Chapter 2

Intro to Cloud Computing

Acknowledgements

- Wikipedia on Google, Hadoop, Spark, Storm, NoSQL, *etc*.
- An In-Depth Look at the HBase Architecture: https://mapr.com/blog/in-depth-look-hbase-architecture/
- Slides from Dan Weld's class at U. Washington
- Jeff Dean, Google, Inc.
- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, Google, Inc. OSDI 2006.
- S. Sudarshan's slides from a talk by Erik Paulson, UW Madison.
- Bigtable: A Distributed Storage System for Structured Data. Jing Zhang, EECS 584, Fall 2011.
- Richard Venutolo's slides on BigTable.
- Introduction to NOSQL Databases. Adopted from slides and/or materials by P. Hoekstra, J. Lu, A. Lakshman, P. Malik, J. Lin, R. Sunderraman, T. Ivarsson, J. Pokorny, N. Lynch, S. Gilbert, J. Widom, R. Jin, P. McFadin, C. Nakhli, and R. Ho.
- A Comparison of SQL and NoSQL Databases. Keith W. Hare JCC Consulting, Inc. Convenor, ISO/IEC JTC1 SC32 WG3.
- COMP9313: Big Data Management. Xin Cao.
- Distributed Software Engineering - Hadoop and Spark. David A. Wheeler. SWE 622, George Mason University.
- Berkeley Data Analytics Stack. Prof. Harld Liu, amplab.
- Storm. Original slides by Nathan Marz @ Twitter and Shyam Rajendran @ Nutanix.
- The Google File Syestem. Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung (Google).
- GFS, Mapreduce and Bigtable. Seminar on big data management, Jiaheng Lu. Spring 2016.
- Hadoop File System. B.Ramamuerthy.
- Cloud Tools Overview. Erik Jonsson, School of Engineering & Computer Science, UT Dallas.

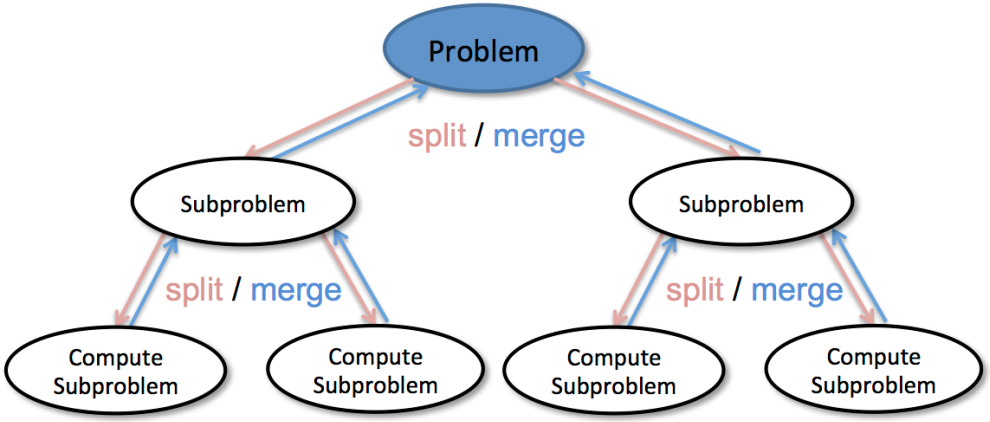
Chapter Outline

1. Philosophy
2. Google
   a. GFS
   b. MapReduce
   c. BigTable
3. Hadoop
   a. HDFS
   b. MapReduce
   c. HBase
4. Spark
5. Storm
6. NoSQL

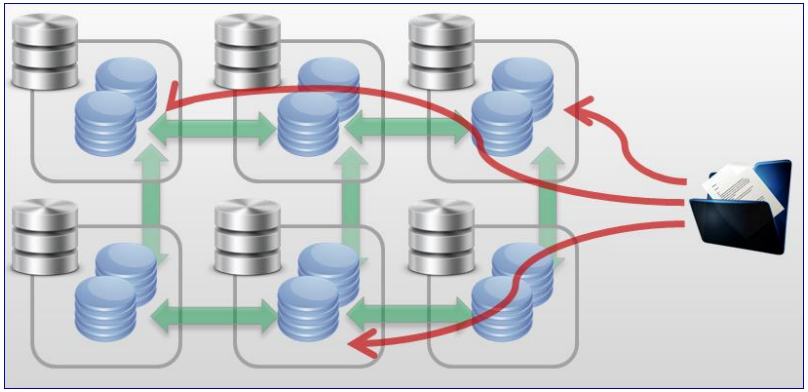
Background

Philosophy

- Divide and conquer



- Move computation to data

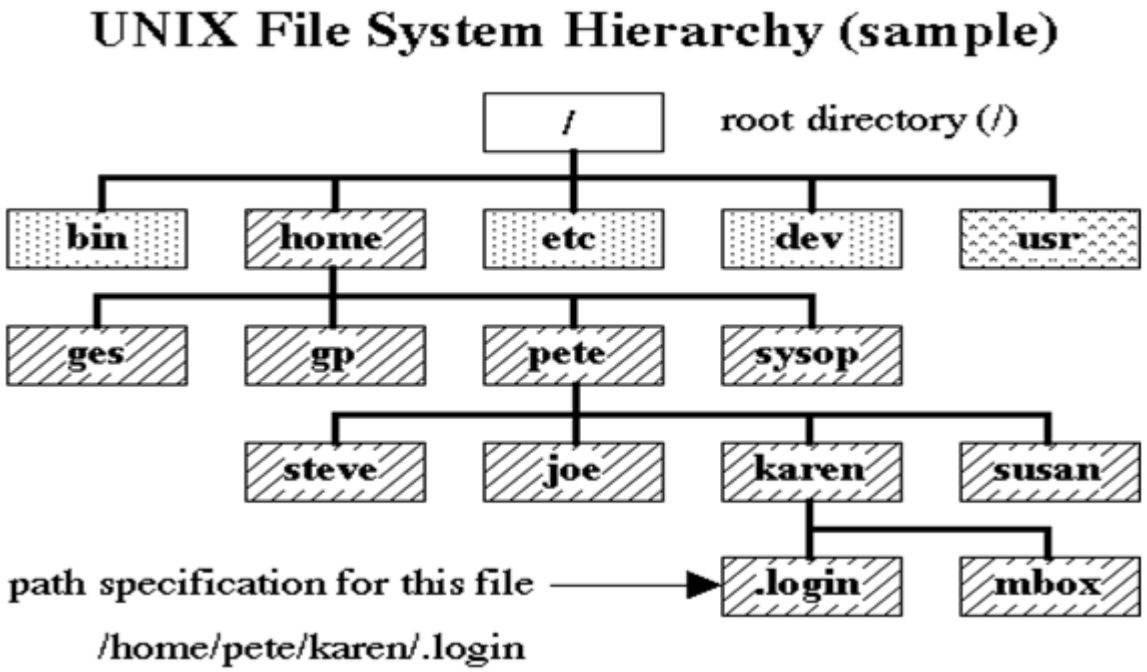


Philosophy

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow (low-latency), but disk throughput is reasonable (high throughput)
- A distributed file system is the answer
  - A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

Background

- File system
  - A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk.



Background

- **Latency** is the time required to perform some action or to produce some result.
  - Measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods.
  - I/O latency: the time that it takes to complete a single I/O.
- **Throughput** is the number of such actions executed or results produced per unit of time.
  - Measured in units of whatever is being produced (e.g., data) per unit of time.
  - Disk throughput: the maximum rate of sequential data transfer, measured by Mb/sec, *etc*.
- **Scalability**
  - Speedup
  - Sizeup
  - Scaleup

Google

Google: GFS

Google: GFS

- Motivation
  - Need for a scalable DFS
  - Large distributed data-intensive applications
  - High data processing needs
  - Performance, Reliability, Scalability and Availability
  - More than traditional DFS

Google: GFS

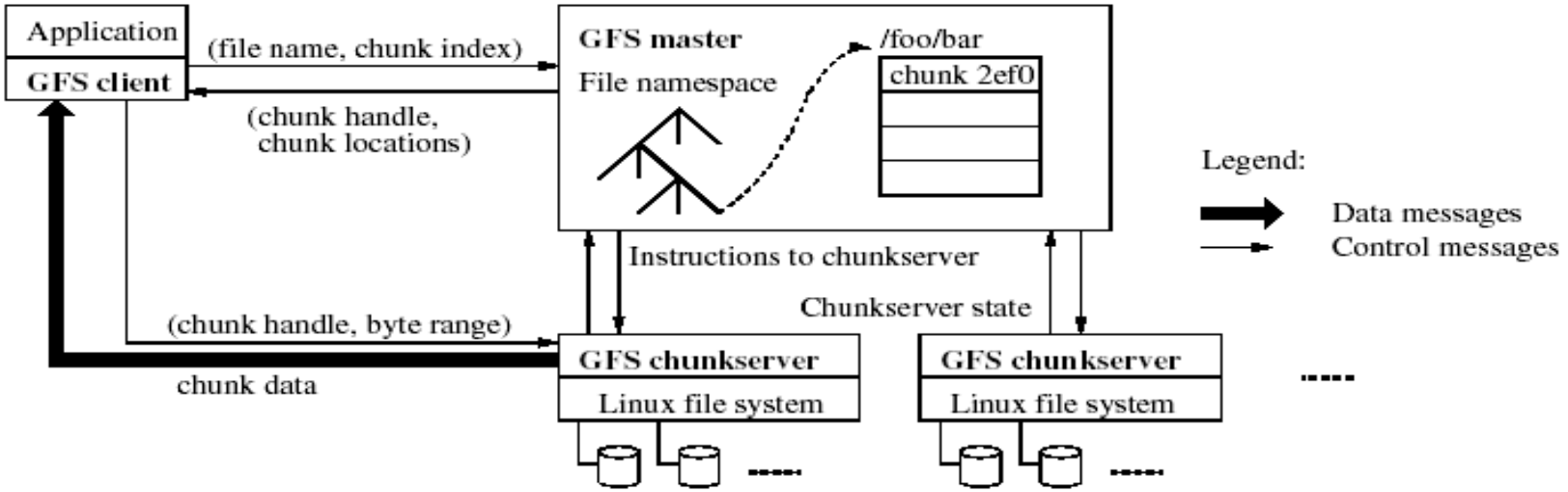
- Assumptions
  - Commodity hardware over “exotic” hardware
    - Inexpensive
    - Scale “out”, not “up”
  - Component failure
    - the norm rather than the exception
  - TBs of Space
    - must support TBs of space
  - Multi-GB files
    - Common
  - Workloads
    - Large streaming reads
    - Small random reads
    - Large, sequential writes that append data to file
    - Multiple clients concurrently append to one file
  - High sustained bandwidth
    - More important than low latency

Google: GFS

- Design decisions
  - Files are divided into chunks
    - Fixed-size chunks (64MB)
  - Replicated over chunkservers, called replicas
    - Each chunk replicated across 3+ chunkservers
  - Single master to coordinate access, keep metadata
    - Simple centralized management
  - No data caching
    - Little benefit due to large datasets, streaming reads
  - Simplify the API
    - Push some of the issues onto the client

Google: GFS

- Architecture
  - Contact single master
  - Obtain chunk locations
  - Contact one of chunkservers
  - Obtain data



Google: GFS

- Master
  - Metadata
    - File & chunk namespaces
    - Mapping from files to chunks
    - Locations of chunks' replicas
    - Replicated on multiple remote machines
    - Access-control information
    - Kept in memory
  - Operations
    - Replica placement
    - New chunk and replica creation
    - Load balancing
    - Unused storage reclaim

Google: GFS

- Chunkserver
  - Files are broken into chunks. Each chunk has a immutable globally unique 64-bit chunkhandle
  - Handle is assigned by the master at chunk creation
  - Chunk size is 64 MB
  - Each chunk is replicated on 3 (default) servers
- Client
  - Linked to apps using the file system API.
  - Communicates with master and chunkservers for reading and writing
    - Master interactions only for metadata
    - Chunkserver interactions for data
  - Only caches metadata information
    - Data is too large to cache

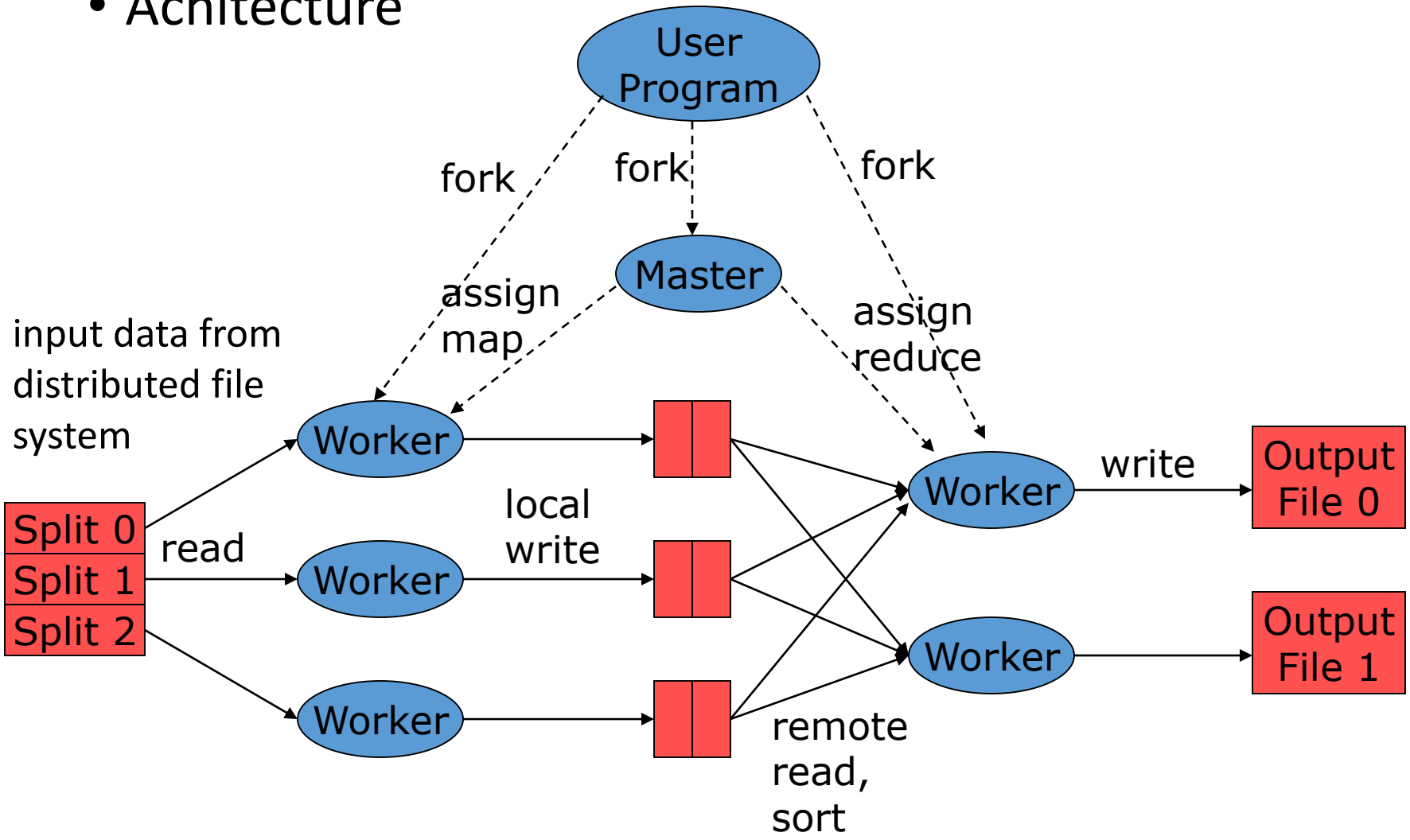
Google: MapReduce

Google: MapReduce

- Motivation
  - Large-Scale Data Processing
    - Want to use 1000s of CPUs
    - But don't want hassle of managing things
  - MapReduce provides
    - User-defined functions
    - Automatic parallelization & distribution
    - Fault tolerance
    - I/O scheduling
    - Monitoring & status updates

Google: MapReduce

- Achitecture

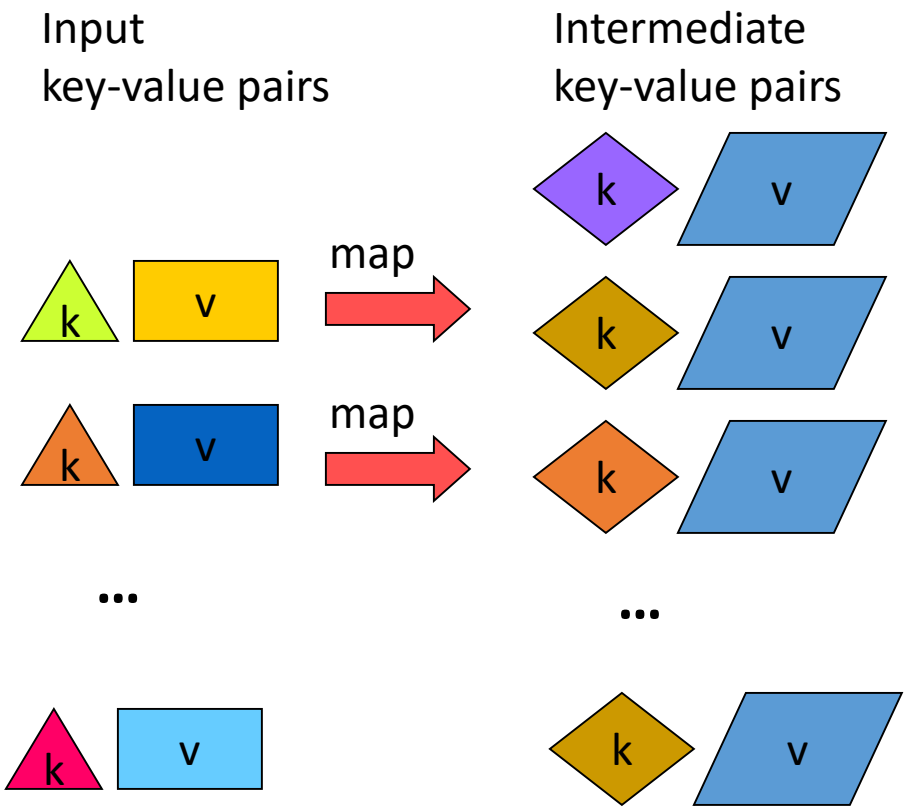


Google: MapReduce

- Map (Mapper)
  - Reads contents of assigned portion of input-file
  - Parses and prepares data for input to map function
  - Passes data into map function and saves result in memory
  - Periodically writes completed work to local disk (intermediate data)
  - Notifies Master of this partially completed work
    Perform a function on individual values in a data set to create a new list of values
- Example: square x = x * x
  map square [1,2,3,4,5]
  returns [1,4,9,16,25]

Google: MapReduce

Map phase:



Google: MapReduce

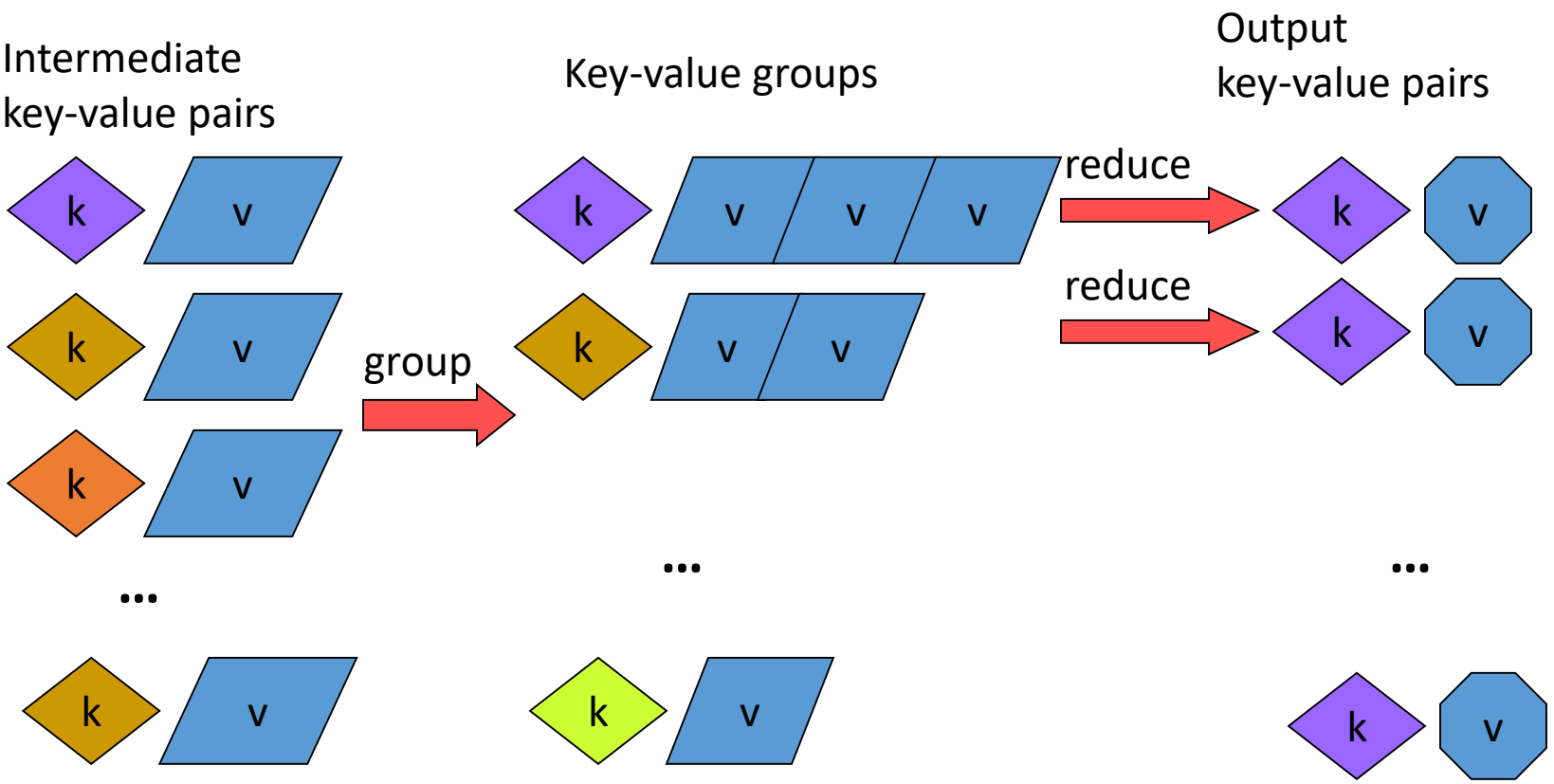
- Reduce (Reducer)
  - Receives notification from Master of partially completed work
  - Retrieves intermediate data from Map-Machine via remote-read
  - Sorts intermediate data by key (e.g. by *target* page)
  - Iterates over intermediate data
    - For each unique key, sends corresponding set through reduce function
  - Appends result of reduce function to final output file (GFS)
- Combine values in a data set to create a new value
- Example: sum = (each elem in arr, total +=)

  reduce [1,2,3,4,5]

  returns 15 (the sum of the elements)

