GOOGLE STACK AND HADOOP

Master Program in Computer Science
University of Calabria

Prof. F. Ricca

Google: the pioneer of big data

- Google was first created in 1996
  - The World Wide Web was already huge
  - Very large scale -> *PageRank become a success*
  - PageRank ranks pages depending on the number of links
- Google Modular Data Center
  - A hardware infrastructure capable of unbounded growth
  - Very large numbers of **commodity servers**
- The Google Software Stack
  - **No operating system or database platform could** operate a huge number of servers
  - Developed a in-house solution to massively parallelize and distribute processing

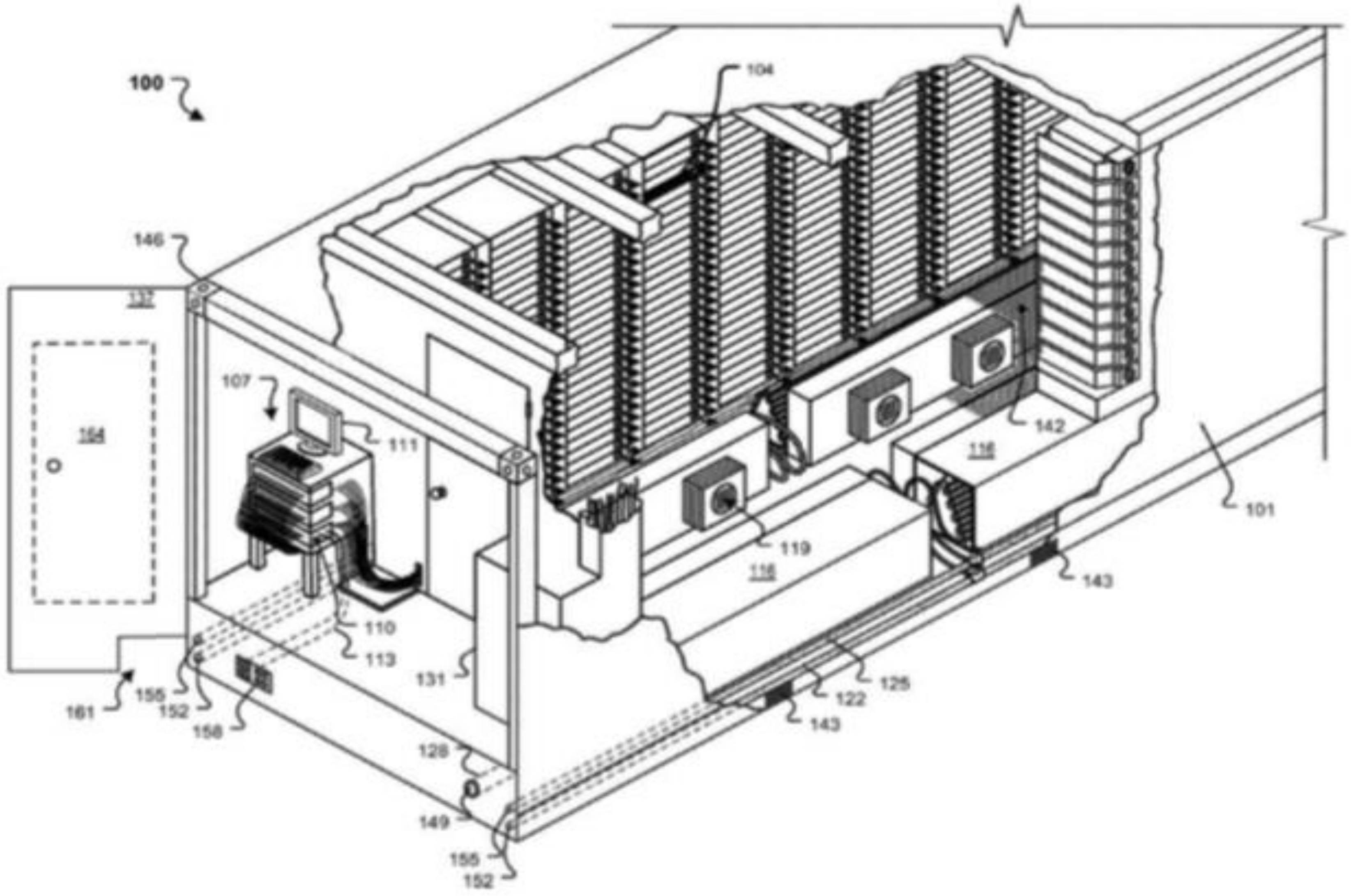


Figure 2-2. *Google Modular Data Center as described in their patent*

Data center capacity is increased by adding new 1,000-server modules!

The Google Software Stack

- Google developed three major software layers:
  - **Google File System (GFS)**
    - a distributed cluster file system that allows all of the disks accessed as one massive, distributed, redundant file system.
    - http://research.google.com/arcHive/gfs.html
  - **MapReduce**
    - a distributed processing framework for parallelizing algorithms across large numbers of potentially unreliable servers
    - http://research.google.com/arcHive/mapreduce.html
  - **BigTable**
    - a nonrelational database system that uses the Google File System for storage
    - http://research.google.com/arcHive/bigtable.html

