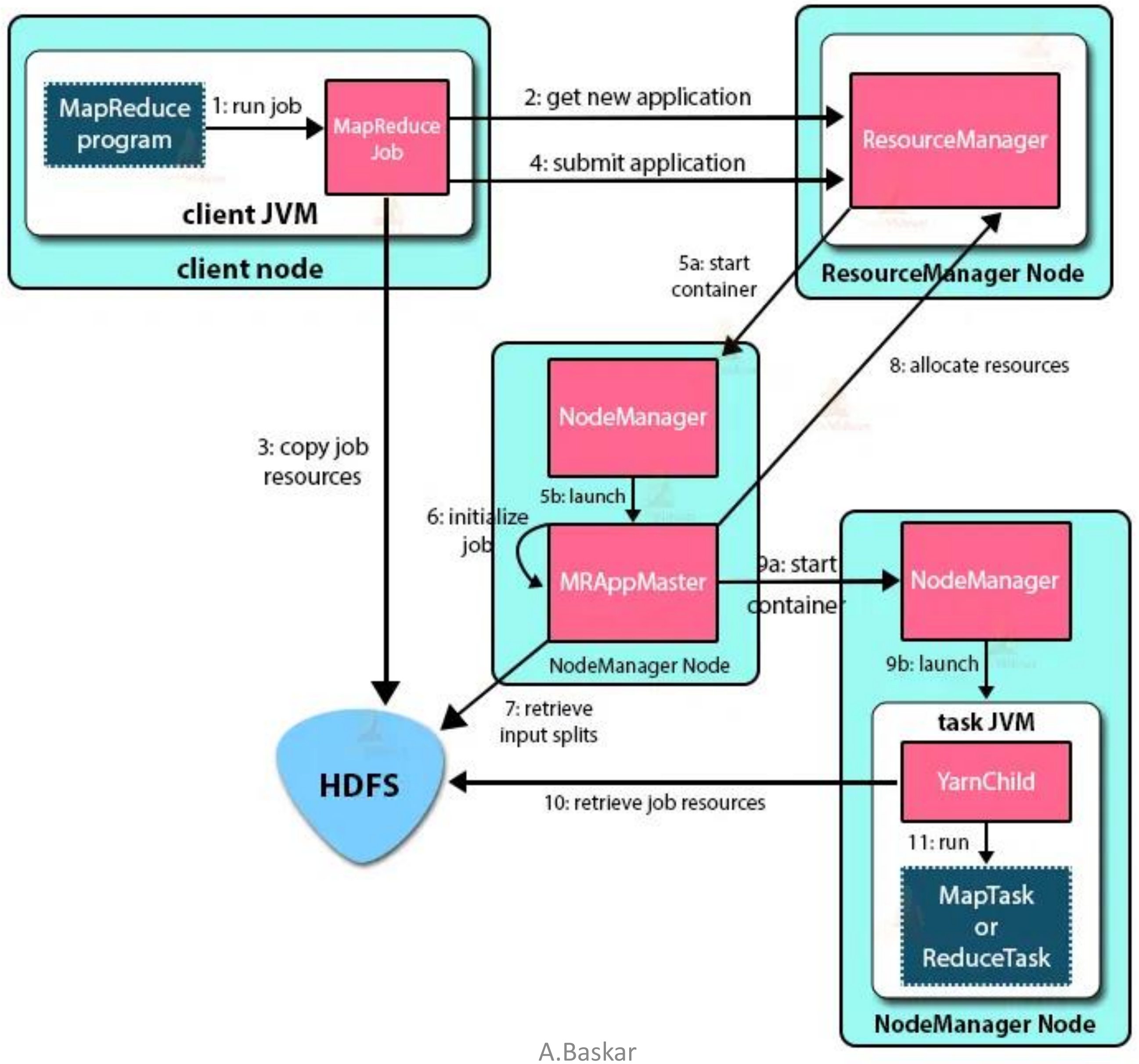
How Hadoop Works – Understand the Working of Hadoop

15CSE334- Bigdata Analytiics

A.Baskar

Internal Working of Hadoop



Hadoop Distributed File system(HDFS)

1. Storage component of Hadoop.
2. Distributed File System.
3. Modelled after Google File System.
4. Optimized for high throughput (HDFS leverages large block size and
5. moves computation where data is stored).
6. You can replicate a file for a configured number of times, which is tolerant in terms of both software and hardware.
7. Re-replicates data blocks automatically on nodes that have failed.
8. You can realize the power of HDFS when you perform read or write on large files (gigabytes and larger).
9. Sits on top of native file system such as ext3 and ext4, which is described

