# Hadoop Design, Architecture & MapReduce Performance

## IO Processing Challenges

Cloud computing is changing the IT landscape in a profound way. Via cloud computing, dynamically scalable, virtualized resources are provided as a service over the Internet. Large firms such as Google, Amazon, IBM, Microsoft, or Apple are providing cloud models that incorporate sound data storage solutions. Today, most companies around the world are processing vast amounts of data. In 2011, IDC (*www.idc.com*) estimated the total world wide data size (labeled the digital data universe) at 1.8 ZB (zettabytes - 10^21). As a comparison, the numeric value for a TB (terabyte) equals to 10^12 (the binary usage equals to 2^40). The ever larger data pools required by most companies today obviously have a profound impact not only on the HW storage requirements and user applications, but also on the file system design, the file system implementation, as well as the actual IO performance and scalability behavior of today's IT environments. To illustrate, the New York Stock Exchange generates approximately 1 TB of new trade data each day. CERN's Large Hadron Collider produces approximately 15 PB (petabytes) of scientific data per year. While hard drive (HDD) storage capacity has increased significantly over the years, the actual HDD data access rate has not been improved much. To illustrate, hypothetically, a contemporary 1TB disk with a 100MB/second transfer speed requires more than 2 1/2 hours to read all the data. Solid State Disks (SSD) may be an option for some IO workloads, but the rather high price per unit (next to a specific workload behavior required to even utilize the performance behavior of the disks) deters many companies to deploy SSD's in large volumes. To improve aggregate IO throughput, the obvious answer is to read/write to/from multiple disks. Assuming a hypothetical setup with 100 drives, each holding 1/100 of the 1TB data pool discussed above. If all disks are accessed in parallel at 100MB/second, the 1TB can be fetched in less than 2 minutes. Any distributed IT environment faces several, similar challenges. First, an appropriate redundancy at the HW and the SW level has to be designed into the solution so that the environment meets the availability, reliability, maintainability, as well as performance goals and objectives. Second, most distributed environments (the applications) require combining the distributed data from multiple sources for post-processing and/or visualization purposes. Ergo, most distributed processing environments are powered by rather large SAN subsystems and a parallel file system solution such as IBM's GPFS, Lustre (GNU/GPL), or Red Hats GFS, respectively.

## Hadoop Introduction

To address the above mentioned issues, the Hadoop framework is designed to provide a reliable, shared storage and analysis infrastructure to the user community. The storage portion of the Hadoop framework is provided by a distributed file system solution such as HDFS, while the analysis functionality is presented by *MapReduce*. Several other components (discussed later in this report) are part of the overall Hadoop solution suite. The *MapReduce* functionality is designed as a tool for deep data analysis and the transformation of very large data sets. Hadoop enables the users to explore/analyze complex data sets by utilizing customized analysis scripts/commands. In other words, via the customized *MapReduce* routines, unstructured data sets can be distributed, analyzed, and explored across thousands of shared-nothing processing systems/clusters/nodes. Hadoop's HDFS replicates the data onto multiple nodes to safeguard the environment from any potential data-loss (to illustrate, if 1 Hadoop node gets fenced, there are at least 2 other nodes holding the same data set).

## Hadoop verses Conventional Database Systems

Throughout the IT community, there are many discussions revolving around comparing MapReduce to traditional RDBMS solutions. In a nutshell, MapReduce and RDBMS systems reflects solutions for completely different (sized) IT processing environments and hence, an actual comparison results into identifying the opportunities and limitations of both solutions based on their respective functionalities and focus areas. To begin with, the data sets processed by traditional database (RDBMS) solutions are normally much smaller than the data pools utilized in a Hadoop environment (see Table 1). Hence, unless an IT infrastructure processes TB's or PB's of unstructured data in a highly parallel environment, the statement can be made that the performance of Hadoop executing MapReduce queries will be sub-par compared to SQL queries running against an (optimized) relational database. Hadoop utilizes a brute-force access method while RDBMS solutions bank on optimized accessing routines such as indexes, as well as read-ahead and write-behind

techniques. Hence, Hadoop really only excels in environments that reveal a massive parallel processing infrastructure where the data is unstructured to the point where no RDBMS optimization techniques can be applied to boost the performance of the queries.

Today's RDBMS solutions do not scale to thousands of nodes. At some level, the vast amount of brute-force processing power provided by Hadoop will outperform the optimized, but scaling restricted relational access methods. To conduct large-scale batch analysis studies with traditional RDBMS systems (where the workload is dominated by seek operations) is from a performance perspective another daunting task. This is mainly due to the performance gap between the disk seek-time and the disk IO transfer rate *potentials*, respectively. Hadoop does provide a dataset streaming facility that takes advantage of the transfer rate *potential*. While updating a small proportion of database records, the traditional B-tree data structures used in relational databases (that are basically limited by the seek rate) works sufficiently well. On the other hand, while updating very large numbers of database system records, a B-tree solution is less efficient than the sort/merge infrastructure embedded into the MapReduce routines. From a structural perspective, there are significant differences between MapReduce and RDBMS solutions. Structured data reflects data that is ordered into entities of a defined format or database tables that conform to a particular, predefined schema (basically the territory of RDBMS solutions). Semi-structured data represents looser formats (may still entail a schema) that may either be ignored or utilized as guidelines for the structure of the data. To illustrate, a spreadsheet represents its *structure* as the grid of cells, although the cells may hold any form of data. Unstructured data is not affiliated with any particular internal structure (such as plain text or image data). The point to be made is that MapReduce operates very well on unstructured or semi-structured data, as MapReduce is designed to interpret the data at processing time. The input keys and values for MapReduce are not an intrinsic property of the data per se, but are chosen by the user analyzing the data. On the other hand, relational data is commonly normalized to retain data integrity, as well as to remove redundancy. The traditional normalization process reflects a problem for MapReduce based solutions, as normalization crafts reading a record a non-local operation, which defies the MapReduce design principle that is based on the idea of performing (high-speed) streaming read operations.

