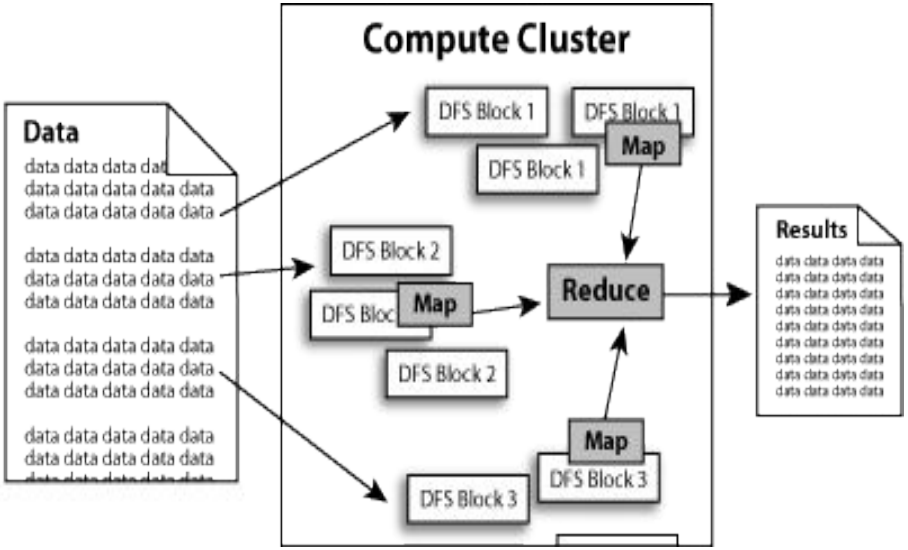
Hadoop: A Software Framework for Data Intensive Computing Applications

What is Hadoop?

- Software platform that lets one easily write and run applications that process vast amounts of data. It includes:
  - MapReduce – offline computing engine
  - HDFS – Hadoop distributed file system
  - HBase (pre-alpha) – online data access
- Yahoo! is the biggest contributor
- Here's what makes it especially useful:
  - **Scalable:** It can reliably store and process petabytes.
  - **Economical:** It distributes the data and processing across clusters of commonly available computers (in thousands).
  - **Efficient:** By distributing the data, it can process it in parallel **on the nodes where the data is located.**
  - **Reliable:** It automatically maintains multiple copies of data and automatically redeploys computing tasks based on failures.

What does it do?

- Hadoop implements Google's MapReduce, using HDFS
- MapReduce divides applications into many small blocks of work.
- HDFS creates multiple replicas of data blocks for reliability, placing them on compute nodes around the cluster.
- MapReduce can then process the data where it is located.
- Hadoop 's target is to run on clusters of the order of 10,000-nodes.



Hadoop: Assumptions

It is written with large clusters of computers in mind and is built around the following assumptions:

- Hardware *will* fail.
- Processing will be run in batches. Thus there is an emphasis on high throughput as opposed to low latency.
- Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size.
- It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.
- Applications need a **write-once-read-many** access model.
- Moving Computation is Cheaper than Moving Data.
- Portability is important.

Apache Hadoop Wins Terabyte Sort Benchmark (July 2008)

- One of Yahoo's Hadoop clusters sorted 1 terabyte of data in **209 seconds**, which beat the previous record of 297 seconds in the annual general purpose (daytona) terabyte sort benchmark. The sort benchmark specifies the input data (10 billion 100 byte records), which must be completely sorted and written to disk.
- The sort used 1800 maps and 1800 reduces and allocated enough memory to buffers to hold the intermediate data in memory.
- The cluster had 910 nodes; 2 quad core Xeons @ 2.0ghz per node; 4 SATA disks per node; 8G RAM per a node; 1 gigabit ethernet on each node; 40 nodes per a rack; 8 gigabit ethernet uplinks from each rack to the core; Red Hat Enterprise Linux Server Release 5.1 (kernel 2.6.18); Sun Java JDK 1.6.0_05-b13