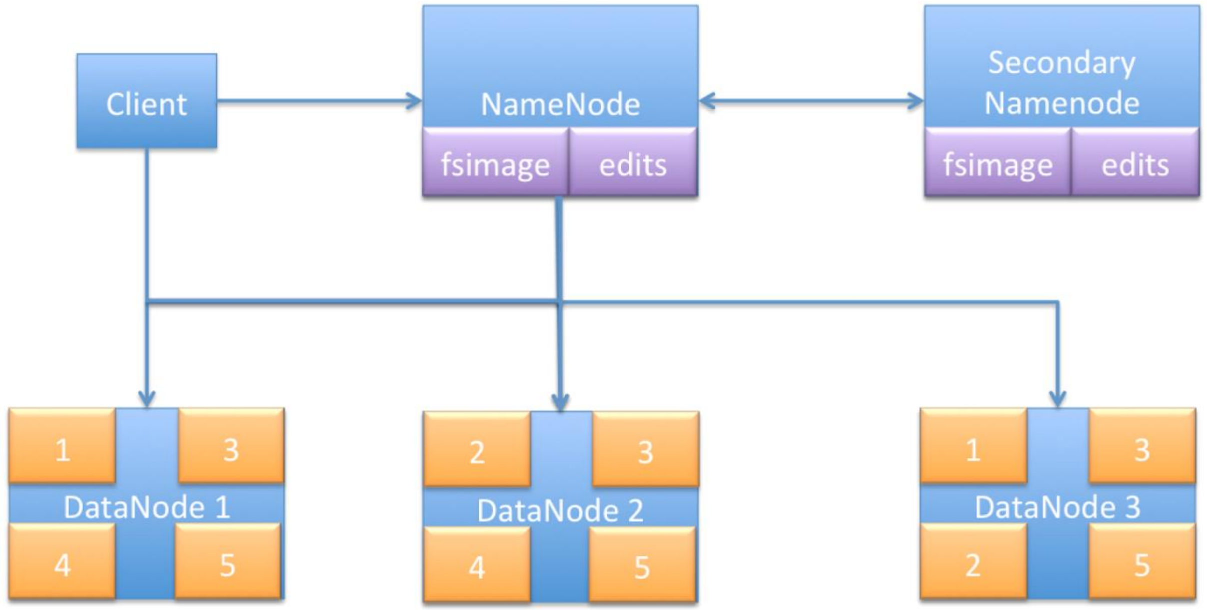
HDFS Characteristics

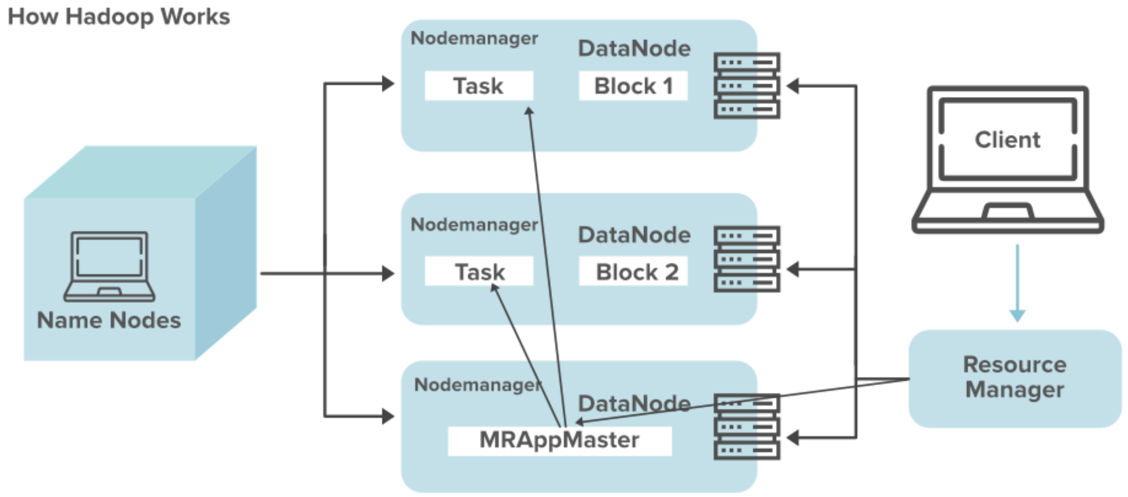
- Streaming data access
- Batch processing rather than interactive user access.
- Large data sets and files: gigabytes to terabytes size
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
- Tens of millions of files in a single instance
- Write-once-read-many: a file once created,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

HDFS Daemons

In multitasking computer operating systems, a daemon is **a computer program that runs as a background process, rather than being under the direct control of an interactive user**.

- Name node
- Data Node
- Secondary name node





HDFS Daemons- Name node

- **NameNode is the master node in Hadoop Distributed File System**.
- It controls all the meta data for the cluster. Eg - what blocks make up a file, and what data nodes those blocks are stored on
- HDFS breaks large data into smaller pieces called **Blocks.**
- **Default block size is 64MB**
- NN uses **RACKID**
- Rack is a collection of data nodes within cluster.
- NN keeps tracks of blocks of a file as it is placed on various Data nodes.
- NN manages file related operations such as read, write, create and delete.
- Its main job is managing the **File System Namespace.**

File system Namespace

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

Name node

- Name node uses:

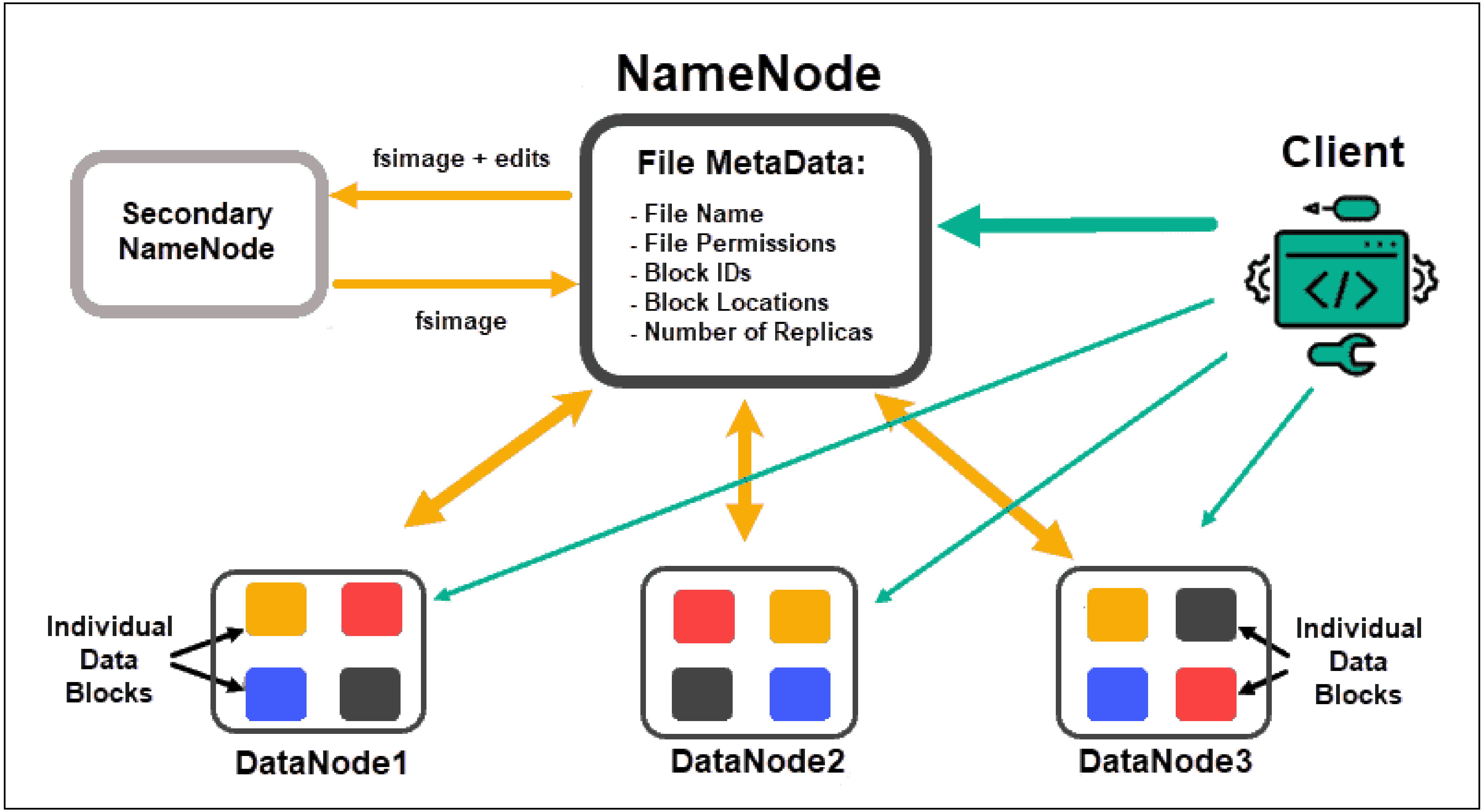
1. **EditLog**: To record every transactions that happens to the file system metadata.
2. **FsImage**: The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in this file.

When NN starts,

1. It reads FsImage and EditLog from local disk and applies to all transactions from the EditLog to in-memory representation of the FsImage.
2. .Then it flushes out new version of FsImage on disk and truncates older EditLog because the changes are updated in the FsImage.
3. Note:

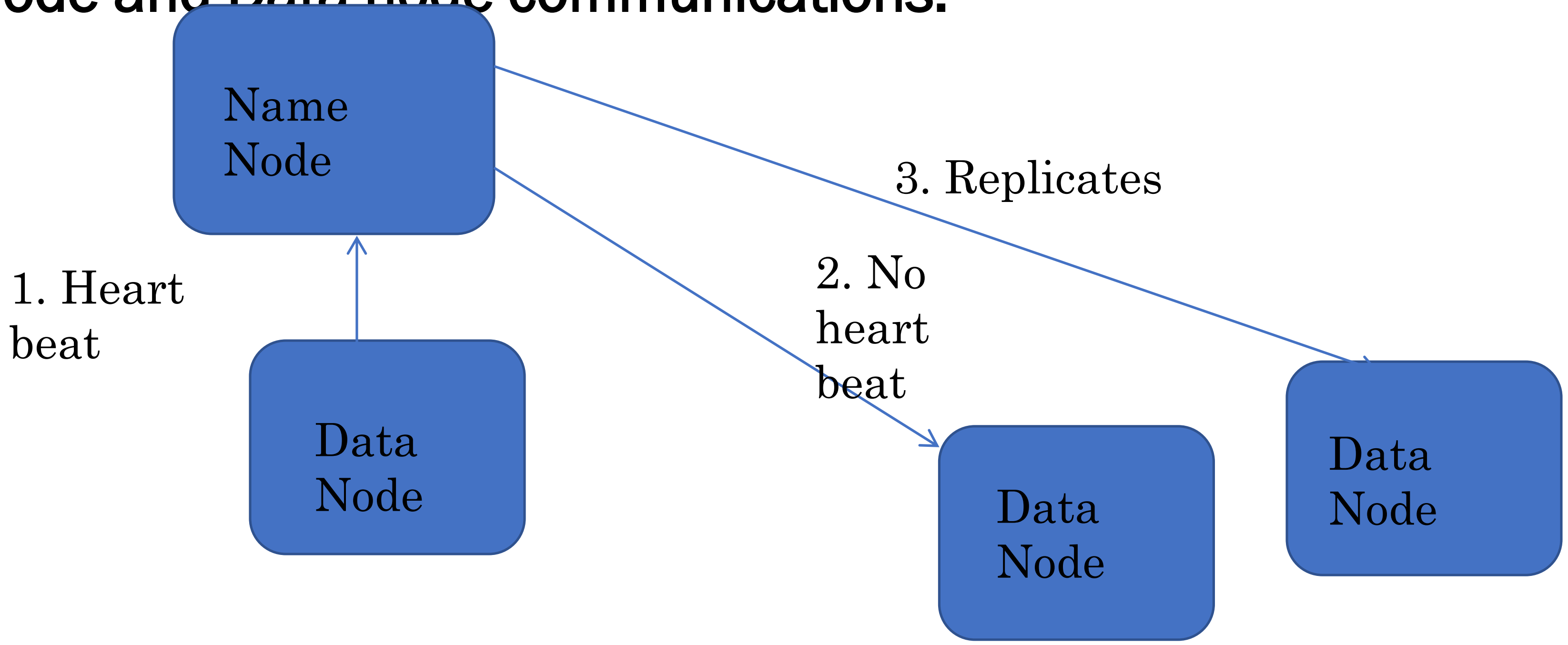
   Default Block size : 64 MB

   Default RF : 3


NameNode
File MetaData:
- File Name
- File Permissions
- Block IDs
- Block Locations
- Number of Replicas
Secondary NameNode
fsimage + edits
fsimage
Client
Individual Data Blocks
DataNode1
DataNode2
DataNode3
Individual Data Blocks

Data node

- where HDFS actually stores the data.
- Multiple data nodes per cluster.
- It stores each block of HDFS data in a separate file.
- Read/Write operations.
- **Name node and Data node communications.**

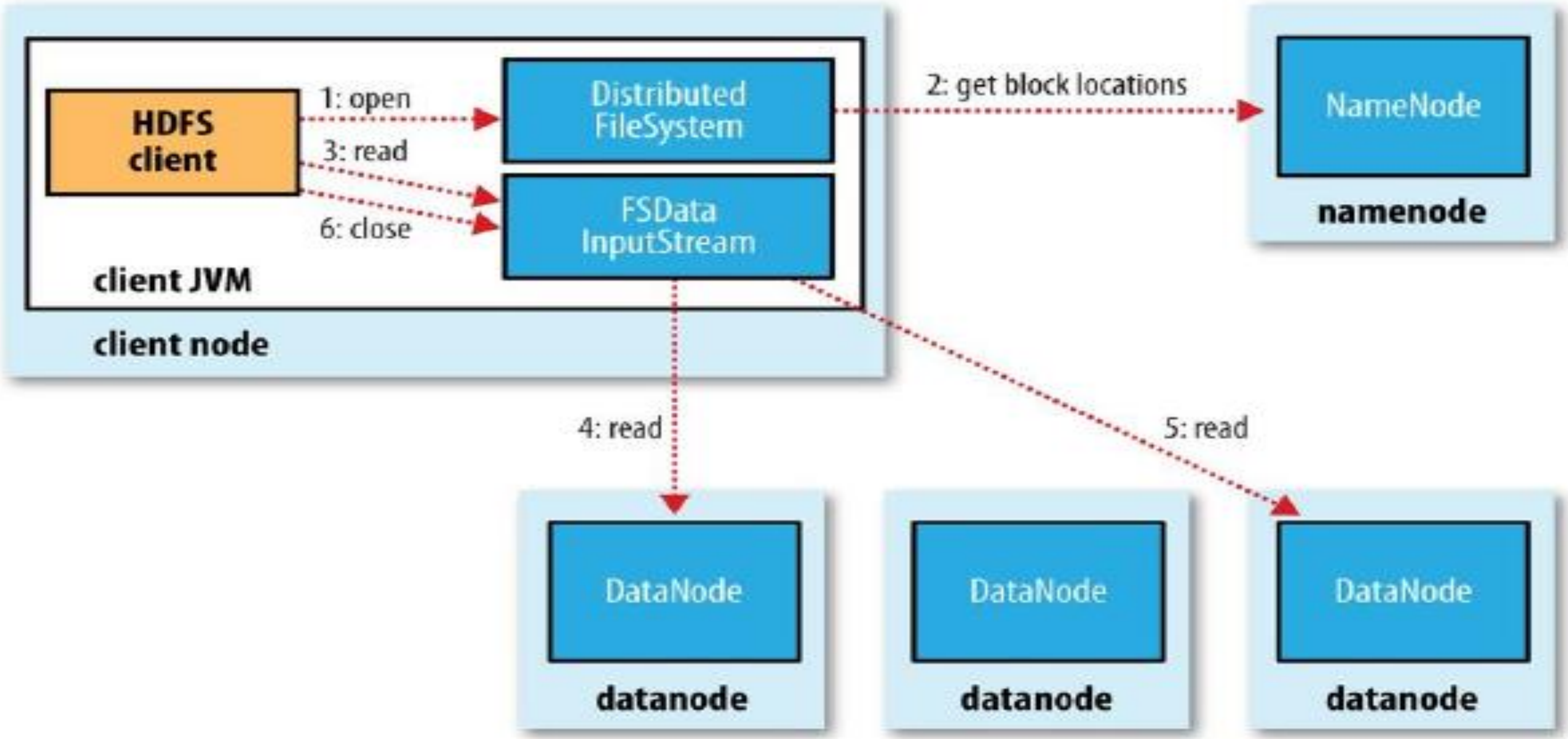


Secondary Name node

- Secondary Name node - this is NOT a backup name node, but is a separate service that keeps a copy of both the edit logs, and file system image, merging them periodically to keep the size reasonable.

- It is better to run Name node and Secondary name node in different machines. (Memory requirement).