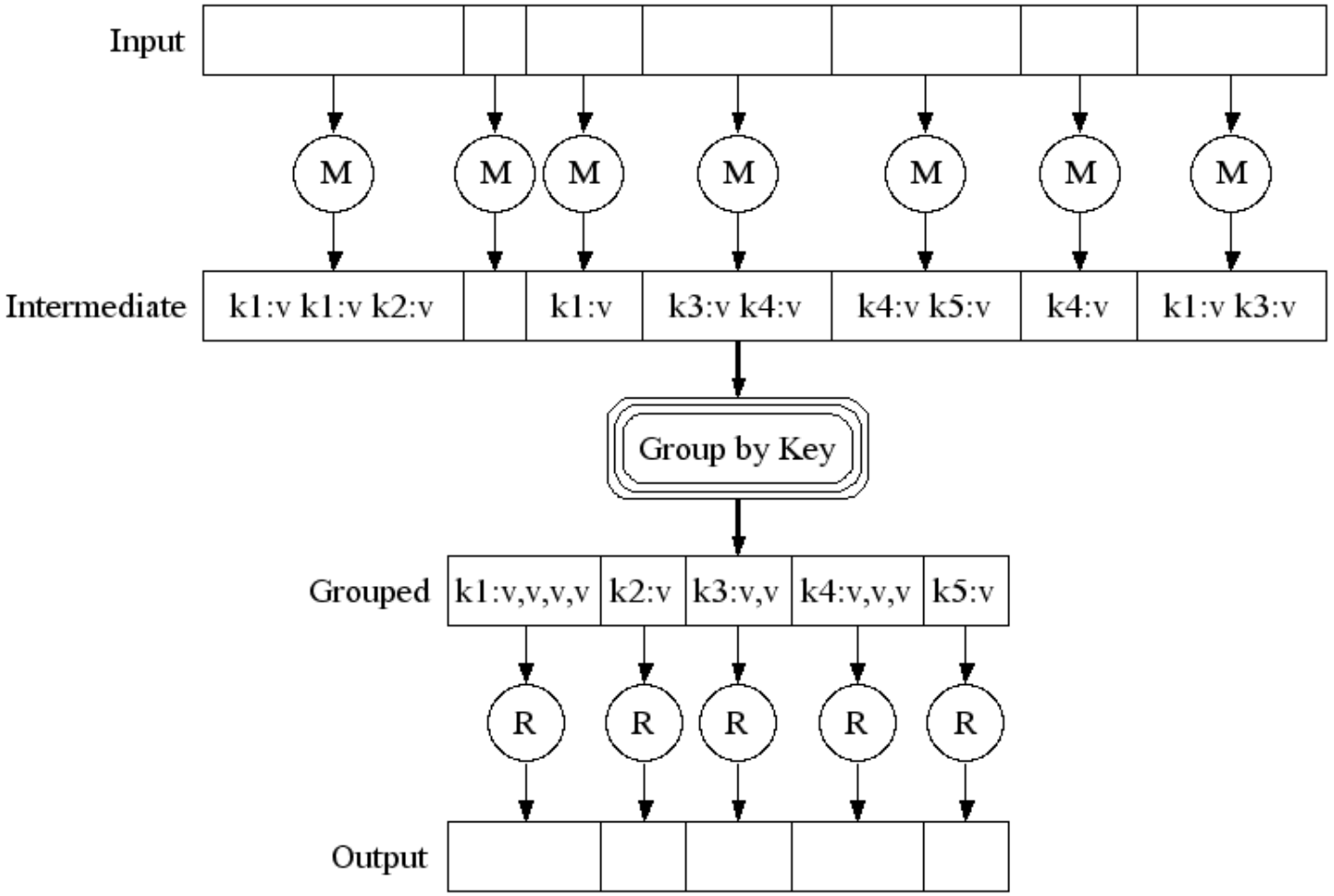
Google: MapReduce

Reduce phase:



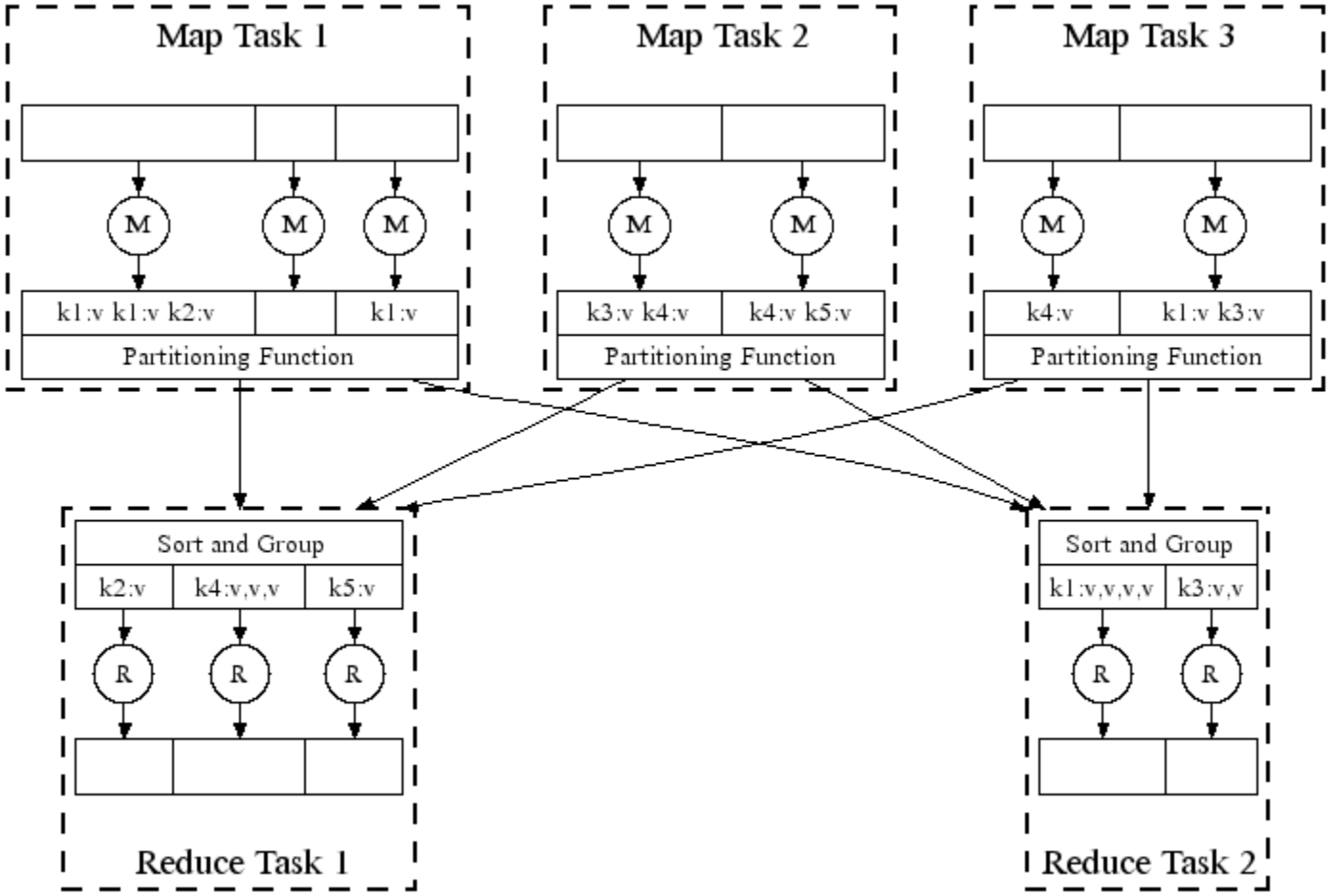
Google: MapReduce

- Execution

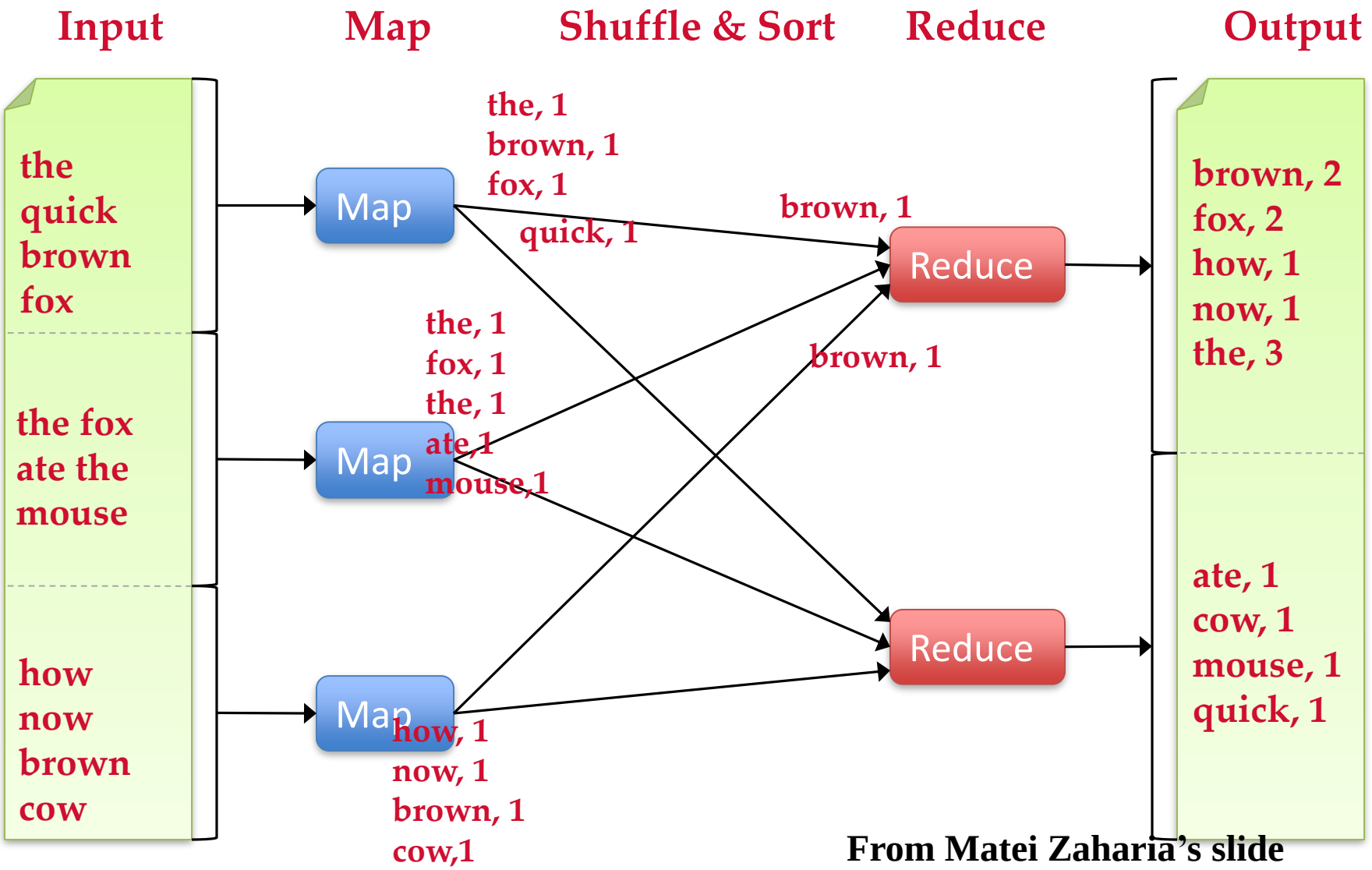


Google: MapReduce

- Parallel execution



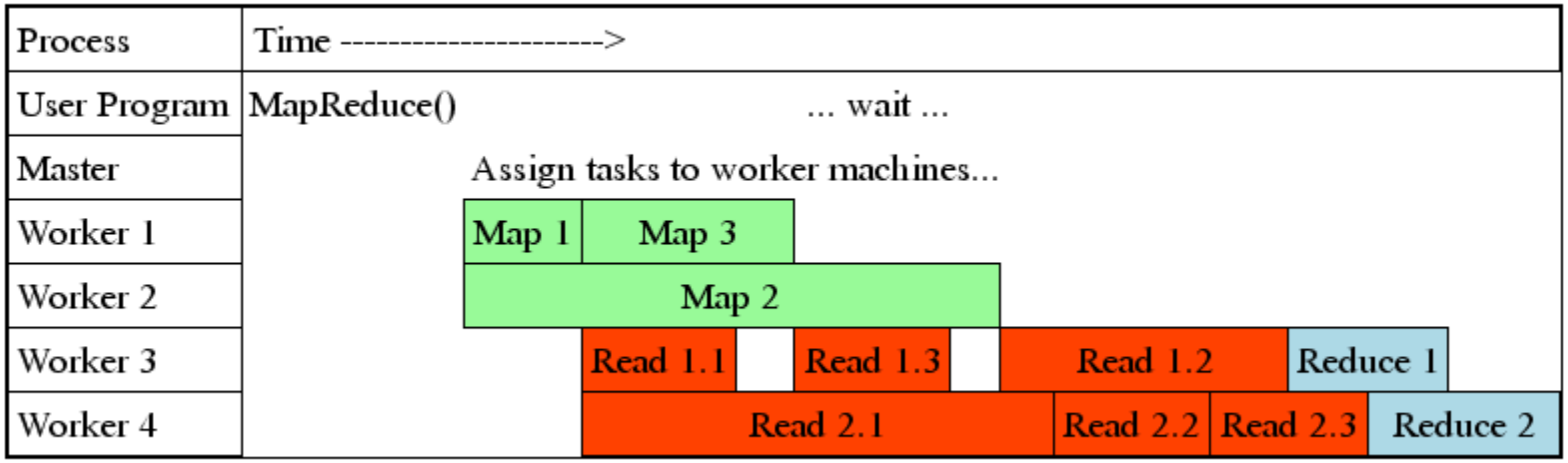
Word Count Execution



From Matei Zaharia's slide

Google: MapReduce

- Task Granularity & Pipelining
  - Fine granularity tasks: map tasks >> machines
    - Minimizes time for fault recovery
    - Can pipeline shuffling with map execution
    - Better dynamic load balancing
  - Often use 200,000 map & 5000 reduce tasks
  - Running on 2000 machines



Google: MapReduce

- Worker failure
  - Detect failure via periodic heartbeats
  - Any machine who does not respond is considered “dead”
  - Failure handled via **re-execution**
    - Re-execute completed & in-progress map tasks
      - Any task in progress gets needs to be re-executed and becomes eligible for scheduling
      - Completed tasks are also reset because results are stored on local disk
      - Reduce-Machines notified to get data from new machine assigned to assume task
    - Re-execute in progress reduce tasks

Google: MapReduce

- Master failure
  - Could handle, … ?
  - But don't yet
  - (master failure unlikely)

Google: BigTable

Google: BigTable

- Google's Motivation
  - Scale Problem
    - Lots of data
    - Millions of machines
    - Different project/applications
    - Hundreds of millions of users
  - Storage for (semi-)structured data
  - No commercial system big enough
    - Couldn't afford if there was one
  - Low-level storage optimization help performance significantly
    - Much harder to do when running on top of a database layer

Google: BigTable

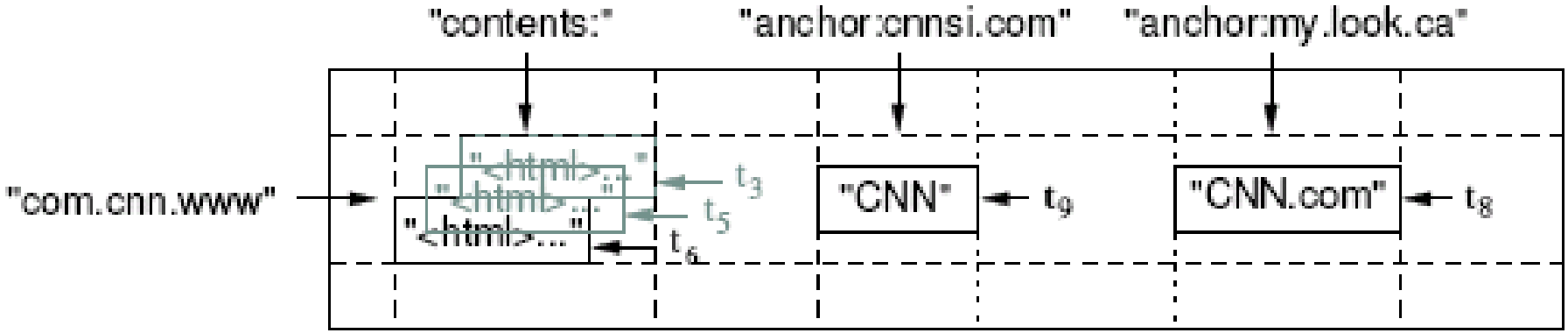
- What is BigTable
  - Distributed multi-level map
  - Fault-tolerant, persistent
  - Scalable
    - Thousands of servers
    - Terabytes of in-memory data
    - Petabyte of disk-based data
    - Millions of reads/writes per second, efficient scans
  - Self-managing
    - Servers can be added/removed dynamically
    - Servers adjust to load imbalance

Google: BigTable

- Building Blocks
  - Google File System (GFS)
    - stores persistent data (SSTable file format)
  - Scheduler
    - schedules jobs onto machines
  - Chubby
    - Lock service: distributed lock manager
    - master election, location bootstrapping
  - MapReduce (optional)
    - Data processing
    - Read/write Bigtable data

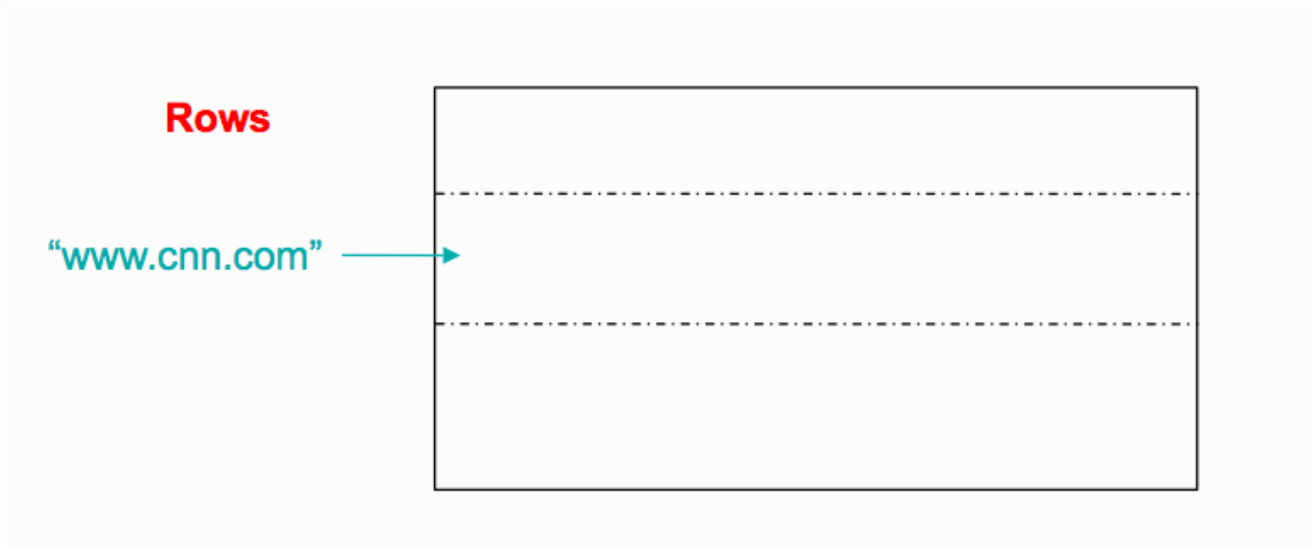
Google: BigTable

- Data model: a big map
  - <Row, Column, Timestamp> triple for key - lookup, insert, and delete API
  - Arbitrary “columns” on a row-by-row basis
    - Column family:qualifier. Family is heavyweight, qualifier lightweight
    - Column-oriented physical store- rows are sparse!
  - Does not support a relational model
    - No table-wide integrity constraints
    - No multirow transactions



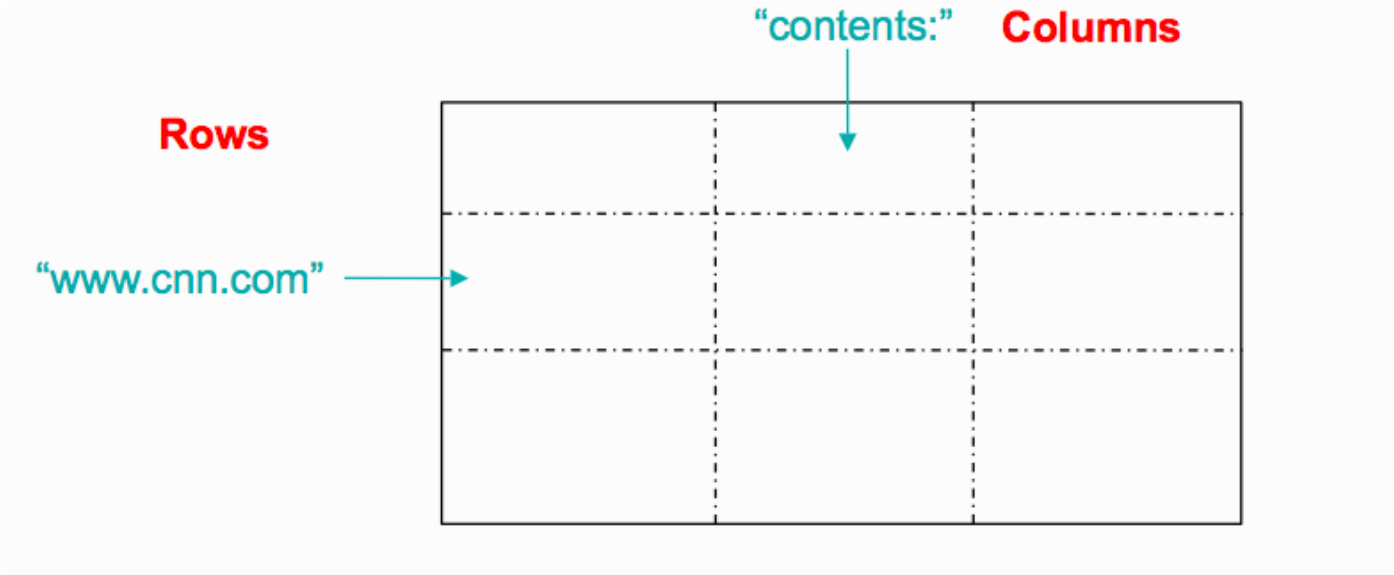
Google: BigTable

- Rows
  - Arbitrary string
  - Access to data in a row is atomic
  - Ordered lexicographically



