Big-data Computing: Hadoop Distributed File System

B. RAMAMURTHY

Reference

- Apache Hadoop: http://hadoop.apache.org/
- http://wiki.apache.org/hadoop/
- Hadoop: The Definitive Guide, by Tom White, 2nd edition, Oreilly's , 2010
- Dean, J. and Ghemawat, S. 2008. **MapReduce: simplified data processing on large clusters**. *Communication of ACM* 51, 1 (Jan. 2008), 107-113.

Background

- Problem space is experiencing explosion of data
- Solution space: emergence of multi-core, virtualization, cloud computing
- Inability of traditional file system to handle data deluge
- The Big-data Computing Model
  - MapReduce Programming Model (Algorithm)
  - Google File System; Hadoop Distributed File System (Data Structure)
  - Microsoft Dryad (Large scale Data-base processing model)

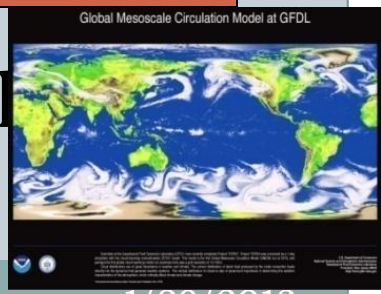
Examples

- Computational models that focus on data: large scale and/or complex data
- Example1: web log

fcrawler.looksmart.com - - [26/Apr/2000:00:00:12 -0400] "GET /contacts.html HTTP/1.0" 200 4595 "-" "FAST-WebCrawler/2.1-pre2 (ashen@looksmart.net)"
fcrawler.looksmart.com - - [26/Apr/2000:00:17:19 -0400] "GET /news/news.html HTTP/1.0" 200 16716 "-" "FAST-WebCrawler/2.1-pre2 (ashen@looksmart.net)"
ppp931.on.bellglobal.com - - [26/Apr/2000:00:16:12 -0400] "GET /download/windows/asctab31.zip HTTP/1.0" 200 1540096
"http://www.htmlgoodies.com/downloads/freeware/webdevelopment/15.html" "Mozilla/4.7 [en]C-SYMPA (Win95; U)"

123.123.123.123 - - [26/Apr/2000:00:23:48 -0400] "GET /pics/wpaper.gif HTTP/1.0" 200 6248 "http://www.jafsoft.com/asctortf/" "Mozilla/4.05 (Macintosh; I; PPC)"
123.123.123.123 - - [26/Apr/2000:00:23:47 -0400] "GET /asctortf/ HTTP/1.0" 200 8130
"http://search.netscape.com/Computers/Data_Formats/Document/Text/RTF" "Mozilla/4.05 (Macintosh; I; PPC)"
123.123.123.123 - - [26/Apr/2000:00:23:48 -0400] "GET /pics/5star2000.gif HTTP/1.0" 200 4005 "http://www.jafsoft.com/asctortf/" "Mozilla/4.05 (Macintosh; I; PPC)"
123.123.123.123 - - [26/Apr/2000:00:23:50 -0400] "GET /pics/5star.gif HTTP/1.0" 200 1031 "http://www.jafsoft.com/asctortf/" "Mozilla/4.05 (Macintosh; I; PPC)"
123.123.123.123 - - [26/Apr/2000:00:23:51 -0400] "GET /pics/a2hlogo.jpg HTTP/1.0" 200 4282 "http://www.jafsoft.com/asctortf/" "Mozilla/4.05 (Macintosh; I; PPC)"
123.123.123.123 - - [26/Apr/2000:00:23:51 -0400] "GET /cgi-bin/newcount?jafsof3&width=4&font=digital&noshow HTTP/1.0" 200 36
"http://www.jafsoft.com/asctortf/" "Mozilla/4.05 (Macintosh; I; PPC)"

- Example 2: Climate/weather data model



Traditional Storage Solutions

Off system/online storage/ secondary memory

File system abstraction/ Databases

Offline/ tertiary memory/ DFS

RAID: Redundant Array of Inexpensive Disks

NAS: Network Accessible Storage

SAN: Storage area networks

Solution Space

Google File System

- Internet introduced a new challenge in the form web logs, web crawler's data: large scale "peta scale"
- But observe that this type of data has an uniquely different characteristic than your transactional or the "customer order" data : "write once read many (WORM)" ;
  - Privacy protected healthcare and patient information;
  - Historical financial data;
  - Other historical data
- Google exploited this characteristics in its Google file system (GFS)

