



## MapReduce and HDFS

This presentation includes course content © University of Washington  
Redistributed under the Creative Commons Attribution 3.0 license.



All other contents:  
© 2009 Cloudera, Inc.

# Overview

- Why MapReduce?
- What is MapReduce?
- The Hadoop Distributed File System

# How MapReduce is Structured

- Functional programming meets distributed computing
- A batch data processing system
- Factors out many reliability concerns from application logic

# MapReduce Provides:

- Automatic parallelization & distribution
- Fault-tolerance
- Status and monitoring tools
- A clean abstraction for programmers

# Programming Model

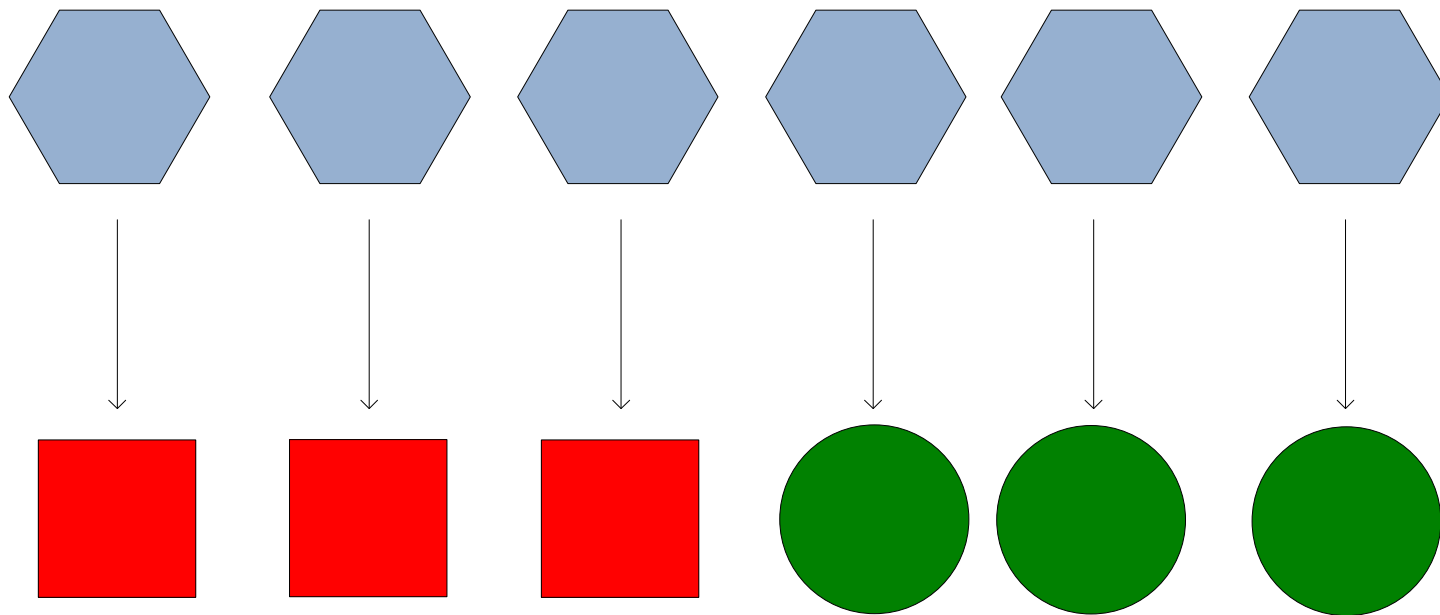
- Borrows from functional programming
- Users implement interface of two functions:
  - `map (in_key, in_value) -> (out_key, intermediate_value) list`
  - `reduce (out_key, intermediate_value list) -> out_value list`

# map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key\*value pairs: e.g., (filename, line).
- map() produces one or more *intermediate* values along with an output key from the input.