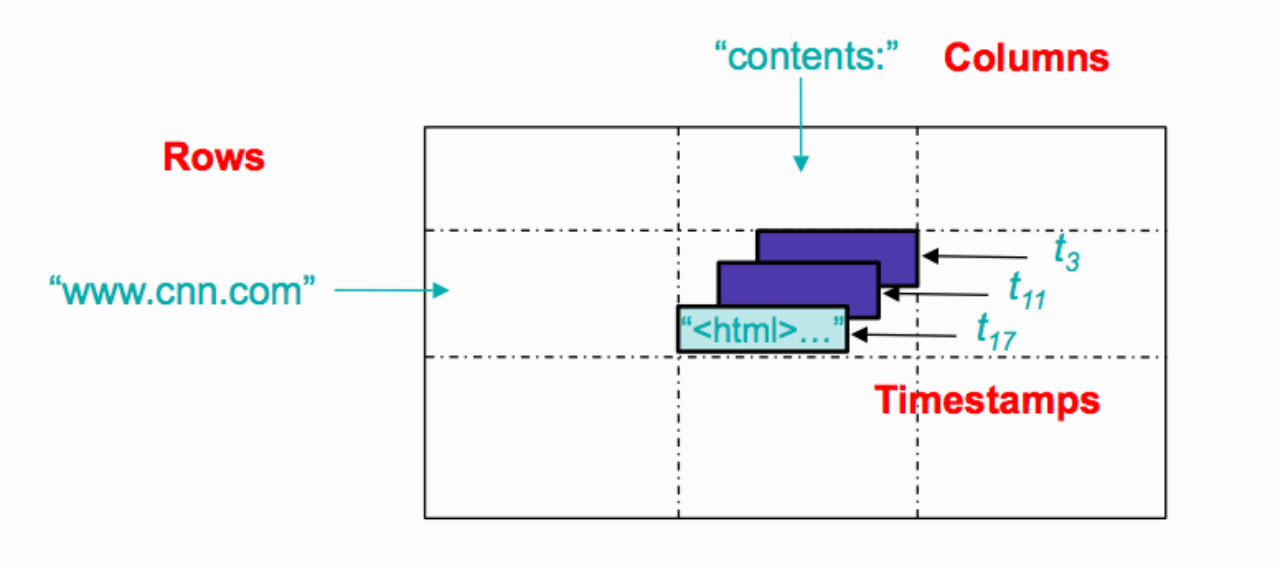
Google: BigTable

- Column
  - Tow-level name structure:
    - family: qualifier
  - Column Family is the unit of access control



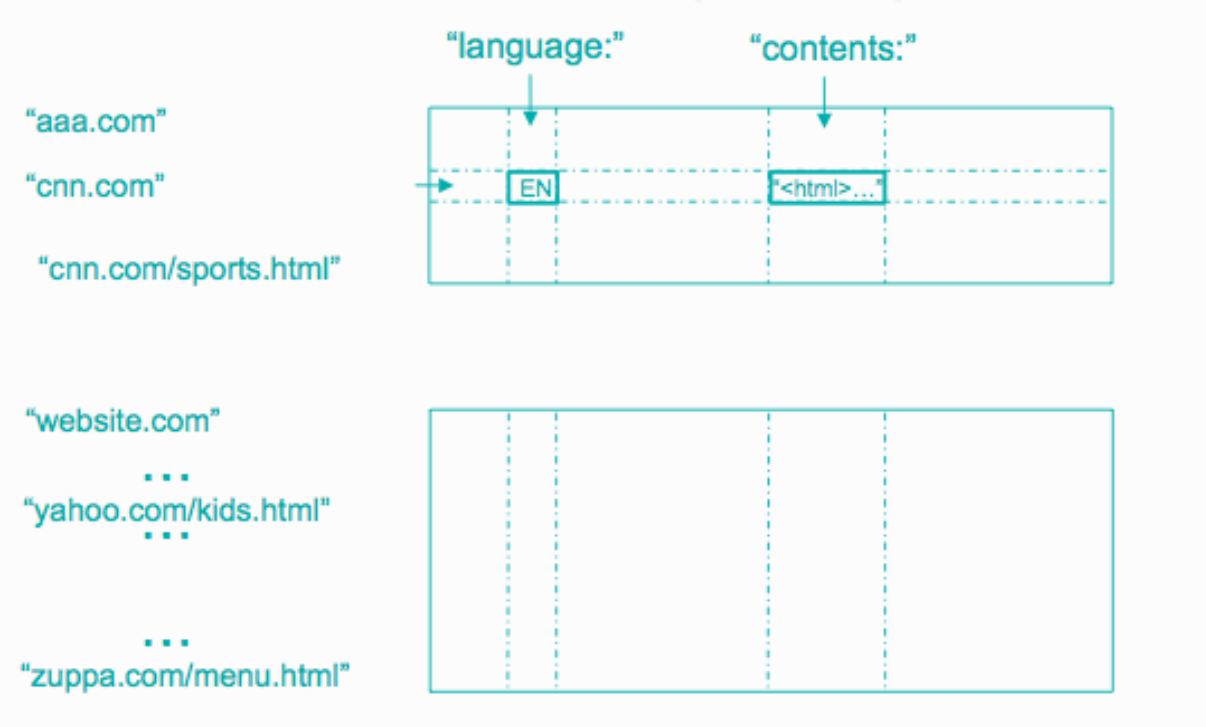
Google: BigTable

- Timestamps
  - Store different versions of data in a cell
  - Lookup options
    - Return most recent K values
    - Return all values



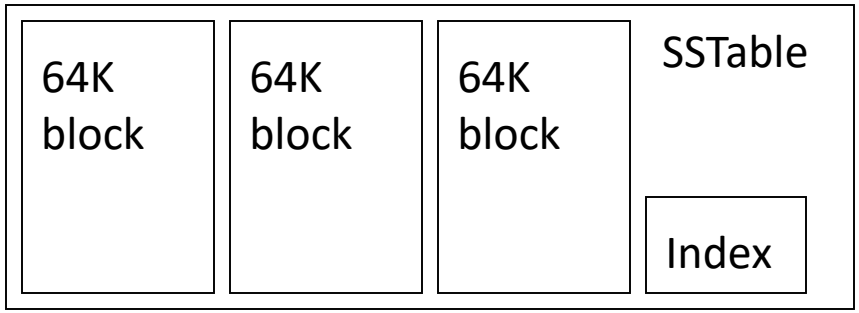
Google: BigTable

- The row range for a table is dynamically partitioned
- Each row range is called a tablet
- Tablet is the unit for distribution and load balancing



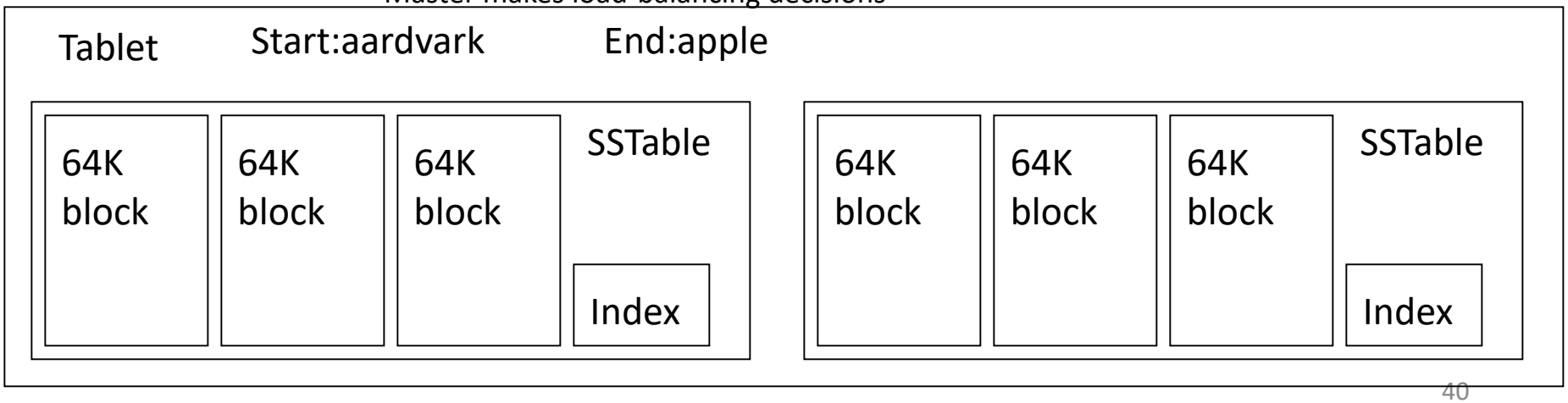
Google: BigTable

- SSTable
  - Immutable, sorted file of key-value pairs
  - Chunks of data plus an index
    - Index is of block ranges, not values



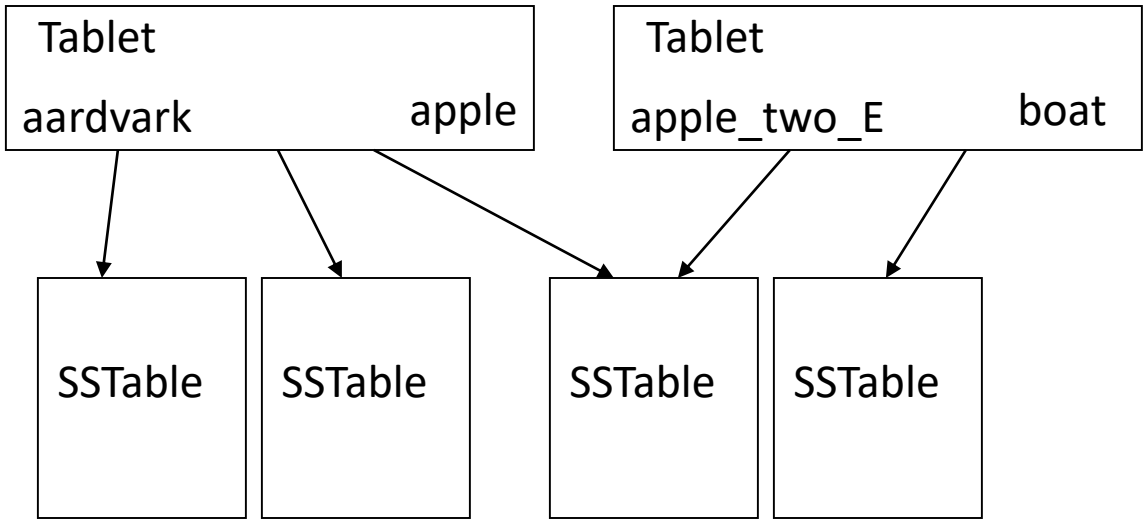
Google: BigTable

- Tablet
  - Built out of multiple SSTables
  - Each Tablets is assigned to one tablet server.
    - Tablet holds contiguous range of rows
    - Aim for ~100MB to 200MB of data per tablet
  - Tablet server is responsible for ~100 tablets
    - Fast recovery:
      - 100 machines each pick up 1 tablet for failed machine
    - Fine-grained load balancing:
      - Migrate tablets away from overloaded machine
      - Master makes load-balancing decisions



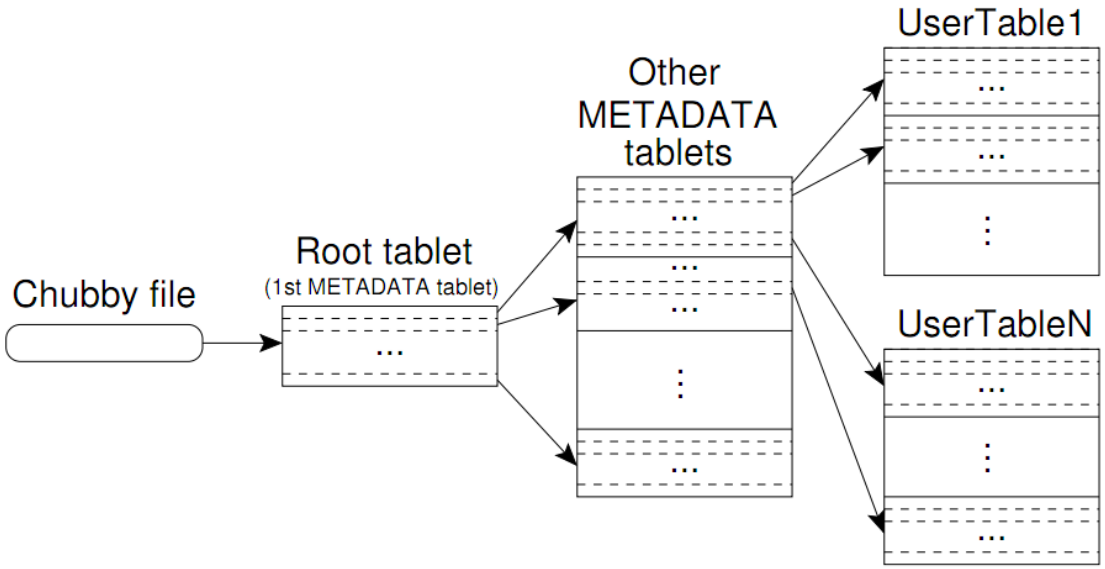
Google: BigTable

- Table
  - Multiple tablets make up the table
  - SSTables can be shared
  - Tablets do not overlap, SSTables can overlap



Google: BigTable

- Locate a tablet
  - Stores: Key: table id + end row, Data: location
  - Cached at clients, which may detect data to be incorrect
    - in which case, lookup on hierarchy performed
  - Also prefetched (for range queries)



Google: BigTable

- Tablet Assignment
  - Each tablet is assigned to one tablet server at a time.
  - Master server keeps track of the set of live tablet servers and current assignments of tablets to servers.
  - When a tablet is unassigned, master assigns the tablet to an tablet server with sufficient room.
  - It uses *Chubby* to monitor health of tablet servers, and restart/replace failed servers.

Google: BigTable

- Chubby
  - Tablet server registers itself by getting a lock in a specific directory chubby
    - Chubby gives "lease" on lock, must be renewed periodically
    - Server loses lock if it gets disconnected
  - Master monitors this directory to find which servers exist/are alive
    - If server not contactable/has lost lock, master grabs lock and reassigns tablets
    - GFS replicates data. Prefer to start tablet server on same machine that the data is already at

Google: BigTable

- Locality Groups
  - Can group multiple column families into a *locality group*
    - Separate SSTable is created for each locality group in each tablet.
  - Segregating columns families that are not typically accessed together enables more efficient reads.
    - In WebTable, page metadata can be in one group and contents of the page in another group.
- Compression
  - Many opportunities for compression
    - Similar values in the cell at different timestamps
    - Similar values in different columns
    - Similar values across adjacent rows

Hadoop

Hadoop: Assumptions

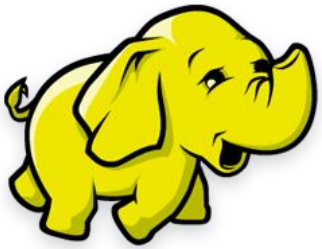
- Hardware will fail.
- Processing will be run in batches. Thus there is an emphasis on high throughput as opposed to low latency.
- Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size.
- It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.
- Applications need a write-once-read-many access model.
- Moving computation is cheaper than moving data.
- Portability is important.

Hadoop: What is Hadoop

- Apache top level project, open-source implementation of frameworks for reliable, scalable, distributed computing and data storage.
- It is a flexible and highly-available architecture for large scale computation and data processing on a network of commodity hardware.
- Designed to answer the question: “How to process big data with reasonable cost and time?”
- An open-source software framework that supports data-intensive distributed applications, licensed under the Apache v2 license.
- Abstract and facilitate the storage and processing of large and/or rapidly growing data sets
- Structured and non-structured data
- Simple programming models
- High scalability and availability
- Use commodity (cheap!) hardware with little redundancy
- Fault-tolerance
- Move computation rather than data

Hadoop: What is Hadoop

- At Google MapReduce operation are run on a special file system called Google File System (GFS) that is highly optimized for this purpose.
- GFS is not open source.
- Doug Cutting and others at Yahoo! reverse engineered the GFS and called it Hadoop Distributed File System (HDFS).
- The software framework that supports HDFS, MapReduce and other related entities is called the project Hadoop or simply Hadoop.
- This is open source and distributed by Apache.

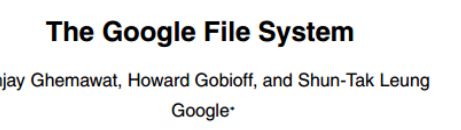


Hadoop: History

- Hadoop history (referred to the slides of ICS, UCIrvine)
  - 2005: Doug Cutting and Michael J. Cafarella developed Hadoop to support distribution for the Nutch search engine project. The project was funded by Yahoo.
  - 2006: Yahoo gave the project to Apache Software Foundation.
  - 2008 - Hadoop Wins Terabyte Sort Benchmark (sorted 1 terabyte of data in 209 seconds, compared to previous record of 297 seconds)
  - 2009 - Avro and Chukwa became new members of Hadoop Framework family
  - 2010 - Hadoop's Hbase, Hive and Pig subprojects completed, adding more computational power to Hadoop framework
  - 2011 - ZooKeeper Completed
  - 2013 - Hadoop 1.1.2 and Hadoop 2.0.3 alpha. Ambari, Cassandra, Mahout have been added

Hadoop vs Google

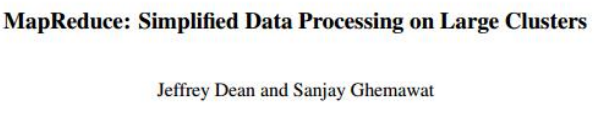
2003


The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

2004

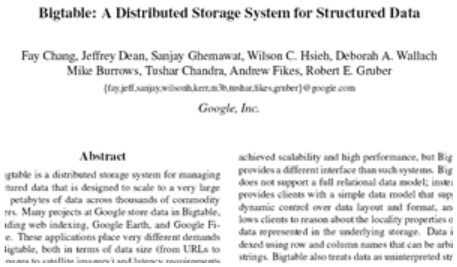

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

2006


Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

igtable is a distributed storage system for managing
tured data that is designed to scale to a very large
petabytes of data across thousands of commodity
ers. Many projects at Google store data in Bigtable,
nding web indexing, Google Earth, and Google Fi-
e. These applications place very different demands
igtable, both in terms of data size (from URLs to

achieved scalability and high performance, but Big
provides a different interface than such systems. Big
does not support a full relational data model; inste
provides clients with a simple data model that sup
dynamic control over data layout and format, an
lows clients to reason about the locality properties o
data represented in the underlying storage. Data i
dexed using row and column names that can be arbi
strings. Bigtable also treats data as uninterpreted str

Hadoop: Common Distributions

- Open Source
  - Apache
- Commercial
  - Cloudera
  - Hortonworks
  - MapR
  - AWS MapReduce
  - Microsoft Azure HDInsight (Beta)

via cs9313, Xin Cao, UNSW

Hadoop: Usage

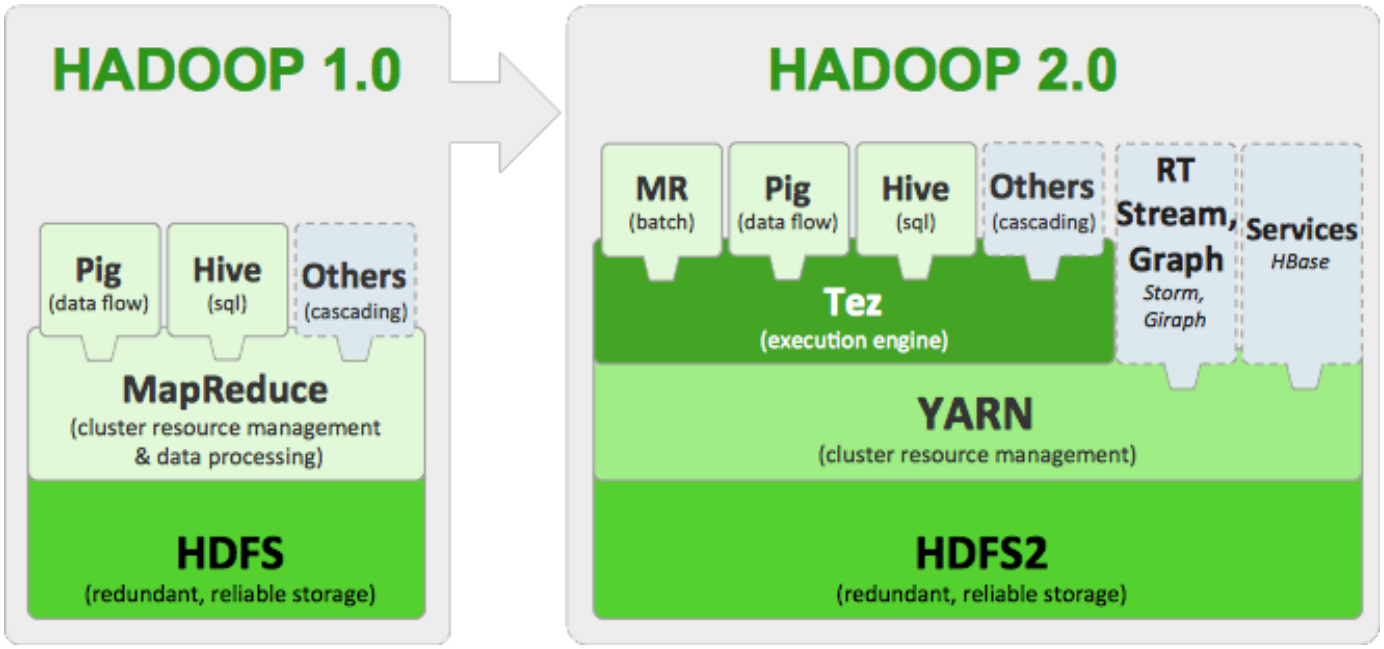
- Yahoo!
  - Yahoo!'s Search Webmap runs on 10,000 core Linux cluster and powers Yahoo! Web search
- Facebook
  - FB's Hadoop cluster hosts 100+ PB of data (July, 2012) & growing at ½ PB/day (Nov, 2012)
  - Facebook Messages
    - Design requirements
      - Integrate display of email, SMS and chat messages between pairs and groups of users
      - Strong control over who users receive messages from
      - Suited for production use between 500 million people immediately after launch
      - Stringent latency & uptime requirements
    - System requirements
      - High write throughput
      - Cheap, elastic storage
      - Low latency
      - High consistency (within a single data center good enough)
      - Disk-efficient sequential and random read performance
    - Facebook's solution
      - Hadoop + HBase as foundations
      - Improve & adapt HDFS and HBase to scale to FB's workload and operational considerations
      - Major concern was availability: NameNode is SPOF & failover times are at least 20 minutes
      - Proprietary "AvatarNode": eliminates SPOF, makes HDFS safe to deploy even with 24/7 uptime requirement
      - Performance improvements for realtime workload: RPC timeout. Rather fail fast and try a different DataNode
- Amazon
- Netflix
- NY Times
  - Non-realtime large dataset computing:
  - dynamically generating PDFs of articles from 1851-1922
  - Wanted to pre-generate & statically serve articles to improve performance
  - Using Hadoop + MapReduce running on EC2 / S3, converted 4TB of TIFFs into 11 million PDF articles in 24 hrs

