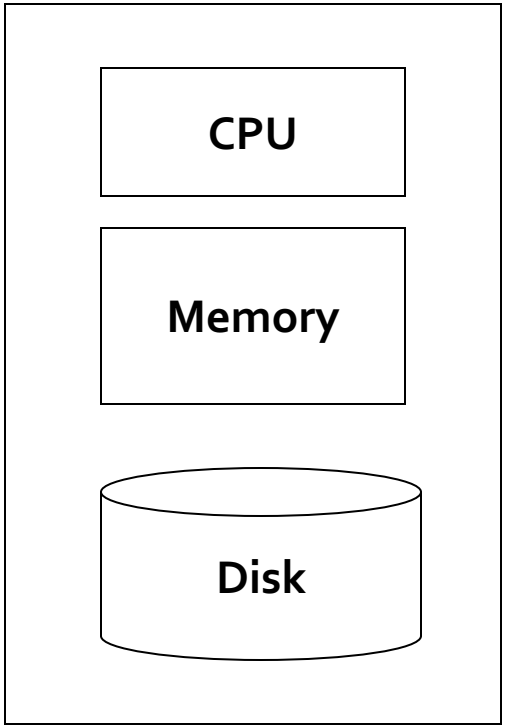
Map-Reduce and the New Software Stack

Mining of Massive Datasets
Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University
http://www.mmds.org

MapReduce

- Much of the course will be devoted to **large scale computing** for **data mining**
- **Challenges:**
  - How to distribute computation?
  - Distributed/parallel programming is hard
- **Map-reduce** addresses all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with big data

Single Node Architecture



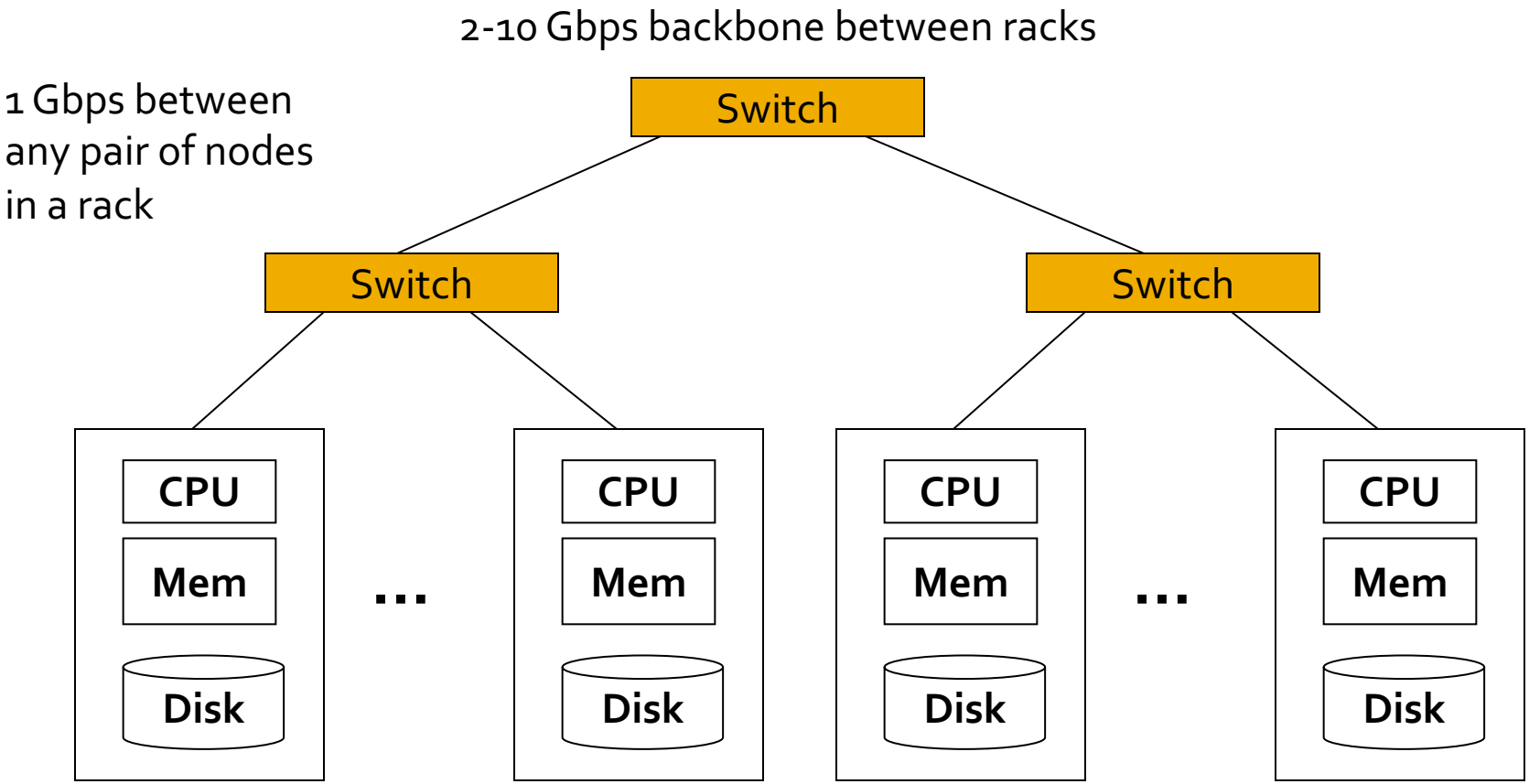
Machine Learning, Statistics

"Classical" Data Mining

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to **do** something useful with the data!
- **Today, a standard architecture for such problems is emerging:**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

Cluster Architecture



Each rack contains 16-64 nodes

In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shh0RO

Large-scale Computing

- **Large-scale computing** for **data mining** problems on **commodity hardware**
- **Challenges:**
  - **How do you distribute computation?**
  - **How can we make it easy to write distributed programs?**
  - **Machines fail:**
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to loose 1/day
    - People estimated Google had ~1M machines in 2011
      - 1,000 machines fail every day!

