8.3: Apache Hadoop

- **Instructor**: Dr. GP Saggese - gsaggese@umd.edu

- **References**
  - Ghemawat et al.: *The Google File System*, 2003
  - Dean et al.: *MapReduce: Simplified Data Processing on Large Clusters*, 2004

- ***Hadoop Distributed File System***
- Hadoop MapReduce

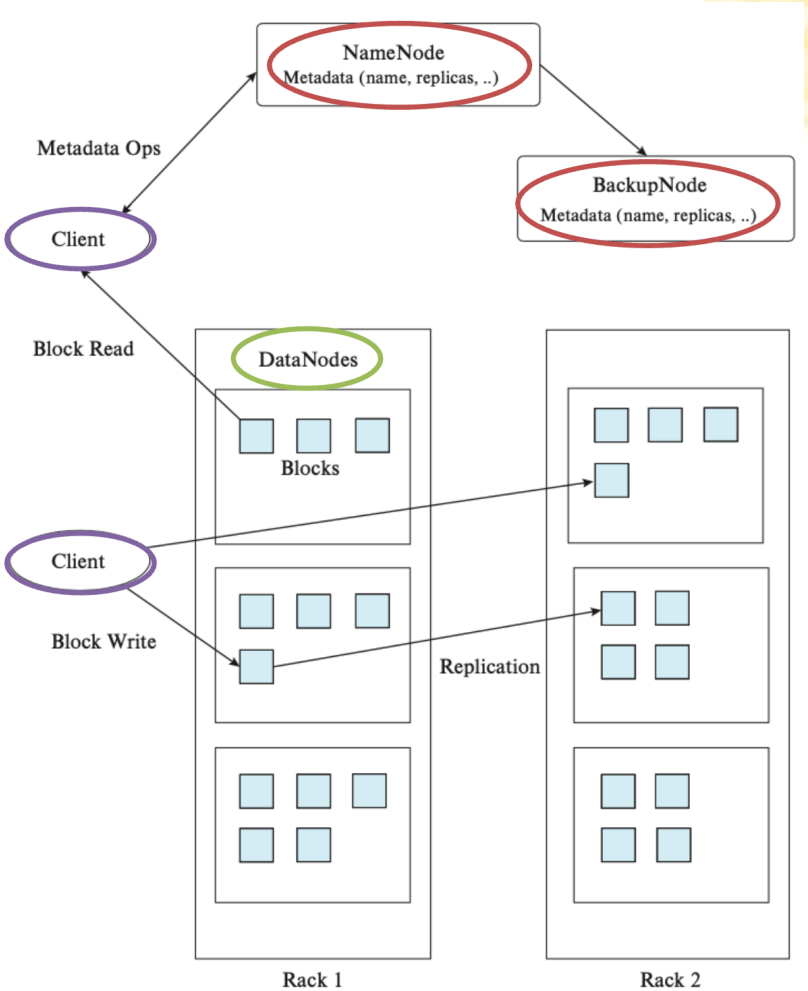
Hadoop Distributed File System (HDFS)

- **HDFS** is a **distributed file system**
  - Designed to **store large data sets reliably**
  - Part of the Apache Hadoop ecosystem
  - Inspired by the Google File System (GFS)

1. Optimized for **high-throughput access** to large files
   - Suitable for batch processing
   - Not low-latency access
2. Designed for **fault tolerance and scalability**
   - Ensures fault tolerance through replication
     - Default replication factor is 3
   - Blocks are stored on different nodes and racks
   - Follows a primary-secondary architecture
   - Provides data availability even if some nodes fail
   - Replication strategy improves read performance

