International Journal of Enhanced Research in Management & Computer Applications, ISSN: 2319-7471
Vol. 3 Issue 10, October-2014, pp: (13-19), Impact Factor: 1.296, Available online at: www.erpublications.com

Architecture for Hadoop Distributed File Systems

S. Devi[1], Dr. K. Kamaraj[2]
[1]Asst. Prof. / Department MCA, SSM College of Engineering, Komarapalayam, Tamil Nadu, India
[2]Director / MCA, SSM College of Engineering, Komarapalayam, Tamil Nadu, India

Abstract: 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 both 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. In this paper focused on the backend architecture and working of the parts of the hadoop framework which are the map reduce for the Computational and analytics section and the Hadoop distributed file system (HDFS) for the storage section and how the files are shared in the distributed environment.

Keywords: Hadoop, HDFS (Hadoop distributed file system).

1. Introduction

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. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets [1]. The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes. Inodes record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file), and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file)[1]. The NameNode maintains the namespace tree and the mapping of blocks to DataNodes. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently. The inodes and the list of blocks that define the metadata of the name system are called the image. NameNode keeps the entire namespace image in RAM. The persistent record of the image stored in the NameNode's local native file system is called a checkpoint. The NameNode records changes to HDFS in a write-ahead log called the journal in its local native file system. The locations of block replicas are not part of the persistent checkpoint.

Each client-initiated transaction is recorded in the journal, and the journal file is flushed and synced before the acknowledgment is sent to the client. The checkpoint file is never changed by the Name Node; a new file is written when a checkpoint is created during restart, when requested by the administrator, or by the Checkpoint Node described in the next section. During startup the Name Node initializes the namespace image from the checkpoint, and then replays changes from the journal. A new checkpoint and an empty journal are written back to the storage directories before the Name Node starts serving clients. Each block replica on a Data Node is represented by two files in the local native file system. The first file contains the data itself and the second file records the block's metadata including checksums for the data and the generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in needs only half of the space of the full block on the local drive. The namespace ID is assigned to the file system instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus protecting the integrity of the file system. A Data Node that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID.

After the handshake the Data Node registers with the Name Node. Data Nodes persistently store their unique storage IDs [2].The storage ID is an internal identifier of the Data Node, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the Data Node when it registers with the Name Node for the first time and never changes after that. A Data Node identifies block replicas in its possession to the Name Node by sending a block report[3]. A block report contains the block ID, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the Data Node registration. Subsequent block

reports are sent every hour and provide the NameNode with an up-to-date view of where block replicas are located on the cluster. User applications access the file system using the HDFS client, a library that exports the HDFS file system interface. Like most conventional file systems, HDFS supports operations to read, write and delete files, and operations to create and delete directories .When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. The list is sorted by the network topology distance from the client. The client contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file [4]. Choice of DataNodes for each block is likely to be different. The interactions among the client, the NameNode and the DataNodes are illustrated in Figure 1.

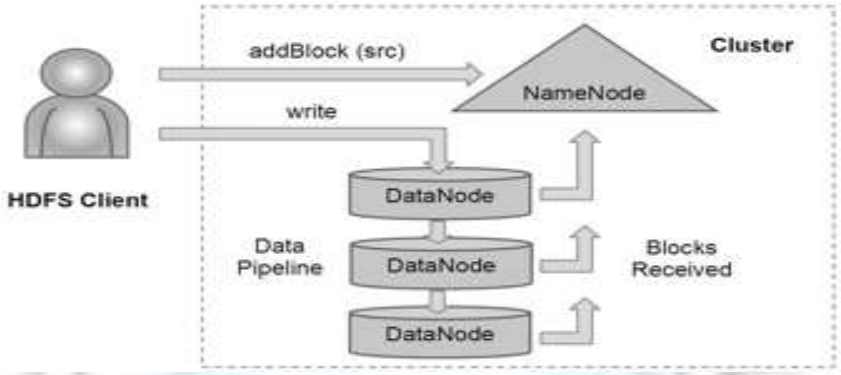

Figure 1: HDFS Client Creates a New File

Checkpoint Node

The Name Node in HDFS, in addition to its primary role serving client requests, can alternatively execute either of two other roles, either a CheckpointNode or a BackupNode. The role is specified at the node startup. The CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal. The CheckpointNode usually runs on a different host from the NameNode since it has the same memory requirements as the NameNode. It downloads the current checkpoint and journal files from the NameNode, merges them locally, and returns the new checkpoint back to the NameNode.Creating periodic checkpoints is one way to protect the file system metadata. The system can start from the most recent checkpoint if all other persistent copies of the namespace image or journal are unavailable. Creating a checkpoint also lets the NameNode truncate the journal when the new checkpoint is uploaded to the NameNode.