Idea and Solution

- **Issue: Copying data over a network takes time**
- **Idea:**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Map-reduce** addresses these problems
  - Google's computational/data manipulation model
  - Elegant way to work with big data
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - Map-Reduce

Storage Infrastructure

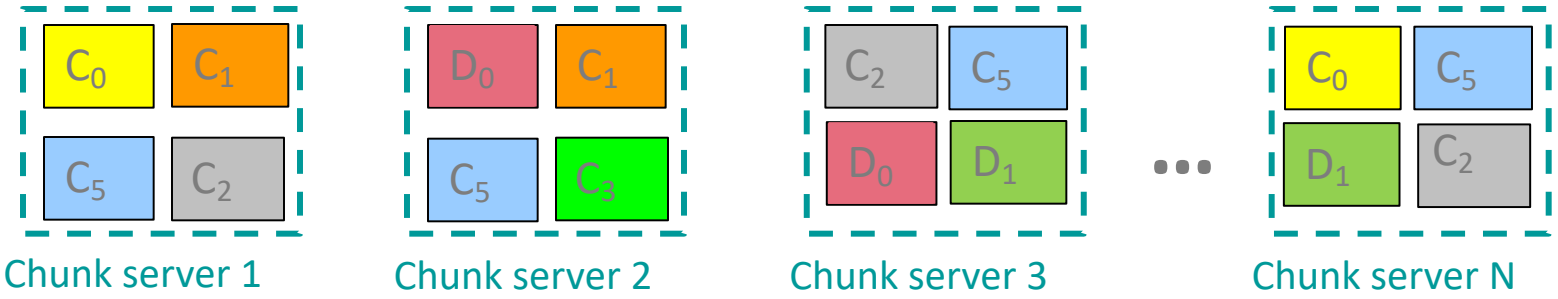
- **Problem:**
  - If nodes fail, how to store data persistently?
- **Answer:**
  - **Distributed File System:**
    - Provides global file namespace
    - Google GFS; Hadoop HDFS;
- **Typical usage pattern**
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed File System

- **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 racks
- **Master node**
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - 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 System

- **Reliable distributed file system**
- Data kept in "chunks" spread across machines
- Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

Programming Model: MapReduce

Warm-up task:

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Sample application:**
  - Analyze web server logs to find popular URLs

Task: Word Count

Case 1:

- File too large for memory, but all <word, count> pairs fit in memory

Case 2:

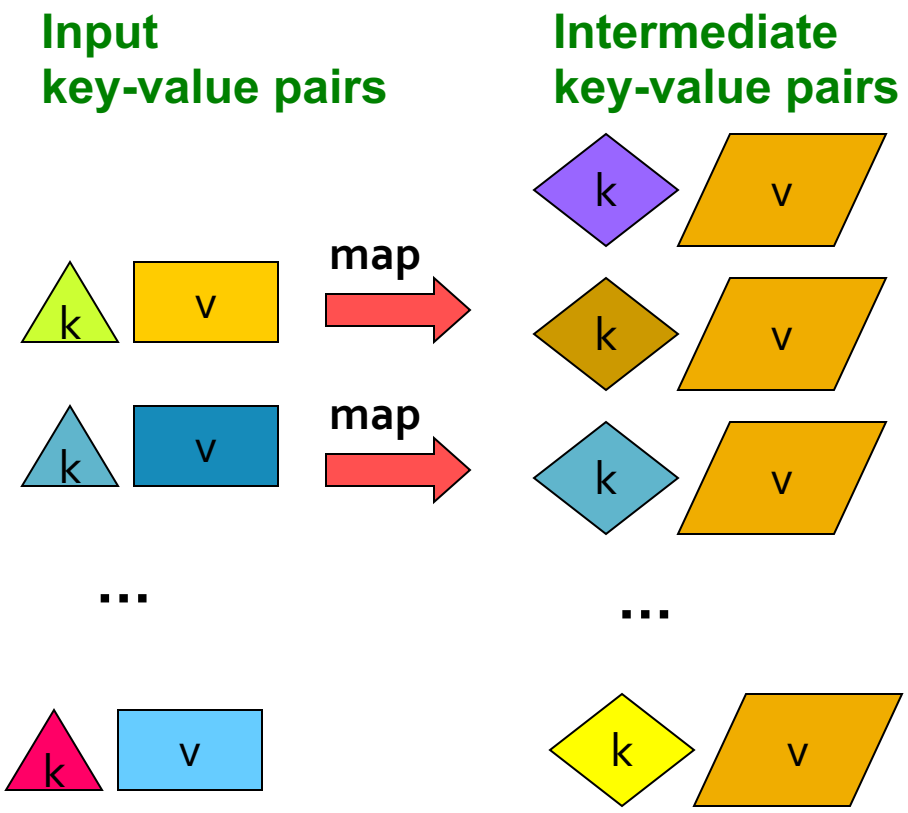
- Count occurrences of words:
  - **`$ words doc.txt | sort | uniq -c`**
    - where **`words`** takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of **MapReduce**
  - Great thing is that it is naturally parallelizable

