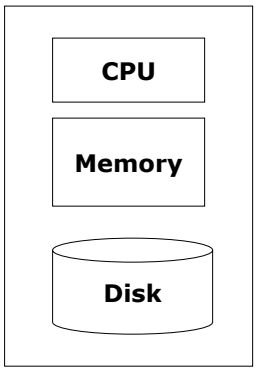
GFS & MAPREDUCE

SINGLE-NODE ARCHITECTURE



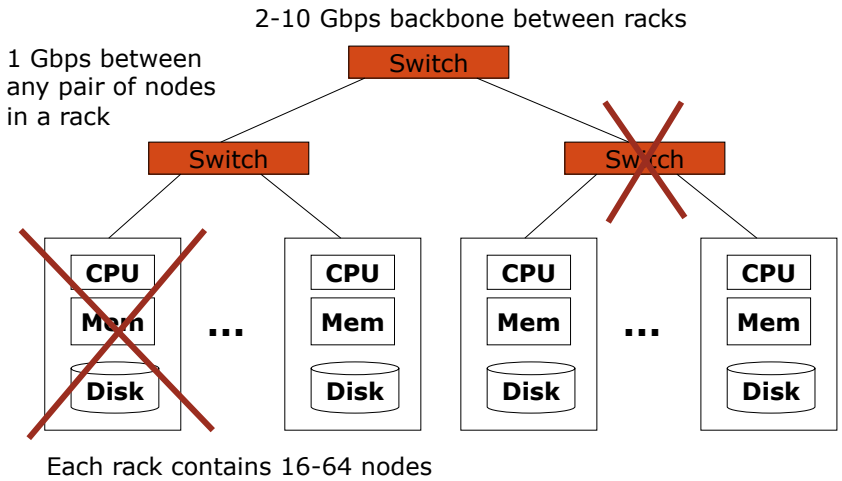
Machine Learning, Statistics

"Classical" Data Mining

COMMODITY CLUSTERS

- Web data sets can be very large
  - Tens to hundreds of terabytes
- Cannot mine on a single server (why?)
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnect
- How to organize computations on this architecture?
  - Mask issues such as hardware failure

CLUSTER ARCHITECTURE



DISTRIBUTED FILE SYSTEMS

STABLE STORAGE

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

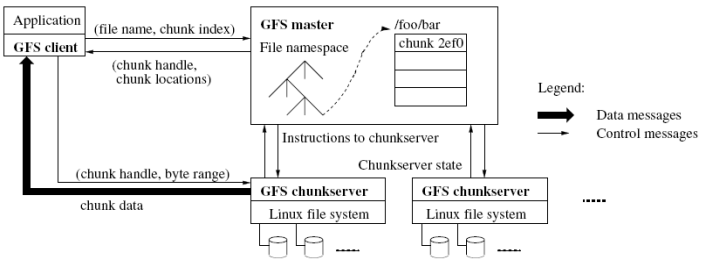
INTRODUCTION TO GFS

- GFS is a scalable, distributed file system
- Developed to meet the rapidly growing data processing needs of Google
- Design is driven by key observations of Google's technological environment:
  - Files are huge by traditional standards
  - Appending new data is common than overwriting existing one
  - Component failures are norm rather than exception

DESIGN OVERVIEW - ASSUMPTIONS

- The system must be able to detect and recover from component failures routinely
- Multi-GB sized files are common. Small files need not be optimized
- Many large, sequential writes that append data
- Synchronization between hundreds of reads and writes should be possible
- Faster processing of data in bulk is more important than faster individual read/write operations

