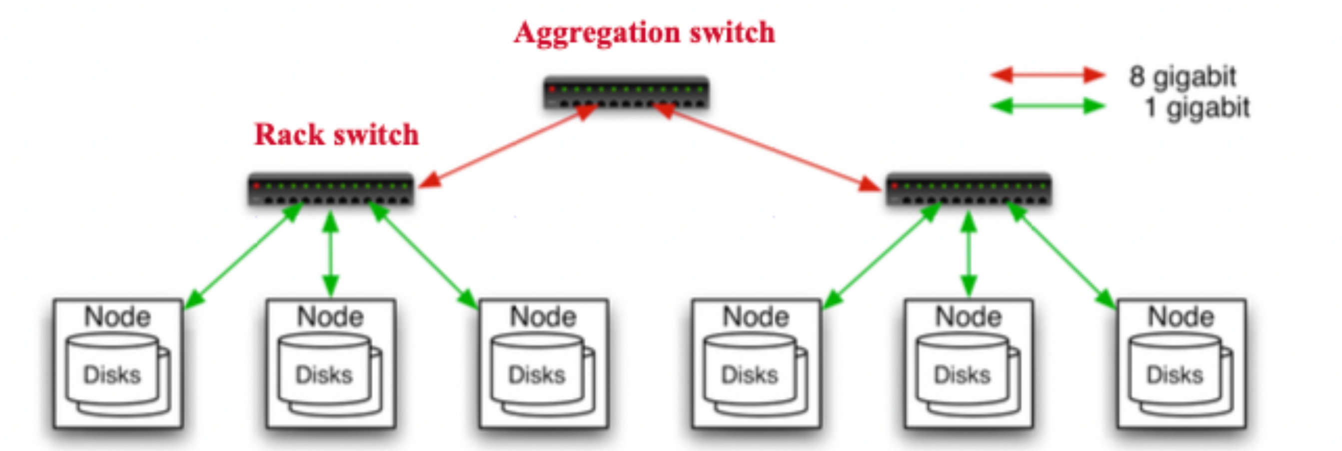
Parallel Programming with Hadoop/MapReduce

Overview

- Related technologies:
    - Hadoop/Google File System.
- MapReduce applications.

Typical Hadoop cluster



- 40 nodes/rack, 1000-4000 nodes in cluster.
- 1 Gbps bandwidth in rack, 8 Gbps out of rack.
- Node specs:
    - 8-16 cores, 32 GB RAM, 8x1.5 TB disks.

MapReduce Programming Model

- Inspired from map and reduce operations commonly used in functional programming languages like Lisp.
- Have multiple map tasks and reduce tasks.
- Users implement interface of two primary methods:
    - Map: (key1, val1) → (key2, val2).
    - Reduce: (key, [val2] → [val3])

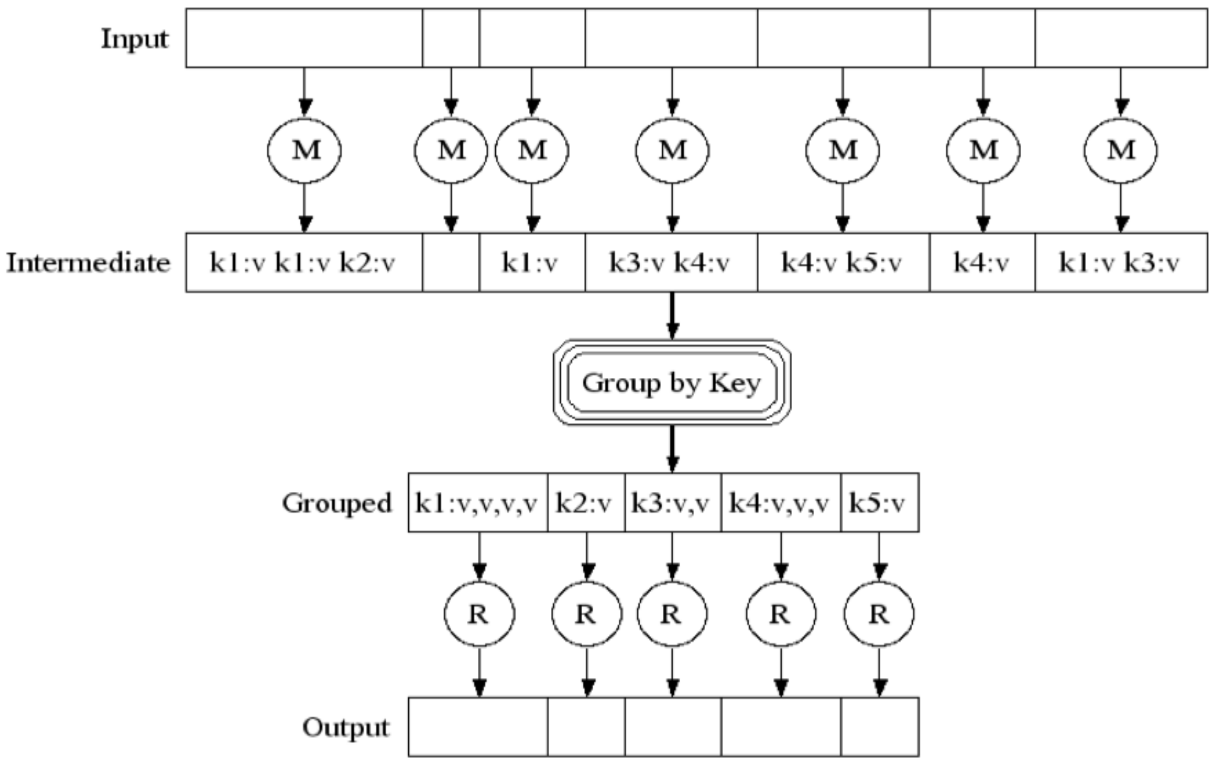
Example: Map Processing in Hadoop

- Given a file:
    - A file may be divided into multiple parts (splits).
- Each record (line) is processed by a Map function.
    - Written by the user.
    - Takes an input key/value pair.