MapReduce: Overview

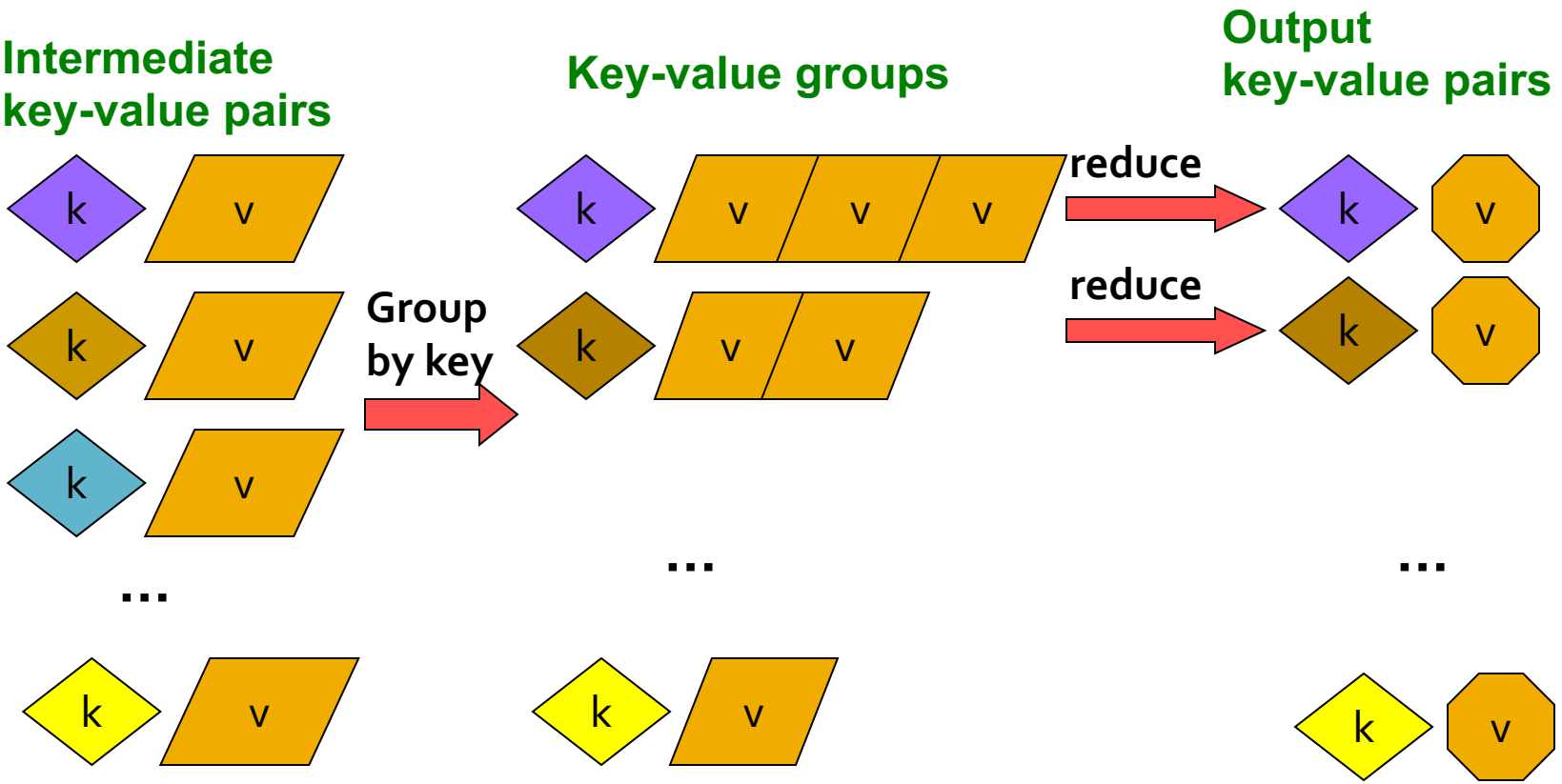
- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce: The Map Step



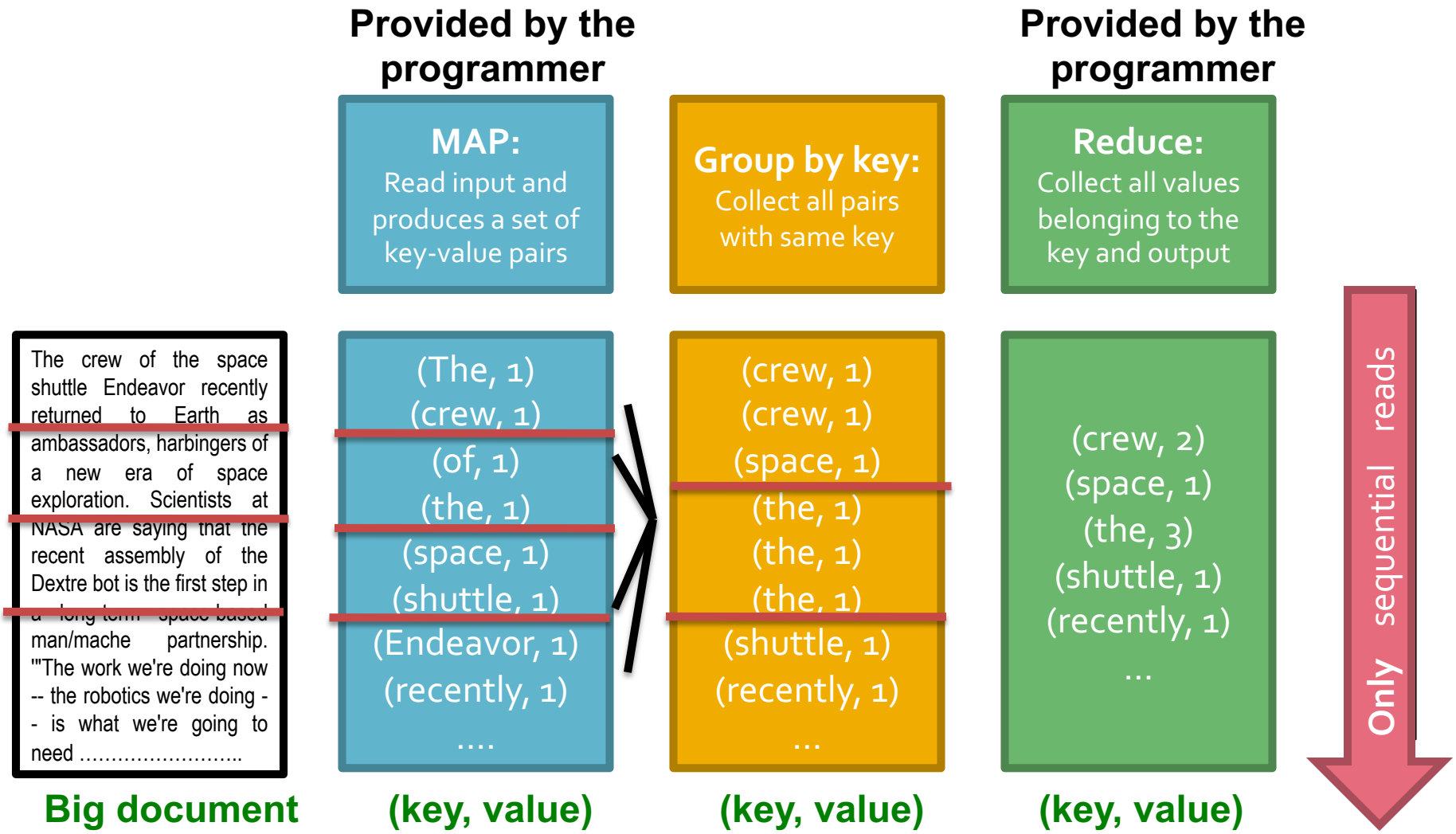
MapReduce: The Reduce Step



More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** $\rightarrow$ <k', v'>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every *(k,v)* pair
  - **Reduce(k', <v'>*)** $\rightarrow$ <k', v''>*
    - **All values *v'* with same key *k'* are reduced together and processed in *v'* order**
    - There is one Reduce function call per unique key *k'*

