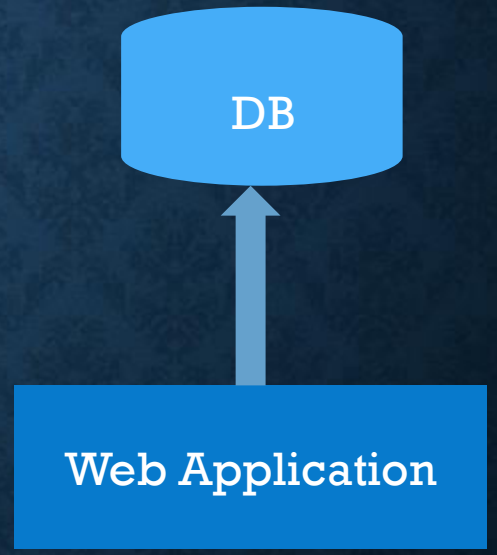
CONCEPTS OF DISTRIBUTED COMPUTING

WHAT IS A DISTRIBUTED SYSTEM?

- A group of computers working together as to appear as a single computer to the end-user.
- Properties:
  - Can be locally or geographically distributed
  - Component machines have a shared state
  - The sysem operates concurrently
  - Component machines can fail independently without resulting in overall downtime

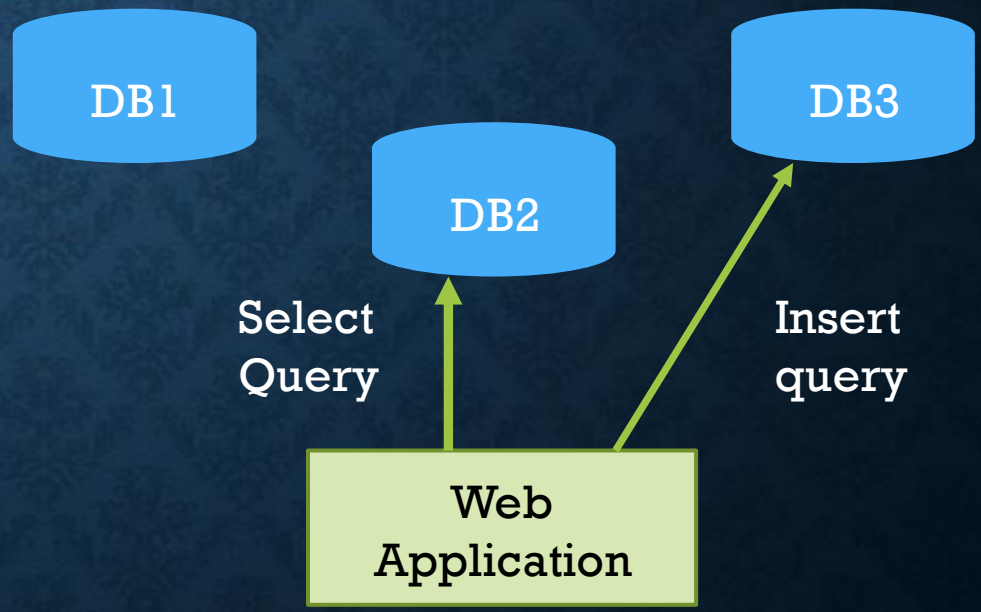
CONSIDER A TRADITIONAL DATABASE

- In a traditional system, the database is on a single machine, stored on one file system.
- If we wish to interact with the database, we use an application that "talks" to the machine storing the database