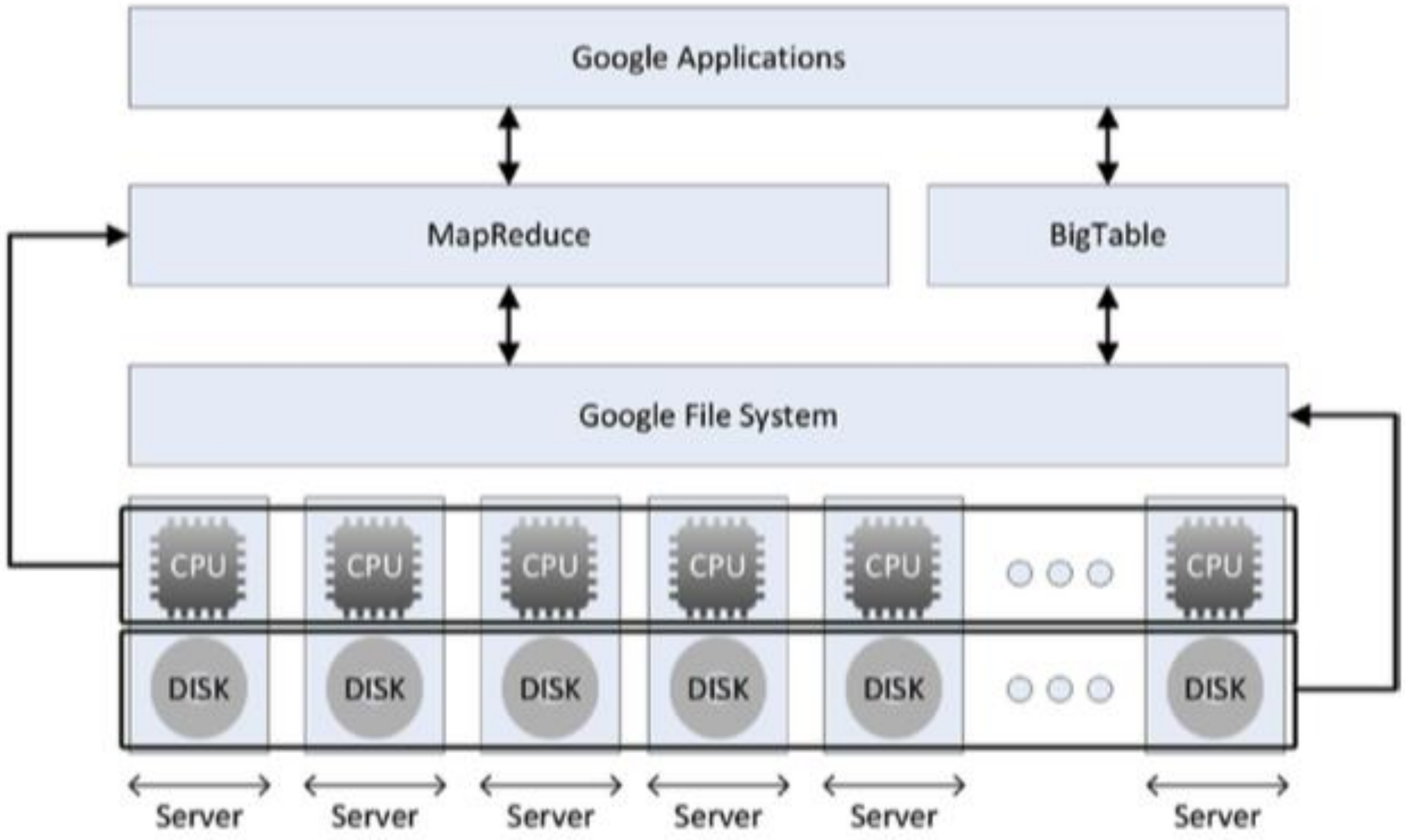


Figure 2-3. *Google software architecture*

The Google File System (GFS)

- A scalable distributed file system for large distributed data-intensive applications
- Same goals as previous distributed file systems
    - performance, scalability, reliability, and availability
- Design driven by Google's systems features
    - Component failures are the norm rather than the exception
    - Constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system
- Key Goals:
    - Fault tolerance (on inexpensive commodity hardware)
    - High sustained bandwidth is more important than low latency
    - Atomicity with minimal synchronization overhead is essential
    - High Availability

GFS Design assumptions

- Many inexpensive commodity components that often fail
  - recover promptly from component failures on a routine basis
- A modest number of large files
  - Multi-GB files should be managed efficiently
- Large streaming reads and small random reads
- Many large, sequential writes that append data to files
  - Once written, files are seldom modified again
- The system must efficiently implement concurrent append
  - Atomicity with minimal synchronization overhead is essential
- High sustained bandwidth is more than low latency
  - Processing data in bulk at a high rate

GFS Cluster (1)

- A GFS cluster consists of
  - a single *master* and
  - multiple *chunkservers* and is
  - accessed by multiple *clients*
- Files are
  - divided into fixed-size chunks
  - identified by an immutable and globally unique 64 bit handle

GFS Cluster (2)

- The master
  - Maintains all file system metadata, in main memory + log for recovery (can be replicated, only one working)
  - Periodically communicates with each chunkserver (HeartBeat messages)
- Chunkservers
  - Store chunks on local disks, has the final word over what it stores
  - Each chunk is replicated on multiple chunkservers on different racks
  - Checksumming to detect corruption of stored data
- Client code
  - Implements the file system API
  - Communicates with the master and chunkservers
  - Atomic append operation called record append.

