MapReduce

Cam Tu Nguyen

阮锦绣

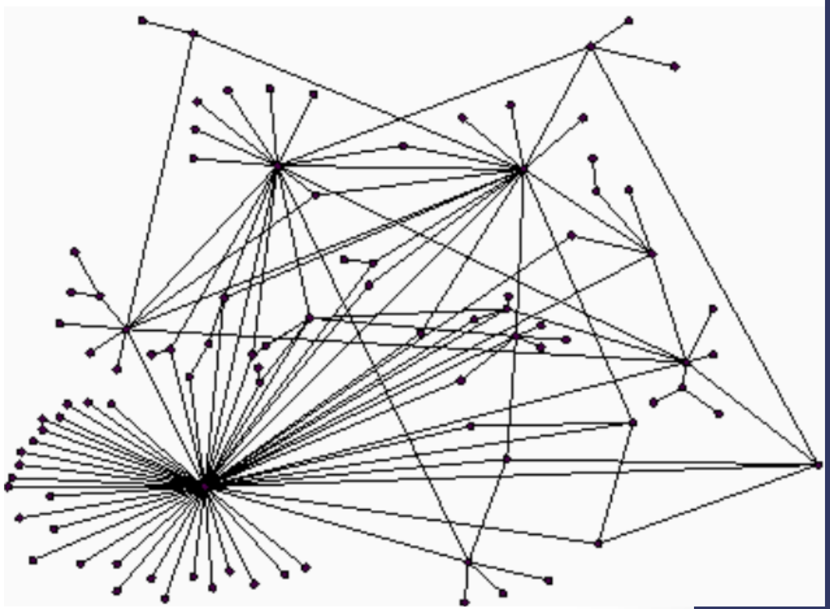
Software Institute, Nanjing University
nguyenct@lamda.nju.edu.cn
ncamtu@gmail.com

Outline

- Large-Scale Computing
- Distributed File Systems
- MapReduce & Algorithms Using MapReduce
  - Matrix-Vector Multiplication
  - Relational-Algebra Operations
  - Finding Frequent Itemsets with Map-Reduce
- The Communication-Cost Model

Motivation

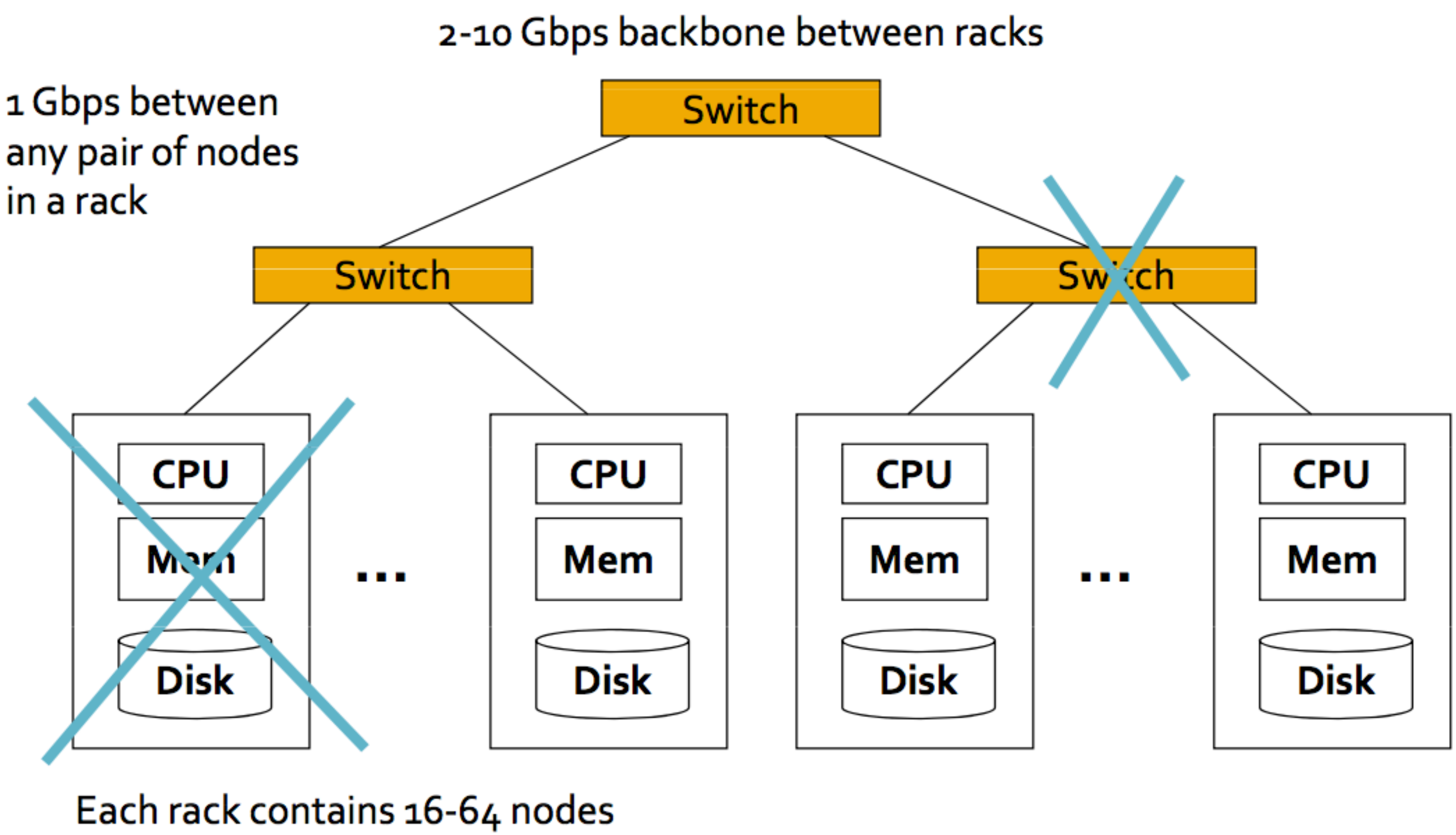
- Big-data analysis
  - The **ranking of Web pages by importance**, which involves an iterated matrix-vector multiplication where the dimension is many billions.
  - **Searches in "friends" networks at social-networking sites**, which involve graphs with hundred of millions of nodes and many billions of edges.



