COMP9313: Big Data Management

Hadoop and HDFS

Hadoop

- Apache Hadoop is an open-source software framework that
  - Stores big data in a distributed manner
  - Processes big data parallelly
  - Builds on large clusters of commodity hardware.
- Based on Google's papers on Google File System(2003) and MapReduce(2004).
- Hadoop is
  - Scalable to Petabytes or more easily (Volume)
  - Offering parallel data processing (Velocity)
  - Storing all kinds of data (Variety)

Hadoop offers

- Redundant, Fault-tolerant data storage (HDFS)
- Parallel computation framework (MapReduce)
- Job coordination/scheduling (YARN)

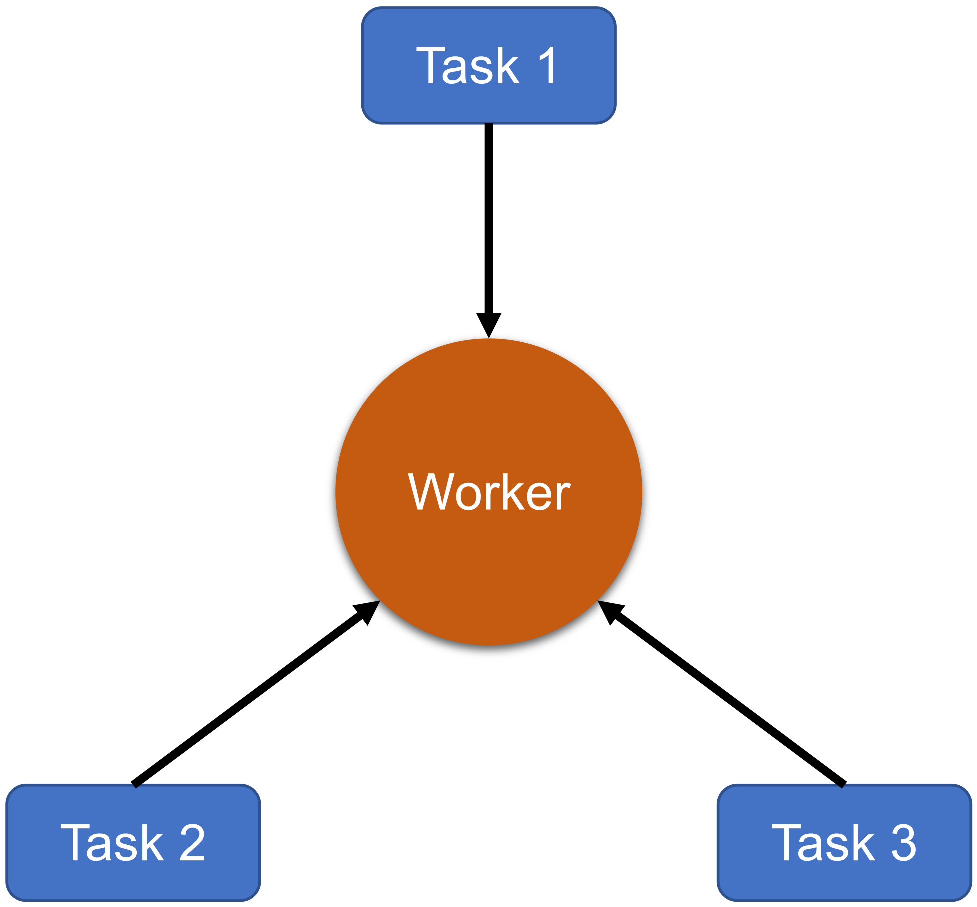
- Programmers no longer need to worry about
  - Where file is located?
  - How to handle failures & data lost?
  - How to divide computation?
  - How to program for scaling?

Hadoop Ecosystem

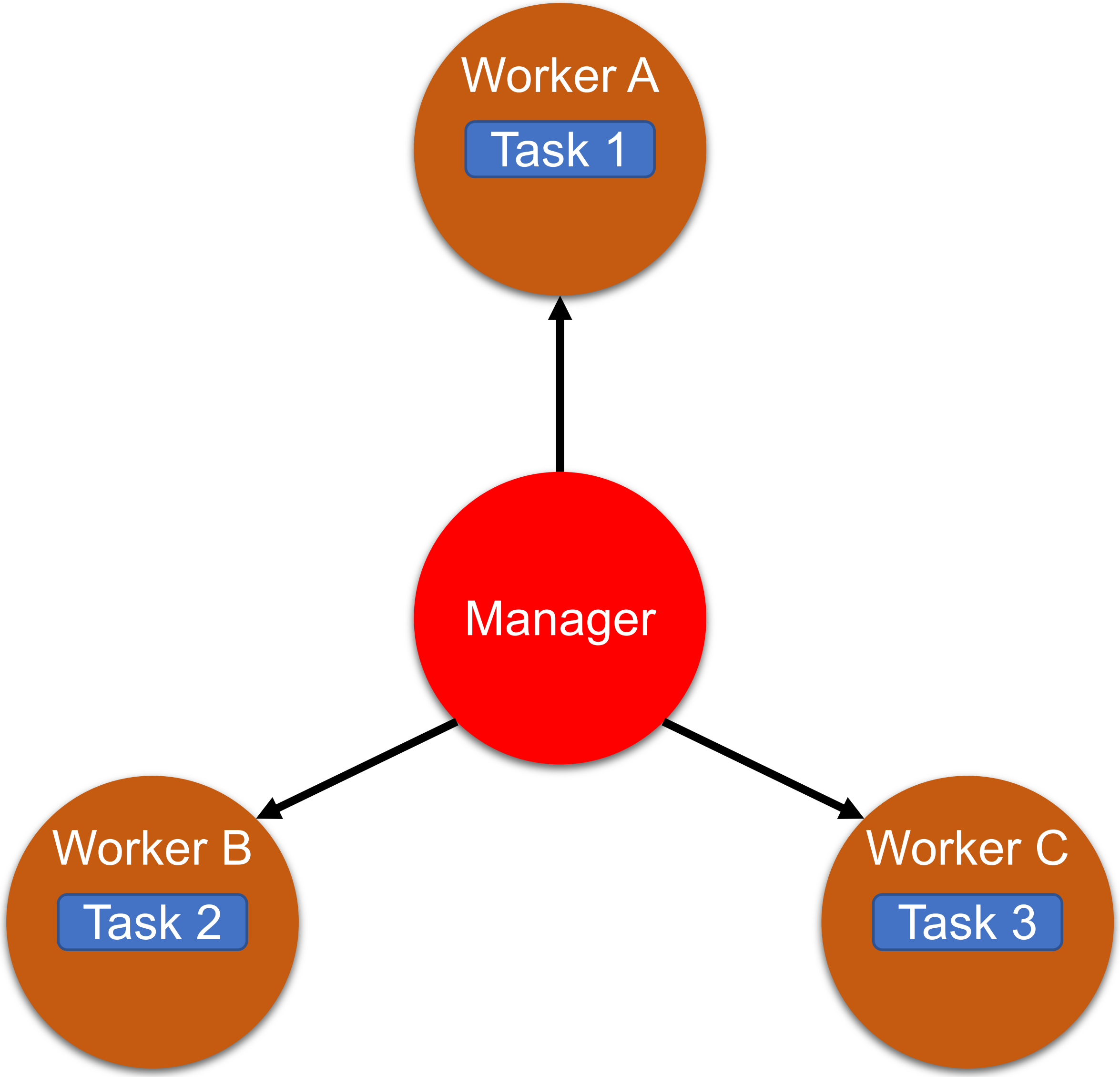
- Core of Hadoop
  - Hadoop distributed file system (HDFS)
  - MapReduce
  - YARN (Yet Another Resource Negotiator) (from Hadoop v2.0)

