MapReduce

The new world of Big Data
(programming model)

Overview

- Background
- Google MapReduce
- The Hadoop Ecosystem
  - Core components:
    - Hadoop MapReduce
    - Hadoop Distributed File System (HDFS)
  - Other selected Hadoop projects:
    - HBase
    - Hive
    - Pig

The Computational Setting

- Computations that need the power of many computers
  - large datasets
  - use of thousands of CPUs in parallel
- Big data management, storage, and analytics
  - cluster as a computer

MapReduce & Hadoop: Historical Background

- 2003: **Google** publishes about its **cluster architecture** & distributed file system (**GFS**)
- 2004: Google publishes about its **MapReduce** programming model used on top of GFS
  - both GFS and MapReduce are written in C++ and are closed-source, with Python and Java APIs available to Google programmers only
- 2006: Apache & Yahoo! -> **Hadoop & HDFS**
  - **open-source**, Java implementations of Google MapReduce and GFS with a diverse set of APIs available to public
  - evolved from Apache Lucene/Nutch open-source web search engine (Nutch MapReduce and NDFS)
- 2008: Hadoop becomes an independent Apache project
  - Yahoo! uses Hadoop in production
- Today: Hadoop is used as a **general-purpose storage and analysis platform for big data**
  - other Hadoop distributions from several vendors including EMC, IBM, Microsoft, Oracle, Cloudera, etc.
  - many users (http://wiki.apache.org/hadoop/PoweredBy)
  - research and development actively continues…

Google: The Data Challenge

- Jeffrey Dean, Google Fellow, PACT'06 keynote speech:
  - 20+ billion web pages x 20KB = 400 TB
  - One computer can read 30-35 MB/sec from disk
    - ~ 4 months to read the web
  - ~ 1,000 hard drives just to store the web
  - Even more to "do" something with the data
  - **But:** Same problem with 1,000 machines < 3 hours

- MapReduce CACM'08 article:
  - 100,000 MapReduce jobs executed in Google every day
  - Total data processed > **20 PB of data per day**

Google Cluster Architecture: Key Ideas

- **Single-thread performance less important than scale-out**
  - For large problems, **total throughput/$** is more important than peak performance.
- **Stuff breaks**
  - If you have 1 server, it may stay up three years (1,000 days).
  - If you have 10,000 servers, expect to lose 10 per day.
- **"Ultra-reliable" hardware doesn't really help**
  - At large scales, the most reliable hardware still fails, albeit less often
    - Software still needs to be fault-tolerant
    - Commodity machines without fancy hardware give better **performance/$**

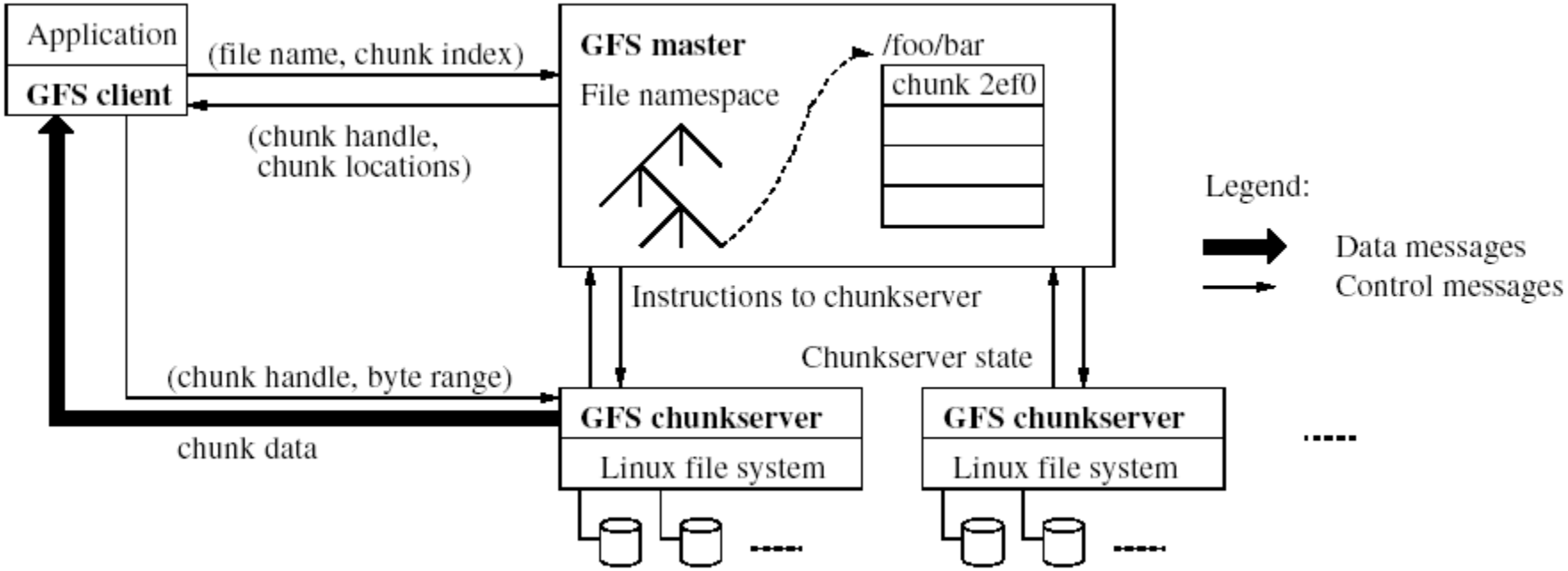
- Have a reliable computing infrastructure from clusters of unreliable commodity PCs.
- Replicate services across many machines to increase request throughput and availability.
- Favor price/performance over peak performance.