Data Characteristics

- Streaming data access
- Applications need streaming access to data
- Batch processing rather than interactive user access.
- Large data sets and files: gigabytes, terabytes, petabytes, exabytes size
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
- Tens of millions of files in a single instance
- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency
- WORM inspired a new programming model called the MapReduce programming model
- Multiple-readers can work on the read-only data concurrently

The Context: Big-data

- Data mining huge amounts of data collected in a wide range of domains from astronomy to healthcare has become essential for planning and performance.
- We are in a knowledge economy.
  - Data is an important asset to any organization
  - Discovery of knowledge; Enabling discovery; annotation of data
  - Complex computational models
  - No single environment is good enough: need elastic, on-demand capacities
- We are looking at newer
  - programming models, and
  - Supporting algorithms and data structures.

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

- Projects Nutch and Lucene were started with "search" as the application in mind;
- Hadoop Distributed file system and mapreduce were found to have applications beyond search.
- HDFS and MapReduce were moved out of Nutch as a sub-project of Lucene and later promoted into a apache project Hadoop
- Lets look at HDFS and MapReduce

Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware
- HDFS provides Java API for applications to use.
- It also provides a streaming API for other languages.
- (See MR in python here)
- A HTTP browser can be used to browse the files of a HDFS instance.

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.

Data Characteristics

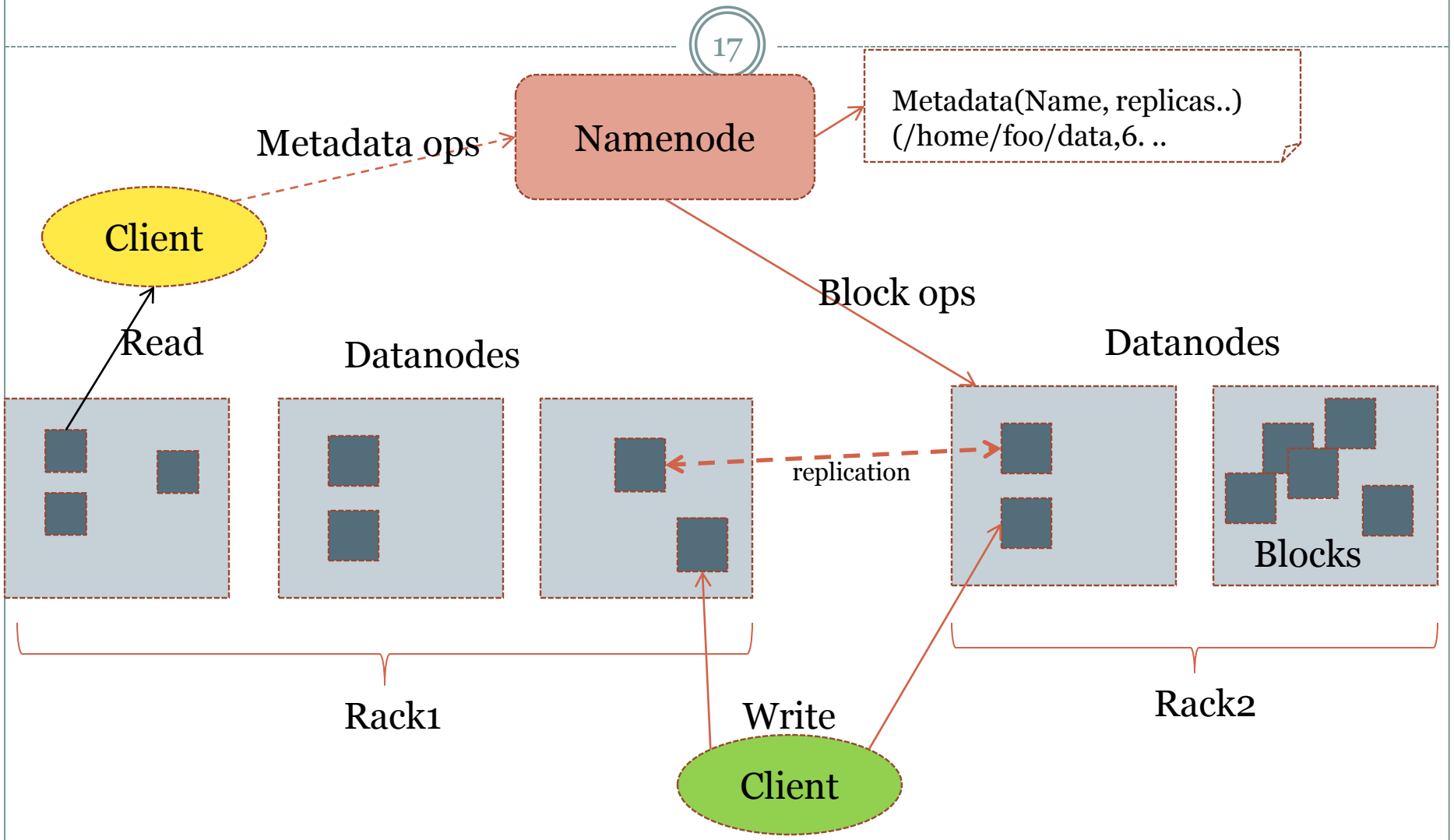
- Batch processing rather than interactive user access.
- Large data sets and files: gigabytes to terabytes size
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
- Tens of millions of files in a single instance
- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency
- A map-reduce application or web-crawler application fits perfectly with this model.

