Security of Cloud Computing

Fabrizio Baiardi
f.baiardi@unipi.it

Syllabus

- Cloud Computing Introduction
  - Definitions
  - Economic Reasons
  - Service Model
  - Deployment Model
- → Supporting Technologies
  - Virtualization Technology
  - Scalable Computing = Elasticity
- Security
  - New Threat Model
  - New Attacks
  - Countermeasures

Elastic, scalable, parallel

- All these term implies that not all the computations can exploit at best a cloud architecture
- If an application running on a single machine is migrated to a single VM on a cloud
  - Benefits in term of availability and reliability because the whole VM can be migrated to another machine if the current one crashes
  - Crash cannot be masked
  - No benefit in terms of performance even if a large number of physical hosts is available
- An application should be conceived from the beginning as a huge number of processess mapped onto a large number of VMs
- The final performance depends on the mapping of the VMs onto the physical hosts under the assumption that there is a minimal performance loss with respect to the sequential case when several VMs are mapped onto the same node

Elastic, Scalable, Parallel Computing

This point discusses

a) how can we decompose an application into a large number of concurrent processes

b) how to structure the software to support an application that has been decomposed into a large number of concurrent processes

Goal

a) simple decomposition

b) linear speed up: the time to implement a computation decreases as 1/n where n is the number of processes resulting from the decomposition

Parallel and Distributed Computing

Parallel computing can apply distinct strategies :

Vector processing of data (SIMD)

Multiple CPUs in a single computer (MIMD)

Distributed computing is multiple CPUs across many computers (MIMD distributed memory)

Example of distributed computing

- Weather prediction
- Indexing the web (Google)
- Simulating an Internet-sized network for networking experiments (PlanetLab)
- Speeding up content delivery (Akamai)

A Brief History

1975-85

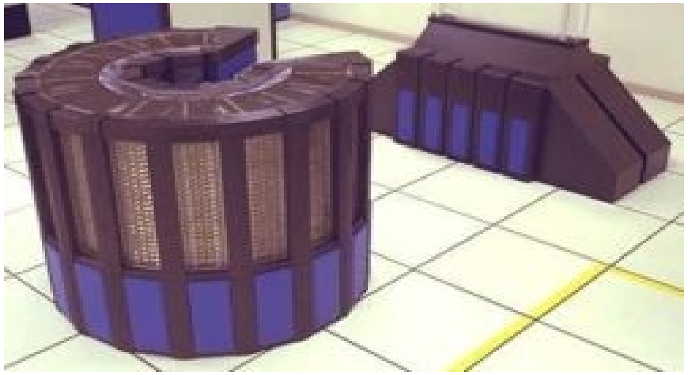
- Parallel computing was favored in the early years
- Primarily vector-based at first
- Gradually more thread-based parallelism was introduced

1985-95

- "Massively parallel architectures" start rising in prominence
- Message Passing Interface (MPI) and other libraries developed
- Bandwidth was a big problem

1995-today

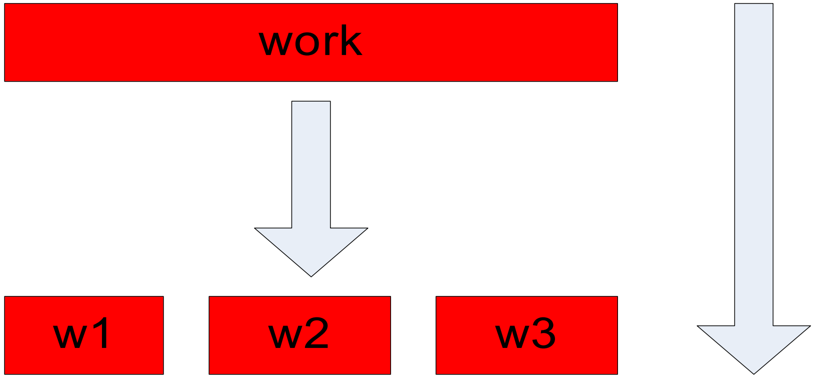
- Cluster/grid architecture increasingly dominant
- COTS technologies
- Web-wide cluster software
- Google = thousands of nodes/cluster



Cray 2 supercomputer

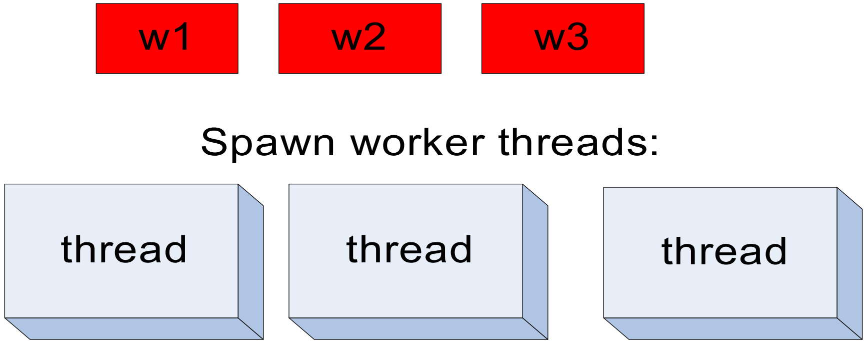
Data Parallelization Idea

Parallelization is “easy” if processing can be cleanly split into n units:



Partition
problem

Data Parallelization Idea (2)



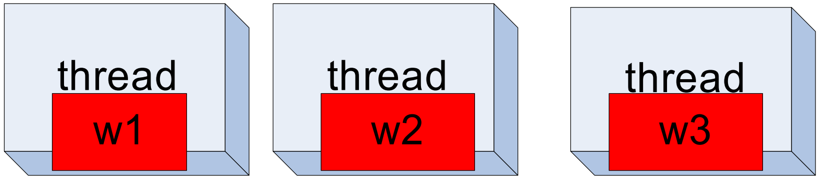
w1 w2 w3

Spawn worker threads:

thread thread thread

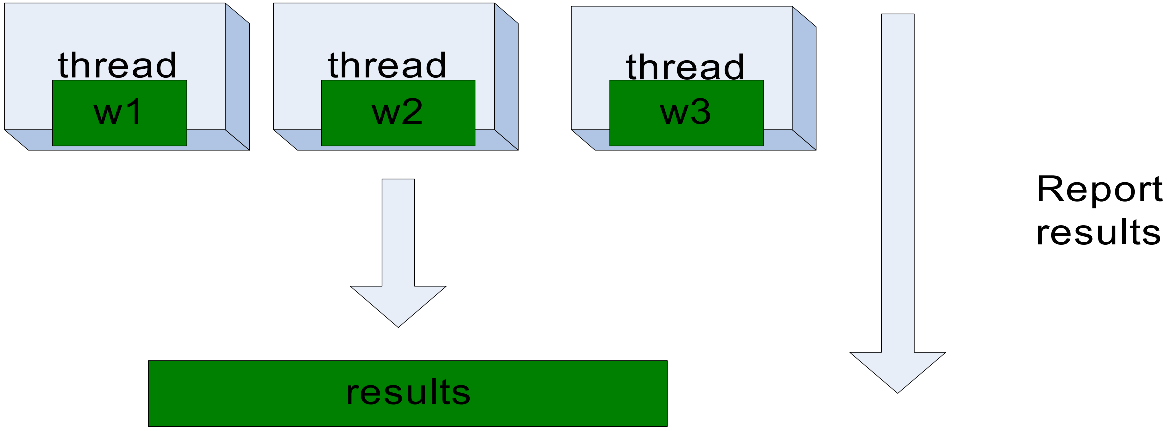
In a parallel computation, we would like to have as many threads as we have processors. e.g., a four-processor computer would be able to run four threads at the same time.

Workers process data:



thread w1 thread w2 thread w3

Parallelization Idea (3)



Embarassing parallel computation

Parallelization Pitfalls

But this model is too simple!

- How do we assign work units to worker threads?
- What if we have more work units than threads?
- How do we aggregate the results at the end?
- How do we know all the workers have finished?
- What if the work cannot be divided into completely separate tasks?

What is MapReduce?

- Simple data-parallel programming model designed by Google ☺ for scalability and fault-tolerance to make a **subset** (albeit a large one) of distributed problems easier to code
- Automates data distribution & result aggregation
- Restricts the ways data can interact to eliminate locks (no shared state = no locks!)
- Pioneered by Google
  - Processes 20 petabytes of data per day
- Popularized by open-source Hadoop project
  - Used at Yahoo!, Facebook, Amazon, …

MapReduce Design Goals

Scalability to large data volumes:

1000’s of machines, 10,000’s of disks

Cost-efficiency:

Commodity machines (cheap, but unreliable)

Commodity network

Automatic fault-tolerance (fewer administrators)

Easy to use (fewer programmers)

Challenges

Cheap nodes fail, especially if you have many

Mean time between failures for 1 node = 3 years

Mean time between failures for 1000 nodes = 1 day

Solution: Build fault-tolerance into system

Commodity network = low bandwidth

Solution: Push computation to the data

Programming distributed systems is hard

Solution: Data-parallel programming model: users write "map" & "reduce" functions, system distributes work and handles faults

