Inspire…Educate…Transform.

Big Data

Session 2: HDFS

Suryaprakash Kompalli
Senior Mentor, INSOFE

Review of Last Lecture

- Different architectures
- Transition from Databases to data warehouses and data lakes
- Thinking of Large Jobs as Task Decompositions
- How BigData is changing IT and business operations

We recently learned of a team at Google that has pushed Hadoop to the limits by creating a cluster whose size is on the order of 100,000 nodes, running on the recently released Nexus One mobile phone hardware, powered by Android. By pushing computation out to these devices, the Nexus One team was able to solve the difficult rendering and scaling problems in situ.

http://blog.cloudera.com/blog/2010/04/pushing-the-limits-of-distributed-processing/

Agenda

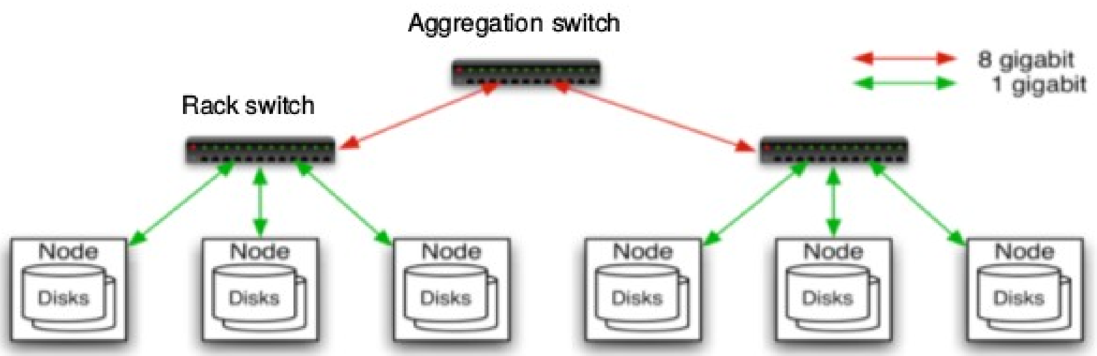
- Hadoop: HDFS

HADOOP

Hadoop: Why?

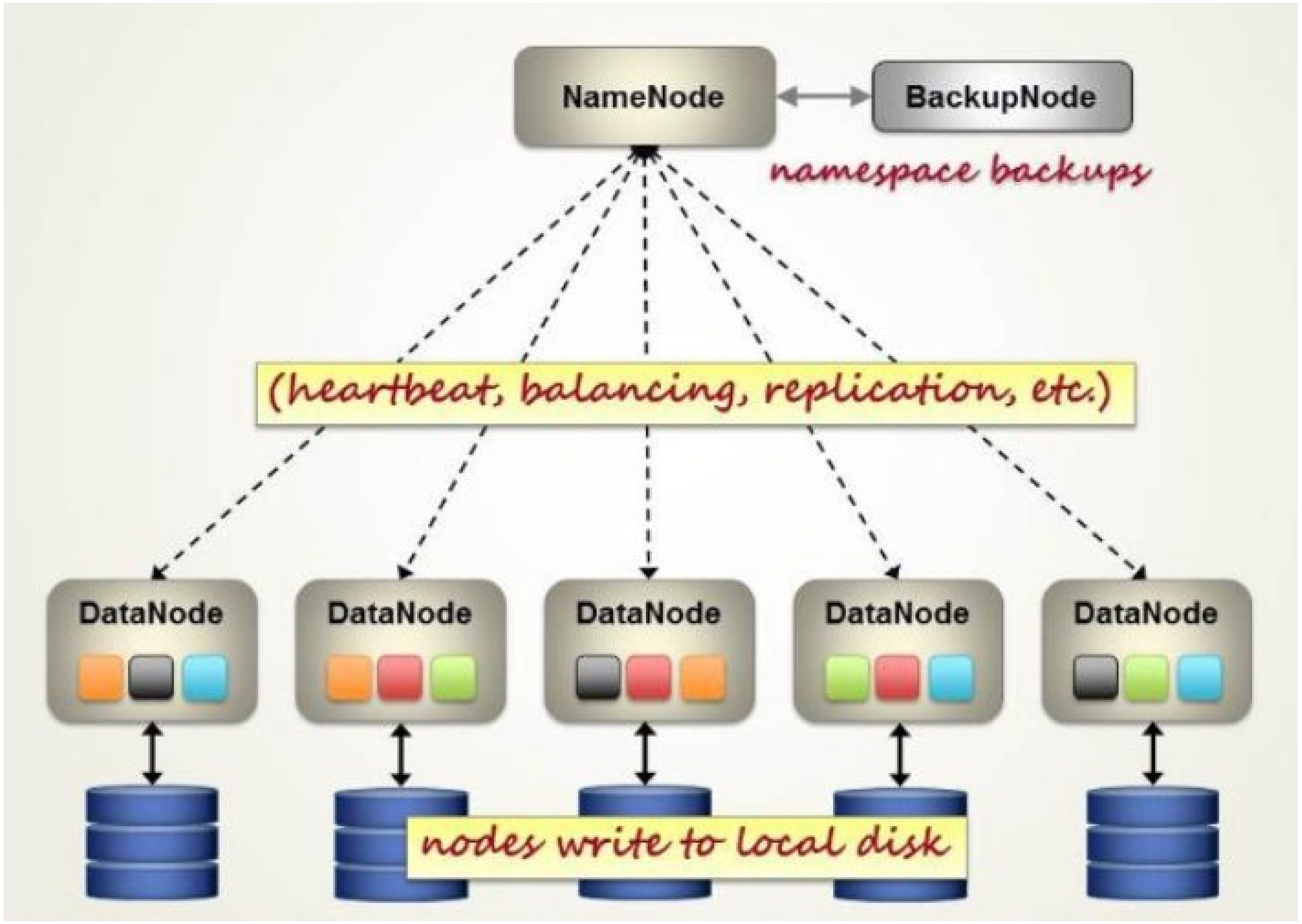
- Nodes may fail:
  - Network, disk, RAM, any other h/w or s/w
- Compute and storage need to scale on demand
- Need common storage infra for variety of applications
  - Storage should be available, failsafe, scalable

Hadoop: What?