DISTRIBUTED DATABASE

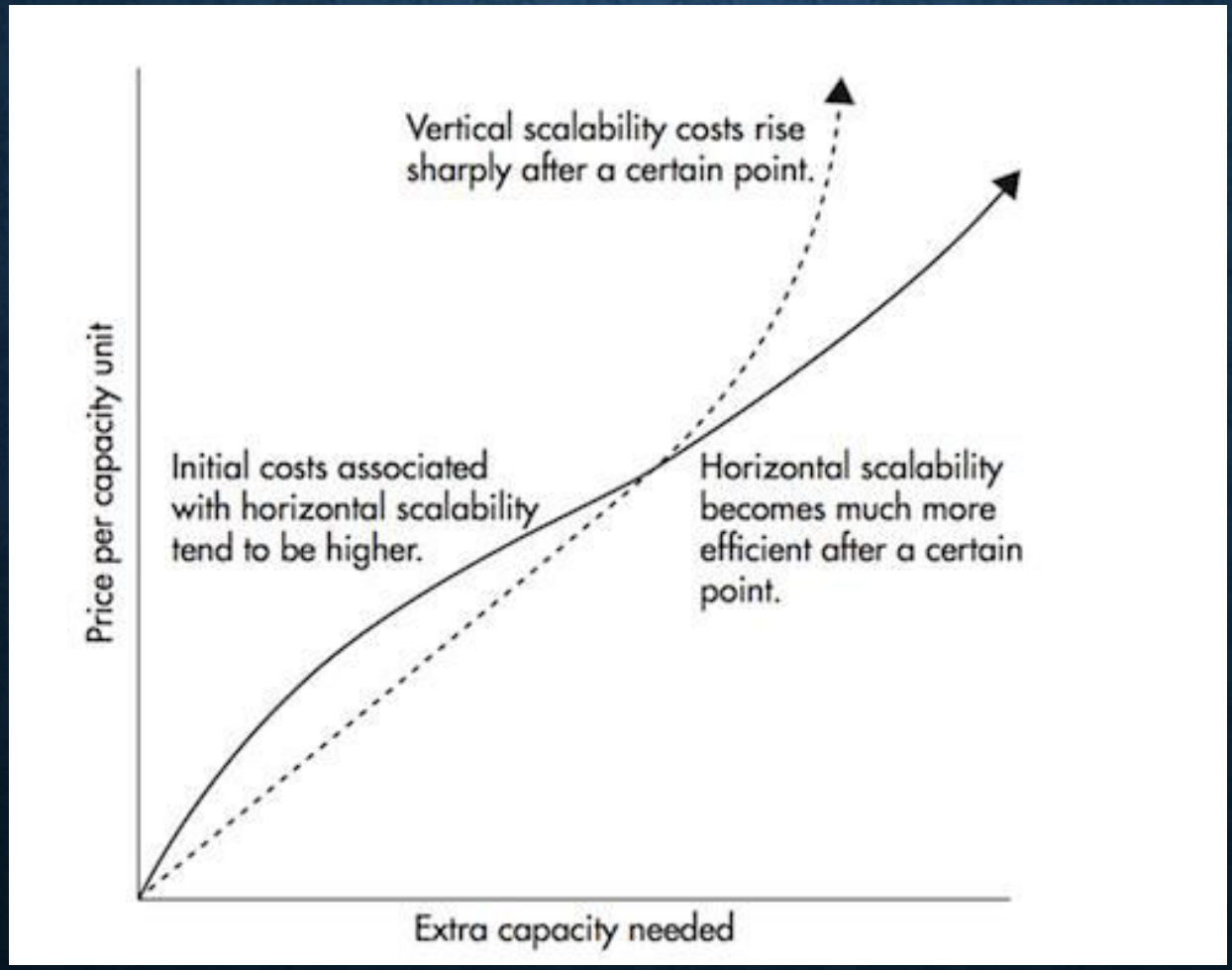
- Now, the database is being stored on different machines, not necessarily in the same place.
- The user, through an application, should be able to "talk" to whichever machine contains the requisite information
- However, the end-user must not be aware of the underlying distribution.



WHY USE DISTRIBUTED SYSTEMS?