- Additional software packages
  - Pig
  - Hive
  - Spark
  - HBase
  - …

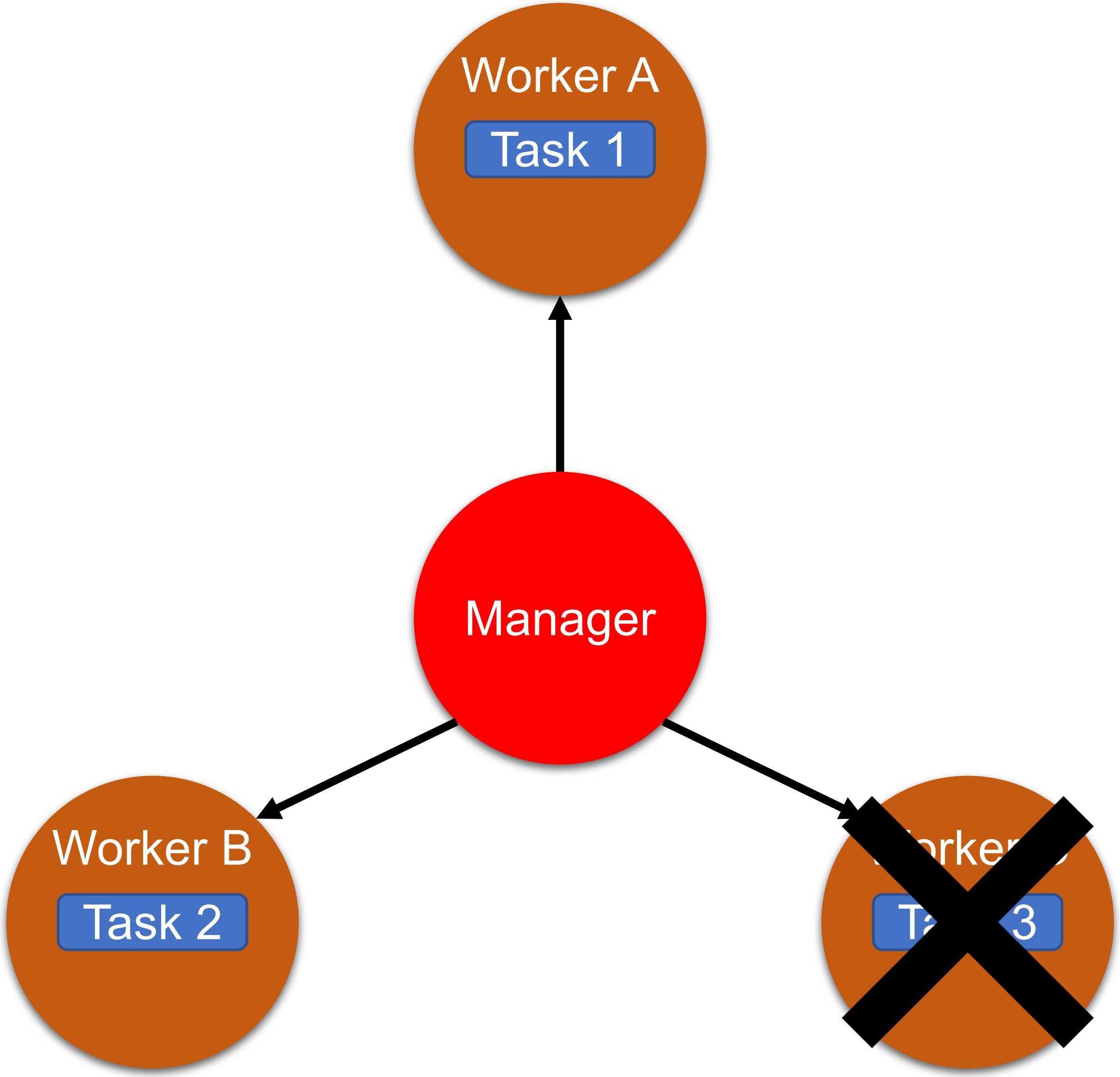
The Master-Slave Architecture of Hadoop



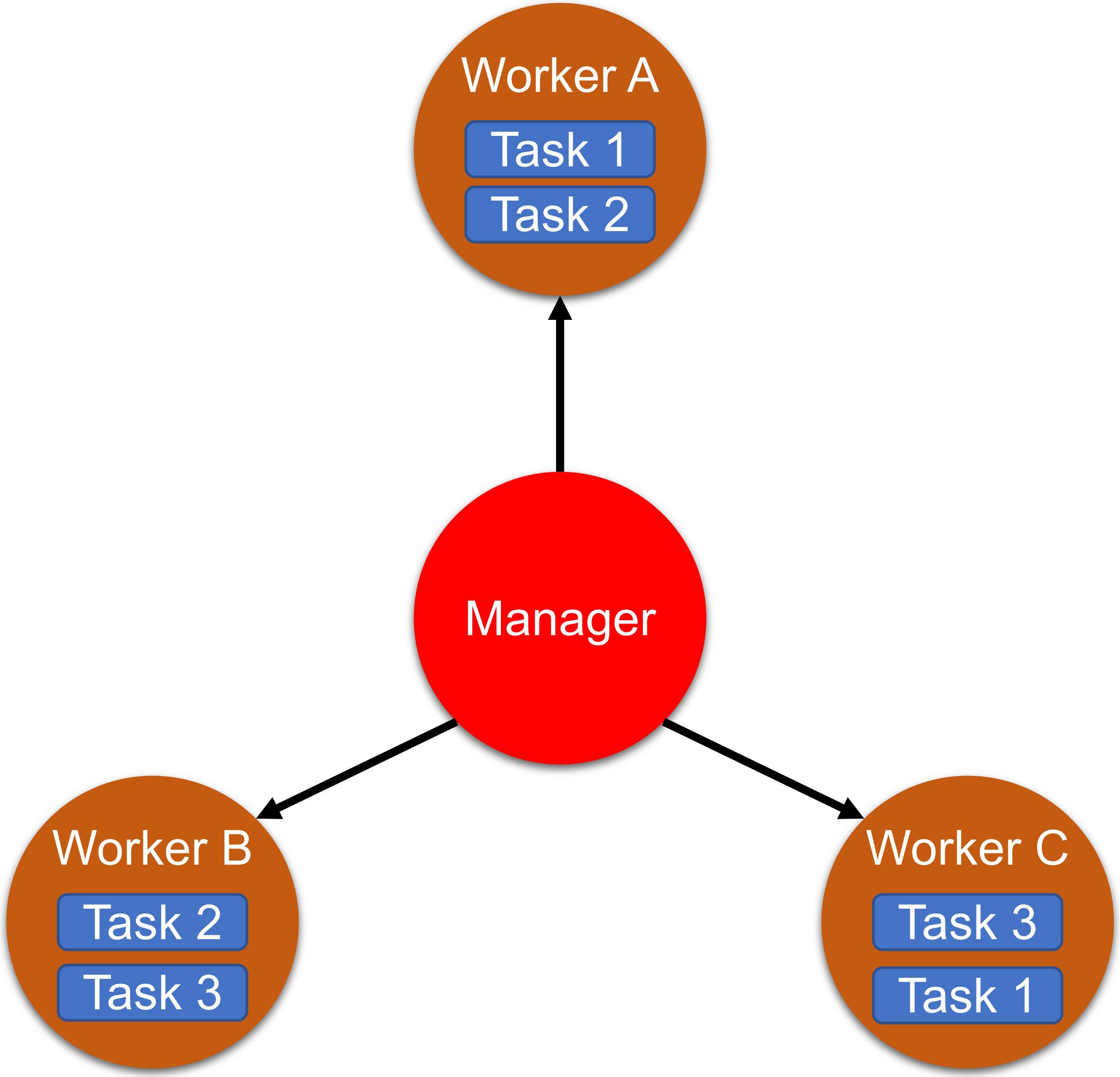
The Master-Slave Architecture of Hadoop



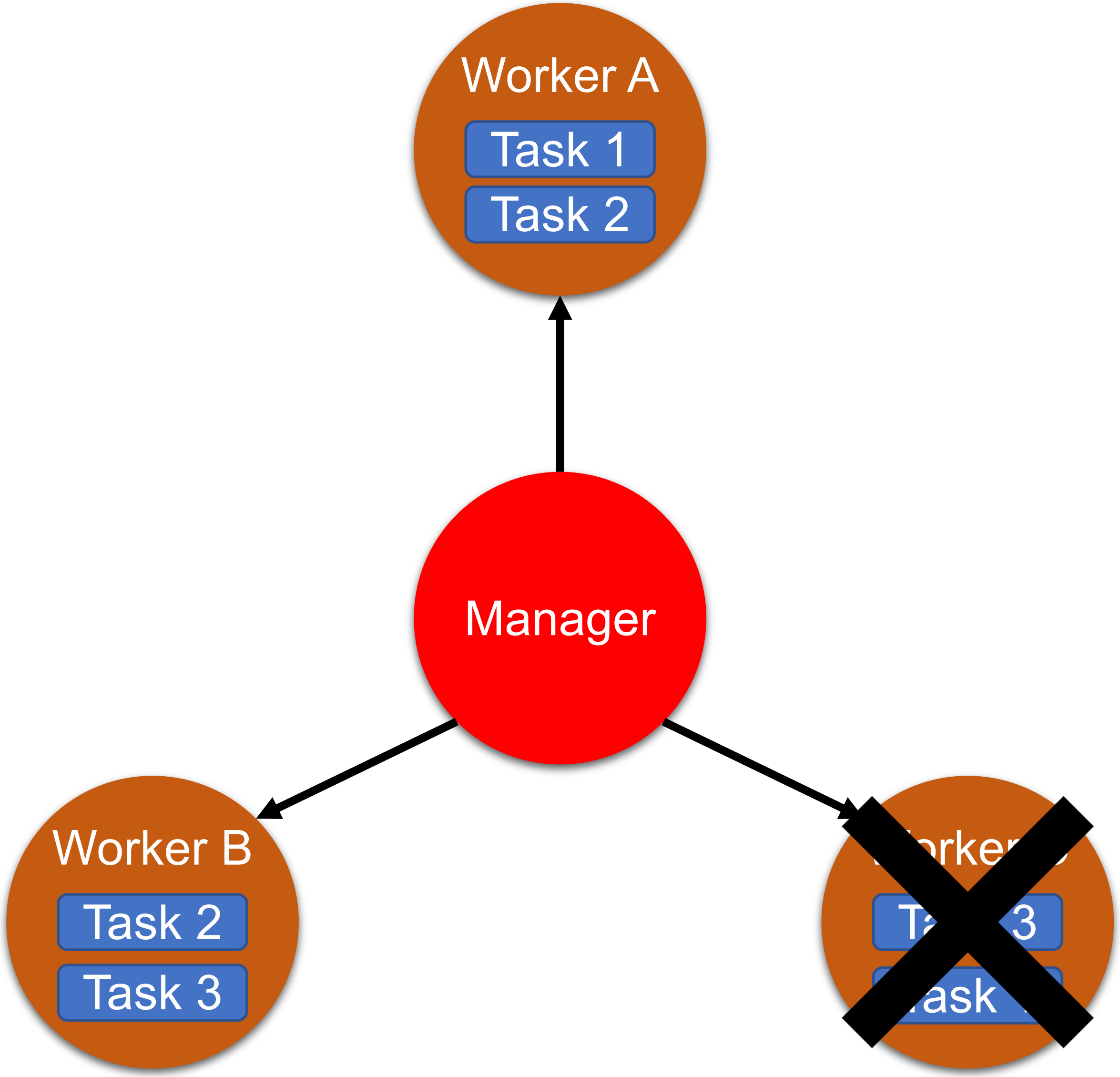
The Master-Slave Architecture of Hadoop



The Master-Slave Architecture of Hadoop



The Master-Slave Architecture of Hadoop



Hadoop Distributed File Systems (HDFS)

- HDFS is a file system that
  - follows master-slave architecture
  - allows us to store data over multiple nodes (machines) ,
  - allows multiple users to access data.
  - just like file systems in your PC

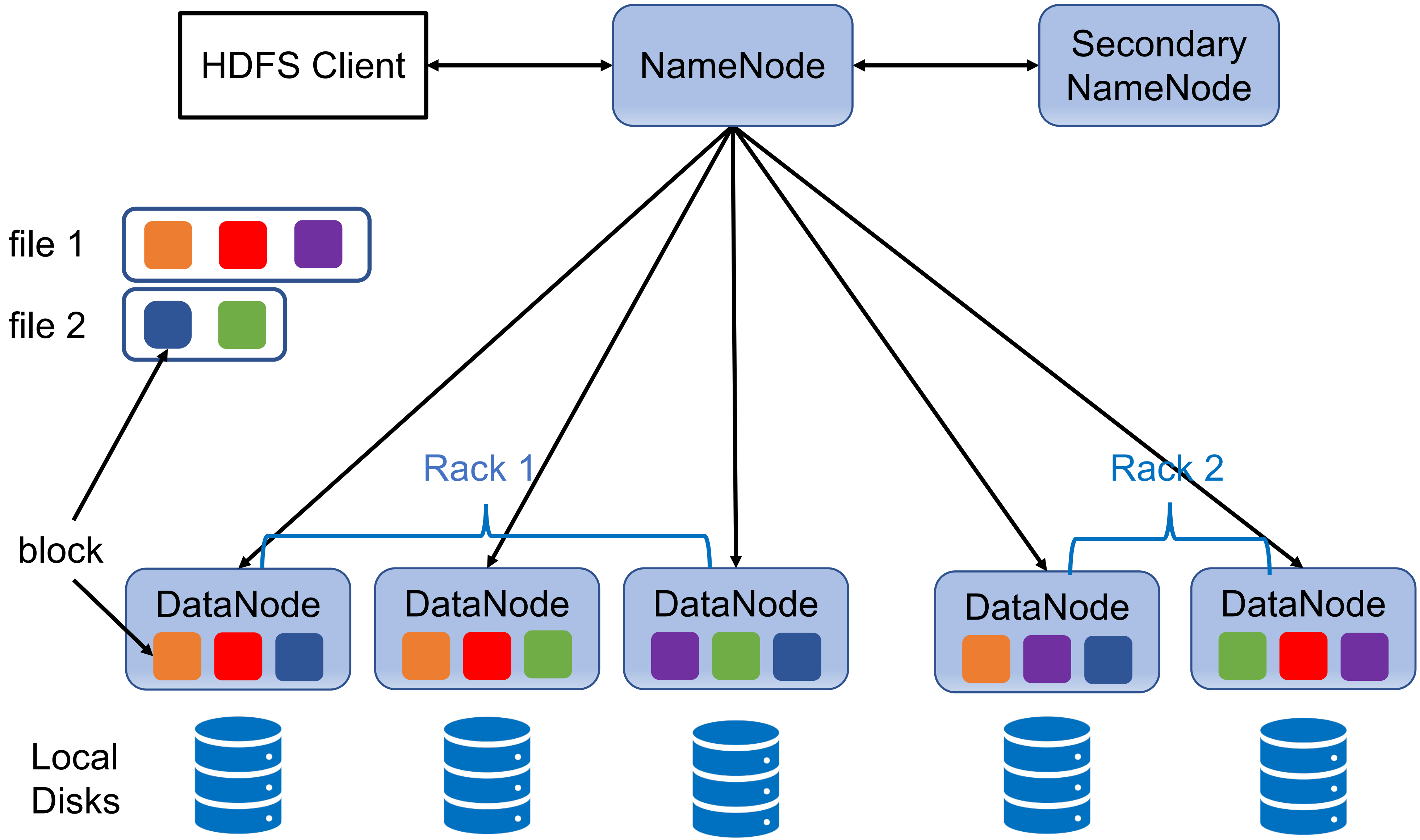
- HDFS supports
  - distributed storage
  - distributed computation
  - horizontal scalability