Example Applications and Organizations using Hadoop

- A9.com – Amazon: To build Amazon's product search indices; process millions of sessions daily for analytics, using both the Java and streaming APIs; clusters vary from 1 to 100 nodes.
- Yahoo! : More than 100,000 CPUs in ~20,000 computers running Hadoop; biggest cluster: 2000 nodes (2*4cpu boxes with 4TB disk each); used to support research for Ad Systems and Web Search
- AOL : Used for a variety of things ranging from statistics generation to running advanced algorithms for doing behavioral analysis and targeting; cluster size is 50 machines, Intel Xeon, dual processors, dual core, each with 16GB Ram and 800 GB hard-disk giving us a total of 37 TB HDFS capacity.
- Facebook: To store copies of internal log and dimension data sources and use it as a source for reporting/analytics and machine learning; 320 machine cluster with 2,560 cores and about 1.3 PB raw storage;
- FOX Interactive Media : 3 X 20 machine cluster (8 cores/machine, 2TB/machine storage) ; 10 machine cluster (8 cores/machine, 1TB/machine storage); Used for log analysis, data mining and machine learning
- University of Nebraska Lincoln: one medium-sized Hadoop cluster (200TB) to store and serve physics data;

More Hadoop Applications

- Adknowledge - to build the recommender system for behavioral targeting, plus other clickstream analytics; clusters vary from 50 to 200 nodes, mostly on EC2.
- Contextweb - to store ad serving log and use it as a source for Ad optimizations/ Analytics/reporting/machine learning; 23 machine cluster with 184 cores and about 35TB raw storage. Each (commodity) node has 8 cores, 8GB RAM and 1.7 TB of storage.
- Cornell University Web Lab: Generating web graphs on 100 nodes (dual 2.4GHz Xeon Processor, 2 GB RAM, 72GB Hard Drive)
- NetSeer - Up to 1000 instances on Amazon EC2 ; Data storage in Amazon S3; Used for crawling, processing, serving and log analysis
- The New York Times : Large scale image conversions ; EC2 to run hadoop on a large virtual cluster
- Powerset / Microsoft - Natural Language Search; up to 400 instances on Amazon EC2 ; data storage in Amazon S3

MapReduce Paradigm

- Programming model developed at Google
- Sort/merge based distributed computing
- Initially, it was intended for their internal search/indexing application, but now used extensively by more organizations (e.g., Yahoo, Amazon.com, IBM, etc.)
- It is functional style programming (e.g., LISP) that is naturally parallelizable across a large cluster of workstations or PCS.
- **The underlying system takes care of the partitioning of the input data, scheduling the program's execution across several machines, handling machine failures, and managing required inter-machine communication. (This is the key for Hadoop's success)**

How does MapReduce work?

- The run time partitions the input and provides it to different Map instances;
- Map (key, value) → (key', value')
- The run time collects the (key', value') pairs and distributes them to several Reduce functions so that each Reduce function gets the pairs with the same key'.
- Each Reduce produces a single (or zero) file output.
- Map and Reduce are user written functions

Example MapReduce: To count the occurrences of words in the given set of documents

map(String key, String value):

// key: document name; value: document contents; map (k1,v1) → list(k2,v2)

for each word w in value: EmitIntermediate(w, "1");

(Example: If input string is (“Saibaba is God. I am I”), Map produces
{<“Saibaba”,1”>, <“is”, 1>, <“God”, 1>, <“I”,1>, <“am”,1>,<“I”,1>}

reduce(String key, Iterator values):

// key: a word; values: a list of counts; reduce (k2,list(v2)) → list(v2)

int result = 0;

for each v in values:

result += ParseInt(v);

Emit(AsString(result));

(Example: reduce(“I”, <1,1>) → 2)

Example applications

- Distributed grep (as in Unix grep command)
- **Count of URL Access Frequency**
- **ReverseWeb-Link Graph:** list of all source URLs associated with a given target URL
- Inverted index: Produces <word, list(Document ID)> pairs
- Distributed sort

MapReduce-Fault tolerance

- **Worker failure:** The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle state, and therefore become eligible for scheduling* on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.
- **Master Failure:** It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, in most cases, the user restarts the job.