Anatomy of File Read



Anatomy of File Read

1. The client opens the file it wishes to read by calling open() on the Filesystem object.
2. Distributed Filesystem calls the name node, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file. For each block, the name node returns the addresses of the data nodes that have a copy of that block.
3. The Distributed Filesystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from. The client then calls read() on the stream.
4. DFSInputStream, which has stored the data node addresses for the first few blocks in the file, then connects to the first (closest) data node for the first block in the file. Data is streamed from the data node back to the client, which calls read() repeatedly on the stream

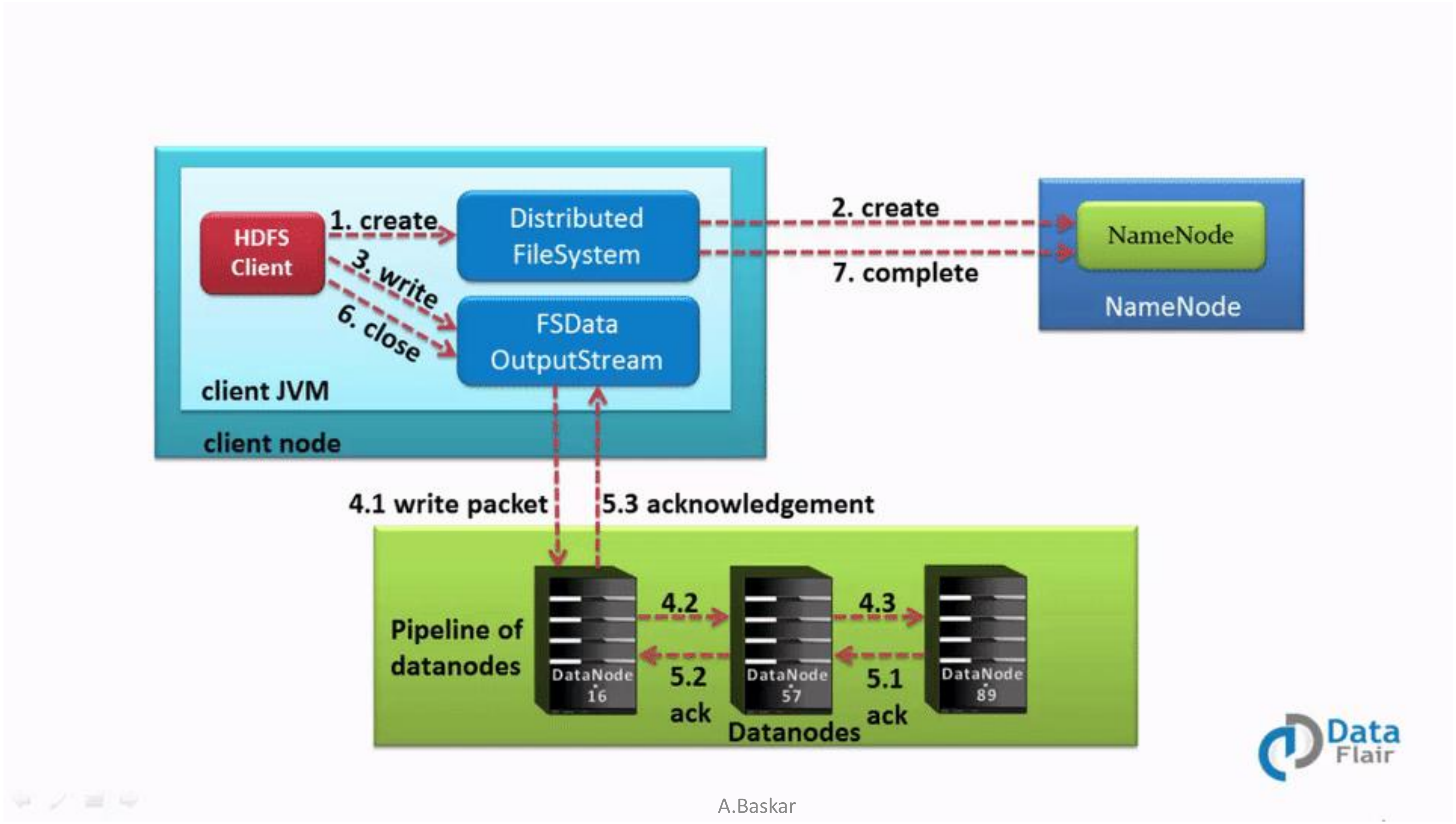
Anatomy of File Read

5. When the end of the block is reached, DFSInputStream will close the connection to the data node, then find the best data node for the next block

This happens transparently to the client, which from its point of view is just reading a continuous stream. Blocks are read in order, with the DFSInputStream opening new connections to data nodes as the client reads through the stream. It will also call the name node to retrieve the data node locations for the next batch of blocks as needed.

