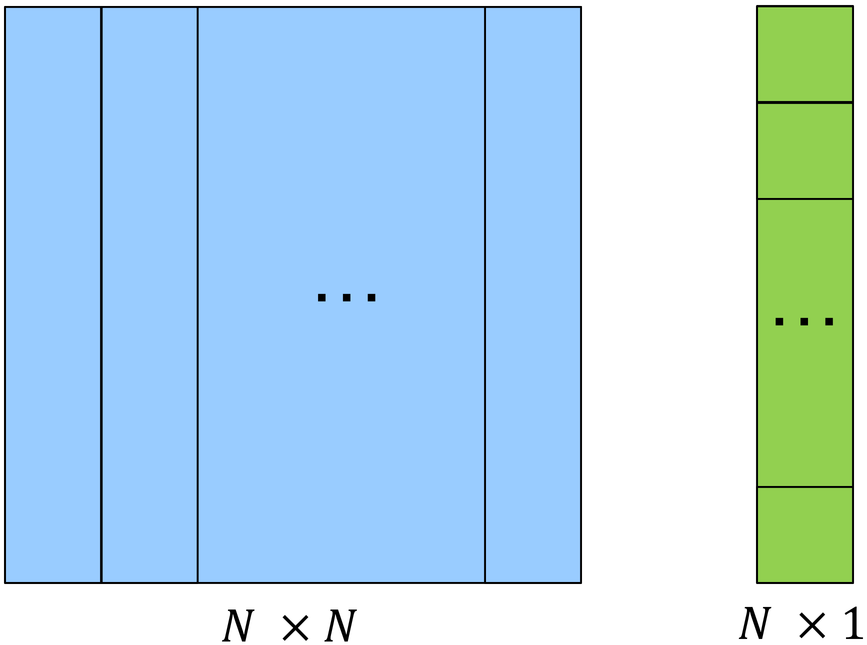
- **Matrix – Vector Multiplication**
  - $N \times N$ matrix M
    - $m_{i,j}$ - element in row i and column j
  - Vector $\vec{v}$ of length N with *j*th element v_j
  - $\vec{x} = M \times \vec{v}$
  - $x_i = \sum_{j=1}^{N} m_{ij} \cdot v_j$
  - M and $\vec{v}$ are stored in DFS
    - Case #1: N is large, but $\vec{v}$ can be stored in the memory
    - Case #2: N is large and it cannot fit in memory

Problems Suited for MapReduce

- **Matrix – Vector Multiplication Case #1**
  - Map function
    - Apply to one element of M
    - $\vec{v}$ is read first, to be available to all Map workers
    - For each m_{ij} map emits $(i, m_{ij} \cdot v_j)$
  - Reduce function
    - Sum all the values associated with the key i
    - The result is a sequence of pairs (i, x_i)

Problems Suited for MapReduce

- **Matrix – Vector Multiplication Case #2**
  - Vector $\vec{v}$ cannot fit in memory of a single machine



  - Split the matrix $\mathbf{M}$ into $\mathbf{R}$ stripes so each stripe can fit in memory
  - There are totally $N * R$ stripes combinations
    - Each **map** gets one combination

Problems Suited for MapReduce

- **Matrix Multiplication**
  - Performed in two MapReduce operations
  - $P = M \times N$
    - $p_{ik} = \sum_j m_{ij} \cdot n_{jk}$
  - Map1
    - For each m_{ij} emit $(j, (M, i, m_{ij}))$
    - For each n_{jk} emit $(j, (N, k, n_{jk}))$
  - Reduce1
    - Fer each key j examine the list of values
    - For each value from (M, i, m_{ij}) and from (N, k, n_{jk}) produce a key-value pair $((i, k), (m_{ij} \cdot n_{jk}))$

Problems Suited for MapReduce

- **Matrix Multiplication**
  - Performed in two MapReduce operations
  - $P \ = M \times N$
    - $p_{ik} = \sum_j m_{ij} \cdot n_{jk}$
  - Map2
    - Identity function $((i,k),v) \rightarrow ((i,k),v)$
  - Reduce2:
    - For each key (i,k) produce a sum of values with that key
    - Result: a sequence of pairs $((i,k),v)$ where $v = P_{ik}$