Architecture

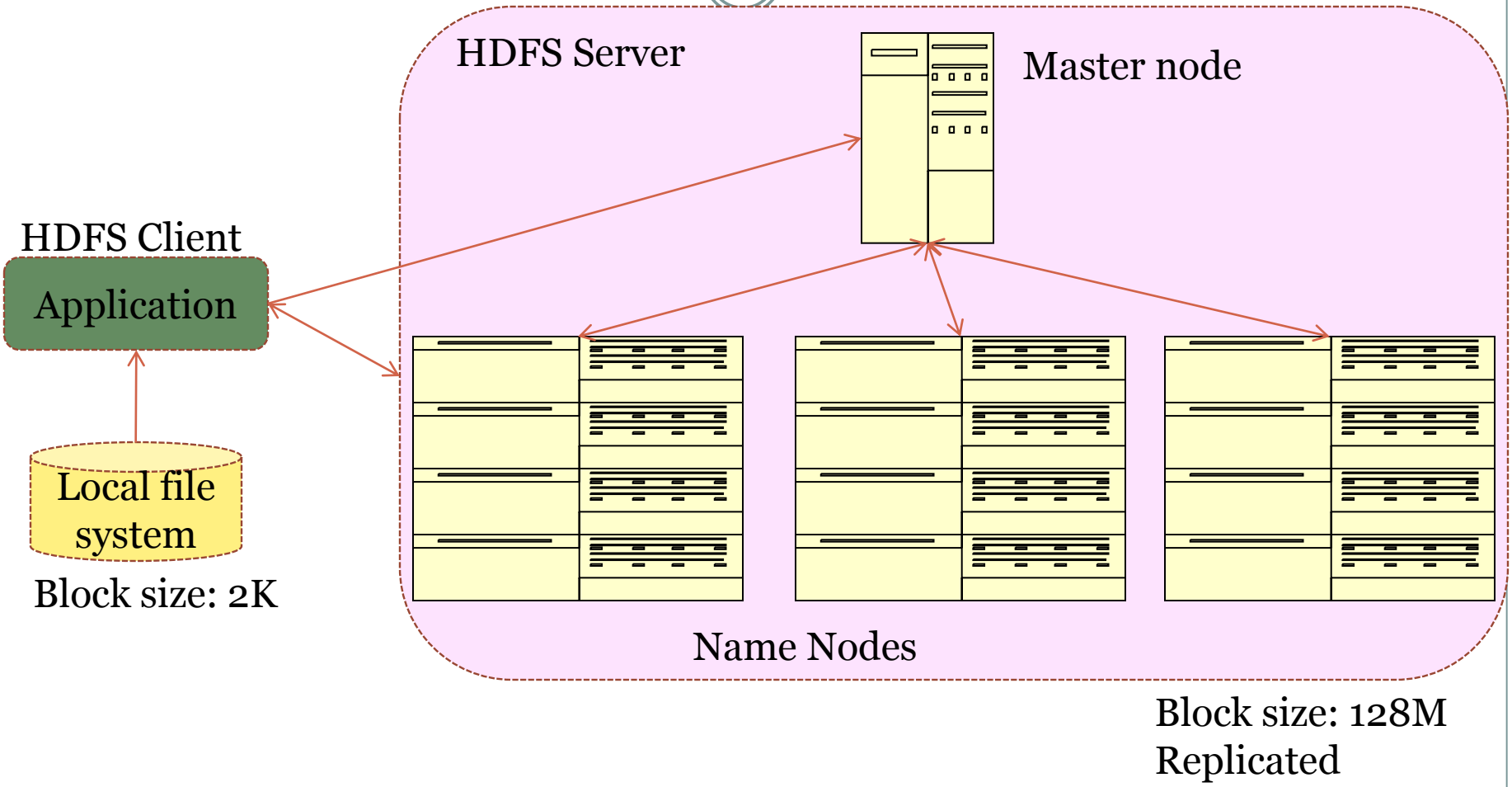
Namenode and Datanodes

- Master/slave architecture
- HDFS cluster consists of a single **Namenode**, a master server that manages the file system namespace and regulates access to files by clients.
- There are a number of **DataNodes** usually one per node in a cluster.
- The DataNodes manage storage attached to the nodes that they run on.
- HDFS exposes a file system namespace and allows user data to be stored in files.
- A file is split into one or more blocks and set of blocks are stored in DataNodes.
- DataNodes: serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode.

HDFS Architecture



Hadoop Distributed File System



File system Namespace

- Hierarchical file system with directories and files
- Create, remove, move, rename etc.
- Namenode maintains the file system
- Any meta information changes to the file system recorded by the Namenode.
- An application can specify the number of replicas of the file needed: replication factor of the file. This information is stored in the Namenode.

Data Replication

- HDFS is designed to store very large files across machines in a large cluster.
- Each file is a sequence of blocks.
- All blocks in the file except the last are of the same size.
- Blocks are replicated for fault tolerance.
- Block size and replicas are configurable per file.
- The Namenode receives a Heartbeat and a BlockReport from each DataNode in the cluster.
- BlockReport contains all the blocks on a Datanode.

Replica Placement

- The placement of the replicas is critical to HDFS reliability and performance.