MapReduce Programming Model

Data type: key-value *records*

Map function:

$$(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})$$

Reduce function:

$$(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})$$

All those with the same key, K_{inter} in the example

Example: Word Count

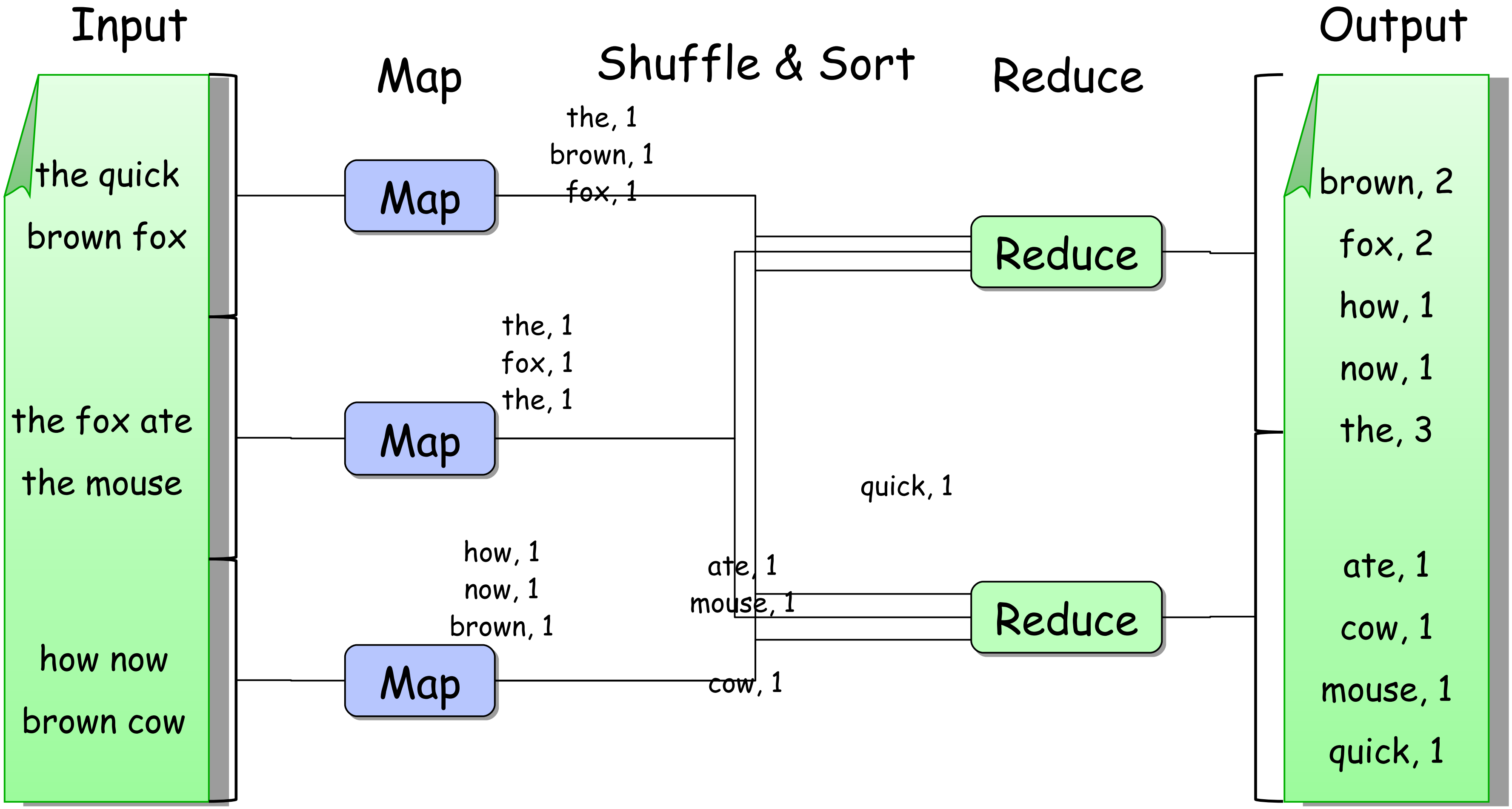
```
def mapper(line):
    foreach word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```

value

key

Each word in the document is a key

Word Count Execution



An Optimization: The Combiner

A combiner is a local aggregation function for repeated keys produced by same map

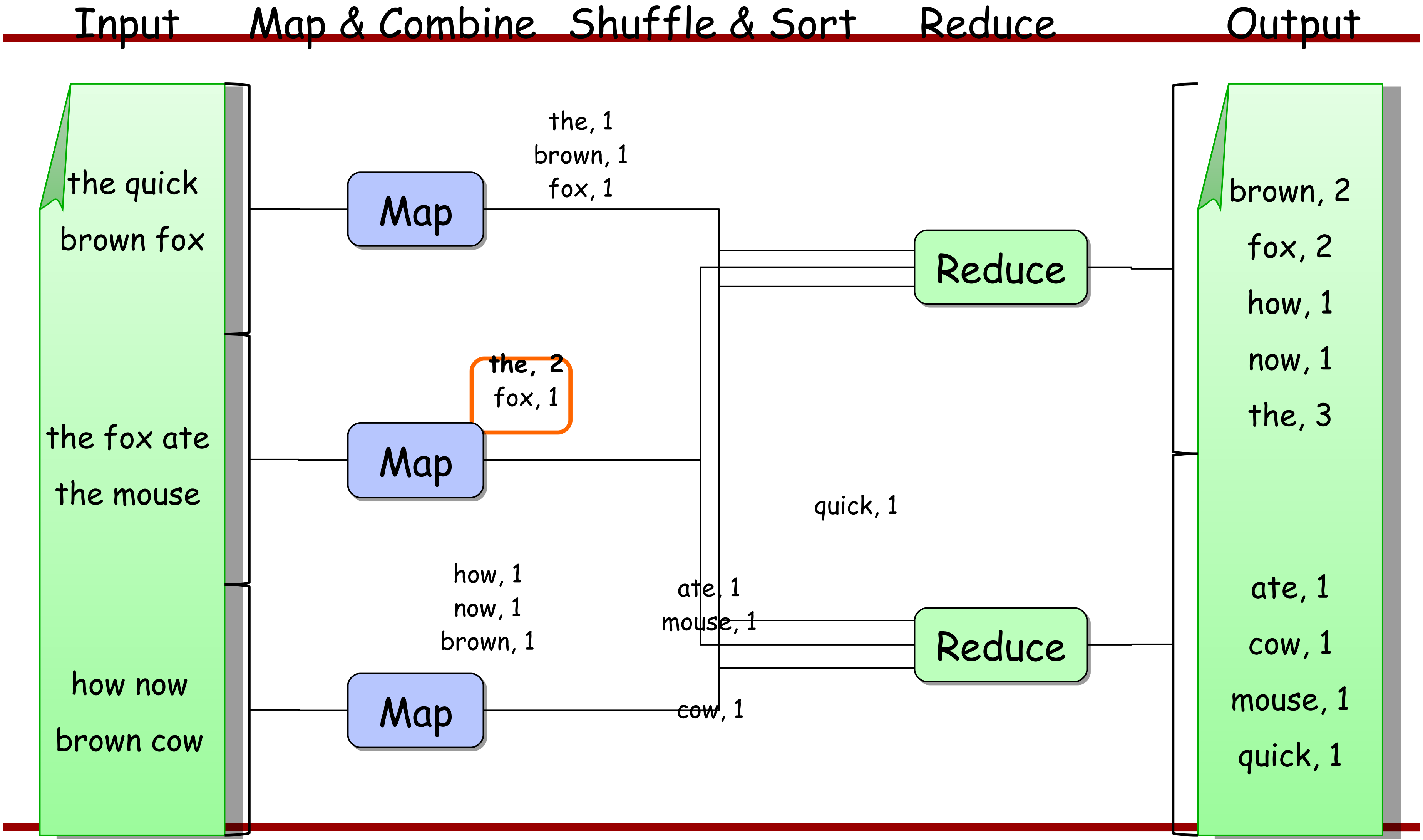
Works for associative functions like sum, count, max

Decreases size of intermediate data

Example: map-side aggregation for Word Count:

```
def combiner(key, values):
    output(key, sum(values))
```

Word Count with Combiner



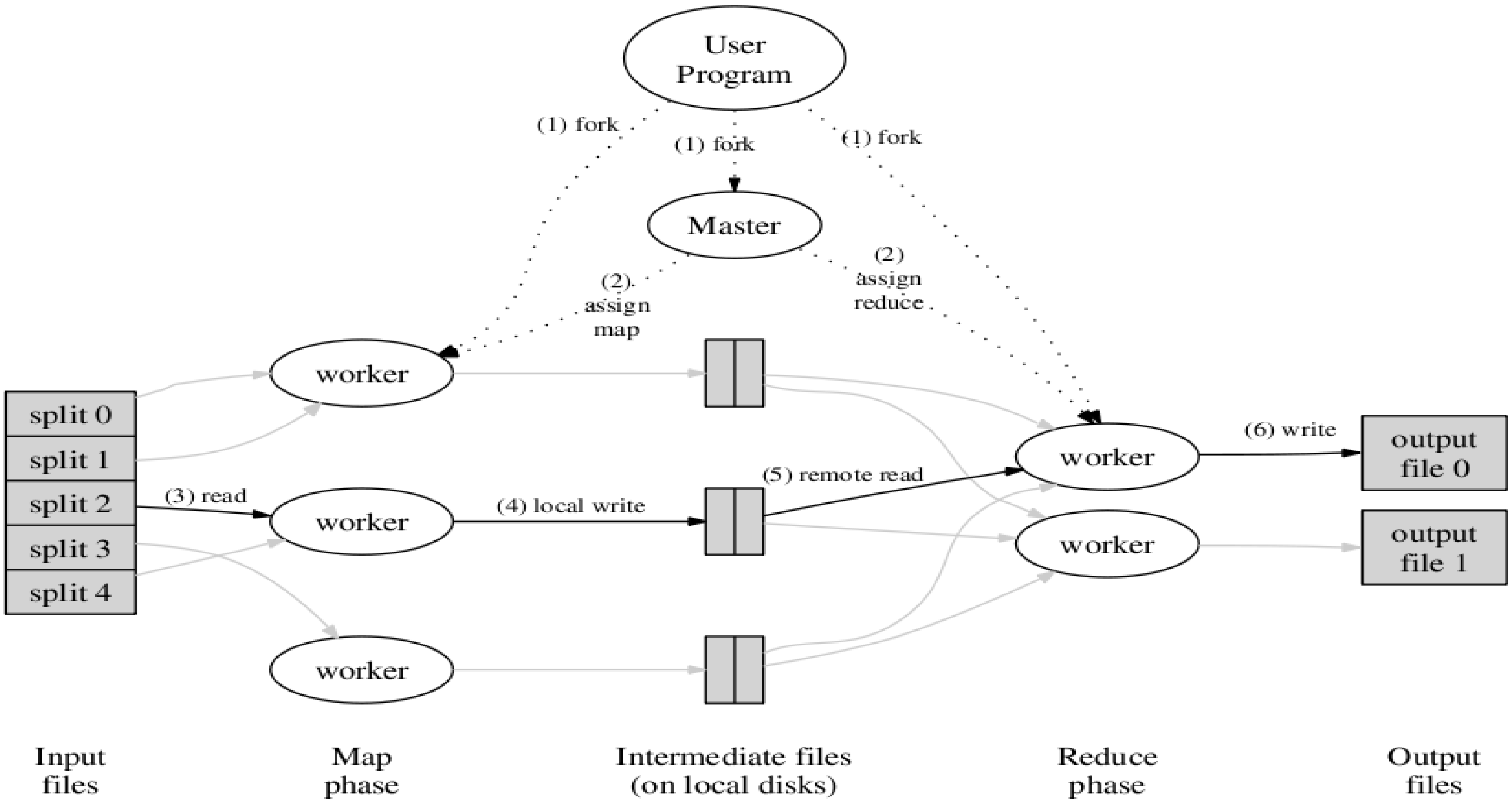
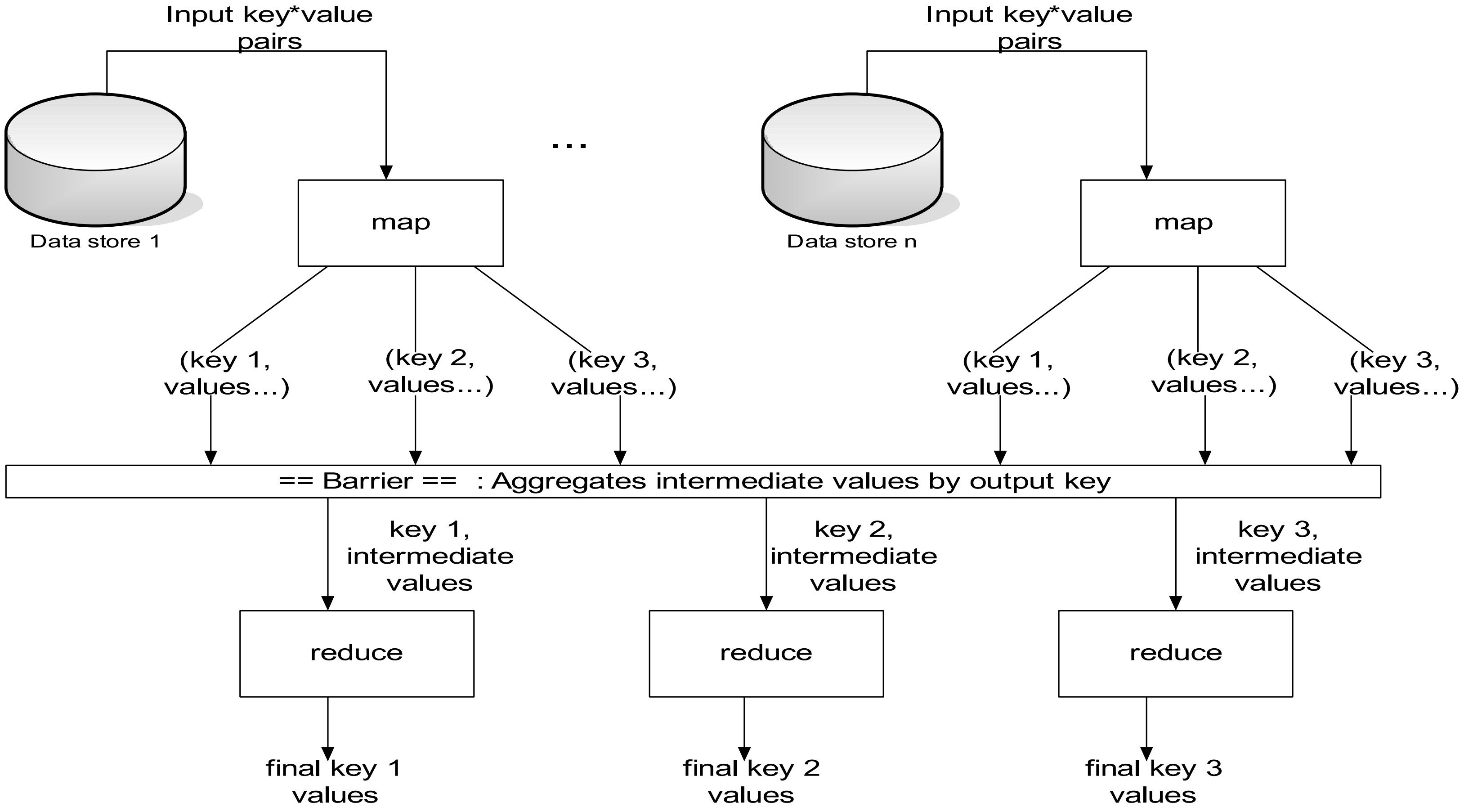


Figure 1: Execution overview

Barrier


Input key*value pairs
Input key*value pairs
Data store 1
map
...
Data store n
map
(key 1, values...)
(key 2, values...)
(key 3, values...)
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
key 1, intermediate values
key 2, intermediate values
key 3, intermediate values
reduce
reduce
reduce
final key 1 values
final key 2 values
final key 3 values

MapReduce Implementation Details

Single *master* controls job execution on multiple *slaves*

Mappers preferentially placed on same node or same rack as their input block

Minimizes network usage

Mappers save outputs to local disk before serving them to reducers

supports recovery if a reducer crashes

enables having more reducers than nodes

Fault Tolerance in MapReduce

1. If a task crashes:
    - Retry on another node
        - OK for a map because it has no dependencies
        - OK for reduce because map outputs are on disk
    - If the same task fails repeatedly, fail the job or ignore that input block (user-controlled)
2. If a node crashes:
    - Re-launch its current tasks on other nodes
    - Re-run any maps the node previously ran
        - Necessary because their output files were lost along with the crashed node

- Note: For these fault tolerance features to work, *map and reduce tasks must be side-effect-free*

Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):

    Launch second copy of task on another node ("speculative execution")

    Take the output of whichever copy finishes first, and kill the other

Surprisingly important in large clusters

    Stragglers occur frequently due to failing hardware, software bugs, misconfiguration, etc

    Single straggler may noticeably slow down a job

“Fault” Tolerance

- Nodes fail
  - Re-run tasks
- Nodes are slow (stragglers)
  - Run backup tasks (speculative execution)
  - To minimize job’s response time
    - Important for short jobs

Speculative execution

The scheduler schedules backup executions of the remaining *in-progress tasks*

The task is marked as completed whenever either the primary or the backup execution completes

Improve job response time by 44% according Google’s experiments

Seems a simple problem, but

Resource for speculative tasks is not free

- How to choose nodes to run speculative tasks?

- How to distinguish “stragglers” from nodes that are slightly slower?

Stragglers should be found out early

Speculative execution vs Monitoring

- How to choose nodes to run speculative tasks?

- How to distinguish "stragglers" from nodes that are slightly slower?

Stragglers should be found out early

All these problems require the adoption of a proper execution monitoring

An execution monitoring is a tool to analyze actual resource usage to discover problems in program execution

Hadoop's scheduler

Start speculative tasks based on a simple heuristic

- Comparing each task's progress to the average

Assumption of homogeneous environment

- The default scheduler works well
- Broken in utility computing

How to robustly perform speculative execution (backup tasks) in heterogeneous environments?

Takeaways

By providing a data-parallel programming model, MapReduce can control job execution in useful ways:

- Automatic division of job into tasks
- Automatic placement of computation near data
- Automatic load balancing
- Recovery from failures & stragglers

User focuses on application, not on complexities of distributed computing

1. Search

Input: (lineNumber, line) records

Output: lines matching a given pattern

Map:

```
if(line matches pattern):
    output(line)
```

Reduce: identity function

Alternative: no reducer (map-only job)

2. Sort

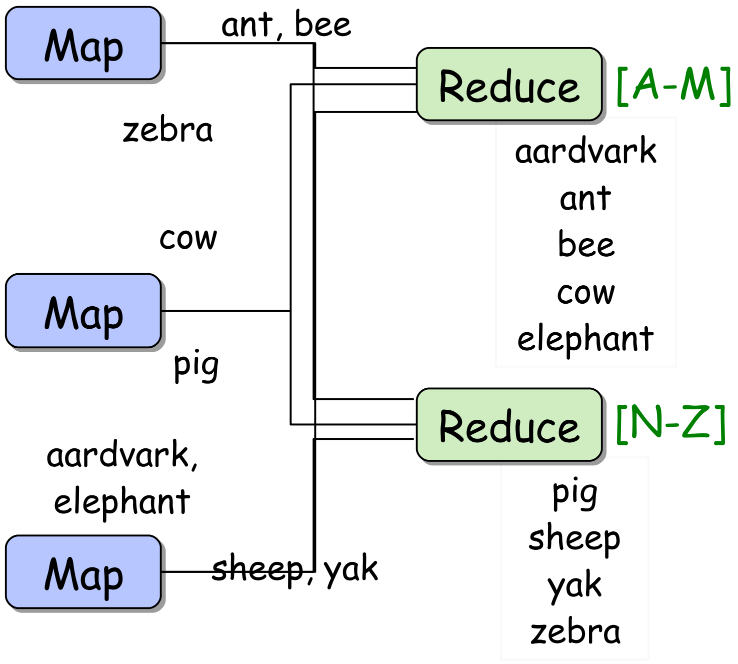
Input: (key, value) records

Output: same records, sorted by key

Map: identity function

Reduce: identity function

Trick: Pick partitioning function h such that $k_1<k_2 \Rightarrow h(k_1)<h(k_2)$



3. Inverted Index

Input: (filename, text) records

Output: list of files containing each word

Map:

```
foreach word in text.split():
   output(word, filename)
```

Combine: uniquify filenames for each word

Optimization to minimize communication

Reduce:

```
def reduce(word, filenames):
    output(word, sort(filenames))
```

4. Most Popular Words

Input: (filename, text) records

Output: top 100 words occurring in the most files

Two-stage solution:

- **Job 1:**
  - Create inverted index, giving (word, list(file)) records
- **Job 2:**
  - Map each (word, list(file)) to (count, word)
  - Sort these records by count as in sort job

Optimizations:

- Map to (word, 1) instead of (word, file) in Job 1
- Count files in job 1's reducer rather than job 2's mapper
- Estimate count distribution in advance and drop rare words

5. Numerical Integration

Input: (start, end) records for sub-ranges to integrate

Easy using custom InputFormat

Output: integral of f(x) dx over entire range

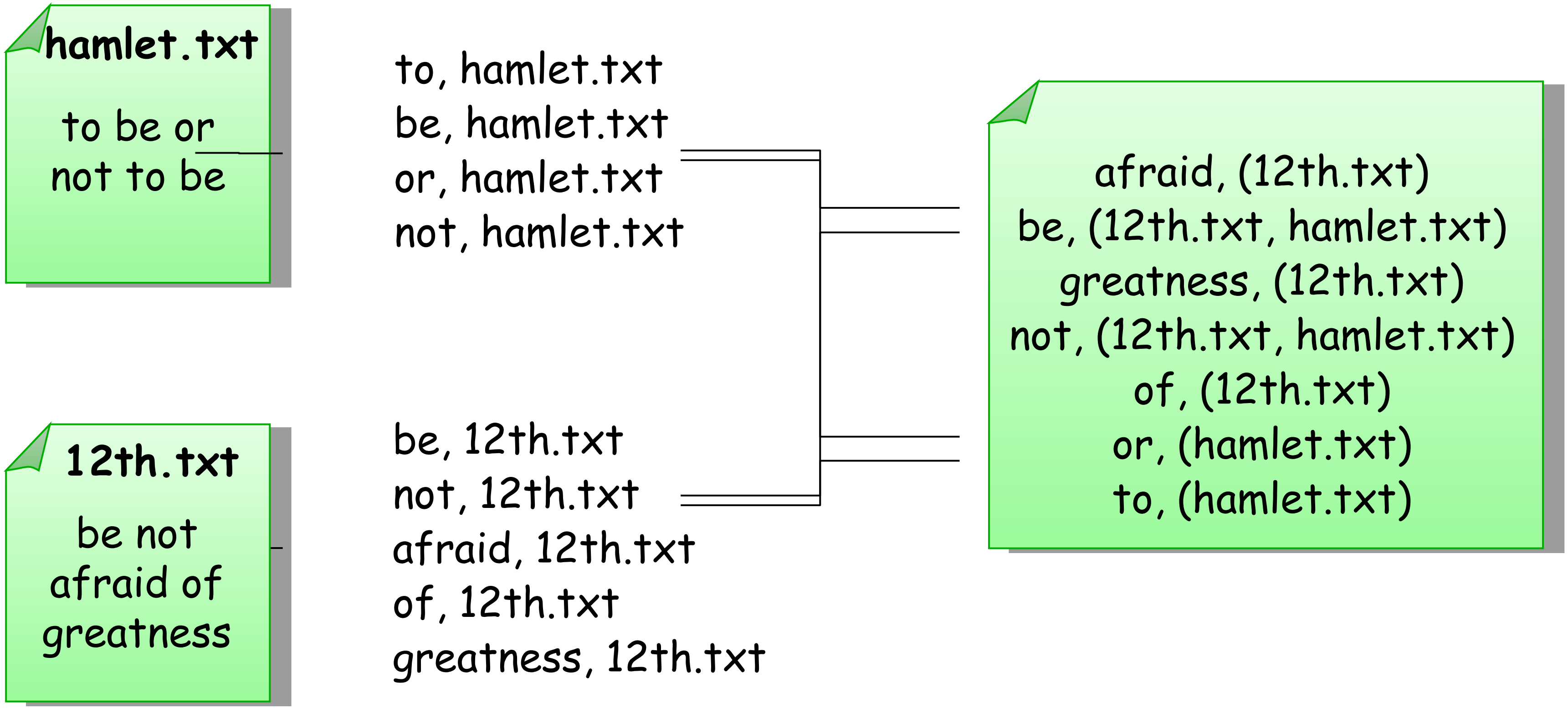
Map:

```
def map(start, end):
   sum = 0
   for(x = start; x < end; x += step):
      sum += f(x) * step
   output(“”, sum)
```

Reduce:

```
def reduce(key, values):
    output(key, sum(values))
```

Inverted Index Example



hamlet.txt

to be or
not to be

12th.txt

be not
afraid of
greatness

to, hamlet.txt
be, hamlet.txt
or, hamlet.txt
not, hamlet.txt

be, 12th.txt
not, 12th.txt
afraid, 12th.txt
of, 12th.txt
greatness, 12th.txt

afraid, (12th.txt)
be, (12th.txt, hamlet.txt)
greatness, (12th.txt)
not, (12th.txt, hamlet.txt)
of, (12th.txt)
or, (hamlet.txt)
to, (hamlet.txt)

What is MapReduce used for?

At Google:

- Index construction for Google Search
- Article clustering for Google News
- Statistical machine translation

At Yahoo!:

- “Web map” powering Yahoo! Search
- Spam detection for Yahoo! Mail

At Facebook:

- Data mining
- Ad optimization
- Spam detection

What is MapReduce used for?

In research:

- Astronomical image analysis (Washington)
- Bioinformatics (Maryland)
- Analyzing Wikipedia conflicts (PARC)
- Natural language processing (CMU)
- Particle physics (Nebraska)
- Ocean climate simulation (Washington

NFS Disadvantages for MapReduce

- Network congestion
- Heavy disk activity of the NFS server adversely affects the NFS’s performance.
- When the client attempts to mount , the client system hangs, although this can be mitigated using a specific mount.
- If the server hosting the exported file system becomes unavailable due to any reason, no one can access the resource.
- NFS has security problems because its design assumes a trusted network.
- Google FS developed as an alternative

Google FS Assumptions

- **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 typically
- **Files are write once ,mostly appended to**
  - Perhaps Concurrently
- **Large streaming reads**.

GFS Design Decisions

Files are stored as chunks.

- Fixed size(64 MB).

Reliability through replication.

- Each chunk is replicated across 3+ chunkservers

Single master to co ordinate access, keep metadata

- Simple centralized management.

No data caching

- Little benefit due to large datasets,streaming reads.

GFS Architecture

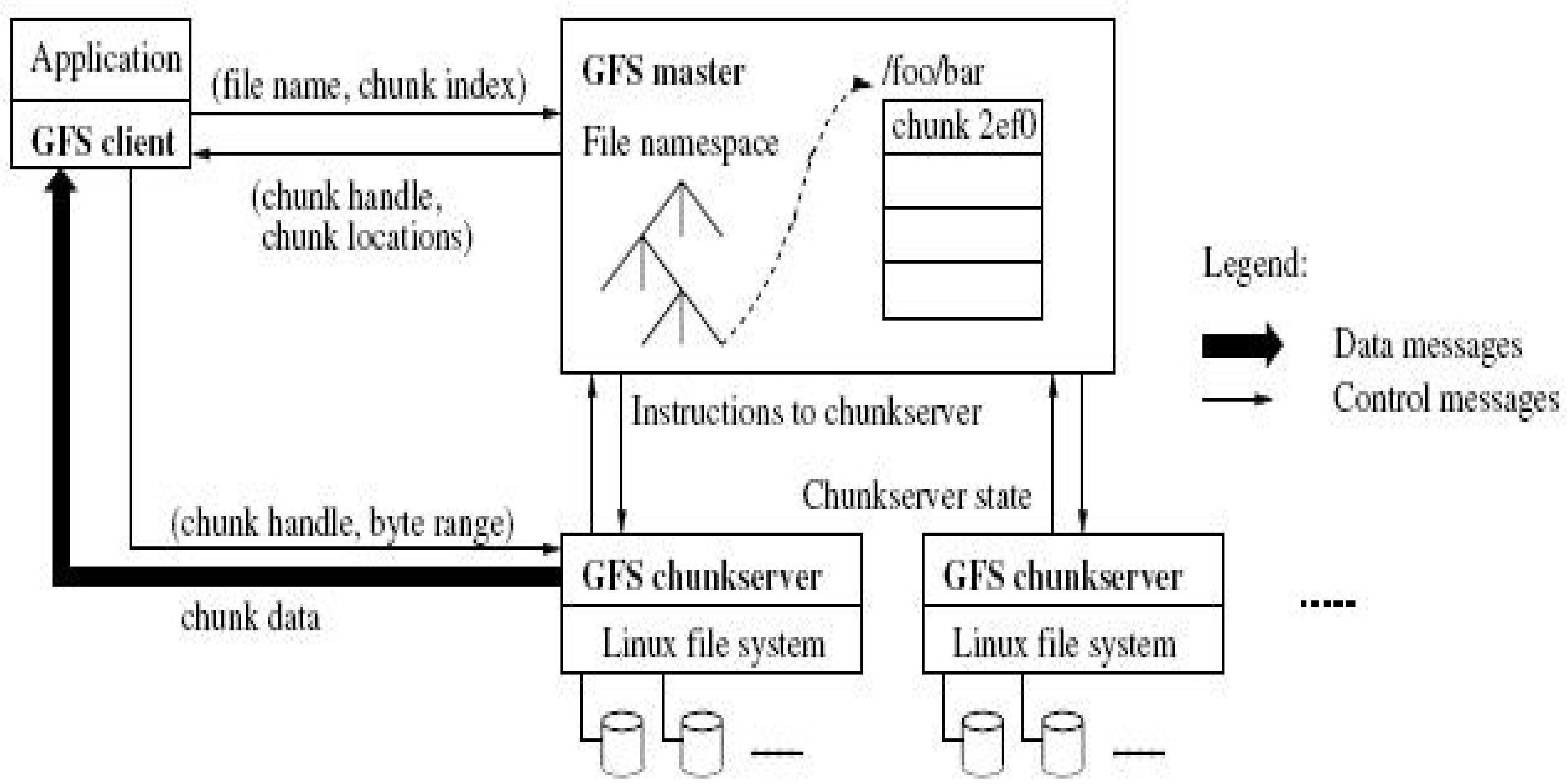


Figure 1: GFS Architecture

Single master

From distributed systems we know this is a:

- Single point of failure
- Scalability bottleneck

GFS solutions:

- Shadow masters
- Minimize master involvement
  a) never move data through it, use only for metadata and cache metadata at clients
  b) large chunk size
  c) master delegates authority to primary replicas in data mutations (chunk leases)

Simple, and good enough!

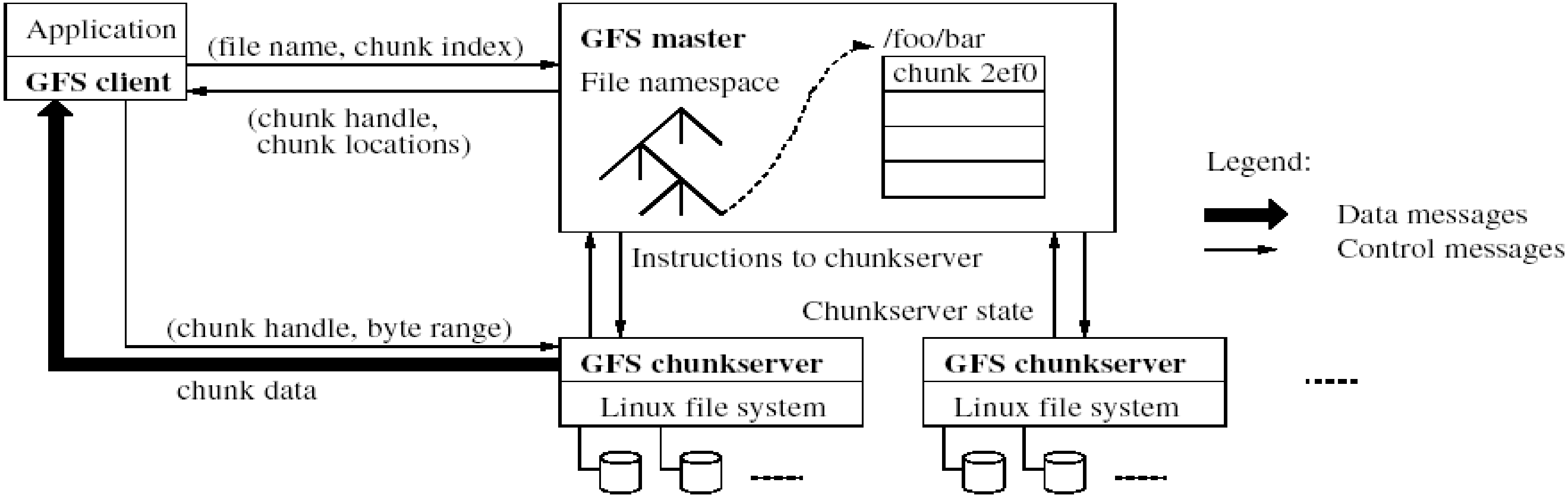
Metadata

- **Global metadata is stored on the master.**
  - File and chunk namespaces.
  - Mapping from files to chunks.
  - Locations of each chunk replicas.
- **All in memory (64bytes/chunk)**
  - Fast
  - Easily Accessible.
- **Master has an operation log for persistent logging of critical metadata updates.**
  - Persistent on local disk
  - Replicated
  - Check points for faster recovery.

GFS Architecture

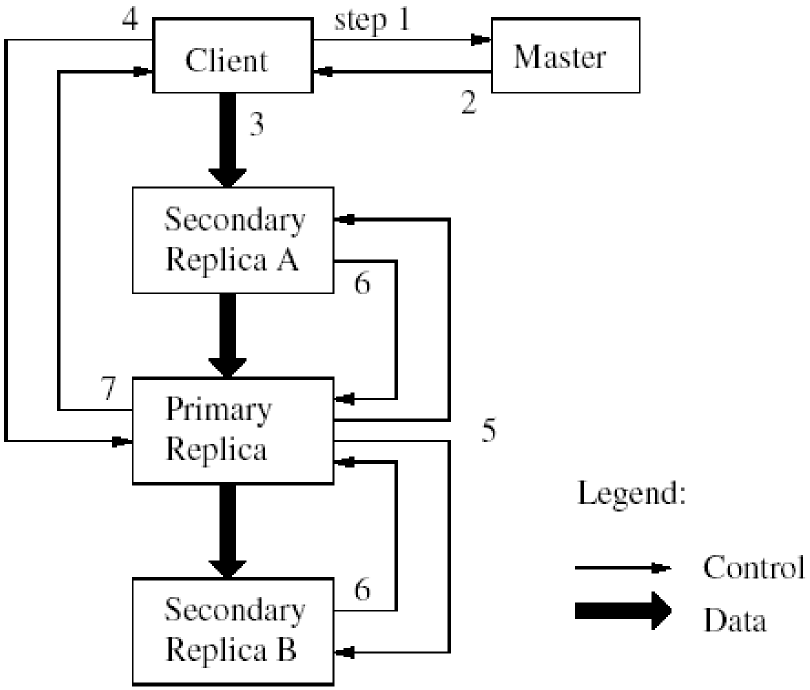
Single master

Mutiple chunkservers



Mutations

- Mutation = write or append
  - □ must be done for all replicas
- Goal: minimize master involvement
- Lease mechanism:
  - □ master picks one replica as primary; gives it a “lease” for mutations
  - □ primary defines a serial order of mutations
  - □ all replicas follow this order
- Data flow decoupled from control flow



Atomic record append

Client specifies data

GFS appends it to the file atomically at least once

- GFS picks the offset
- works for concurrent writers

Used heavily by Google apps

- e.g., for files that serve as multiple-producer/single-consumer queues

Relaxed consistency model - I

“Consistent” = all replicas have the same value

“Defined” = replica reflects the mutation, consistent

Some properties:

- concurrent writes leave region consistent, but possibly undefined
- failed writes leave the region inconsistent

Some work has moved into the applications:

- e.g., self-validating, self-identifying records

Simple, efficient

- Google apps can live with it
- what about other apps?

Namespace updates atomic and serializable

Relaxed consistency model - II

- Long after a successful mutation, component failures can of course still corrupt or destroy data.