Vertical Scaling vs. Horizontal Scaling

Vertical Scaling

Horizontal Scaling

HDFS Architecture



NameNode

- NameNode maintains and manages the blocks in the DataNodes (slave nodes).
  - Master node

- Functions:
  - records the metadata of all the files
    - FsImage: file system namespace
    - EditLogs: all the recent modifications
  - records each change to the metadata
  - regularly checks the status of datanodes
  - keeps a record of all the blocks in HDFS
  - if the DataNode fails, handle data recovery

DataNode

- A commodity hardware stores the data
  - Slave node

- Functions
  - stores actual data
  - performs the read and write requests
  - reports the health to NameNode (heartbeat)

NameNode vs. DataNode

	NameNode	DataNode
Quantity	One	Multiple
Role	Master	Slave
Stores	Metadata of files	Blocks
Hardware Requirements	High Memory	High Volume Hard Drive
Failure rate	Lower	Higher
Solution to Failure	Secondary NameNode	Replications

If NameNode failed…

- All the files on HDFS will be lost
  - there’s no way to reconstruct the files from the blocks in DataNodes without the metadata in NameNode

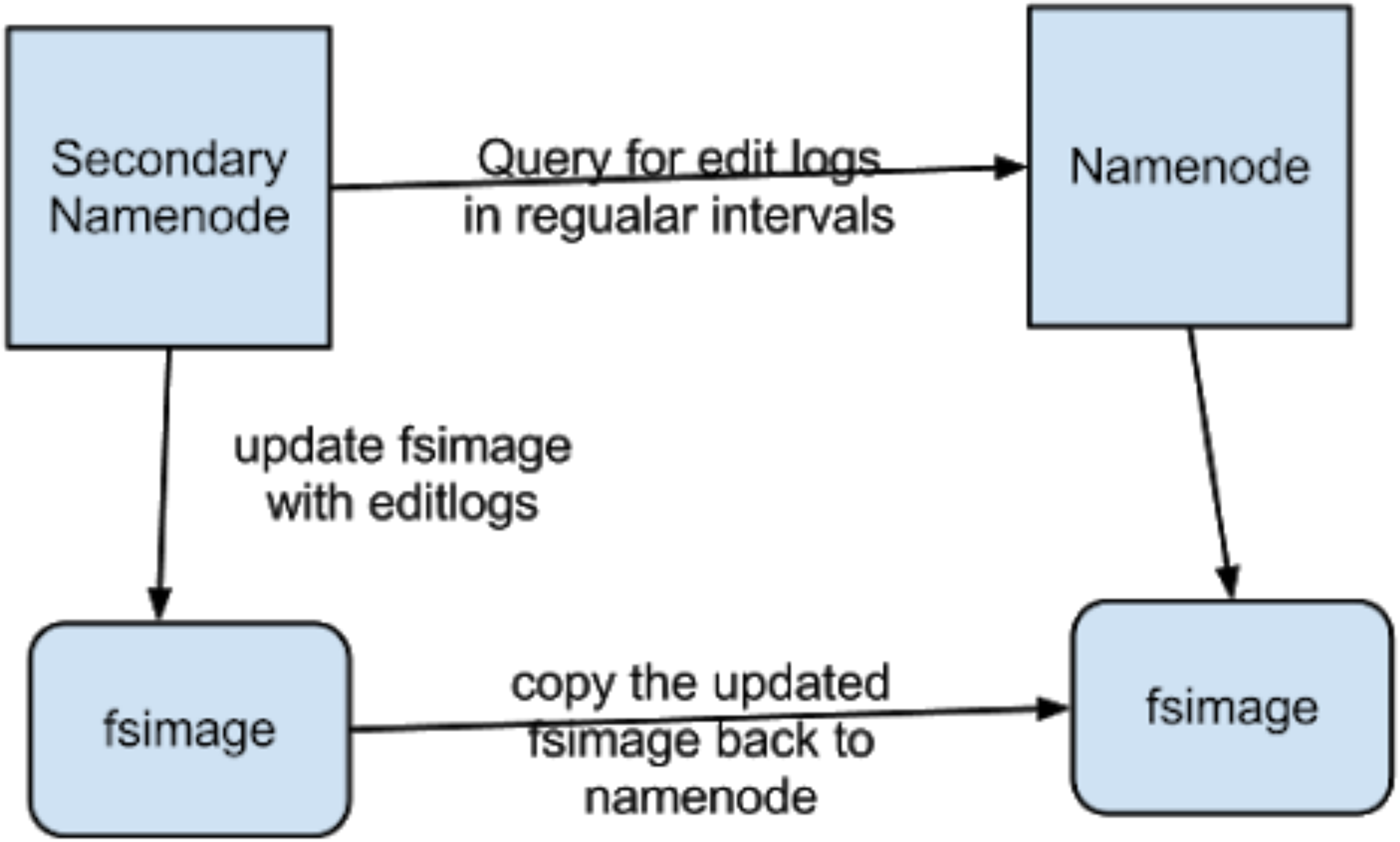
- In order to make NameNode resilient to failure
  - back up metadata in NameNode (with a remote NFS mount)
  - Secondary NameNode

Secondary NameNode

- Take checkpoints of the file system metadata present on NameNode
  - It is not a backup NameNode!

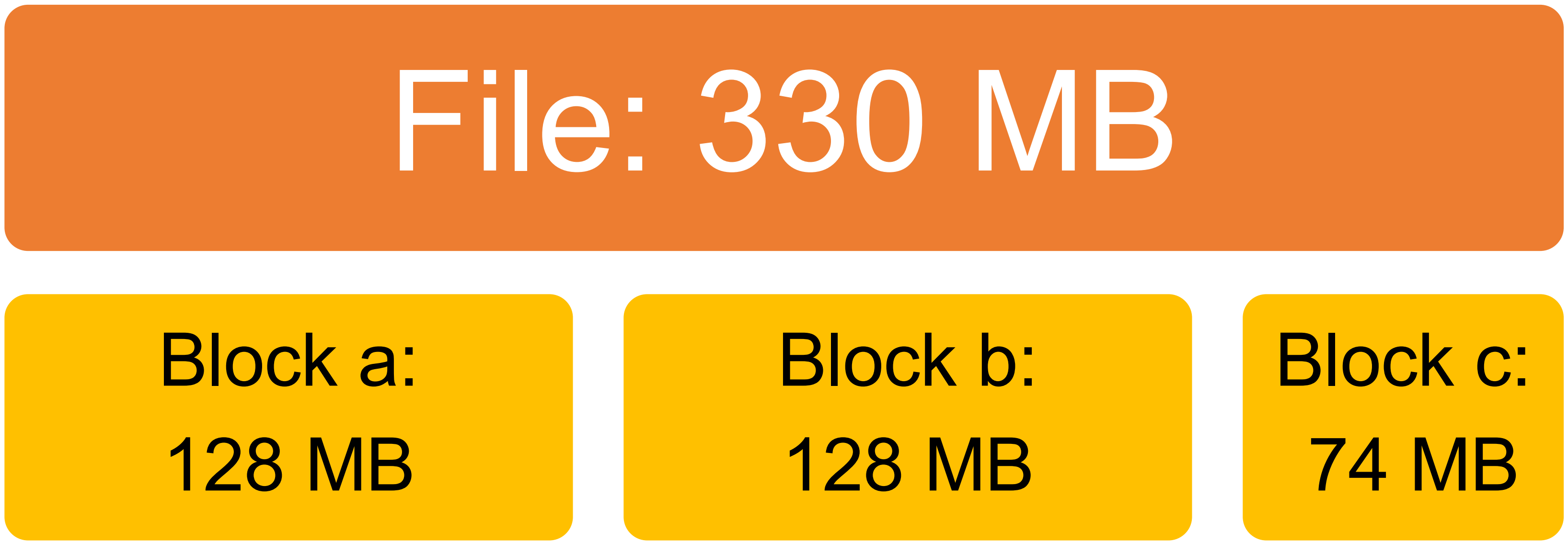
- Functions:
  - Stores a copy of FsImage file and Editlogs
  - Periodically applies Editlogs to FsImage and refreshes the Editlogs.
  - If NameNode is failed, File System metadata can be recovered from the last saved FsImage on the Secondary NameNode.

NameNode vs. Secondary NameNode



Blocks

- Block is a sequence of bytes that stores data
  - Data stores as a set of blocks in HDFS
  - Default block size is 128MB (Hadoop 2.x and 3.x)
  - A file is spitted into multiple blocks



Why Large Block Size?

- HDFS stores huge datasets
- If block size is small (e.g., 4KB in Linux), then the number of blocks is large:
  - too much metadata for NameNode
  - too many seeks affect the read speed
  - harm the performance of MapReduce too

- We don’t recommend using HDFS for small files due to similar reasons.
  - Even a 4KB file will occupy a whole block.