Large-Scale Computing

- Compute node: a single processor with its main memory, cache and local disks.
- In the past: **special-purpose parallel computers**
  - Many processors
  - Specialized hardware
- Recently: **cluster computing**
  - Thousands of compute nodes operating more or less independently
  - The compute nodes are commodity hardware
  - Greatly reduce the cost compared with special-purpose parallel machines.

Large-Scale Computing: Cluster Architecture



Large-Scale Computing

- **Large scale computing** for **data mining** problem on **commodity hardware**
  - PCs connected in a network
  - Need to process huge datasets on large clusters of computers
- **Challenges**:
  - How do you distribute computation
  - Distributed programming is hard
  - Machines fail
- **Map-reduce** address all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with big data.
- Note: Map-reduce is suitable for batch-operations
  - It is not for real-time operations

Idea and solution

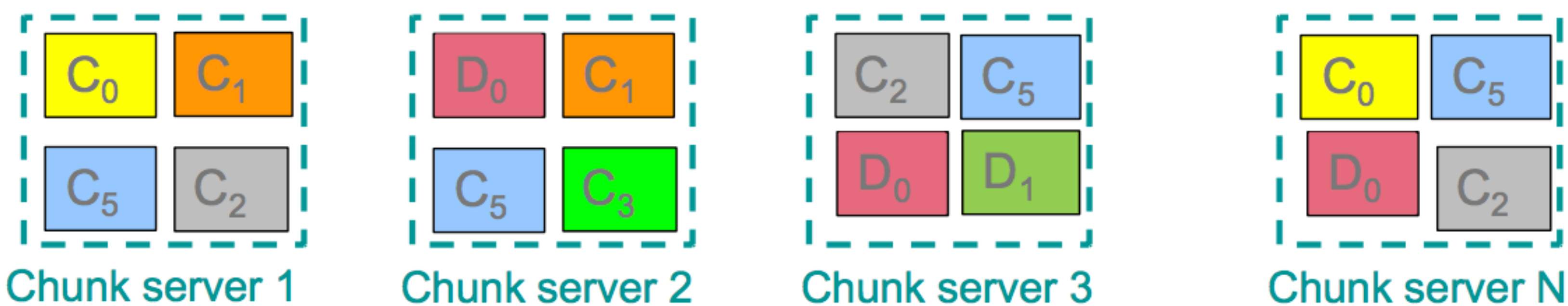
- **Idea**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Need**
  - Programming model
    - Map-Reduce
  - Infrastructure: distributed file system
    - Google: GFS
    - Hadoop: HDFS

Outline

- Large-Scale Computing
- Distributed File Systems
- MapReduce & Algorithms Using MapReduce
  - Matrix-Vector Multiplication
  - Relational-Algebra Operations
  - Finding Frequent Itemsets with Map-Reduce
- The Communication-Cost Model

Distributed File Systems

- Reliable distributed file system for petabyte scale
- Data kept in 64-megabyte “chunks” spread across thousands of machines
- Each chunk **replicated**, usually 3 times on different machines
  - Seamless recovery from disk or machine failure



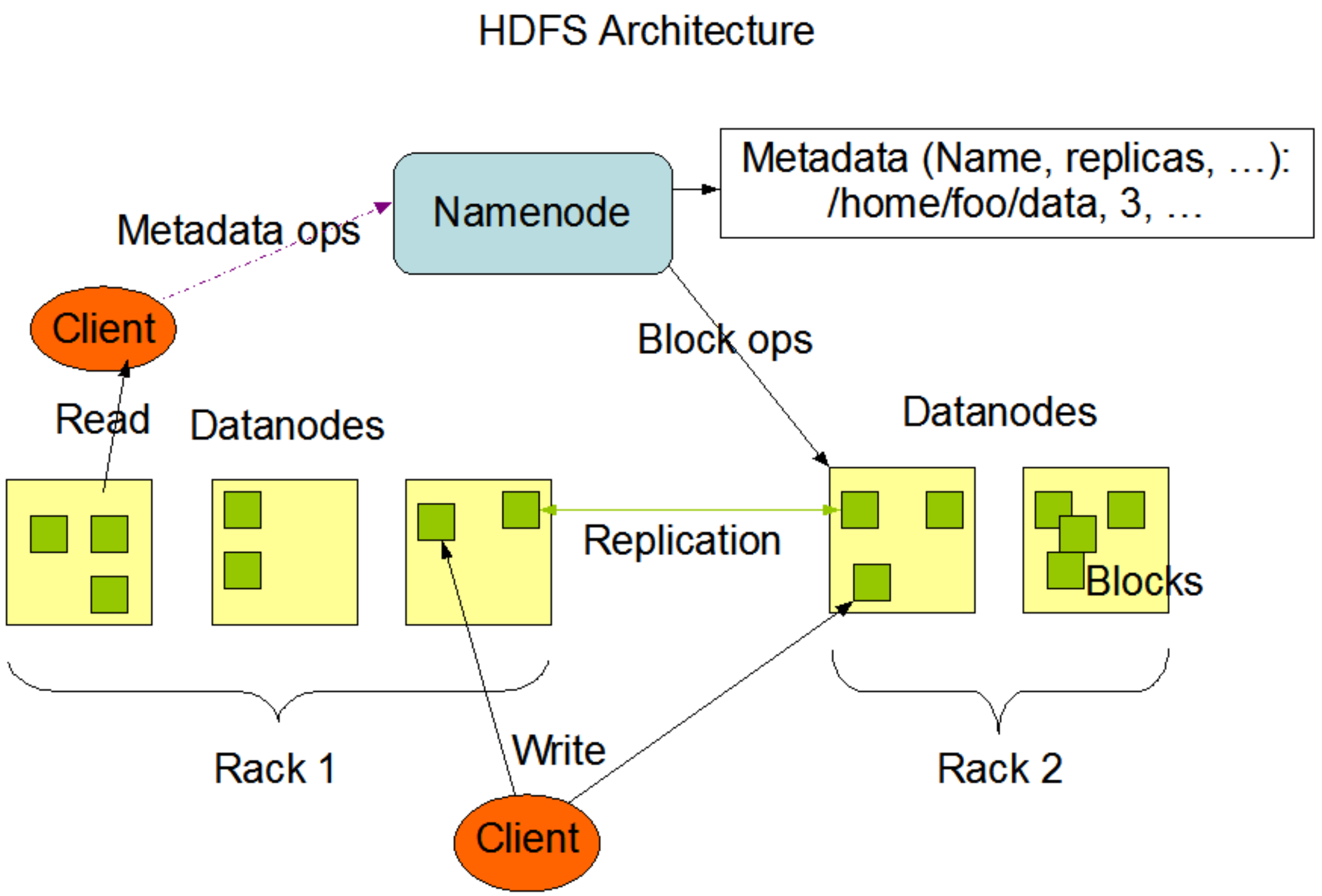
Bring computation directly to the data!