- Managing a distributed system is a complicated tasks – we need to stay clear of several potential pitfalls. So, why do it at all?
- Sometimes there is no choice!
  - Horizontal Scaling is required
  - Reducing system latency
  - Increasing fault tolerance

VERTICAL VS HORIZONTAL SCALING



VERTICAL SCALING

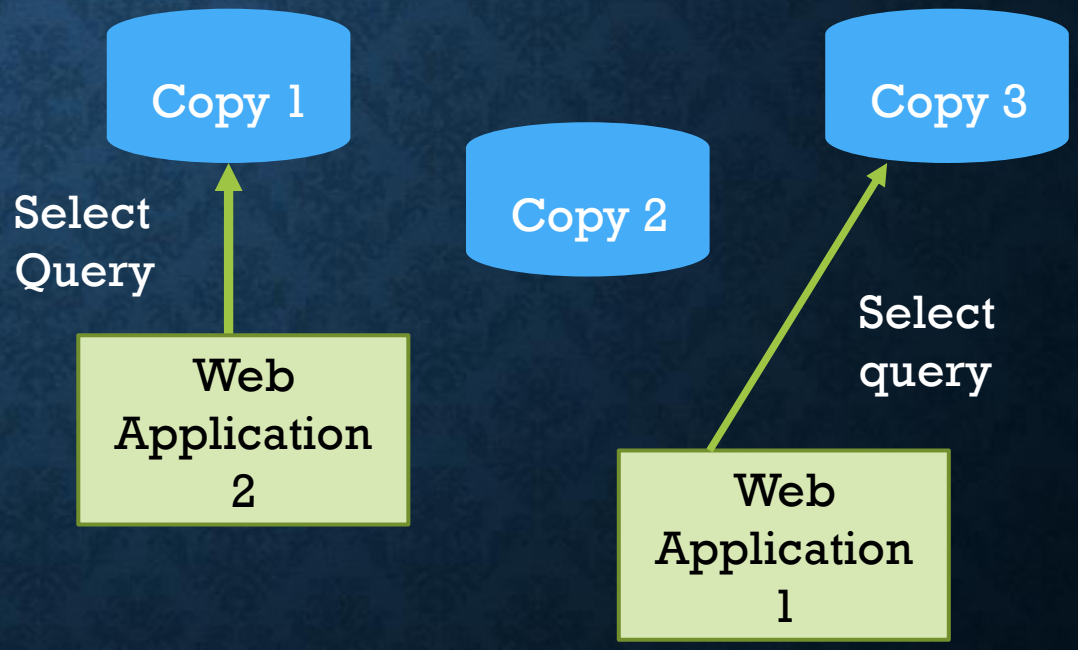
- Increase performance on the same system by upgrading the capabilities of the system.
- Advantages:
  - Not much of a change to software design
  - Low investment:benefit ratio in the short time
- Disadvantages:
  - Limited by hardware capabilities
  - Lower gains in the long run

HORIZONTAL SCALING

- Adding more computers instead of increasing the capabilities of the existing system
- Advantages:
  - Significantly cheaper over the long run
  - No cap over the scaling
  - Easy fault tolerance – byzantine
  - Low latency by geographical distribution
- Disadvantages:
  - Requires software redesign
  - High initial investment

PRIMARY REPLICATION

- Easiest approach
- Just make copies of the database and move them to different servers.
- Implement load balancing to direct traffic