# map

```
map (in_key, in_value) ->  
    (out_key, intermediate_value) list
```



# Example: Upper-case Mapper

```
let map(k, v) =  
    emit(k.toUpper(), v.toUpper())
```

("foo", "bar") → ("FOO", "BAR")

("Foo", "other") → ("FOO", "OTHER")

("key2", "data") → ("KEY2", "DATA")



# Example: Explode Mapper

```
let map(k, v) =  
  foreach char c in v:  
    emit(k, c)
```

`("A", "cats") → ("A", "c"), ("A", "a"),  
("A", "t"), ("A", "s")`

`("B", "hi") → ("B", "h"), ("B", "i")`

# Example: Filter Mapper

```
let map(k, v) =  
  if (isPrime(v)) then emit(k, v)
```

("foo", 7) → ("foo", 7)

("test", 10) → *(nothing)*

# Example: Changing Keyspaces

```
let map(k, v) = emit(v.length(), v)
```

```
("hi", "test") → (4, "test")
```

```
("x", "quux") → (4, "quux")
```

```
("y", "abracadabra") → (10, "abracadabra")
```

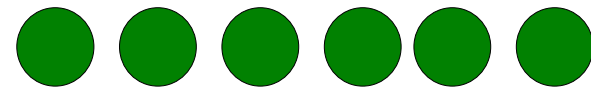
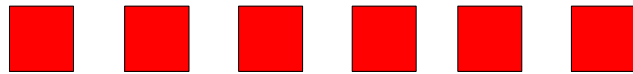
# reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- `reduce()` combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)

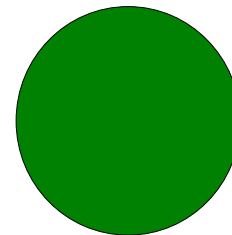
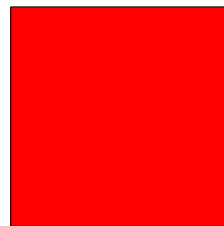
# reduce

`reduce (out_key, intermediate_value list) ->  
out_value list`

initial



returned



# Example: Sum Reducer

```
let reduce(k, vals) =  
  sum = 0  
  foreach int v in vals:  
    sum += v  
  emit(k, sum)
```

("A", [42, 100, 312]) → ("A", 454)

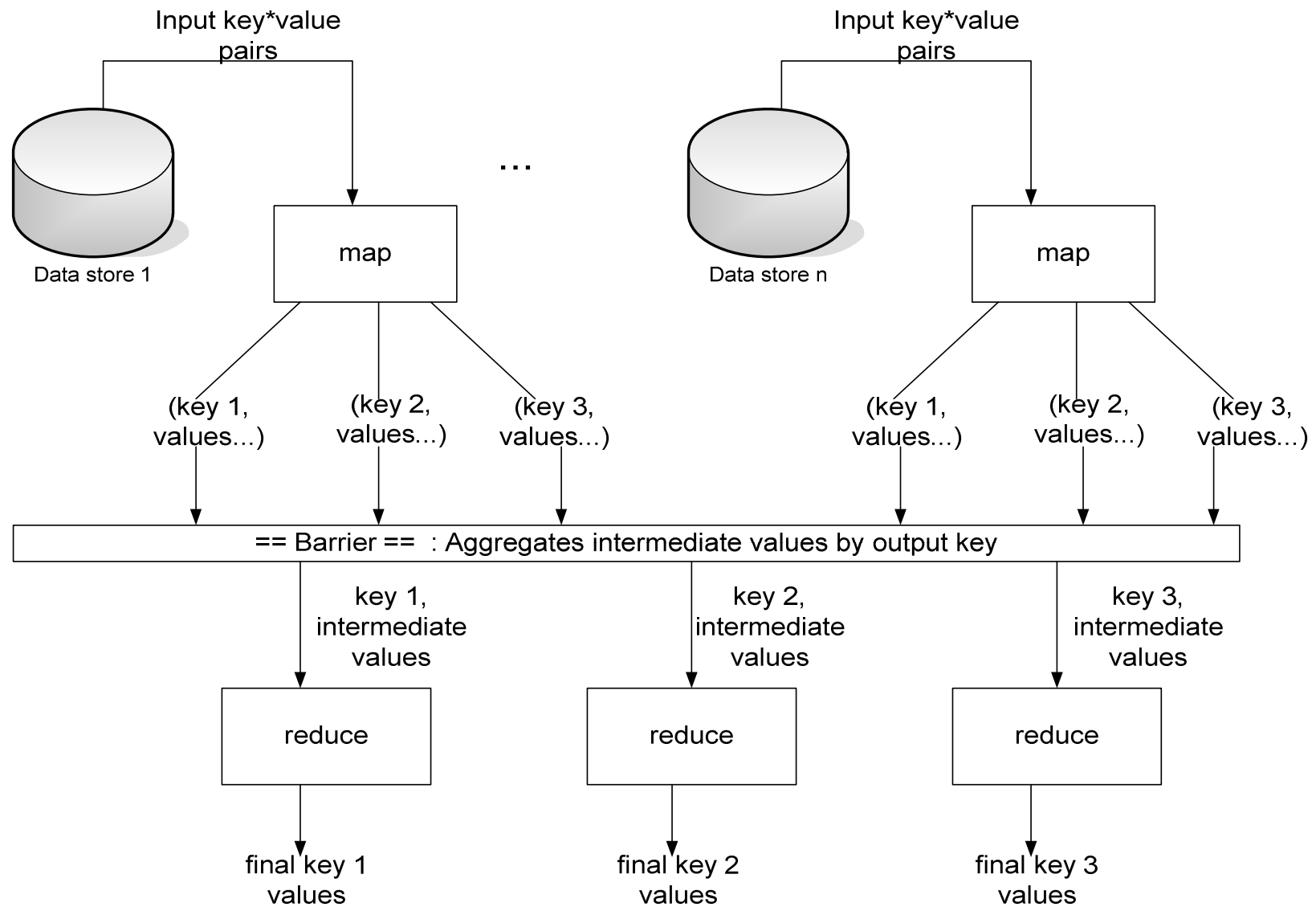
("B", [12, 6, -2]) → ("B", 16)