Table 1: RDBMS and MapReduce Features

| Feature/Application | Traditional RDBMS | MapReduce |
|---|---|---|
| Operating Data Size | Gigabytes + | Petabytes + |
| Access Pattern | Interactive and Batch | Batch |
| Update Scenarios | Repeated Read and Write | Write Once, Repeated Read |
| Data Structure | Static Schema | Dynamic Schema |
| Scaling Potential | Non-linear | Somewhat Close to linear |

**Hadoop Subprojects (Partial View of the Ecosystem)**

While Hadoop is in general best known for the MapReduce and the HDFS components, there are several other subprojects that as a unit, comprise the Hadoop offering:

- Hadoop Common: The common utilities that support the other Hadoop modules.
- HDFS: A distributed file system that provides high-throughput access to application data.
- Hadoop YARN: A framework for job scheduling and cluster resource management.
- Hadoop MapReduce: A YARN-based system for parallel processing of large data sets
- Avro: A data serialization system
- Cassandra: A scalable multi-master database with no single points of failure
- Chukwa: A data collection system for managing large distributed systems
- HBase: A scalable, distributed database that supports structured data storage for large tables
- Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying
- Mahout: A Scalable machine learning and data mining library
- Pig: A high-level data-flow language and execution framework for parallel computation
- ZooKeeper: A high-performance coordination service for distributed applications

Hadoop is basically designed to efficiently process very large data volumes by linking many commodity systems together to work as a parallel entity. In other words, the Hadoop philosophy is to

bind many smaller (and hence more reasonably priced) nodes together to form a single, cost-effective compute cluster environment. It has to be pointed out that performing compute operations on large volumes of data in a distributed setup has obviously been done before (see LLNC ASC Purple or Blue Gene). What separates Hadoop from the herd is the simplified programming model that allows users to rapidly develop and test distributed systems, as well as the efficient, automated distribution of data and work across the nodes (by effectively utilizing the underlying parallelism of the CPU cores).

**Hadoop Data Distribution**

In a Hadoop cluster environment (commodity HW is normally used to setup the cluster), the data is distributed among all the nodes during the data load phase. The HDFS splits large data files into chunks that are managed by different nodes in the cluster. Each chunk is replicated across several nodes to address single node outage or fencing scenarios. An active monitoring system re-replicates the data during node failure events. Despite the fact that the file chunks are replicated and distributed across several nodes, Hadoop operates in a single namespace and hence, the cluster content is collectively accessible. In the Hadoop programming framework, data is conceptually record-oriented. The individual input files are carved up into lines or other (application logic) specific formats. Hence, each thread executing on a cluster node processes a subset of the records. The Hadoop framework schedules these threads in proximity to the location of the data/records by utilizing *knowledge* obtained from the distributed file system. Which data chunk is operated on by a node is chosen based on the data chunks locality to a node. The design goal is that most data is read from a local disk straight into the CPU subsystem to economize on the number of network transfers necessary to complete the processing cycle. The design strategy of moving the actual computation to the data (instead of moving the data to the computation) allows Hadoop to achieve high data locality reference values that result in increased performance scenarios.

**The Hadoop MapReduce Isolated Process Environment**

The Hadoop design limits the amount of communication performed by the threads, as each individual record is processed in isolation by a single task. While such a design choice may be considered an actual systems limitation, it aids in stabilizing the entire IT framework. It has to be pointed out that the traditional Hadoop design requires applications to be developed in ways to conform to the MapReduce programming model. As the Hadoop ecosystem expanded over time, some Hadoop applications today are actually bypassing the MapReduce functionality and access data directly in either HBase or HDFS. Based on the MapReduce design, records are processed in isolation via tasks called *Mappers*. The output from the Mapper tasks is further processed by a second set of tasks, the *Reducers*, where the results from the different Mapper tasks are merged together. Individual nodes in a Hadoop cluster still communicate with each other, just to a lesser extend compared to more conventional distributed systems where application developers explicitly assemble node to node byte streams via sockets or some form of message passing interface (MPI). Hadoop node communication is performed implicitly, as chunks of data can be tagged with key names that inform Hadoop how to send related bits of information to a common destination node. Hadoop internally manages all the data transfer and cluster topology issues. By restricting the communication among nodes, Hadoop improves the reliability potential of the distributed system environment. Individual node failures are addressed by restarting the tasks on other nodes. As user-level tasks do not communicate explicitly among each other, no messages have to be exchanged by the user applications. Further, the nodes do not have to roll back to pre-arranged checkpoints to partially restart the computation process. In failure scenarios, the surviving user worker threads continue to operate, while the underlying Hadoop layer addresses the challenging aspects of partially restarting the application. It is important to point out that *MapReduce* is simply a divide-and-conquer strategy that has been used for approximately 40 years to solve big data science problems and hence, is not a component that has just been introduced by Hadoop or any other big data analysis solution.

**The Basic MapReduce Architecture**

In a nutshell, the MapReduce programming Model is to process large datasets. The model itself is based on the concept of parallel programming. Based on the parallel programming notion, processing is decomposed into *n* sub-entities that are executed concurrently. The instructions for

each sub-entity are executing simultaneously on different CPU's. Depending on the IT infrastructure setup, the CPU's may be available on the same server system or on remote nodes that are accessible via some form of interconnect/network. The MapReduce model itself is derived from the *map* and *reduce* primitives available in a functional language such as *Lisp*. A MapReduce job usually splits the input datasets into smaller chunks that are processed by the map task in a completely parallel manner. The outputs obtained from all the map tasks are then sorted by the framework and made available as input(s) into the reduce tasks. It has to be pointed out that a MapReduce model is normally suitable for long running batch jobs, as the *reduc*e function has to accumulate the results for the entire job, a process that is associated with overhead concerning the collection and submission of the processed datasets. In the MapReduce model, actual parallelism is achieved via a split/sort/merge/join process (see Figure 1) that can be described as:

- Initially, a master program (basically a central coordinator) is started by the MapReduce job that utilizes the predefined datasets as input.
- According to the Job Configuration, the master program initiates the *Ma*p and the *Reduce* programs on various systems. Next, it starts the input reader to retrieve the data from some directory (hosted in a distributed file system). The input reader then decomposes the data into small chunks and submits them to randomly chosen mapper programs. This process concludes the split phase and initiates the parallel processing stage.
- After receiving the data, the mapper program executes a user supplied map function, and generates a collection of [key, value] pairs. Each produced item is sorted and submitted to the reducer.
- The reducer program collects all the items with the same key values and invokes a user supplied reduce function to produce a single entity as a result (this is labeled the merge phase).
- The output of the reduce program is collected by the output writer (effective join phase) and this process basically terminates the parallel processing phase.