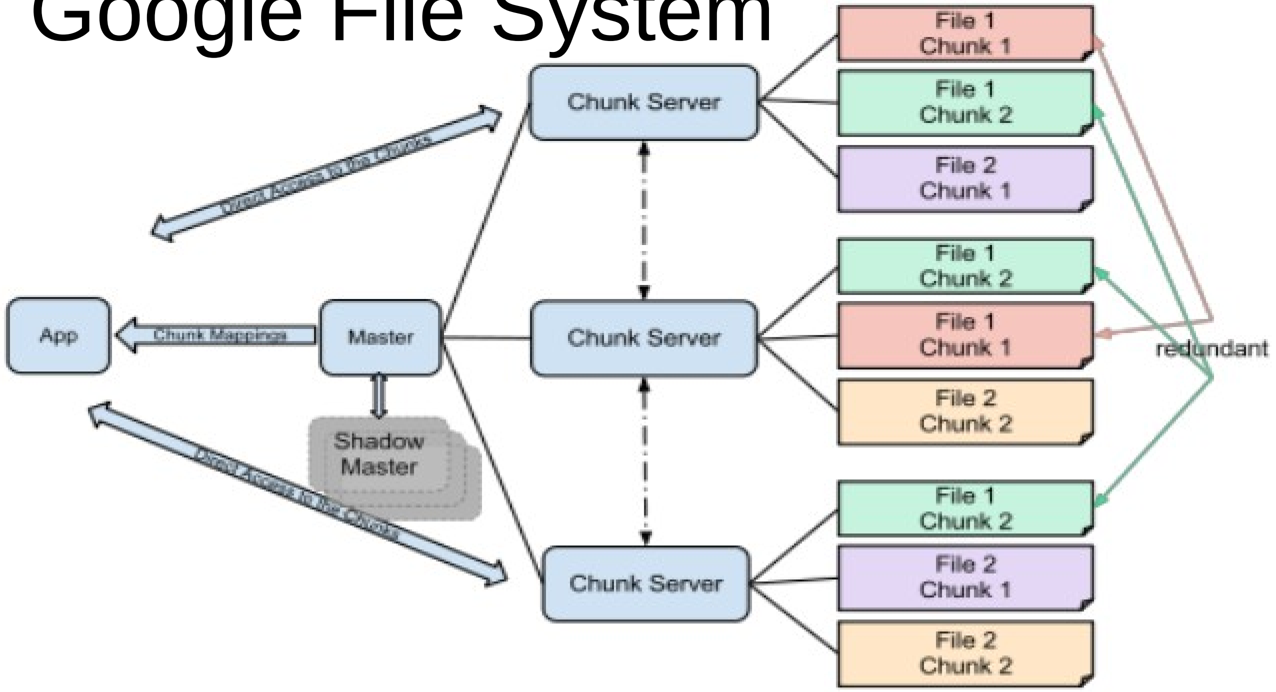


- Commodity hardware with multi-level architecture
  - Tolerate higher failure rates than custom designed systems
  - 3-4 Gigabit connections
  - Replicate data multiple times for increased availability
- Distribute data initially
  - Let processors / nodes work on local data
- Minimize data transfer over network
- Write applications at a high level
  - Just like Hardware Abstraction Layer in OS, programmers should offload to hadoop network programming, temporal dependencies, low level infrastructure, etc

HDFS


NameNode
BackupNode
namespace backups
(heartbeat, balancing, replication, etc.)
DataNode
DataNode
DataNode
DataNode
DataNode
nodes write to local disk

Hadoop: Google File System



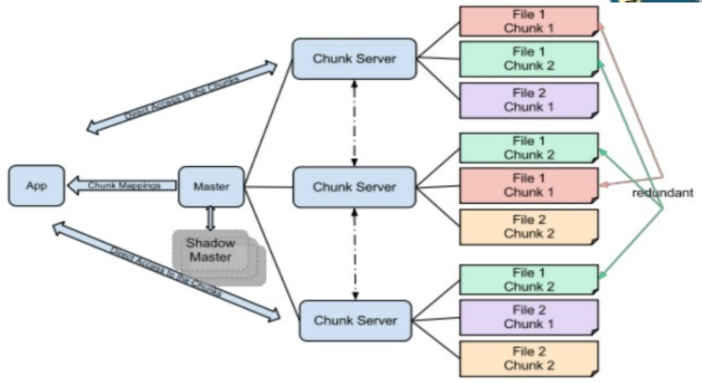
- Masters manage metadata (naming, chunk location, etc.)
- Data transfers happen directly between application/chunkservers
  - Files are broken into chunks (typically 64 MB)
- Each chunk replicated on 3 chunkservers
- No data caching on applications
- Written in C++

When a user says I want to write 2GB:

- Master decides to divide data into: 2*1024 / 64 MB = 32 chunks
- Master finds out which chunk servers have space (No of chunk servers = 3 * 32, since each chunk is replicated 3 times)
- Asks the user to transfer file directly to chunk servers, updates Master entry for the file

Reverse process happens while reading

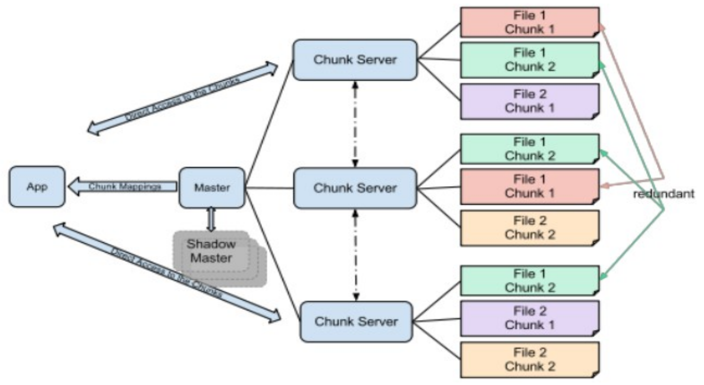
GFS: Master responsibility



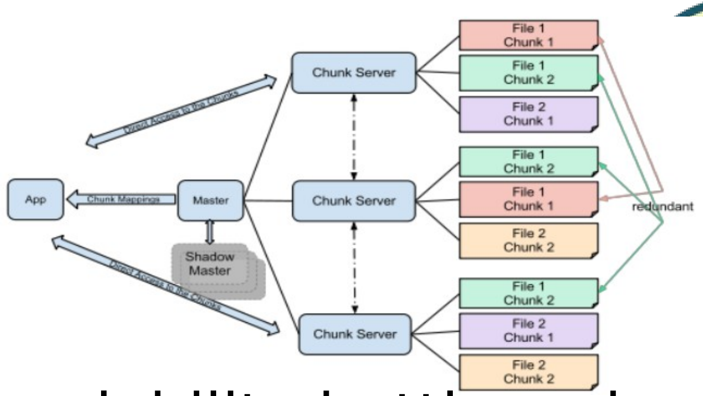
- Meta data storage
- Namespace management
- Periodic monitoring of chunk server
  - Give instruction, handle failure
- Chunk creation, re-replication, re-balancing
  - Optimize space utilization
  - Re-replicate data if chunk server fails (no health signal)
  - Rebalance data to smooth out storage and request load
  - Garbage collection
  - Stale replica deletion

GFS: What is Metadata?