MapReduce: Word Counting



Word Count Using MapReduce

```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)
```

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

Example: Host size

- **Suppose we have a large web corpus**
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- **For each host, find the total number of bytes**
  - That is, the sum of the page sizes for all URLs from that particular host

- **Other examples:**
  - Link analysis and graph processing
  - Machine Learning algorithms

Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

Map-Reduce: Environment

Map-Reduce environment takes care of:

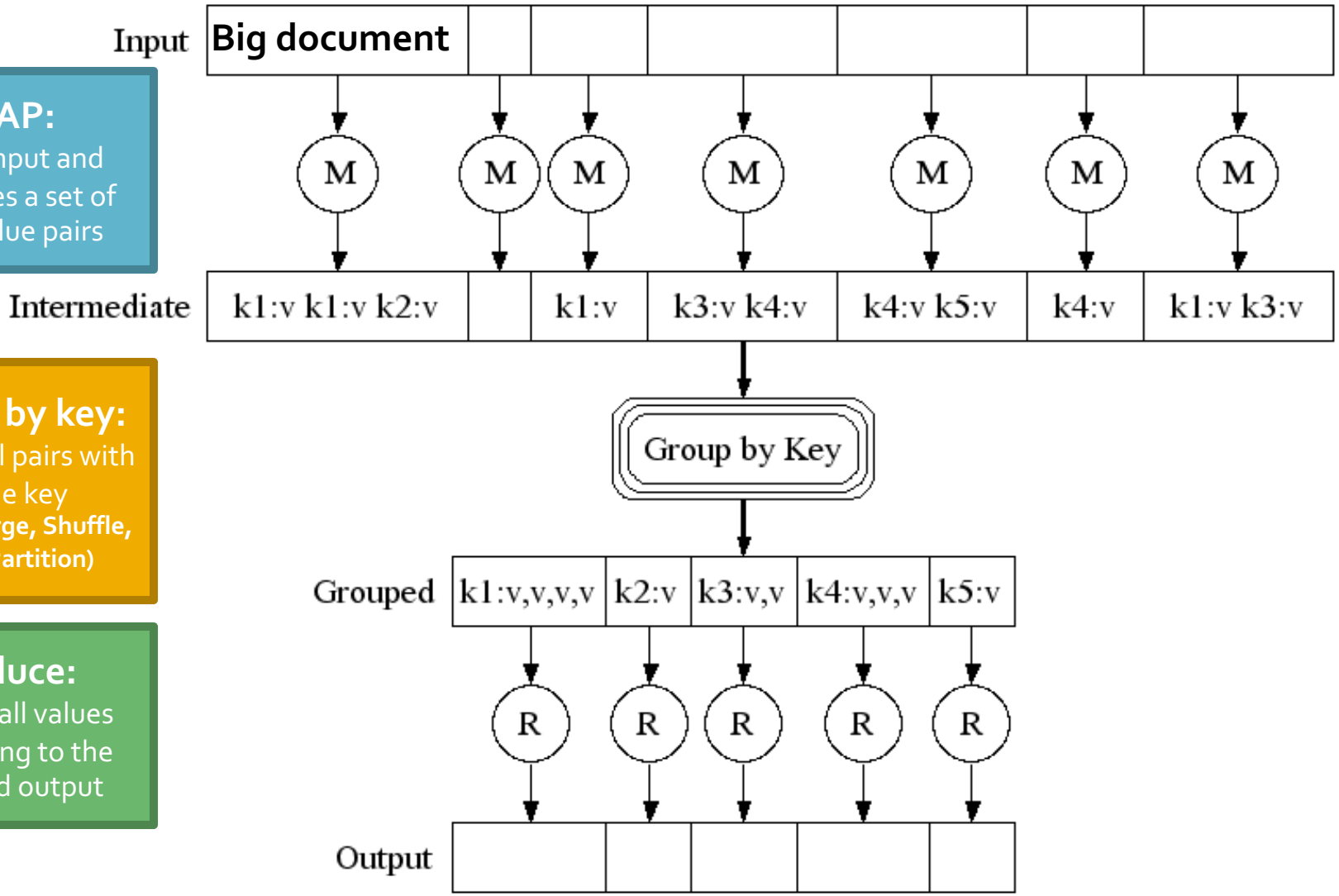
- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the **group by key** step
- Handling machine failures
- Managing required inter-machine communication