Mapping workers to Processors

- The input data (on HDFS) is stored on the local disks of the machines in the cluster. HDFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines.

- The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data. When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

Task Granularity

- The map phase has M pieces and the reduce phase has R pieces.
- M and R should be much larger than the number of **worker machines**.
- Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails.
- Larger the M and R, more the decisions the master must make
- R is often constrained by users because the output of each reduce task ends up in a separate output file.
- Typically, (at Google), M = 200,000 and R = 5,000, using 2,000 worker machines.

Additional support functions

- **Partitioning function:** The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. .hash(key) mod R.). In some cases, it may be useful to partition data by some other function of the key. The user of the MapReduce library can provide a special partitioning function.
- **Combiner function:** User can specify a *Combiner function that does partial merging of* the intermediate local disk data before it is sent over the network. The *Combiner function is executed on each machine* that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions.

Problem: seeks are expensive

- CPU & transfer speed, RAM & disk size – double every 18-24 months
- Seek time nearly constant (~5%/year)
- Time to read entire drive is growing – scalable computing **must go at transfer rate**
- Example: Updating a terabyte DB, given: 10MB/s transfer, 10ms/seek, 100B/entry (10Billion entries), 10kB/page (1Billion pages)
- Updating 1% of entries (100Million) takes:
  - 1000 days with random B-Tree updates
  - 100 days with batched B-Tree updates
  - 1 day with sort & merge
- To process 100TB datasets
  - on 1 node: – scanning @ 50MB/s = 23 days
  - on 1000 node cluster: – scanning @ 50MB/s = 33 min