PRIMARY REPLICATION

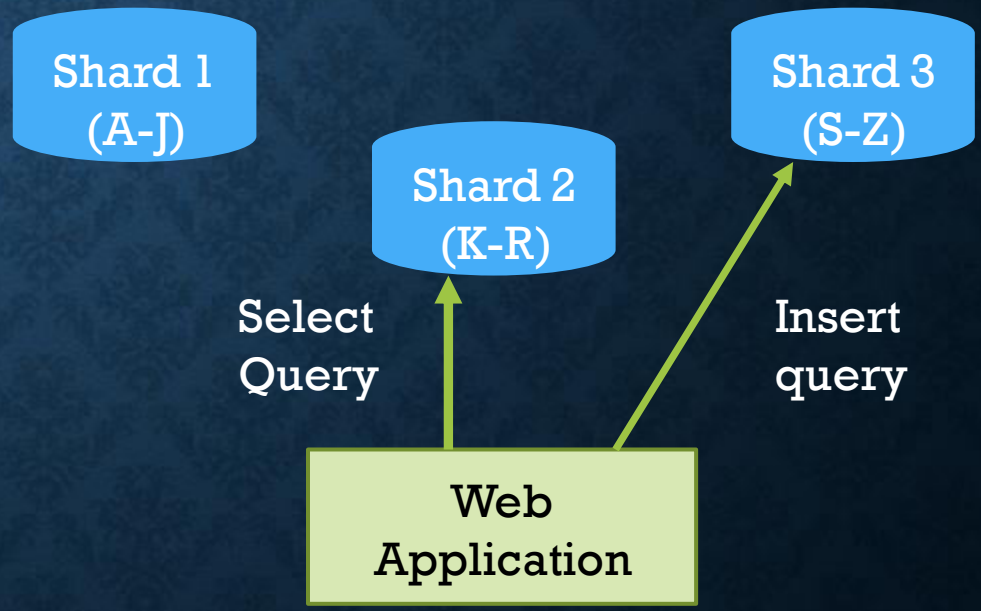
- One way to implement horizontal scaling is to store copies of data in different servers
- Done to give us better read
- Gives us better performance – low latency, fault tolerant
- Capable of handling more requests
- Issue:
  - Loses Consistency upon write – propagation of information is not instantaneous

MULI-PRIMARY REPLICATION

- Multiple primary nodes that can handle both reads and writes
- Needs locking/synchronization mechanisms
- Would still create pre-propagation conflicts
- Can get really complicated, really quickly.

PARTITIONING

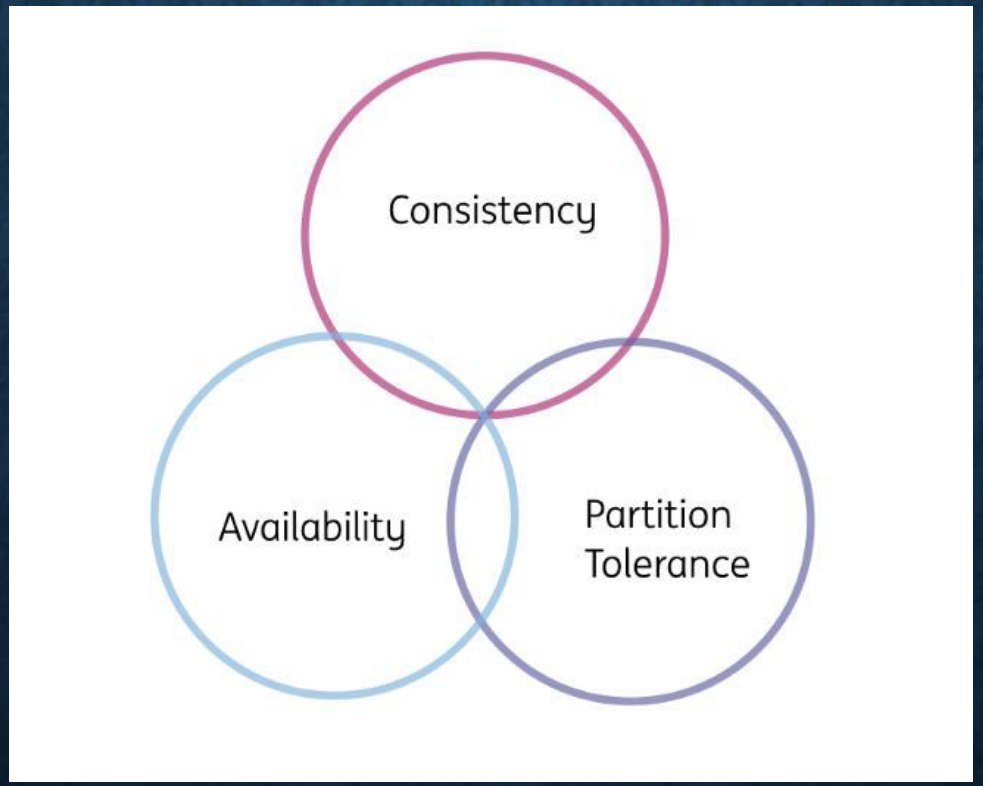
- Splits server into multiple smaller servers, called shards
- Each shard holds a different set of records – defined by a business rule
- Shards should be defined with uniformity in mind
- Issue:
  - Load balancing can be really hard to define and predict