Backup Node

A recently introduced feature of HDFS is the BackupNode. Like a CheckpointNode, the BackupNode is capable of creating periodic checkpoints, but in addition it maintains an in-memory, up-to-date image of the file system namespace that is always synchronized with the state of the NameNode. The Backup Node accepts the journal stream of namespace transactions from the active Name Node, saves them in journal on its own storage directories, and applies these transactions to its own namespace image in memory[4].

The Backup Node can create a checkpoint without downloading checkpoint and journal files from the active NameNode, since it already has an up-to-date namespace image in its memory. This makes the checkpoint process on the BackupNode more efficient as it only needs to save the namespace into its local storage directories. The BackupNode can be viewed as a read-only NameNode. It contains all file system metadata information except for block locations. It can perform all operations of the regular NameNode that do not involve modification of the namespace or knowledge of block locations. Use of a BackupNode provides the option of running the NameNode without persistent storage, delegating responsibility of persisting the namespace state to the BackupNode.

Upgrades and File system Snapshots

During software upgrades the possibility of corrupting the file system due to software bugs or human mistakes increases. The purpose of creating snapshots in HDFS is to minimize potential damage to the data stored in the system during upgrades. The snapshot mechanism lets administrators persistently save the current state of the file system, so

that if the upgrade results in data loss or corruption it is possible to rollback the upgrade and return HDFS to the namespace and storage state as they were at the time of the snapshot. The snapshot (only one can exist) is created at the cluster administrator's option whenever the system is started. If a snapshot is requested, the NameNode first reads the checkpoint and journal files and merges them in memory. Then it writes the new checkpoint and the empty journal to a new location, so that the old checkpoint and journal remain unchanged. During handshake the NameNode instructs DataNodes whether to create a local snapshot. The local snapshot on the DataNode cannot be created by replicating the directories containing the data files as this would require doubling the storage capacity of every DataNode on the cluster. Instead each DataNode creates a copy of the storage directory and hard links existing block files into it. When the DataNode removes a block it removes only the hard link, and block modifications during appends use the copy-on-write technique. Thus old block replicas remain untouched in their old directories.

2. File I/O Operations and Replica Management

2.1. 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. 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 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 acknowledgment for the previous packets.

In a cluster of thousands of nodes, failures of a node (most commonly storage faults) are daily occurrences. 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 checksums in a metadata file separate from the block's data file.

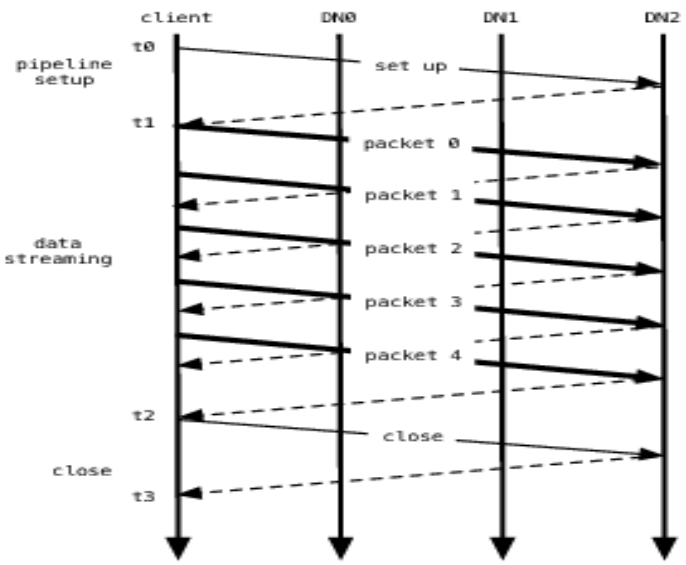

Figure 2.1: Data Pipeline While Writing a Bloc

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.