If DataNode Failed…

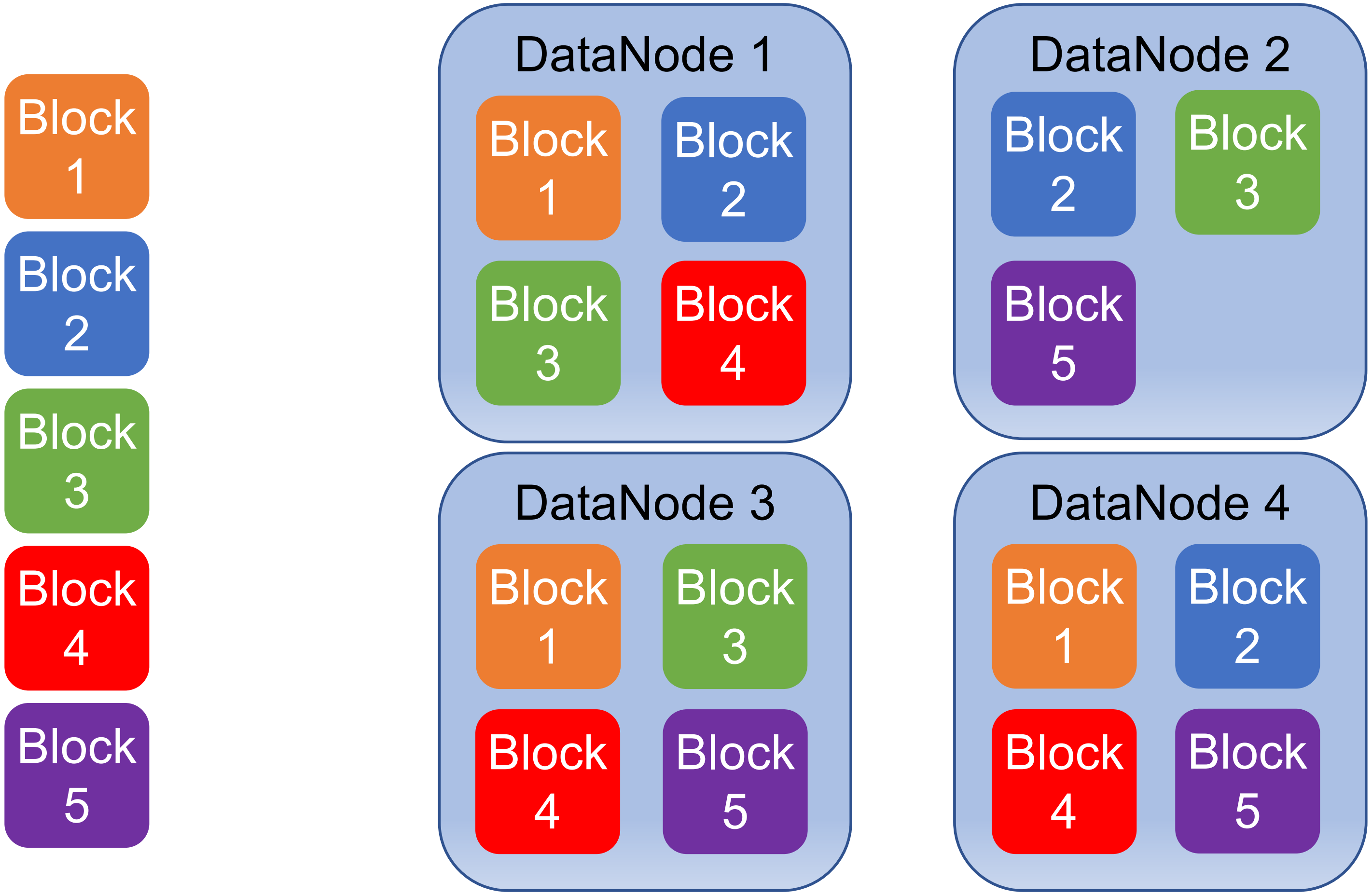
- Commodity hardware fails
  - If NameNode hasn't heard from a DataNode for 10mins, The DataNode is considered dead…
- HDFS guarantees data reliability by generating multiple replications of data
  - each block has 3 replications by default
  - replications will be stored on different DataNodes
  - if blocks were lost due to the failure of a DataNode, they can be recovered from other replications
  - the total consumed space is 3 times the data size
- It also helps to maintain data integrity

File, Block and Replica

- A file contains one or more blocks
  - Blocks are different
  - Depends on the file size and block size
  - $\# = \left\lceil \frac{file\ size}{block\ size} \right\rceil$
- A block has multiple replicas
  - Replicas are the same
  - Depends on the preset replication factor

Replication Management

- Each block is replicated 3 times and stored on different DataNodes



Why default replication factor = 3?

- If 1 replicate
  - DataNode fails, block lost
- Assume
  - # of nodes N = 4000
  - # of blocks R = 1,000,000
  - Node failure rate FPD = 1 per day
- If one node fails, then R/N = 250 blocks are lost
  - E(# of losing blocks in one day) = 250
- Let the number of losing blocks follows Poisson distribution, then
  - Pr[# of losing blocks in one day >= 250] = 0.508

Why default replication factor = 3?

- Assume
  - # of nodes N = 4000
  - Capacity of each node GB = 4000 Gigabytes
  - # of block replicas R = 1,000,000 * 3
  - Node failure rate FPD = 1 per day
  - Replication speed = 1.35 MB per second per node
- If one node fails, B = R/N = 750 replicas/blocks are unavailable
- There are on average S = 2B/(N-1) = 0.38 replicas per node for the blocks in the failed node
- So if second node fails, 0.38 blocks now have only a single replica

Why default replication factor = 3?

- If the third node fails,
  - The probability that it has the only remaining replica of a particular block is
    - Pr[last] = 1/(N-2) = 0.000250
  - The probability that it has none of those replicas is
    - Pr[none] = $(1\text{-}Pr[last])^S$ = 0.999906
  - The probability of losing the last replica of a block is
    - Pr[lose] = 1- Pr[none] = 9.3828E-05

- Recall:
  - N is # of nodes
  - S is the # of replicas per node for the blocks in the first failed node

Why default replication factor = 3?

- Assume # of node failures follows Poisson distribution with rate
  - ω=FPD/(24*3600)=1.1574E-05 per second
- Re-replication is a fully parallel operation on the remaining nodes
  - Recovery (re-create the lost replicas) time is
    - 1000 * GB / MPS / (N-1) = 740.93 seconds
    - Recovery rate μ= 1/ 740.93 per second
  - E(# of failed nodes in 1 sec) =ω/μ = 0.008576
- At any second, the probability of k failed nodes follows Poisson distribution
  - Pr[0 failed node] = 0.991461
  - Pr[1 failed node] = 0.008502
  - Pr[2 or more failed nodes] = 1- Pr(0) - Pr(1) = 0.00003656
- Thus, the rate of third failure is
  - Pr[2 or more failed nodes] *ω= 4.2315E-10 per sec
- The rate of losing a data block is
  - λ=Pr[2 or more failed nodes] *ω* Pr[lose] = 3.9703E-14

Why default replication factor = 3?

- Recall that in one second, the rate of losing a data block is
  - λ = 3.9703E-14 per second
- According to exponential distribution, we have:
  - Pr[losing a block in one year] = $1-e^{-\lambda t}$ = 0.00000125
    - t = 365*24*3600

- So replication factor = 3 is good enough.

What about Simultaneous Failure?

- If one node fails, we've lost B (first) replicas
- If two nodes fail, we've lost some second replicas and more first replicas
- If three nodes fail, we've lost some third replicas, some second replicas and some first replicas
- …

What about Simultaneous Failure?

- Assume k of N nodes have failed simultaneously, let there be
  - L1(k,N) blocks have lost one replica
  - L2(k,N) blocks have lost two replicas
  - L3(k,N) blocks have lost three replicas
  - B is # of unavailable blocks if one node fails
- k=0:
  - L1(0,N) = L2(0,N) = L3(0,N) = 0
- k=1:
  - L1(1,N) = B
  - L2(1,N) = L3(1,N) = 0
- k=2:
  - L1(2,N) = 2B-2*L2(2,N)
  - L2(2,N) = 2*L1(1,N)/(N-1)
  - L3(2,N) = 0
- k=3:
  - L1(3,N) = 3B-2*L2(3,N)-3*L3(3,N)
  - L2(3,N) = 2*L1(2,N)/(N-2)+L2(2,N)- L2(2,N)/(N-2)
  - L3(3,N) = L2(2,N)/(N-2)

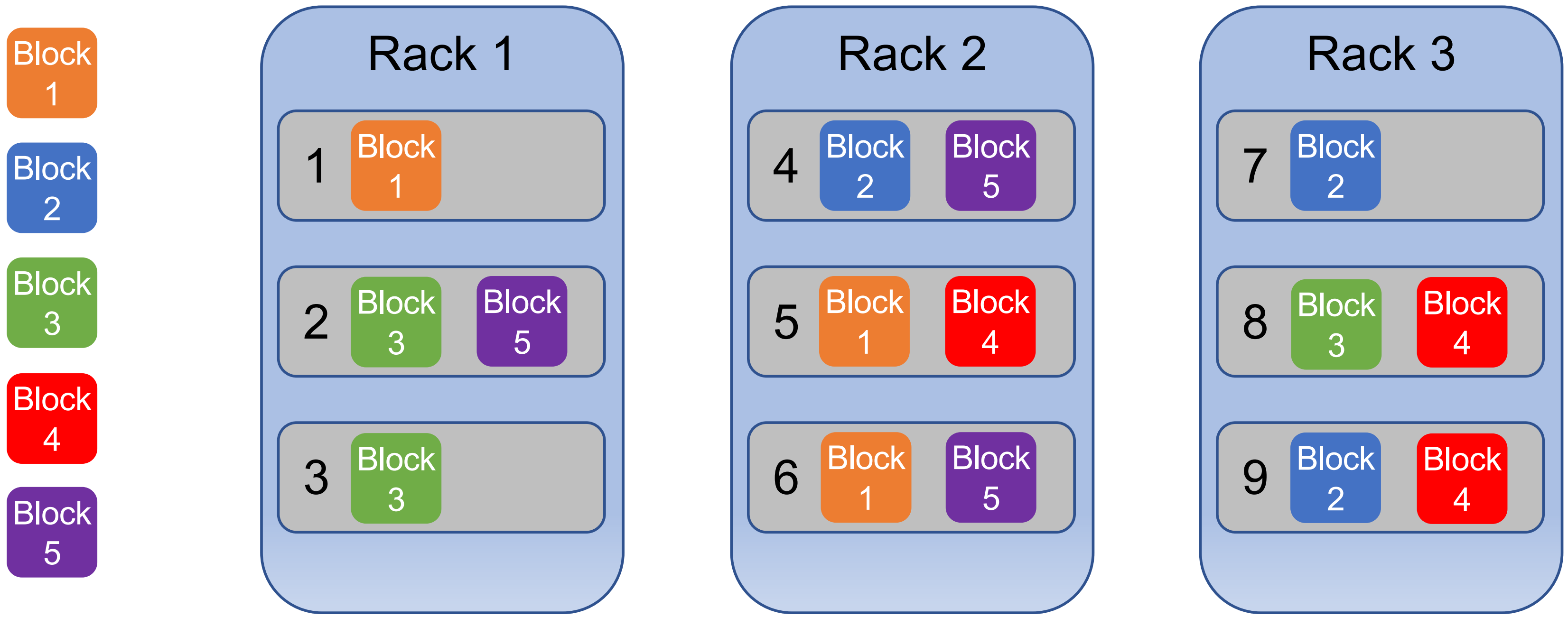
What about Simultaneous Failure?