- GFS identifies failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming
- Once a problem surfaces, the data is restored from valid replicas as soon as possible
- A chunk is lost irreversibly only if all its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable, not corrupted: applications receive clear errors rather than corrupt data.

Relaxed consistency model - III

- GFS applications accommodate the relaxed consistency model with a few simple techniques
  - relying on appends rather than overwrites,
  - checkpointing,
  - writing self-validating, self-identifying records.
- Practically all applications mutate files by appending rather than overwriting.
  - A writer generates a file from beginning to end and atomically renames the file to a permanent name after writing all the data, or periodically checkpoints how much has been successfully written. checkpoints may also include application-level checksums.
- Readers verify and process only the file region up to the last checkpoint, which is known to be in the defined state.

Relaxed consistency model - IV

- Google words
- Appending is far more efficient and more resilient to failures than random writes. Checkpointing allows writers to restart incrementally and keeps readers from processing successfully written file data that is still incomplete from the application’s perspective. In the other typical use, many writers concurrently append to a file for merged results or as a producer-consumer queue. Record append’s append-at-least-once semantics preserves each writer’s output

Master’s responsibilities

Metadata storage

Namespace management/locking

Periodic communication with chunkservers

- give instructions, collect state, track cluster health

Chunk creation, re-replication, rebalancing

- balance space utilization and access speed
- spread replicas across racks to reduce correlated failures
- re-replicate data if redundancy falls below threshold
- rebalance data to smooth out storage and request load

Garbage Collection

- simpler, more reliable than traditional file delete
- master logs the deletion, renames the file to a hidden name
- lazily garbage collects hidden files

Stale replica deletion

- detect “stale” replicas using chunk version numbers

Fault Tolerance

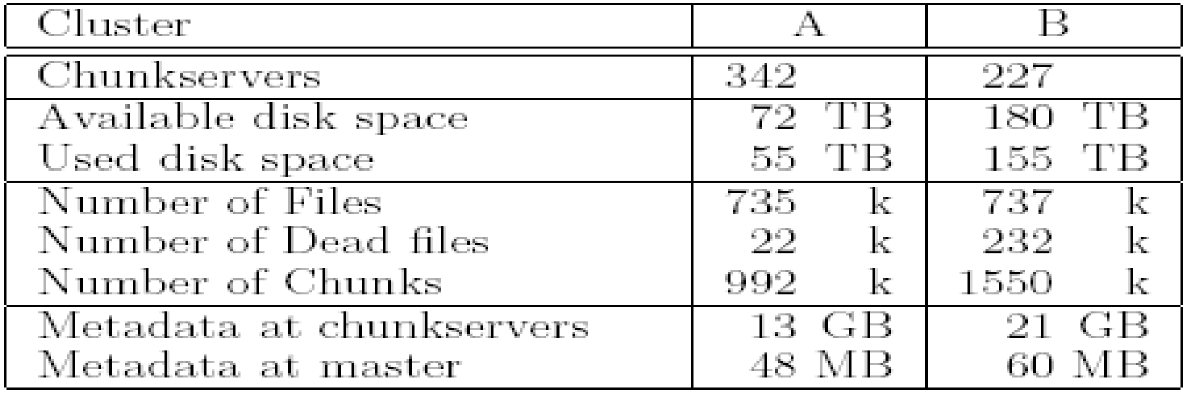
High availability

- fast recovery : master and chunkservers restartable in a few seconds
- chunk replication: default: 3 replicas.
- shadow masters

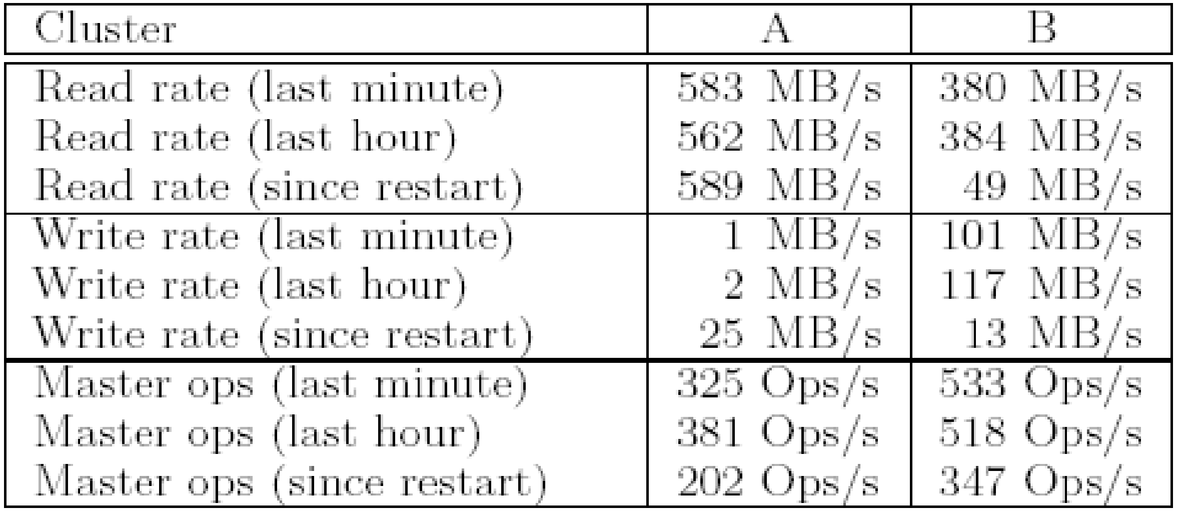
Data integrity

- checksum every 64KB block in each chunk

Performance



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



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

Conclusion

GFS demonstrates how to support large-scale processing workloads on commodity hardware

- design to tolerate frequent component failures
- optimize for huge files that are mostly appended and read
- feel free to relax and extend FS interface as required
- go for simple solutions (e.g., single master)

GFS has met Google's storage needs… it must be good!

What’s hadoop

- **Framework for running applications on large clusters of commodity hardware**
- **Scale: petabytes of data on thousands of nodes**
  - **Storage: Hadoop Distributed FS**
  - **Processing: MapReduce Requirements**
  - **Economy: use cluster of comodity computers**
  - **Easy to use**
    - **Users: no need to deal with the complexity of distributed computing**
  - **Reliable: can handle node failures automatically**

Hadoop: Motivation

- Previous discussion shows that several parallel algorithms can be expressed by a series of MapReduce jobs
- But MapReduce is fairly low-level: must think about keys, values, partitioning, etc
- Can we capture common "job building blocks"?
- Hadoop was inspired by
  - MapReduce
  - Google File System

Hadoop Components

Distributed file system (HDFS)

Single namespace for entire cluster

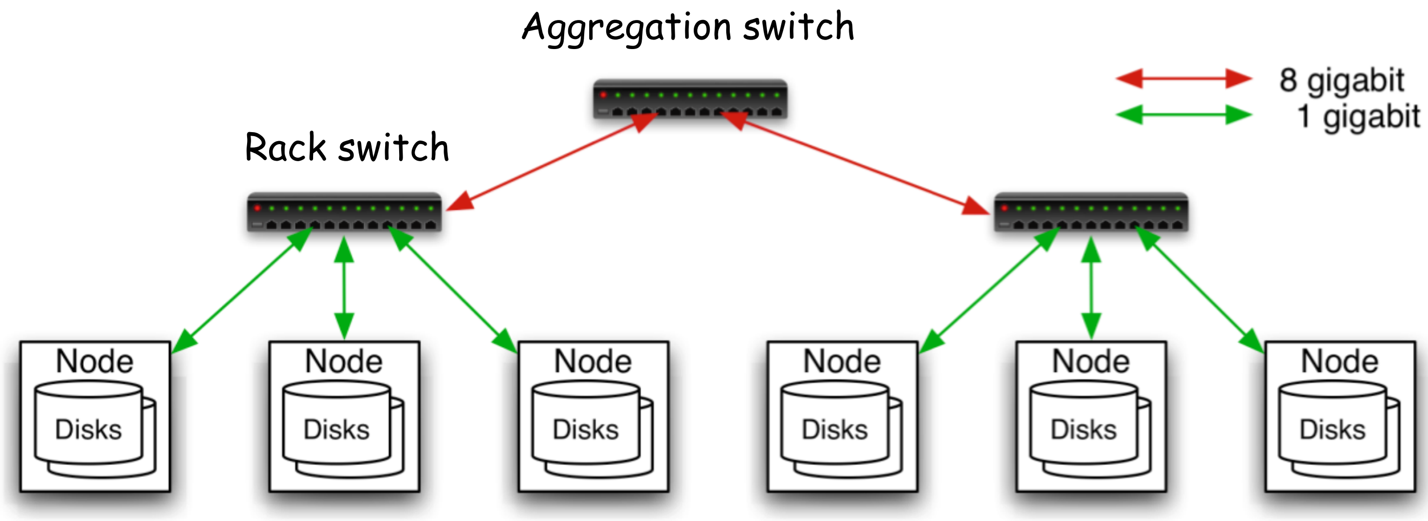
Replicates data 3x for fault-tolerance

MapReduce framework

Executes user jobs specified as “map” and “reduce” functions

Manages work distribution & fault-tolerance

Typical Hadoop Cluster



40 nodes/rack, 1000-4000 nodes in cluster