Hadoop: Usage

- Companies Using Hadoop

Hadoop

- Hadoop framework



Hadoop: HDFS

Hadoop: HDFS: Assumptions

- Very large datasets
  - 10K nodes, 100 million files, 10PB
- Streaming data access
  - Designed more for batch processing rather than interactive use by users
  - The emphasis is on high throughput of data access rather than low latency of data access.
- Simple coherency model
  - Built around the idea that the most efficient data processing pattern is a write-once read-many-times pattern
  - A file once created, written, and closed need not be changed except for appends and truncates
- "Moving computation is cheaper than moving data"
  - Data locations exposed so that computations can move to where data resides
- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Hardware failure is normal rather than exception. Detect failures and recover from them
- Portability across heterogeneous hardware and software platforms
  - designed to be easily portable from one platform to another

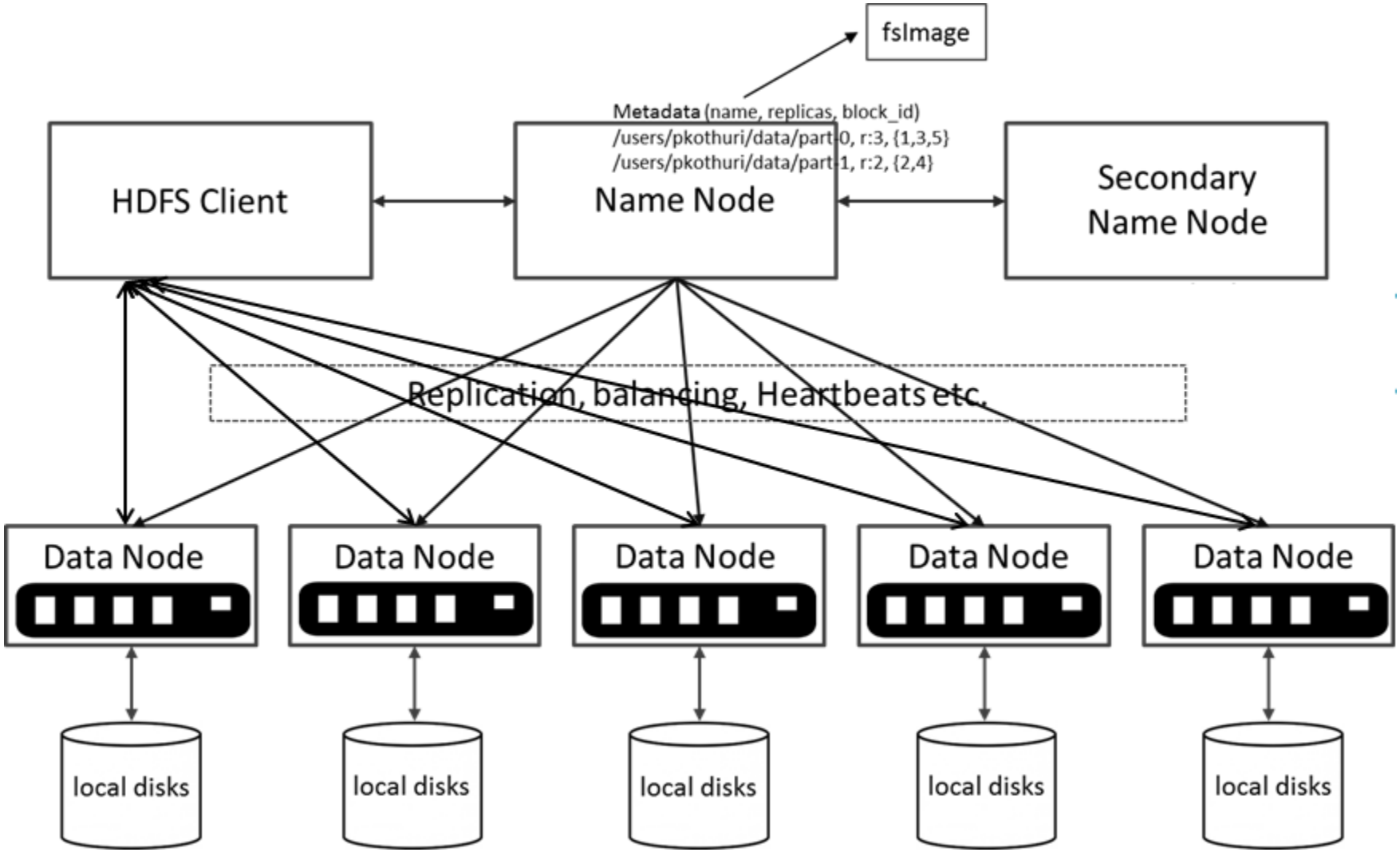
Hadoop: HDFS: Goals

- HDFS is **not** suited for:
  - Low-latency data access (HBase is a better option)
  - Lots of small files (NameNodes hold metadata in memory)

Hadoop: HDFS

- What is HDFS
  - Tailored to needs of MapReduce
  - Targeted towards many reads of filestreams
  - Writes are more costly
  - Open Data Format
  - Flexible Schema
  - High throughput
  - Fault Tolerance
  - High degree of data replication (3x by default)
  - No need for RAID on normal nodes
  - Large blocksize (64MB)
  - Location awareness of DataNodes in network

Hadoop: HDFS


fsImage
Metadata (name, replicas, block_id)
/users/pkothuri/data/part-0, r:3, {1,3,5}
/users/pkothuri/data/part-1, r:2, {2,4}
HDFS Client
Name Node
Secondary Name Node
Replication, balancing, Heartbeats etc.
Data Node
Data Node
Data Node
Data Node
Data Node
local disks
local disks
local disks
local disks
local disks

Hadoop: HDFS: NameNode

- **Managing the file system namespace**
  - Maintain the namespace tree operations like opening, closing, and renaming files and directories.
  - Determine the mapping of file blocks to DataNodes (the physical location of file data).
  - Store file metadata:
    - List of files
    - List of Blocks for each file
    - List of DataNodes for each block
    - File attributes, e.g. creation time, replication factor
    - Transaction logs (EditLog): records file creations, file deletions etc
- **Coordinating file operations**
  - Directs clients to DataNodes for reads and writes
  - No data is moved through the NameNode
- **Maintaining overall health**
  - Collect block reports and heartbeats from DataNodes
  - Block re-replication and rebalancing
  - Garbage collection

Hadoop: HDFS: DataNodes

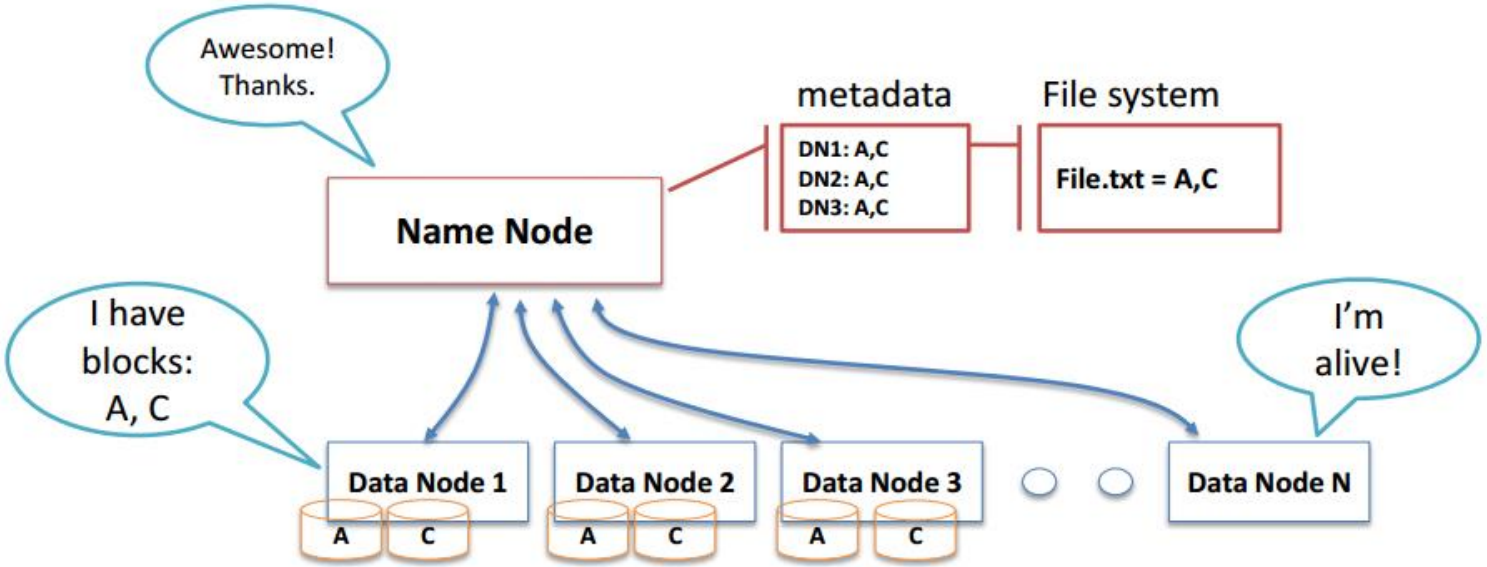
- Responsible for serving read and write requests from the file system's clients
- Perform block creation, deletion, and replication upon instruction from the NameNode
- Stores the actual data in HDFS
- Can run on any underlying filesystem (ext3/4, NTFS, etc)
- NameNode replicates blocks 2x in local rack, 1x elsewhere
- Periodically sends a report of all existing blocks to the NameNode (Blockreport)
- Facilitates pipelining of data
  - Forwards data to other specified DataNodes

Hadoop: HDFS: Communication

- Communication between NameNode and DataDode
  - The Namenode receives a **Heartbeat** and a **BlockReport** from each DataNode in the cluster periodically
  - Heartbeats
    - DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available
    - The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them
  - Blockreports
    - A Blockreport contains a list of all blocks on a DataNode

Hadoop: HDFS: Communication

- Communication between NameNode and DataDode
  - TCP – every 3 seconds a Heartbeat
  - Every 10th heartbeat is a Blockreport
  - Name Node builds metadata from Blockreports
  - If Name Node is down, HDFS is down



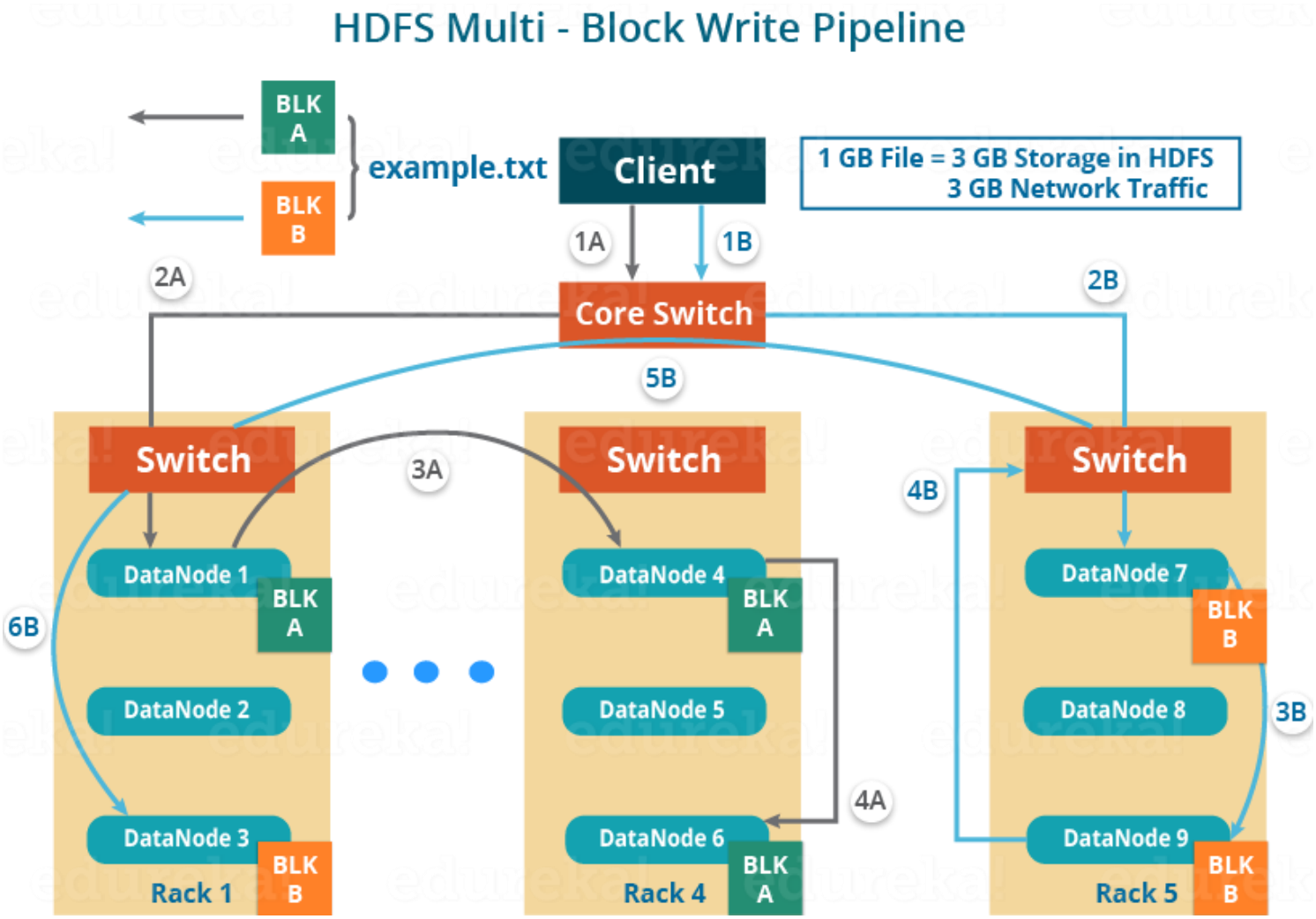
Hadoop: HDFS: Secondary NameNode

- helps to overcome the above issues by taking over responsibility of merging EditLogs with FsImage from the NameNode.
  - It gets the **EditLogs** from the NameNode periodically and applies to **FsImage**
  - Once it has new FsImage, it copies back to NameNode
  - NameNode will use this FsImage for the next restart, which will reduce the startup time

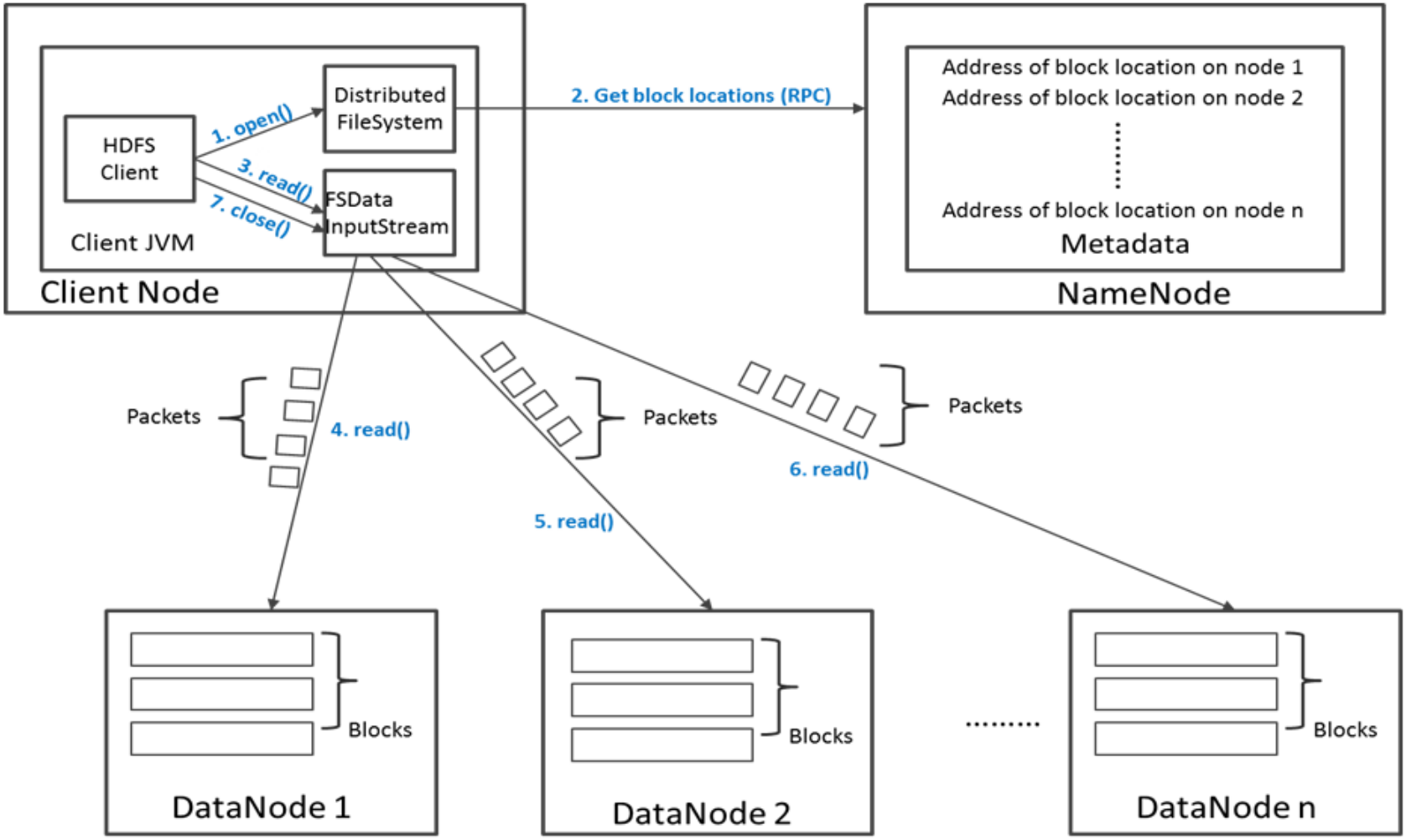
Hadoop: HDFS: Replication

- One replica on local node
- Second replica on a remote rack
- Third replica on same remote rack
- Additional replicas are randomly placed
- Clients read from nearest replica