- Optimizing replica placement distinguishes HDFS from other distributed file systems.
- Rack-aware replica placement:
  - Goal: improve reliability, availability and network bandwidth utilization
- Many racks, communication between racks are through switches.
- Network bandwidth between machines on the same rack is greater than those in different racks.
- Namenode determines the rack id for each DataNode.
- Replicas are typically placed on unique racks
  - Simple but non-optimal
  - Writes are expensive
  - Replication factor is 3
- Replicas are placed: one on a node in a local rack, one on a different node in the local rack and one on a node in a different rack.
- 1/3 of the replica on a node, 2/3 on a rack and 1/3 distributed evenly across remaining racks.

Replica Selection

- Replica selection for READ operation: HDFS tries to minimize the bandwidth consumption and latency.
- If there is a replica on the Reader node then that is preferred.
- HDFS cluster may span multiple data centers: replica in the local data center is preferred over the remote one.

Safemode Startup

- On startup Namenode enters Safemode.
- Replication of data blocks do not occur in Safemode.
- Each DataNode checks in with Heartbeat and BlockReport.
- Namenode verifies that each block has acceptable number of replicas
- After a configurable percentage of safely replicated blocks check in with the Namenode, Namenode exits Safemode.
- It then makes the list of blocks that need to be replicated.
- Namenode then proceeds to replicate these blocks to other Datanodes.