Map-Reduce: A diagram

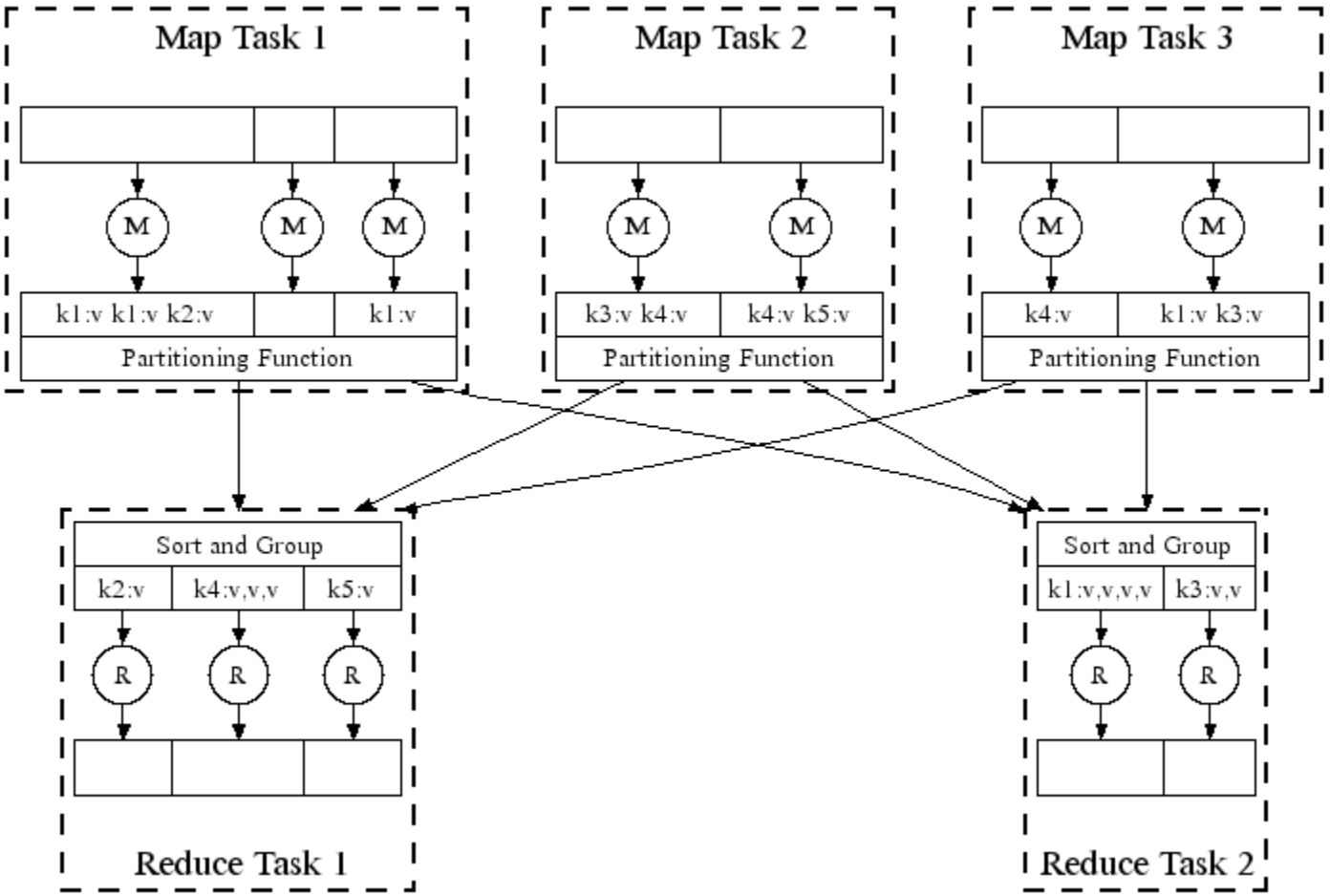
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



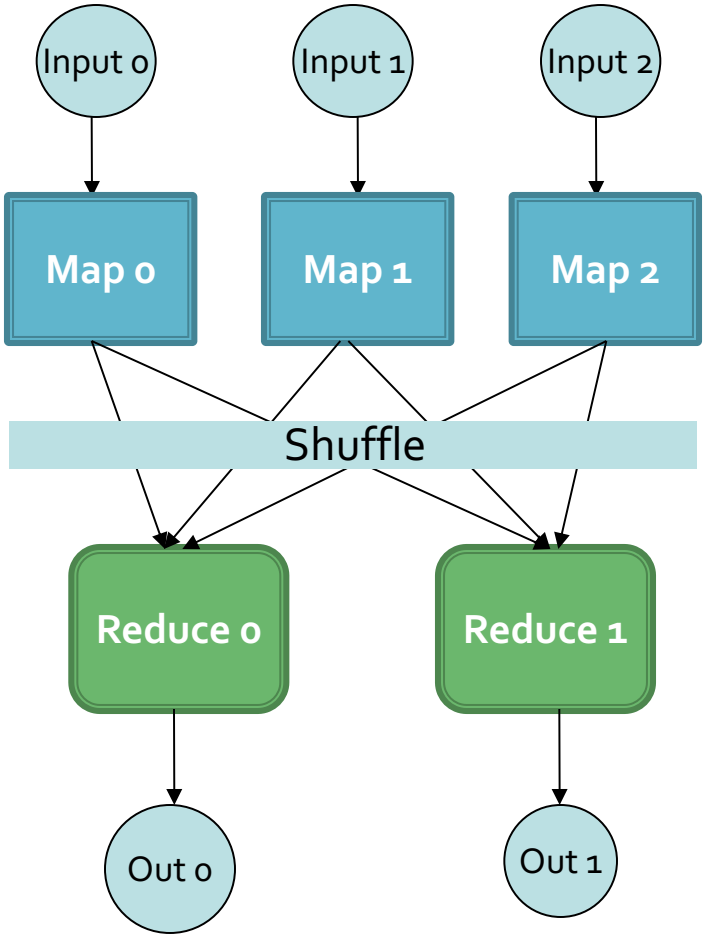
Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

Map-Reduce

- Programmer specifies:
  - Map and Reduce and input files
- **Workflow:**
  - Read inputs as a set of key-value-pairs
  - **Map** transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to output nodes
  - All k'v'-pairs with a given k' are sent to the same **reduce**
  - **Reduce** processes all k'v'-pairs grouped by key into new k''v''-pairs
  - Write the resulting pairs to files
- All phases are distributed with many tasks doing the work



Data Flow

- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks "close" to physical storage location of input data
- **Intermediate results are stored on local FS of Map and Reduce workers**
- **Output is often input to another MapReduce task**

Coordination: Master

- **Master node takes care of coordination:**
  - **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

Dealing with Failures

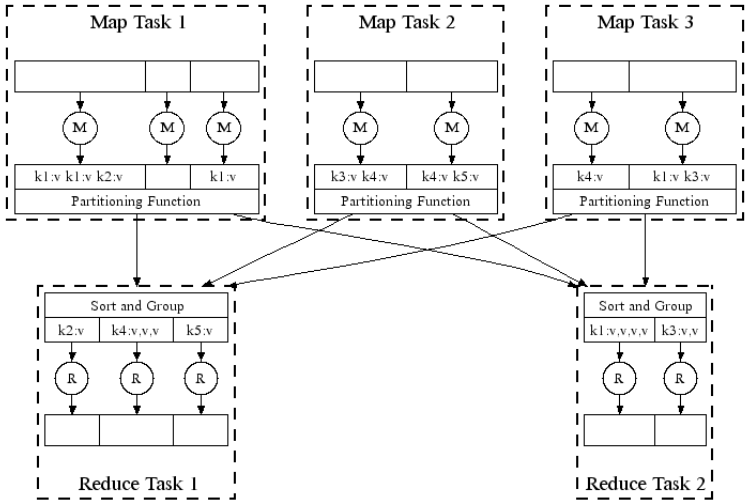
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- *M* map tasks, *R* reduce tasks
- **Rule of a thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