Google File System (GFS) Architecture

- Files divided into fixed-sized chunks (64 MB)
  - Each chunk gets a chunk handle from the master
  - Stored as Linux files
- One master
  - Maintains all file system metadata
  - Talks to each chunkserver periodically
- Multiple chunkservers
  - Store chunks on local disks
  - No caching of chunks (not worth it)
- Multiple clients
  - Clients talk to the master for metadata operations
  - Metadata can be cached at the clients
  - Read / write data from chunkservers

GFS Architecture

- Single master, multiple chunkservers



- To overcome single-point of failure & scalability bottleneck:
  - Use shadow masters
  - Minimize master involvement (large chunks; use only for metadata)

Overview of this Lecture Module

- Background
- Google MapReduce
- The Hadoop Ecosystem
  - Core components:
    - Hadoop MapReduce
    - Hadoop Distributed File System (HDFS)
  - Other selected Hadoop projects:
    - HBase
    - Hive
    - Pig

MapReduce

- a **software framework** first introduced by Google in 2004 to support parallel and fault-tolerant computations over large data sets on clusters of computers

- based on the **map/reduce functions** commonly used in the functional programming world

MapReduce in a Nutshell

- Given:
  - a very large dataset
  - a well-defined computation task to be performed on elements of this dataset (preferably, in a parallel fashion on a large cluster)

- MapReduce framework:
  - Just express what you want to compute (map() & reduce()).
  - Don’t worry about parallelization, fault tolerance, data distribution, load balancing (MapReduce takes care of these).
  - What changes from one application to another is the actual computation; the programming structure stays similar.

MapReduce in a Nutshell

- Here is the framework in simple terms:
  - Read lots of data.
  - **Map**: extract something that you care about from each record.
  - Shuffle and sort.
  - **Reduce**: aggregate, summarize, filter, or transform.
  - Write the results.

- One can use as many Maps and Reduces as needed to model a given problem.

MapReduce vs. Traditional RDBMS

	MapReduce	Traditional RDBMS
Data size	Petabytes	Gigabytes
Access	Batch	Interactive and batch
Updates	Write once, read many times	Read and write many times
Structure	Dynamic schema	Static schema
Integrity	Low	High (normalized data)
Scaling	Linear	Non-linear (general SQL)

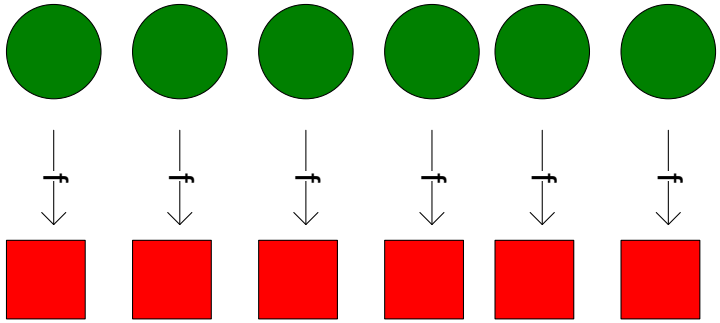
Functional Programming Foundations

- map in MapReduce ↔ map in FP
- reduce in MapReduce ↔ fold in FP

- Note: There is no precise 1-1 correspondence, but the general idea is similar.

map() in Haskell

- Create a new list by applying f to each element of the input list.