Filesystem Metadata

- The HDFS namespace is stored by Namenode.
- Namenode uses a transaction log called the EditLog to record every change that occurs to the filesystem meta data.
  - For example, creating a new file.
  - Change replication factor of a file
  - EditLog is stored in the Namenode's local filesystem
- Entire filesystem namespace including mapping of blocks to files and file system properties is stored in a file FsImage. Stored in Namenode's local filesystem.

Namenode

- Keeps image of entire file system namespace and file Blockmap in memory.
- 4GB of local RAM is sufficient to support the above data structures that represent the huge number of files and directories.
- When the Namenode starts up it gets the FsImage and Editlog from its local file system, update FsImage with EditLog information and then stores a copy of the FsImage on the filesytstem as a checkpoint.
- Periodic checkpointing is done. So that the system can recover back to the last checkpointed state in case of a crash.

Datanode

- A Datanode stores data in files in its local file system.
- Datanode has no knowledge about HDFS filesystem
- It stores each block of HDFS data in a separate file.
- Datanode does not create all files in the same directory.
- It uses heuristics to determine optimal number of files per directory and creates directories appropriately:
- When the filesystem starts up it generates a list of all HDFS blocks and send this report to Namenode: Blockreport.

Protocol

The Communication Protocol

- All HDFS communication protocols are layered on top of the TCP/IP protocol
- A client establishes a connection to a configurable TCP port on the Namenode machine. It talks ClientProtocol with the Namenode.
- The Datanodes talk to the Namenode using Datanode protocol.
- RPC abstraction wraps both ClientProtocol and Datanode protocol.
- Namenode is simply a server and never initiates a request; it only responds to RPC requests issued by DataNodes or clients.

Robustness

Possible Failures

- Primary objective of HDFS is to store data reliably in the presence of failures.
- Three common failures are: Namenode failure, Datanode failure and network partition.

DataNode failure and heartbeat

- A network partition can cause a subset of Datanodes to lose connectivity with the Namenode.
- Namenode detects this condition by the absence of a Heartbeat message.
- Namenode marks Datanodes without Hearbeat and does not send any IO requests to them.
- Any data registered to the failed Datanode is not available to the HDFS.
- Also the death of a Datanode may cause replication factor of some of the blocks to fall below their specified value.

Re-replication

- The necessity for re-replication may arise due to:
  - A Datanode may become unavailable,
  - A replica may become corrupted,
  - A hard disk on a Datanode may fail, or
  - The replication factor on the block may be increased.

Cluster Rebalancing

- HDFS architecture is compatible with data rebalancing schemes.
- A scheme might move data from one Datanode to another if the free space on a Datanode falls below a certain threshold.
- In the event of a sudden high demand for a particular file, a scheme might dynamically create additional replicas and rebalance other data in the cluster.
- These types of data rebalancing are not yet implemented: research issue.

Data Integrity

- Consider a situation: a block of data fetched from Datanode arrives corrupted.
- This corruption may occur because of faults in a storage device, network faults, or buggy software.
- A HDFS client creates the checksum of every block of its file and stores it in hidden files in the HDFS namespace.
- When a clients retrieves the contents of file, it verifies that the corresponding checksums match.
- If does not match, the client can retrieve the block from a replica.

Metadata Disk Failure

- FsImage and EditLog are central data structures of HDFS.
- A corruption of these files can cause a HDFS instance to be non-functional.
- For this reason, a Namenode can be configured to maintain multiple copies of the FsImage and EditLog.
- Multiple copies of the FsImage and EditLog files are updated synchronously.
- Meta-data is not data-intensive.
- The Namenode could be single point failure: automatic failover has been recently added with a backup namenode.

Data Organization

Data Blocks

- HDFS support write-once-read-many with reads at streaming speeds.
- A typical block size is 64MB (or even 128 MB).
- A file is chopped into 64MB chunks and stored.

Staging

- A client request to create a file does not reach Namenode immediately.
- HDFS client caches the data into a temporary file. When the data reached a HDFS block size the client contacts the Namenode.
- Namenode inserts the filename into its hierarchy and allocates a data block for it.
- The Namenode responds to the client with the identity of the Datanode and the destination of the replicas (Datanodes) for the block.
- Then the client flushes it from its local memory.

Staging (contd.)