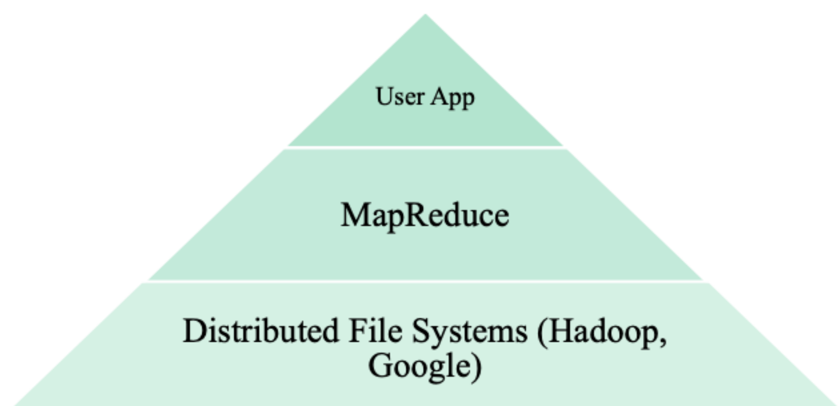
- Produces a set of intermediate key/value pairs.
- E.g., (doc-id, doc-content).

- Draw an analogy to SQL group-by clause.

Put Map and reduce Tasks Together



System Support for MapReduce



Distributed Filesystems

- The interface is the same as single-machine file system:

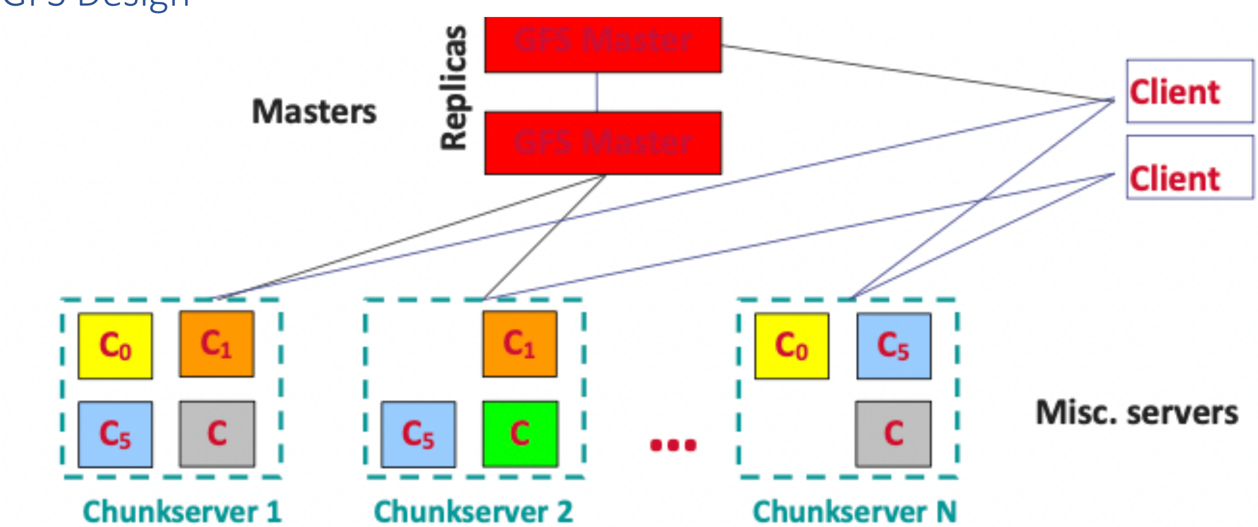
- create(), open(), read(), write(), close().

- Distributed file data to several machines (storage units).
  - Support replication.
- Support concurrent data access.
  - Fetch content from remote servers. Local caching.
- Different implementations sit in different places on complexity/feature scale.
  - Google file system and Hadoop HDFS:
    - Highly scalable for large data-intensive applications.
    - Provides redundant storage of massive amounts of data on cheap and unreliable computers.

Assumptions of GFS/Hadoop DFS

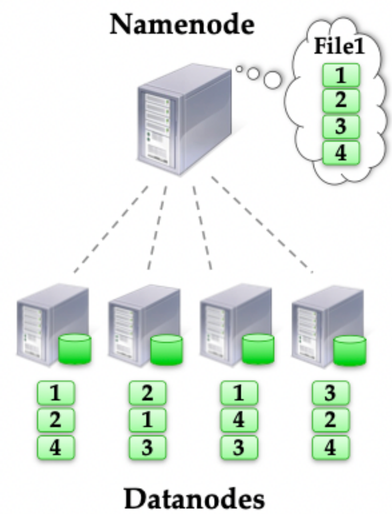
- High component failure rates:
  - Inexpensive commodity components fail all the time.
- "Modest" number of HUGE files:
  - Just a few million.
  - Each is 100 MB or larger: multi-GB files typical.
- Files are write-once, mostly appended to:
  - Perhaps concurrently.
- Large streaming reads.
- Highly sustained throughput favored over low latency.

GFS Design



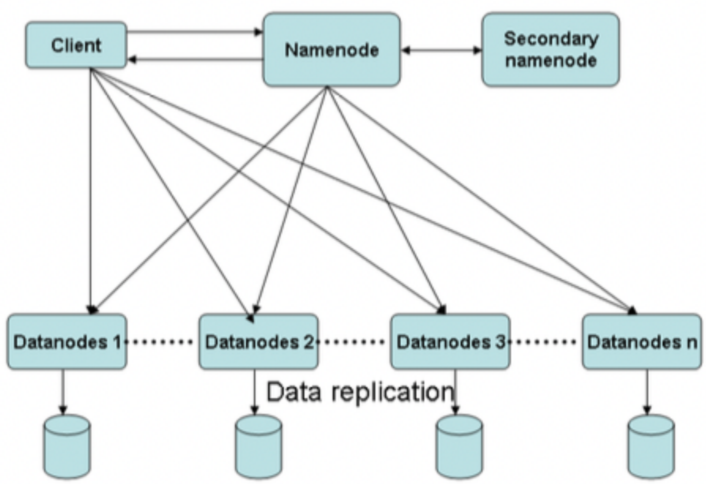
- Files are broken into chunks (typically 64 MB) and serve in chunk servers.
- Master manages metadata, but clients may cache meta data obtained.
  - Data transfers happened directly between clients/chunk-servers.
- Reliability through replication.
  - Each chunk replicated across 3+ chunk-servers.