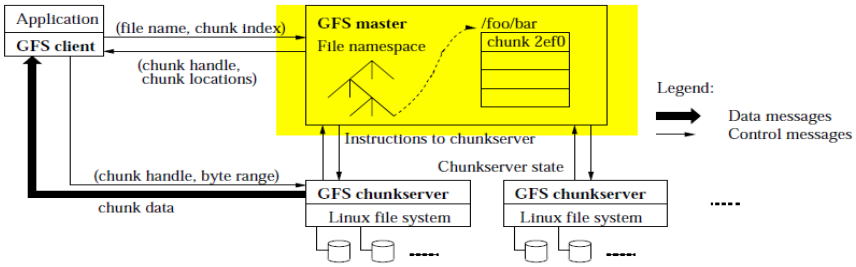
ARCHITECTURE



- A GFS cluster consists of single master and multiple chunkservers
- Each of these is a Linux machine running user level server process
- Files are divided into fixed-size chunks (16-64MB, replicated 2x or 3x) each having a 64 bit handle
- Chunkservers store chunks on local disks as Linux files

9

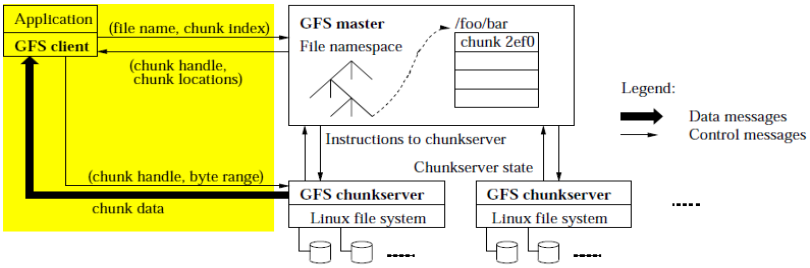
ARCHITECTURE – GFS MASTER



- The master maintains all the file system metadata which includes namespace, access control information, file-to-chunk mapping
- Controls system-wide activities such as garbage collection of chunks
- Communicates with each chunkserver to give instructions and collect its state

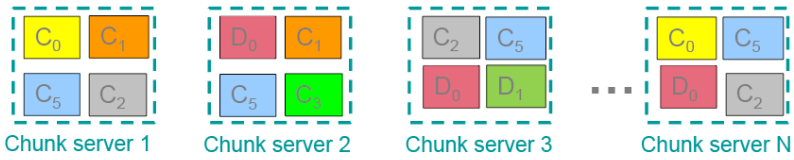
10

ARCHITECTURE – GFS CLIENT



- GFS client code is linked into each application
- Clients communicate with master for metadata operations
- Clients interact with chunkservers for data-bearing operations
- Client code implements the file system APIs
- Clients do not cache data, but they cache metadata

DISTRIBUTED FILE SYSTEM



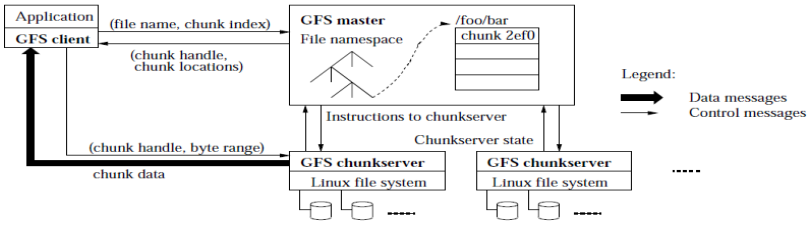
Bring computation directly to the data!

METADATA

- All metadata is stored in master's memory
- Three types of metadata:
  - File and chunk namespaces
  - Mapping from files to chunks
  - Location of each chunk's replica
- Master does not store chunk information persistently
- Collects information from chunkservers at start up
- Periodic scanning
  - Implement chunk garbage collection
  - Chunk migration for load and disk space balancing

13

READ OPERATION

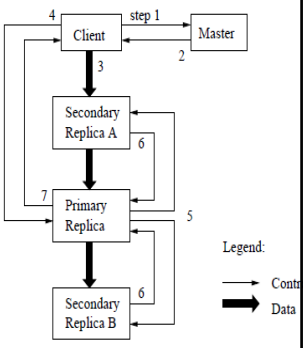


- Client translates file name and bytes offset into a chunk index within a file
- Sends a request to master with file name and index
- Master replies with chunk handle and location of replicas
- Client sends request to the nearest replica (chunkserver)
- Chunkserver replies with the requested data