Problems Suited for MapReduce

- **Iterative Message Passing (Graph Processing)**
  - General pattern
    - There is a network of entities and relationships between them
      - It is required to calculate a state of each entity
      - The state of each entity is influenced by other entities in its neighborhood
    - The state can be…
      - Distance to other nodes
      - Indication that there is a node with a certain property
    - MapReduce jobs are performed iteratively
      - At each iteration each node sends messages to its neighbors
      - Each neighbor updates its state based on the received messages
    - Iterations are terminated by some condition
      - E.g. max number of iterations, negligible state change, etc.

Problems Suited for MapReduce

- Iterative Message Passing (Graph Processing)

```
map (key, value):
// key: node id, value: node object
      emit(key, value)
      for each neighbor m in value.neighbors:
            emit(m.id, get_message(value))

reduce (key, values):
// key: node id, values: received messages
      M = null
      messages = []
      for each message v in values:
            if v is Object:
                  M = v
            else:
                  messages.append(v)
      M.state = calculate_state(messages)
      emit(key, M)
```

Problems Suited for MapReduce

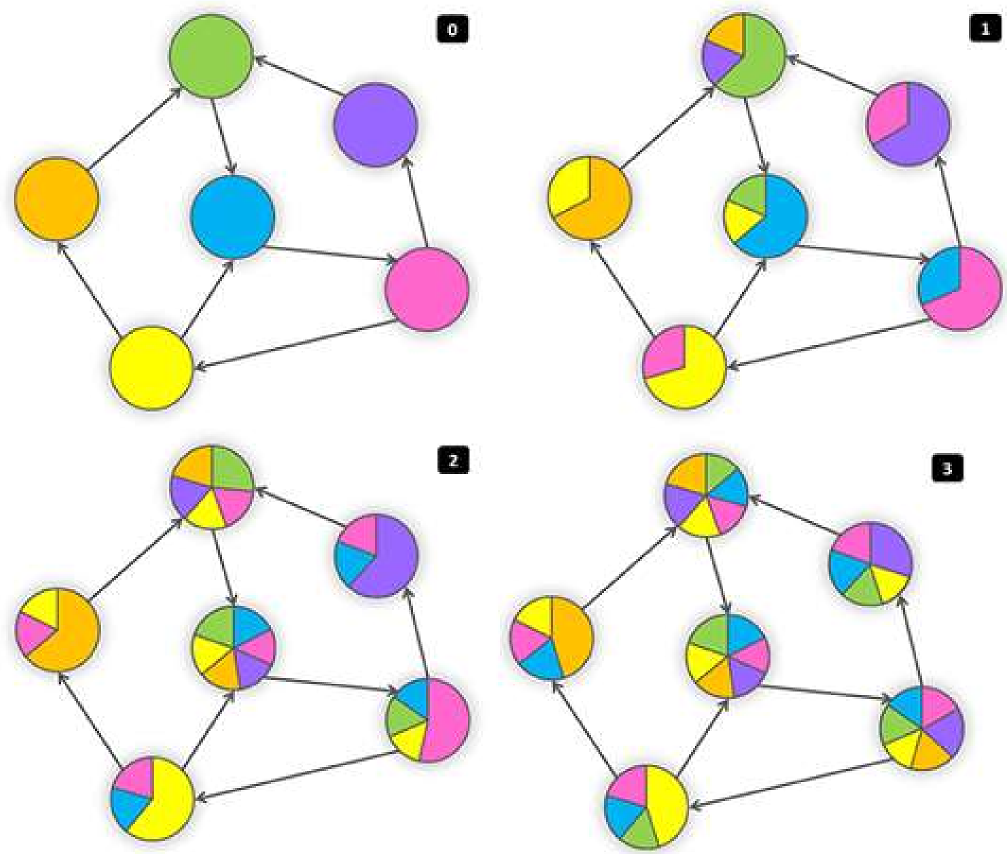
- **Iterative Message Passing (Graph Processing)**