Hadoop Distributed File System

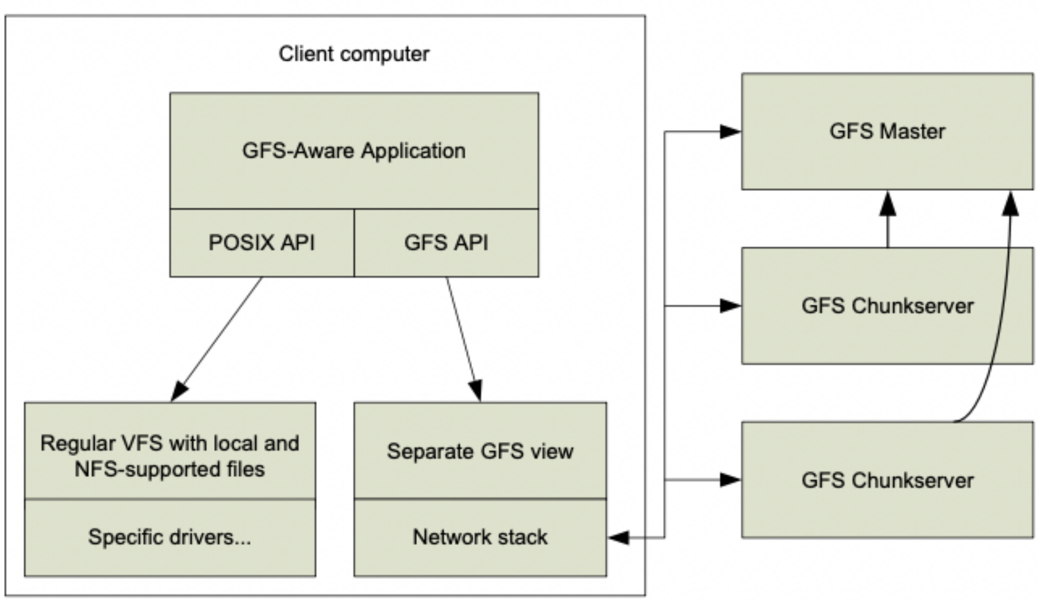


- Files split into 128 MB blocks.
- Blocks replicated across several data nodes (often 3).
- Name node stores metadata (file names, locations, etc.).
- Optimized for large files, sequential reads.
- Files are append-only.

HDFS

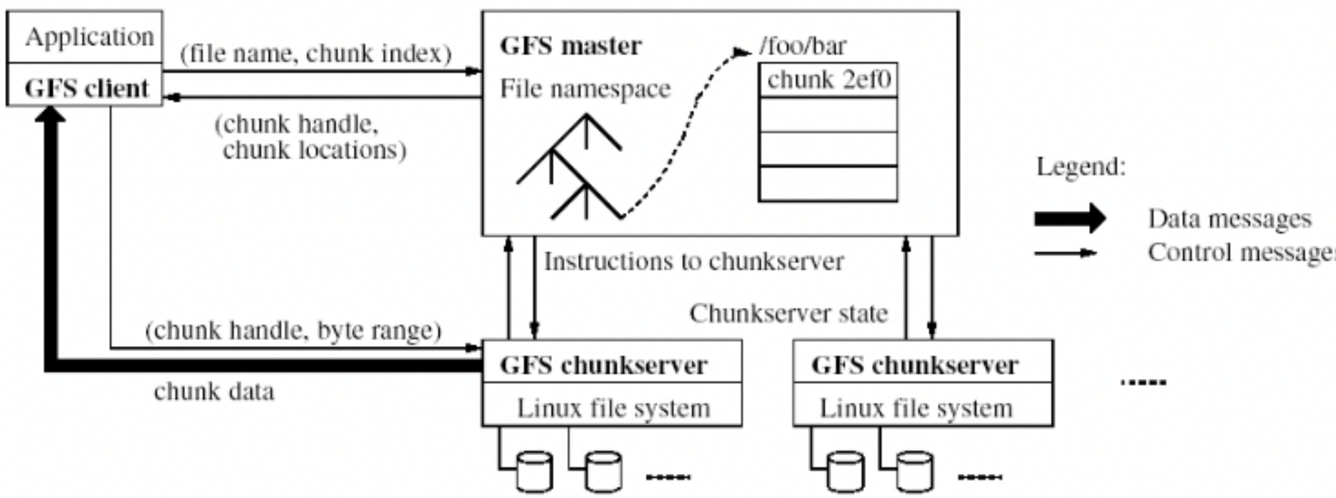


GFS Client Block Diagram

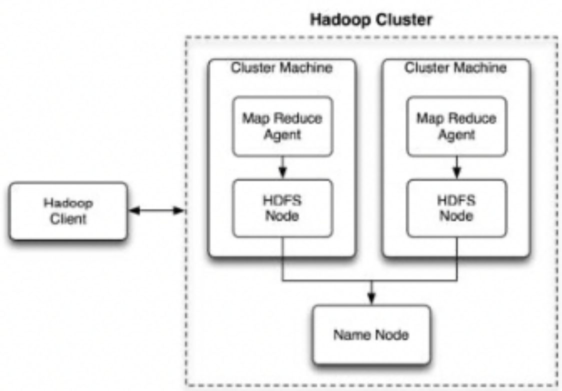


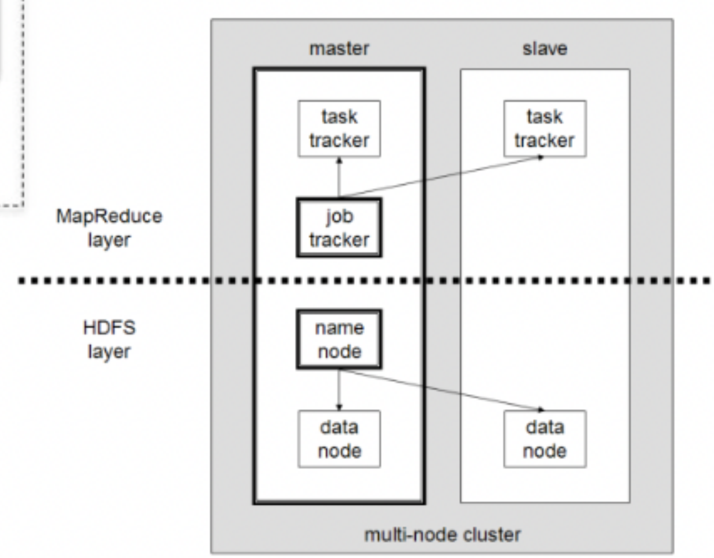
- Provide both POSIX standard file interface, and customed API.
- Can cache metadata for direct client-chunk server access.