Distributed File Systems

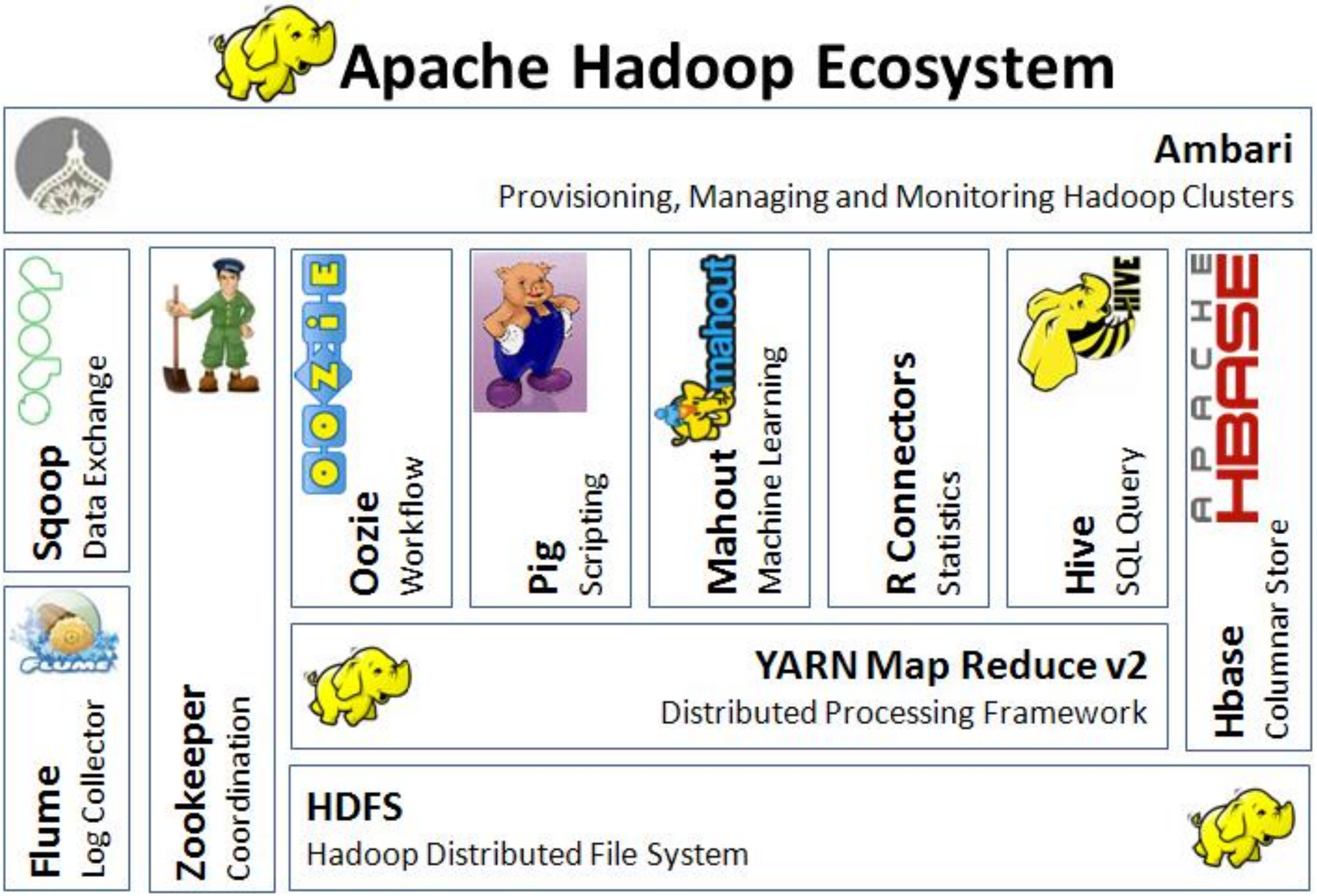
- Chunk Servers
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different tracks
- Master node
  - Stores meta data
  - Might be replicated
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