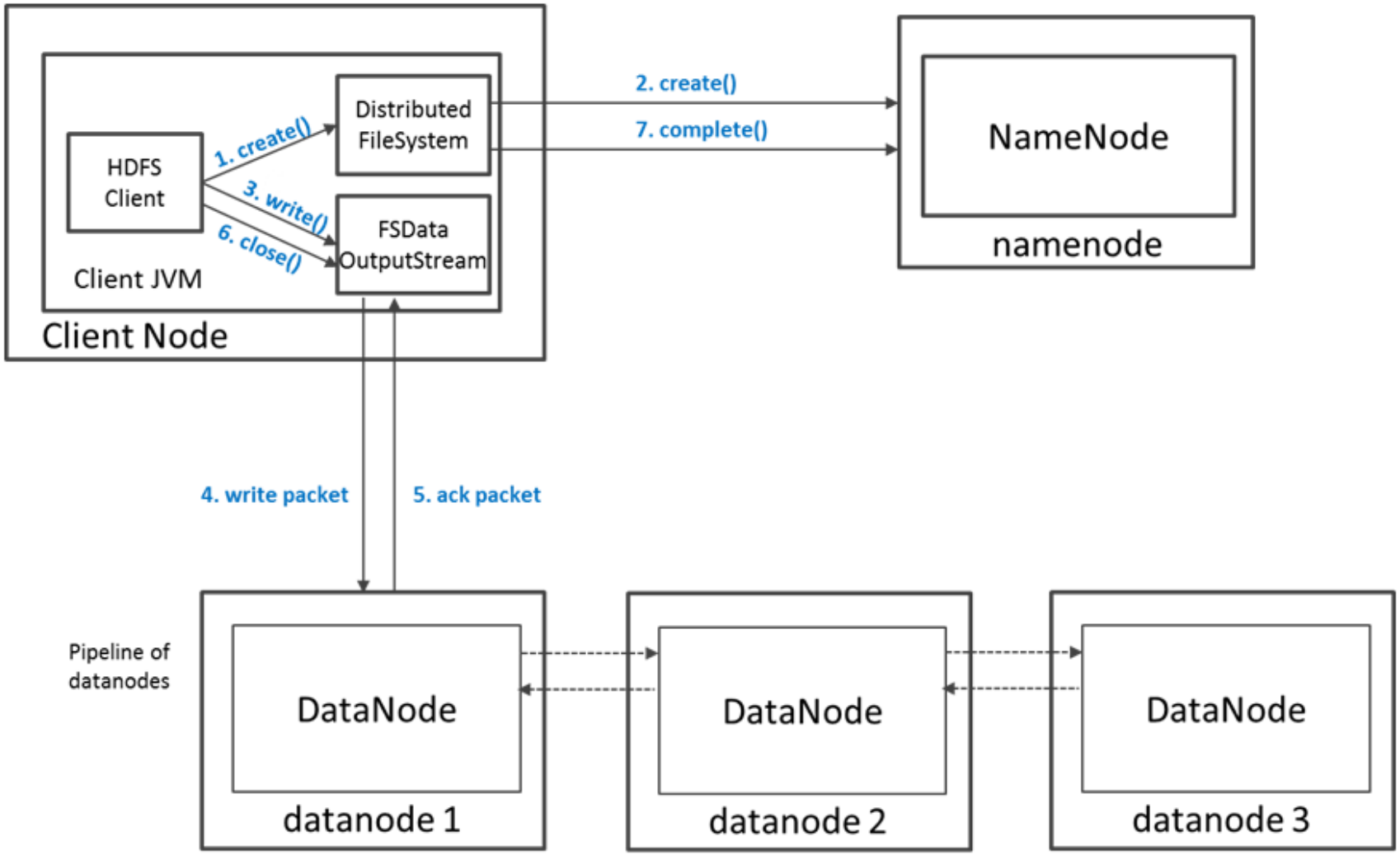
Hadoop: HDFS: Replication



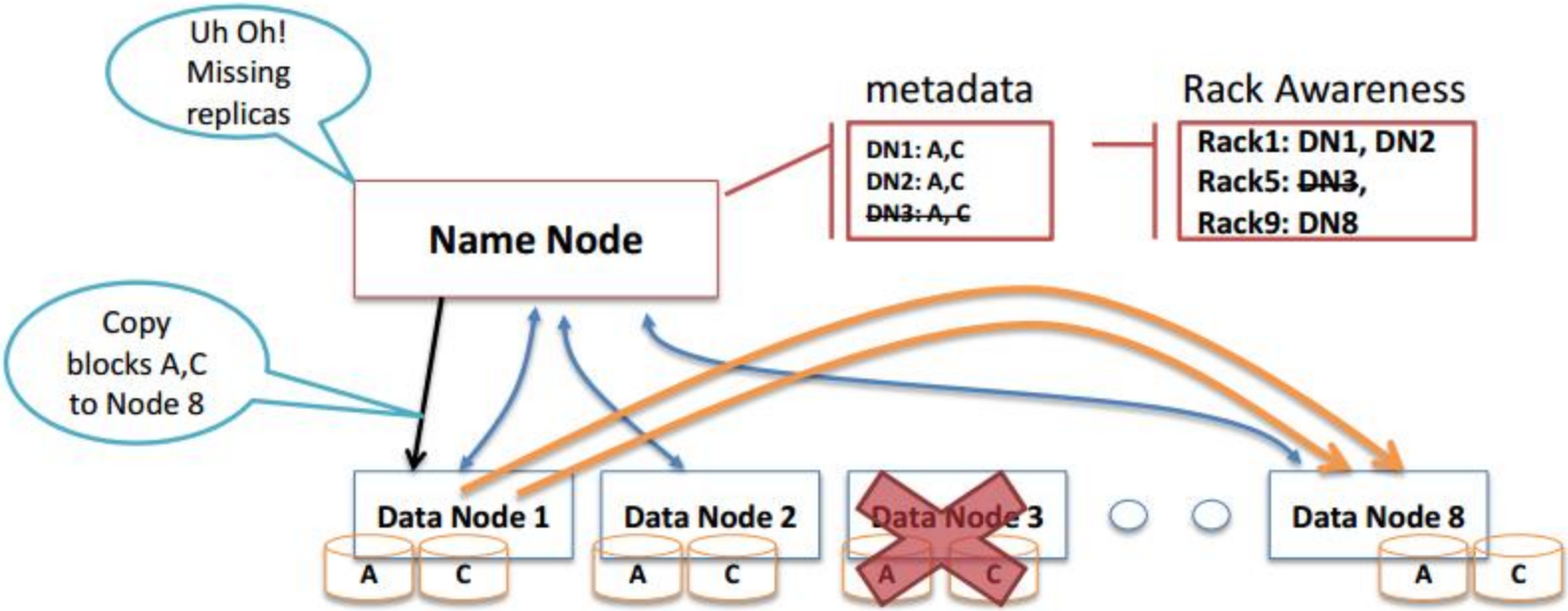
Hadoop: HDFS: File Read Data Flow



Hadoop: HDFS: File Write Data Flow



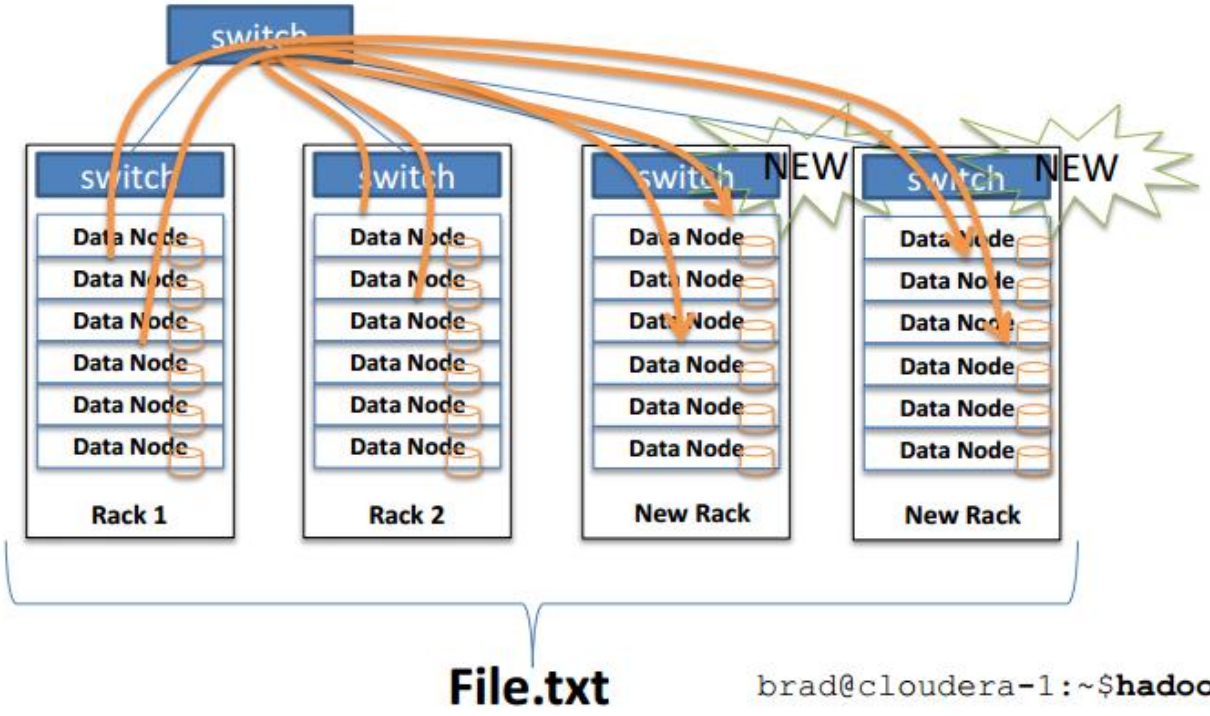
Hadoop: HDFS: Replication



- NameNode detects DataNode failures
  - Missing Heartbeats signify lost Nodes
  - NameNode consults metadata, finds affected data
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

Hadoop: HDFS: Cluster Rebalancing

- Goal: % disk full on DataNodes should be similar
  - Usually run when new DataNodes are added
  - Rebalancer is throttled to avoid network congestion
  - Does not interfere with MapReduce or HDFS
  - Command line tool



Hadoop: HDFS: Fault tolerance

- Failure is the norm rather than exception
- A HDFS instance may consist of thousands of server machines, each storing part of the file system's data.
- Since we have huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Hadoop: HDFS: Unique features

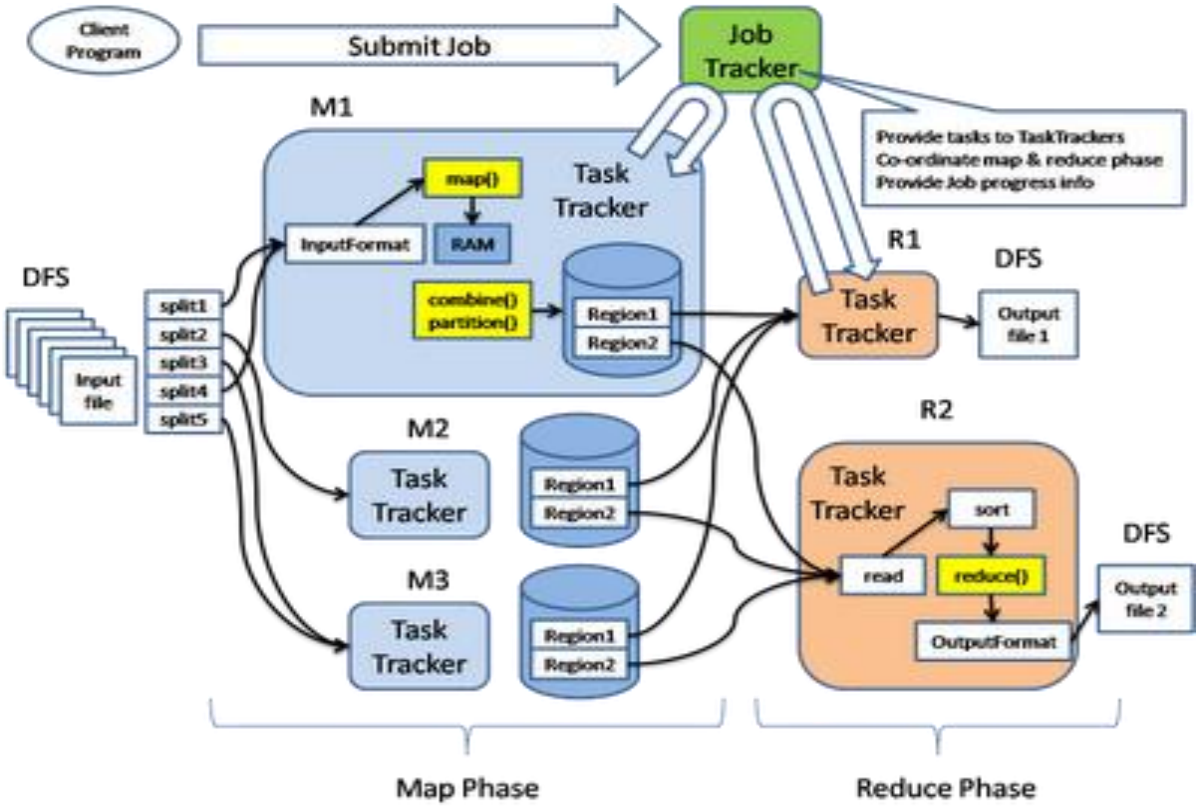
- HDFS has a bunch of unique features that make it ideal for distributed systems:
  - **Failure tolerant** - data is duplicated across multiple DataNodes to protect against machine failures. The default is a replication factor of 3 (every block is stored on three machines).
  - **Scalability** - data transfers happen directly with the DataNodes so your read/write capacity scales fairly well with the number of DataNodes
  - **Space** - need more disk space? Just add more DataNodes and re-balance
  - **Industry standard** - Other distributed applications are built on top of HDFS (HBase, Map-Reduce)
- HDFS is designed to process large data sets with **write-once-read-many** semantics, it is **not for low latency access**

Hadoop: MapReduce

Hadoop: MapReduce

- A MapReduce Process (org.apache.hadoop.mapred)
  - JobClient
    - Submit job
  - JobTracker
    - Manage and schedule job, split job into tasks;
    - Splits up data into smaller tasks(“Map”) and sends it to the TaskTracker process in each node
  - TaskTracker
    - Start and monitor the task execution;
    - reports back to the JobTracker node and reports on job progress, sends data (“Reduce”) or requests new jobs
  - Child
    - The process that really executes the task
- Architecture
  - Distributed, with some centralization
  - Main nodes of cluster are where most of the computational power and storage of the system lies
  - Main nodes run TaskTracker to accept and reply to MapReduce tasks, Main Nodes run DataNode to store needed blocks closely as possible
  - Central control node runs NameNode to keep track of HDFS directories & files, and JobTracker to dispatch compute tasks to TaskTracker
  - Written in Java, also supports Python and Ruby

Hadoop: MapReduce



Hadoop: MapReduce

- Task Granularity
  - The map phase has *M* pieces and the reduce phase has *R* pieces.
  - *M* and *R* should be much larger than the number of worker machines.
  - Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails.
  - Larger the *M* and *R*, more the decisions the master must make
  - *R* is often constrained by users because the output of each reduce task ends up in a separate output file.
  - Typically, (at Google), *M* = 200,000 and *R* = 5,000, using 2,000 worker machines.

Hadoop: MapReduce

- Additional support functions
  - Partitioning function
    - The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. .hash(key) mod R.). In some cases, it may be useful to partition data by some other function of the key. The user of the MapReduce library can provide a special partitioning function.
  - Combiner function
    - User can specify a Combiner function that does partial merging of the intermediate local disk data before it is sent over the network. The Combiner function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions.

Hadoop: HBase

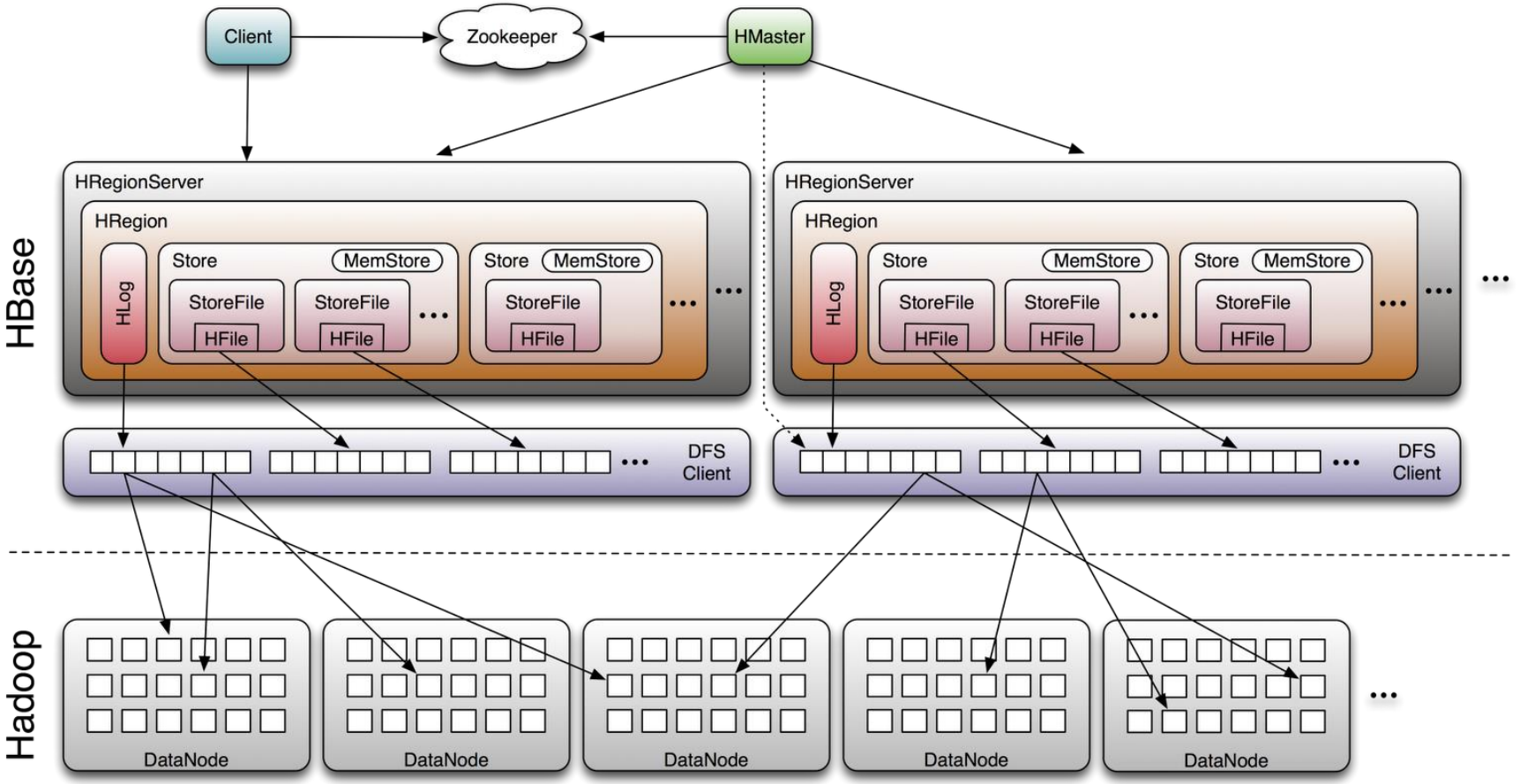
Hadoop: HBase: Introduction

- HBase is an open source, non-relational, distributed database modeled.
- HBase is a clone of Google's BigTable and is written in Java
- It is developed as part of Apache Software Foundation's Apache Hadoop project and runs on top of , providing BigTable-like capabilities for Hadoop.
- It has a Column oriented semi-structured data store.
- That is, it provides a fault-tolerant way of storing large quantities of sparse data and also provides strong consistancy.

Hadoop: HBase

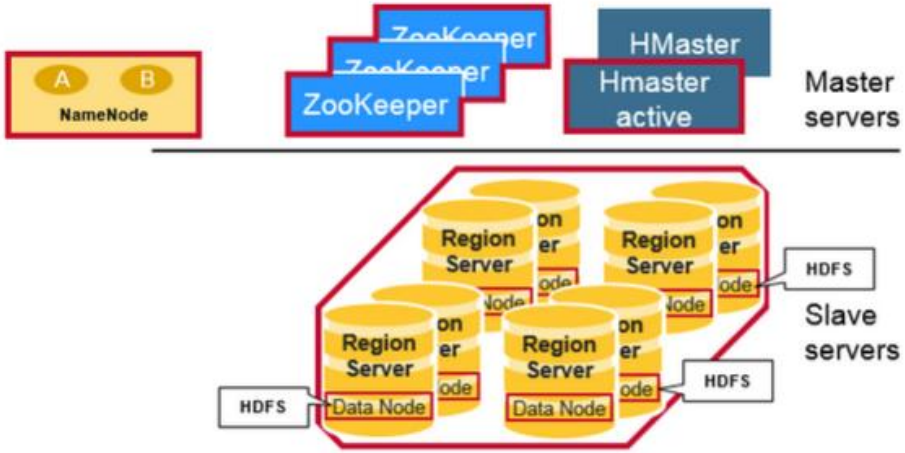
- Applications
  - Facebook's Messaging Platform is built using HBase.
  - Twitter runs HBase across its entire Hadoop cluster.
  - Yahoo! Uses HBase to store document fingerprints for detecting duplicates.

Hadoop: HBase: Architecture



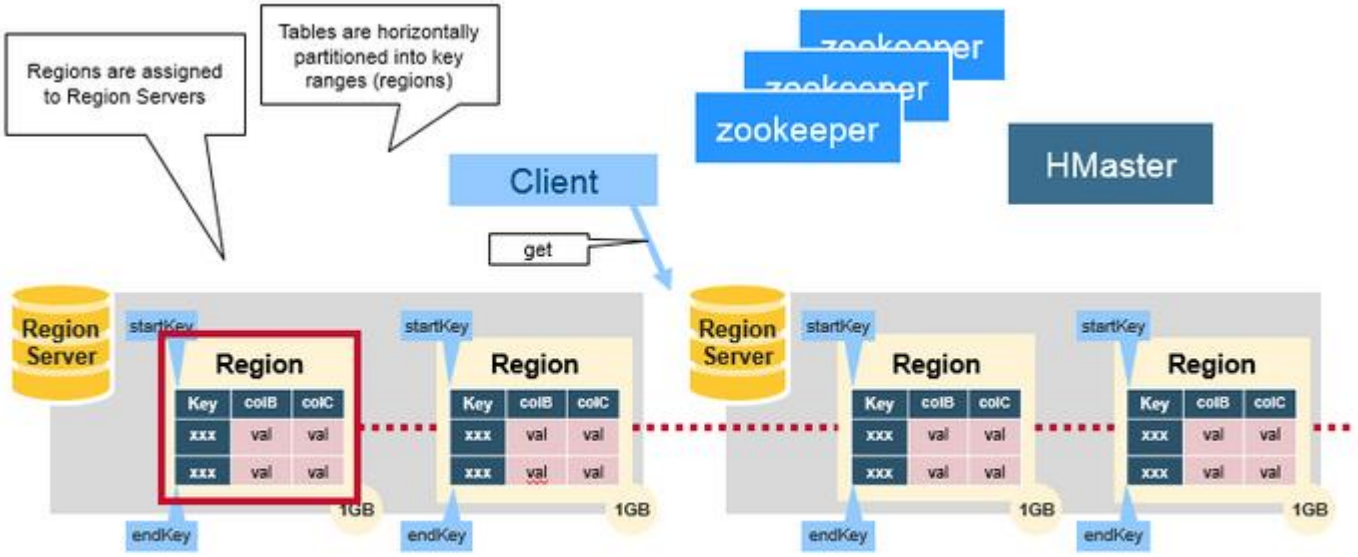
Hadoop: HBase: Components

- **Region Servers**: serve data for reads and writes. When accessing data, clients communicate with HBase RegionServers directly.
- **HBase HMaster**: Region assignment, DDL (create, delete tables) operations are handled by the HBase Master process.
- **Zookeeper**: (which is part of HDFS) maintains a live cluster state.