Iteration

1.) Node passes its state to its neighbors

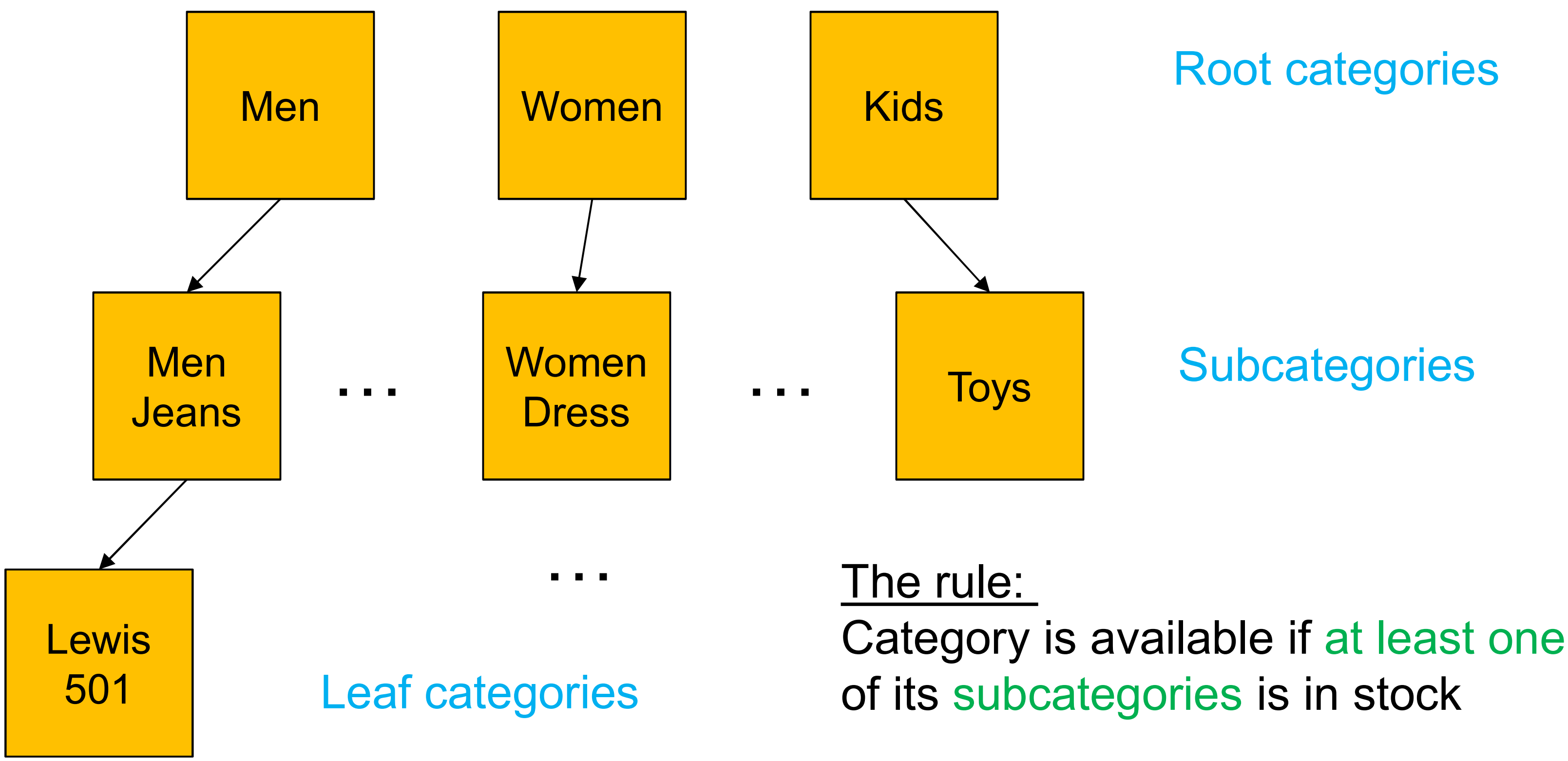
2.) Based on the received states of its neighbors, the node updates its own state