Distributed File Systems: HDFS

- HDFS (Hadoop File System): a part of Apache Hadoop subproject



Distributed File Systems: Hadoop ecosystem



Outline

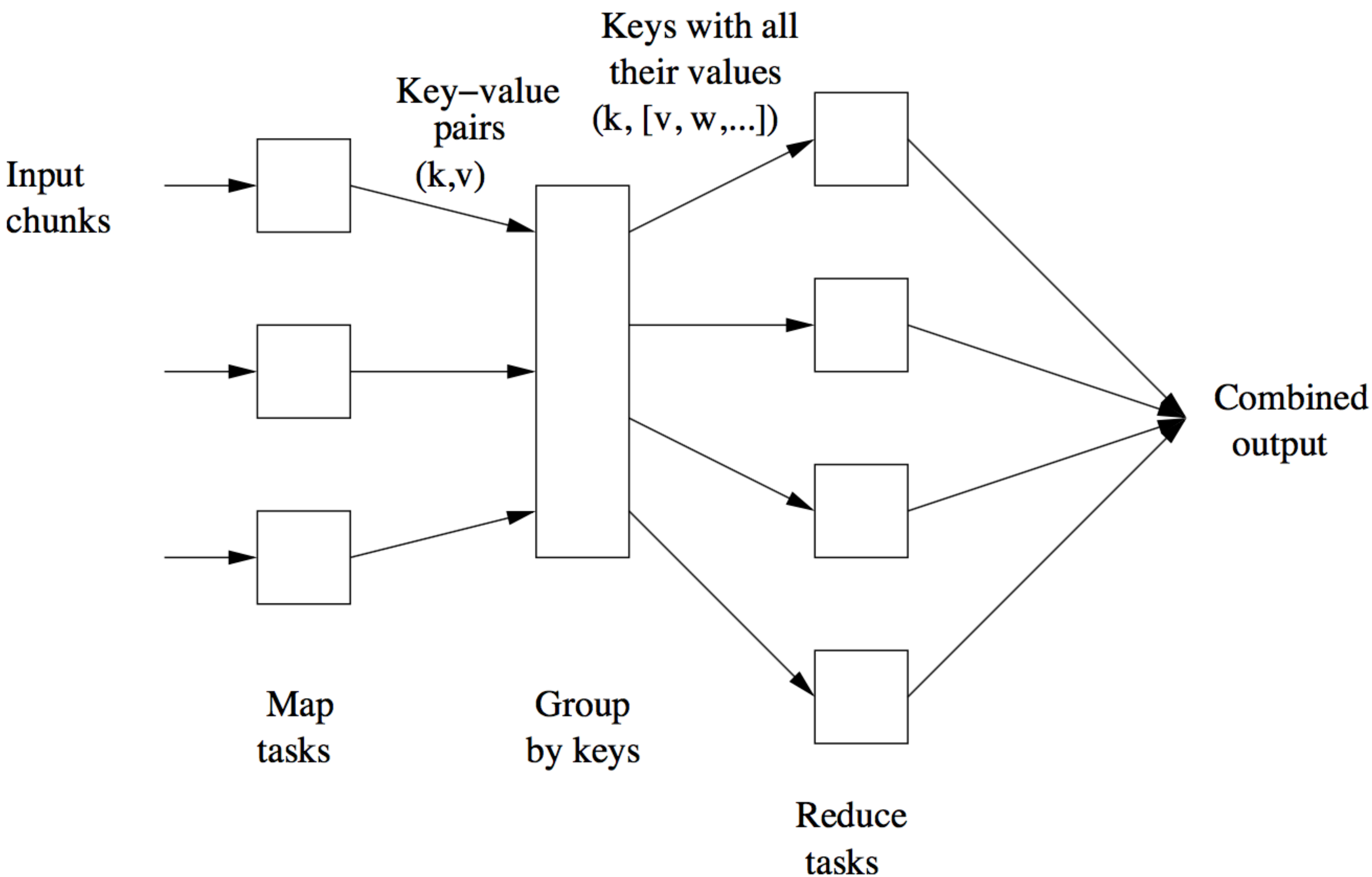
- Large-Scale Computing
- Distributed File Systems
- MapReduce & Algorithms Using MapReduce
  - Matrix-Vector Multiplication
  - Relational-Algebra Operations
  - Finding Frequent Itemsets with Map-Reduce
- The Communication-Cost Model

MapReduce

- MapReduce manages large-scale computations in a way that is tolerant of hardware faults
  - **Map:** Some number of **Map tasks** turn the chunks (from DFS) into a sequence of **key-value pairs**.
  - Sort and Shuffle:
    - The key-values pairs from each Map tasks are collected by a **master controller** and sorted by key.
    - The keys are divided among all the Reduce Task (using some kind of hash function so that pairs of same key end up in the same Reduce task)
  - **Reduce**: each key is associated with a list of values; Reduce taks aggregate, summarize, filer or transform data and provide output.

Outline stays the same, map and reduce change to fit the problem.