- **Definition of map:**

```
map :: (a → b) → [a] → [b]  -- type of map
map f []      = []  -- the empty list case
map f (x:xs)      = f x : map f xs -- the non-empty list case
```

- **Example: Double all numbers in a list.**

```
Haskell-prompt > map ((*) 2) [1, 2, 3]
[2, 4, 6]
```

Implicit Parallelism in map()

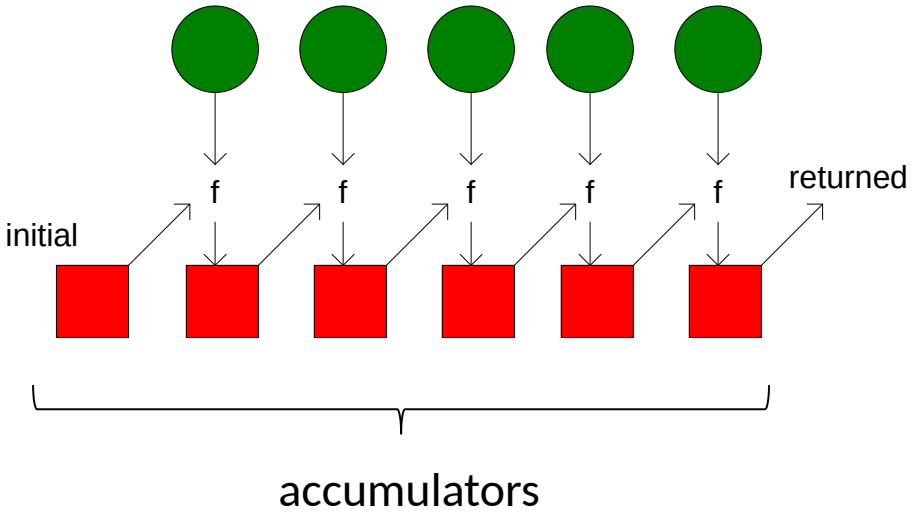
- In a purely functional setting, an element of a list being computed by map cannot see the effects of the computations on other elements.

- If the order of application of a function f to elements in a list is commutative, then we can reorder or parallelize execution.

- This is the “secret” that MapReduce exploits.

fold() in Haskell

- Move across a list, applying a function **f** to each element plus an **accumulator**. f returns the next accumulator value, which is combined with the next element of the list.



- Two versions: fold left & fold right

fold() in Haskell

- **Definition of fold left:**

```
foldl :: (b → a → b) → b → [a] → b -- type of foldl
foldl f y []     = y                -- the empty list case
foldl f y (x:xs) = foldl f (f y x) xs   -- the non-empty list case
```

- **Definition of fold right:**

```
foldr :: (a → b → b) → b → [a] → b -- type of foldr
foldr f y []     = y                -- the empty list case
foldr f y (x:xs) = f x (foldr f y xs)   -- the non-empty list case
```

- **Example: Compute the sum of all numbers in a list.**

```
Haskell-prompt > foldl (+) 0 [1, 2, 3]
6
```

```
foldl (+) 0 [1, 2, 3]
⇒ (((0 + 1) + 2) + 3)
⇒ 6
```

reduce() in Haskell

- reduce is a type-specialized version of fold.