# Example: Identity Reducer

```
let reduce(k, vals) =  
  foreach v in vals:  
    emit(k, v)
```

("A", [42, 100, 312]) → ("A", 42),  
("A", 100), ("A", 312)

("B", [12, 6, -2]) → ("B", 12), ("B", 6),  
("B", -2)





# Parallelism

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

# Example: Count word occurrences

```
map(String input_key, String input_value):  
    // input_key: document name  
    // input_value: document contents  
    for each word w in input_value:  
        emit(w, 1);  
  
reduce(String output_key, Iterator<int>  
    intermediate_values):  
    // output_key: a word  
    // output_values: a list of counts  
    int result = 0;  
    for each v in intermediate_values:  
        result += v;  
    emit(output_key, result);
```

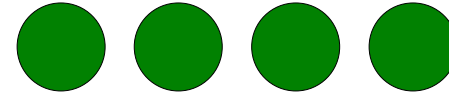
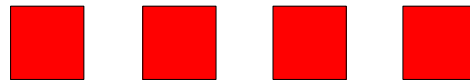
# Combining Phase

- Run on mapper nodes after map phase
- “Mini-reduce,” only on local map output
- Used to save bandwidth before sending data to full reducer
- Reducer can be combiner if commutative & associative
  - e.g., SumReducer

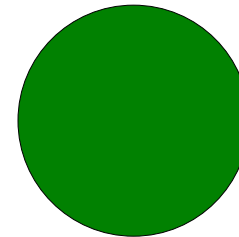
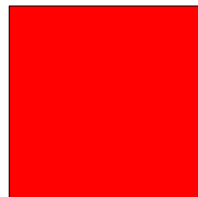
# Combiner, graphically

On one mapper machine:

Map output



Combiner  
replaces with:



To reducer

To reducer

# WordCount Redux

```
map(String input_key, String input_value):  
    // input_key: document name  
    // input_value: document contents  
    for each word w in input_value:  
        emit(w, 1);  
  
reduce(String output_key, Iterator<int>  
    intermediate_values):  
    // output_key: a word  
    // output_values: a list of counts  
    int result = 0;  
    for each v in intermediate_values:  
        result += v;  
    emit(output_key, result);
```

# MapReduce Conclusions

- MapReduce has proven to be a useful abstraction in many areas
- Greatly simplifies large-scale computations
- Functional programming paradigm can be applied to large-scale applications
- You focus on the “real” problem, library deals with messy details