Figure 1 : MapReduce Dataflow

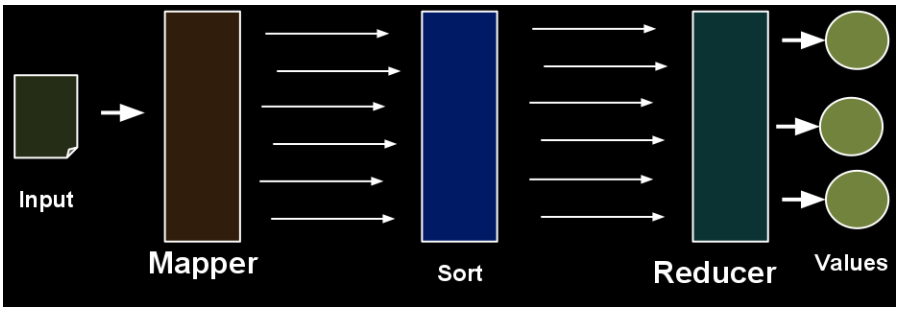



Figure courtesy of Apache/Hadoop

**Hadoop Map Reduce Architecture**

The Hadoop MapReduce MRv1 framework is based on a centralized master/slave architecture. The architecture utilizes a single master server (*JobTracker*) and several slave servers (*TaskTracker's*). Please see Appendix A for a discussion on the MapReduce MRv2 framework. The JobTracker represents a centralized program that keeps track of the slave nodes, and provides an interface infrastructure for job submission. The TaskTracker executes on each of the slave nodes where the actual data is normally stored. In other words, the JobTracker reflects the interaction point among the users and the Hadoop framework. Users submit MapReduce jobs to the JobTracker, which inserts the jobs into the pending jobs queue and executes them (normally) on a FIFO basis (it has to be pointed out that other job schedulers are available - see Hadoop Schedulers below). The JobTracker manages the map and reduce task assignments with the TaskTracker's. The TaskTracker's execute the jobs based on the instructions from the JobTracker and handle the data movement between the *map* and *reduce* phases, respectively. Any map/reduce construct basically reflects a special form of a Directed Acyclic Graph (DAG). A DAG can execute anywhere in parallel,

as long as one entity is not an ancestor of another entity. In other words, parallelism is achieved when there are no hidden dependencies among shared states. In the MapReduce model, the internal organization is based on the map function that transforms a piece of data into entities of [key, value] pairs. Each of these elements is sorted (via their key) and ultimately reaches the same cluster node where a reduce function is used to merge the values (with the same key) into a single result (see code below). The Map/Reduce DAG is organized as depicted in Figure 2.

**Map Function**

```
map(input_record){
…
emit(k1,v1)
…
emit(k2,v2)
…
}
```

**Reduce Function**

```
reduce(key,values){
while(values.has_next){
aggregate=merge(values.next)
  }
collect(key,aggregate)
}
```

Figure 2 : Representation of a Map/Reduce DAG

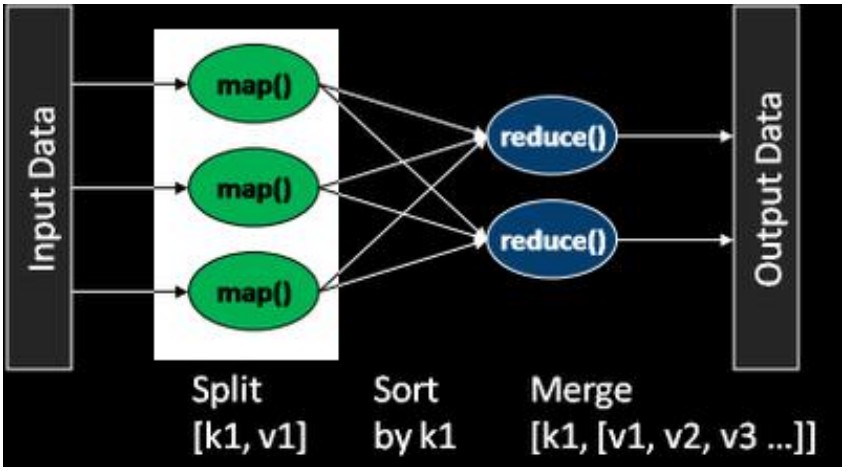



Figure courtesy of Apache/Hadoop

The Hadoop MapReduce framework is based on a *pull* model, where multiple TaskTracker's communicate with the JobTracker requesting tasks (either map or reduce tasks). After an initial setup phase, the JobTracker is informed about a job submission. The JobTracker provides a job ID to the client program, and starts allocating map tasks to idle TaskTracker's requesting work items (see Figure 3). Each TaskTracker contains a defined number of task slots based on the capacity potential of the system. Via the heartbeat protocol, the JobTracker knows the number of free slots in the TaskTracker (the TaskTracker's send heartbeat messages indicating the free slots - true for the FIFO scheduler). Hence, the JobTracker can determine the appropriate job setup for a TaskTracker based on the actual availability behavior. The assigned TaskTracker will fork a *MapTask* to execute the map processing cycle (the MapReduce framework spawns 1 MapTask for each InputSplit generated by the InputFormat). In other words, the MapTask extracts the input data from the splits by using the RecordReader and InputFormat for the job, and it invokes the user provided map function, which emits a number of [key, value] pairs in the memory buffer.