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- Types of 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 Log
  - Records file creations, file deletions etc



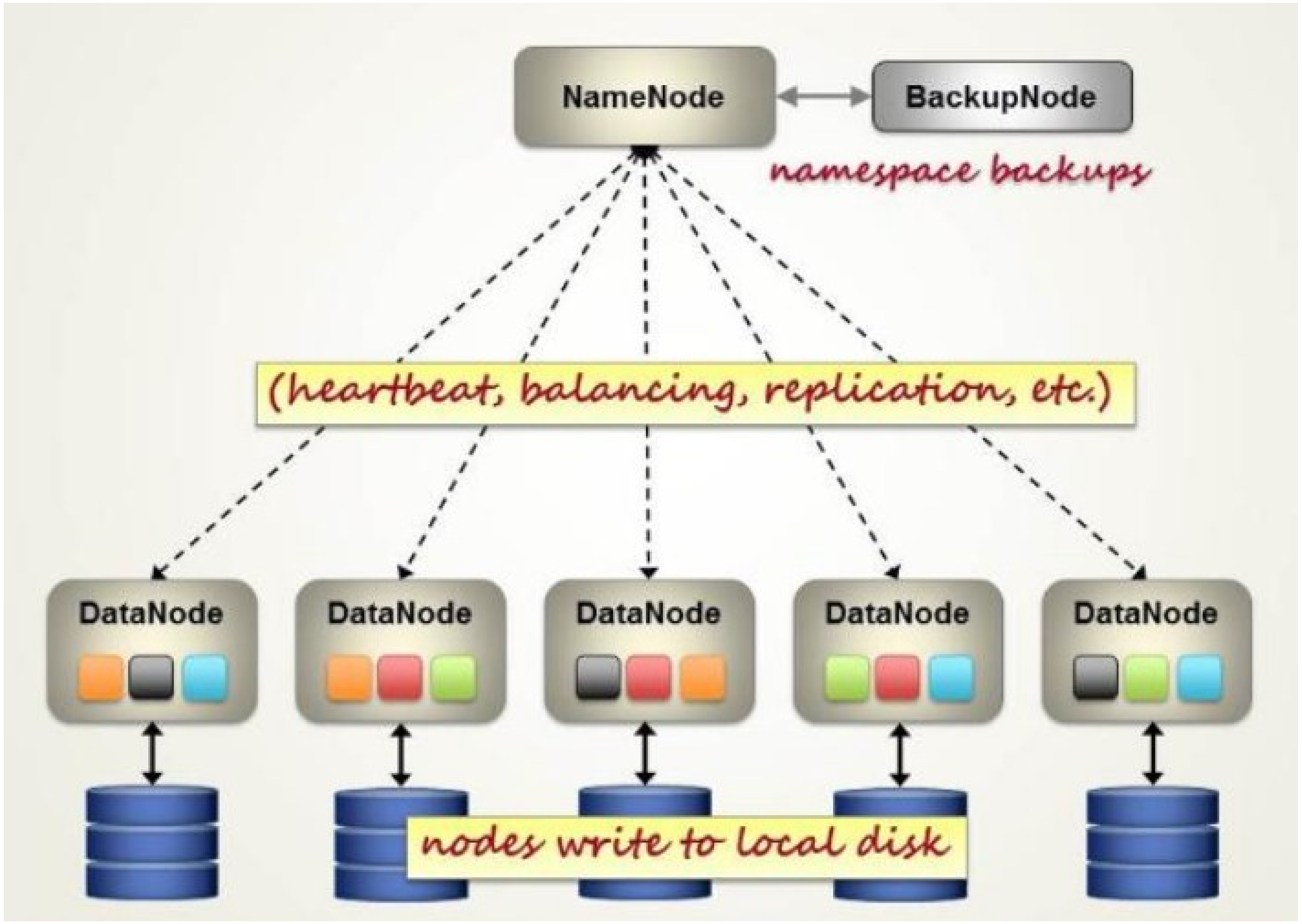
GFS: Single Master



- Master is single point of failure and scalability bottleneck
- GFS handles these by:
  - Shadow master
    - Write all master logs to disk, create checkpoints
    - If master reboots or fails, shadow master is used to recreate master
      - System operator has to manually determine a new physical machine that gets selected as master and recreates the master from the shadow master
  - Minimize master role:
    - Never move data through master
    - Large chunk size
    - Master delegates to primary replicas for mutations
      - In case of data append, one replica is given authority to: make the append on its own chunk, and to perform append everywhere else

net.pku.edu.cn/~course/cs402/2008/slide/DistributedFilesystems.ppt

HDFS


NameNode
BackupNode
namespace backups
(heartbeat, balancing, replication, etc.)
DataNode
DataNode
DataNode
DataNode
DataNode
nodes write to local disk

GFS vs HDFS

GFS	HDFS
Master	Name Node
Chunk Server	Data Node
Shadow master	Secondary name node
Chunk	Block
Allows append	Does not allow append
C++	Java

What is the rationale behind this?
Think about the primary workload on Yahoo or Google:

- Search (Crawling/Indexing/Perform search on index)
- Video and photo uploads (Upload/Add comments/Display some videos and photos)

Most of these need:

- Write once, rarely append or modify, random reads that need to be very fast

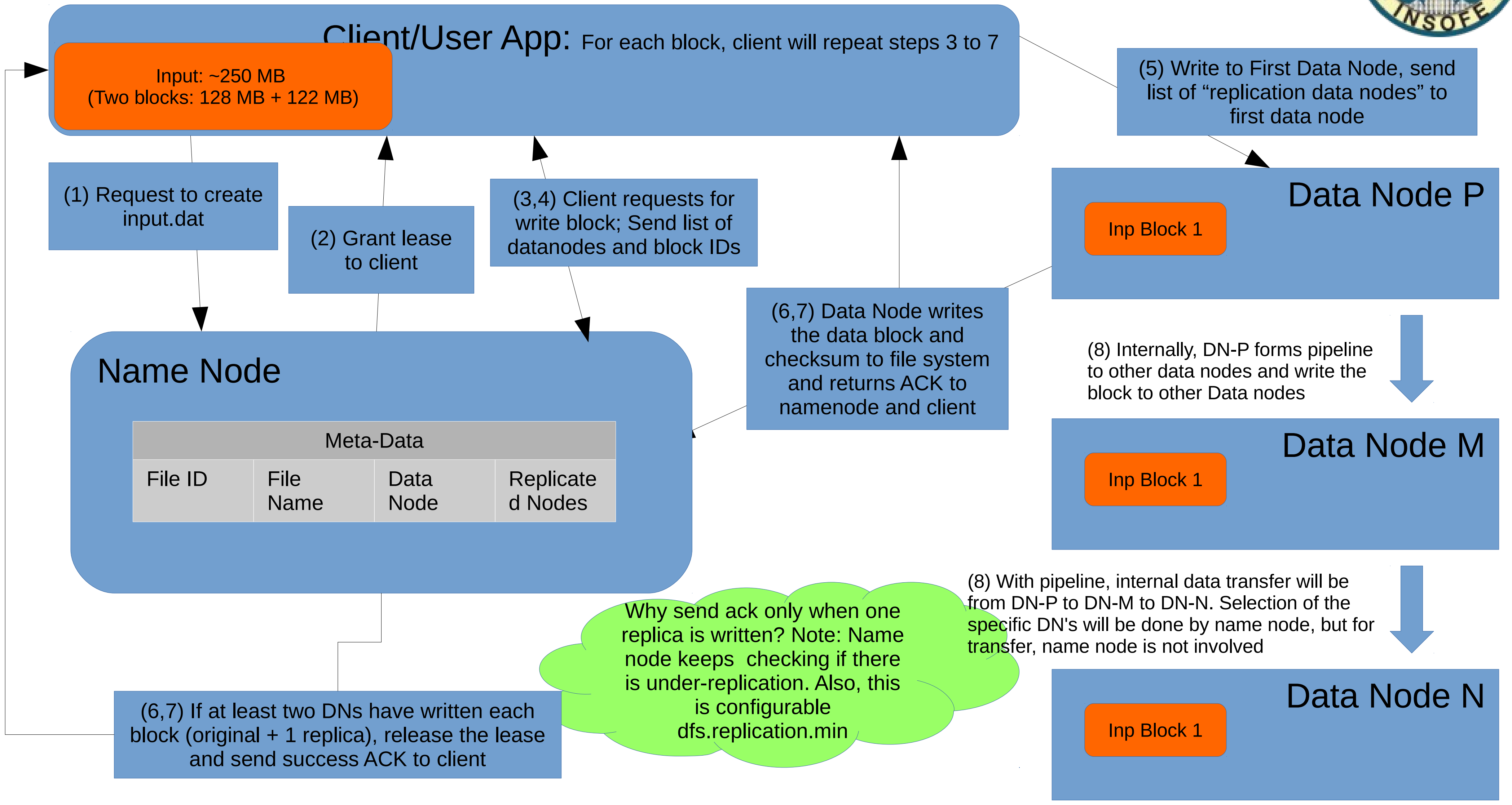
HDFS: Additional Details

- Current Strategy
  - 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 replicas
- DataNodes send hearbeat to NameNode Once every 3 seconds
- NameNode uses heartbeats to detect DataNode failure

HDFS: Additional Details

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes
- Rebalancer goal: % disk full on DataNodes should be similar
  - Usually run when new DataNodes are added
  - Cluster is online when Rebalancer is active
  - Rebalancer is throttled to avoid network congestion
  - Command line tool

Hadoop: Write Operation


Client/User App: For each block, client will repeat steps 3 to 7
Input: ~250 MB
(Two blocks: 128 MB + 122 MB)
(1) Request to create input.dat
(2) Grant lease to client
(3,4) Client requests for write block; Send list of datanodes and block IDs
(5) Write to First Data Node, send list of "replication data nodes" to first data node
(6,7) Data Node writes the data block and checksum to file system and returns ACK to namenode and client
Name Node
Meta-Data
File ID
File Name
Data Node
Replicated Nodes
Data Node P
Inp Block 1
(8) Internally, DN-P forms pipeline to other data nodes and write the block to other Data nodes
Data Node M
Inp Block 1
(8) With pipeline, internal data transfer will be from DN-P to DN-M to DN-N. Selection of the specific DN's will be done by name node, but for transfer, name node is not involved
Data Node N
Inp Block 1
Why send ack only when one replica is written? Note: Name node keeps checking if there is under-replication. Also, this is configurable dfs.replication.min
(6,7) If at least two DNs have written each block (original + 1 replica), release the lease and send success ACK to client

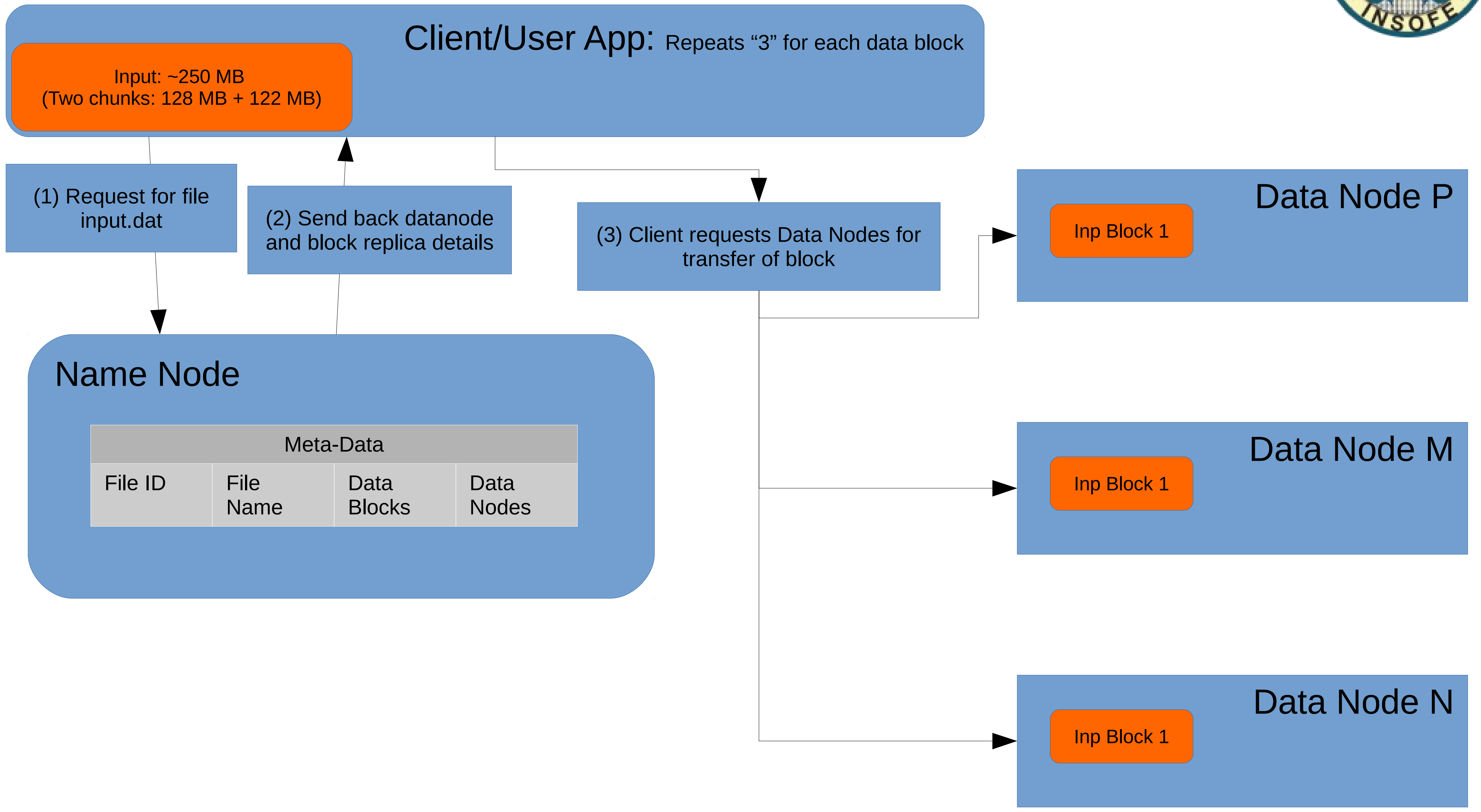
Source: http://www.bigdataplanet.info/2013/10/Hadoop-Tutorial-Part-4-Write-Operations-in-HDFS.html
http://www.devinline.com/2015/03/read-and-write-operation-in-hadoop.html

