International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

A BRIEF SURVEY OF HADOOP FILE SYSTEM

[1]SUNIL KUMAR THOTA, [2] DR. SRIKANTH LAKUMARAPU, [3]R. ASHOK KUMAR
[1,2]Assistant Professor, Department of CSE, Sphoorthy Engineering College
[3]Associate Professor, Department of CSE, Sphoorthy Engineering College

Abstract
Hadoop is a popular open source implementation based on distributed computing having HDFS file system (Hadoop Distributed File System). Hadoop is highly fault-tolerant and can be deployed on low cost hardware. Hadoop is very much suitable for high volume of data. It also provides the highspeed access to application data. The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to stream those data sets at high bandwidth to user applications. In a large cluster, thousands of servers host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size. Hadoop architecture is cluster based, which consists of nodes (data note, name node), physically separate to each other, in ideal condition. The performance of Hadoop can be increased by proper assignment of the tasks in the default scheduler. In Hadoop a program known as Map-Reduce is used to collect data according to query. MapReduce is a powerful data processing platform for commercial and academic applications. MapReduce platforms run on dedicated environments like clusters or clouds. As Hadoop is used for huge amount of data, scheduling in Hadoop must be efficient for better performance.

Keywords: Map Reduce, Hadoop, Name Node, Data Node, HDFS, HBase

I. Introduction

Hadoop is a popular open-source implementation of MapReduce for the analysis of large datasets. To manage the storage resources across the cluster, Hadoop uses a distributed user-level file system. This file system — HDFS — is written in Java and designed for portability across heterogeneous hardware and software platforms. Hadoop [1][2][3] provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce [4] paradigm. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application computations in parallel close to their data. A Hadoop cluster scales computation capacity, storage capacity and IO bandwidth by simply adding commodity servers.

The components of Hadoop are

HDFS	Distributed File System
MAPREDUCE	Distributed computation framework

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

HBASE	Column-oriented table service
PIG	Dataflow language and parallel execution framework
HIVE	Data warehouse infrastructure
ZOOKEEPER	Distributed coordination service
CHUKWA	System for collecting management data
AVRO	Data serialization system

Hadoop is an Apache project. All components are available via the Apache open source license. Yahoo! has developed and contributed to 80% of the core of Hadoop (HDFS and MapReduce). HBase was originally developed at Powerset, now a department at Microsoft. Hive [15] was originated and developed at Facebook. Pig [5], ZooKeeper [6], and Chukwa were originated and developed at Yahoo! Avro was originated at Yahoo! and is being co-developed with Cloudera. HDFS is the file system component of Hadoop. While the interface to HDFS is patterned after the UNIX file system, faithfulness to standards was sacrificed in favor of improved performance for the applications at hand. HDFS stores file system metadata and application data separately. As in other distributed file systems, like PVFS [9], Lustre [7] and GFS[8], HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols.

II. Architechture

Hadoop [1] is an open source framework that implements the MapReduce parallel programming model [4]. Hadoop is composed of a MapReduce engine and a user-level file system that manages storage resources across the cluster. For portability across a variety of platforms — Linux, FreeBSD, Mac OS/X, Solaris, and Windows — both components are written in Java and only require commodity hardware.

A. MapReduce Engine

In the MapReduce model, computation is divided into a *map* function and a *reduce* function. The map function takes a key/value pair and produces one or more intermediate key/value pairs. The reduce function then takes these intermediate key/value pairs and merges all values corresponding to a single key [13]. The map function can run independently on each key/value pair, exposing enormous amounts of parallelism. Similarly, the reduce function can run independently on each intermediate key, also exposing significant parallelism. In Hadoop, a centralized JobTracker service is responsible for splitting the input data into pieces for processing by independent map and reduces tasks, scheduling each task on a cluster node for execution, and recovering from failures by re-running tasks. On each node, a Task Tracker service runs MapReduce tasks and periodically contacts the JobTracker to report task completions and request new tasks. By default, when a

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

new task is received, a new JVM instance will be spawned to execute it.

B. Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) provides global access to files in the cluster [10]. For maximum portability, HDFS is implemented as a user-level file system in Java which exploits the native file system on each node, such as ext3 or NTFS, to store data. Files in HDFS are divided into large blocks, typically 64MB, and each block is stored as a separate file in the local file system[13].

HDFS is implemented by two services: the *NameNode* and *DataNode*. The *NameNode* is responsible for maintaining the HDFS directory tree, and is a centralized service in the cluster operating on a single node. Clients contact the NameNode in order to perform common file system operations, such as open, close, rename, and delete. The NameNode does not store HDFS data itself, but rather maintains a mapping between HDFS file name, a list of blocks in the file, and the DataNode(s) on which those blocks are stored. In addition to a centralized NameNode, all remaining cluster nodes provide the *DataNode* service. Each DataNode stores HDFS blocks on behalf of local or remote clients. Each block is saved as a separate file in the node's local file system. Because the DataNode abstracts away details of the local storage arrangement, all nodes do not have to use the same local file system. Blocks are created or destroyed on DataNodes at the request of the NameNode, which validates and processes requests from clients. Although the NameNode manages the namespace, clients communicate directly with DataNodes in order to read or write data at the HDFS block level. Hadoop MapReduce applications use storage in a manner that is different from general-purpose computing [11]. First, the data files accessed are large, typically tens to hundreds of gigabytes in size. Second, these files are manipulated via streaming access patterns typical of batch-processing workloads. When reading files, large data segments (several hundred kilobytes or more) are retrieved per operation, with successive requests from the same client iterating through a file region sequentially. Similarly, files are also written in a sequential manner. This emphasis on streaming workloads is evident in the design of HDFS. First, a simple coherence model (write-once, read-many) is used that does not allow data to be modified once written. This is well suited to the streaming access pattern of target applications, and improves cluster scaling by simplifying synchronization requirements.

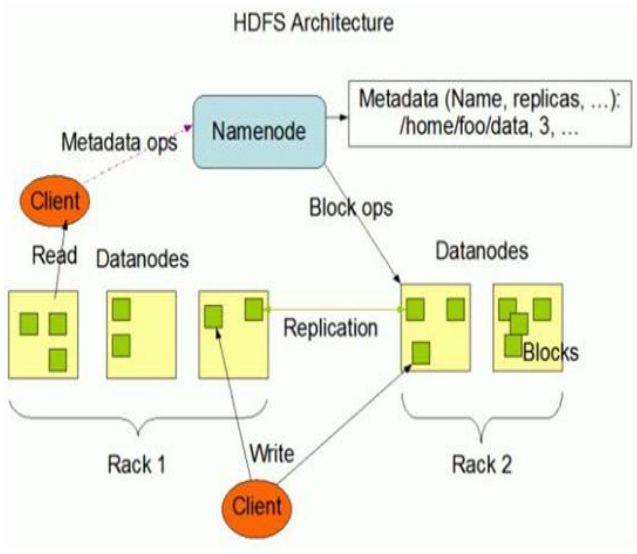
Second, each file in HDFS is divided into large blocks for storage and access, typically 64MB in size. Portions of the file can be stored on different cluster nodes, balancing storage resources and demand. Manipulating data at this granularity is efficient because streaming-style applications are likely to read or write the entire block before moving on to the next. In addition, this design choice improves performance by decreasing the amount of metadata that must be tracked in the file system, and allows access latency to

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

be amortized over a large volume of data. Thus, the file system is optimized for high bandwidth instead of low latency. This allows non-interactive applications to process data at the fastest rate. To read an HDFS file, client applications simply use a standard Java file input stream, as if the file was in the native file system. First, the NameNode is contacted to request access permission. If granted, the NameNode will translate the HDFS filename into a list of the HDFS block IDs comprising that file and a list of DataNodes that store each block, and return the lists to the client. Next, the client opens a connection to the "closest" DataNode (based on Hadoop rack-awareness, but optimally the same node) and requests a specific block ID. That HDFS block is returned over the same connection, and the data delivered to the application. To write data to HDFS, client applications see the HDFS file as a standard output stream. Internally, however, stream data is first fragmented into HDFS-sized blocks (64MB) and then smaller packets (64kB) by the client thread. Each packet is enqueued into a FIFO that can hold up to 5MB of data, thus decoupling the application thread from storage system latency during normal operation. A second thread is responsible for dequeuing packets from the FIFO, coordinating with the NameNode to assign HDFS block IDs and destinations, and transmitting blocks to the DataNodes (either local or remote) for storage. A third thread manages acknowledgements from the DataNodes that data has been committed to disk.