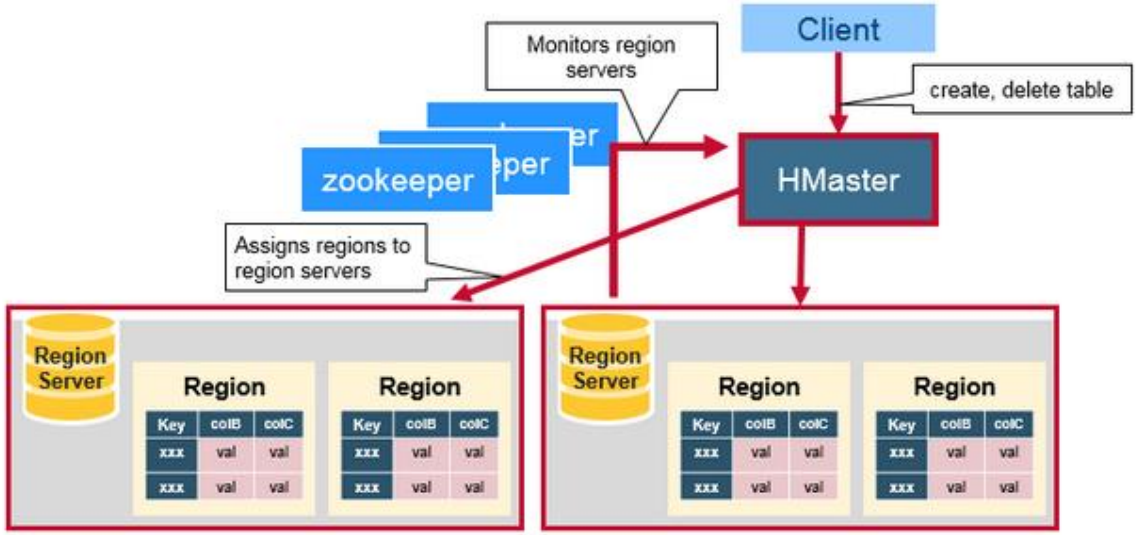
Hadoop: HBase: Regions

- HBase Tables are divided horizontally by row key range into "**Regions**". A region contains all rows in the table between the region's start key and end key. Regions are assigned to the nodes in the cluster, called "**Region Servers**", and these serve data for reads and writes. A region server can serve about 1,000 regions.



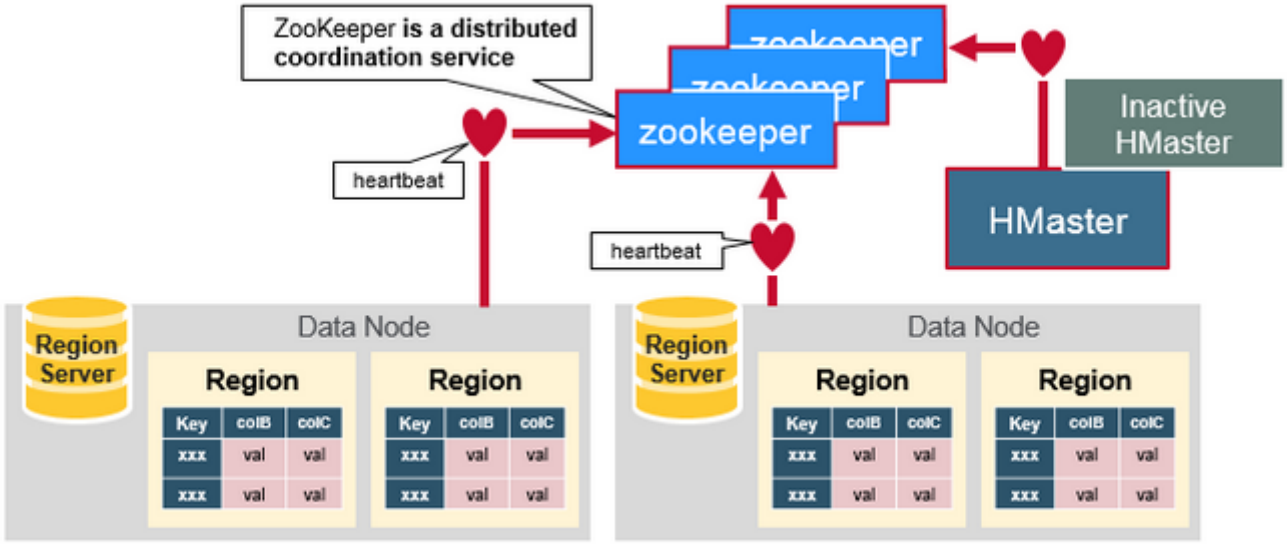
Hadoop: HBase: HMaster

- Coordinating the region servers
  - Assigning regions on startup , re-assigning regions for recovery or load balancing
  - Monitoring all RegionServer instances in the cluster (listens for notifications from zookeeper)
- Admin functions
  - Interface for creating, deleting, updating tables



Hadoop: HBase: ZooKeeper

- The Coordinator
  - A distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available, and provides server failure notification. Zookeeper uses consensus to guarantee common shared state.

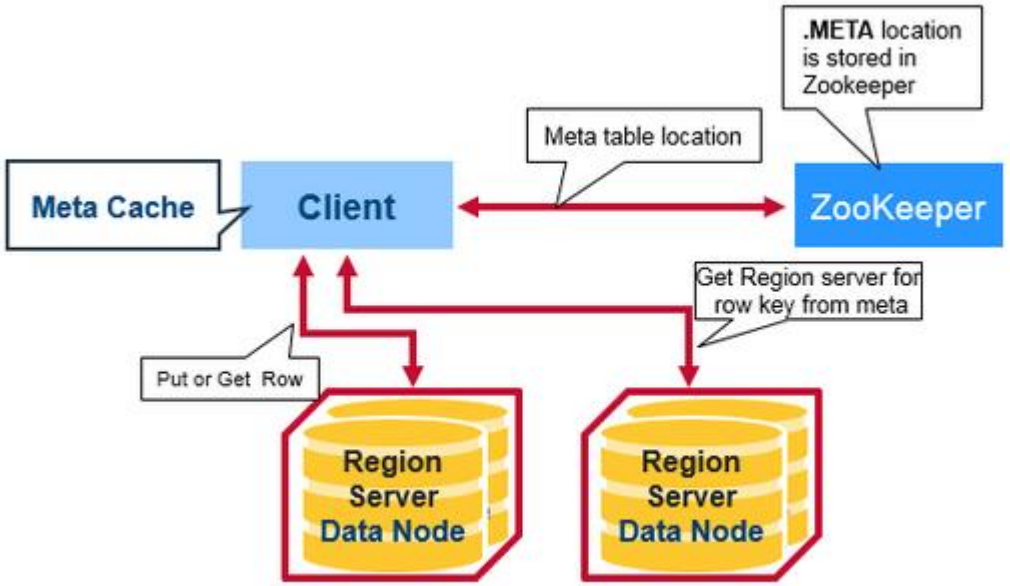


Hadoop: HBase

- First read or write
  - There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster. ZooKeeper stores the location of the META table.
  - This is what happens the first time a client reads or writes to HBase:
    - The client gets the Region server that hosts the META table from ZooKeeper.
    - The client will query the .META. server to get the region server corresponding to the row key it wants to access. The client caches this information along with the META table location.
    - It will get the Row from the corresponding Region Server.

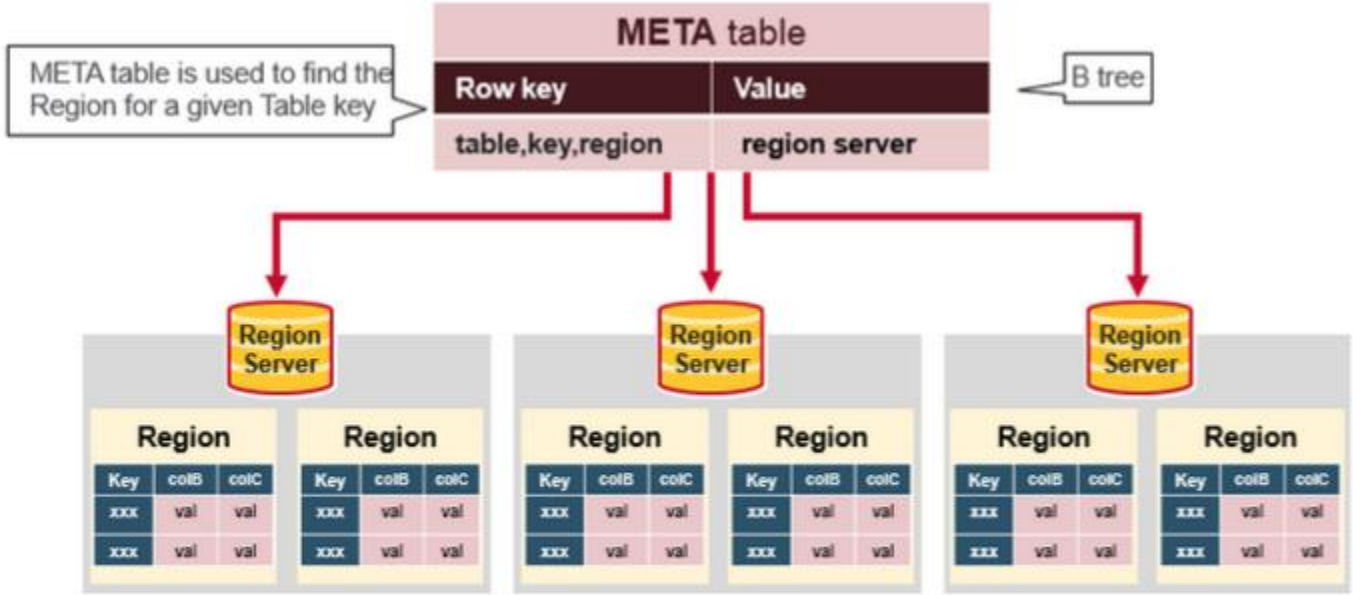
Hadoop: HBase

- Future reads
  - The client uses the cache to retrieve the META location and previously read row keys. Over time, it does not need to query the META table, unless there is a miss because a region has moved; then it will re-query and update the cache.



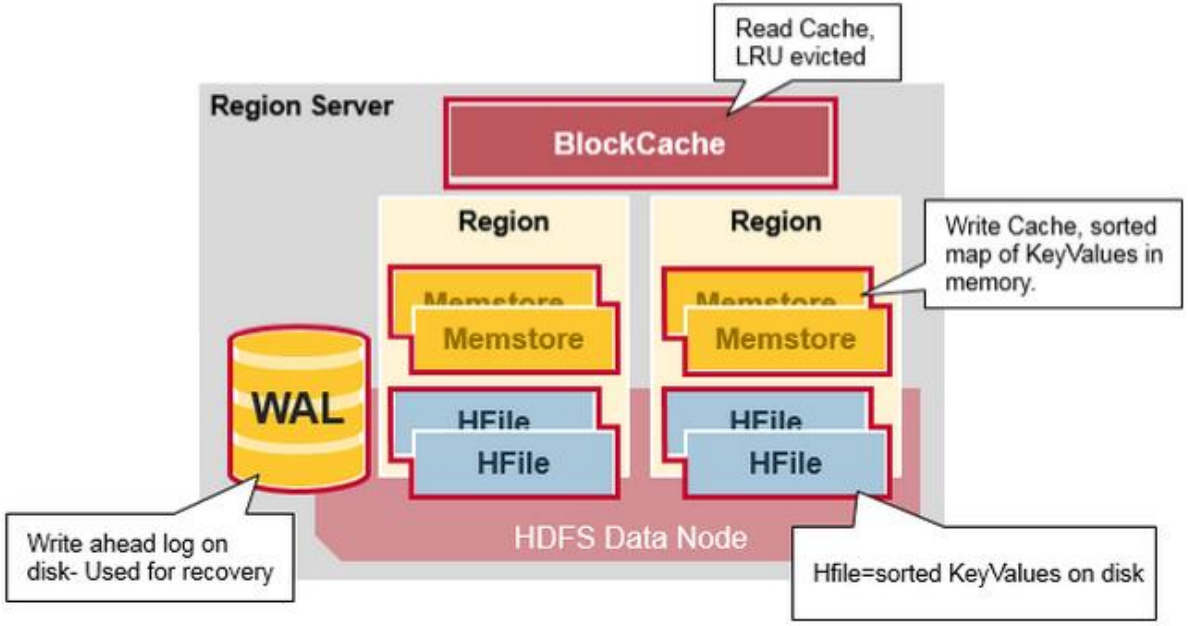
Hadoop: HBase: Meta Table

- an HBase table that keeps a list of all regions in the system.
- The .META. table is like a B-tree.
- The .META. table structure is as follows:
  - Key: region start key, region id
  - Values: RegionServer



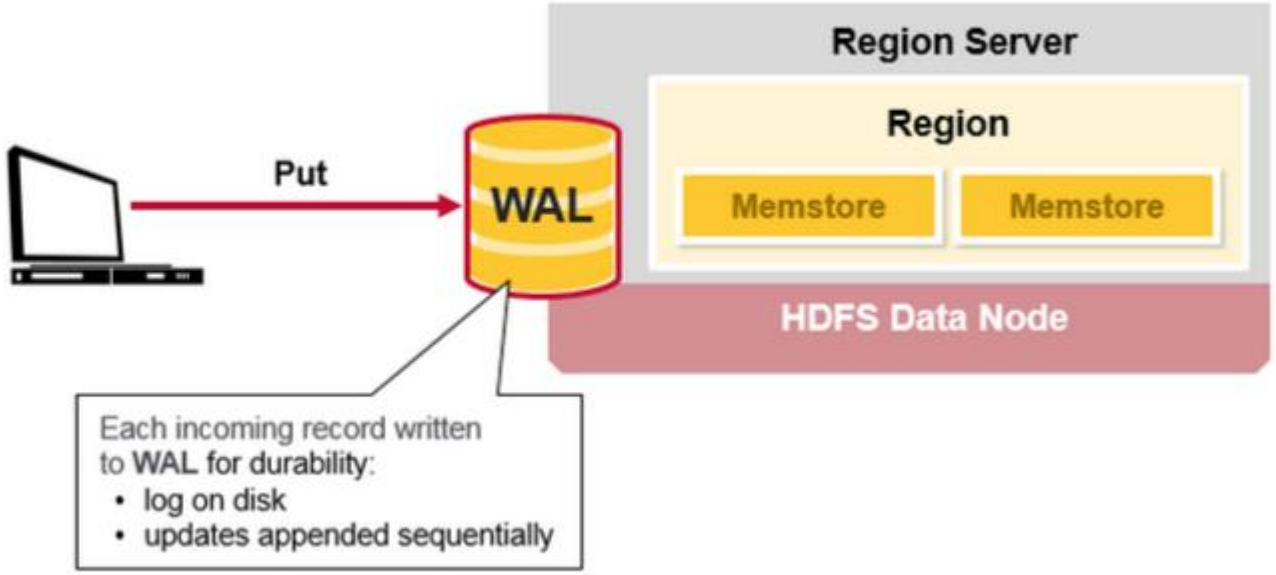
Hadoop: HBase

- Region Server Components
  - **WAL** (Write Ahead Log): used to store new data that hasn't yet been persisted to permanent storage; it is used for recovery in the case of failure.
  - **BlockCache**: is the read cache. It stores frequently read data in memory. Least recently used data is evicted when full.
  - **MemStore**: is the write cache. It stores new data which has not yet been written to disk. There is one MemStore per column family per region.
  - **Hfiles**: store the rows as sorted KevValues on disk.



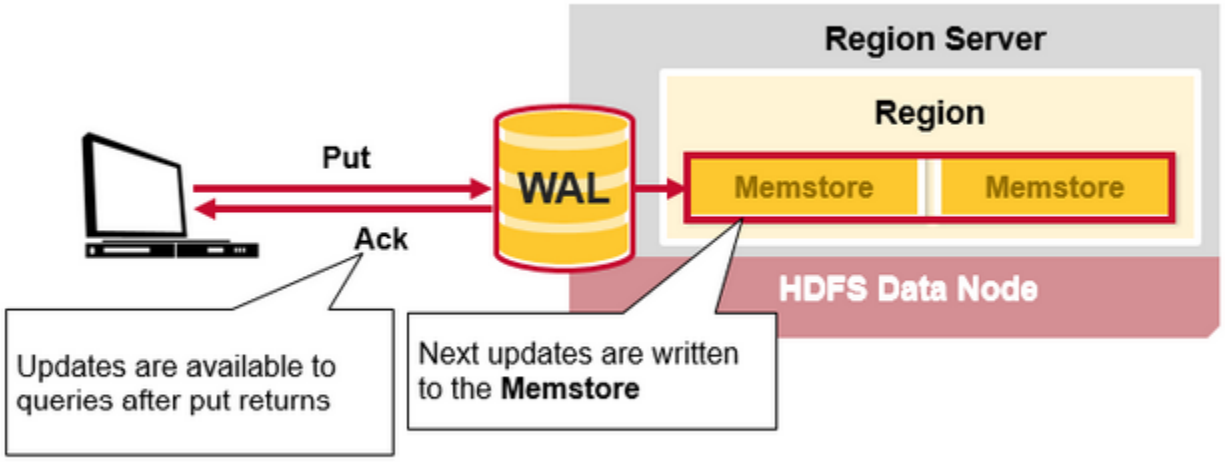
Hadoop: HBase

- HBase Write Step 1
  - When the client issues a Put request, the first step is to write the data to the write-ahead log, the WAL:
    - Edits are appended to the end of the WAL file that is stored on disk.
    - The WAL is used to recover not-yet-persisted data in case a server crashes.



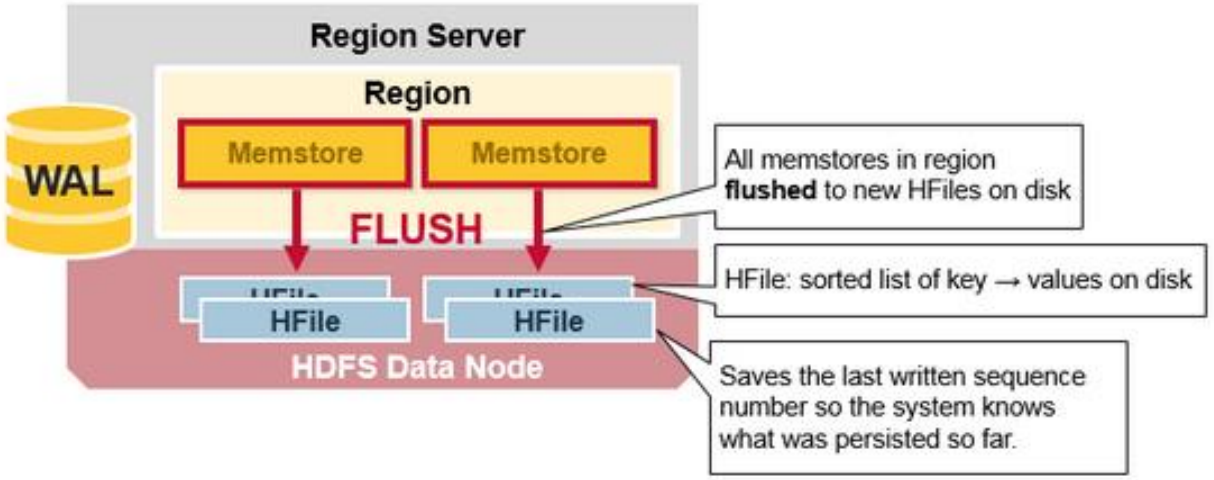
Hadoop: HBase

- HBase Write Step 2
  - Once the data is written to the WAL, it is placed in the MemStore. Then, the put request acknowledgement returns to the client.



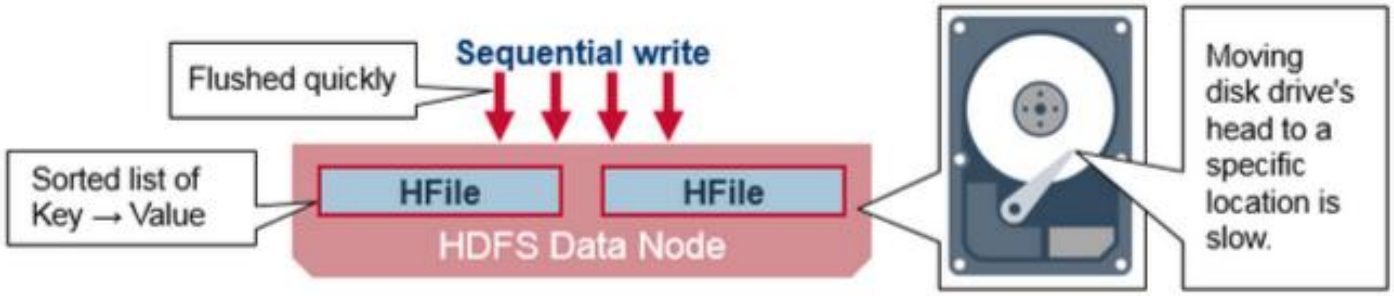
Hadoop: HBase

- HBase Region Flush



Hadoop: HBase: HFile

- Data is stored in an HFile which contains sorted key/values. When the MemStore accumulates enough data, the entire sorted KeyValue set is written to a new HFile in HDFS. This is a sequential write. It is very fast, as it avoids moving the disk drive head.