Hadoop: Write Operation

- What happens if a data node fails while writing a block?
  - Notice that a block will be made of many packets
  - Any packets that are written as part of this block are setup to be written again
  - Failed data node is removed from pipeline (subsequently deleted by the GC when the failed data node recovers)
  - Name node will eventually notice that the block is under replicated and a new replica is selected

Also look at early versions of HDFS reliability: http://blog.cloudera.com/wp-content/uploads/2010/03/HDFS_Reliability.pdf
Some background about how n/w packets and data transfer will be useful: https://www.google.co.in/search?q=packet+and+ack+basics, http://packetlife.net/blog/2010/jun/7/understanding-tcp-sequence-acknowledgment-numbers/

Hadoop: Read Operation



Client/User App: Repeats "3" for each data block
Input: ~250 MB
(Two chunks: 128 MB + 122 MB)
(1) Request for file input.dat
(2) Send back datanode and block replica details
(3) Client requests Data Nodes for transfer of block
Name Node
Meta-Data
File ID
File Name
Data Blocks
Data Nodes
Data Node P
Inp Block 1
Data Node M
Inp Block 1
Data Node N
Inp Block 1

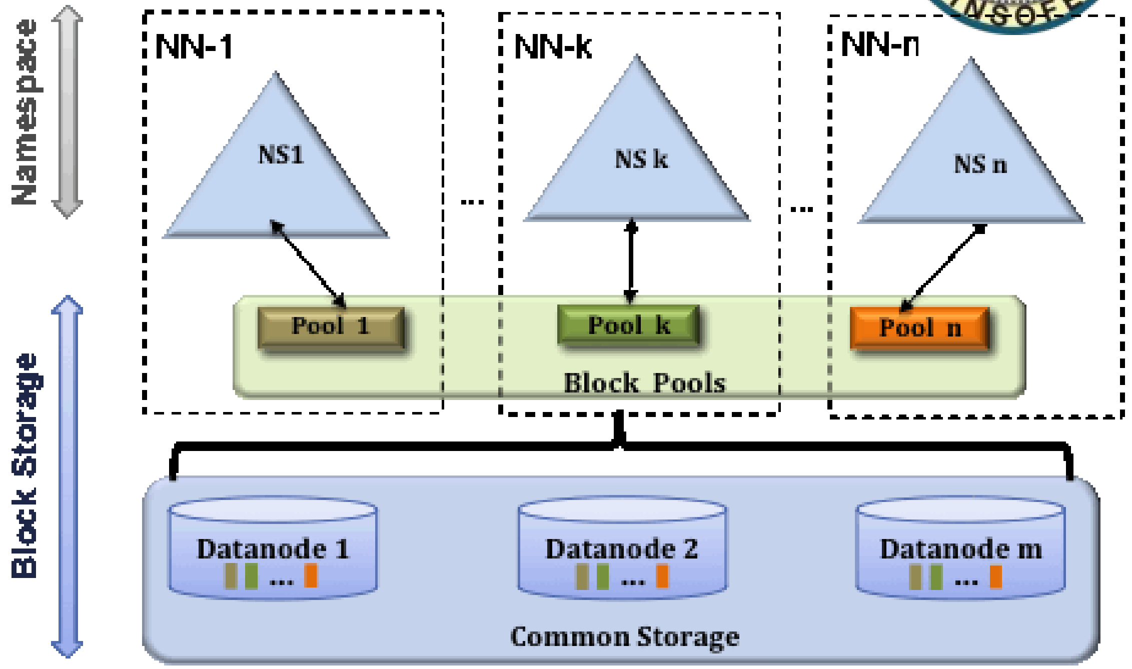
Hadoop: Typical hardware configuration

- Namenodes will have large RAM:
  - Let us say you have 32PB storage in a cluster
  - With 128 MB blocks, 32 PB, i.e. 32,000 TB storage equates 250,000,000 blocks
  - With replication factor of 3, this becomes 84 million blocks
  - To index 84 Million blocks, we will need 27 bits to address the blocks (2^^{27}= 134,217,728)
  - For index, RAM needed: ~(27 * 84) million = (27 * 84)/8 MB = 283 MB, not so much
  - But, let us assume a VERY modest 256 bytes of data for filename, app security keys etc are needed
  - With a block size of 128MB, we can have maximum of 84,000,000 files
  - 84 * 256 MBytes or ~21 GB of RAM needed for storing filename to block association
- In reality, data structures will require more RAM than above, and you need RAM for the OS, networking, and other processes
  - Not uncommon to have data nodes with ~4TB storage with 60-70% load factor (40-30% disk free)
    - Data nodes have dumb storage – no RAID or any other backup
  - In large systems, 2000-4000 data nodes may be managed by one name node

https://www.usenix.org/legacy/publications/login/2010-04/openpdfs/shvachko.pdf
https://www.safaribooksonline.com/library/view/hadoop-operations/9781449327279/ch04.html