3.) The node passes its new state to its neighbors



Problems Suited for MapReduce

- **Iterative Message Passing (Graph Processing)**
  - Example:
    Availability propagation through the tree of categories



Problems Suited for MapReduce

- **Iterative Message Passing (Graph Processing)**
  - Example:
    Availability propagation through the tree of categories
  - Implementation:

```
get_message(node):
// node = {id, availability(initialized true or false)}
      return node.availability


calculate_state(values):
// values = {list of its subcategories availabilities}
      state = false
      for each availability in values:
            state = state or availability
      return state
```

Problems Suited for MapReduce

- **Database Queries**
  - In case queries are too large for common relational databases
  - Map task can read input data from database
- **Relational Algebra Operations**
  - Selection – condition C, $\sigma_C(R)$
  - Projection – subset S, $\pi_S(R)$
  - Union, intersection, difference
  - Natural join $R \bowtie S$
  - Grouping and aggregation

Problems Suited for MapReduce

- **Selection**
  - Condition C, $\sigma_C(R)$
  - Map: If a touple $t \in R$ satisfies C, emit (t, t)
  - Reduce: Identity
- **Projection**
  - Subset S, $\pi_S(R)$
  - Map:
    - $\forall t \in R$ construct a touple t' without components not in S
    - Emit (t', t')
  - Reduce: Duplicate elimination
    - Reduce $(t', [t', t', \dots, t'])$ into (t', t')

Problems Suited for MapReduce

- **Natural Join** $R(A,B) \bowtie S(B,C)$

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

$\bowtie$

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	B	C
a_3	b_2	c_1
a_3	b_2	c_2
a_4	b_3	c_3

$R \bowtie S$

- Map:
  - For input $R(a,b)$ emit $\big(b,(a,R)\big)$
  - For input $S(b,c)$ emit $\big(b,(c,S)\big)$
- Reduce: Match $\big(b,(a,R)\big)$ with $\big(b,(c,S)\big)$
  - Emit $(a,b,c\ )$

Problems Suited for MapReduce

- **Limitations of MapReduce**
  - Restricted programming framework
    - Tasks written as acyclic stateless dataflow programs
      - Repeated querying of datasets is difficult
      - Difficult to implement iterative algorithms that revisit a single working set multiple times
  - In case where computation depends on previously computed values
    - E.g., Fibonacci series, each value is a sum of previous two
    - If the dataset is small enough, compute it on a single machine
  - Algorithms that depend on shared global state
    - If task synchronization is required, MapReduce is not a choice
    - E.g. Online learning, Monte Carlo simulation

MapReduce Other Implementations

- **Google**
  - Not available outside of Google
- **Hadoop**
  - An open source implementation in Java
  - Uses HDFS for storage
- **Aster Data**
  - Cluster-optimized SQL Database that implements MapReduce

Literature – Further Reading

- Dean, Jeffrey and Ghemawat, Sanjay; „MapReduce: Simplified Data Processing on Large Clusters”, Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6, Pages 10-10 , 2004.
- Rajaraman, Anand, and Jeff Ullman; "Mining of Massive Datasets”, New York, NY, USA: Cambridge University Press, 2011.
- Leskovec, Jure, Rajaraman, Anand, and Ullman, Jeff; "Mining of Massive Datasets – Online Course”, http://www.mmds.org/

Literature – Further Reading

- Ilya Katsov, "MapReduce Patterns, Algorithms, and Use Cases", Highly Scalable Blog, February 1st 2012. https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/
- Jon Weissman, "Limitations of Mapreduce – where not to use Mapreduce", CSci 8980 Big Data and the Cloud, October 10th 2012. http://csci8980-2.blogspot.hr/2012/10/limitations-of-mapreduce-where-not-to.html