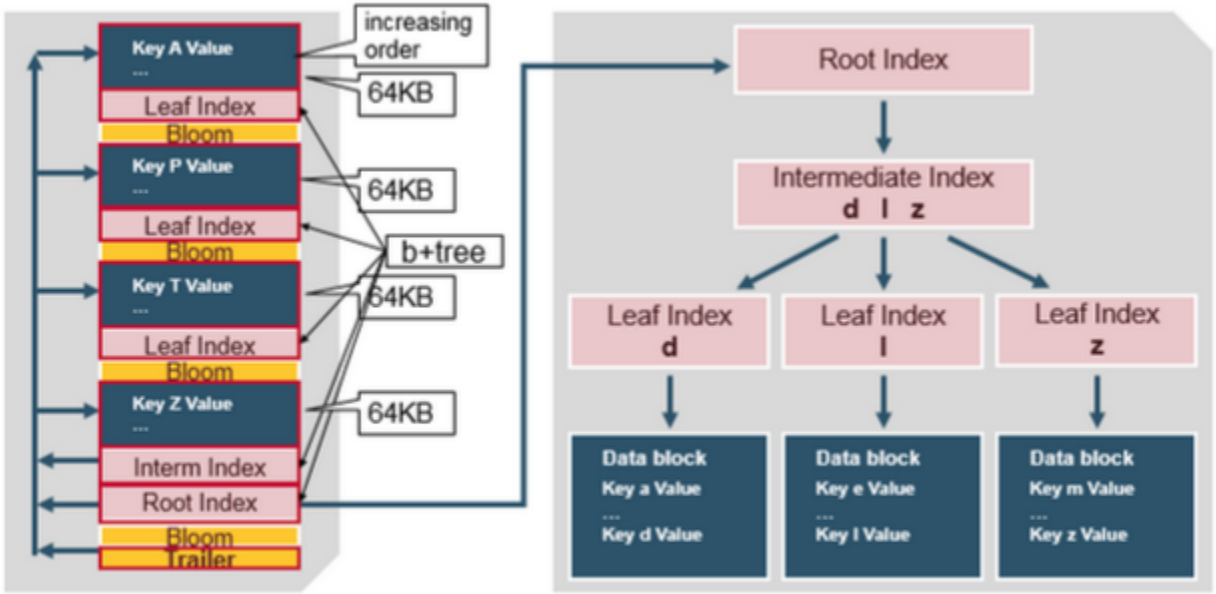


Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

Hadoop: HBase

- HBase HFile Structure



Hadoop: HBase

- Region = Contiguous Keys
  - Let's do a quick review of regions:
    - A table can be divided horizontally into one or more regions. A region contains a contiguous, sorted range of rows between a start key and an end key
    - Each region is 1GB in size (default)
    - A region of a table is served to the client by a RegionServer
    - A region server can serve about 1,000 regions (which may belong to the same table or different tables)

Hadoop: HBase

- Region Split
  - Initially there is one region per table. When a region grows too large, it splits into two child regions. Both child regions, representing one-half of the original region, are opened in parallel on the same Region server, and then the split is reported to the HMaster. For load balancing reasons, the HMaster may schedule for new regions to be moved off to other servers.

Hadoop: HBase

- Read Load Balancing
  - Splitting happens initially on the same region server, but for load balancing reasons, the HMaster may schedule for new regions to be moved off to other servers. This results in the new Region server serving data from a remote HDFS node until a major compaction moves the data files to the Regions server's local node. HBase data is local when it is written, but when a region is moved (for load balancing or recovery), it is not local until major compaction.

Hadoop: HBase

- HBase Crash Recovery
  - When a RegionServer fails, Crashed Regions are unavailable until detection and recovery steps have happened. Zookeeper will determine Node failure when it loses region server heart beats. The HMaster will then be notified that the Region Server has failed.
  - When the HMaster detects that a region server has crashed, the HMaster reassigns the regions from the crashed server to active Region servers. In order to recover the crashed region server's memstore edits that were not flushed to disk. The HMaster splits the WAL belonging to the crashed region server into separate files and stores these file in the new region servers' data nodes. Each Region Server then replays the WAL from the respective split WAL, to rebuild the memstore for that region.

Hadoop: HBase

- Data Recovery
  - WAL files contain a list of edits, with one edit representing a single put or delete. Edits are written chronologically, so, for persistence, additions are appended to the end of the WAL file that is stored on disk.
  - What happens if there is a failure when the data is still in memory and not persisted to an HFile? The WAL is replayed. Replaying a WAL is done by reading the WAL, adding and sorting the contained edits to the current MemStore. At the end, the MemStore is flush to write changes to an HFile.

Hadoop: HBase

- Architecture Benefits
  - Strong consistency model
    - When a write returns, all readers will see same value
  - Scales automatically
    - Regions split when data grows too large
    - Uses HDFS to spread and replicate data
  - Built-in recovery
    - Using Write Ahead Log (similar to journaling on file system)
  - Integrated with Hadoop
    - MapReduce on HBase is straightforward

Hadoop: HBase

- Column oriented vs. row oriented

Row-Oriented Database	Column-Oriented Database
It is suitable for Online Transaction Process (OLTP).	It is suitable for Online Analytical Processing (OLAP).
Such databases are designed for small number of rows and columns.	Column-oriented databases are designed for huge tables.

Hadoop: HBase

- HBase vs. RDBMS

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

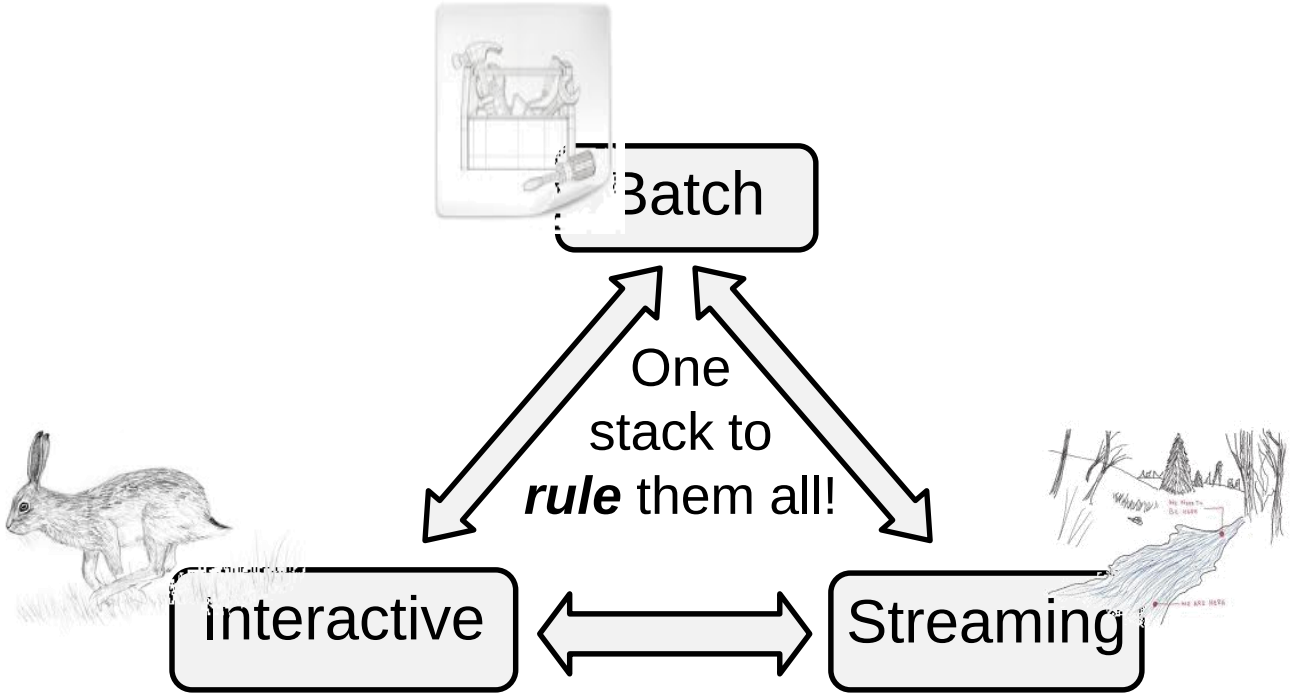
Hadoop: HBase

- Data operations
  - **Get** Returns attributes for a specific row
  - **Put** Add new rows to a table or updates existing rows.
  - **Scans** Allows iteration over multiple rows for specified attributes.
  - **Delete** Removes a row from the table

Spark

Spark: Motivation

- Easy to combine batch, streaming, and interactive computations
- Easy to develop sophisticated algorithms
- Compatible with existing open source ecosystem (Hadoop/HDFS)

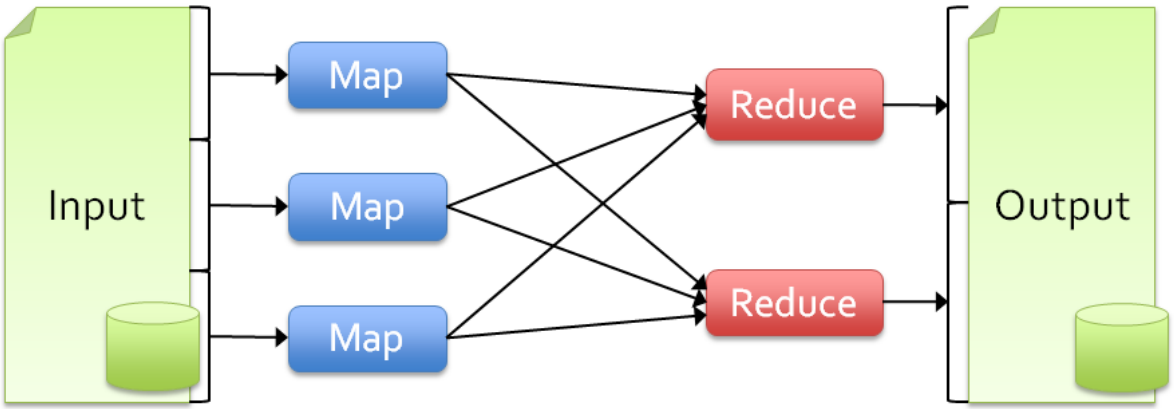


Spark: Motivation

- MapReduce greatly simplified big data analysis on large, unreliable clusters. It is great at one-pass computation.
- But as soon as it got popular, users wanted more:
  - More complex, multi-pass analytics (e.g. ML, graph)
  - More interactive ad-hoc queries
  - More real-time stream processing
- All 3 need faster data sharing across parallel jobs
  - One reaction: specialized models for some of these apps, e.g.,
    - Pregel (graph processing)
    - Storm (stream processing)

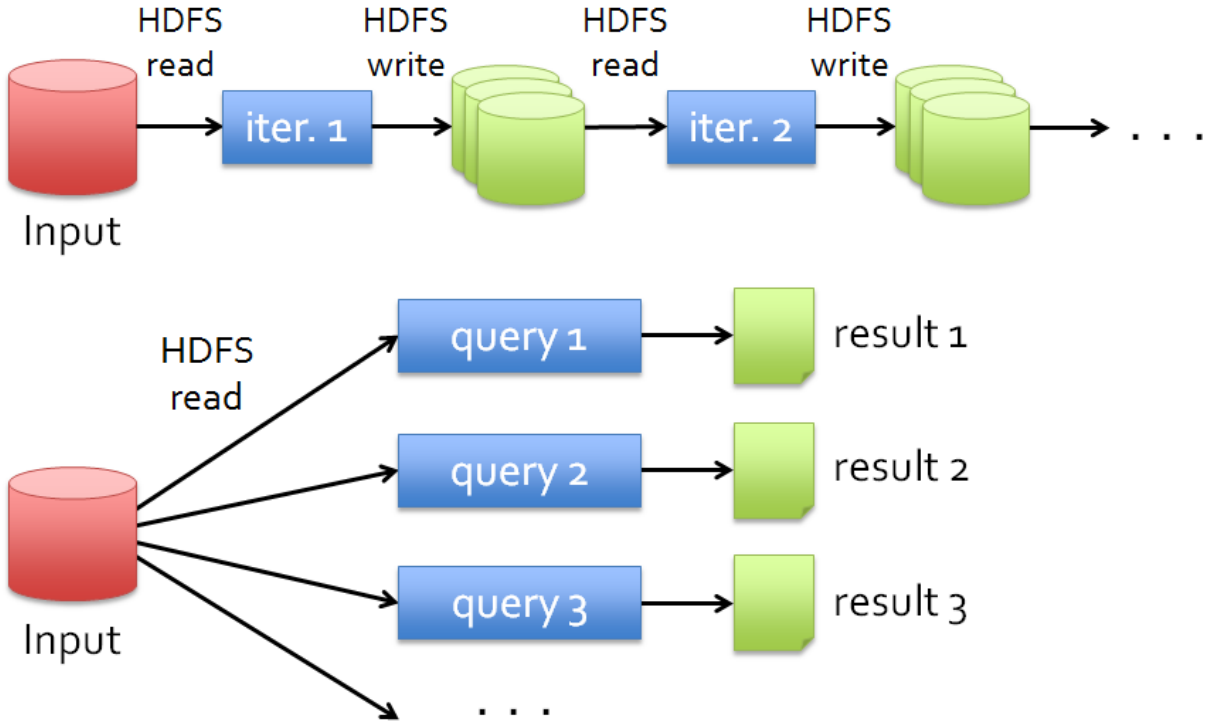
Spark: Limitations of MR

- As a general programming model:
  - It is more suitable for one-pass computation on a large dataset
  - Hard to compose and nest multiple operations
  - No means of expressing iterative operations
- As implemented in Hadoop
  - All datasets are read from disk, then stored back on to disk
  - All data is (usually) triple-replicated for reliability
  - Not easy to write MapReduce programs using Java



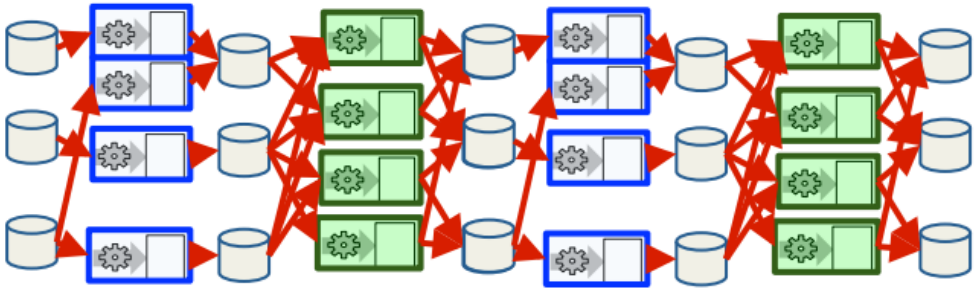
Spark: Data Sharing in MR

- Slow due to replication, serialization, and disk IO
- Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks: Efficient primitives for data sharing

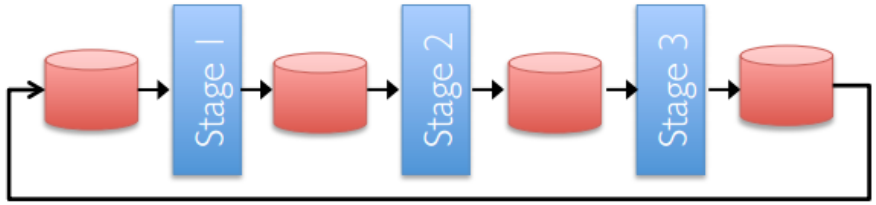


Spark: Data Sharing in MR

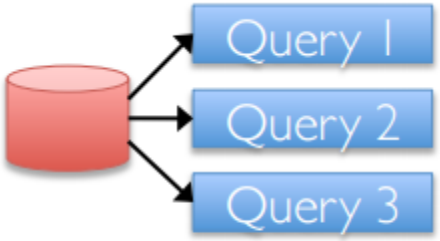
- Iterative jobs involve a lot of disk I/O for each repetition



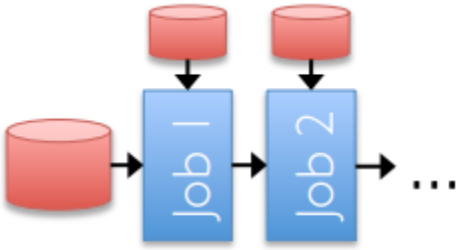
Disk I/O is very slow!



- Interactive queries and online processing involves lots of disk I/O



Interactive mining



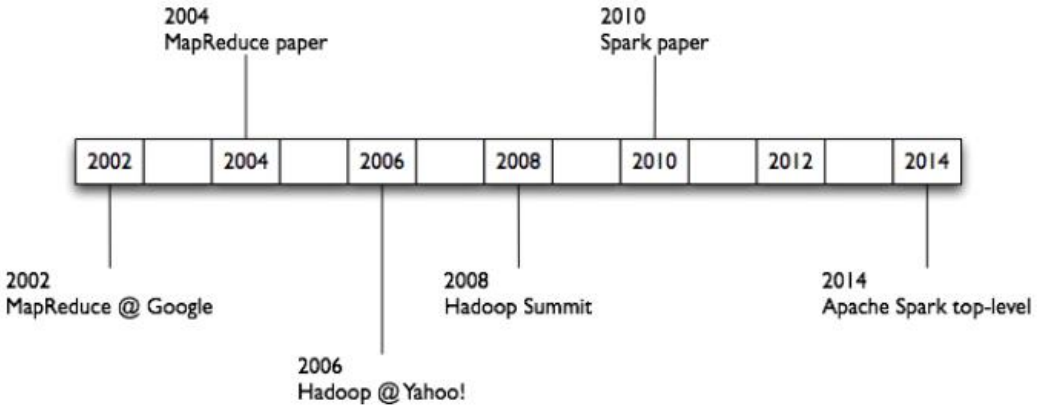
Stream processing

Spark: Goals

- Keep more data in-memory to improve the performance!
- Extend the MapReduce model to better support two common classes of analytics apps:
  - Iterative algorithms (machine learning, graphs)
  - Interactive data mining
- Enhance programmability:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

Spark: What is Spark

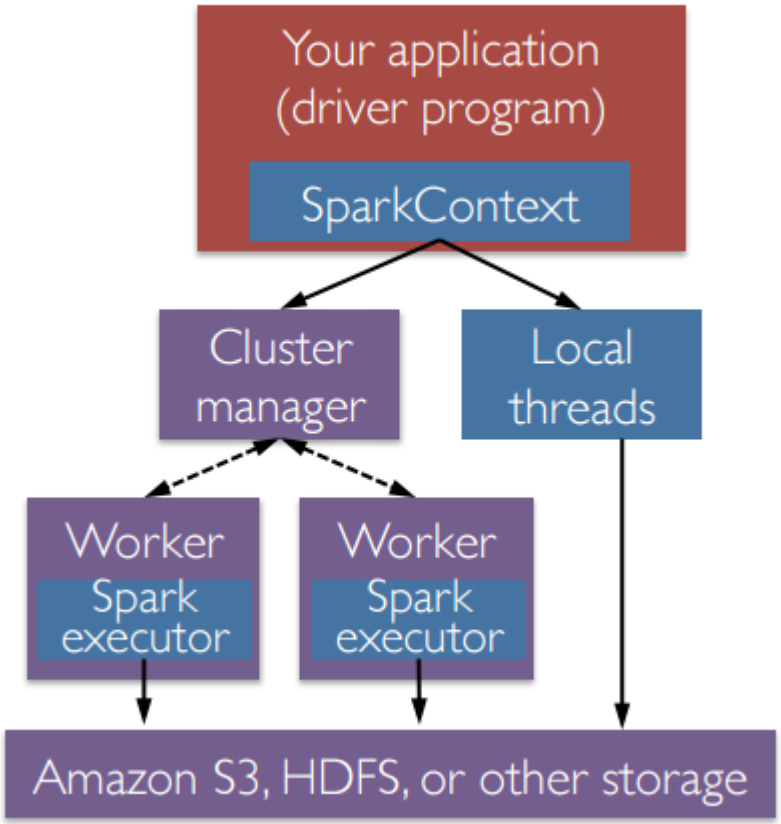
- One popular answer to “What’s beyond MapReduce?”
- Open-source engine for large-scale data processing
  - Supports generalized dataflows
  - Written in Scala, with bindings in Java and Python
- Spark is not:
  - a modified version of Hadoop
  - dependent on Hadoop because it has its own cluster management
  - Spark uses Hadoop for storage purpose only



Spark: Ideas

- Expressive computing system, not limited to map-reduce model
- Facilitate system memory
  - avoid saving intermediate results to disk
  - cache data for repetitive queries (e.g. for machine learning)
- Layer an in-memory system on top of Hadoop
- Achieve fault-tolerance by re-execution instead of replication

Spark: Components



- A Spark program first creates a SparkContext object
  - Tells Spark how and where to access a cluster
  - Connect to several types of cluster managers (e.g., YARN or its own manager)
- Cluster manager:
  - Allocate resources across applications
- Spark executor:
  - Run computations
  - Access data storage

Spark: Challenge

- Existing Systems
  - Existing in-memory storage systems have interfaces based on fine-grained updates
    - Reads and writes to cells in a table
    - E.g., databases, key-value stores, distributed memory
  - Requires replicating data or logs across nodes for fault tolerance -> expensive!
    - 10-100x slower than memory write
- How to design a distributed memory abstraction that is both fault-tolerant and efficient?

Spark: Key Solution

- Resilient Distributed Dataset (RDD)
  - Distributed collections of objects that can be cached in memory across cluster
  - Manipulated through parallel operators
  - Automatically recomputed on failure based on lineage
  - RDDs can express many parallel algorithms, and capture many current programming models
    - Data flow models: MapReduce, SQL, …
    - Specialized models for iterative apps: Pregel, …

Spark: RDD

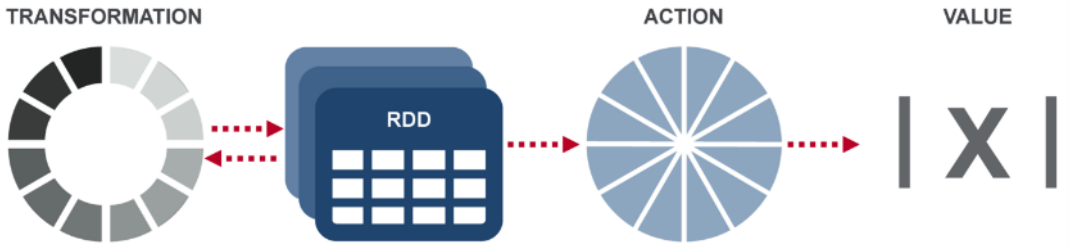
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, et al. NSDI'12
  - RDD is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.
- **Resilient**: Fault-tolerant, is able to recompute missing or damaged partitions due to node failures.
- **Distributed**: Data residing on multiple nodes in a cluster.
- **Dataset**: A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with).