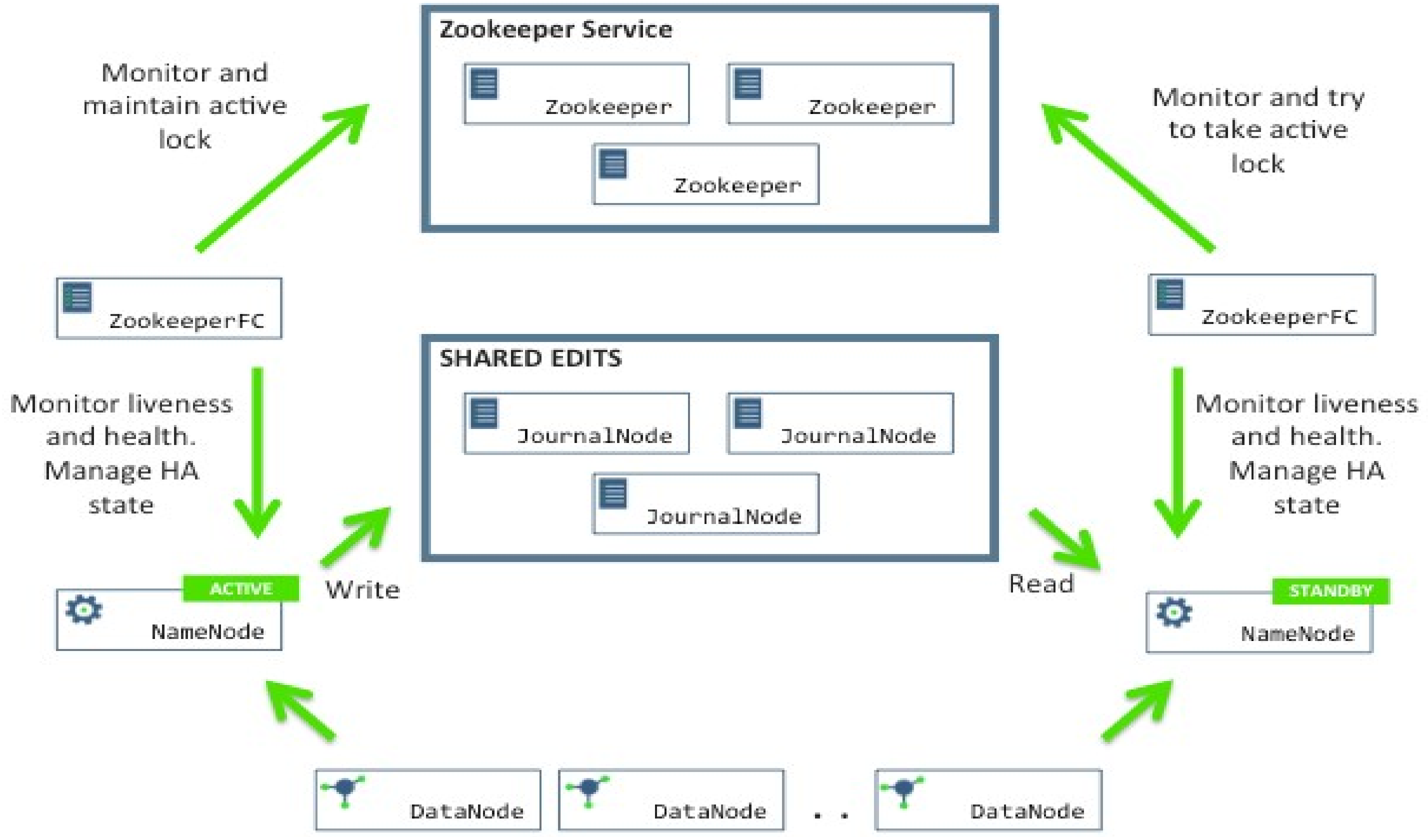
CDH4: NameNode Federation

- Each DataNodes may serve multiple name nodes
- Blocks within a data node that belong to a single namenode are called “block pools”
- Now DataNode will need to send heartbeats to multiple namenodes
  - With advent of zookeeper, datanode needs to send one heartbeat, zookeeper controls the broadcast



https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/Federation.html
http://hortonworks.com/blog/an-introduction-to-hdfs-federation/

CDH4: High-Availability Name Node



- Eliminate need for manual intervention to create name node from secondary name node
  - Both active and standby NN store everything in main memory
- You need additional book-keeping (typically done by Zookeeper):
  - Determine if Active NN is performing poorly
  - Inform all DNs and other systems as to which NN is active or standby

http://hortonworks.com/blog/namenode-high-availability-in-hdp-2-0/

Hadoop: 30,000 feet view (contd.)

- Distribute data initially
  - Let processors / nodes work on local data
  - Minimize data transfer over network
  - Replicate data multiple times for increased availability
- Write applications at a high level
  - Programmers should not have to worry about network programming, temporal dependencies, low level infrastructure, etc
- Minimize talking between nodes (share-nothing)

Inspire...Educate...Transform.

International School of Engineering

For Individuals: +91-9502334561/63 or 040-65743991

For Corporates: +91-9618483483

Web:	http://www.insofe.edu.in
Facebook:	https://www.facebook.com/insofe
Twitter:	https://twitter.com/Insofeedu
YouTube:	http://www.youtube.com/InsofeVideos
SlideShare:	http://www.slideshare.net/INSOFE
LinkedIn:	http://www.linkedin.com/company/international-school-of-engineering

Additional Material

Concurrent

ALGORITHM DESIGN

Introduction to Algorithm design and analysis

Example: sorting problem.