- The client sends a message that the file is closed.
- Namenode proceeds to commit the file for creation operation into the persistent store.
- If the Namenode dies before file is closed, the file is lost.
- This client side caching is required to avoid network congestion; also it has precedence is AFS (Andrew file system).

Replication Pipelining

- When the client receives response from Namenode, it flushes its block in small pieces (4K) to the first replica, that in turn copies it to the next replica and so on.
- Thus data is pipelined from Datanode to the next.

API (Accessibility)

FS Shell, Admin and Browser Interface

- HDFS organizes its data in files and directories.
- It provides a command line interface called the FS shell that lets the user interact with data in the HDFS.
- The syntax of the commands is similar to bash and csh.
- Example: to create a directory /foodir

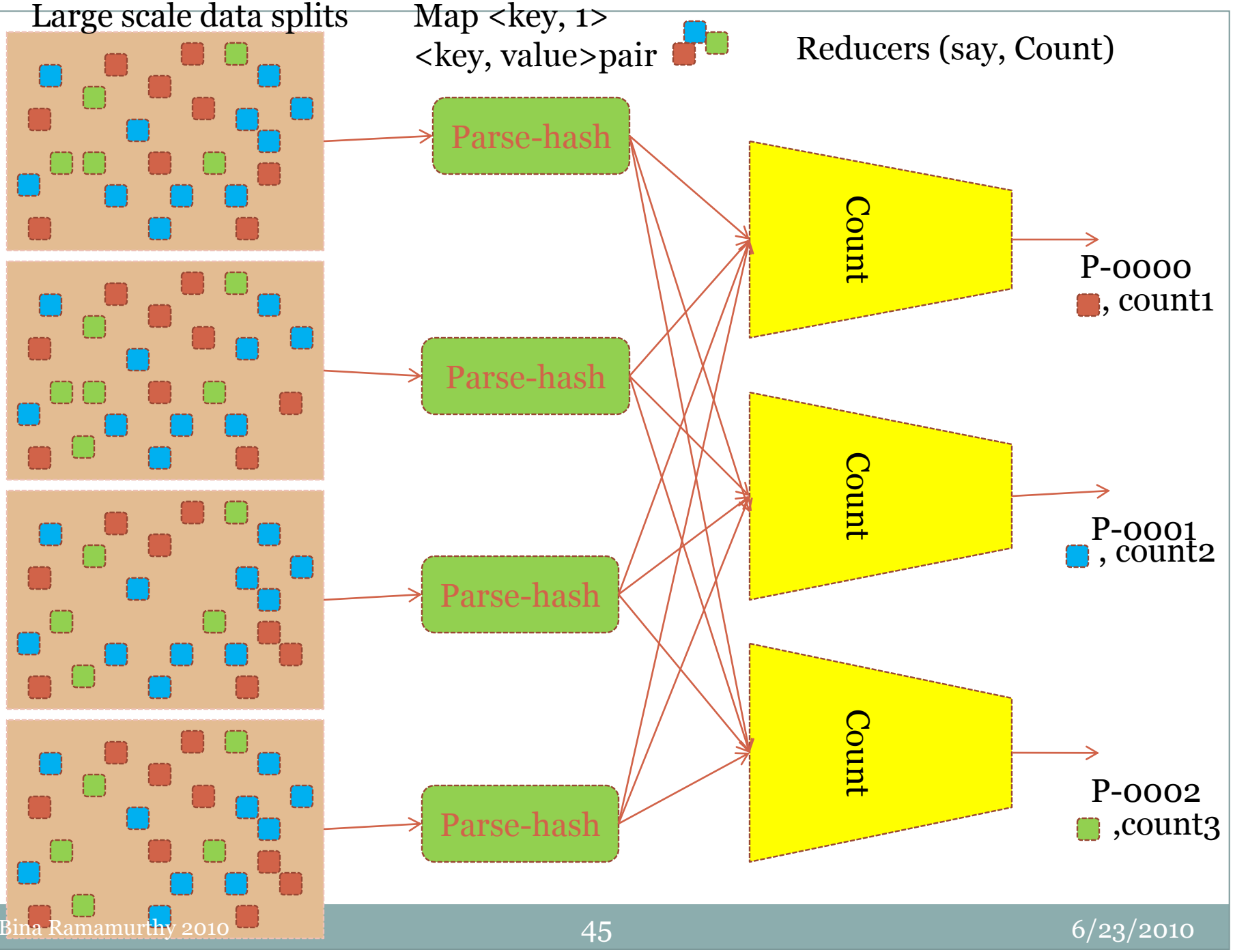
/bin/hadoop dfs –mkdir /foodir

- There is also DFSAdmin interface available
- Browser interface is also available to view the namespace.