Schematic of a MapReduce Computation

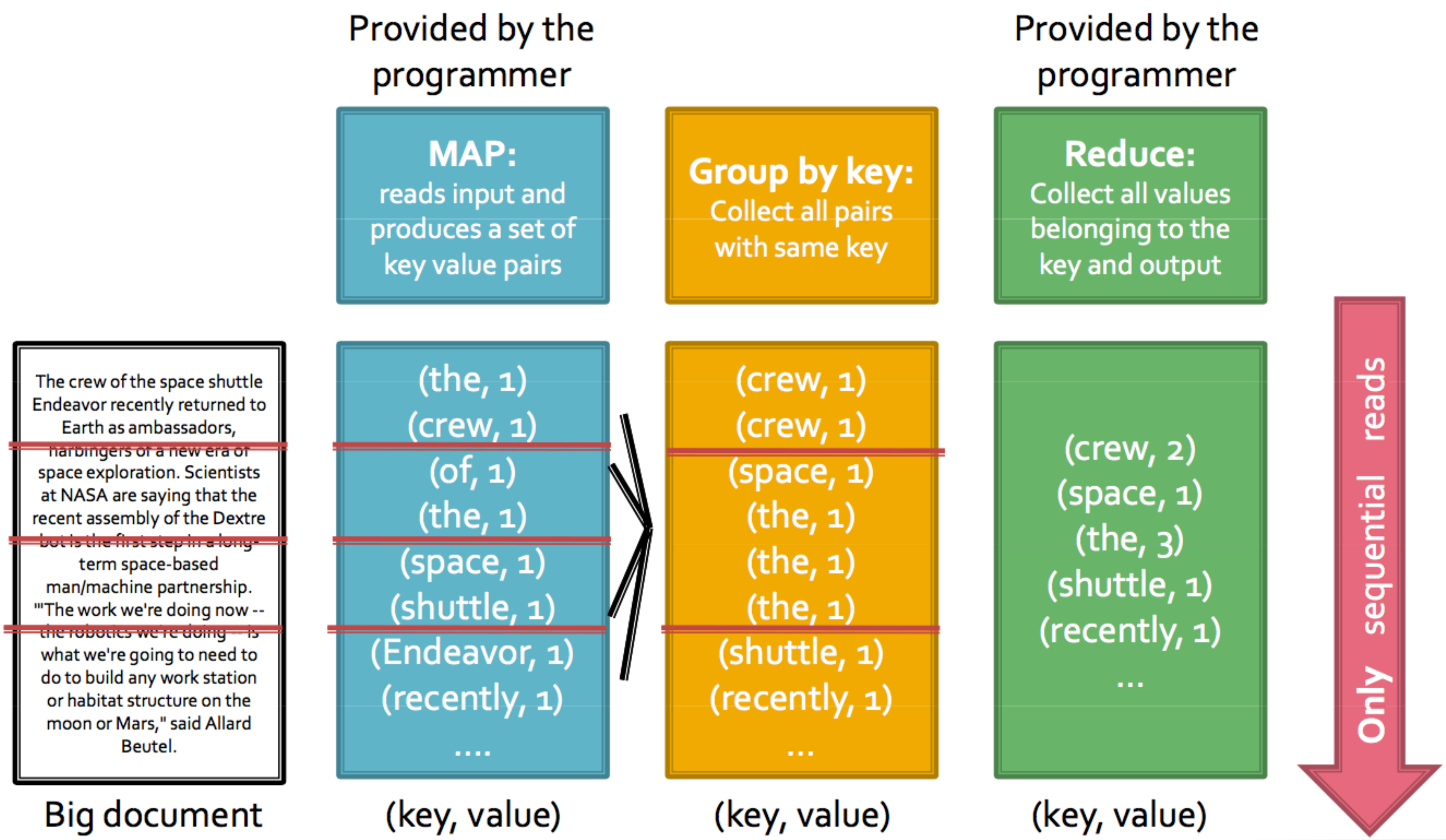


Word Count

- Given a large corpus of documents
- Count the number of times each distinct word occurs in the corpus
- **Sample application**
  - Analyze web server logs to find popular URLs

- The above problem captures the essence of MapReduce
  - Great thing is it is naturally parallelizable

Map-Reduce: Word counting



Map-Reduce Environment

- Map-Reduce environment takes care of:
  - Partitioning the input data
  - Scheduling the program's execution across a set of machines
  - Handling machine failures
  - Managing required inter-machine communication

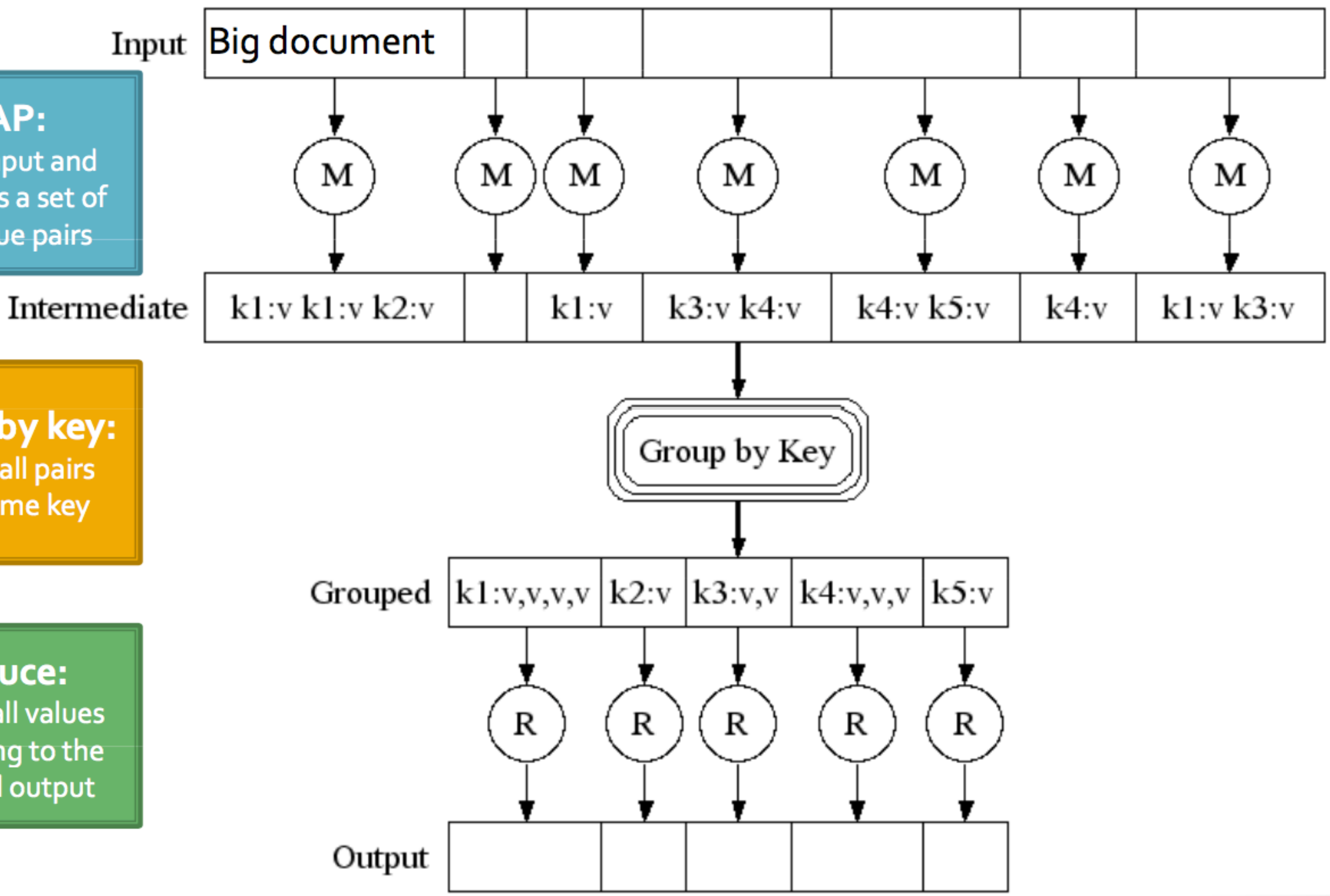
- Allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed cluster