- In general
  - L1(k,N) = k*B-2*L2(k,N)-3*L3(k,N)
  - L2(k,N) = 2*L1(k-1,N)/(N-k+1)+L2(k-1,N)-L2(k-1,N)/(N-k+1)
  - L3(k,N) = L2(k-1,N)/(N-k+1)+L3(k-1,N)
- Let N = 4000, B = 750, we have

Failed Nodes	1st replicas lost	2nd replicas lost	3rd replicas lost
50	36,587	454	2
100	71,332	1,811	15
150	104,272	4,037	52
200	135,441	7,095	123

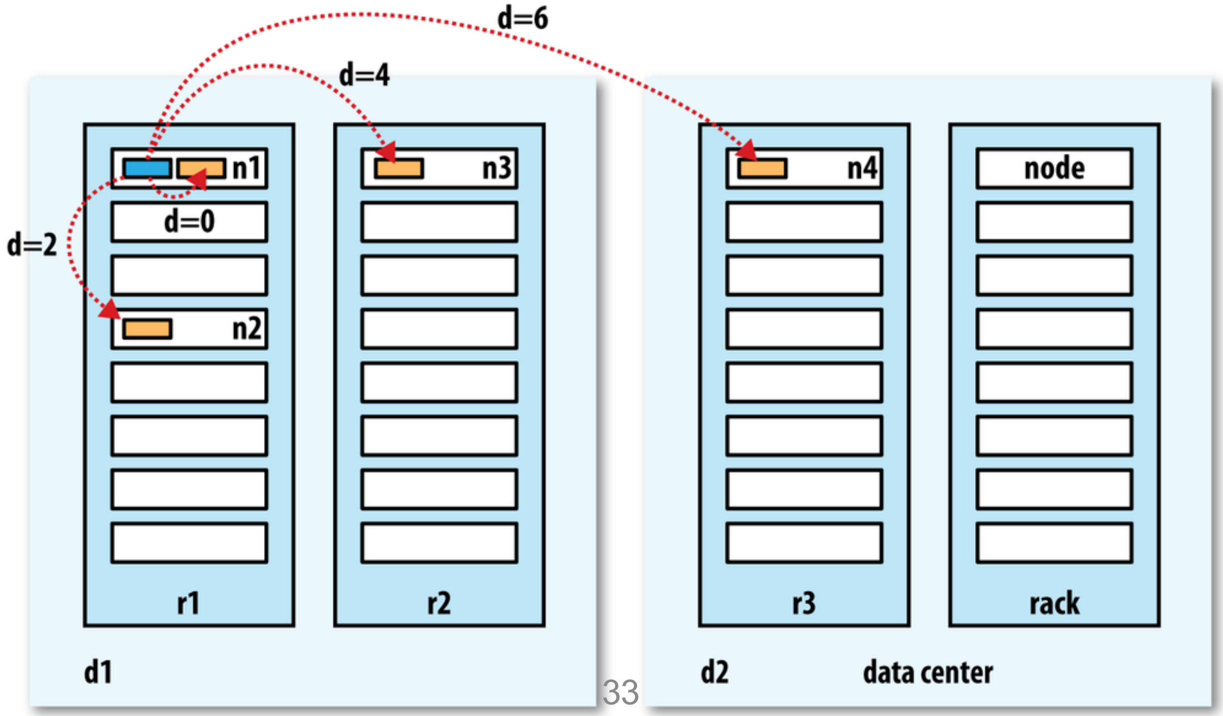
Rack Awareness Algorithm

- If the replication factor is 3:
  - 1st replica will be stored on the local DataNode
  - 2nd on a different rack from the first.
  - 3rd on the same rack as 2nd, but on a different node.



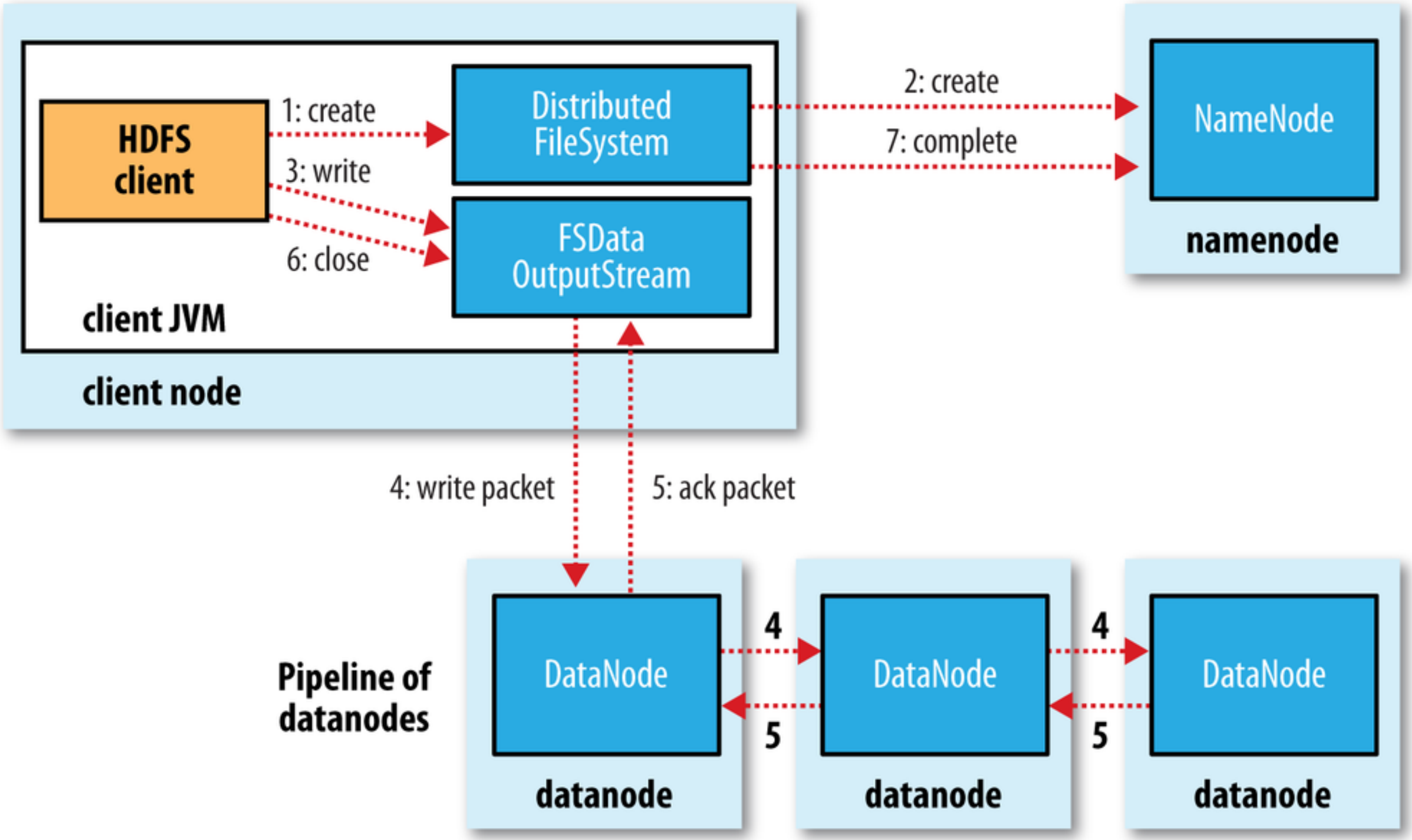
Why Rack Awareness?

- Reduce latency
  - Write: to 2 racks instead of 3 per block
  - Read: blocks from multiple racks
- Fault tolerance
  - Never put your eggs in the same basket



Write in HDFS

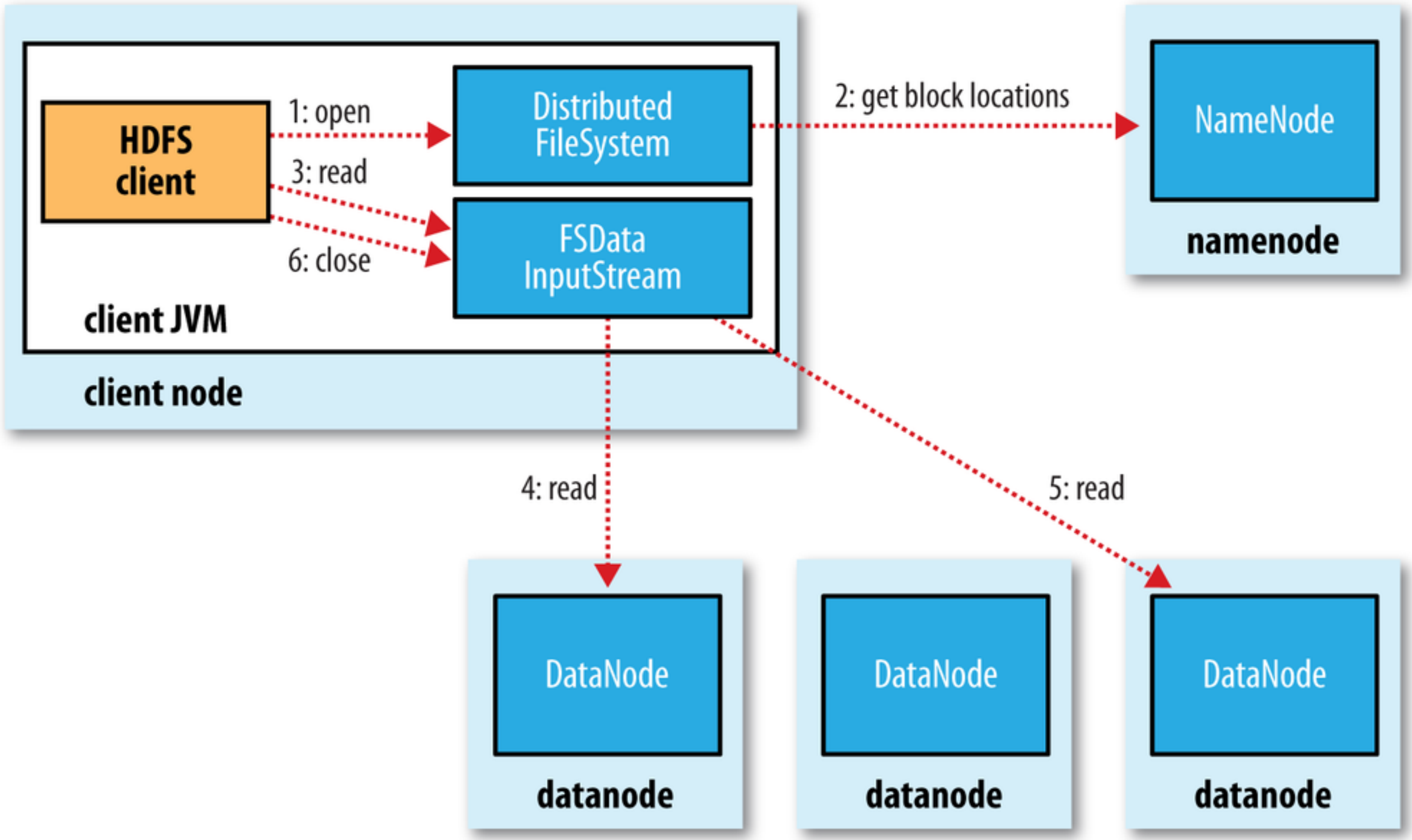
- Create file – Write file – Close file



Write in HDFS

- There is only single writer allowed at any time
- The blocks are writing simultaneously
- For one block, the replications are replicating sequentially
- The choose of DataNodes is random, based on replication management policy, rack awareness, …