14

CONTROL FLOW OF A WRITE OPERATION

1. Client asks master which chunkserver holds lease for the chunk
2. Master replies with identity of primary and locations of secondary replicas
3. Client pushes data to all replicas which is stored in an LRU buffer cache by replicas
4. Client sends a write request to primary replica to apply mutation to local state
5. Primary forwards the write request to all replicas
6. Secondaries reply to primary indicating operation completion
7. Primary replies to the client with either success message or with any errors encountered during this operation



15

GARBAGE COLLECTION

- After a file is deleted, GFS does not immediately reclaim the available physical storage.
  - All the references to chunks are in the file-to-chunk mappings, which is maintained by the master
- The other replica not known to the master is "garbage"
- Garbage collection is done when master is relatively free in a background activity
- Provides a safety net against accidental and irreversible deletion

16

FAULT TOLERANCE AND DIAGNOSIS

- One major challenge is to deal with component failures
- Strategies adopted for high availability:
  - Fast Recovery: both master and chunkservers are designed to restore their state and start in seconds
  - Chunk Replication: each chunk is replicated on multiple racks
  - Master replication: The master state is replicated for reliability. Its operation log and checkpoints are replicated
- Each chunkserver uses checksums to detect corruption of stored data

FAULT TOLERANCE AND DIAGNOSIS

- A chunk is broken up into 64 KB blocks and each such block has a 32 bit checksum.
  - Checksums are kept in memory, stored persistently with logging
- GFS servers generate diagnostic logs that record
  - Significant events like chunk servers going up and down
  - RPC requests and replies

REAL WORLD CLUSTERS

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

REAL WORLD CLUSTERS——READ/WRITE RATES

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

CONCLUSIONS

- GFS is a system for handling huge data-processing workloads using commodity hardware
  - Delivers high aggregate throughput to many concurrent readers and writers
  - File system control is kept separate, which passes through master
  - Data transfer directly passes between chunk servers and client

MAPREDUCE

OUTLINE

- Parallelism
  - Data parallelism
  - Task parallelism
- MapReduce programming model
- Implementation Issues

DATA PARALLELISM

```
x := (a * b) + (y * z);
```

computation A computation B

- At the micro level, independent algebraic operations can commute – be processed in any order.
- If commutative operations are applied to different memory addresses, then they can also occur at the same time
- Compilers, CPUs often do so automatically

HIGHER-LEVEL PARALLELISM

```
x :=  foo(a)  +  bar(b)
```

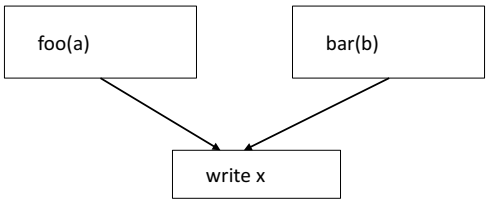
computation A

computation B

- Commutativity can apply to larger operations. If foo() and bar() do not manipulate the same memory, then there is no reason why these cannot occur at the same time

PARALLELISM: DEPENDENCY GRAPHS

```
x :=  foo(a)  +  bar(b)
```



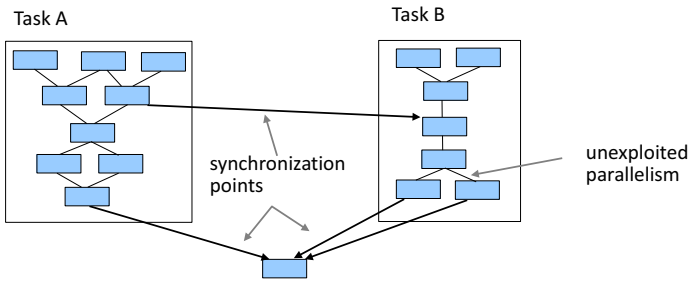
- Arrows indicate dependent operations
- write x operation waits for predecessors to complete
- If foo and bar do not access the same memory, there is not a dependency between them
- These operations can occur in parallel in different threads

DEPENDENCY GRAPHS: FULL PARALLELISM?