- **Definition of reduce:**

  reduce :: (a → a → a) → a → [a] → a -- type of reduce

  reduce = foldl -- definition of reduce

MapReduce Basic Programming Model

- Transform a set of input key-value pairs to a set of output values:
  - Map: (k1, v1) → list(k2, v2)
  - MapReduce library groups all intermediate pairs with same key together.
  - Reduce: (k2, list(v2)) → list(v2)

MapReduce Canonical Example

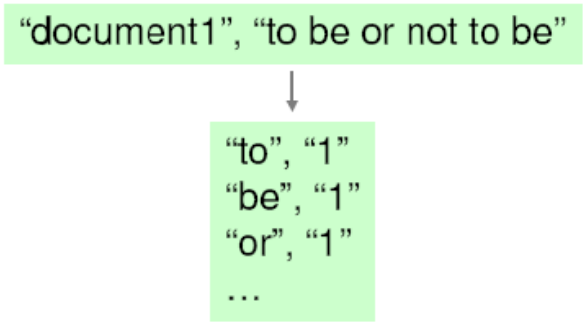
"Count word occurrences in a set of documents."

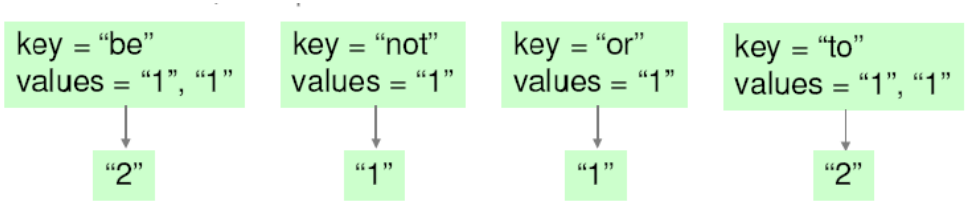
map(k1, v1) → list(k2, v2)

map (String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(k2, list(v2)) → list(v2)