Read in HDFS



HDFS client
1: open
3: read
6: close
Distributed FileSystem
FSData InputStream
client JVM
client node
2: get block locations
NameNode
namenode
4: read
5: read
DataNode
datanode
DataNode
datanode
DataNode
datanode

Read in HDFS

- Multiple readers are allowed to read at the same time
- The blocks are reading simultaneously
- Always choose the closest DataNodes to the client (based on the network topology)
- Handling errors and corrupted blocks
  - avoid visiting the dataNode again
  - report to NameNode

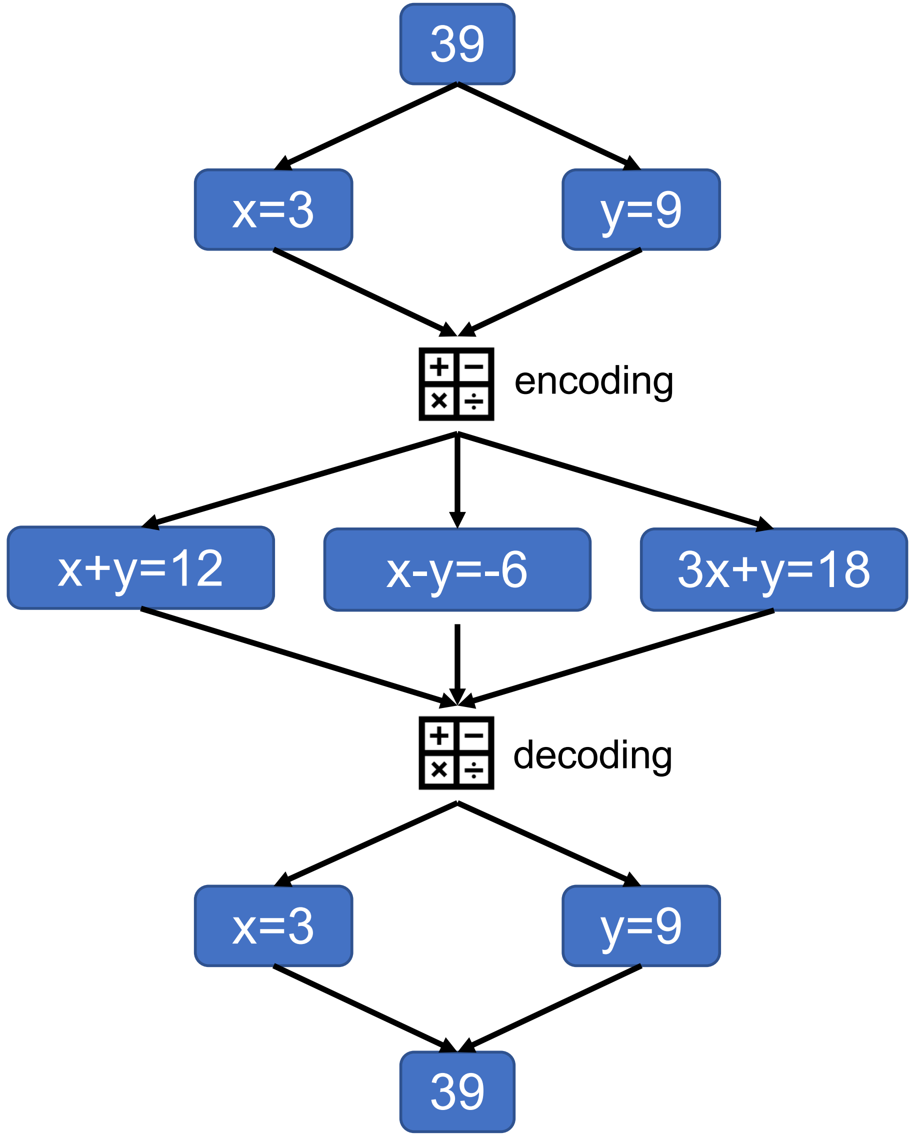
HDFS Erasure Coding

- Drawback of replication
  - space overhead (e.g., 200%)
  - rarely accessed replicas

- Erasure coding
  - same or better level of fault-tolerance
  - much less overhead
  - used in RAID

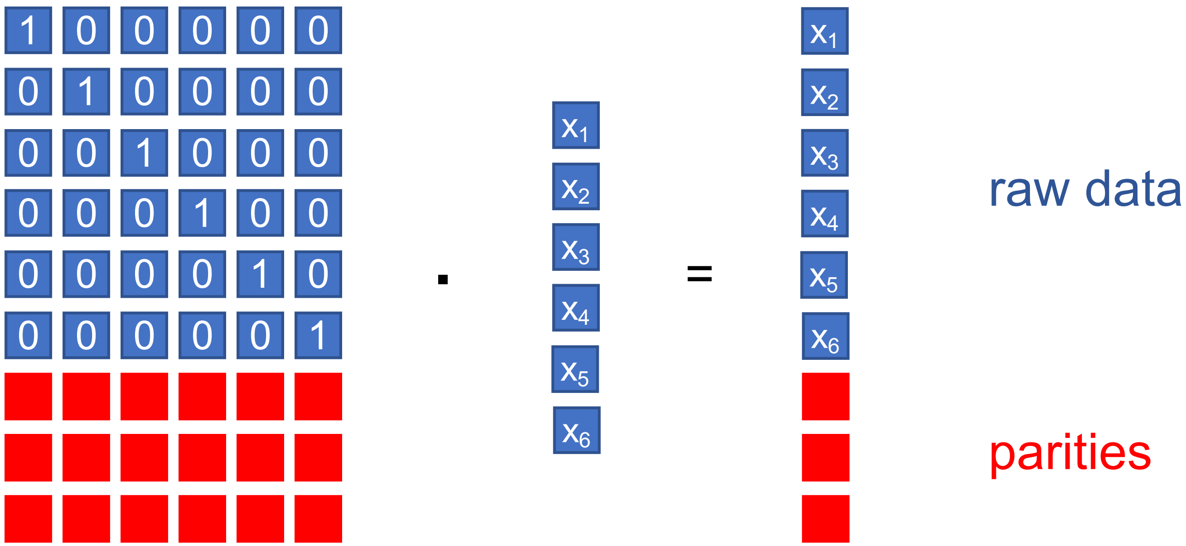
Erasure Coding: Idea

- We can decode 39 using any two of the three equations
  - lose one equation does not matter, and we can recover it easily!



Erasure Coding: (6,3)-Reed-Solomon

- Now consider $X = [x_1, \cdots, x_6]^T$ and $G = \begin{bmatrix} I_6 \\ g_1 \\ g_2 \\ g_3 \end{bmatrix}$
  - a matrix with any 6 rows from G has full rank.
- Then we can have $P = G \cdot X$
- We can recover X using any 6 rows from G and P
  - $X = G'^{-1} \cdot P'$

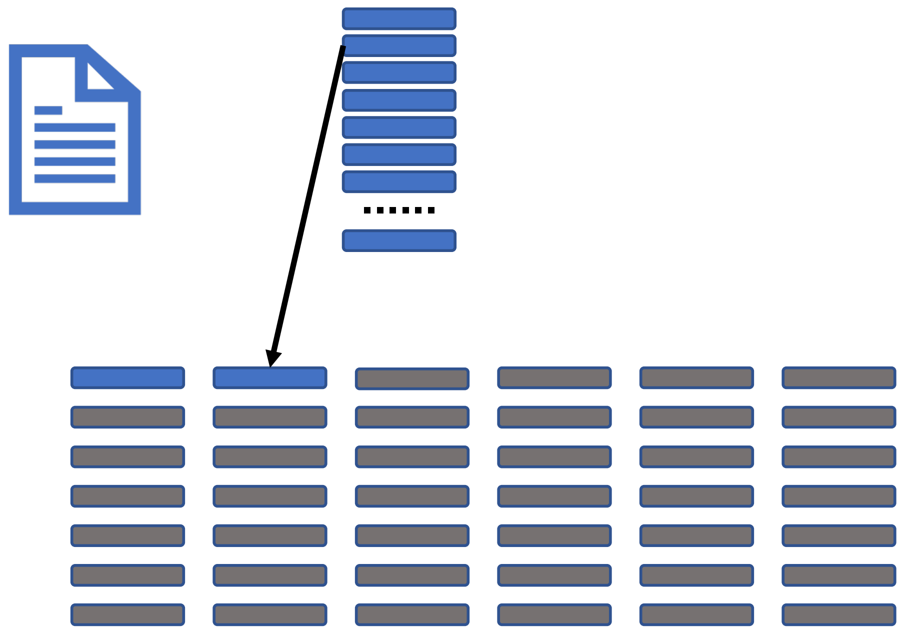


Striped Block Management

- Raw data is striped into cells
  - each cell is 64KB
- The cells are written into blocks in order
  - with striped layout

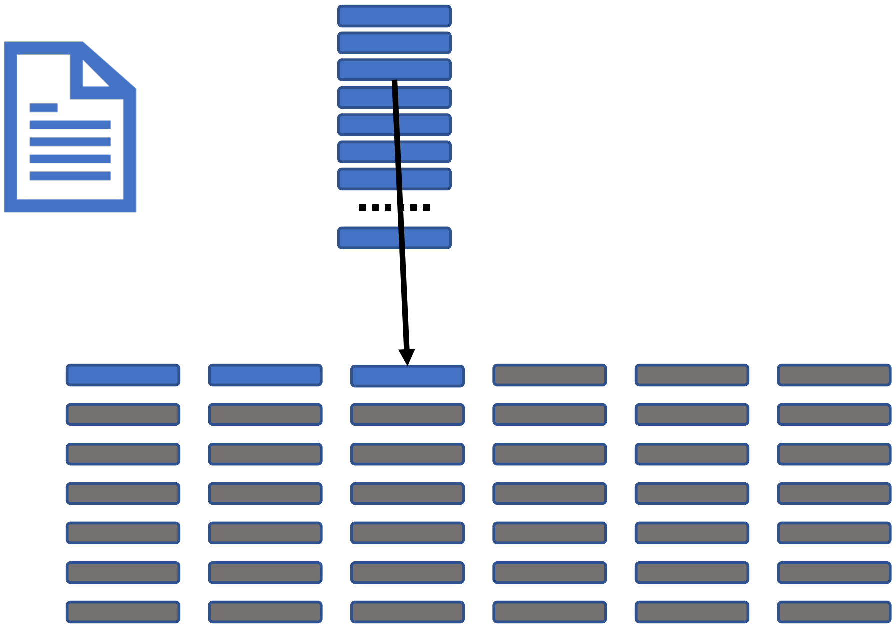
Striped Block Management

- Raw data is striped into cells
  - each cell is 64KB
- The cells are written into blocks in order
  - with striped layout