- Creating dependency graphs requires sometimes-difficult reasoning about isolated processes
- I/O and other shared resources besides memory introduce dependencies
- More threads => more communication; this adds overhead and complexity

TASK-LEVEL PARALLELISM

- Dividing work into larger "tasks" identifies logical units for parallelization as threads

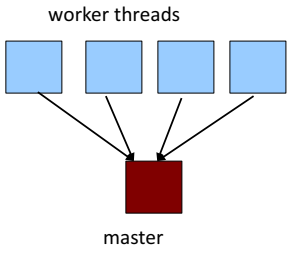


TASK-LEVEL PARALLELISM

- Intelligent task design eliminates as many synchronization points as possible, but some will be inevitable
- Independent tasks can operate on different physical machines in distributed fashion
- Good task design requires identifying common data and functionality to move as a unit

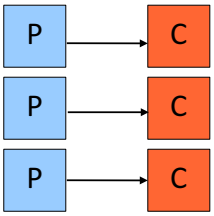
MASTER/WORKERS

- One object called the master initially owns all data.
- Creates several workers to process individual elements
- Waits for workers to report results back



PRODUCER/CONSUMER FLOW

- Producer threads create work items
- Consumer threads process them
- Can be daisy-chained



WARM UP: WORD COUNT

- We have a large file of words, one word to a line
- Count the number of times each distinct word appears in the file
- Sample application: analyze web server logs to find popular URLs

WORD COUNT (2)

- Case 1: Entire file fits in memory
- Case 2: File too large for mem, but all <word, count> pairs fit in mem
- Case 3: File on disk, too many distinct words to fit in memory

WORD COUNT (3)

- To make it slightly harder, suppose we have a large corpus of documents
- Count the number of times each distinct word occurs in the corpus
- The above captures the essence of MapReduce
  - Great thing is that it is naturally parallelizable