After the MapTask finished executing all input records, the commit process cycle is initiated by flushing the memory buffer to the index and data file pair. The next step consists of merging all the index and data file pairs into a single construct that is (once again) being divided up into local directories. As some map tasks are completed, the JobTracker starts initiating the reduce tasks phase. The TaskTracker's involved in this step download the completed files from the map task nodes, and basically concatenate the files into a single entity. As more map tasks are being

completed, the JobTracker notifies the involved TaskTracker's, requesting the download of the additional region files and to merge the files with the previous target file. Based on this design, the process of downloading the region files is interleaved with the on-going map task procedures.

Figure 3 : Hadoop Job Execution

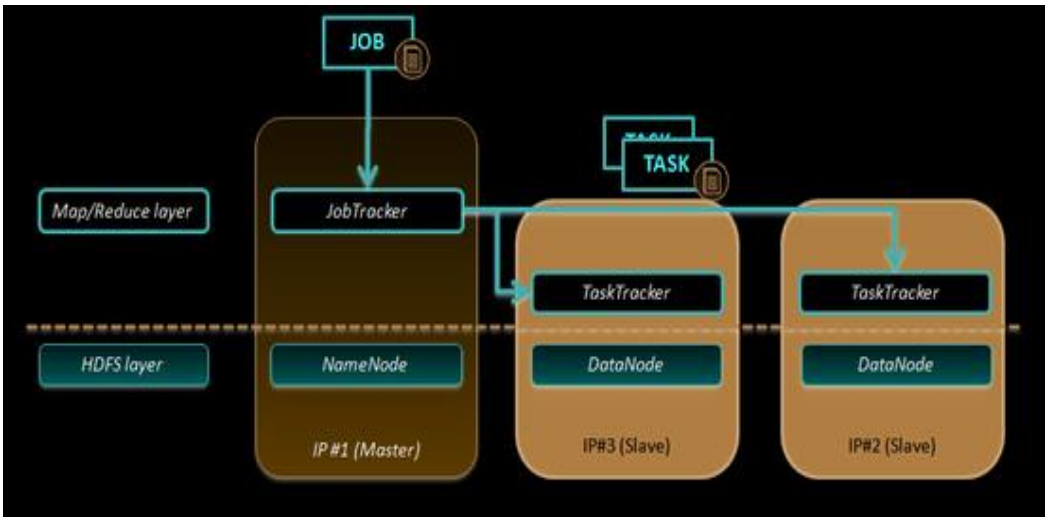



Figure courtesy of Apache/Hadoop

Eventually, all the map tasks will be completed, at which point the JobTracker notifies the involved TaskTracker's to proceed with the reduce phase. Each TaskTracker will fork a *ReduceTask* (separate JVM's are used), read the downloaded file (that is already sorted by key), and invoke the reduce function that assembles the key and aggregated value structure into the final output file (there is one file per reducer node). Each reduce task (or map task) is single threaded, and this thread invokes the reduce [key, values] function in either ascending or descending order. The output of each reducer task is written to a temp file in HDFS. When the reducer finishes processing all keys, the temp file is atomically renamed into its final destination file name.

As the MapReduce library is designed to process vast amounts of data by potentially utilizing hundreds or thousands of nodes, the library has to be able to gracefully handle any failure scenarios. The TaskTracker nodes periodically report their status to the JobTracker that oversees the overall job progress. In scenarios where the JobTracker has not been contacted by a TaskTracker for a certain amount of time, the JobTracker assumes a TaskTracker node failure and hence, reassigns the tasks to other available TaskTracker nodes. As the results of the map phase are stored locally, the data will no longer be available if a TaskTracker node goes offline. In such a scenario, all the m*ap tasks* from the failed node (regardless of the actual completion percentage) will have to be reassigned to a different TaskTracker node that will re-execute *all* the newly assigned splits. The results of the reduce phase are stored in HDFS and hence, the data is globally available even if a TaskTracker node goes offline. Hence, in a scenario where during the reduce phase a TaskTracker node goes offline, only the set of incomplete r*educe tasks* have to be reassigned to a different TaskTracker node for re-execution.

Even within today's high-performance IT infrastructures, network bandwidth is still considered a relatively scarce resource. The Hadoop design aims at conserving network bandwidth by utilizing as much local (input) data as possible. In other words, the MapReduce master node takes the location information of the input files into account and attempts to schedule a map task on a system that already contains a replica of the corresponding input data. If such a scenario is not possible, an attempt is made to schedule the map task near a replica node, basically on a system that is located within the same network (switch) environment as the node containing the data. Hence, the design goal is to assure that most input data is available locally and hence, the read requests do not consume any network bandwidth (or at least as little as possible).

## The Hadoop Streaming & Pipe Facilities

Hadoop does provides an API to MapReduce that allows developing map and reduce functions in languages other than Java. The Hadoop streaming facility utilizes Unix standard streams as the interface between Hadoop and the application. Hence, any language capable of reading standard input and writing to standard output can be used to develop MapReduce programs. While streaming is generally suited for text processing purposes, binary streams are supported as well. The map input data is passed (via standard input) to the map function that processes the input on a line by line basis, and writes the lines out to standard output. A map output [key, value] pair is written as a single tab-delimited line. Input to the reduce function follows the same format, a tab-separated key-value pair that is passed over standard input.

The reduce function reads the lines from standard input (the Hadoop framework guarantees sorting by key), and writes the results to standard output (see Figure 4). On the other hand, Hadoop pipes represent the C++ interface to Hadoop MapReduce. Unlike the streaming interface that utilizes standard input/output to communicate with the map/reduce code, pipes utilize sockets as the interface that the *TaskTracker* uses to communicate with the threads that are running the C++ map/reduce functions. In general, Hadoop pipes generate less overhead than Hadoop streaming based solutions.