Input: a sequence of n number $<a_1, a_2, \ldots, a_n>$

Output: a permutation (reordering) $<a_1', a_2', \ldots, a_n'>$

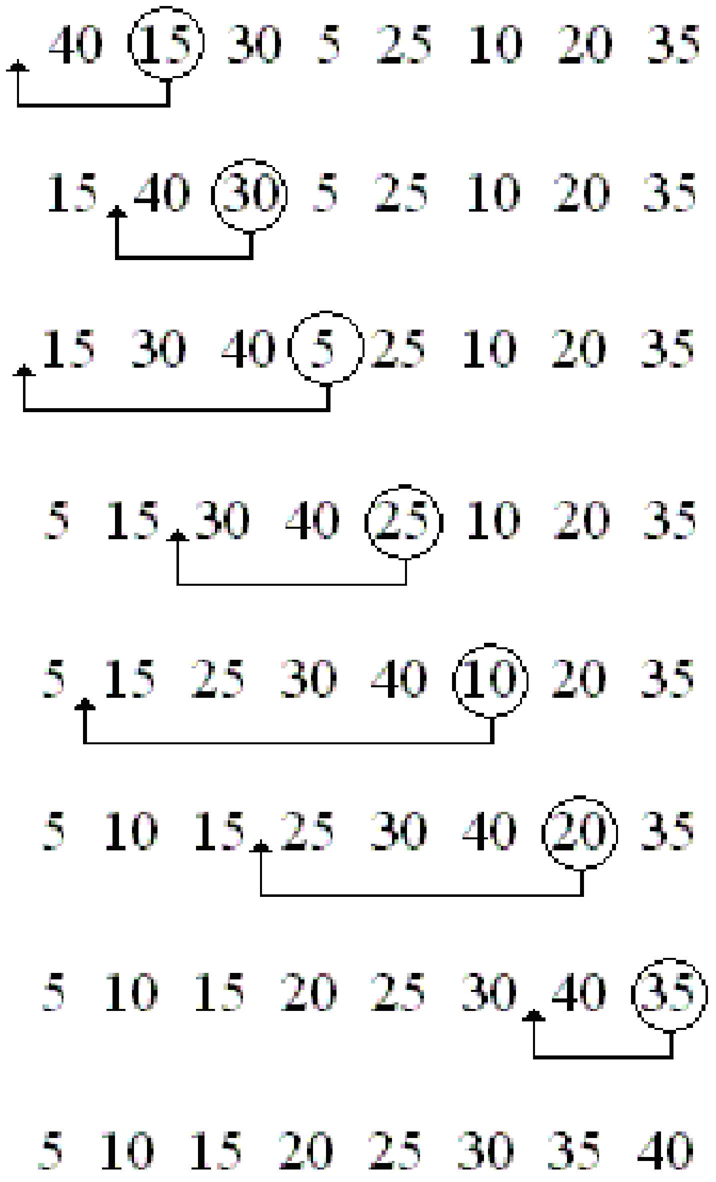
such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.

Different sorting algorithms:

Insertion sort and Mergesort.

Insertion Sort Algorithm

```
INSERTION-SORT(A)
1.  for $j$ = 2 to length[A]
2.      do $key \leftarrow$ A[$j$]
3.          //insert A[$j$] to sorted sequence A[1..$j$-1]
4.          $i \leftarrow j$-1
5.          while $i$ >0 and A[$i$]>$key$
6.              do A[$i$+1] $\leftarrow$ A[$i$]   //move A[$i$] one position right
7.                 $i \leftarrow i$-1
8.          A[$i$+1] $\leftarrow$ $key$
```



Compute worst case time for a list with “n” numbers:

Let us say numbers are indexed from 1 to n.

Item at #2: Compare **#2 with #1**, insert #2 before #1

Item at #3: Compare **#3 with #2, #1**, insert #3 before #2 or before #1

Item at #4: Compare **#4 with #3, #2, #1**, insert #4 immidiately before #3, #2, #1

In some cases, you do not have to do all the comparisons. In the given example, “30”, the number at #3 had to be compared only with 40, the number at #2.

However, in the worst case, a number at position “k” will have to be compared with k-1 elements before it.

So, number of comparisons for a list of “n” numbers will be:
#2: 1 comparison
#3: 2 comparisons
#4: 3 comparisons
...
#n: n-1 comparisons.

If you add these, number of comparisons: n(n-1)/2

In Big-O notation, you stop here and say this is O(n^2)

If you also consider the number of moves, then the moves will be 1, 2, 3....n-1 = n(n-1)/2
At each step, one additional move will be needed to move the number itself. (n)
A reason why number of moves is not considered is that you can implement this in linked list, needing only O(n) moves

https://en.wikipedia.org/wiki/Best,_worst_and_average_case

MERGE-SORT(A)

```
mergesort(A)
    If (length(A) = 2)
        return merge(A[1], A[2])
    else
        q = floor([length(A)] /2)
        Part1=mergesort(A[1, q])
        Part2=mergesort(A[q+1,length(A)])
        return merge(Part1, Part2)

merge(P, Q)
(a)i = j = 1
(b)num=length(P)+length(Q)
(c)for k = 1 to num
(d)     [Check boundary conditions:
        j < length(Q) and i < length(P)]
(e)     if P[i] < Q[j]
(f)         C[k] = P[i]
(g)         i++
        else
(h)         C[k] = Q[j]
(i)         j++
(j) return C
```

The for loop in **merge** function has the following commands: increase k (1 instruction (c)), check boundary conditions (2 instructions, (d)), check if P<Q (1, (e)), assign C, increase i or j (2). Overall, 6 instrictuons for each element that is passed to merge.