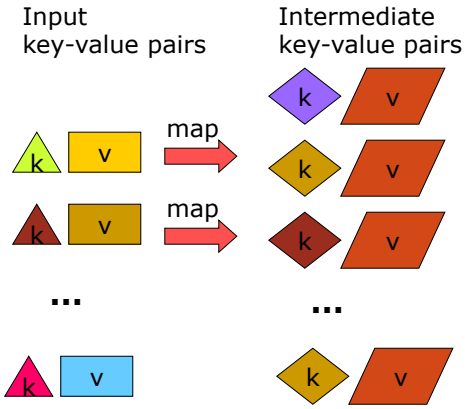
MAPREDUCE MOTIVATION

- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- Want to make this easy

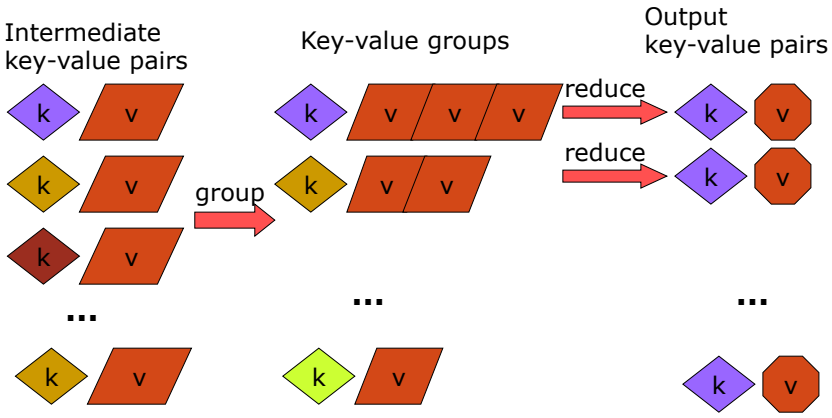
MAPREDUCE

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers

MAPREDUCE: THE MAP STEP



MAPREDUCE: THE REDUCE STEP



MAPREDUCE

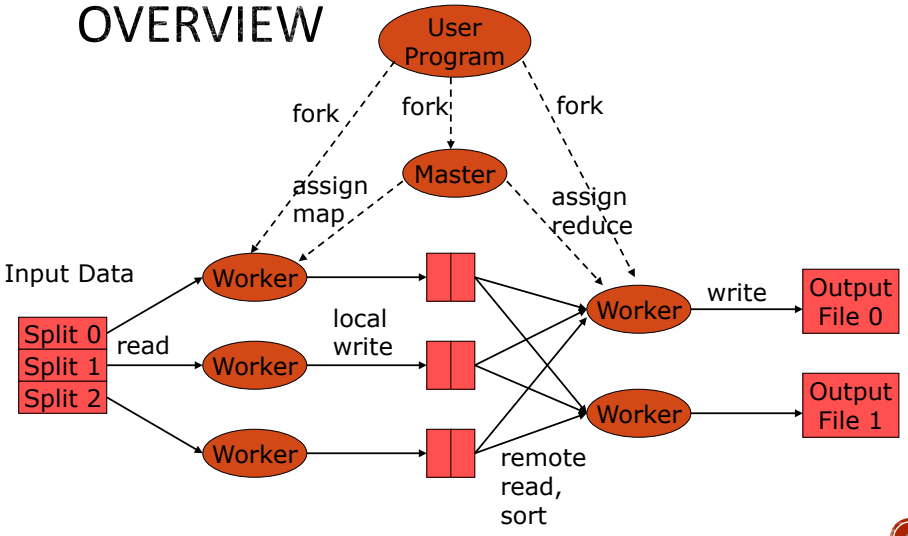
- Input: a set of key/value pairs
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs

EXAMPLE: WORD COUNTING IN A LARGE CORPUS

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

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

DISTRIBUTED EXECUTION OVERVIEW



41

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.

42

DATA FLOW

- Input, final output are stored on a distributed file system
  - 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

MORE EXAMPLES

- Distributed Grep:
  - Map() emits a line if it matches a supplied pattern
  - Reduce() is an identity function that just copies the supplied intermediate data to output.
- Count of URL Access Frequency
  - Map() processes logs of web page requests and outputs (URL, 1)
  - Reduce() adds together all values for the same URL and emits (URL, total count)

OTHER EXAMPLES

- Distributed sort
- Web link-graph reversal
- Term-vector per host
- Web access log stats
- Inverted index construction
- Document clustering
- Machine learning
- Statistical machine translation
- …

COORDINATION

- 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
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks

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

HOW MANY MAP AND REDUCE JOBS?

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

COMBINERS

- Often a map task will produce many pairs of the form (k,v1), (k,v2), ... for the same key k
  - E.g., popular words in Word Count
- Can save network time by pre-aggregating at mapper
  - combine(k1, list(v1)) → v2
  - Usually same as reduce function
- Works only if reduce function is commutative and associative

PARTITION FUNCTION

- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function e.g., hash(key) mod R
- Sometimes useful to override
  - E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file

IMPLEMENTATIONS

- Google
  - Not available outside Google
- Hadoop
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://hadoop.apache.org
- Aster Data
  - Cluster-optimized SQL Database that also implements MapReduce

CLOUD COMPUTING

- Ability to rent computing by the hour
  - Additional services e.g., persistent storage
- Amazon's "Elastic Compute Cloud" (EC2)
- Aster Data and Hadoop can both be run on EC2

FURTHER READING

- Jeffrey Dean and Sanjay Ghemawat,

  MapReduce: Simplified Data Processing on Large Clusters

http://labs.google.com/papers/mapreduce.html

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, **The Google File System**

http://labs.google.com/papers/gfs.html