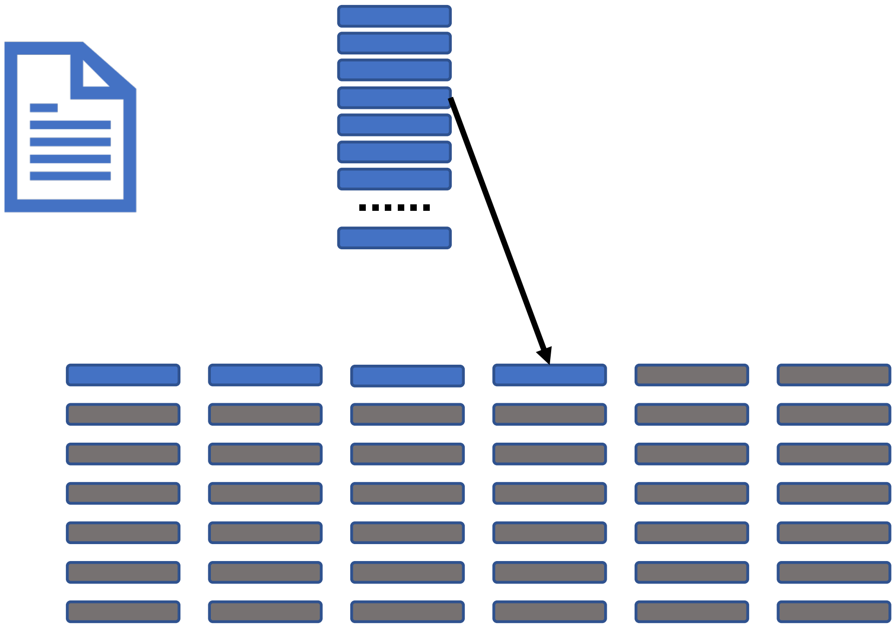
Striped Block Management

- Raw data is striped into cells
  - each cell is 64KB
- The cells are written into blocks in order
  - with striped layout



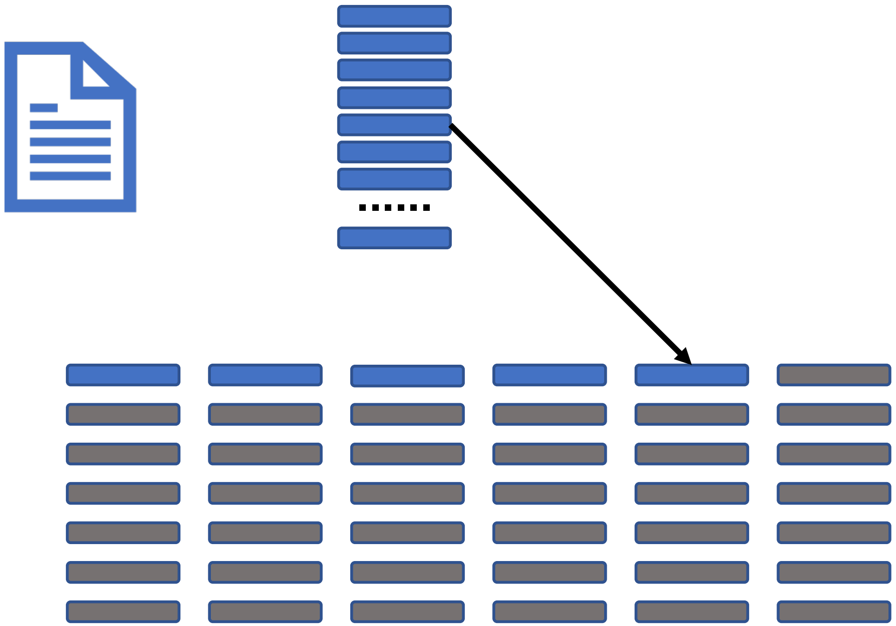
Striped Block Management

- Raw data is striped into cells
  - each cell is 64KB
- The cells are written into blocks in order
  - with striped layout



Striped Block Management

- Raw data is striped into cells
  - each cell is 64KB
- The cells are written into blocks in order
  - with striped layout

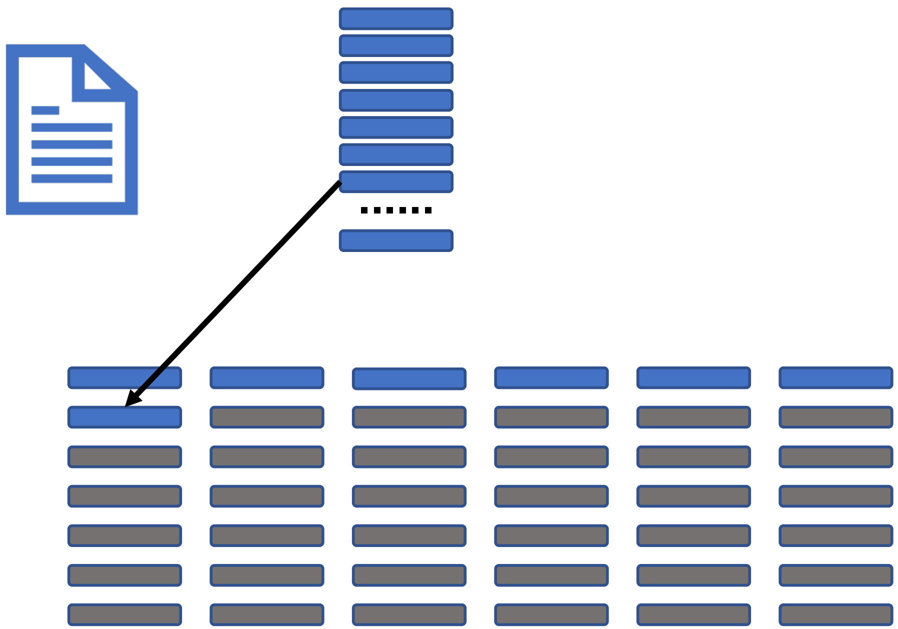


Striped Block Management

- Raw data is striped into cells
  - each cell is 64KB
- The cells are written into blocks in order
  - with striped layout

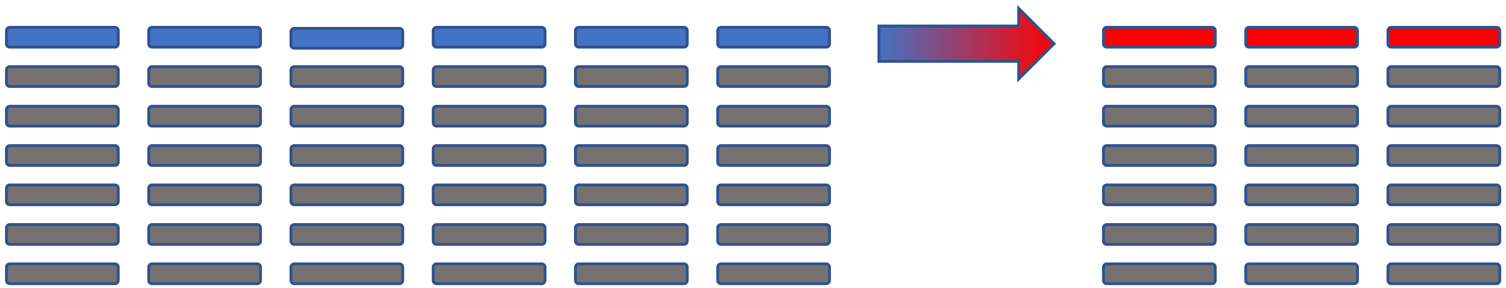
Striped Block Management

- Raw data is striped into cells
  - each cell is 64KB
- The cells are written into blocks in order
  - with striped layout



Striped Block Management

- Use six cells to calculate three parities
- Six cells and three parities form a stripe



Striped Block Management

- Use six cells to calculate three parities
- Six cells and three parities form a stripe

Striped Block Management

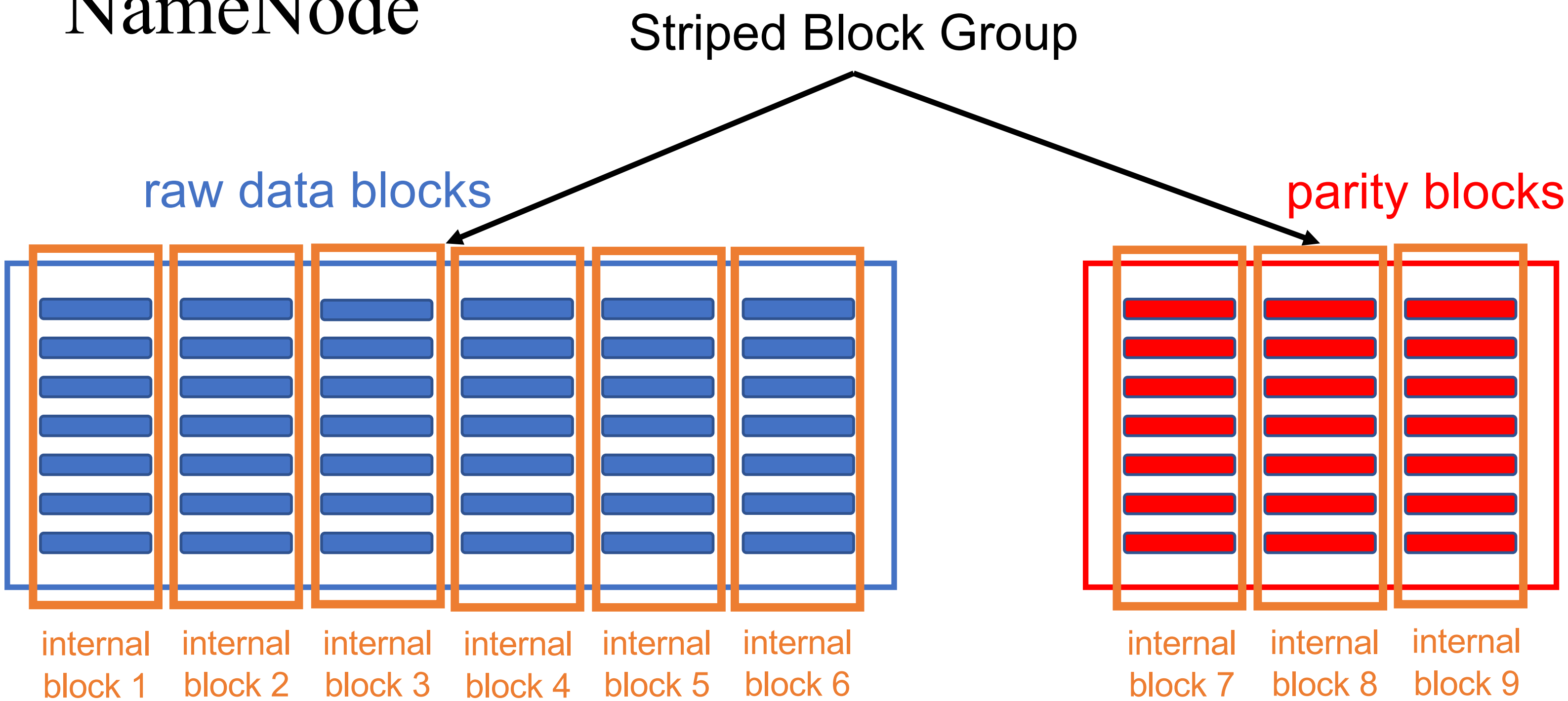
- Use six cells to calculate three parities
- Six cells and three parities form a stripe