Figure 4: Hadoop Framework

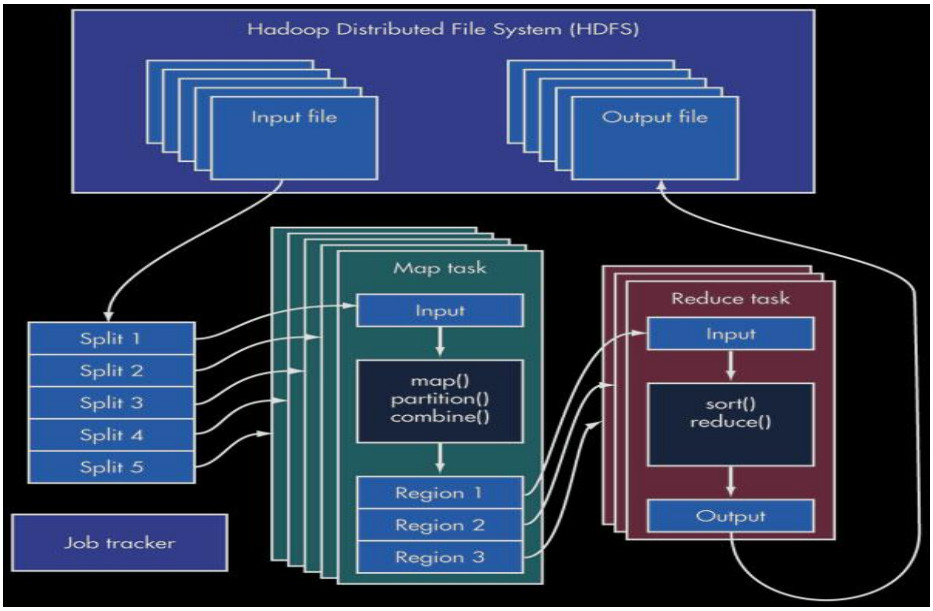



Figure courtesy of Apache/Hadoop

## Hadoop HDFS Overview

Local file systems (such as ZFS, ext4, or Btrfs) support a persistent name space, and hence view devices as being locally attached. There is no need in the file system design to enforce device-sharing semantics. Contemporary (high-speed) network/interconnect solutions allow multiple cluster nodes to share storage devices. IBM's GPFS, Lustre (GNU/GPL), or Red Hat's GFS parallel file systems take a shared, network-attached storage approach. The Hadoop Distributed File System (HDFS) represents a distributed file system that is designed to accommodate very large amounts of data (TB or PB) and to provide high-throughput (streaming) access to the data sets. Based on the HDFS design, the files are redundantly stored across multiple nodes to ensure high-availability for the parallel applications. In any IT environment, HW failures do happen. A single HDFS instance may consist of thousands of server nodes, each storing and maintaining part of the file system data. Hence, most Hadoop installations are comprised of a rather large number of HW components, where each HW component reflects a probability for a failure scenario. Therefore, fault detection and rapid,

automated recovery features are at the core of the HDFS design/architecture. To reiterate, HDFS represents a distributed file system solution designed for storing very large files (TB, PB) and where the user applications reveal *streaming* data access patterns. Or in other words, HDFS does not really provide an adequate solution in distributed environments with vast amounts of small files, random IO patterns, and/or frequent file update scenarios. Further, applications that require low latency data access (in the 10s of milliseconds) or that necessitate multiple write IO tasks are not really suited for a Hadoop setup. As a matter of fact, Hadoop only supports single write tasks, and the write requests always occur at the end of a file (no arbitrary file offset modifications are supported at this point). The HDFS design is basically focused on a write-once, read-often IO pattern where a dataset is generated, and where various analysis cycles are performed on the dataset over time. Each analysis cycle generally entails a large quantity (or all) of the dataset. Hence, the Hadoop *performance focus* is on aggregate dataset read performance, and not on any individual latency scenarios surrounding the performance behavior of reading the first few data records. It has to be pointed out that next to HDFS, Hadoop also supports other third-party file systems such as *CloudStore* (implemented in C++) or the Amazon Simple Storage Service (*S3*). Further, The Cassandra File System (CFS) reflects a HDFS compatible filesystem that was developed to replace the traditional Hadoop NameNode, Secondary NameNode and DataNode daemons, respectively.

**HDFS Design**

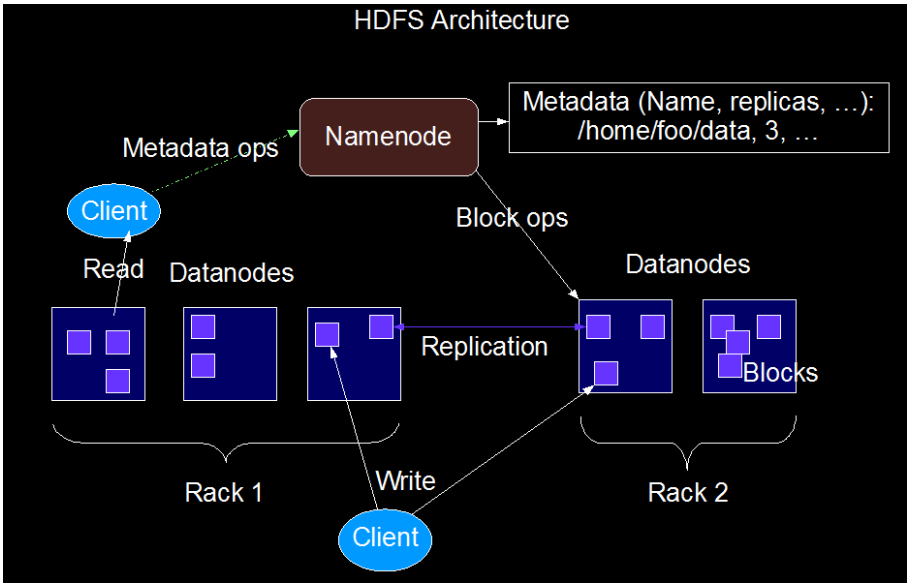
Any file system operates on either a block or an extent size. To illustrate, ZFS uses variable-sized blocks of up to 1024 kilobytes, whereas GPFS supports block sizes ranging from 16 KB to 4MB (with a default block size of 256 KB). The HDFS design also incorporates the notion of a block size, just at a much larger unit size of 64MB (default) compared to other file system solutions. The choice made for the rather large block size is based on the HDFS design philosophy to minimize/optimize the cost of any potential seek scenarios. In other words, with large block sizes, the time to transfer the data from the disk is significantly longer than the time to seek to the start of the block. Hence, the time to transfer a large file that is comprised of multiple blocks is governed by the disk transfer rate. To illustrate, assuming a disk seek time of approximately 10ms and a disk transfer rate of approximately 100 MB/second, in order to keep the seek time at 1% of the transfer time, it is necessary to operate on a block size of approximately 100MB. While the HDFS default block size is 64MB, many HDFS installations operate on a 128MB block size setup. In a Hadoop setup, the command *hadoop fsck -files -blocks* lists all the blocks that comprise each file in the file system.