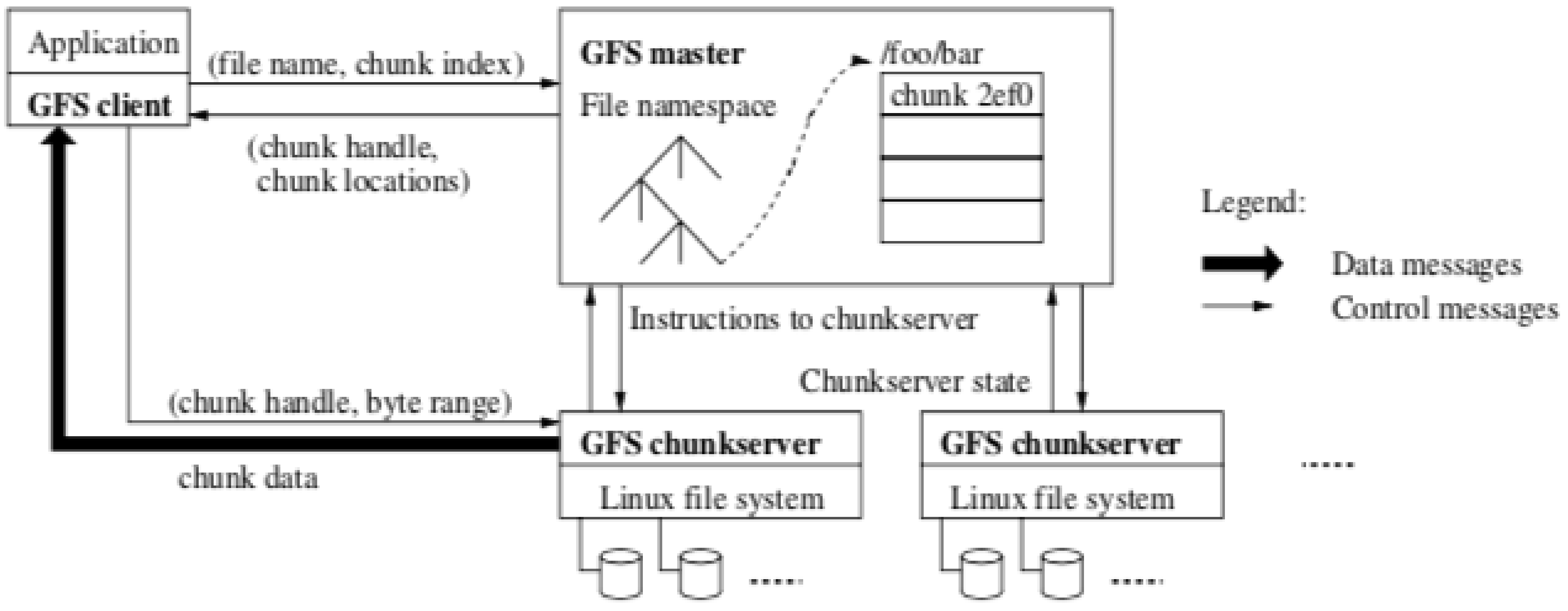


Figure 1: GFS Architecture

- Clients never read and write file data through the master
- A client asks the master which chunkservers it should contact
  - Caches this information for a limited time
  - Interacts with the chunkservers directly for many subsequent operations
  - Applications should mutate files by appending rather than overwriting
- Master controls chunkservers' state and instructs operations

Map Reduce paradigm

- A programming model
  - for parallelization of data-intensive processing
  - Inspired to map/reduce functions in LISP

- Two main phases
  - **Map**: data is broken up into chunks that are processed in parallel
  - **Reduce**: combines the output from the mappers into the final result

- Computation processes as MAP-Reduce pipelines
  - A brute-force approach to processing
  - *Not always the most efficient or elegant solution*

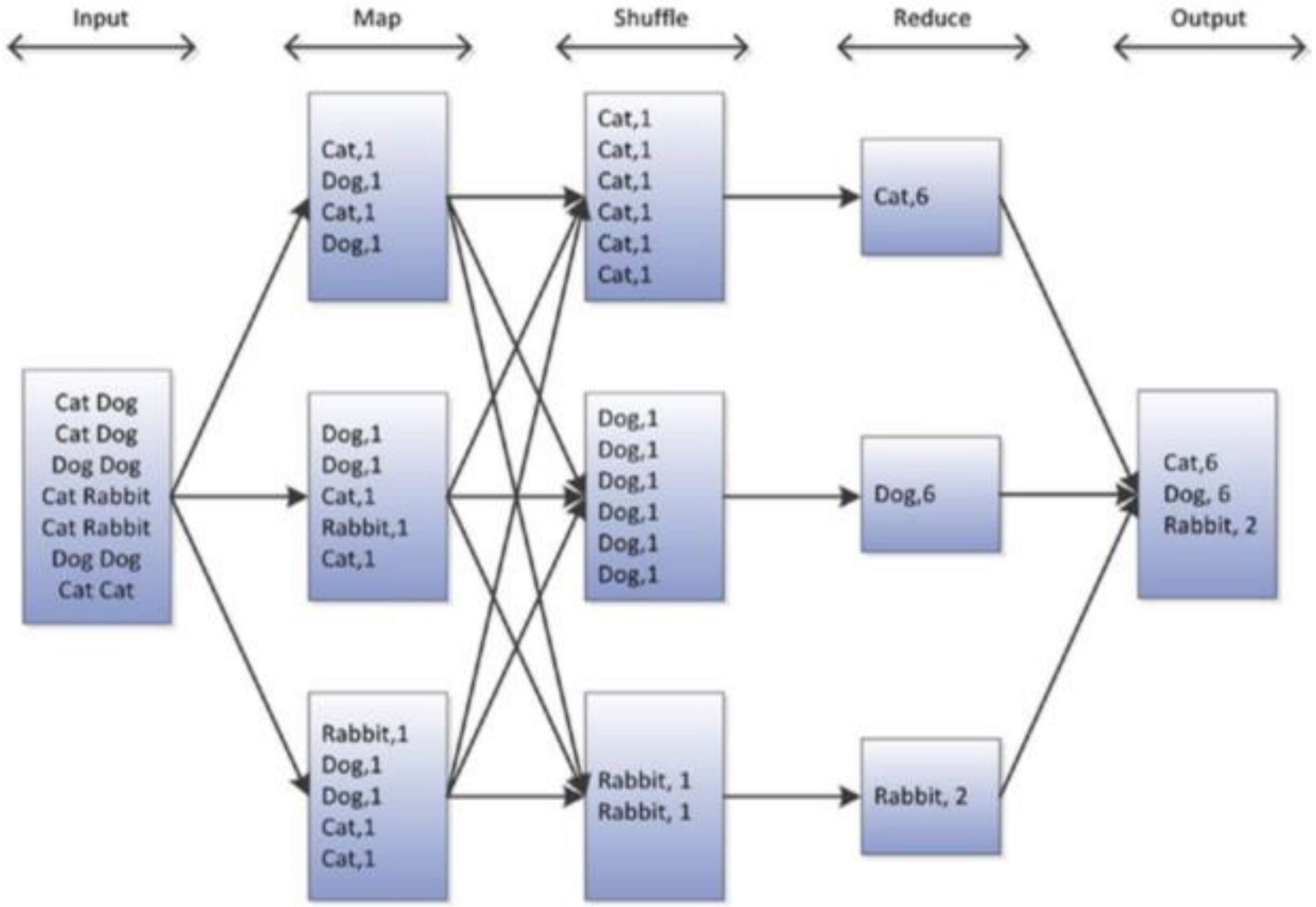


Figure 2-4. *Simple MapReduce pipeline*

Big Table (1)

- The basis for one of the first NoSQL database systems
  - does not support a full relational data model
  - simple data model with dynamic control over data layout and format
  - clients reason about the locality properties of the data

BigTable is a Sparse Distributed Persistent Multi-dimensional sorted map

- The map is indexed by
  - a row key, column key, and a timestamp
  - each value in the map is an uninterpreted array of bytes.
  - *(row:string, column:string, time:int64) → string*

Big Table (2)