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

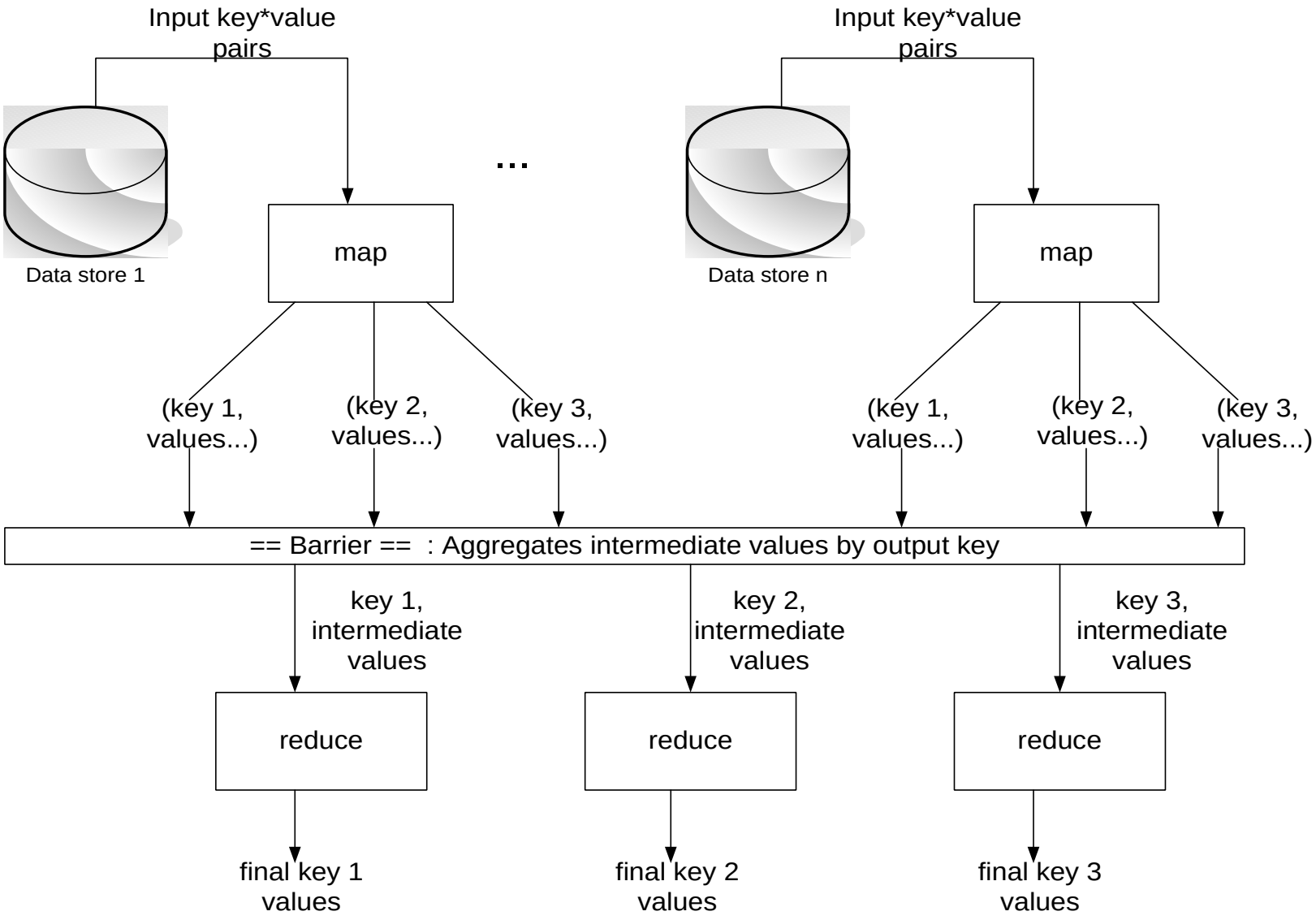




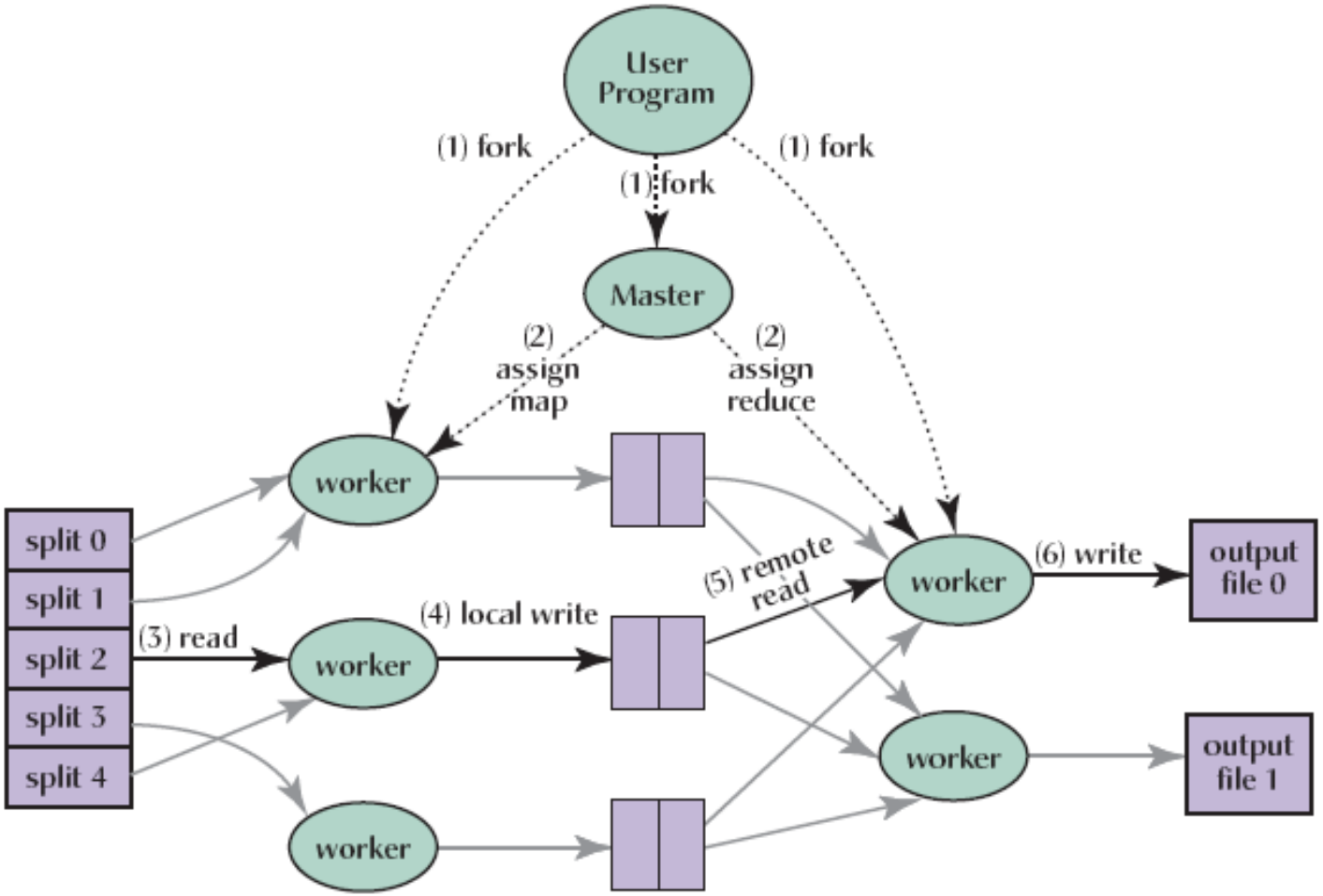
MapReduce Parallelization

- Multiple map() functions run in parallel, creating different intermediate values from different input data sets.
- Multiple reduce() functions also run in parallel, each working on a different output key.
- All values are processed independently.
- **Bottleneck: The reduce phase can’t start until the map phase is completely finished.**

MapReduce Parallel Processing Model



MapReduce Execution Overview



MapReduce Scheduling

- One master, many workers
  - Input data split into M map tasks (typically 64 MB (~ chunk size in GFS))
  - Reduce phase partitioned into R reduce tasks (hash(k) mod R)
  - Tasks are assigned to workers dynamically

- Master assigns each map task to a free worker
  - Considers locality of data to worker when assigning a task
  - Worker reads task input (often from local disk)
  - Worker produces R local files containing intermediate k/v pairs

- Master assigns each reduce task to a free worker
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user's reduce operation to produce the output

Choosing M and R

- M = number of map tasks, R = number of reduce tasks
- Larger M, R: creates smaller tasks, enabling easier load balancing and faster recovery (many small tasks from failed machine)
- Limitation: O(M+R) scheduling decisions and O(M*R) in-memory state at master
  - Very small tasks not worth the startup cost
- Recommendation:
  - Choose M so that split size is approximately 64 MB
  - Choose R a small multiple of the number of workers; alternatively choose R a little smaller than #workers to finish reduce phase in one "wave"

MapReduce Fault Tolerance

- On worker failure:
  - Master detects failure via periodic heartbeats.
  - Both completed and in-progress map tasks on that worker should be re-executed (→ output stored on local disk).
  - Only in-progress reduce tasks on that worker should be re-executed (→ output stored in global file system).