C. HDFS Replication

For reliability, HDFS implements an automatic replication system[13]. By default, two copies of each block are stored by different DataNodes in the same rack and a third copy is stored on a DataNode in a different rack (for greater reliability). Thus, in normal cluster operation, each DataNode is servicing both local and remote clients simultaneously. HDFS replication is transparent to the client application. When writing a block, a pipeline is established whereby the client only communicates with the first DataNode, which then echoes the data to a second DataNode, and so on, until the desired number of replicas have been created. The block is only finished when all nodes in this replication pipeline have successfully committed all data to disk. DataNodes periodically report a list of all blocks stored to the NameNode, which will verify that each file is sufficiently replicated and, in the case of failure, instruct DataNodes to make additional copies.



International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

III. File Operations And Replica Management

A. File Read and Write

An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model[12]. The HDFS client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease by sending a heartbeat to the NameNode. When the file is closed, the lease is revoked. The lease duration is bound by a soft limit and a hard limit. Until the soft limit expires, the writer is certain of exclusive access to the file. If the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. If after the hard limit expires (one hour) and the client has failed to renew the lease, HDFS assumes that the client has quit and will automatically close the file on behalf of the writer, and recover the lease. The writer's lease does not prevent other clients from reading the file; a file may have many concurrent readers. An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically 64 KB), the data are pushed to the pipeline. The next packet can be pushed to the pipeline before receiving the acknowledgement for the previous packets. The number of outstanding packets is limited by the outstanding packets window size of the client. After data are written to an HDFS file, HDFS does not provide any guarantee that data are visible to a new reader until the file is closed. If a user application needs the visibility guarantee, it can explicitly call the *hflush* operation. Then the current packet is immediately pushed to the pipeline, and the hflush operation will wait until all DataNodes in the pipeline acknowledge the successful transmission of the packet. All data written before the hflush operation are then certain to be visible to readers.

A replica stored on a DataNode may become corrupted because of faults in memory, disk, or network. HDFS generates and stores checksums for each data block of an HDFS file. Checksums are verified by the HDFS client while reading to help detect any corruption caused either by client, DataNodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. A DataNode stores checksum in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are shipped to the client. The client computes the checksum for the received data and verifies that the newly

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

computed checksums matches the checksums it received. If not, the client notifies the NameNode of the corrupt replica and then fetches a different replica of the block from another DataNode. When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested. HDFS permits a client to read a file that is open for writing. When reading a file open for writing, the length of the last block still being written is unknown to the NameNode. In this case, the client asks one of the replicas for the latest length before starting to read its content. The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes. However, many efforts have been put to improve its read/write response time in order to support applications like Scribe that provide real-time data streaming to HDFS, or HBase that provides random, realtime access to large tables.

B. Block Placement

For a large cluster, it may not be practical to connect all nodes in a flat topology. A common practice is to spread the nodes across multiple racks. Nodes of a rack share a switch, and rack switches are connected by one or more core switches[12]. Communication between two nodes in different racks has to go through multiple switches. In most cases, network bandwidth between nodes in the same rack is greater than network bandwidth between nodes in different racks. HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing up their distances to their closest common ancestor. A shorter distance between two nodes means that the greater bandwidth they can utilize to transfer data. HDFS allows an administrator to configure a script that returns a node's rack identification given a node's address. The NameNode is the central place that resolves the rack location of each DataNode. When a DataNode registers with the NameNode, the NameNode runs a configured script to decide which rack the node belongs to. If no such a script is configured, the NameNode assumes that all the nodes belong to a default single rack. The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. Currently HDFS provides a configurable block placement policy interface so that the users and researchers can experiment and test any policy that's optimal for their applications. The default HDFS block placement policy provides a tradeoff between minimizing the