Read/write Access Flow in GFS



Hadoop DFS with MapReduce



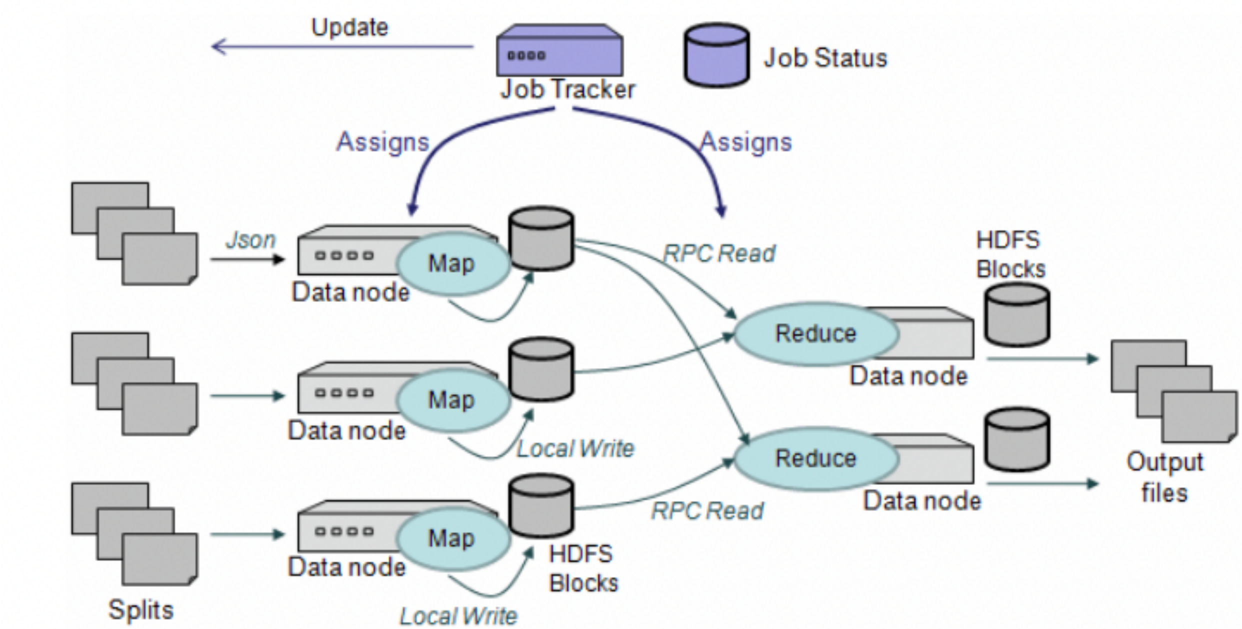


MapReduce: Execution overview

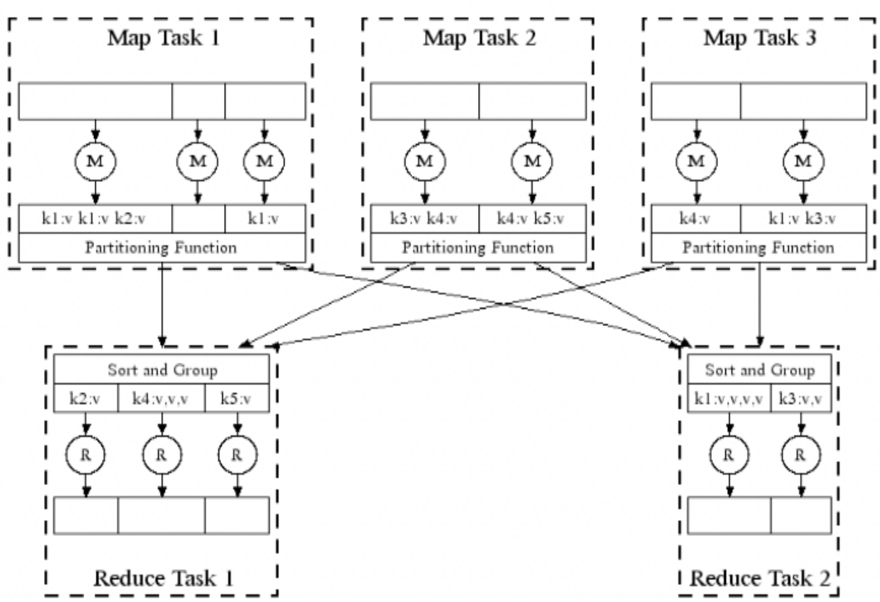
1. Master Server distributes M map tasks to machines and monitors their progress.
2. Map task reads the allocated data, saves the map results in local buffer.

3. Shuffle phase assigns reducers to these buffers, which are remotely read and processed by reducers.
4. Reducers output the result on stable storage.