1 Gbps bandwidth within rack, 8 Gbps out of rack

Node specs (Yahoo terasort):
8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf

Hadoop Distributed File System

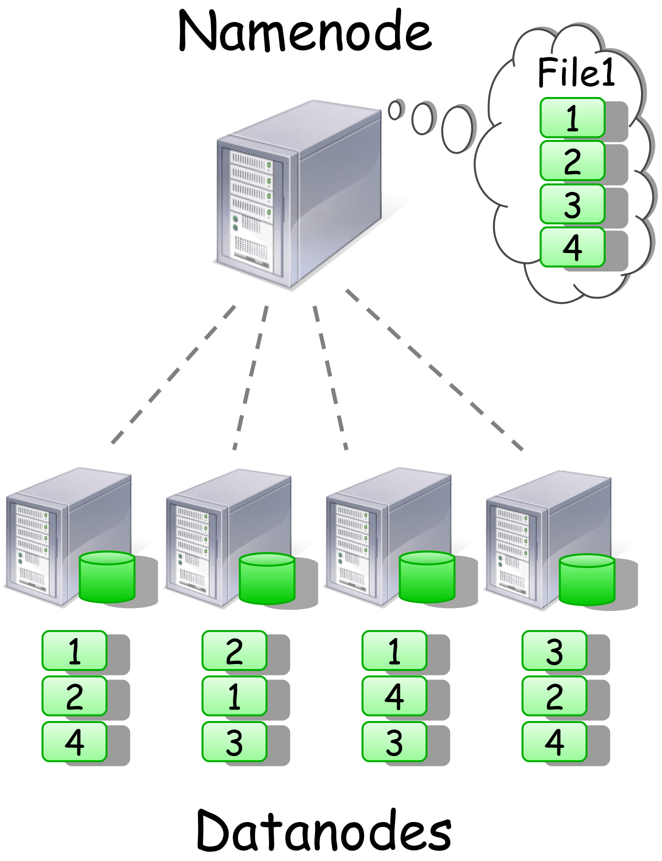
Files split into 128MB *blocks*

Blocks replicated across several *datanodes* (usually 3)

Single *namenode* stores metadata (file names, block locations, etc)

Optimized for large files, sequential reads

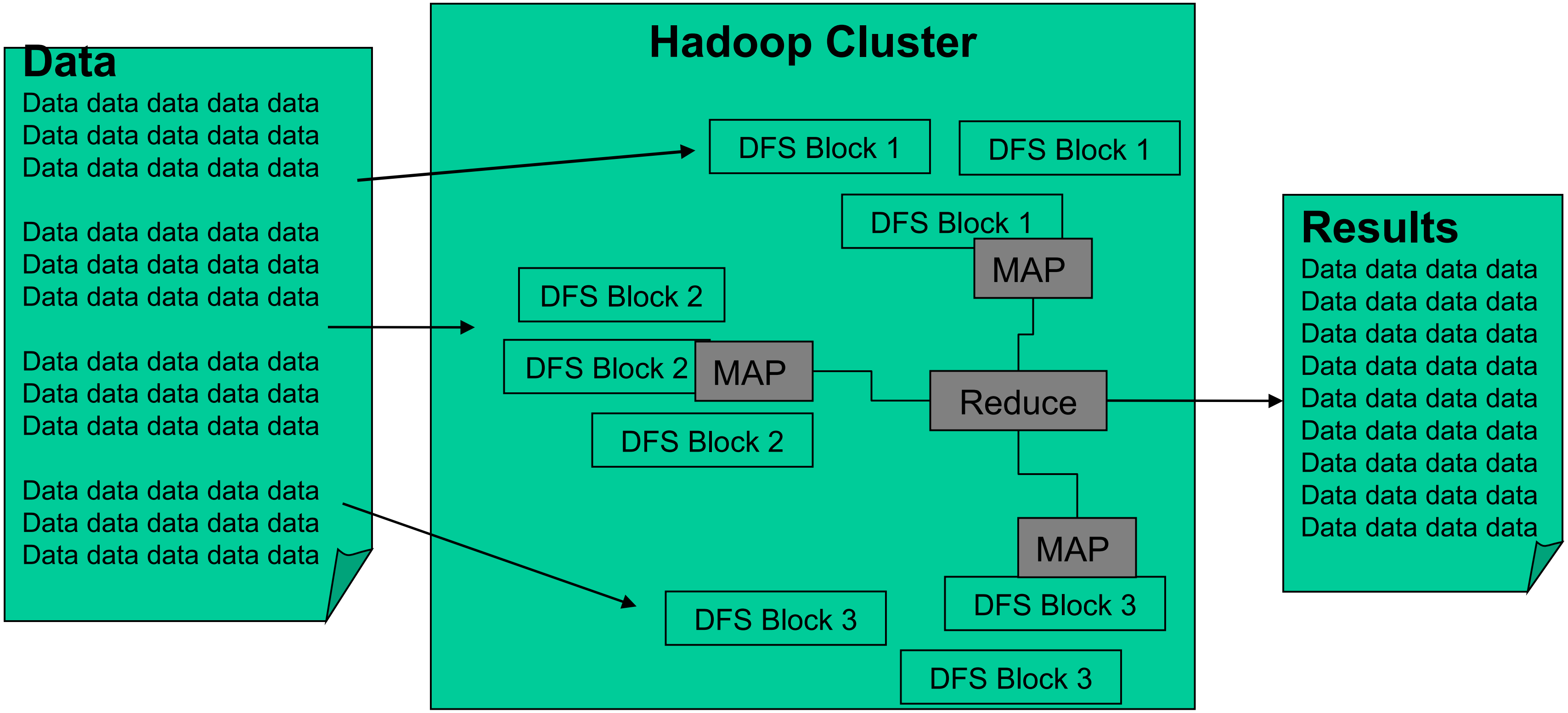
Files are append-only



HDFS

- Hadoop implements MapReduce using the Hadoop Distributed File System (HDFS)
- MapReduce divides applications into many small blocks of work = HDFS creates multiple replicas of data blocks for reliability, placing them on compute nodes around the cluster so that MapReduce can then process the data where it is located.
- Hadoop has been demonstrated on clusters with 2000 nodes. The current design target is 10,000 node clusters.

Hadoop Architecture



HDFS

- HDFS assumes that hardware is unreliable and will eventually fail.
- Similar to RAID level except HDFS can replicate data across several machines
- Provides Fault tolerance
- Extremely high capacity storage
- Moving Computation is cheaper than moving data= HDFS is said to be rack aware.

Hadoop Map-Reduce Architecture

Master-Slave architecture

Map-Reduce Master “Jobtracker”

- Accepts MR jobs submitted by users
- Assigns Map and Reduce tasks to Tasktrackers
- Monitors task and tasktracker status, re-executes tasks upon failure

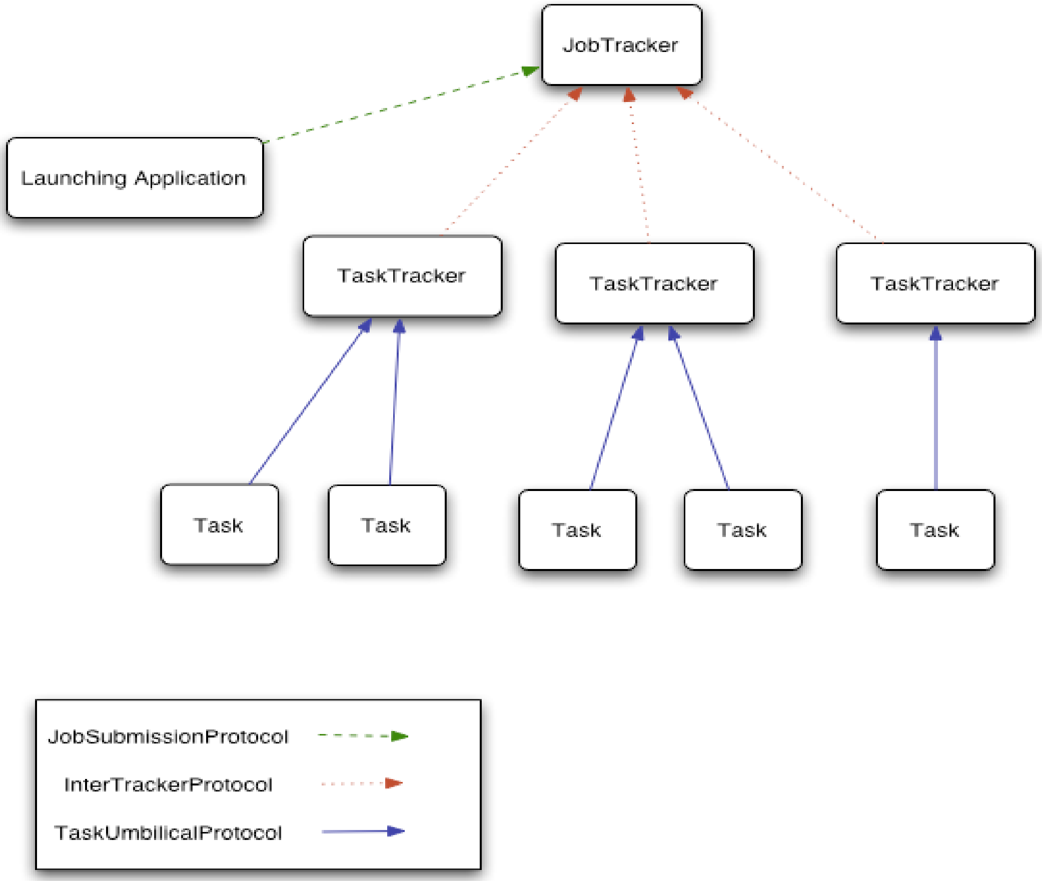
Map-Reduce Slaves “Tasktrackers”

- Run Map and Reduce tasks upon instruction from the Jobtracker
- Manage storage and transmission of intermediate output

Scheduling

- FIFO to schedule jobs from a work queue other schedulers share resources among users
- **Fair scheduler**
  - Provides fast response times for small jobs and QOS for production jobs.
  - Jobs are grouped into Pools, a default one for uncategorized jobs
  - Each pool is assigned a guaranteed minimum share.
  - Excess capacity is split between jobs.
  - Pools specify the minimum number of map slots, reduce slots, and a limit on the number of running jobs.
- **Capacity scheduler**
  - supports several features which are similar to the fair scheduler
  - Jobs are submitted into Queues
  - Queues are allocated a fraction of the total resource capacity
  - Free resources are allocated to queues beyond their total capacity
  - Within a queue high priority jobs will have access to the queue's resources
  - There is no preemption once a job is running.

Process Diagram



Example Data Analysis Task

Find users who tend to visit “good” pages.

Visits

user	url	time
Amy	www.cnn.com	8:00
Amy	www.crap.com	8:05
Amy	www.myblog.com	10:00
Amy	www.flickr.com	10:05
Fred	cnn.com/index.htm	12:00

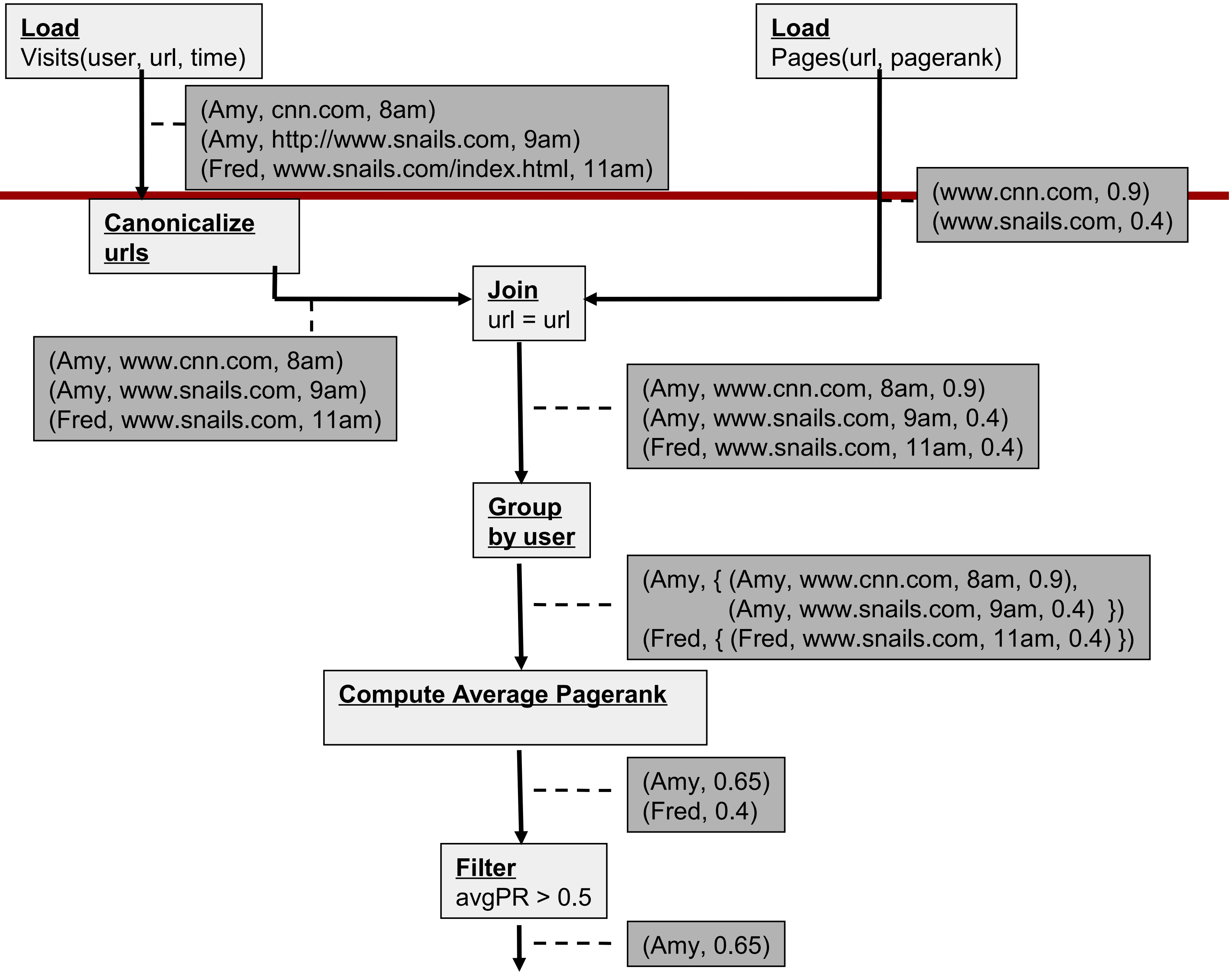
⋮

Pages

url	pagerank
www.cnn.com	0.9
www.flickr.com	0.9
www.myblog.com	0.7
www.crap.com	0.2

⋮

Conceptual Dataflow



Load
Visits(user, url, time)

(Amy, cnn.com, 8am)
(Amy, http://www.snails.com, 9am)
(Fred, www.snails.com/index.html, 11am)

Load
Pages(url, pagerank)

(www.cnn.com, 0.9)
(www.snails.com, 0.4)

Canonicalize urls

(Amy, www.cnn.com, 8am)
(Amy, www.snails.com, 9am)
(Fred, www.snails.com, 11am)

Join
url = url

(Amy, www.cnn.com, 8am, 0.9)
(Amy, www.snails.com, 9am, 0.4)
(Fred, www.snails.com, 11am, 0.4)

Group by user

(Amy, { (Amy, www.cnn.com, 8am, 0.9),
(Amy, www.snails.com, 9am, 0.4) })
(Fred, { (Fred, www.snails.com, 11am, 0.4) })

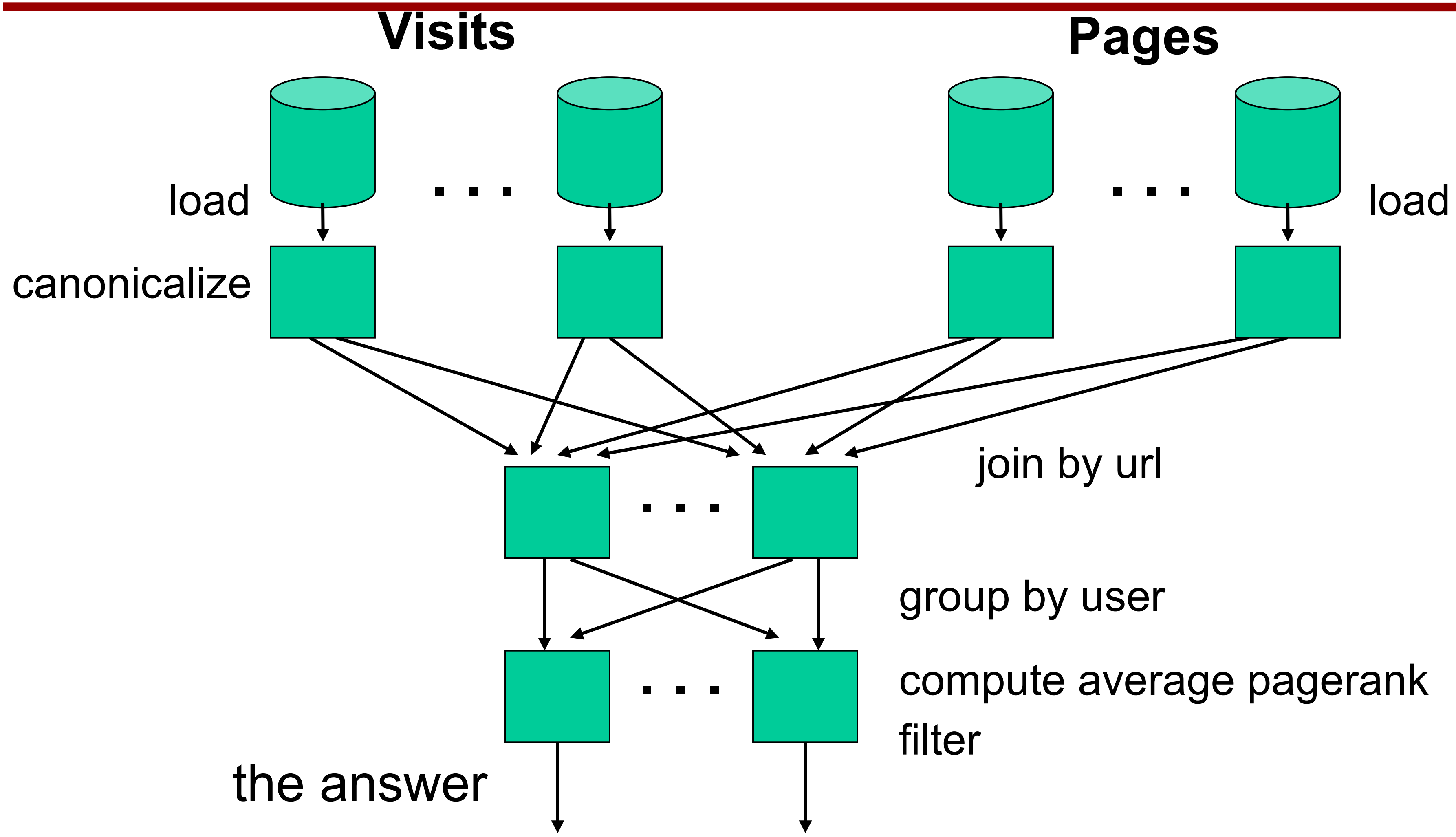
Compute Average Pagerank

(Amy, 0.65)
(Fred, 0.4)

Filter
avgPR > 0.5

(Amy, 0.65)

System-Level Dataflow



In General

Users' data processing tasks:

- K steps, N inputs, M outputs
- Mix of standard operations (e.g., filter, join) & custom operations (e.g., sentence segmentation)

Map-Reduce programming model:

- 2 steps, 1 input, 1 output
- Users chain together Map-Reduce jobs by hand
- Users hack to get multiple inputs/outputs
- Users code standard operations, e.g. join, by hand

Needed: dataflow programming model on top of Map-Reduce, e.g., *Pig Latin to avoid doing everything by hand*

Pig Latin Program

(textual representation of conceptual dataflow)