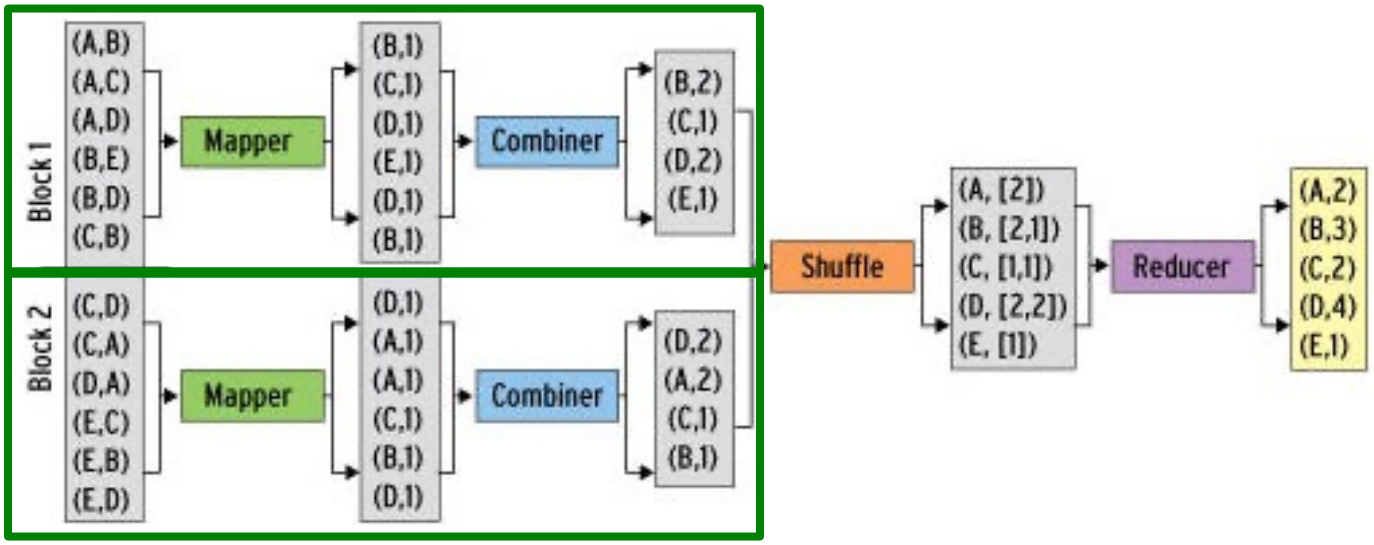
Refinement: Combiners

- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key k
  - E.g., popular words in the word count example
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function



Refinement: Combiners

- **Back to our word counting example:**
  - Combiner combines the values of all keys of a single mapper (single machine):



  - Much less data needs to be copied and shuffled!

Refinement: Combiners

- Combiner trick works only if reduce function is **commutative** and **associative**
- Sum
- Average
- Median

Refinement: Partition Function

- **Want to control how keys get partitioned**
  - Inputs to map tasks are created by contiguous splits of input file
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- **System uses a default partition function:**
  - **hash(key) mod *R***
- **Sometimes useful to override the hash function:**
  - E.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

Problems Suited for Map-Reduce

Algorithms Using MapReduce

- MapReduce is not a solution to every problem,
  - Not even every problem that profitably can use many compute nodes operating in parallel.
- DFS makes sense only when files are very large and are rarely updated in place.
  - For example it's not useful for managing on-line retail sales
- We might use MapReduce to perform certain analytic queries on large amounts of data
  - Google PageRank
  - Relational-Algebra Operations

Matrix-Vector Multiplication by MapReduce

- Suppose we have an n × n matrix M
- The matrix-vector product of M with v is:

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

- If n = 100, we do not want to use a DFS or MapReduce for this calculation.

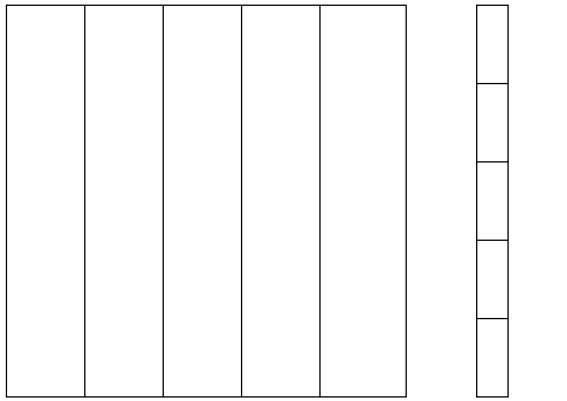
- But in ranking of Web pages, n is on the order of trillions.

Matrix-Vector Multiplication by MapReduce

- Case 1: vector **v** fit in main memory
  - Thus be available to every Map task.
  - The Map Function:
    - The Map function is written to apply to one element of M.
    - Each Map task will operate on a chunk of the matrix M. From each $\mathbf{m_{ij}}$ it produces the key-value pair $\mathbf{(i, m_{ij}, v_j)}$. Thus, all terms of the sum that make up the component $\mathbf{x_i}$ of the matrix-vector product will get the same key, **i**.
  - The Reduce Function:
    - The Reduce function simply sums all the values associated with a given key **i**. The result will be a pair $\mathbf{(i,x_i)}$.

Matrix-Vector Multiplication by MapReduce

- Case 2: the vector v cannot fit in main memory
  - We can divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height



  - The i^{th} stripe of the matrix multiplies only components from the i^{th} stripe of the vector
  - Each Map task is assigned a chunk from one of the stripes of the matrix and gets the entire corresponding stripe of the vector

Relational-Algebra Operations

Relational-Algebra Operations

- **Reminder**:
  - A relation is a table with column headers called attributes.
  - Rows of the relation are called tuples.
  - The set of attributes of a relation is called its schema.
  - We often write an expression like $R(A_1, A_2, \ldots, A_n)$ to say that the relation name is R and its attributes are $A_1, A_2, \ldots, A_n$.