Execute MapReduce on a cluster of machines with Hadoop DFS

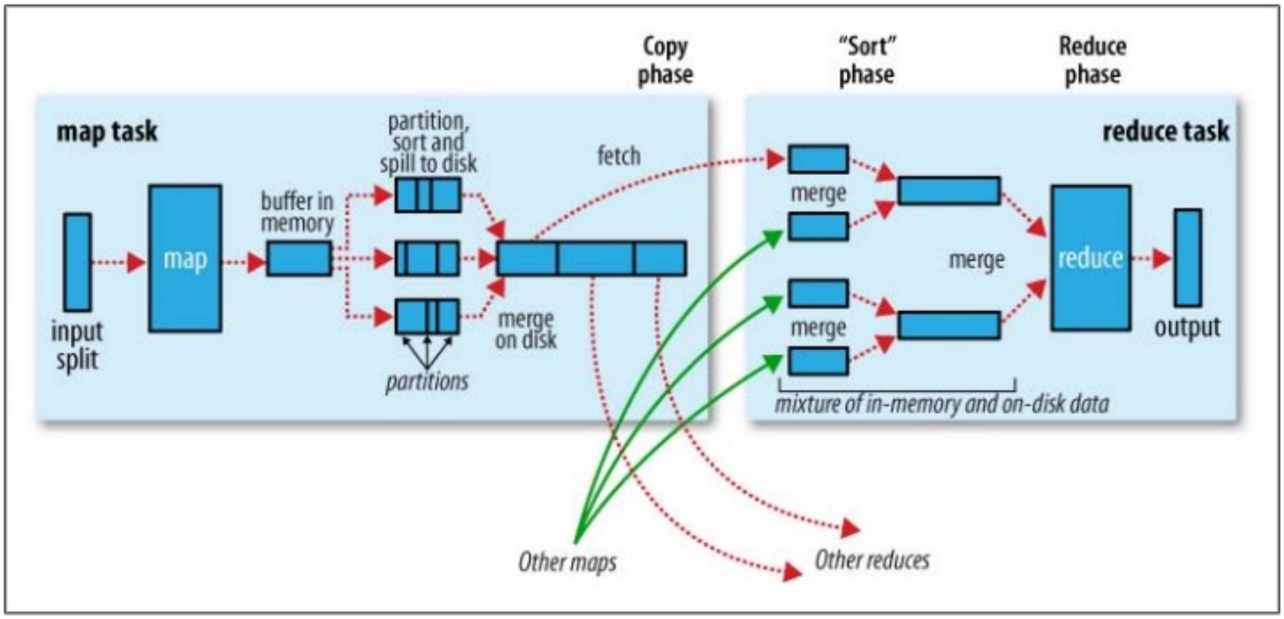


MapReduce in Parallel: Example

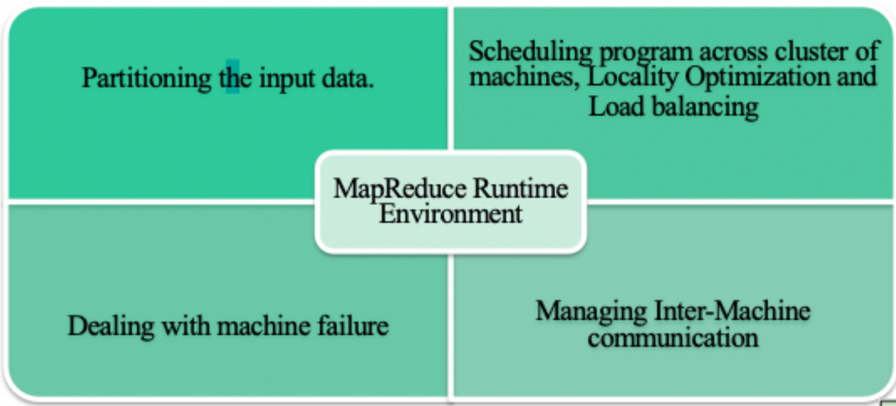


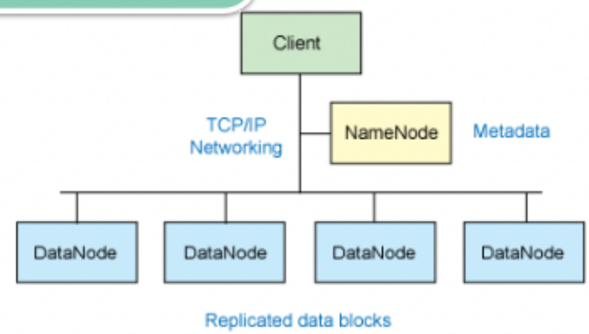
MapReduce: Execution Details

- Input reader:
    - Divide input into splits, assign each split to a Map task.
- Map task:
    - Apply the Map function to each record in the split.
    - Each Map function returns a list of (key, value) pairs.
- Shuffle/Partition and Sort:
    - Shuffle distributes sorting & aggregation to many reducers.
    - All records for key k are directed to the same reduce processor.
    - Sort groups the same keys together and prepares for aggregation.
- Reduce task:
    - Apply the Reduce function to each key.
    - The result of the Reduce function is a list of (key, value) pairs.

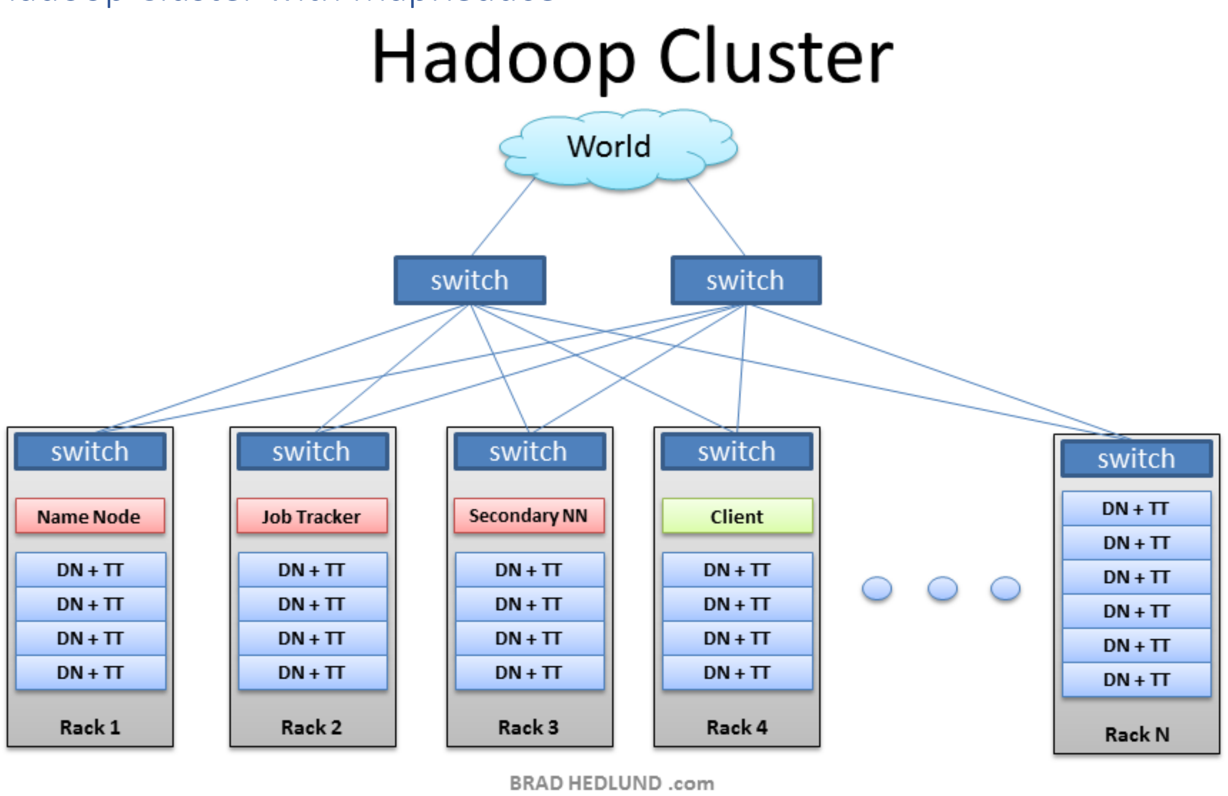


MapReduce: Runtime Environment & Hadoop





Hadoop Cluster with MapReduce



MapReduce: Fault Tolerance

- Handled via re-execution of tasks.
    - Task completion committed through master.
- Mappers save outputs to local disk before serving to reducers.
    - Allows recovery if a reducer crashes.
    - Allows running more reducers than # of nodes.
- If a task crashes:
    - Retry on another node.
      → OK for a map because it had no dependencies.
      → OK for reduce because map outputs are on disk.
    - If the same task repeatedly fails, fail the job, or ignore that input block.
    - For the fault tolerance to work, user tasks must be deterministic and side-effect-free.