6. When the client has finished reading, it calls close() on the FSDataInputStream

Anatomy of File Write



Anatomy of File Write

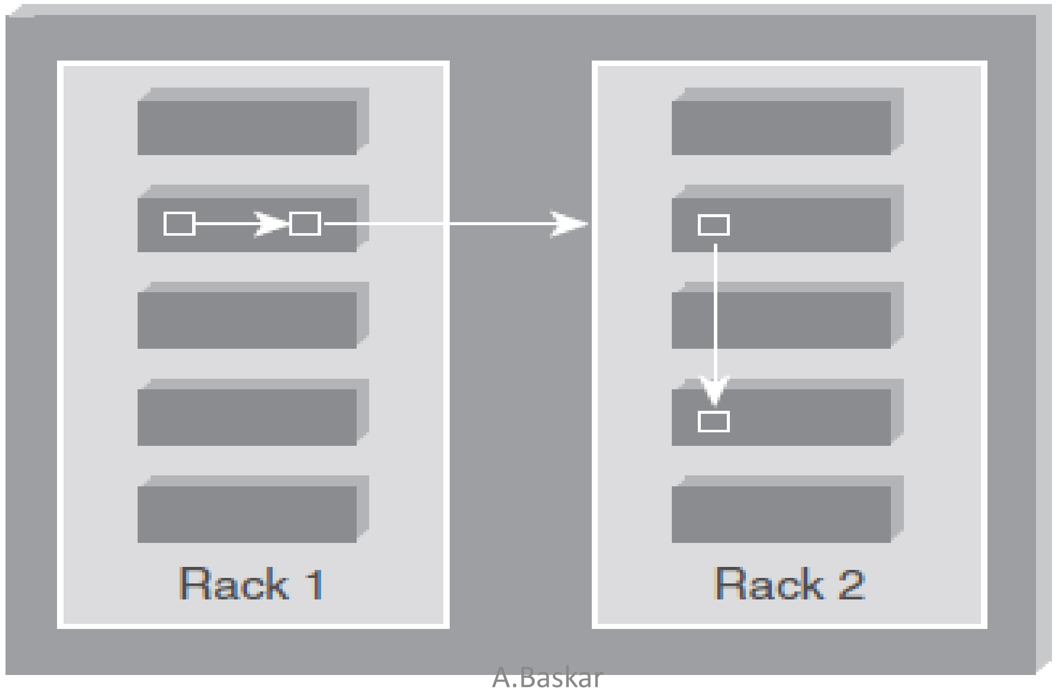
1. The HDFS client sends a **create** request on *DistributedFileSystem* APIs.
2. *DistributedFileSystem* makes an RPC call to the name node to create a new file in the file system's namespace.
3. The namenode performs various checks to make sure that the file doesn't already exist and that the client has the permissions to create the file. When these checks pass, then only the name node makes a record of the new file; otherwise, file creation fails and the client is thrown an *IOException*
4. The *DistributedFileSystem* returns a *FSDataOutputStream* for the client to start writing data to. As the client writes data, *DFSOutputStream* splits it into packets, which it writes to an internal queue, called the data queue. The data queue is consumed by the *DataStreamer, which* is responsible for asking the name node to allocate new blocks by picking a list of suitable data nodes to store the replicas.

Anatomy of File Write

- The list of data nodes form a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The *DataStreamer* streams the packets to the first data node in the pipeline, which stores the packet and forwards it to the second data node in the pipeline. Similarly, the second data node stores the packet and forwards it to the third (and last) data node in the pipeline.
- *DFSOutputStream* also maintains an internal queue of packets that are waiting to be acknowledged by data nodes, called the *ack queue*. A packet is removed from the ack queue only when it has been acknowledged by the data nodes in the pipeline. Data node sends the acknowledgment once required replicas are created (3 by default). Similarly, all the blocks are stored and replicated on the different data nodes, the data blocks are copied in parallel.
- **vi)** When the client has finished writing data, it calls **close()** on the stream.
- **vii)** This action flushes all the remaining packets to the data node pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete. The namenode already knows which blocks the file is made up of, so it only has to wait for blocks to be minimally replicated before returning successfully.

Replica Placement Strategy

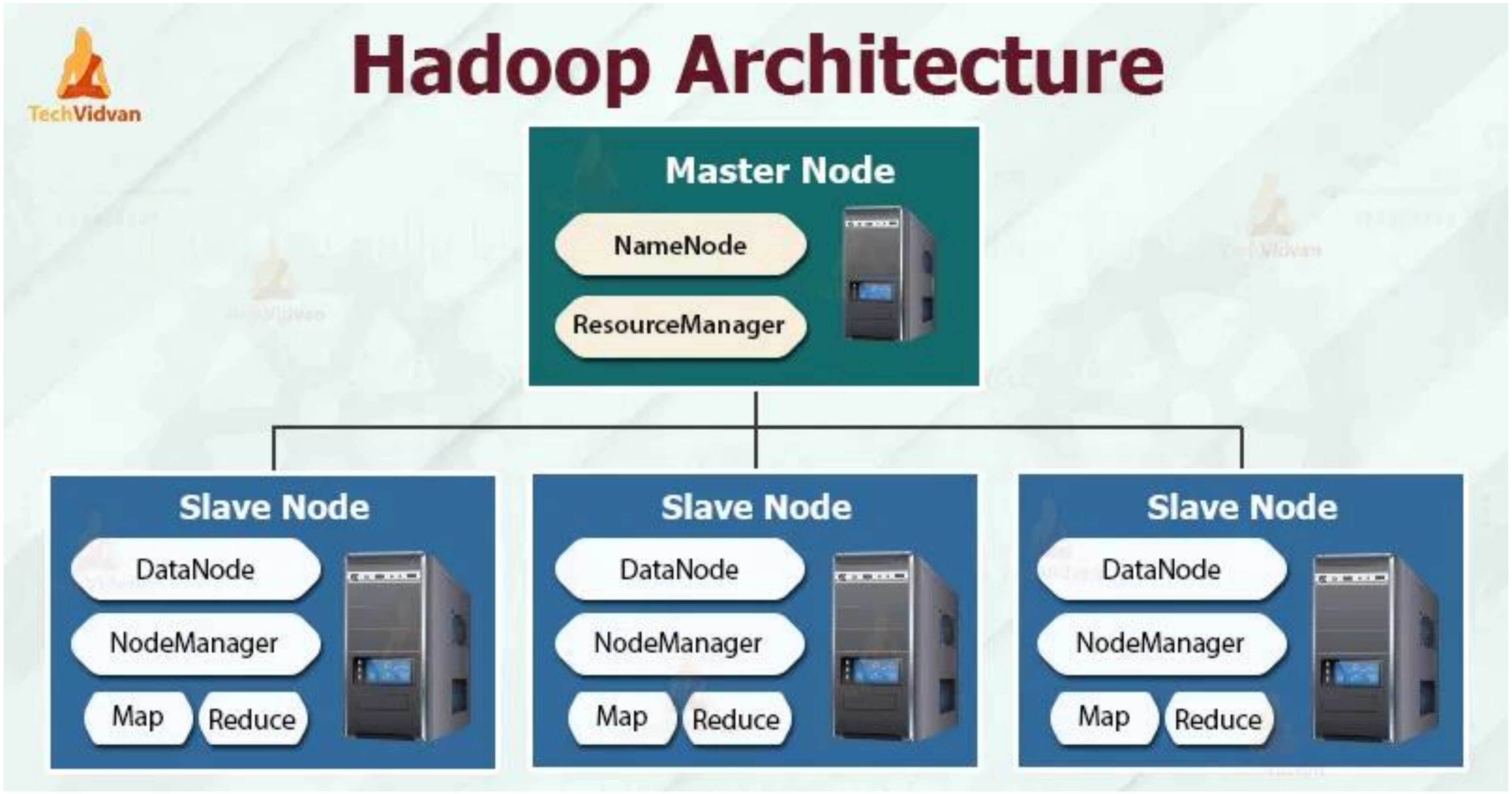
As per the Hadoop Replica Placement Strategy, first replica is placed on the same node as the client. Then it places second replica on a node that is present on different rack. It places the third replica on the same rack as second, but on a different node in the rack. Once replica locations have been set, a pipeline is built. This strategy provides good reliability.