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

  - **Example**: Relation Links consists of the set of pairs of URL's such that the first has one or more links to the second.

Relational-Algebra Operations

- The relational-algebra operations:
  - Selection: Apply a condition C to each tuple in the relation and produce as output only those tuples that satisfy C.
    - The result of this selection is denoted $\sigma_C(R)$.
  - Projection: For some subset S of the attributes of the relation, produce from each tuple only the components for the attributes in S.
    - The result of this projection is denoted $\pi_S(R)$.
  - Union, Intersection, and Difference: These well-known set operations apply to the sets of tuples in two relations that have the same schema.

Relational-Algebra Operations

- Natural Join: Given two relations, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute.
  - The natural join of relations R and S is denoted R⋈S.
- Grouping and Aggregation: Given a relation R, partition its tuples according to their values in one set of attributes G, called the grouping attributes. Then, for each group, aggregate the values in certain other attributes.
  - We denote a grouping-and-aggregation operation on a relation R by $\gamma_X(R)$.

Examples of Relational-Algebra Operations

- Example 1:
  - Find the triples of URL's (u,v,w) such that there is a link from u to v and a link from v to w.
  - Thus, imagine that there are two copies of Links, namely L1(U1,U2) and L2(U2,U3). Now, if we compute L1⋈L2, we shall have exactly what we want.
- Example 2:
  - A social-networking site has a relation

    Friends(User, Friend)
  - What is the number of friends each member have?
  - Compute a count of the number of friends of each user. Then, for each group the count of the number of friends of that user is made.

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

Selections by MapReduce

- Selections really do not need the full power of MapReduce.
  - Can be done in Map portion or the Reduce portion alone.
- **The Map Function:** For each tuple t in R, test if it satisfies C. If so, produce the key-value pair *(t,t)*.
- **The Reduce Function:** The Reduce function is the identity. It simply passes each key-value pair to the output.

Projections by MapReduce

- Projection is performed similarly to selection.
  - But Reduce function must eliminate duplicates.

- **The Map Function:**
  - For each tuple t in R, construct a tuple t′ by eliminating from t those components whose attributes are not in S.
  - Output the key-value pair (t′, t′).

- **The Reduce Function:**
  - For each key t′ produced by any of the Map tasks, there will be one or more key-value pairs (t′,t′).
  - Reduce function turns (t′, [t′, t′, . . . , t′]) into (t′, t′).

Union by MapReduce

- **The Map Function:**
  - Turn each input tuple t into a key-value pair (t, t).
- **The Reduce Function:**
  - Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

Intersection by MapReduce

- **The Map Function:**
  - Turn each input tuple t into a key-value pair (t, t).
- **The Reduce Function:**
  - If key t has value list [t, t], then produce (t, t). Otherwise, produce nothing.

Join by MapReduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Join by MapReduce

- We shall use the B-value of tuples from either relation as the key.
- **The 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)).
- **The Reduce Function:**
  - Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c).
  - Construct all pairs consisting of one with first component R and the other with first component S, say (R, a) and (S, c).
  - Each value is one of the triples (a, b, c) such that (R, a) and (S, c) are on the input list of values for key b.
- The same algorithm works if the relations have more than two attributes.

Grouping and Aggregation by MapReduce

- Let R(A,B,C) be a relation to which we apply the operator $\gamma_{A,\theta(B)}(R)$.
- Map will perform 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 aggregation operator θ to the list $[b_1, b_2, \ldots, b_n]$ of B-values associated with key a.
  - The output is the pair (a,x), where x is the result of applying θ to the list.
  - For example, if θ is SUM, then $x = b_1+b_2+\ldots+b_n$.

Matrix Multiplication

- P = MN

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

- We can think of a matrix as a relation with three attributes:
  - the row number,
  - the column number,
  - and the value in that row and column
  - Relation M(I,J,V), with tuples (i,j,m_{ij})
  - Relation N(J,K,W), with tuples (j,k,n_{jk})
- The product MN is almost a join followed by grouping and aggregation

Matrix Multiplication

- First MapReduce:
  - The Map Function:
    - For each matrix element m_{ij}, produce the key value pair $(j, (M, i, m_{ij}))$.
    - Likewise, for each matrix element n_{jk}, produce the key value pair $(j, (N, k, n_{jk}))$.
  - The Reduce Function:
    - For each key j, examine its list of associated values.
    - For each value that comes from M, say (M, i, m_{ij}), and each value that comes from N, say (N, k, n_{jk}), produce a key-value pair with key equal to (i, k) and value equal to the product of these elements, $m_{ij} * n_{jk}$.

Matrix Multiplication

- Second MapReduce (grouping and aggregation):
  - The Map Function:
    - This function is just the identity. That is, for every input element with key (i, k) and value v, produce exactly this key-value pair.
  - The Reduce Function:
    - For each key (i, k), produce the sum of the list of values associated with this key. The result is a pair ((i, k), v), where v is the value of the element in row i and column k of the matrix P = MN.