Striped Block Management

- Use six cells to calculate three parities
- Six cells and three parities form a stripe

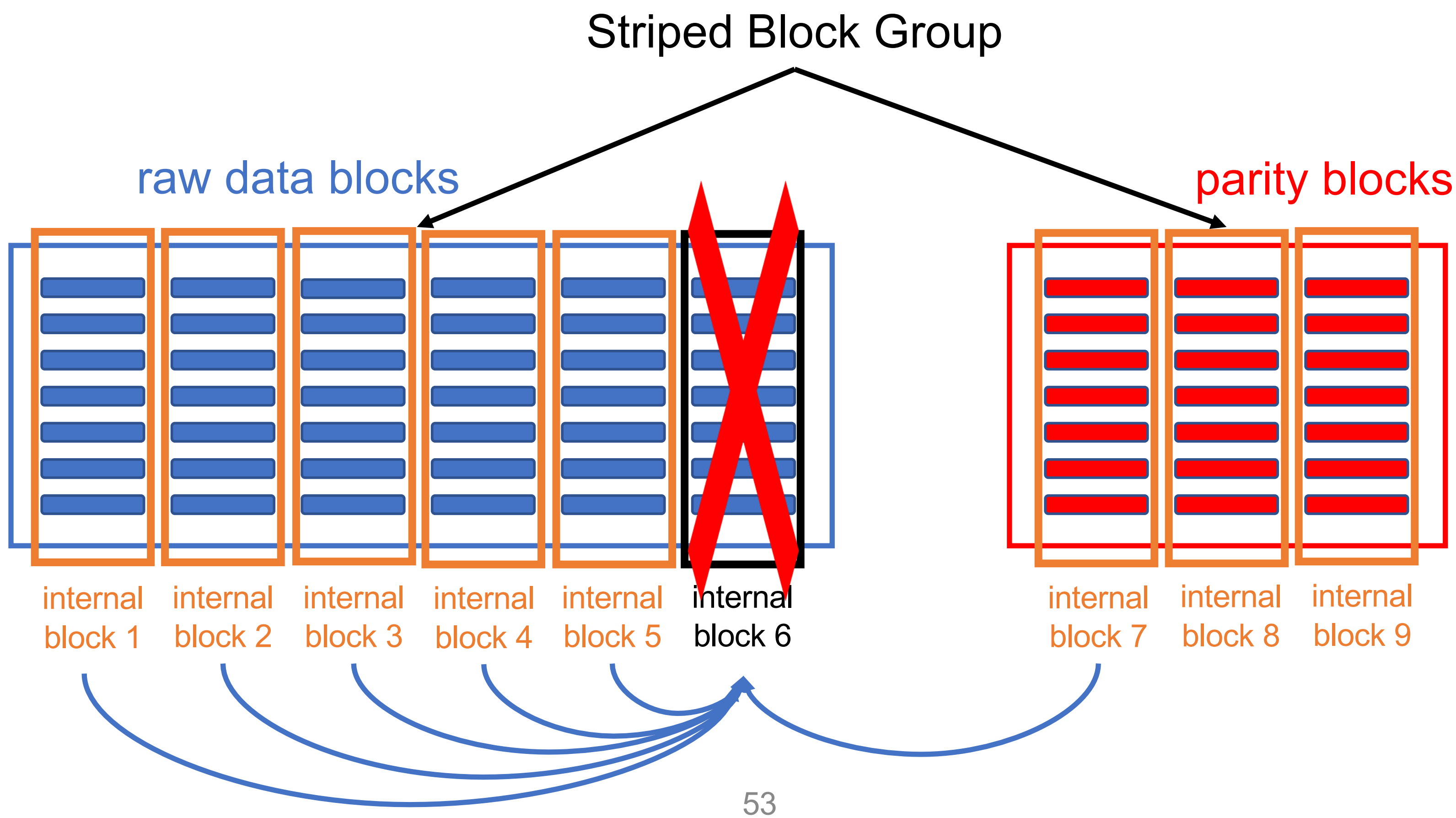
Striped Block Management

- Block group
  - Contains 6 raw data blocks and 3 parity blocks
  - The blocks will be stored in different DataNodes
  - Information of the block group will be stored in NameNode



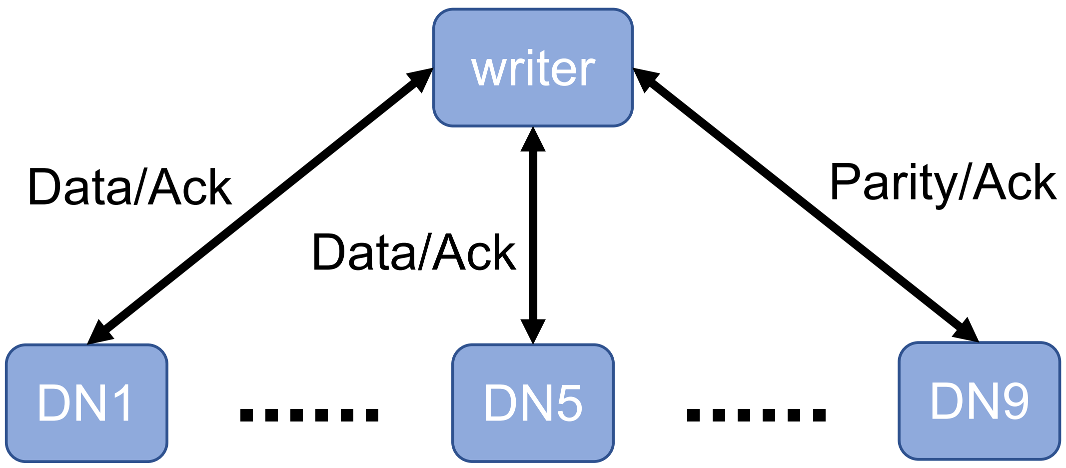
When a (or more?) node fails…

- We can recover the data from any 6 internal blocks



Parallel write

- Client writes a block group of 9 DataNodes simultaneously

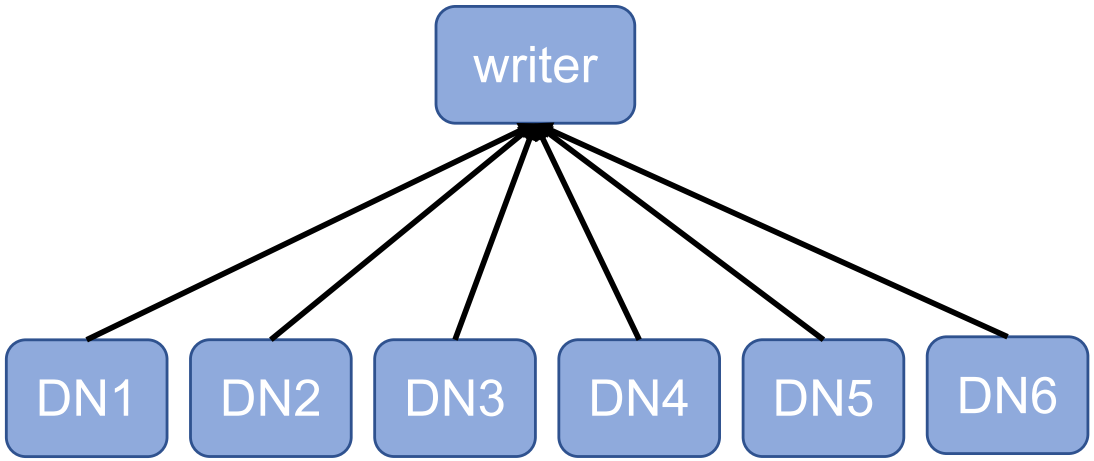


Handle Write Failure

- Client ignores the failed DataNode and continue writing
- Can tolerant up to 3 failures
- Missing blocks will be reconstructed later

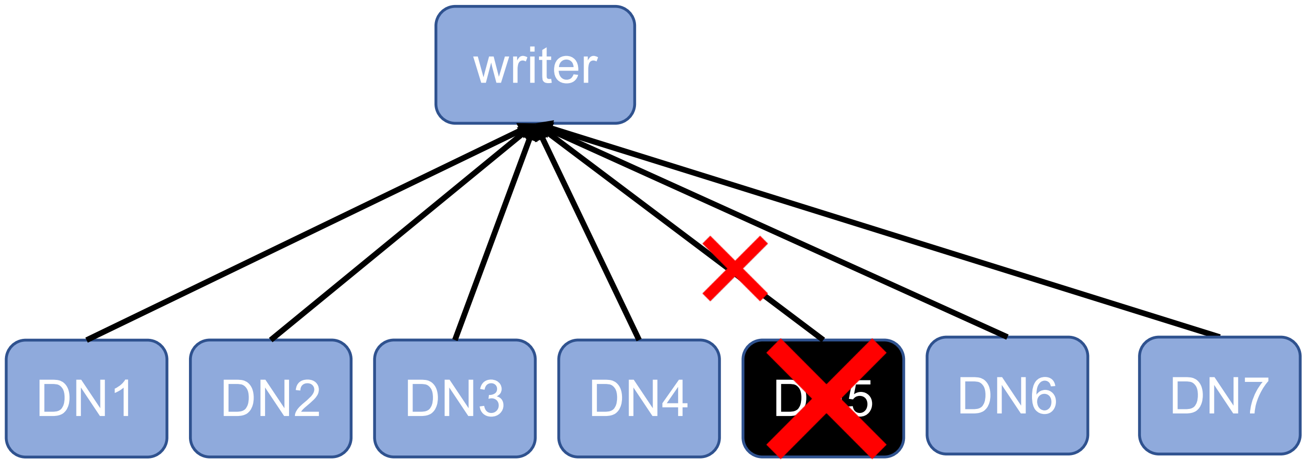
Read

- Parallelly read from 6 DataNodes with data blocks



Handle Read Failure

- Continue reading from any of the remaining DataNodes
- Reconstruct the failed nodes later



Replication vs. Erasure Coding

- EC is better for large and rarely accessed files.
  - HDFS users and admins can turn on and off erasure coding for individual files or directories.

	Replication	Erasure Coding
storage overhead	High	Low
data durability	Yes	Yes (better)
data locality	Yes	No
write performance	Good	Poor
read performance	Good	Poor
recovery cost	Low	High

3-Replication vs. (6,3)-RS

	3-Replication	(6,3)-RS
Durability		
Maximum Toleration	2	3
Disk Space Consumption		
Data: n bytes	3n	1.5n
Number of Client-DataNode connections		
Write	1	9
Read	1	6

3-Replication vs. (6,3)-RS

- Number of blocks required to read the data

<table>
<tr><th># of Blocks</th><th>3-Replication</th><th>(6,3)-RS</th></tr>
<tr><td>1</td><td>1</td><td rowspan="6">6</td></tr>
<tr><td>2</td><td>2</td></tr>
<tr><td>3</td><td>3</td></tr>
<tr><td>4</td><td>4</td></tr>
<tr><td>5</td><td>5</td></tr>
<tr><td>6</td><td>6</td></tr>
</table>