DISTRIBUTED DATA STORES

- Most recognized form of a distributed system
- Some form of NOSQL non-relational database
- Usually using key-value semantics
- Example: Apple is known to use 75,000 Apache Cassandra nodes storing over 10 petabytes of data, back in 2015

CAP THEOREM



CAP THEOREM

- Consistency — What you read and write sequentially is what is expected
- Availability — the whole system does not die — every non-failing node always returns a response.
- Partition Tolerant — The system continues to function and uphold its consistency/availability guarantees in spite of network partitions
- Proven in 2002

CAP IN PRACTICE

- In reality, partition tolerance must be a given for any distributed data store
- We have to make a choice between strong consistency and high availability under a network partition.
- Practice shows that most applications value availability more. You do not necessarily always need strong consistency.
- Such databases settle with the weakest consistency model — eventual consistency

BASE PROPERTIES

Provide BASE properties (as opposed to traditional databases' ACID)

- Basically Available — The system always returns a response
- Soft state — The system could change over time, even during times of no input (due to eventual consistency)
- Eventual consistency — In the absence of input, the data will spread to every node sooner or later — thus becoming consistent

DISTRIBUTED COMPUTING

- Distributed computing is the key to the influx of Big Data processing we've seen in recent years.
- It is the technique of splitting an enormous task (e.g aggregate 100 billion records), of which no single computer is capable of practically executing on its own, into many smaller tasks, each of which can fit into a single commodity machine.
- You split your huge task into many smaller ones, have them execute on many machines in parallel, aggregate the data appropriately and you have solved your initial problem.
- This approach again enables you to scale horizontally — when you have a bigger task, simply include more nodes in the calculation.

MAP REDUCE

- MapReduce can be simply defined as two steps — mapping the data and reducing it to something meaningful
- Each Map job is a separate node transforming as much data as it can. Each job traverses all of the data in the given storage node and maps it to a simple tuple of the date and the number one.
- Then, three intermediary steps *(which nobody talks about)* are done — Shuffle, Sort and Partition.
- They basically further arrange the data and delete it to the appropriate reduce job. As we're dealing with big data, we have each Reduce job separated to work on a single date only..

BETTER TECHNIQUES

- Map-reduce is simple, and brings its own problems with it
- Because it works in batches (jobs) a problem arises where if your job fails — you need to restart the whole thing
- Not real-time
- More recent systems use lambda and kappa architectures
- Examples include Spark, Kafka, Samza, etc.