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

write cost, and maximizing data reliability, availability and aggregate read bandwidth. When a new block is created, HDFS places the first replica on the node where the writer is located, the second and the third replicas on two different nodes in a different rack, and the rest are placed on random nodes with restrictions that no more than one replica is placed at one node and no more than two replicas are placed in the same rack when the number of replicas is less than twice the number of racks. The choice to place the second and third replicas on a different rack better distributes the block replicas for a single file across the cluster. If the first two replicas were placed on the same rack, for any file, two-thirds of its block replicas would be on the same rack. After all target nodes are selected, nodes are organized as a pipeline in the order of their proximity to the first replica. Data are pushed to nodes in this order. For reading, the NameNode first checks if the client's host is located in the cluster. If yes, block locations are returned to the client in the order of its closeness to the reader. The block is read from DataNodes in this preference order. (It is usual for MapReduce applications to run on cluster nodes, but as long as a host can connect to the NameNode and DataNodes, it can execute the HDFS client. This policy reduces the inter-rack and inter-node write traffic and generally improves write performance. Because the chance of a rack failure is far less than that of a node failure, this policy does not impact data reliability and availability guarantees. In the usual case of three replicas, it can reduce the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than three. The default HDFS replica placement policy can be summarized as follows:

1. No DataNode contains more than one replica of any block.
2. No rack contains more than two replicas of the same block, provided there are sufficient racks on the cluster.

C. Replication management

The NameNode endeavors to ensure that each block always has the intended number of replicas. The NameNode detects that a block has become under- or over-replicated when a block report from a DataNode arrives. When a block becomes over replicated, the NameNode chooses a replica to remove. The NameNode will prefer not to reduce the number of racks that host replicas, and secondly prefer to remove a replica from the DataNode with the least amount of available disk space. The goal is to balance storage utilization across DataNodes without reducing the block's availability. When a block becomes under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. A background thread periodically scans the head of the replication queue to decide where to place new replicas. Block replication follows a similar policy as that of the new block placement. If the number of existing replicas is one, HDFS places the next replica on a different rack. In case that

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

the block has two existing replicas, if the two existing replicas are on the same rack, the third replica is placed on a different rack; otherwise, the third replica is placed on a different node in the same rack as an existing replica. Here the goal is to reduce the cost of creating new replicas. The NameNode also makes sure that not all replicas of a block are located on one rack. If the NameNode detects that a block's replicas end up at one rack, the NameNode treats the block as under-replicated and replicates the block to a different rack using the same block placement policy described above. After the NameNode receives the notification that the replica is created, the block becomes over-replicated. The NameNode then will decides to remove an old replica because the over replication policy prefers not to reduce the number of racks.

D. Balancer

HDFS block placement strategy does not take into account DataNode disk space utilization [12]. This is to avoid placing new—more likely to be referenced—data at a small subset of the DataNodes. Therefore, data might not always be placed uniformly across DataNodes. Imbalance also occurs when new nodes are added to the cluster. The balancer is a tool that balances disk space usage on an HDFS cluster. It takes a threshold value as an input parameter, which is a fraction in the range of (0, 1). A cluster is balanced if for each DataNode, the utilization of the node (ratio of used space at the node to total capacity of the node) differs from the utilization of the whole cluster (ratio of used space in the cluster to total capacity of the cluster) by no more than the threshold value. The tool is deployed as an application program that can be run by the cluster administrator. It iteratively moves replicas from DataNodes with higher utilization to DataNodes with lower utilization. One key requirement for the balancer is to maintain data availability. When choosing a replica to move and deciding its destination, the balancer guarantees that the decision does not reduce either the number of replicas or the number of racks. The balancer optimizes the balancing process by minimizing the inter-rack data copying. If the balancer decides that a replica A needs to be moved to a different rack and the destination rack happens to have a replica B of the same block, the data will be copied from replica B instead of replica A. A second configuration parameter limits the bandwidth consumed by rebalancing operations. The higher the allowed bandwidth, the faster a cluster can reach the balanced state, but with greater competition with application processes.