- The design was guided by the needs of Google
  - *"we want to keep a copy of a large collection of web pages and related information that could be used by many different projects"*
  - *" we would use URLs as row keys, various aspects of web pages as column names, and store the contents of the web pages in the contents column under the timestamps when they were fetched"*

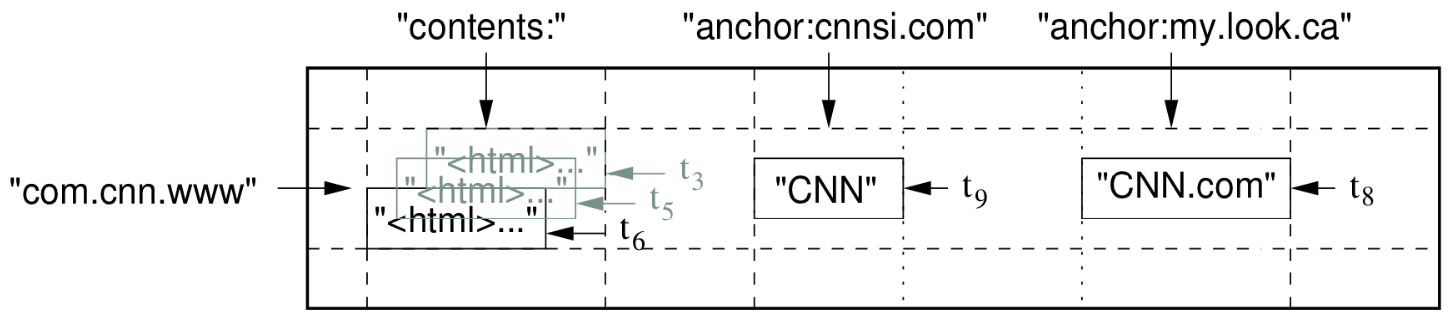


Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t_3, t_5, and t_6.

Rows and Columns in Big Table

- The row keys in a table are arbitrary strings (up to 64KB in size)
    - Every read or write of data under a single row key is atomic
    - Bigtable maintains data in lexicographic order by row key
    - The row range (tablet) for a table is dynamically partitioned

- Column keys are grouped into sets called column families
    - Named using the following syntax: *family:qualifier*
    - All data stored in a column family is usually of the same type
    - Column family must be created before data can be stored
    - Access control and both disk and memory accounting are performed at the column-family level

Timestamps in Big Table

- Timestamps for multiple versions of the same data
    - Timestamps are 64-bit integers (“real time” in microseconds)
    - Applications avoid collisions with unique timestamps themselves
    - Different versions of a cell are stored in decreasing timestamp order
    - Bigtable to garbage-collect cell versions automatically

Implementation (1)

- Three major components of BigTable:
  - Client (API)
  - Master server (only one)
    - Assign/detect tablets in tablet servers,
    - Balances tablet-server load, and does garbage collection
  - Tablet server (many, added/removed dynamically)
    - Manages a set of tablets (between ten to a thousand)
- Tablet servers
  - Handle table splits
    (tablets, 100-200 MB in size, 10-100 tablets per server)
  - Handles read and write requests
  - In Google *SSTable* file format
    - persistent, ordered immutable map from keys to values,
      keys and values are strings

Implementation (2)

- Distributed lock service (Chubby)
  - Provides a namespace that consists of directories and small files
    - Each directory or file can be used as a lock
    - Reads and writes to a file are atomic
  - Ensure there is at most one active master
  - Store the bootstrap location of Bigtable data
  - Discover tablet servers and finalize tablet server deaths
  - Store Bigtable schema information
  - Store access control lists
- Client data does not move through the master
  - communicate directly with tablet servers

Hadoop ecosystem

- In 2004, Doug Cutting and Mike Cafarella
  - Open-source project to build a web search engine called Nutch
  - Nutch could not scale to index the entire web
  - After google published GFS, MapReduce and BigTable
    - *The Nutch team implemented their GFS and MapReduce equivalents*
  - Hadoop public in 2007, a top-level Apache project in 2008
- Hadoop is the open source version of Google's software
  - In 2006, Yahoo! hired Cutting to work on Hadoop
  - In 2007 Facebook started experimenting with Hadoop
    - FB had a cluster utilizing 2,500 CPU cores in production
- Hadoop has become the de facto solution for big data
  - Microsoft, Oracle, and IBM oofer Hadoop within its product portfolio!