Space Reclamation

- When a file is deleted by a client, HDFS renames file to a file in be the /trash directory for a configurable amount of time.
- A client can request for an undelete in this allowed time.
- After the specified time the file is deleted and the space is reclaimed.
- When the replication factor is reduced, the Namenode selects excess replicas that can be deleted.
- Next heartbeat transfers this information to the Datanode that clears the blocks for use.

MapReduce Engine


Large scale data splits
Map <key, 1>
<key, value>pair
Reducers (say, Count)
Parse-hash
Parse-hash
Parse-hash
Parse-hash
Count
Count
Count
P-0000
, count1
P-0001
, count2
P-0002
,count3

MapReduce Engine

- MapReduce requires a distributed file system and an engine that can distribute, coordinate, monitor and gather the results.
- Hadoop provides that engine through (the file system we discussed earlier) and the JobTracker + TaskTracker system.
- JobTracker is simply a scheduler.
- TaskTracker is assigned a Map or Reduce (or other operations); Map or Reduce run on node and so is the TaskTracker; each task is run on its own JVM on a node.

Job Tracker

- Is a service with Hadoop system
- It is like a scheduler
- Client application is sent to the JobTracker
- It talks to the Namenode, locates the TaskTracker near the data (remember the data has been populated already).
- JobTracker moves the work to the chosen TaskTracker node.
- TaskTracker monitors the execution of the task and updates the JobTracker through heartbeat. Any failure of a task is detected through missing heartbeat.
- Intermediate merging on the nodes are also taken care of by the JobTracker

TaskTracker

- It accepts tasks (Map, Reduce, Shuffle, etc.) from JobTracker
- Each TaskTracker has a number of slots for the tasks; these are execution slots available on the machine or machines on the same rack;
- It spawns a sepearte JVM for execution of the tasks;
- It indicates the number of available slots through the hearbeat message to the JobTracker

MapReduce Example: Mapper

This is a cat

Cat sits on a roof

<this 1> <is 1> <a <1,1,>> <cat <1,1>> <sits 1> <on 1> <roof 1>

The roof is a tin roof

There is a tin can on the roof

<the <1,1>> <roof <1,1,1>> <is <1,1>> <a <1,1>> <tin <1,1>> <then 1> <can 1> <on 1>

Cat kicks the can

It rolls on the roof and falls on the next roof

<cat 1> <kicks 1> <the <1,1>> <can 1> <it 1> <roll 1> <on <1,1>> <roof <1,1>> <and 1>
<falls 1> <next 1>

The cat rolls too

It sits on the can

<the <1,1>> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <cat 1>

MapReduce Example: Combiner, Reducer

<this 1> <is 1> <a <1,1,>> <cat <1,1>> <sits 1> <on 1> <roof 1>

<the <1,1>> <roof <1,1,1>> <is <1,1>> <a <1,1>> <tin <1,1>> <then 1> <can 1> <on 1>

<cat 1> <kicks 1> <the <1,1>> <can 1> <it 1> <roll 1> <on <1,1>> <roof <1,1>> <and 1> <falls 1> <next 1>

<the <1,1>> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <cat 1>

Combine the counts of all the same words:

<cat <1,1,1,1>>

<roof <1,1,1,1,1,1>>

<can <1, 1,1>>

…

Reduce (sum in this case) the counts:

<cat 4>

<can 3>

<roof 6>

Summary

- We discussed the features of the Hadoop File System, a peta-scale file system to handle big-data sets.
- We discussed: Architecture, Protocol, API, etc.
- Also MapReduce Engine, Application Architecture
- Next task is to understand mapreduce and implement a simple mapreduce job on HDFS