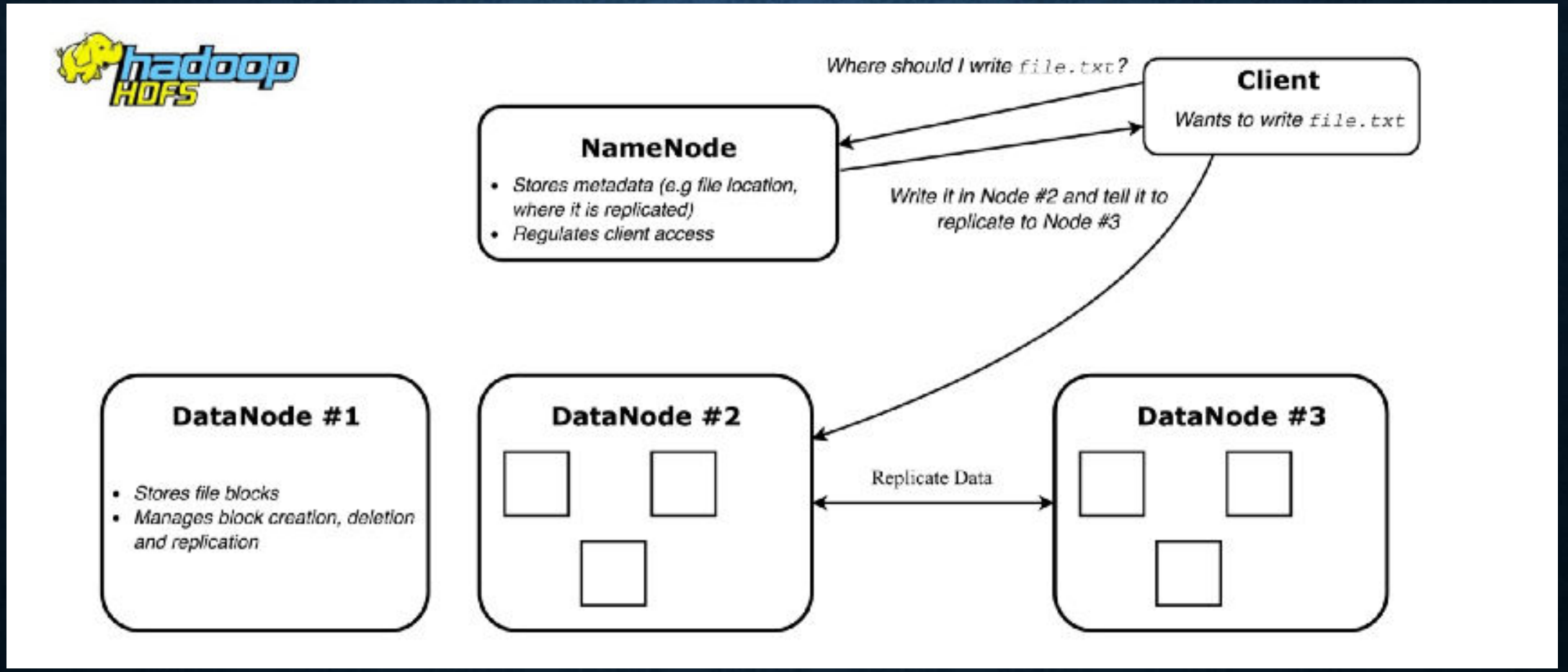
DISTRIBUTED FILE SYSTEMS

- Distributed file systems can be thought of as distributed data stores.
- They're the same thing as a concept
- However, the difference is that distributed file systems allow files to be accessed using the same interfaces and semantics as local files, not through a custom API
- Cassandra needs the CQL, but Hadoop is just a file system.

HDFS

- Hadoop Distributed File System (HDFS) is the distributed file system used for distributed computing via the Hadoop framework.
- Boasting widespread adoption, it is used to store and replicate large files (GB or TB in size) across many machines.
- Its architecture consists mainly of ***NameNodes*** and ***DataNodes***.
- NameNodes are responsible for keeping metadata about the cluster, like which node contains which file blocks.
- They act as coordinators for the network by figuring out where best to store and replicate files, tracking the system's health. DataNodes simply store files and execute commands like replicating a file, writing a new one and others.

HDFS


hadoop
HDFS
Where should I write file.txt?
Client
Wants to write file.txt
NameNode
Stores metadata (e.g file location, where it is replicated)
Regulates client access
Write it in Node #2 and tell it to replicate to Node #3
DataNode #1
Stores file blocks
Manages block creation, deletion and replication
DataNode #2
Replicate Data
DataNode #3

IPFS

- Interplanetary File System (IPFS) is an exciting new peer-to-peer protocol/network for a distributed file system.
- Leveraging Blockchain technology, it boasts a completely decentralized architecture with no single owner nor point of failure.

- IPFS offers a naming system (similar to DNS) called IPNS and lets users easily access information. It stores file via historic versioning, similar to how Git does. This allows for accessing all of a file's previous states.

DISTRIBUTED MESSAGING

- Messaging systems provide a central place for storage and propagation of messages/events inside your overall system.
- They allow you to decouple your application logic from directly talking with your other systems.
- Simply put, a messaging platform works in the following way:
- A message is broadcast from the application which potentially create it (called a **producer**), goes into the platform and is read by potentially multiple applications which are interested in it (called **consumers**).
- Examples Kafka, RabbitMO, ApacheActiveMO