# HDFS

Some slides designed by Alex Moschuk, University of Washington  
Redistributed under the Creative Commons Attribution 3.0 license

# HDFS: Motivation

- Based on Google's GFS
- Redundant storage of massive amounts of data on cheap and unreliable computers
- Why not use an existing file system?
  - Different workload and design priorities
  - Handles much bigger dataset sizes than other filesystems



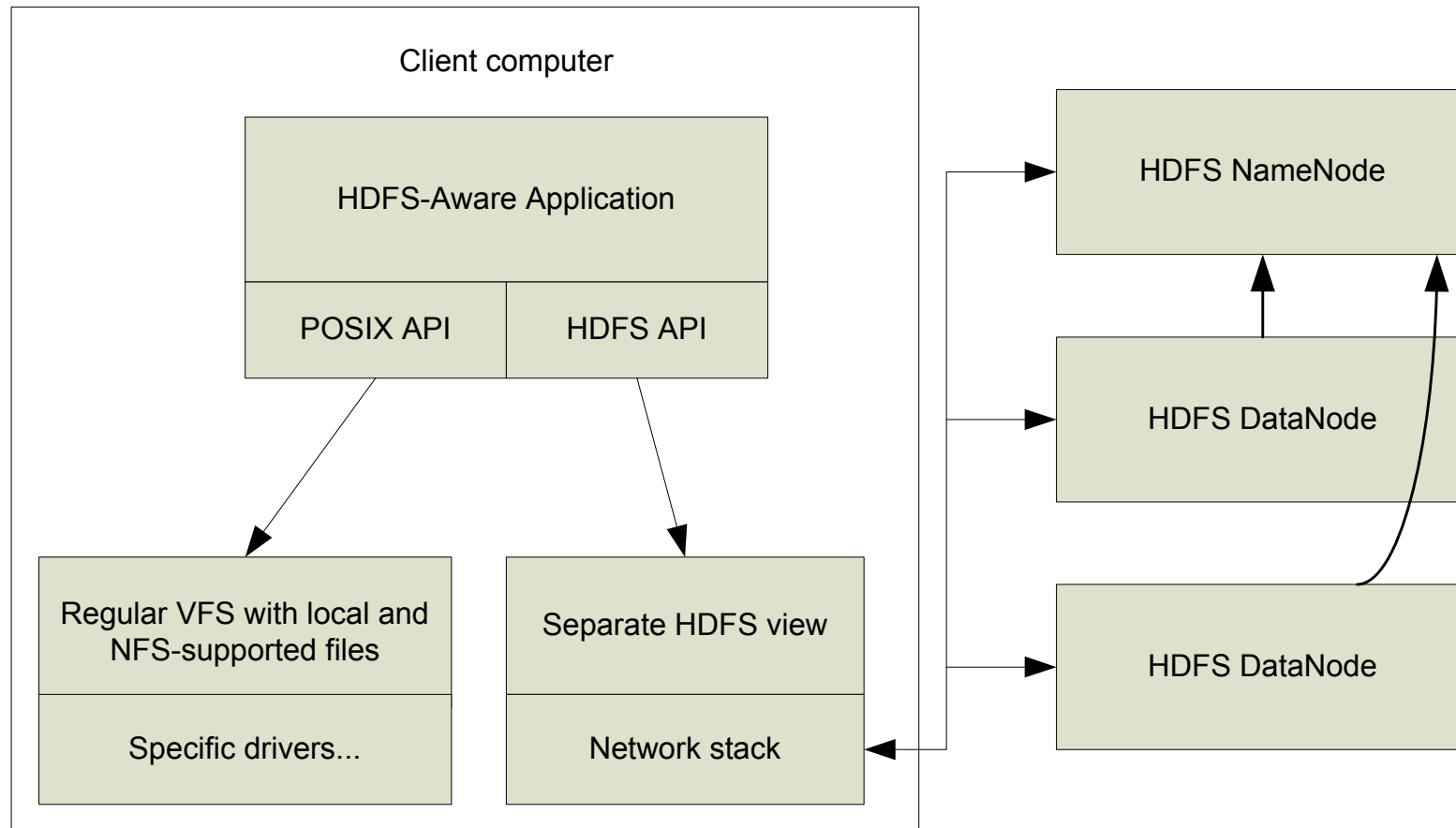
# Assumptions

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

# HDFS Design Decisions

- Files stored as blocks
  - Much larger size than most filesystems (default is 64MB)
- Reliability through replication
  - Each block replicated across 3+ *DataNodes*
- Single master (NameNode) coordinates access, metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large data sets, streaming reads
- Familiar interface, but customize the API
  - Simplify the problem; focus on distributed apps