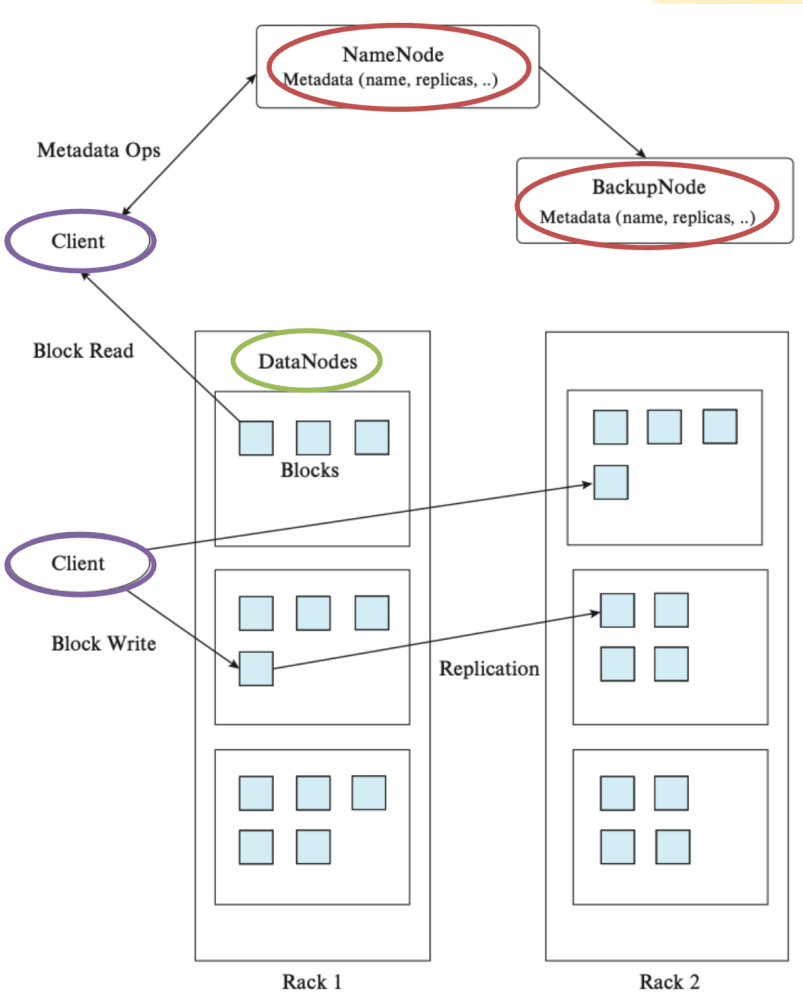
HDFS Architecture

- **NameNode**
  - Store file/dir hierarchy
  - Store file metadata (block location, size, permissions)
- **DataNodes**
  - Store actual data blocks
  - Split file into 16-64MB blocks
  - Replicate chunks (2x or 3x) across multiple *DataNodes*
  - Keep replicas in different racks



Hadoop Distributed File System

- **Library for file access**
  - Read:
    - Contact *NameNode* for *DataNode* and block pointer
    - Choose the nearest *DataNode* for each block
    - Connect to *DataNode* for data access
    - Reads blocks in parallel to improve performance
    - Data is reassembled by the client in correct order
  - Write:
    - *NameNode* creates blocks
    - Assign blocks to multiple *DataNodes*
    - Client sends data to *DataNodes*
    - *DataNodes* store



Fault Tolerance and Recovery

- *NameNode* monitors *DataNode* heartbeat signals
- On failure, blocks are re-replicated to maintain replication factor
- *NameNode* itself is a single point of failure
  - Solved with HDFS High Availability
- Data integrity ensured using checksums

HDFS vs Traditional File Systems

- Best for storing and processing large-scale logs, media, sensor data
- Commonly used in data lakes and ETL pipelines
- Optimized for write-once, read-many access pattern
- Supports very large files and directories
  - Performance degrades with many small files
- Lacks low-latency access, but provides high throughput
  - Not suitable for transactional or low-latency systems

- Hadoop Distributed File System
- ***Hadoop MapReduce***

Implementations of MapReduce

- **Google**
  - Not available outside Google
- **Hadoop**
  - Open-source in Java
  - Uses HDFS for storage
  - Hadoop Wiki: Intro, Getting Started, Map/Reduce Overview
- **Amazon Elastic MapReduce (EMR)**
  - Hadoop MapReduce on Amazon EC2
  - Also runs Spark, HBase, Hive,
- **Spark**
- **Dask**

MapReduce: Hadoop

- Hadoop: open-source MapReduce implementation
- Functionalities
  - Partition input data (HDFS)
  - Input adapters
    - E.g., hBase, MongoDB, Cassandra, Amazon Dynamo
  - Schedule program execution across machines
  - Handle machine failures
  - Manage inter-machine communication
  - Perform *GroupByKey* step
  - Output adapters
    - E.g., Avro, ORC, Parquet
  - Schedule multiple *MapReduce* jobs

Data Flow

- Store input, intermediate, final outputs in HDFS
  - Operations in Hadoop move disk to disk
- Use adapters to read/partition data in chunks
- Scheduler places map tasks near physical storage of input data
  - Store intermediate results on local FS of Map and Reduce workers
- Output often serves as input for another MapReduce task

Hadoop Distributed File System (HDFS): Overview

- Designed for distributed storage of large datasets
- Built on master-slave architecture
  - NameNode: manages metadata and directory structure
  - DataNodes: store actual data blocks
- Optimized for high throughput rather than low latency
- Stores large files across multiple machines
- Writes are append-only, simplifying consistency
- Fault-tolerant using data replication
  - Default: each block is replicated 3 times
- Ideal for batch processing and big data workloads

HDFS: Data Storage and Access

- Files are split into fixed-size blocks (default: 128MB)
- Blocks distributed across DataNodes for parallelism
- NameNode maintains block-to-DataNode mapping
- Client reads data directly from DataNodes
- Ensures reliability through block replication
- If a DataNode fails, replicas serve the data
- Data locality: computation is moved to where data resides

Hadoop MapReduce: Overview

- Programming model for distributed data processing
- Processes data in parallel across a cluster
- Consists of two main functions:
  - Map: filters and sorts input data
  - Reduce: aggregates intermediate outputs
- Suited for batch jobs over large datasets
- Fault-tolerant: tasks are retried upon failure

MapReduce: Execution Phases

- Input data split into chunks processed by Mappers
- Mapper outputs key-value pairs
- Shuffle and Sort: organizes data by key
  - Intermediate keys are grouped and sent to Reducers
- Reducers aggregate values by key
- Final output written back to HDFS

Example: Word Count with MapReduce

- Input: text files containing words
- Mapper:
  - Reads lines and emits (word, 1) for each word
- Shuffle and Sort:
  - Groups by word, e.g., (word, [1,1,1])
- Reducer:
  - Sums values: emits (word, total_count)
- Output: word frequencies stored in HDFS

HDFS vs MapReduce: Complementary Roles

- HDFS: distributed storage system
  - Stores large datasets efficiently
- MapReduce: distributed compute model
  - Processes data stored in HDFS
- Together enable scalable and fault-tolerant data analysis

Benefits and Limitations

- Benefits:
  - Scalable and fault-tolerant
  - Handles petabytes of data
  - Open-source and cost-effective
- Limitations:
  - High latency, not suitable for real-time
  - Difficult for complex iterative algorithms
  - Superseded in many cases by Spark and other engines