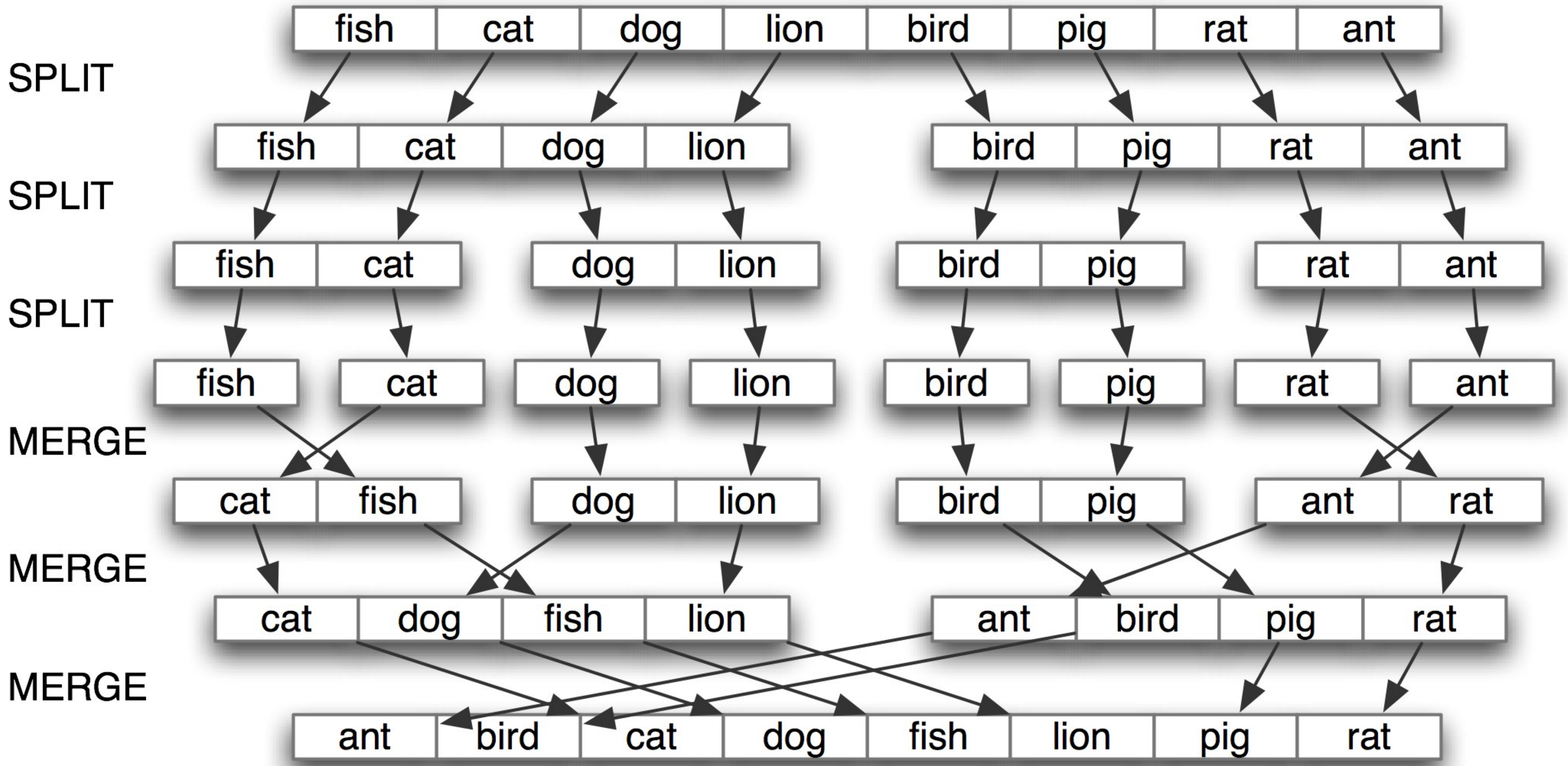
The for loop runs for M=length(P)+length(Q). So number of steps is 6xM.

mergesort breaks an array successively into two parts till it gets to a stage where there is only one element in each part. If an array has N elements, you will break it log(N) times. The following table illustrates number of calls:

$$\frac{N}{2^{(\log N)}}=1$$

Merge sort called	Original array size	Size of part1, part2	No of arrays created, no of merges to be performed	Number of commands in merge(part1, part2)	Effective steps
Iteration 1	N	N/2, N/2	2	6N	6N
Iteration 2	N/2	N/4,N/4	4, 2	6(N/2)	2*6(N/2) = 6N
Iteration 3	N/4	N/8,N/8	8, 4	6(N/4)	4*6(N/4) = 6N
log(N)	2	1,1	$2^{\log(N)}$=N, N/2	6*2	6N

Total number of commands would be:6N*log(N)
In Big-O notation, complexity: O(N.log(N))


SPLIT
SPLIT
SPLIT
MERGE
MERGE
MERGE
fish cat dog lion bird pig rat ant
fish cat dog lion | bird pig rat ant
fish cat | dog lion | bird pig | rat ant
fish | cat | dog | lion | bird | pig | rat | ant
cat fish | dog lion | bird pig | ant rat
cat dog fish lion | ant bird pig rat
ant bird cat dog fish lion pig rat

Few More Sorting Techniques

- Quicksort:
  - Same time complexity as merge sort
  - Less space complexity than merge-sort
  - Each loop needs access to the entire array or dataset.
    - This last condition makes it difficult to truly parallelize it
    - It may be used in combination with merge sort to design hybrid sorting methods
- Binary Search Tree
  - O(log n) time complexity
  - Data needs to be organized as a binary tree
    - Space complexity is O(n), constant factor is low if done intelligently
    - O(log n) time complexity to insert each element
  - Needs access to entire array or dataset
    - Hybrid BST designs are not uncommon

Time Complexity

- Is the algorithm "fast enough" for my needs?
- How much longer will the algorithm take if I increase the amount of data it must process
- Given a set of algorithms that accomplish the same thing, which is the right one to choose?

Ranking of Algorithmic Behaviors

Function	Common Name
$N!$	factorial
2^N	Exponential
N^d, $d > 3$	Polynomial
N^3	Cubic
N^2	Quadratic
$N\sqrt{N}$	
$N \log N$	
N	Linear
$\sqrt{N}$	Root - n
$\log N$	Logarithmic
1	Constant

slowest

$\downarrow$

fastest

Efficiency comparison of Insertion Sort & Merge Sort

- Suppose n=10^6 numbers:
  - Insertion sort: $c_1 n^2$
  - Merge sort: $c_2 n$ (lg n)
  - Best programmer (c_1=2), machine language, one billion/second computer A.
  - Bad programmer (c_2=50), high-language, ten million/second computer B.
  - 2 $(10^6)^2$ instructions/10^9 instructions per second = 2000 seconds.
  - 50 (10^6 lg 10^6) instructions/10^7 instructions per second ≈ 100 seconds.
  - Thus, merge sort on B is 20 times faster than insertion sort on A!
  - If sorting ten million numbers, 2.3 days VS. 20 minutes.

Running Times

- Assume N = 100,000 and processor speed is 1,000,000 operations per second

Function	Running Time
2^N	over 100 years
N^3	31.7 years
N^2	2.8 hours
$N\sqrt{N}$	31.6 seconds
N log N	1.2 seconds
N	0.1 seconds
$\sqrt{N}$	3.2×10^{-4} seconds
log N	1.2×10^{-5} seconds

Conclusions – Efficiency Comparison

- Algorithms for solving the same problem can differ dramatically in their efficiency.
- These differences are *much more* significant than the differences due to hardware and software.

This is true for concurrent programming also. We must design/choose the right algorithms.

In concurrent programming, we must also look at some more metrics.

Additional References

(Not NECESSARY. Only for optional, extra reading)

- http://en.wikipedia.org/wiki/Sorting_algorithm
- http://en.wikipedia.org/wiki/Big_O_notation
- http://www.amazon.com/Art-Computer-Programming-Sorting-Searching/dp/0201896850
- http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262033844
/
- http://blog.cloudera.com/blog/2010/04/pushing-the-limits-of-distributed-processing/