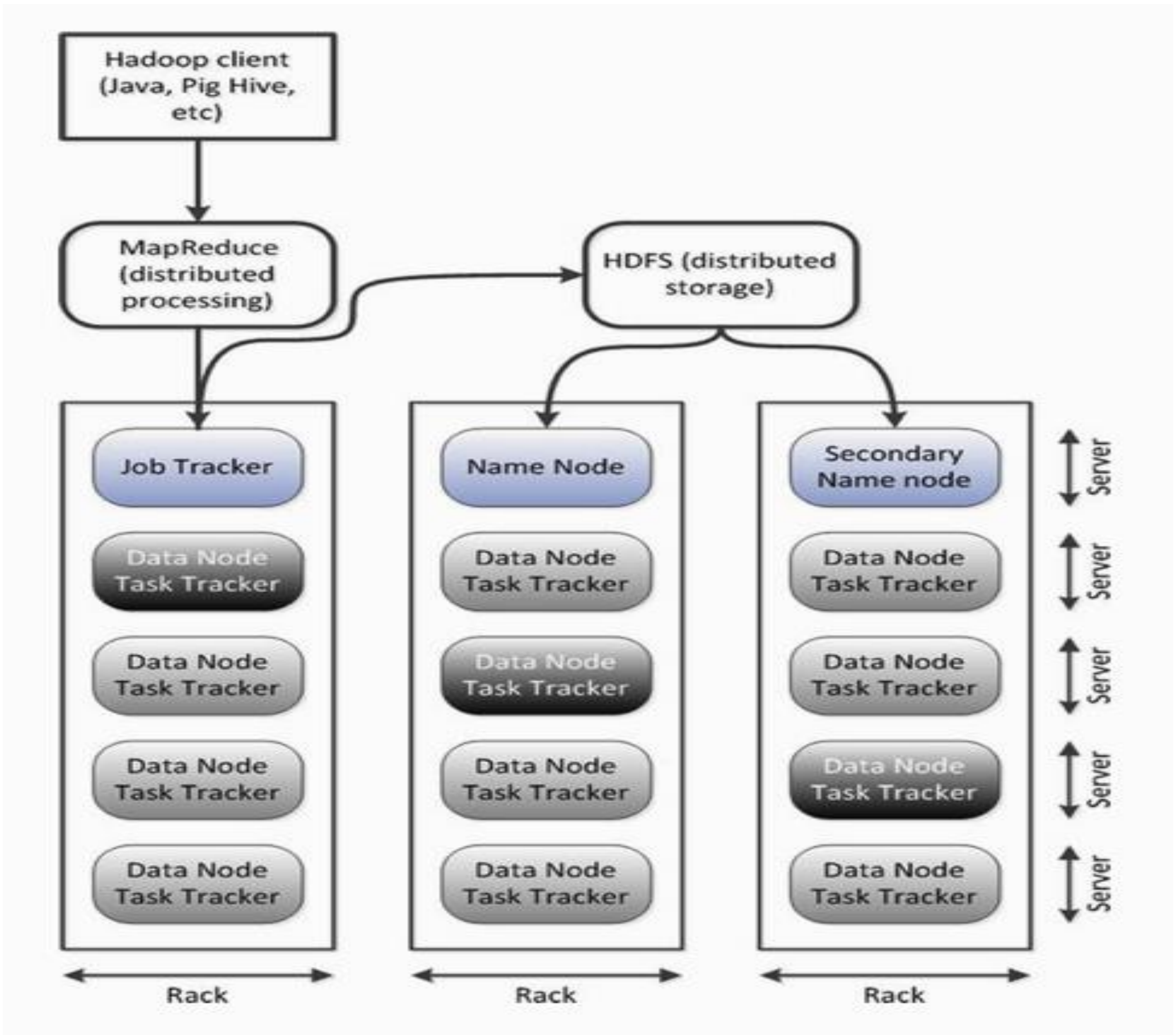
The power of Hadoop

- **An economical scalable storage model**
  - Hadoop can run on commodity hardware that in turn utilizes commodity disks
- **Massive scalable IO capability**
  - The aggregate IO and network capacity is higher than that provided by dedicated storage arrays
  - Adding new servers to Hadoop adds storage, IO, CPU, and network capacity all at once
- **Reliability**
  - Data is stored redundantly in multiple servers distributed across multiple computer racks
  - Failure of a server does not result in a loss of data
- **A scalable processing model**
  - MapReduce represents a widely applicable and scalable distributed processing model.
- **Schema on read**
  - Quickly ingest data from various forms, no conversion to a normalized format
  - The imposition of structure can be delayed until the data is accessed
  - *Schema on read* vs *schema on write* (of relational data warehouses)

The parallel with Google Software

- *Hadoop's architecture roughly parallels that of Google*
- Google File System
  - *Hadoop Distributed File System* (*HDFS*)
- Map Reduce processing model
  - Implemented in JAVA
  - Refined in verison 2 with YARN
  - MapReduce is one of the possible execution frameworks
- BigTable
  - *Hbase*

- *…and many other "tools"*
  - Hive - "SQL for Hadoop" …


Hadoop client (Java, Pig Hive, etc)
MapReduce (distributed processing)
HDFS (distributed storage)
Job Tracker
Name Node
Secondary Name node
Data Node Task Tracker
Server
Rack

Hadoop vs RDBMS

- Scale-out instead of scale-up
- Key/value pairs instead of relational tables
- Functional programming (mapreduce) (instead of declarative queries in SQL)
- Offline batch processing instead of online transactions

The building blocks of Hadoop

- “Running Hadoop” means running a set of daemons
  - NameNode (only one)
    - The master and bookkeeper of HDFS
      - how your files are broken down into file blocks
      - which nodes store those blocks, health of the distributed filesystem
  - DataNode (many)
    - Each slave machine in your cluster will host a DataNode daemon
    - The grunt work of the distributed filesystem ("chunkserver")
  - Secondary NameNode (possibly one)
    - Take snapshots of the HDFS metadata at intervals defined by the cluster configuration
    - Recovery in case of NameNode failure
  - JobTracker (only one, master)
    - Determines the execution plan by determining which files to process
    - Assigns nodes to different tasks, and monitors all tasks as they’re running
  - TaskTracker (many, slave)
    - Manages the execution of individual tasks on each slave node
    - There is a single TaskTracker per slave node
    - Can spawn multiple JVMs to handle many map or reduce tasks in parallel