```
       Visits = load    ‘/data/visits’ as (user, url, time);
       Visits = foreach Visits generate user, Canonicalize(url), time;

        Pages = load    ‘/data/pages’ as (url, pagerank);

           VP = join    Visits by url, Pages by url;
   UserVisits = group   VP by user;
UserPageranks = foreach UserVisits generate user, AVG(VP.pagerank) as avgpr;
    GoodUsers = filter  UserPageranks by avgpr > ‘0.5’;

                store   GoodUsers into '/data/good_users';
```

Pig Takes Care of …

- Schema & type checking
- Translating into efficient physical dataflow
  (sequence of one or more Map-Reduce jobs)
- Exploiting data reduction opportunities
  (e.g., early partial aggregation via a “combiner”)
- Executing the physical dataflow (M-R jobs)
- Tracking progress, errors, etc.

Pig Latin ≠ SQL

A dataflow language, not a constraint language

- User specifies order of operations
- Does not rely on a query optimizer

Custom code is a first-class citizen

- Can stream records through any user-supplied executable, as part of dataflow

Users retain control of their data

- Operates directly over user files (can be any format)
- User supplies file format & schema at runtime

Ways to Run Pig

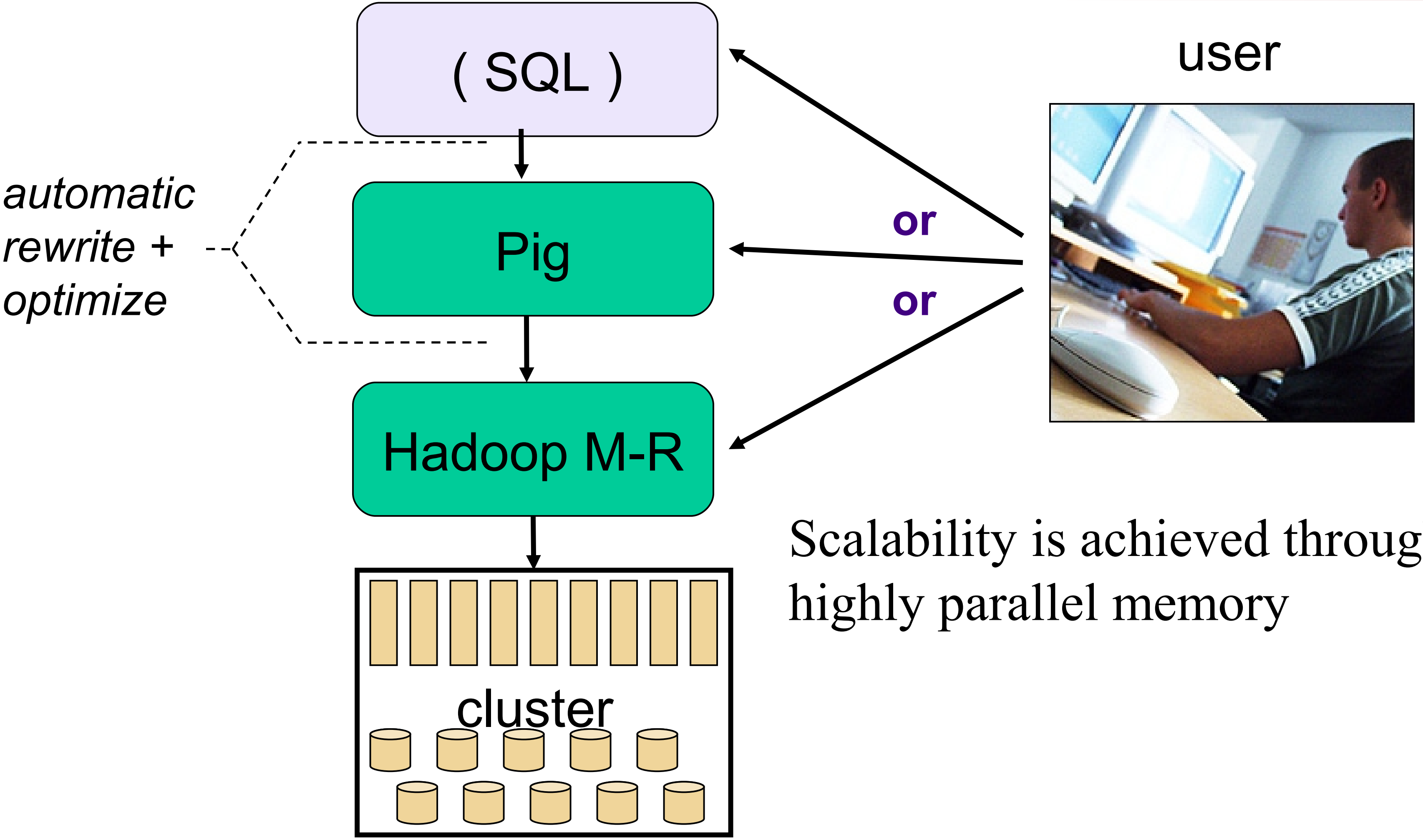
Interactive shell

Script file

Embed in host language (e.g., Java)

soon: Graphical editor

The Big Picture



Scalability is achieved through a highly parallel memory