Map-Reduce: A diagram

MAP:
reads input and produces a set of key value pairs

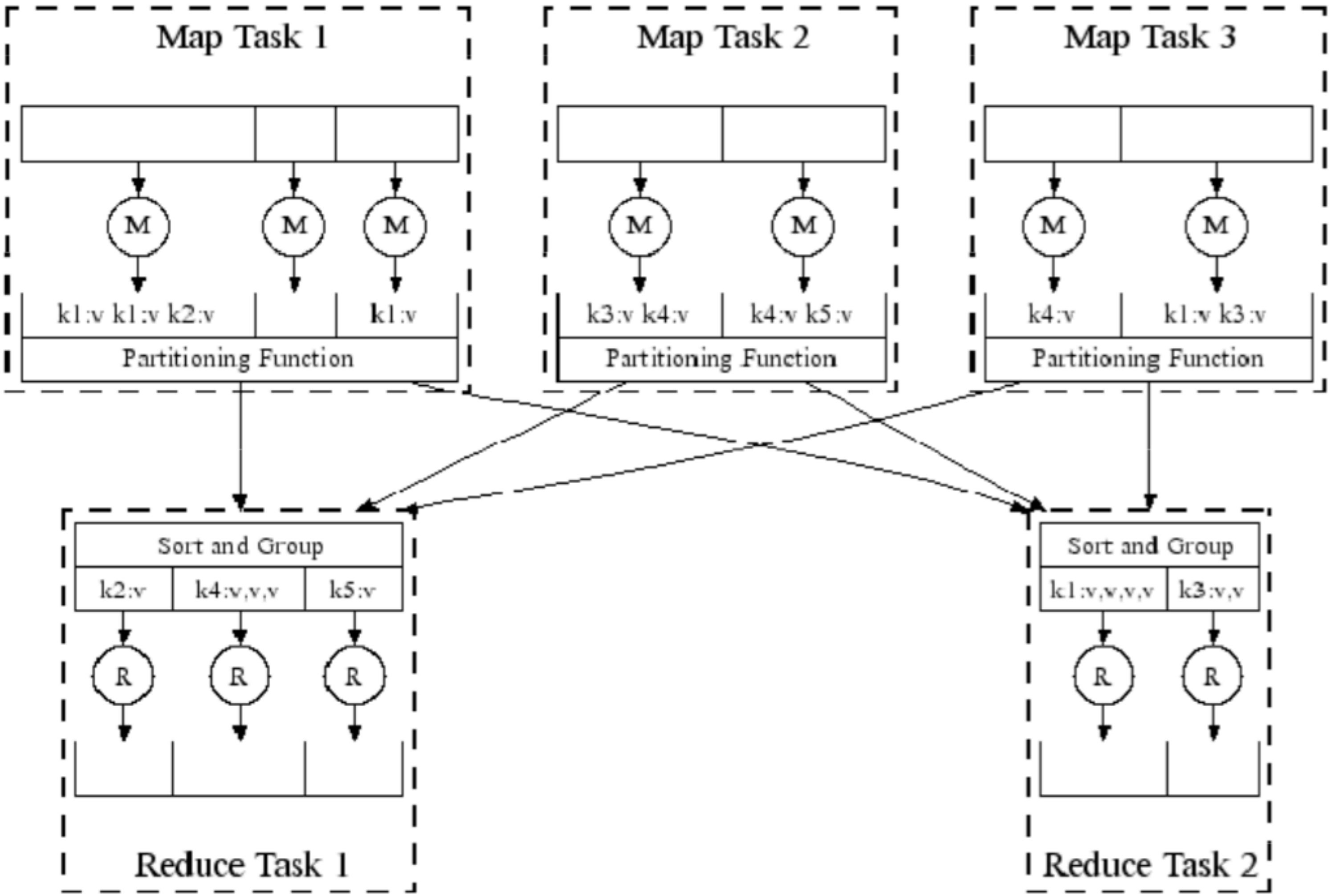
Group by key:
Collect all pairs with same key