If a node crashes:

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

MapReduce: Locality Optimization

- Leverage the distributed file system to schedule a map task on a machine that contains a replica of the corresponding input data.
- Thousands of machines read input at local disk speed.
- Without this, rack switches limit read rate.

MapReduce: Redundant Execution

- Slow works are source of bottleneck, may delay completion time.
- Near end of phase, spawn backup tasks, one to finish first wins.
- Effectively utilizes computing power, reducing job completion time by a factor.

MapReduce: Skipping Bad Records

- Map/Reduce functions sometimes fail for some inputs.
- Fixing the Bug might not be possible: Third Party Libraries.
- On Error:
    - Worker sends signal to Master.
    - If multiple error on the same record, skip record.

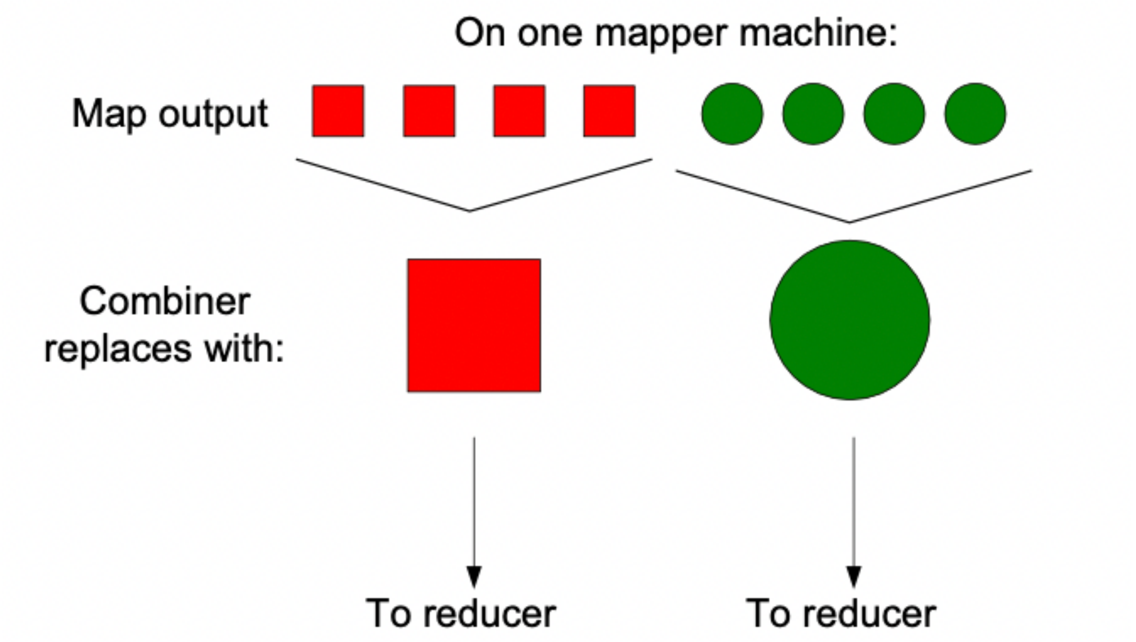
MapReduce: Miscellaneous Refinements

- Combiner function at a map task.
- Sorting Guarantees within each reduce partition.
- Local execution for debugging/testing.
- User-defined counters.

Combining Phase

- Run on map machines after map phase.
- “Mini-reduce”, only on local map output.
- Used to save bandwidth before sending data to full reduce tasks.
- Reduce tasks can be combiner if commutative & associative.

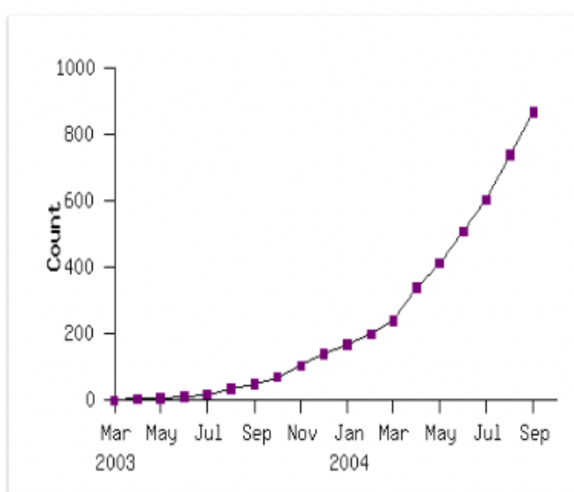
Combiner, Graphically



Examples of MapReduce Usage in Web Applications

- Distributed Grep.
- Count of URL Access Frequency.
- Clustering (K-means)
- Graph Algorithms.
- Indexing Systems

MapReduce Programs In Google Source Tree



Hadoop and Tools

- **Various Linux Hadoop clusters around**
  - Cluster +Hadoop
    - » http://hadoop.apache.org
  - Amazon EC2
- **Winows and other platforms**
  - The NetBeans plugin simulates Hadoop
  - The workflow view works on Windows
- **Hadoop-based tools**
  - For Developing in Java, NetBeans plugin
- **Pig Latin,** a SQL-like high level data processing script language
- **Hive,** Data warehouse, SQL
- **Mahout,** Machine Learning algorithms on Hadoop
- **HBase,** Distributed data store as a large table

More Reduce Applications

- Map Only processing.
- Filtering and accumulation.
- Database join.
- Reversing graph edges.
- Producing inverted index for web search.
- PageRank graph processing.

MapReduce Use Case 1: Map Only

Data distributive tasks – Map Only

- **E.g. classify individual documents**
- **Map does everything**
  - Input: (docno, doc_content), …
  - Output: (docno, [class, class, …]), …
- **No reduce tasks**

MapReduce Use Case 2: Filtering and Accumulation

Filtering & Accumulation – Map and Reduce

- **E.g. Counting total enrollments of two given student classes**
- **Map** selects records and outputs initial counts
  - In: (Jamie, 11741), (Tom, 11493), …
  - Out: (11741, 1), (11493, 1), …
- **Shuffle/Partition** by class_id
- **Sort**
  - In: (11741, 1), (11493, 1), (11741, 1), …
  - Out: (11493, 1), …, (11741, 1), (11741, 1), …
- **Reduce accumulates counts**
  - In: (11493, [1, 1, …]), (11741, [1, 1, …])
  - Sum and Output: (11493, 16), (11741, 35)

MapReduce Use Case 3: Database Join

- **A JOIN is a means for combining fields from two tables by using values common to each.**
- **Example :For each employee, find the department he works in**

Employee Table

LastName	DepartmentID
Rafferty	31
Jones	33
Steinberg	33
Robinson	34
Smith	34

JOIN

Pred:

EMPLOYEE.DepID=
DEPARTMENT.DepID

Department Table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

JOIN RESULT

LastName	DepartmentName
Rafferty	Sales
Jones	Engineering
Steinberg	Engineering
…	…

Problem: Massive lookups

- Given two large lists: (URL, ID) and (URL, doc_content) pairs
- Produce (URL, ID, doc_content) or (ID, doc_content)

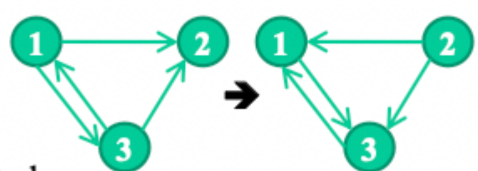
Solution:

- **Input stream**: both (URL, ID) and (URL, doc_content) lists
  - (http://del.icio.us/post, 0), (http://digg.com/submit, 1), …
  - (http://del.icio.us/post, <html0>), (http://digg.com/submit, <html1>), …
- **Map** simply passes input along,
- **Shuffle and Sort on URL** (group ID & doc_content for the same URL together)
  - Out: (http://del.icio.us/post, 0), (http://del.icio.us/post, <html0>), (http://digg.com/submit, <html1>), (http://digg.com/submit, 1), …
- **Reduce** outputs result stream of (ID, doc_content) pairs
  - In: (http://del.icio.us/post, [0, html0]), (http://digg.com/submit, [html1, 1]), …
  - Out: (0, <html0>), (1, <html1>), …

MapReduce Use Case 4: Reverse Graph Edge Directions & Output in Node Order

- **Input example: adjacency list of graph (3 nodes and 4 edges)**

(3, [1, 2]) (1, [3])
(1, [2, 3]) ➔ (2, [1, 3])
(3, [1])



- node_ids in the output **values** are also sorted. But Hadoop only sorts on keys!
- **MapReduce format**
  - Input: (3, [1, 2]), (1, [2, 3]).
  - Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse edge direction)
  - Out: (1,[3]) (2, [1, 3]) (3, [[1]).

MapReduce Use Case 5: Inverted Indexing Preliminaries

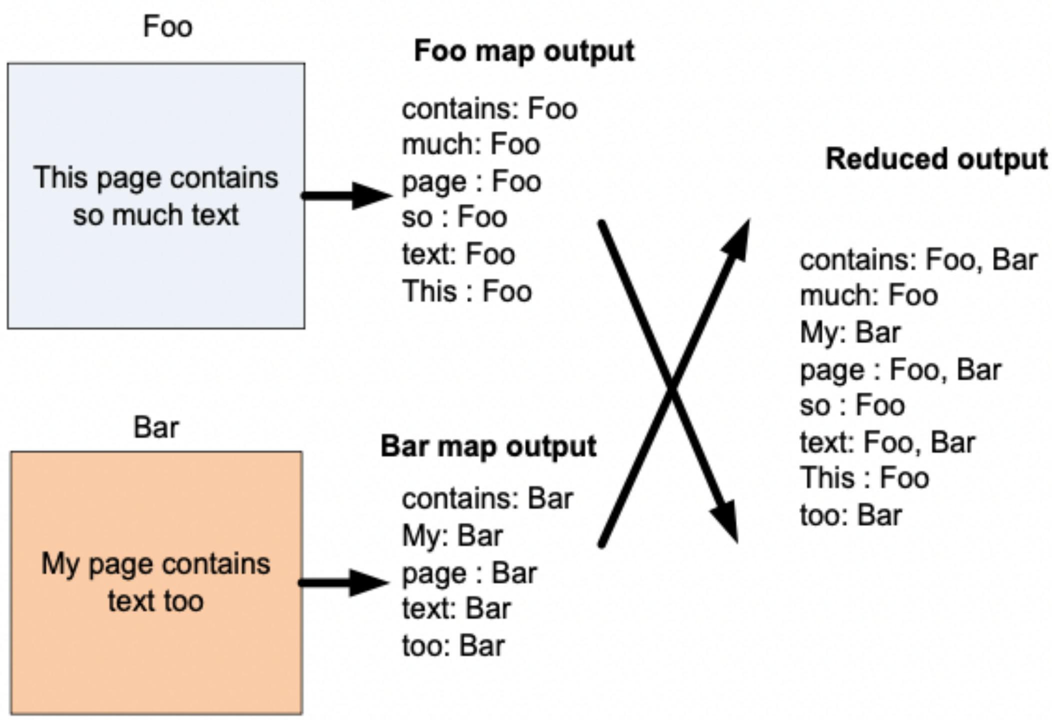
Construction of _inverted lists for document search

- Input: documents: (docid, [term, term..]), (docid, [term, ..]), ..
- Output: (term, [docid, docid, …])
  - E.g., (apple, [1, 23, 49, 127, …])

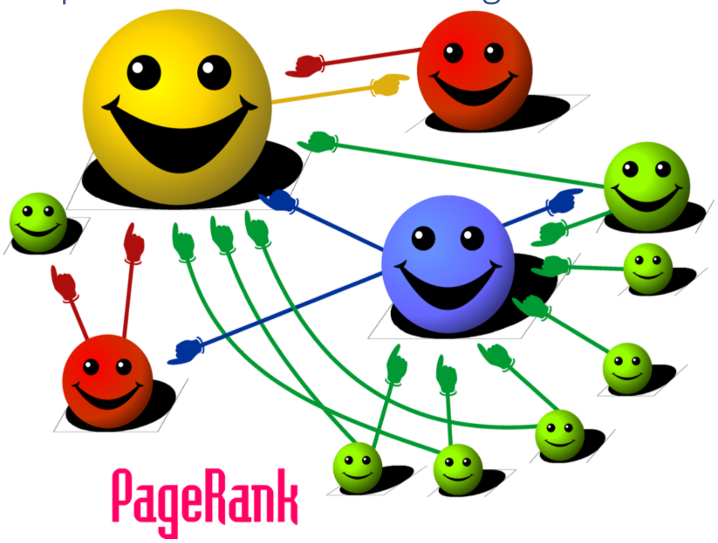
A document id is an internal document id, e.g., a unique integer