An HDFS cluster encompasses 2 types of nodes (Name and DataNodes) that operate in a master worker (or master slave) relationship. In the HDFS design, the NameNode reflects the master, while the DataNodes represent the workers/slaves (see Figure 5). The NameNode manages the file system namespace, maintains the file system tree, as well as the metadata for all the files and directories in the tree. All this information is persistently stored on a local disk via 2 files that are labelled the namespace image and the edit log, respectively. The NameNode keeps track of all the DataNodes where the blocks for a given file are located. That information is dynamic (and hence is not persistently stored), as it is reconstructed every time the system starts up. Any client can access the file system on behalf of a user task by communicating with the NameNode and the DataNodes, respectively. The clients basically represent a POSIX alike file system interface, so that the user code's functionalities do not require any actual knowledge about the Hadoop Name and DataNodes. The DataNodes store and retrieve blocks based on requests made by the by clients or the NameNode, and they do periodically update the NameNode with lists of the actual blocks that they are responsible for. Without an active NameNode, the file system is considered non-functional. Hence, it is paramount to safeguard the NameNode by insuring that the node is resilient to any potential failure scenarios.

The Hadoop design allows configuring the NameNode in a way to backup the files that encompass the persistent state of the filesystem metadata to multiple file systems. These write operations are processed in a synchronous and atomic manner. It is rather common for HDFS setups to write the metadata to a local disk, as well as to 1 to *n* NFS mount points. Further, Hadoop HDFS provides the option to configure a secondary NameNode. The secondary NameNode does not act as a NameNode per se, but periodically merges the namespace image with the edit log to prevent the edit log from becoming too large. The secondary NameNode is configured on a separate physical

system, and it has to be pointed out that the merge activities are rather CPU intensive. The secondary NameNode also stores a copy of the merged namespace image, which can be utilized when the primary NameNode gets fenced. It is paramount to point out though that the state of the secondary NameNode lags behind the primary and hence, if the primary data is completely lost, the file system becomes non-functional. In such a scenario, it is common to copy the metadata files from one of the NFS mount points to the secondary NameNode and to run that server as the new primary node. As already discussed, the NameNode maintains the file system namespace. Any changes to the file system namespace (or the properties) is recorded. An application can specify the number of file replicas (labeled the replication factor) to be maintained by HDFS (the NameNode). Based on the HDFS design, a file is stored as a sequence of (same-sized) blocks. The only exception is the last file block that depending on the file size may allocate only a sub-size of a block (no internal fragmentation). The blocks of a file are replicated for fault tolerance purposes. The block size and the replication factor are configurable on a per file basis. The replication factor is determined at file creation time, and can be changed later on. The NameNode periodically receives a heartbeat and a block report from each DataNode in the cluster. The reception of a heartbeat implies that the DataNode is functional, while the block report lists all the blocks maintained by a particular DataNode.

Figure 5: HDFS Architecture

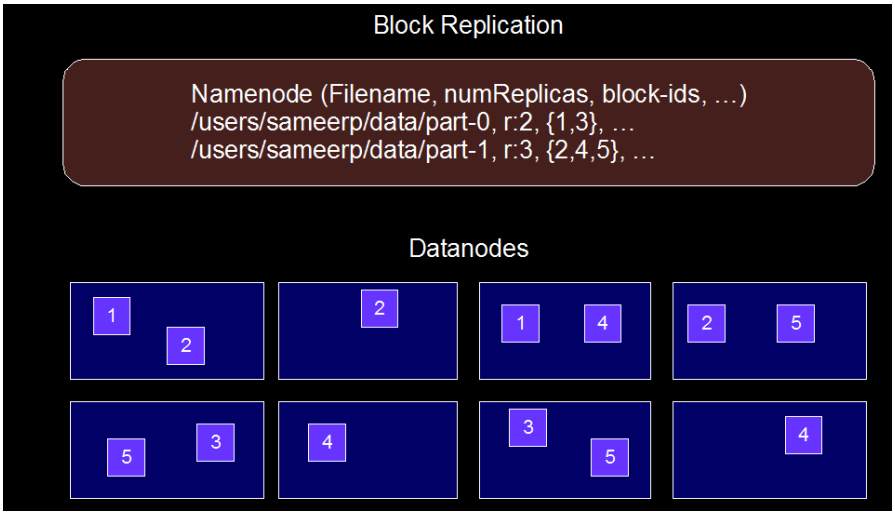



Note: Picture courtesy of Apache/Hadoop

The placement of the replicas is paramount to HDFS reliability and performance. Having the opportunity to optimize the replica placement scenarios is considered a feature that distinguishes HDFS from most of the other distributed file systems. It has to be pointed out though that the process of optimizing the replica placements requires tuning and experience. A rack-aware replica placement policy focuses on improving data reliability, availability, as well as to optimize network bandwidth utilization. Most HDFS installations execute in a cluster environment that encompasses many racks. Inter-rack node communication travels through switches. In most configurations, the bandwidth potential among nodes in the same rack is greater than the network bandwidth among nodes hosted in different racks. It has to be pointed out that the rack ID for the DataNodes can be manually defined and maintained. One policy option would be to place replicas on individual racks, preventing data loss scenarios when an entire rack fails, while allowing for aggregate bandwidth usage from multiple racks while reading data. Such a policy evenly distributes the replicas in the cluster, nicely focusing on load balancing and component failure scenarios. However, this policy increases the cost of write operations, as each write request requires transferring blocks to multiple racks.