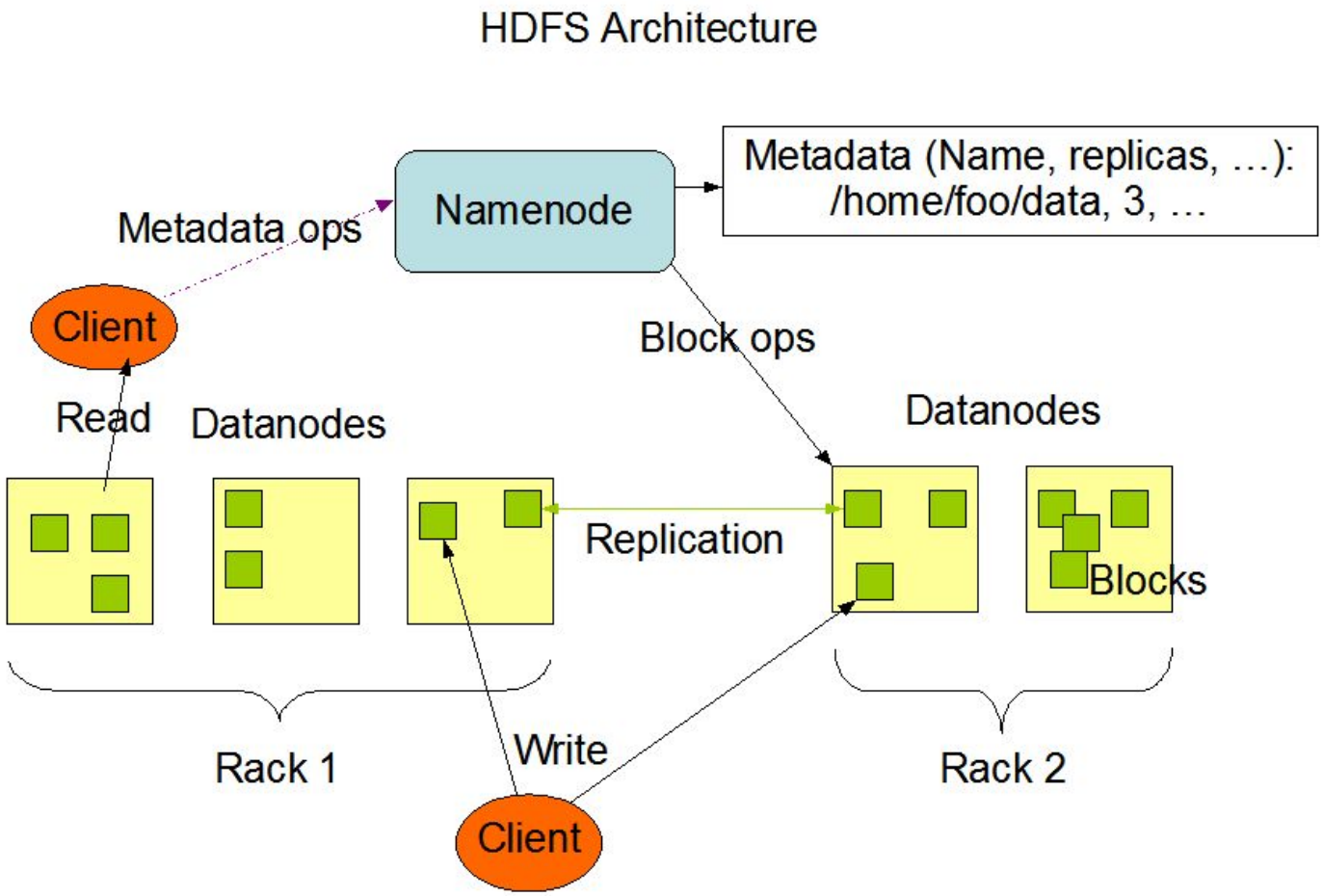
– MTBF = 1 day

• Need framework for distribution – efficient, reliable, easy to use

HDFS

- The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant.
  - highly fault-tolerant and is designed to be deployed on low-cost hardware.
  - provides high throughput access to application data and is suitable for applications that have large data sets.
  - relaxes a few POSIX requirements to enable streaming access to file system data.
  - part of the Apache Hadoop Core project. The project URL is http://hadoop.apache.org/core/.

HDFS Architecture



Example runs [1]

- Cluster configuration: ≈1800 machines; each with two 2GHz Intel Xeon processors with 4GB of memory (1-1.5 GB reserved for other tasks), two 160GB IDE disks, and a gigabit Ethernet link. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.
- Grep: *Scans through 10^{10} 100-byte records (distributed over 1000 input file by GFS),* searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces (M = 15000), and the entire output is placed in one file (R = 1). The entire computation took approximately 150 seconds from start to finish including 60 seconds to start the job.
- Sort: *Sorts 10^{10} 100-byte records (approximately* 1 terabyte of data). As before, the input data is split into 64MB pieces (M = 15000) and R = 4000. Including startup overhead, the entire computation took 891 seconds.

Execution overview

1. The MapReduce library in the user program first splits input files into M pieces of typically 16 MB to 64 MB/piece. It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the assigned input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. The locations of these buffered pairs on the local disk are passed back to the master, who forwards these locations to the reduce workers.

Execution overview (cont.)

5. When a reduce worker is notified by the master about these locations, it uses RPC remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.
6. The reduce worker iterates over the sorted intermediate data and for each **unique intermediate key** encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce function.* The output of the *Reduce function is appended* to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program---the MapReduce call in the user program returns back to the user code. The output of the mapreduce execution is available in the R output files (one per reduce task).

References

[1] J. Dean and S. Ghemawat, ``MapReduce: Simplied Data Processing on Large Clusters,'' OSDI 2004. (Google)

[2] D. Cutting and E. Baldeschwieler, ``Meet Hadoop,'' OSCON, Portland, OR, USA, 25 July 2007 (Yahoo!)

[3] R. E. Brayant, "Data Intensive Scalable computing: The case for DISC," Tech Report: CMU-CS-07-128, http://www.cs.cmu.edu/~bryant