- Not an external document id such as a url

Inverted Index: Data Flow



MapReduce User Case 6: PageRank

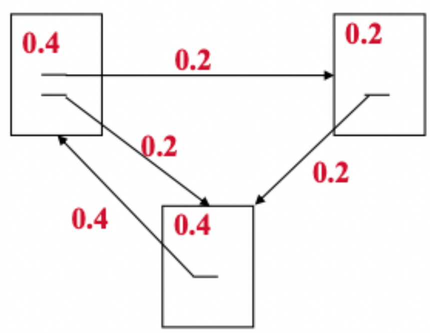


PageRank

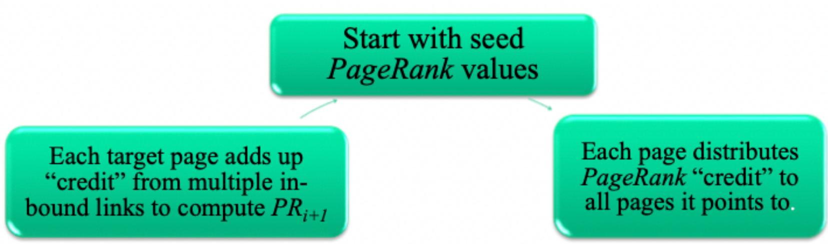
- Model page reputation on the web:

$$PR(x) = (1-d) + d\sum_{i=1}^{n}\frac{PR(t_i)}{C(t_i)}$$

- i=1, n lists all parents of page x.
- PR(x) is the page rank of each page.
- C(t) is the out-degree of t.
- d is damping factor.

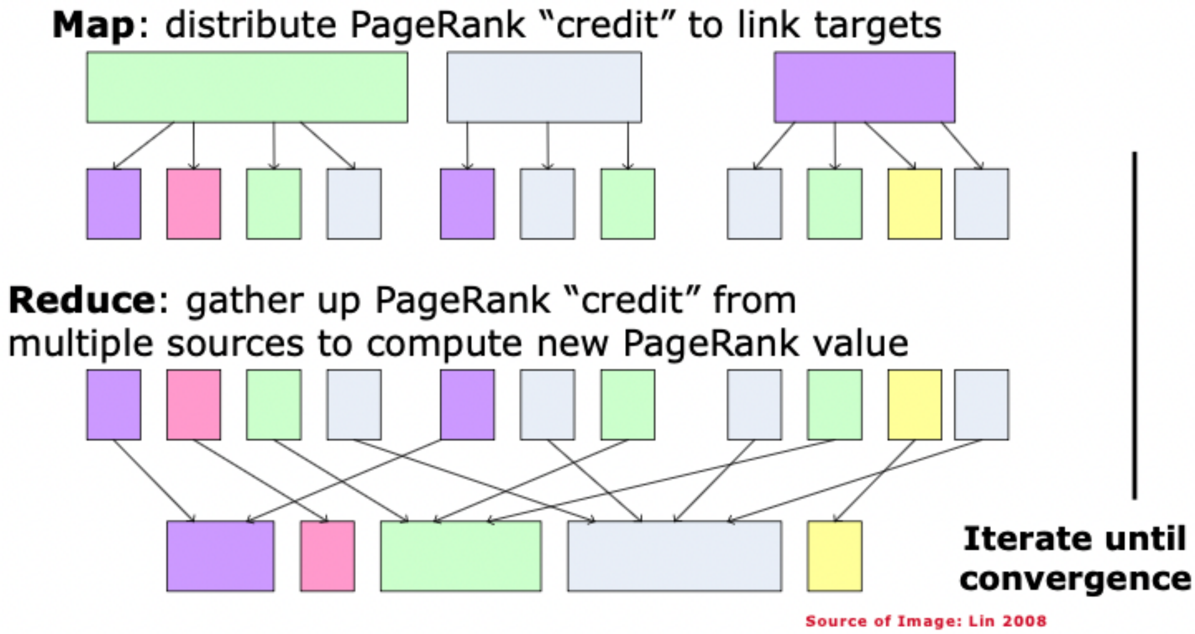


Computing PageRank Iteratively



- Effects at each iteration is local, i+1th iteration depends only on ith iteration.
- At iteration I, PageRank for individual nodes can be computed independently.

PageRank Using MapReduce



PageRank Calculation: Preliminaries

One PageRank iteration:

- Input:
  - $(id_1, [score_1^{(t)}, out_{11}, out_{12}, ..]), (id_2, [score_2^{(t)}, out_{21}, out_{22}, ..])$ ..
- Output:
  - $(id_1, [score_1^{(t+1)}, out_{11}, out_{12}, ..]), (id_2, [score_2^{(t+1)}, out_{21}, out_{22}, ..])$ ..

MapReduce elements

- Score distribution and accumulation
- Database join

PageRank: Score Distribution and Accumulation

- **Map**
  - In: (id_1, [$score_1^{(t)}$, out_{11}, out_{12}, ..]), (id_2, [$score_2^{(t)}$, out_{21}, out_{22}, ..]) ..
  - Out: (out_{11}, $score_1^{(t)}/n_1$), (out_{12}, $score_1^{(t)}/n_1$) .., (out_{21}, $score_2^{(t)}/n_2$), ..
- **Shuffle & Sort by node_id**
  - In: (id_2, $score_1$), (id_1, $score_2$), (id_1, $score_1$), ..
  - Out: (id_1, $score_1$), (id_1, $score_2$), .., (id_2, $score_1$), ..
- **Reduce**
  - In: (id_1, [$score_1$, $score_2$, ..]), (id_2, [$score_1$, ..]), ..
  - Out: (id_1, $score_1^{(t+1)}$), (id_2, $score_2^{(t+1)}$), ..

PageRank: Database Join to Associate out links With Score

- **Map**
  - In & Out: (id_1, $score_1^{(t+1)}$), (id_2, $score_2^{(t+1)}$), .., (id_1, [out_{11}, out_{12}, ..]), (id_2, [out_{21}, out_{22}, ..]) ..
- **Shuffle & Sort by node_id**
  - Out: (id_1, $score_1^{(t+1)}$), (id_1, [out_{11}, out_{12}, ..]), (id_2, [out_{21}, out_{22}, ..]), (id_2, $score_2^{(t+1)}$), ..
- **Reduce**
  - In: (id_1, [$score_1^{(t+1)}$, out_{11}, out_{12}, ..]), (id_2, [out_{21}, out_{22}, .., $score_2^{(t+1)}$]), ..
  - Out: (id_1, [$score_1^{(t+1)}$, out_{11}, out_{12}, ..]), (id_2, [$score_2^{(t+1)}$, out_{21}, out_{22}, ..]) ..

Conclusions

- **MapReduce advantages**
- **Application cases**
  - Map only: for totally distributive computation
  - Map+Reduce: for filtering & aggregation
  - Database join: for massive dictionary lookups
  - Secondary sort: for sorting on values
  - Inverted indexing: combiner, complex keys
  - PageRank: side effect files

Table of Contents