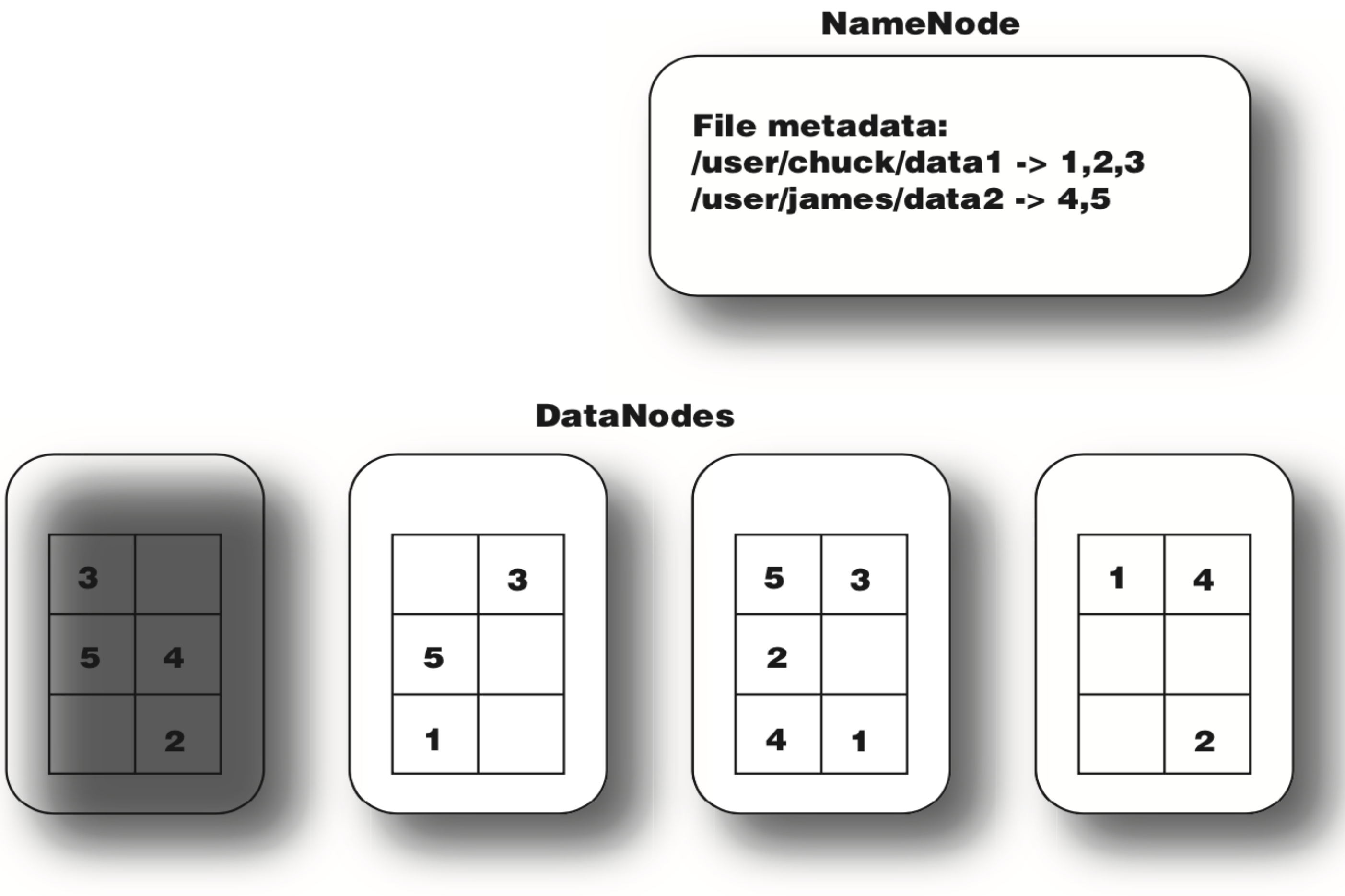


Figure 2.1 NameNode/DataNode interaction in HDFS. The NameNode keeps track of the file metadata—which files are in the system and how each file is broken down into blocks. The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.

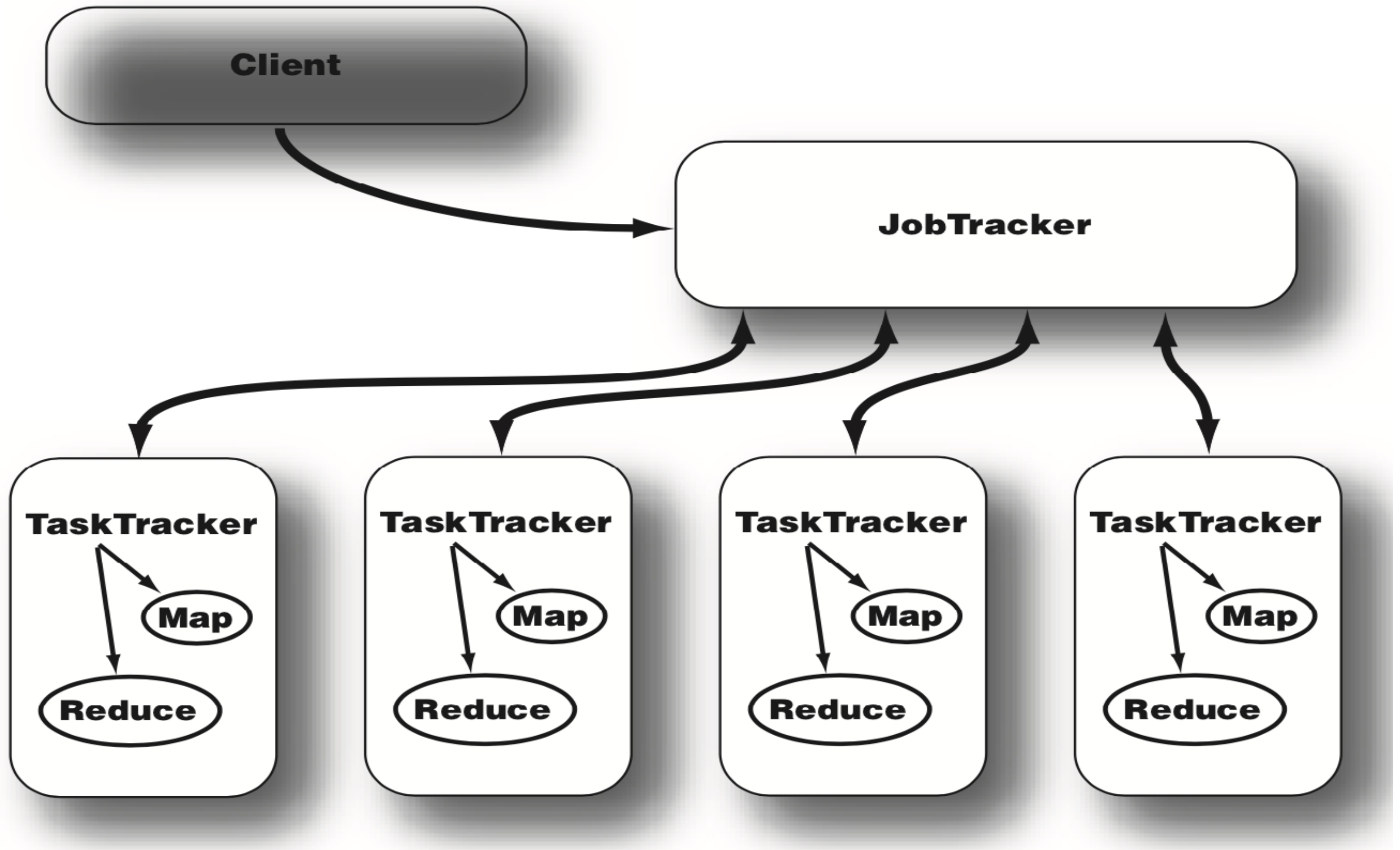


Figure 2.2 JobTracker and TaskTracker interaction. After a client calls the JobTracker to begin a data processing job, the JobTracker partitions the work and assigns different map and reduce tasks to each TaskTracker in the cluster.