  - All reduce workers will be notified about any map re-executions.
- On master failure:
  - State is check-pointed to GFS: new master recovers & continues.
- Robustness:
  - Example: Lost 1600 of 1800 machines once, but finished fine.

MapReduce Data Locality

- Goal: To conserve network bandwidth.
- In GFS, data files are divided into 64 MB blocks and 3 copies of each are stored on different machines.
- Master program schedules map() tasks based on the location of these replicas:
  - Put map() tasks physically on the same machine as one of the input replicas (or, at least on the same rack / network switch).
- This way, thousands of machines can read input at local disk speed. Otherwise, rack switches would limit read rate.

Stragglers & Backup Tasks

- Problem: “Stragglers” (i.e., slow workers) significantly lengthen the completion time.
- Solution: Close to completion, spawn backup copies of the remaining in-progress tasks.
  - Whichever one finishes first, “wins”.
- Additional cost: a few percent more resource usage.
- Example: A sort program without backup = 44% longer.

Other Practical Extensions

- User-specified **combiner functions** for partial combination within a map task can save network bandwidth (~ mini-reduce)
  - Example: Word Count?
- User-specified **partitioning functions** for mapping intermediate key values to reduce workers (by default: hash(key) mod R)
  - Example: hash(Hostname(urlkey)) mod R
- **Ordering guarantees**: Processing intermediate k/v pairs in increasing order
  - Example: reduce of Word Count outputs ordered results.
- Custom input and output format handlers
- Single-machine execution option for testing & debugging

Basic MapReduce Program Design

- Tasks that can be performed independently on a data object, large number of them: Map
- Tasks that require combining of multiple data objects: Reduce
- Sometimes it is easier to start program design with Map, sometimes with Reduce
- Select keys and values such that the right objects end up together in the same Reduce invocation
- Might have to partition a complex task into multiple MapReduce sub-tasks