Reduce:
Collect all values belonging to the key and output



Jure Leskovec, Stanford CS345a: Data Mining

Map-Reduce: In Parallel

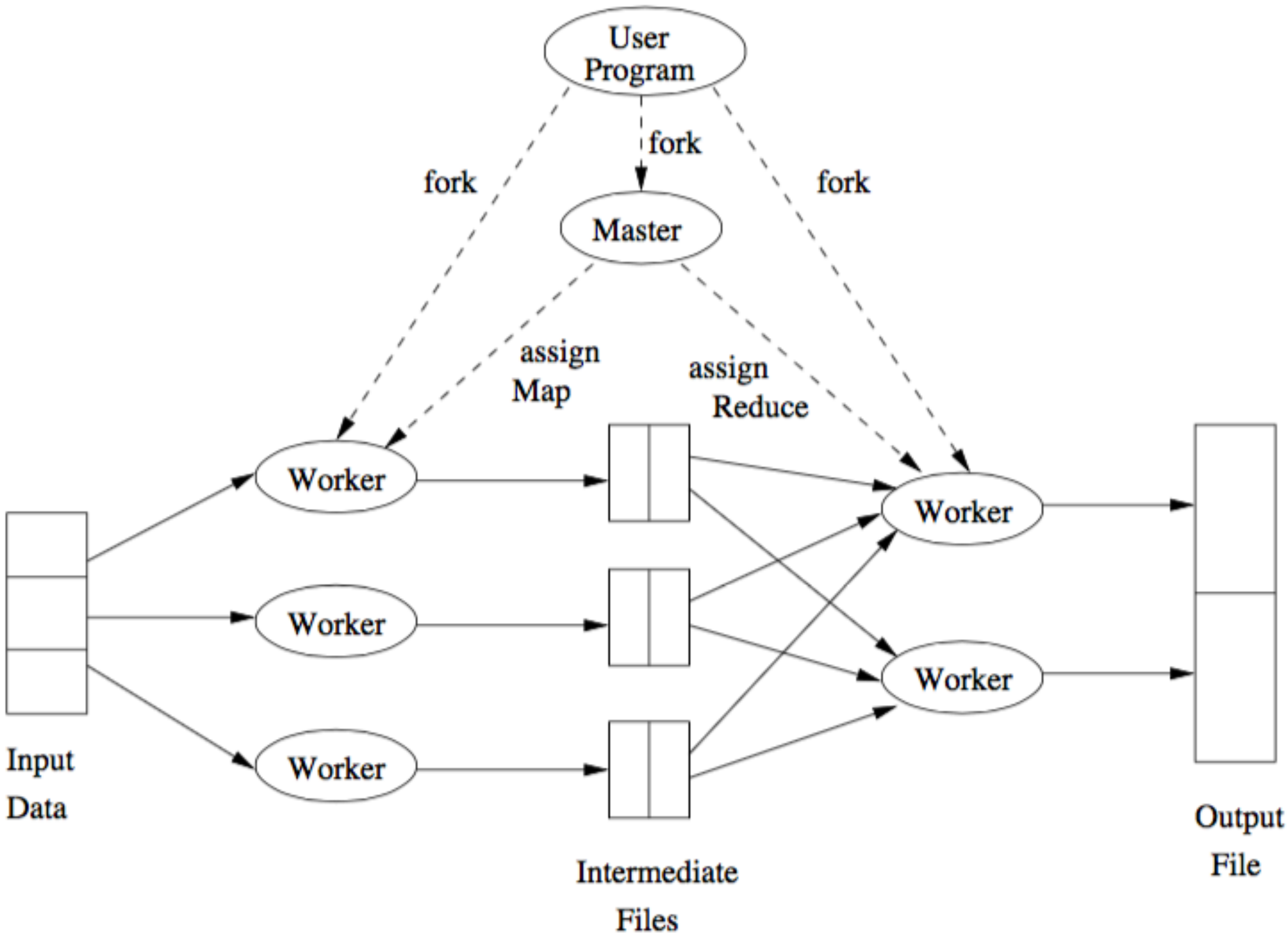


Combiner

- If a Reduce function is **associative and commutative**, we can push some of what the reducers do to the Map tasks.
  - Associative and commutative: the values to be combined can be combined in any order, with the same result.

- **Example:**
  - Instead of a Map task in Word-Count producing k pairs (w,1), (w, 1), …, we could apply the Reduce function within the Map task, to produce (w, k).
  - Note that we still need Reduce task to combine output of different Map tasks.

MapReduce Execution



MapReduce Execution

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers

- Master pings workers periodically to detect failures

Coping with Failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker

- Reduce worker failure
  - Only in-progress tasks are reset to idle

- Master failure
  - MapReduce task is aborted and client is notified

Algorithms using MapReduce

- MapReduce is not a solution to every problem
  - Example: online sale operators such as searching for products, recording sales in Amazon.com. The reason is that the processes involve relatively little calculation and that change the database often.

- MapReduce is more suitable for batch operations on write-one read-many distributed files.
  - Example: Amazon might use MapReduce to perform certain analytic queries on large amounts of data, such as finding for each user those users whose buying patterns were most similar.

Matrix-Vector Multiplication by MapReduce

- The matrix-vector product between an n x n matrix *M* and a vector **v** of length n is the vector **x** of length n, whose ith element x_i is given by.

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

- Application: This kind of calculation is used for Page Rank algorithm.
- If n is small, we do not want to use DFS or MapReduce

Matrix-Vector Multiplication by MapReduce

- If n is large but the vector **v still can fit into the memory**

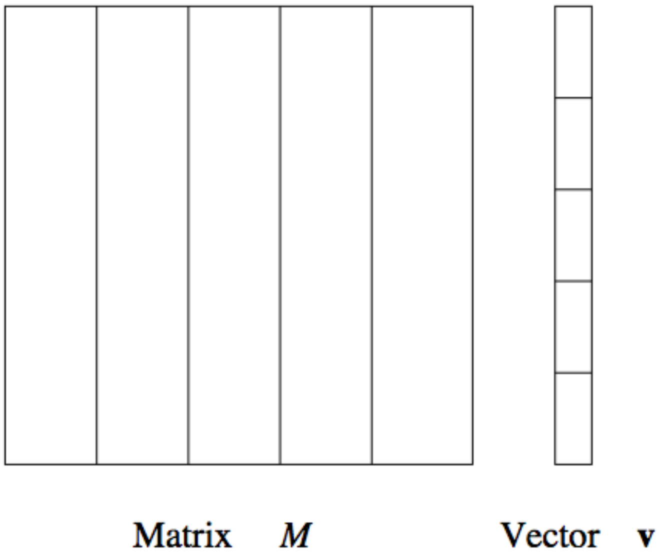
Map(key, value):
// key: chunk id of the matrix M; value: elements in the matrix chunk
read **v** into memory
for each element m_{ij} in value:
emit(i, $m_{ij}*v_j$)

Group by key: collect all the terms with the same key; all the pair ($i,m_{ij}*v_j$) with the same key i, that make up the element x_i, to a Reduce task.

Reduce(key, values):
// key: a row index i, values: an iterator over ($m_{ij}*v_j$) for j in [1,..n]
result = 0
for each element $m_{ij}*v_j$ in values
result += $m_{ij}*v_j$

Matrix-Vector Multiplication by MapReduce

- If n is large but the vector **v cannot fit into the memory**
  - Divide the matrix into vertical stripes of equal width, and divide the vector into an equal number of horizontal stripes.
  - Each stripe is written to one file in DFS
  - Each **Map** task is assigned to **a chunk from one the stripes** of the matrix, and gets **the entire corresponding stripe** of the vector.



Relational-Algebra Operations

- R(A1, A2, …, An) is a schema of a relation named R with its attributes A1, A2, …, An.
- In a relational database, queries are given in the query language SQL.
- A relation can be stored as a large file in a DFS system, each element is a tuple of the relation. We would like to answer similar queries using MapReduce.
- Relational Algebra
  - Selection
  - Projection
  - Union, Intersection and Difference
  - Natural Join
  - Grouping and Aggregation

Relational-Algebra Operation: Selection

- Selection
  - Apply a condition C to each tuple in the relation and produce as output only those tuples that satisfy C.

- A MapReduce implementation of selection
  - **Map Function:** for each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t,t). That is, both the key and value are t.
  - **Reduce Function:** the **Reduce** function is the identity. It simply parses each key-value pair to the output.