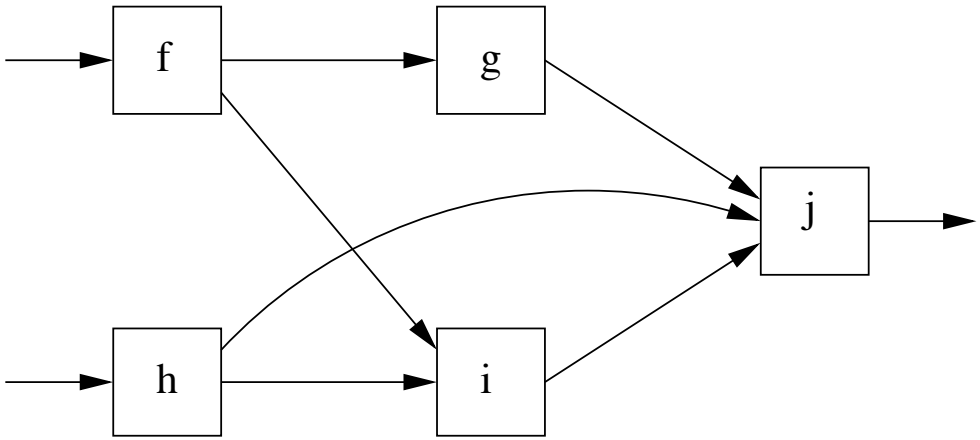
Matrix Multiplication

- With One MapReduce Step
  - By using the Map function to create the sets of matrix elements that are needed to compute each element of the answer P.
  - The Map Function:
    - For each element m_{ij} of M, produce all the key-value pairs ((i,k), (M,j,m_{ij})) for k = 1,2,...col_N.
    - For each element n_{jk} of N, produce all the key-value pairs ((i,k), (N,j,n_{jk})) for i = 1,2,... $rows_M$
  - The Reduce Function:
    - Each key (i,k) will have an associated list with all the values (M,j,m_{ij}) and (N,j,n_{jk}), for all possible values of j.
    - sort by j the values that begin with M and sort by j the values that begin with N, in separate lists.
    - The j^{th} values on each list must have their third components, m_{ij} and n_{jk} extracted and multiplied.

Extensions to MapReduce

Workflow Systems

- Workflow systems extend MapReduce from the simple two-step workflow to any collection of functions,
  - with an acyclic graph representing workflow among the functions.
- The data passed from one function to the next is a file of elements of one type.



Workflow Systems

- In analogy to Map and Reduce functions:
  - each function of a workflow can be executed by many tasks,
  - each of which is assigned a portion of the input to the function.
  - A master controller is responsible for dividing the work among the tasks that implement a function
- The functions of a workflow has the blocking property, in that they only deliver output after they complete.
  - if a task fails, a master controller can therefore restart the failed task at another compute node.

Workflow Systems

- Some applications of workflow systems are effectively cascades of MapReduce jobs
  - Example: join of three relations
- One advantage:
  - We don't need to store the temporary file that is output of one MapReduce job in the distributed file system.

Apache Spark

Apache Spark

- Introduced in 2009 at the UC Berkley AMPLab
- Spark is, at its heart, a workflow system.
- Some advantages:
  - A more efficient way of coping with failures.
  - A more efficient way of grouping tasks among compute nodes and scheduling execution of functions.
  - Integration of programming language features such as looping and function libraries.

Apache Spark

- Central data abstraction of Spark is:
  - *Resilient Distributed Dataset (RDD)*.
- An RDD is a file of objects of one type.
- Example: files of key-value pairs.
- RDDs are distributed
  - Broken into chunks.
- RDDs are resilient
  - We can recover from the loss of any or all chunks of an RDD
- There is no restriction on the type of the elements in RDD.

Apache Spark

- A Spark program consists of a sequence of steps,
  - each of which typically applies some function to an RDD to produce another RDD
    - Such operations are called transformations.
  - We can produce a result that is passed back to a Spark program.
    - Theses operations are called actions.

Apache Spark

- Some commonly used operations:
  - Map
    - Takes a parameter that is a function, and it applies that function to every element of an RDD, producing another RDD
  - Flatmap
    - Analogous to Map of MapReduce, but without the requirement that all types be key-value
  - Filter
    - Takes a predicate that applies to the type of objects in the input RDD
    - The predicate returns true or false for each object

Apache Spark

- Some commonly used operations:
  - Reduce
    - It is an action that returns a value and not another RDD.
    - Reduce is applied repeatedly to each pair of consecutive elements, reducing them to a single element.
  - Join
    - takes two RDD's, each representing one of the relations.
    - The type of each RDD must be a key-value pair, and the key types of both relations must be the same.
  - GroupByKey
    - takes as input an RDD whose type is key-value pairs.
    - The output RDD is also a set of key-value pairs with the same key type.
    - The value type for the output is a list of values of the input type.

Example

- Suppose we want to count the words in a document but avoid counting stop words
- Steps:
  1. Load the document to a RDD (R0)
  2. Use Flatmap to produce a RDD of words (R1)
  3. Use Map to produce the **(w, 1)** tuples (R2)
  4. Use Filter to remove stop words (R3)
  5. Apply Reduce action on R3 to count the number of each words.

Spark Implementation

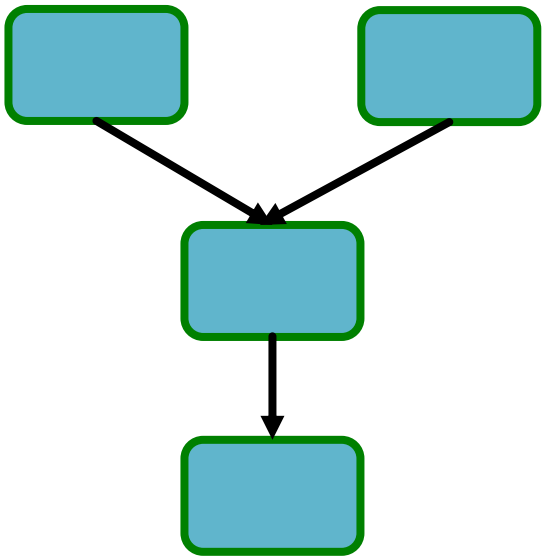
- Similarity to MapReduce:
  - Spark allows any RDD to be divided into chunks, which it calls splits.
    - Each split can be given to a different compute node,
    - and the transformation on that RDD can be performed in parallel on each of the splits.
- Two important improvements over MapReduce:
  - Lazy evaluation of RDD's
  - lineage for RDD's

Lazy Evaluation

- Spark will not start the execution of the process until an ACTION is called.
- Once Spark sees an ACTION being called, it starts looking at all the transformations and creates a DAG.
- DAG Simply sequence of operations that need to be performed in a process to get the resultant output.
- It may merge some transformation or skip some unnecessary transformation and prepare a perfect execution plan.

Lazy Evaluation

- Directed Acyclic Graph (DAG)



```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
    .count()
```

Resilience of RDD's

- Spark's substitute for redundant storage of intermediate values is to record the lineage of every RDD.
- The lineage tells the Spark system how to recreate the RDD, or a split of the RDD.
- Recovery from a node failure can be more complex in Spark than in MapReduce or in workflow systems that store intermediate values redundantly
  - But it results in greater speed when things go right.

Spark Components