2.2. 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. 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. Figure 2.2 describes a cluster with two racks, each of which contains three nodes. 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 the distances to their closest common ancestor.

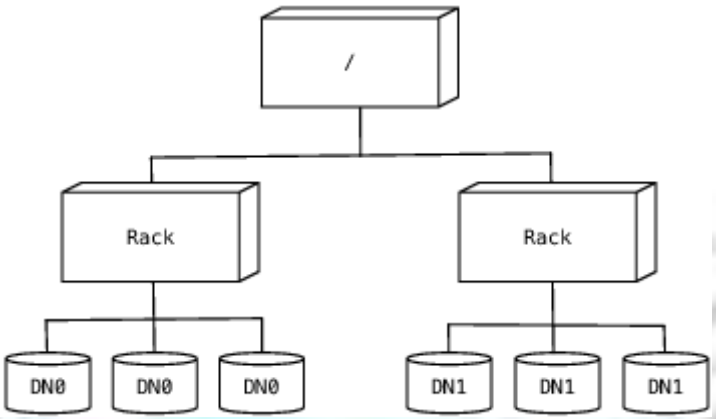

Figure 2.2: Cluster Topology

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 the 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 alternate policies that are optimal for their applications.

The default HDFS block placement policy provides a tradeoff between minimizing the 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 are placed on two different nodes in a different rack. The rest are placed on random nodes with restrictions that no more than one replica is placed at any one node and no more than two replicas are placed in the same rack, if possible. 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. 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.

2.3. 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 [4]. 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 new block placement. If the number of existing replicas is one, HDFS places the next replica on a different rack. In case that 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 mis-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.

2.4. Balancer

HDFS block placement strategy does not take into account DataNode disk space utilization. This is to avoid placing new—more likely to be referenced—data at a small subset of the DataNodes with a lot of free storage. 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 between 0 and 1. A cluster is balanced if, for each DataNode, the utilization of the node differs from the utilization of the whole 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.

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.

2.5. Block Scanner

Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data. 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 is stored in a human-readable log file. At any time there are up to two files in the top-level DataNode directory, the current and previous logs. New verification times are appended to the 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.

3. Durability of Data

Replication of data three times is a robust guard against loss of data due to uncorrelated node failures. It is unlikely Yahoo! has ever lost a block in this way; for a large cluster, the probability of losing a block during one year is less than 0.005. The key understanding is that about 0.8 percent of nodes fail each month. (Even if the node is eventually recovered, no effort is taken to recover data it may have hosted.) So for the sample large cluster as described above, a node or two is lost each day. That same cluster will re-create the 60 000 block replicas hosted on a failed node in about two minutes: re-replication is fast because it is a parallel problem that scales with the size of the cluster. The probability of several nodes failing within two minutes such that all replicas of some block are lost is indeed small.

Correlated failure of nodes is a different threat. The most commonly observed fault in this regard is the failure of a rack or core switch. HDFS can tolerate losing a rack switch (each block has a replica on some other rack). Some failures of a core switch can effectively disconnect a slice of the cluster from multiple racks, in which case it is probable that some blocks will become unavailable. In either case, repairing the switch restores unavailable replicas to the cluster. Another kind of correlated failure is the accidental or deliberate loss of electrical power to the cluster. If the loss of power spans racks, it is likely that some blocks will become unavailable. But restoring power may not be a remedy because one-half to one percent of the nodes will not survive a full power-on restart. Statistically, and in practice, a large cluster will lose a handful of blocks during a power-on restart.

4. Features for Sharing HDFS

As the use of HDFS has grown, the file system itself has had to introduce means to share the resource among a large number of diverse users. The first such feature was a permissions framework closely modeled on the UNIX permissions scheme for file and directories. In this framework, files and directories have separate access permissions for the owner, for other members of the user group associated with the file or directory, and for all other users. In the earlier version of HDFS, user identity was weak: you were who your host said you are. When accessing HDFS, the application client simply queries the local operating system for user identity and group membership. In the new framework, the application client must present to the name system credentials obtained from a trusted source. Different credential administrations are possible; the initial implementation uses Kerberos. The user application can use the same framework to confirm that the name system also has a trustworthy identity. And the name system also can demand credentials from each of the data nodes participating in the cluster.