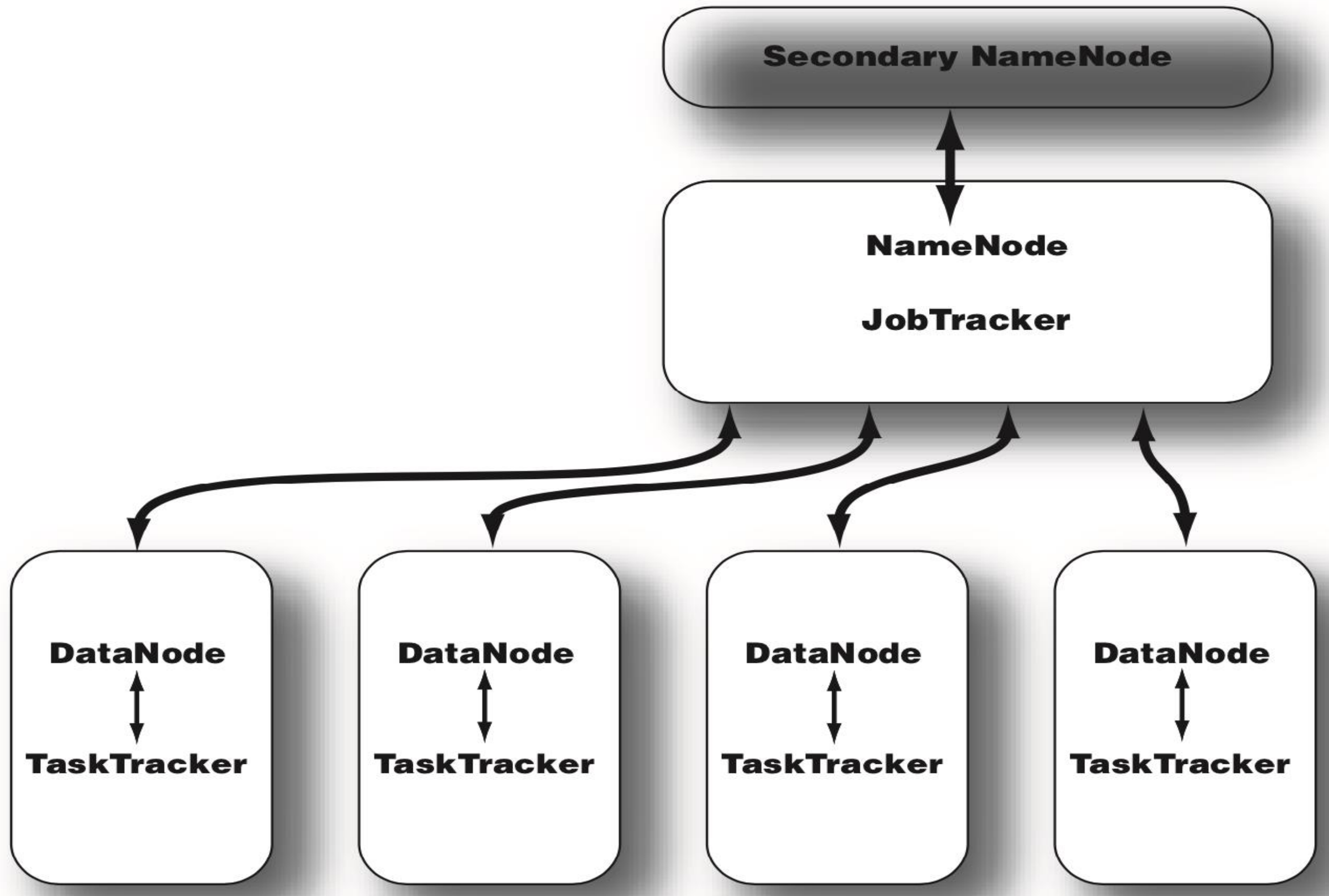
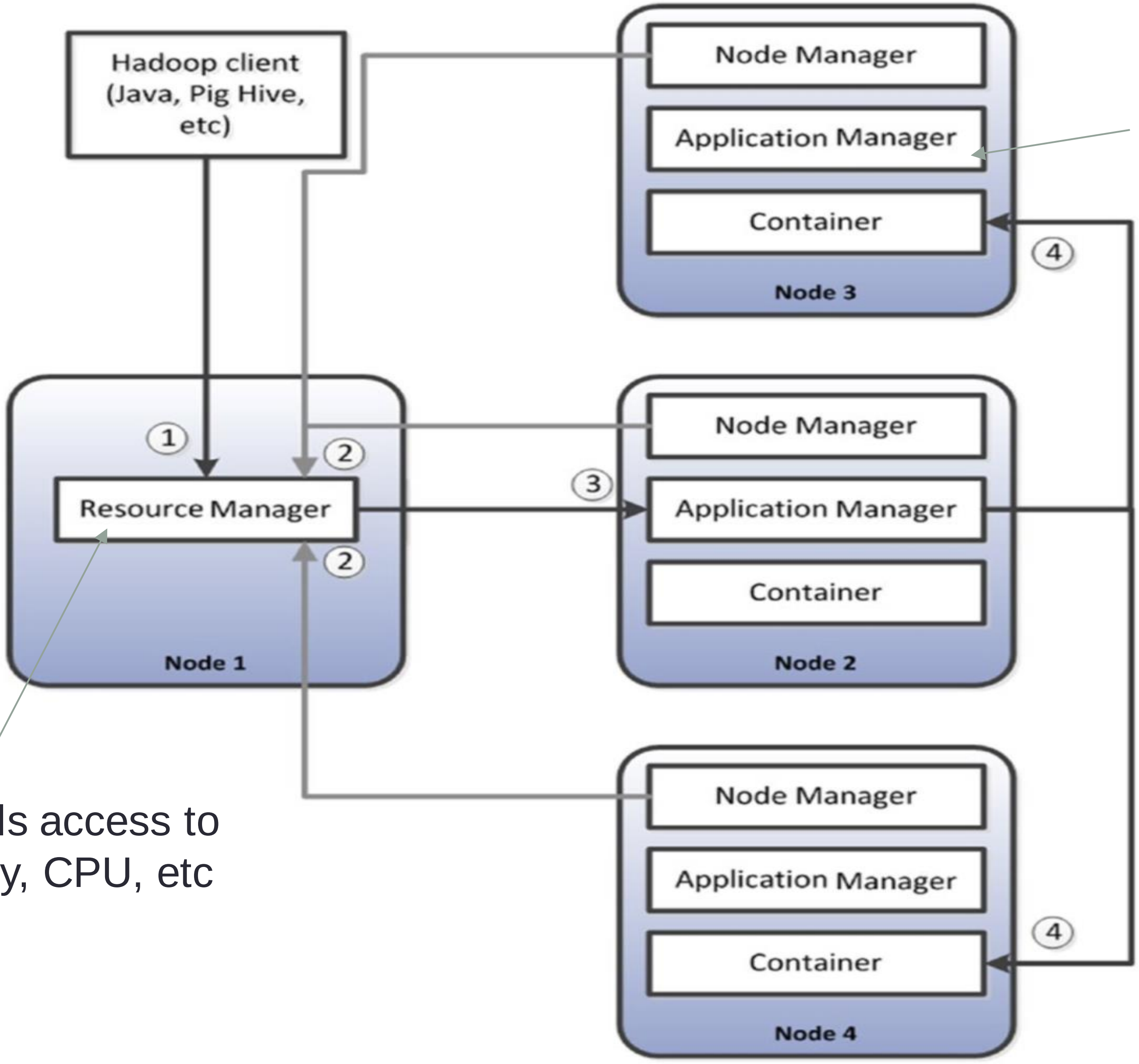