Special Features of HDFS

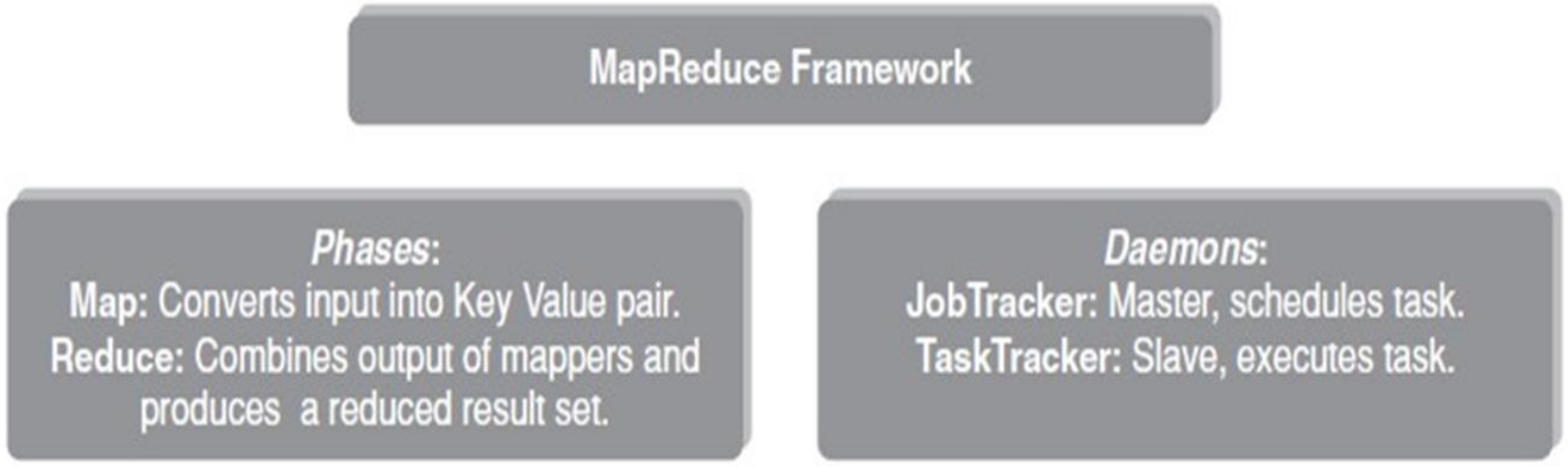
Data Replication: There is absolutely no need for a client application to track all blocks. It directs the client to the nearest replica to ensure high performance.