For a common case that operates on a replication factor of 3, the HDFS placement policy is to store 1 replica on a node in the local rack, another replica on a different node in the same local rack, and the last replica on a different node in a different rack. This policy addresses the inter-rack write traffic scenario discussed above and generally improves the write performance. The chance of a rack failure is far less than that of a node failure and hence, this policy does not impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used when reading data, as a block is placed in only two unique racks rather than three. With this policy, the replicas of a file do not evenly distribute across the racks. 2/3 of the replicas are on one rack, and the other third is evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance. To minimize network bandwidth consumption and maximize read latency, HDFS' design objective is to process any read request from the data replica closest to the read task. Hence, if there is a replica on the same rack as the read task, that replica represents the preferred data location for the read request. In large Hadoop/HDFS environments that may span multiple data centers, the goal is to operate on a replica that is available in the local data center where the read task is issued.

Figure 6: HDFS Block Replication




Note: Picture courtesy of Apache/Hadoop

At startup, the NameNode enters a special (transient) state that is labeled the *safemode*. While in this state, the NameNode accepts heartbeat and block report messages from the DataNodes. As already discussed, the block report lists the data blocks that the DataNodes are hosting. Each data block is linked to a specific (minimum) number of replicas. A data block is considered fully replicated when the minimum number of replicas for that particular data block has been confirmed with the NameNode. After a certain percentage (this is tunable) of fully replicated data blocks is reached, the NameNode stalls for another 30 seconds, and afterwards exits the safemode state. The next task for the NameNode is to compile a list of data blocks (if any) that still have fewer than the specified number of replicas available, and to replicate these data blocks unto some other DataNodes. The NameNode retains a transaction log (labeled the *editlog*) to persistently record every change made to file system metadata. To illustrate, creating a new file or changing the replication factor for a file triggers a record insert operation into the editlog. The editlog file itself is stored in the NameNode's local host file system. The HDFS namespace is maintained by the NameNode, as the entire namespace, including the data block file mapping and file system properties are stored in the HDFS *fsimage* file. Similar to the editlog file, the fsimage file is located in the NameNode's local file system. The NameNode retains an image of the file system namespace and the block map file in memory. As the NameNode boots up, the fsimage and editlog files are read from disk and all the (potential) editlog transactions are mapped into the fsimage in-memory copy and then flushed out to a new version of the fsimage file. At that point, the old editlog file is truncated, as the transactions have

been securely applied to the persistent fsimage file (this process is labeled as a checkpoint). The fsimage and the editlog files are considered core HDFS data structures. A corruption of these files can cause the HDFS instance to become non-functional. Hence, a NameNode should be configured so that multiple copies of the 2 files are being maintained. In other words, any update to either the fsimage or editlog triggers a synchronous update to multiple copies of the 2 files. As most Hadoop/HDFS applications are not metadata intensive, maintaining several copies of the 2 files is not considered a significant performance issue. The HDFS communication protocols are stacked on top of TCP/IP. A client system basically establishes a TCP connection (via a configurable port) to the NameNode, utilizing the *clientprotocol* as the communication entity. The DataNodes communicate with the NameNode via the DataNode protocol. A remote procedure call (RPC) *wrapper* encompasses the client, as well as the DataNode protocols, respectively. By design, the NameNode never initiates any RPC requests. Instead, the NameNode only responds to RPC requests that are issued by either the DataNodes or the clients.

It has to be pointed out that a client request to create a file does not reach the NameNode instantaneously, as the HDFS client caches the file data in a temporary local file. Any application write requests are transparently redirected to this temporary local file until the size of the local file reaches the HDFS block size. At that point, the client contacts the NameNode which adds the file name into the file system hierarchy and allocates an HDFS data block. Next, the NameNode responds to the client request by identifying a DataNode and a target data block. After that, the client flushes the data block from the temporary local file to the specified DataNode. When a file is closed, any (potentially) un-flushed data being held in the temporary local file is transferred to the DataNode as well. After the *close()* operation, the client informs the NameNode that the file has been closed. That indicates to the NameNode to commit the file creation operation into persistent storage. This client-side caching architecture (which is similar to techniques used by other distributed file systems) was chosen to minimize network congestion and to optimize network throughput. To illustrate, based on the discussed write example, the assumption made is that the HDFS file is configured with a replication factor of 3. As the local file accumulated enough data for an entire block, the client retrieves a list from the NameNode that outlines the DataNodes that will host a replica block. Based on the list, the client starts transferring data to the first DataNode. The first DataNode receives the data in small chunks (normally 4KB blocks), writes each portion of the data block to a local repository, and then transfers that portion of the data to the second DataNode on the list. The second DataNode operates in the same way, and hence transfers its data chunks to the 3d DataNode on the list. Ultimately, the 3d DataNode receives the data and writes the data chunks into its local repository. In other words, depending on the order, a DataNode may simultaneously be receiving data from a previous node while sending data to the next DataNode in the pipeline (concurrent daisy-chain alike fashion).

**The Hadoop Schedulers**

Since the pluggable scheduler framework (similar to the Linux IO schedulers) was introduced, several different scheduler algorithms have been designed, developed, and made available to the Hadoop community. In the next few paragraphs, the *FIFO*, the *Fair*, as well as the *Capacity* schedulers are briefly introduced.