E. Block Scanner

Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data [12]. In each scan period, the block scanner adjusts the read bandwidth in order to complete the verification in a configurable period. If a client reads a complete block and checksum verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica. The verification time of each block

International Journal For Advanced Research In Science & Technology
A peer reviewed international journal
www.ijarst.in
ISSN: 2457-0362

is stored in a human readable log file. At any time, there are up to two files in toplevel DataNode directory, current and prev logs. New verification times are appended to current file. Correspondingly each DataNode has an in-memory scanning list ordered by the replica's verification time. Whenever a read client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt, but does not schedule deletion of the replica immediately. Instead, it starts to replicate a good copy of the block. Only when the good replica count reaches the replication factor of the block the corrupt replica is scheduled to be removed. This policy aims to preserve data as long as possible. So even if all replicas of a block are corrupt, the policy allows the user to retrieve its data from the corrupt replicas.

F. Decommissioing

The cluster administrator specifies which nodes can join the cluster by listing the host addresses of nodes that are permitted to register and the host addresses of nodes that are *not* permitted to register [12]. The administrator can command the system to re-evaluate these include and exclude lists. A present member of the cluster that becomes excluded is marked for decommissioning. Once a DataNode is marked as decommissioning, it will not be selected as the target of replica placement, but it will continue to serve read requests. The NameNode starts to schedule replication of its blocks to other DataNodes. Once the NameNode detects that all blocks on the decommissioning DataNode are replicated, the node enters the decommissioned state. Then it can be safely removed from the cluster without jeopardizing any data availability.

G. Inter-Cluster Data Copy

When working with large datasets, copying data into and out of a HDFS cluster is daunting[12]. HDFS provides a tool called DistCp for large inter/intra-cluster parallel copying. It is a MapReduce job; each of the map tasks copies a portion of the source data into the destination file system. The MapReduce framework automatically handles parallel task scheduling, error detection and recovery.

IV. Conclusion

Big Data (Hadoop) is in huge demand in the market now a days. As there is huge amount of data is lying in the industry but there is not tool to handle it and hadoop can implemented on low cost hardware and can be used by large set of audience on large number of dataset. In Hadoop MapReduce is the most important component in Hadoop. In this document we have studied the architecture of Hadoop and the File Operations and Replica management in Hadoop

References

[1]ApacheHadoop.http://hadoop.apache.org/
[2] J. Venner, Pro Hadoop. Apress, June 22, 2009.
[3] T. White, Hadoop: The Definitive Guide. O'Reilly Media, Yahoo! Press, June 5, 2009.
[4] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large

Clusters," In Proc. of the 6th Symposium on Operating Systems Design and Implementation, San Francisco CA, Dec. 2004.

[5] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C.Olston, B. Reed, S. Srinivasan, U. Srivastava. "Building a High-Level Dataflow System on top of MapReduce: The Pig Experience," In Proc. of Very Large Data Bases, vol 2 no. 2, 2009, pp. 1414–1425

[6] S. Ghemawat, H. Gobioff, S. Leung. "The Google file system," In Proc. of ACM Symposium on Operating Systems Principles, Lake George, NY, Oct 2003, pp 29–43.

[7] Lustre File System.http://www.lustre.org

[8] M. K. McKusick, S. Quinlan. "GFS: Evolution on Fast-forward," ACM Queue, vol. 7, no. 7, New York, NY. August 2009.

[9] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. "PVFS: A parallel file system for Linux clusters," in Proc. of 4th Annual Linux Showcase and Conference, 2000, pp. 317–327.

[10] HDFS (hadoop distributed file system) architecture. http://hadoop.apache.org/common/docs/current/hdfs design.html, 2009.

[11] K. DeLathouwer, A. Y. Lee, and P. Shah. Improve database performance on file system containers in IBM DB2 universal database v8.2 using concurrent I/O on AIX. Technical report, IBM, 2004.

[12] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler , The Hadoop Distributed File System

[13] Jeffrey Shafer, Scott Rixner, and Alan L. Cox, The Hadoop Distributed Filesystem: Balancing Portability and Performance