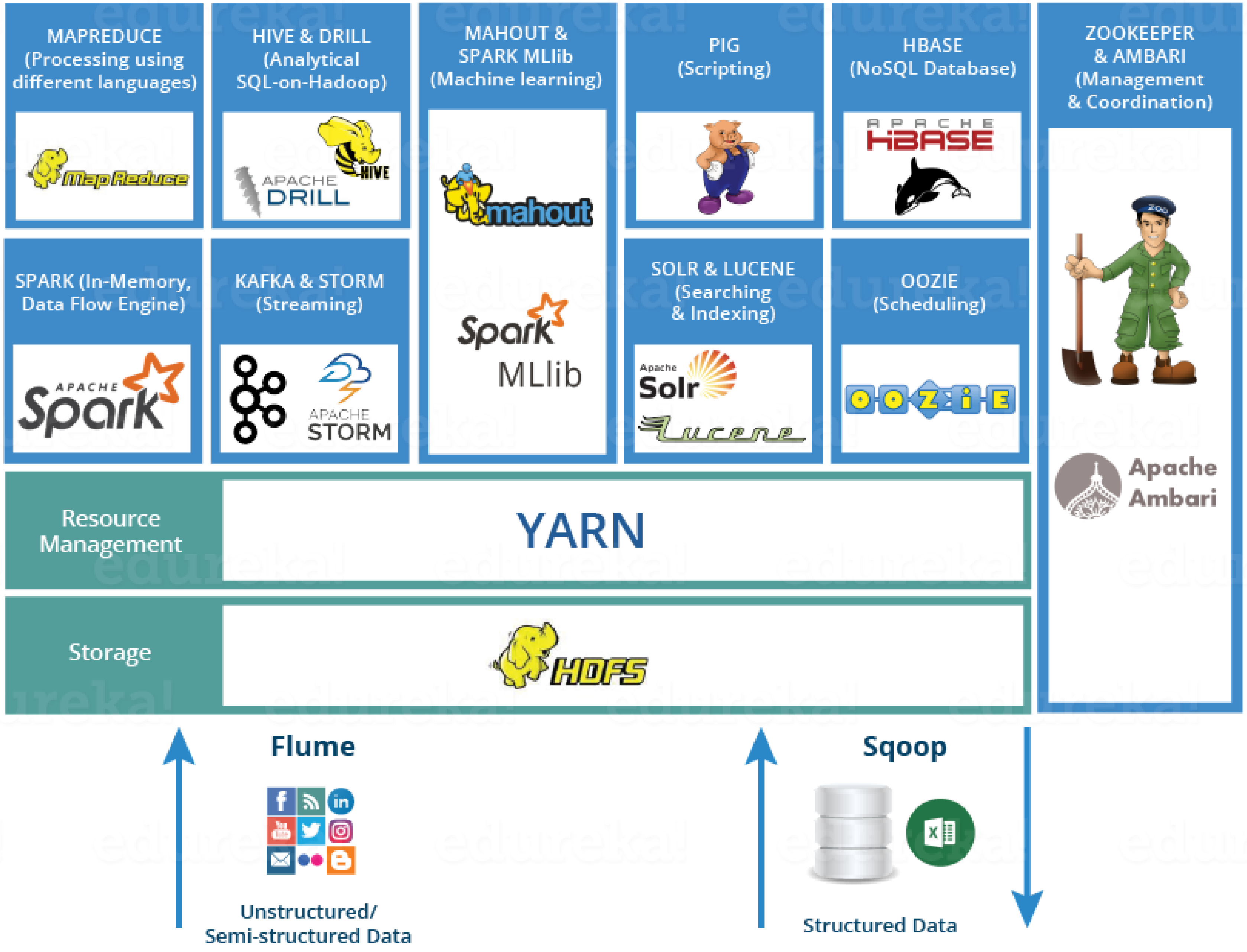
Figure 2.3 **Topology of a typical Hadoop cluster. It's a master/slave architecture in which the NameNode and JobTracker are masters and the DataNodes and TaskTrackers are slaves.**



controls task execution (one per job)

Controls access to memory, CPU, etc

Figure 2-7. *Hadoop 2.0 YARN architecture*


MAPREDUCE (Processing using different languages)
HIVE & DRILL (Analytical SQL-on-Hadoop)
MAHOUT & SPARK MLlib (Machine learning)
PIG (Scripting)
HBASE (NoSQL Database)
ZOOKEEPER & AMBARI (Management & Coordination)
SPARK (In-Memory, Data Flow Engine)
KAFKA & STORM (Streaming)
SOLR & LUCENE (Searching & Indexing)
OOZIE (Scheduling)
Resource Management
YARN
Storage
HDFS
Flume
Unstructured/ Semi-structured Data
Sqoop
Structured Data
Apache Ambari