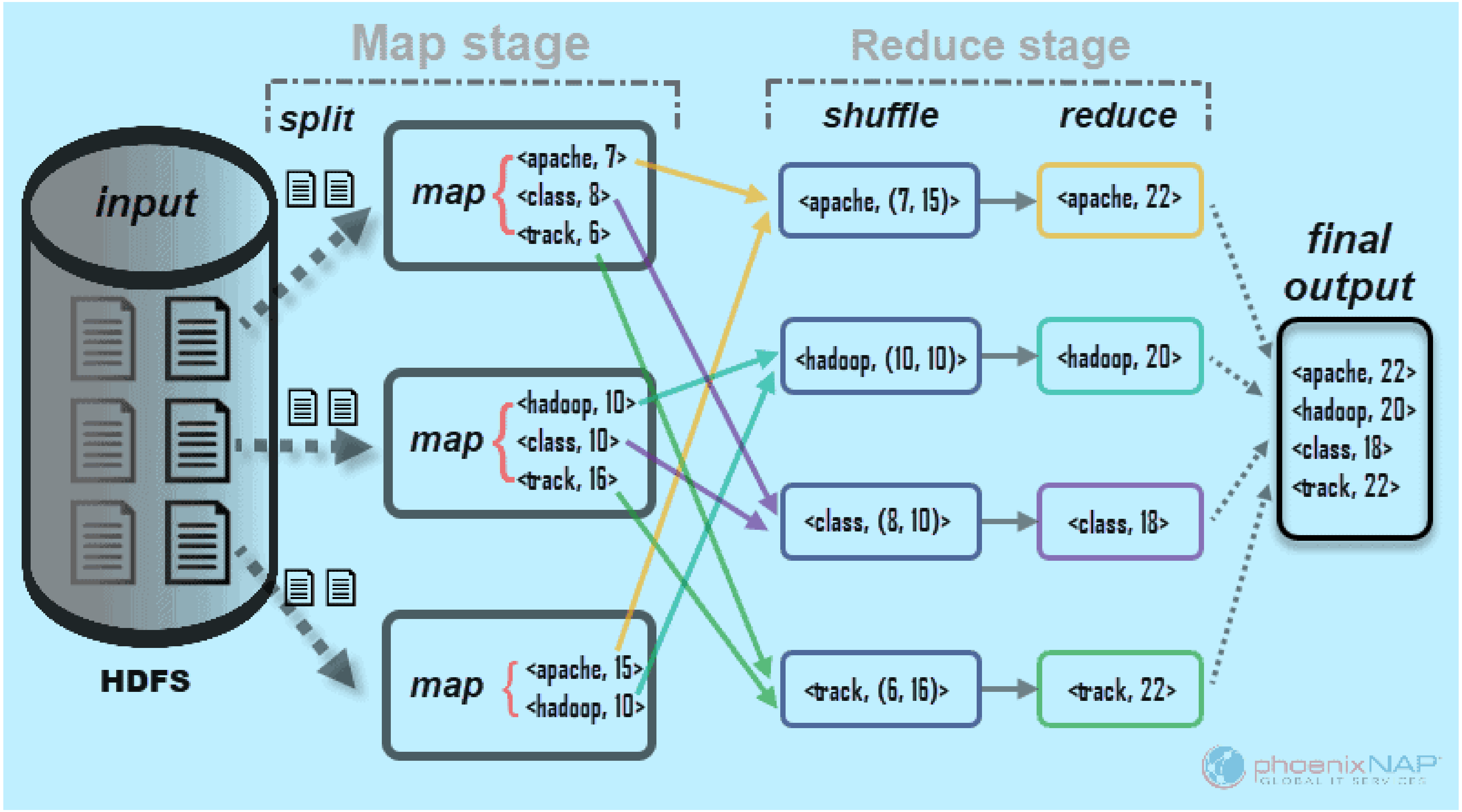
Data Pipeline: A client application writes a block to the first DataNode in the pipeline. Then this DataNode takes over and forwards the data to the next node in the pipeline. This process continues for all the data blocks, and subsequently all the replicas are written to the disk.


TechVidvan
Hadoop Architecture
Master Node
NameNode
ResourceManager
Slave Node
DataNode
NodeManager
Map
Reduce
Slave Node
DataNode
NodeManager
Map
Reduce
Slave Node
DataNode
NodeManager
Map
Reduce

Data Processing with Hadoop

- MapReduce Programming is a software framework.
- MapReduce Programming helps you to process massive amounts of data in parallel.


MapReduce Framework
Phases:
Map: Converts input into Key Value pair.
Reduce: Combines output of mappers and produces a reduced result set.
Daemons:
JobTracker: Master, schedules task.
TaskTracker: Slave, executes task.


Map stage
Reduce stage
split
shuffle
reduce
input
HDFS
map
<apache, 7>
<class, 8>
<track, 6>
map
<hadoop, 10>
<class, 10>
<track, 16>
map
<apache, 15>
<hadoop, 10>
<apache, (7, 15)>
<hadoop, (10, 10)>
<class, (8, 10)>
<track, (6, 16)>
<apache, 22>
<hadoop, 20>
<class, 18>
<track, 22>
final output
<apache, 22>
<hadoop, 20>
<class, 18>
<track, 22>
phoenixNAP GLOBAL IT SERVICES

MapReduce

- In MR programming the input data is split into smaller chunks (64MB)
- **Map Tasks** process these chunks in parallelly.
- The out put produced by this task is <Key, Value> which will be intermediate output and its stored in local disk of the server.
- The output of the mapper is automatically shuffled and sorted based on the key.
- The final output becomes input to the Reducer Phase.

MapReduce

- **Reduce task** provides the output by combining various mapper output.

- Job Input and Output stored in file systems.

- MR tasks also take care of other functionalities like ,
  - Monitoring
  - Scheduling
  - Re-executing failed tasks

HDFS and MR framework run on same set of nodes. This configurations allows effective scheduling of task for the data present. This is called **Data Locality.** This results in very high throughput.

MapReduce Daemons

- Job Tracker
  - A single Job tracker per cluster.
  - Job tracker is responsible for scheduling the jobs to task tracker and monitor the tasks and re-executing the failed task.
  - It creates a connection between Hadoop and your application. When you submit the code the job tracker creates a execution plan.
  - Job tracker is responsible for over all MR task.

MapReduce Daemons

- **Task Tracker**
  - Task tracker is responsible for executing the task assigned by Job tracker.
  - There is a single task tracker per slave and spawns multiple JVM to execute multiple map and reduce.
  - Task tracker continuously send heartbeat to job tracker.
  - If Job tracker failed to receive heartbeat then it assumes TT is failed then re assign the job to another TT which holds the same data.