Spark: Traits

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable or Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed**, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)].
- **Partitioned**, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

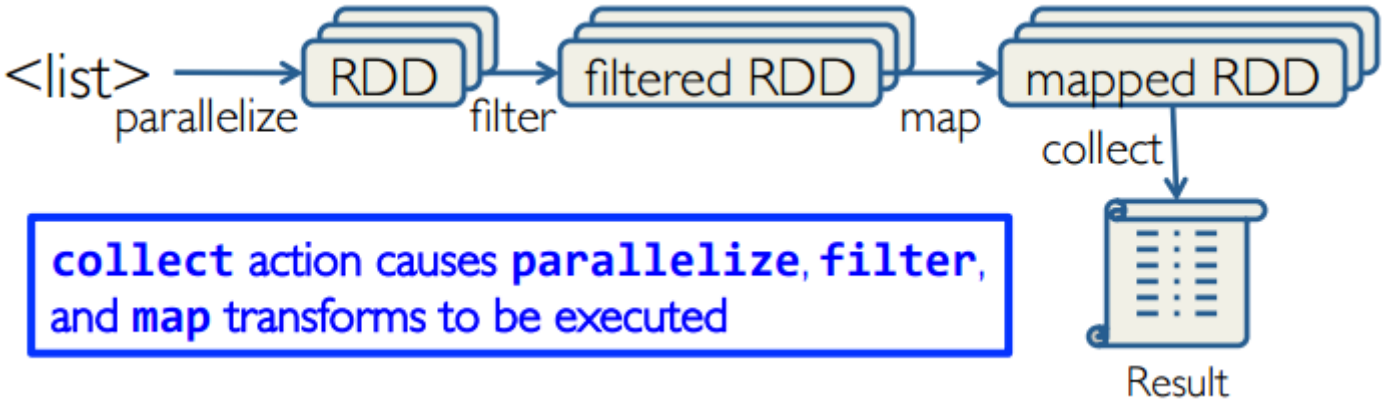
Spark: RDD Operations

- **Transformation**: returns a new RDD.
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
  - Transformation functions include map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, filter, join, etc.
- **Action**: evaluates and returns a new value.
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
  - Action operations include reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc.



Spark: Working with RDDs

- Create an RDD from a data source
  - by parallelizing existing Python collections (lists)
  - by transforming an existing RDDs
  - from files in HDFS or any other storage system
- Apply transformations to an RDD: e.g., map, filter
- Apply actions to an RDD: e.g., collect, count



- Users can control two other aspects:
  - Persistence
  - Partitioning

Spark: vs. Hadoop MapReduce

- **Performance**: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to do well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- **Ease of use**: Spark is easier to program
- **Data processing**: Spark is more general
- **Maturity**: Spark maturing, Hadoop MapReduce mature

"Spark vs. Hadoop MapReduce" by Saggi Neumann (November 24, 2014)
https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

Storm

Storm: Motivation

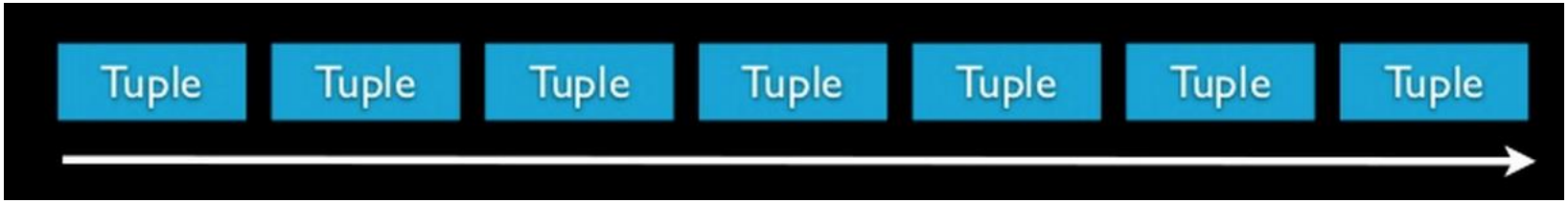
- Lots of tools for data (i.e. batch) processing: Hadoop, Pig, HBase, Hive, … But none of them are realtime systems which are becoming a real requirement for businesses
- Hadoop ?
  - For parallel batch processing : No Hacks for realtime
  - Map/Reduce is built to leverage data localization on HDFS to distribute computational jobs.
  - Works on big data.
- Storm !
  - Stream process data in realtime with no latency!
  - Generates big data!

Storm: What is Storm?

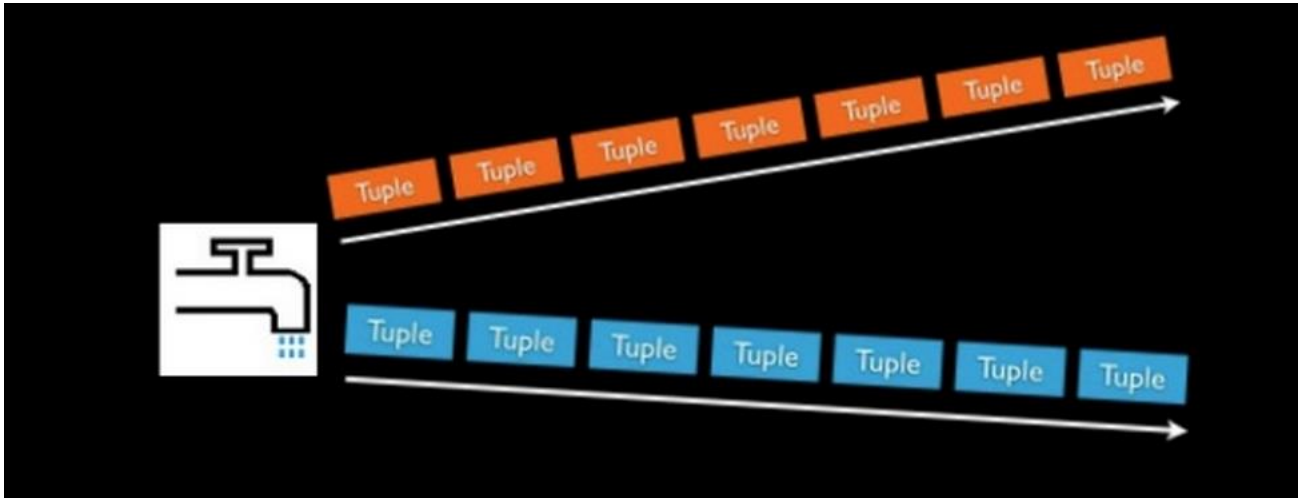
- Developed by BackType which was acquired by Twitter
- Storm provides realtime computation
  - Scalable
  - Guarantees no data loss
  - Extremely robust and fault-tolerant
  - Programming language agnostic

Storm: Stream & Spouts

- Stream
  - Unbounded sequence of tuples (storm data model)
  - <key, value(s)> pair ex. <“UIUC”, 5>

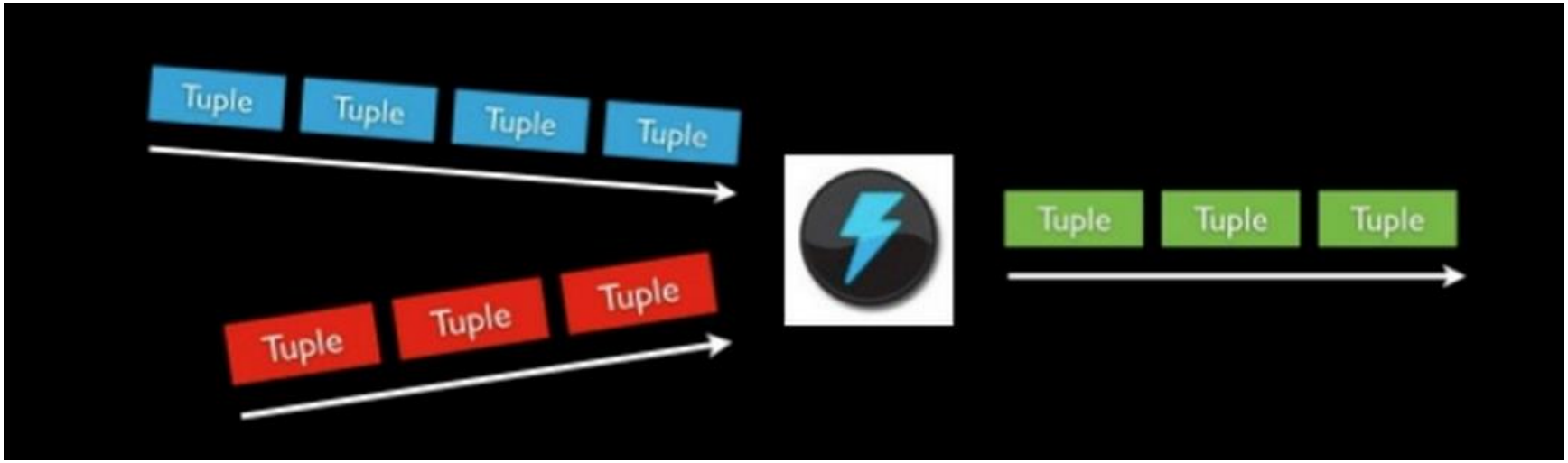


- Spouts
  - Source of streams : Twitterhose API
  - Stream of tweets or some crawler



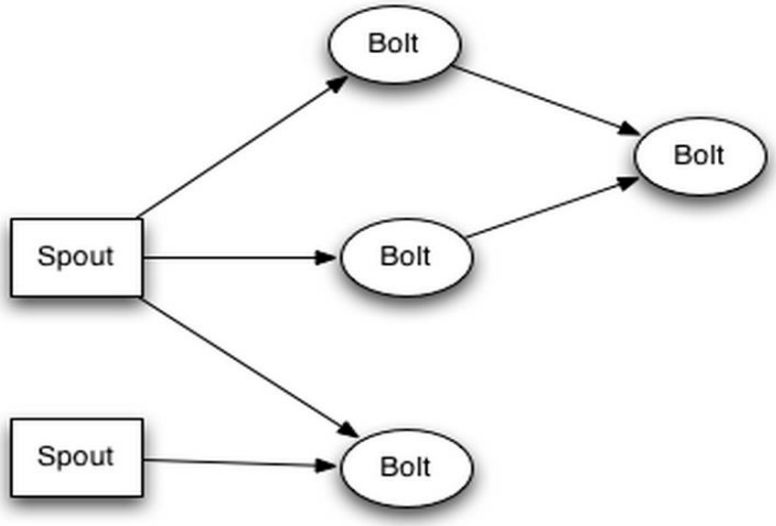
Storm: Bolts

- Process (one or more) input stream and produce new streams
- Functions: Filter, Join, Apply/Transform, *etc*.



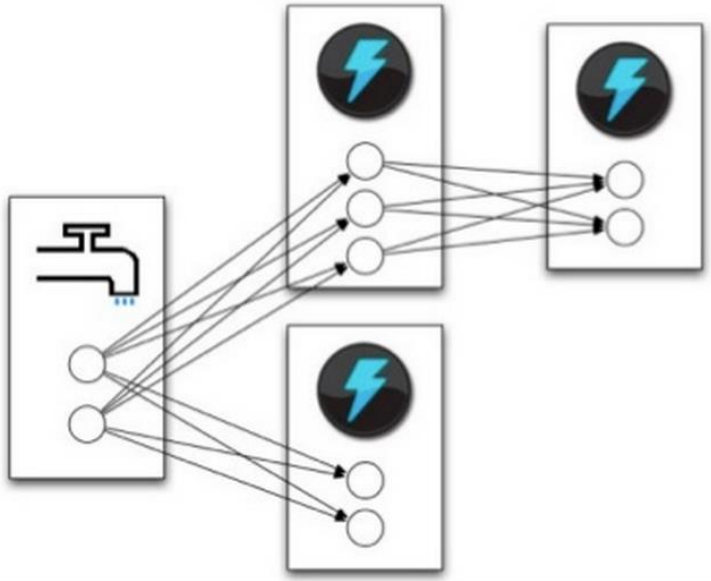
Storm: Topology

- Topology
  - Graph of computation – can have cycles
  - Network of Spouts and Bolts
  - Spouts and bolts execute as many tasks across the cluster



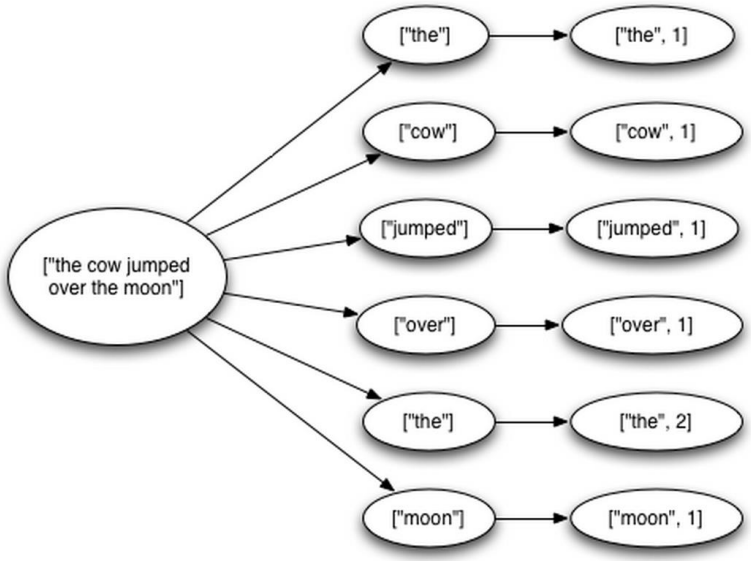
Storm: Grouping

- Shuffle Grouping
  - Distribute streams “randomly” to bolt’s tasks
- Fields Grouping
  - Group a stream by a subset of its fields
- All Grouping
  - All tasks of bolt receive all input tuples
  - Useful for joins
- Global Grouping
  - Pick task with lowest id



Storm: Message Processing

- When is a message “Fully Proceed” ?
  - "fully processed" when the tuple tree has been exhausted and every message in the tree has been processed
  - A tuple is considered failed when its tree of messages fails to be fully processed within a specified timeout.



NoSQL

NoSQL: Background

- Issues with scaling up
  - Best way to provide ACID and rich query model is to have the dataset on a single machine
  - Limits to scaling up (or vertical scaling: make a "single" machine more powerful) dataset is just too big!
  - Scaling out (or horizontal scaling: adding more smaller/cheaper servers) is a better choice
  - Different approaches for horizontal scaling (multi-node database):
    - Master/Slave
    - Sharding (partitioning)

NoSQL: Background

- Scaling out RDBMS: Master/Slave
  - All writes are written to the master
  - All reads performed against the replicated slave databases
  - Critical reads may be incorrect as writes may not have been propagated down
  - Large datasets can pose problems as master needs to duplicate data to slaves

NoSQL: Background

- Scaling out RDBMS: Sharding
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
  - Can no longer have relationships/joins across partitions
  - Loss of referential integrity across shards
- Other ways to scale out RDBMS
  - Multi-Master replication
  - INSERT only, not UPDATES/DELETES
  - No JOINs, thereby reducing query time
  - This involves de-normalizing data
  - In-memory databases

NoSQL

- What is NoSQL
  - Stands for Not Only SQL
  - The term NoSQL was introduced by Carl Strozzi in 1998 to name his file-based database
  - It was again re-introduced by Eric Evans when an event was organized to discuss open source distributed databases
  - Eric states that "… but the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for. …"

NoSQL

- Avoids:
  - Overhead of ACID transactions
  - Complexity of SQL query
  - Burden of up-front schema design
  - DBA presence
  - Transactions (which should be handled at application layer)
- Provides:
  - Easy and frequent changes to DB
  - Horizontal scaling (scale out)
  - Solution to impedance mismatch
  - Fast development

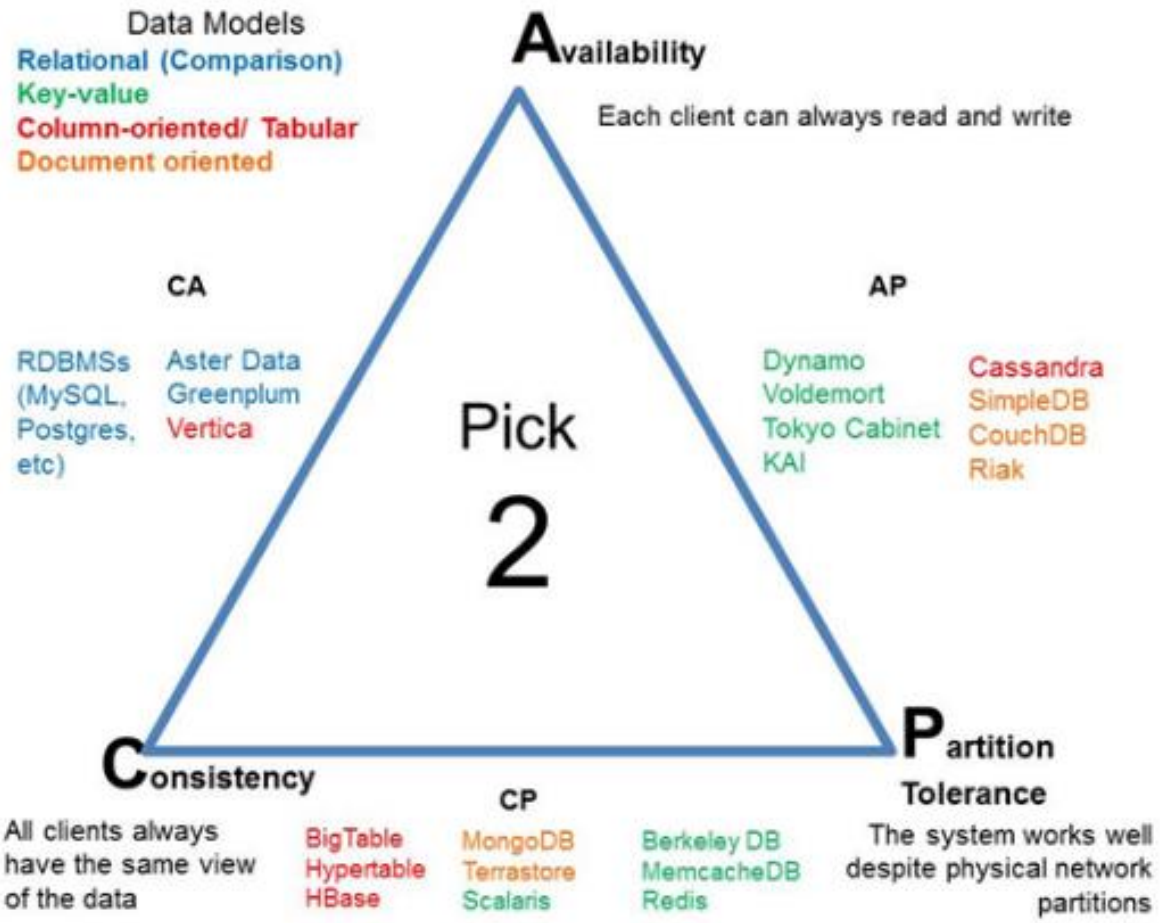
NoSQL

- Pros:
  - very fast
  - very scalable
  - simple model
  - able to distribute horizontally
- Cons:
  - many data structures (objects) can't be easily modeled as key value pairs

NoSQL

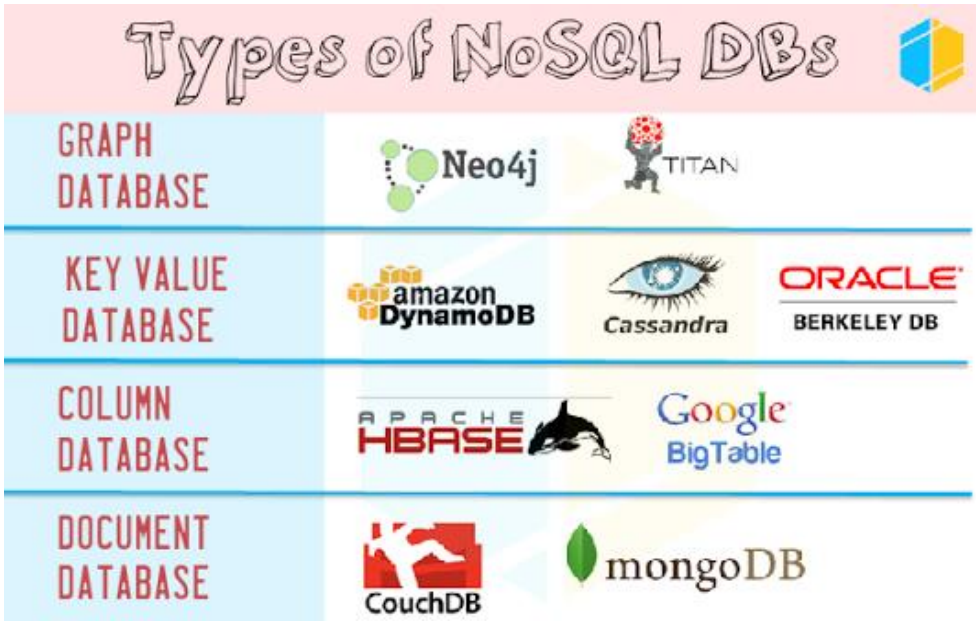
- Advantages
  - Cheap, easy to implement (open source)
  - Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - Down nodes easily replaced
  - No single point of failure
  - Easy to distribute
  - Don't require a schema
  - Can scale up and down
  - Relax the data consistency requirement (CAP)
- Giving up
  - joins
  - group by
  - order by
  - ACID transactions
  - SQL as a sometimes frustrating but still powerful query language
  - easy integration with other applications that support SQL

NoSQL


Data Models
Relational (Comparison)
Key-value
Column-oriented/ Tabular
Document oriented
Availability
Each client can always read and write
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick 2
Consistency
All clients always have the same view of the data
Partition Tolerance
The system works well despite physical network partitions
CP
BigTable
Hypertable
HBase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis

NoSQL

- Categories



NoSQL

- Which NoSQL Database to choose?

Data Base Type Based on Feature	Example of Database	Use case (When to Use)	Pros	Cons
Key/ Value	Redis, MemcacheDB	Caching, Queue-ing, Distributing information	Fast query	Unstructured, treated as strings or binaries
Column Oriented	Cassandra, HBase	Scaling, Keeping Unstructured, non-volatile	Fast query, high scalability	Relatively simple functions
Document Oriented	MongoDB, Couchbase	Nested Information, JavaScript friendly	relaxed data scheme	Slow query, lack of unified query processing language
Graph Based	OrientDB, Neo4J	Handling Complex relational information. Modeling and Handling classification.	Graph related algorithms	Complex to realize implementation

End of Chapter 2