Relational-Algebra Operation: Natural Join

- **Natural Join:**
  - Given two relations R, S, to conduct $R \bowtie S$, we compare each pair of tuples, one from each relation. If ***the tuples agrees on all the attributes that are common to the two schemes, the produce the tuples.***
  - Example:
    - Given a *Links(From, To)* relation, find the paths of length 2 in the Web, i.e. finds (u,v, w) so that there is a link from u to v, and a link from v to w.
    - Take the natural link of *Links* with itself: Imagine there are two copies of the relation *Links,* which are L1(U1, U2) and L2(U2, U3); we compute $L1 \bowtie L2$

From	*To*
url1	url2
url1	url3
url2	url3
url2	url4
...	...

Relational-Algebra Operation: Natural Join

- Natural Join via MapReduce
  - Consider the special case of joining R(A, B) with S(B,C)
  - We use the B-value of tuples from either relation as the key

- **Map Function:** for each tuple (a,b) of R, produce the key-value pair (b, (R, a)). For each tuple (b,c) of S, produce the key-value pair (b, (S, c)); where R, S are relation names.

- **Reduce Function (key, values)**
  - Key: a value b of attribute B.
  - Values: a list of pairs that are either of the form (R, a) or (S,c).
  - Construct all pairs coming from two different relations, i.e. (R,a) and (S,c) to form a triple (a,b,c)

- The algorithm still works if A, B, C are sets of attributes.

Relational-Algebra Operation: Grouping and Aggregation

- **Grouping and Aggregation:**
  - Given a relation R, partition its tuples according to their values in a set of attributes G, called the **grouping attributes.**
  - For each group, aggregate the values in certain other attributes using operators such as SUM, COUNT, AVG, MIN, and MAX.

- **Example**
  - Imagine a social-networking site with a relation
    - Friends(User, Friend)
  - Question: collect statistics about the number of friends members have.
  - This question can be answered by grouping and aggregation

$$\gamma_{\text{User},\text{COUNT}(\text{Friend})}(\text{Friends})$$

Relational-Algebra Operation: Grouping and Aggregation

- Grouping and Aggregation by MapReduce
  - Let R(A, B, C) is the relation to which we want to apply the operator $\gamma_{A,\theta(B)}(R)$

- Map will produce the grouping, while Reduce does the aggregation
- The Map Function
  - For each tuple (a,b,c), produce the key-value pair (a,b)
- The Reduce Function
  - Each key **a** represents a group, apply the operator θ to the list [b1, b2, …, bn] of B-values associated with key **a.**
  - The output is the pair (a,x) where x is the result of applying the aggregation to the list.

Finding Frequent Itemsets with MapReduce

- Recall:
  - Frequent Itemsets: the set of items with **support** larger than **MinSup**
  - Algorithms for fining frequent itemsets: Apriori, FP-tree

- Mining Frequent Itemsets from Large-scale data
  - SON algorithm with MapReduce

Finding Frequent Itemsets with MapReduce

- A Simple, randomized algorithm
  - Procedure:
    - Pick a random sample of transactions (baskets) from the data set
    - Adjust the MinSup to be sample_size/data_size*MinSup
    - Apply some algorithm for finding frequent itemsets on the sample.
  - Errors:
    - False Negative: a frequent itemset that is frequent in the whole but not in the sample.
    - False Positive: a frequent itemset that is frequent in the sample but not in the whole
    - We can eliminate frequent positive to make another pass on the whole dataset.

Finding Frequent Itemsets with MapReduce

- **The Algorithm of Savasere, Omiecinski, and Navathe (SON)**
  - Divide the data set into equal chunks
  - Treat each chunk as a sample
  - Run some algorithm to find frequent itemsets (e.g. Apriori)
  - We use p*MinSup as the minimum support for each chunk, where p is the fraction of a chunk compared to the whole data set.
  - Take the union of all the frequent items from all the chunks → candidates
    - Since an itemset if frequent in the whole data set must be frequent in at least one of the chunk, we don't have False Negative
  - Take another pass through the data set, to remove any false positive.

Finding Frequent Itemsets with MapReduce

- SON Algorithm with MapReduce
  - First Map Function: Take the assigned subset of data set, and find the frequent itemsets in the subset using the simple algorithm we just described.
    - Note that we need to lower the min support as described in SON algorithm.
    - The output is a set of key-value pairs (F,1) where F is the frequent itemset from the sample. The value is set to 1 since it is irrelevant.
  - First Reduce Function:
    - Each Reduce task is assigned a set of keys, which are itemsets. The value is ignored, and the Reduce task simply produces the itemsets that appear one or more times.
    - The output is the candidate itemsets.

Finding Frequent Itemsets with MapReduce

- SON Algorithm with MapReduce
  - Second Map Function:
    - Take all the output from the first Reduce Function.
    - Count the number of occurrences of each of the candidate itemsets in the chunk of the data set that is assigned to this Map worker.
    - The output is a set of key-value pairs (C,v), where C is the itemset, and v is the support count for that item set in the assigned chunk.
  - Second Reduce Function:
    - Take the itemsets which are given and sum the associated values
    - The result is the total support count for each itemset.
    - Output itemsets with support larger than **MinSup**

Outline

- Large-Scale Computing
- Distributed File Systems
- MapReduce & Algorithms Using MapReduce
  - Matrix-Vector Multiplication
  - Relational-Algebra Operations
  - Finding Frequent Itemsets with Map-Reduce
- The Communication-Cost Model

Communication-Cost Model

- **Communication-Cost for Task Networks**
  - Communication cost for a task is the size of the input to the task.
    - Measured in bytes or the number of data objects (tuples) in the input
  - Communication cost of an algorithm is the sum of the communication cost of all the tasks implementing that algorithm.
    - Communication cost often dominates execution cost.

  - **Example**: natural join between R(A,B) and S(B,C) where the sizes of A and B relations are **r,** and **s** respectively.
    - The communication cost for all the Map tasks is **r + s**
    - The communication cost for all the Reduce tasks depends on the total output size of the Map tasks, which is approximately the input size for all the Map tasks, i.e. **r+s**
    - The communication cost is O(r+s)

Communication-Cost Model

- Wall-Clock Time
  - The time it takes a parallel algorithm to finish.

- Total communication cost is minimized by assigning all the work to one task (no parallelization). However, the wall-clock time of such algorithm will be high.

- A good algorithm should balance between the communication cost and the wall-clock time.