# HDFS Client Block Diagram



# Based on GFS Architecture

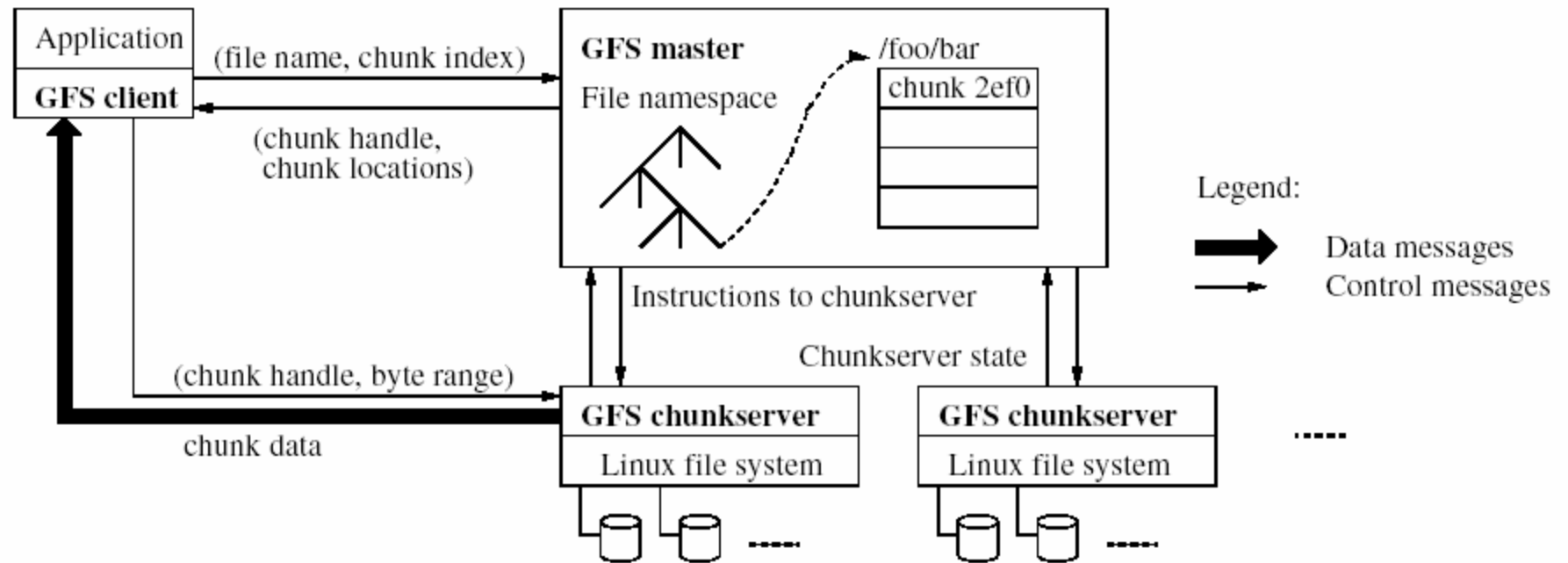


Figure from "The Google File System,"  
Ghemawat et. al., SOSP 2003

# Metadata

- Single *NameNode* stores all metadata
  - Filenames, locations on DataNodes of each file
- Maintained entirely in RAM for fast lookup
- DataNodes store opaque file contents in “block” objects on underlying local filesystem

# HDFS Conclusions

- HDFS supports large-scale processing workloads on commodity hardware
  - designed to tolerate frequent component failures
  - optimized for huge files that are mostly appended and read
  - filesystem interface is customized for the job, but still retains familiarity for developers
  - simple solutions can work (e.g., single master)
- Reliably stores several TB in individual clusters