The total space available for data storage is set by the number of data nodes and the storage provisioned for each node. Early experience with HDFS demonstrated a need for some means to enforce the resource allocation policy across user communities. Not only must fairness of sharing be enforced, but when a user application might involve thousands of hosts writing data, protection against applications inadvertently exhausting resources is also important. For HDFS, because the system metadata are always in RAM, the size of the namespace (number of files and directories) is also a finite resource. To manage storage and namespace resources, each directory may be assigned a quota for the total space occupied by files in the sub-tree of the namespace beginning at that directory. A separate quota may also be set for the total number of files and directories in the sub-tree.

4.2. Scaling and HDFS Federation

Scalability of the NameNode has been a key struggle because the NameNode keeps all the namespace and block locations in memory, the size of the NameNode heap limits the number of files and also the number of blocks addressable. This also limits the total cluster storage that can be supported by the NameNode. Users are encouraged to create larger files, but this has not happened since it would require changes in application behavior. Furthermore, we are seeing new classes of applications for HDFS that need to store a large number of small files. Quotas were added to manage the usage, and an archive tool has been provided, but these do not fundamentally address the scalability problem. A new feature allows multiple independent namespaces (and NameNodes) to share the physical storage within a cluster. Namespaces use blocks grouped under a Block Pool. Block pools are analogous to logical units (LUNs) in a SAN storage system and a namespace with its pool of blocks is analogous to a file system volume[5].
This approach offers a number of advantages besides scalability: it can isolate namespaces of different applications improving the overall availability of the cluster. Block pool abstraction allows other services to use the block storage with perhaps a different namespace structure. We plan to explore other approaches to scaling such as storing only partial namespace in memory, and truly distributed implementation of the NameNode.

5. Data Organization

5.1. Data Blocks

HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 64MB. Thus, an HDFS file is chopped up into 64 MB chunks, and if possible, each chunk will reside on a different DataNode.

5.2. Staging

A client request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When

the local file accumulates data worth over one HDFS block size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it[6].

The NameNode responds to the client request with the identity of the DataNode and the destination data block. Then the client flushes the block of data from the local temporary file to the specified DataNode. When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost. The above approach has been adopted after careful consideration of target applications that run on HDFS. These applications need streaming writes to files[7].

Conclusion

This paper describes the hadoop using distributed file systems. Hadoop Distributed File System (HDFS) is a Java-based file system that provides scalable and reliable data storage that is designed to span large clusters of commodity servers. This paper gives the details of how the data is splitted, stored and shared from one system to another system. It also focuses on file I/O operations and replica management. For bulk amount of the data the block placement method is used to place the block of data by scanning the block at each node.Thereby, ensuring the durability for the data and storage of data is acquired by staging. Thus the Hadoop Distributed File System serves the future data management in an efficient manner.

References

[1]. Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler Yahoo! Sunnyvale, California USA {Shv, Hairong, SRadia, Chansler}@Yahoo-Inc.com," The Hadoop Distributed File System".
[2]. K. V. Shvachko, "HDFS Scalability: The limits to growth," ;login:.April 2010, pp. 6–16.
[3]. W. Tantisiriroj, S. Patil, G. Gibson. "Data-intensive file systems for Internet services: A rose by any other name ..." Technical Report CMUPDL-08-114, Parallel Data Laboratory, Carnegie Mellon niversity,Pittsburgh, PA, October 2008.
[4]. The Hadoop Distributed File System Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler Yahoo! Sunnyvale, California USA {Shv, Hairong, SRadia, Chansler}@Yahoo-Inc.com.
[5]. J. Venner, Pro Hadoop. Apress, June 22, 2009.
[6]. S. Weil, S. Brandt, E. Miller, D. Long, C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," In Proc. of the 7th Symposium on Operating Systems Design and Implementation, Seattle, WA, November 2006.
[7]. B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J.Zelenka, B. Zhou, "Scalable Performance of the Panasas Parallel file System", In Proc. of the 6th USENIX Conference on File and Storage Technologies, San Jose, CA, February 2008.