- The FIFO Scheduler -> FIFO reflects the original Hadoop scheduling algorithm that was integrated into the JobTracker framework. With FIFO scheduling, a JobTracker basically just pulls the oldest job from the work queue. The FIFO scheduling approach has no concept of either job priority or job size, but is rather simple to implement and efficient to execute (very low overhead).

- The Fair Scheduler -> Originally, the Fair scheduler was developed by *Facebook*. The fundamental design objective for the Fair scheduler revolves around the idea to assign resources to jobs in a way that (on average) over time, each job receives an equal share of the available resources. With the Fair scheduler, there is a certain degree of interactivity among Hadoop jobs, permitting a Hadoop cluster to better response to the variety of job types that are submitted over time. From an implementation perspective, a set of pools is setup, and the jobs are placed into these pools and so are made available for selection by the scheduler. Each pool operates on (assigned) shares to balance the resource usage among

the jobs in the pools. The heuristic used is that the more shares the greater the resource usage potential to execute the jobs. By default, all pools are setup with equal shares, but configuration based pool share adjustments can be made based on job types. The number of concurrent active jobs can be constrained to minimize congestion and to allow the workload to be processed in a timely manner. To ensure fairness, each user is assigned to a pool. Regardless of the shares that are assigned to the pools, if the system is underutilized (based on the current workload), the active jobs receive the unused shares (the shares are split among the current jobs). For each job, the scheduler keeps track of the compute time. Periodically, the scheduler examines the jobs to calculate the delta between the actual compute time received and the compute time that the job *should* have received. The results reflect the deficit matrix for the tasks. It is the scheduler's responsibility to schedule the task with the greatest deficit.

- The Capacity Scheduler -> Originally, the Capacity scheduler was developed by *Yahoo.* The design focus for the Capacity scheduler was on large cluster environments that execute many independent applications. Hence, the Capacity scheduler provides the ability to provide a minimum capacity guarantee, as well as to share excess capacity among the users. The Capacity scheduler operates on queues. Each queue can be configured with a certain number of map and reduce slots. Further, each queue can be assigned a guaranteed capacity while the overall capacity of the cluster equals to the sum of all the individual queue capacity values. All the queues are actively monitored and in scenarios where a queue is not consuming its allocated capacity potential, the excess capacity can be (temporarily) allocated to other queues. Compared to the Fair scheduler, the Capacity scheduler controls all the prioritizing tasks within a queue. In general, higher priority jobs are allowed access to the cluster resources earlier than lower priority jobs. With the Capacity scheduler, queue properties can be adjusted on-the-fly, and hence do not require any disruptions in the cluster usage/processing.

While not considered a scheduler per se, Hadoop also supports the scheme of provisioning virtual cluster environments (within physical clusters). This concept is labeled *Hadoop On Demand* (HOD). The HOD approach utilizes the Torque resource manager for node allocation to *size* the virtual cluster. Within the virtual environment, the HOD system prepares the configuration files in an automated manner, and initializes the system based on the nodes that comprise the virtual cluster. Once initialized, the HOD virtual cluster can be used in a rather independent manner. A certain level of elasticity is build into HOD, as the system adapts to changing workload conditions. To illustrate, HOD automatically de-allocates nodes from the virtual cluster after detecting no active jobs over a certain time period. This shrinking behavior allows for the most efficient usage of the overall physical cluster assets. HOD is considered as a valuable option for deploying Hadoop clusters within a cloud infrastructure.

**MapReduce Performance Evaluation - NN Algorithm**

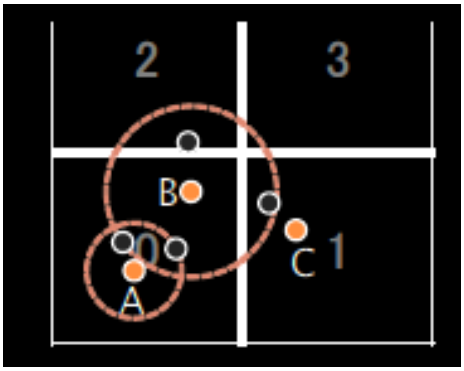
A Nearest Neighbor (NN) analysis represents a method to classify cases based on their similarity to other cases. In machine learning, NN algorithms are utilized to recognize data patterns without requiring an exact match to any of the stored cases. Similar cases are in close proximity while dissimilar cases are more distant from each other. Hence, the distance between 2 cases is used as the measure of (dis)similarity. Cases that are in close proximity are labeled *neighbors*. As a new case is presented, its distance from each of the cases in the model is computed. The classifications of the most similar cases (aka the nearest neighbors) are tallied, and the new case is placed into the category that contains the greatest number of nearest neighbors (the number of nearest neighbors to examine is normally labeled as *k*).

Hence, a *k*-nearest neighbor (*k*NN) analysis retrieves the *k nearest* points from a dataset. A *k*NN query is considered as one of the fundamental query types for spatial databases. A spatial database reflects a database solution that is optimized to store/query data that is related to objects in space (including points, lines, and polygons). While most database solutions do understand the concept of various numeric and character data types, additional functionality is required for

database systems to process spatial data types (typically called the geometry). An *all k-nearest neighbor* (a*k*NN) analysis depicts a variation of a *k*NN based study, as a *ak*NN query determines the *k*-nearest neighbors for each point in the dataset. A akNN based approach is extensively being used for batch-based processing of large, distributed (point) datasets. To illustrate, location-based services that identify for each user their nearby counterparts (or new friends) are good candidates for a *ak*NN based approach. Given that the user locations are maintained by the underlying database systems, the resulting recommendation lists can be generated by issuing an *ak*NN query against the database. Further, *ak*NN queries are often employed to preprocess data for subsequent data mining purposes.

For this study, the goal is to quantify the performance behavior of processing a*k*NN queries in a *Hadoop* environment. The methodology used to execute the queries in a MapReduce framework closely follows the guidelines discussed in Yokoyama [1], Afrati [2], and Vernica [3]. As extensively discussed in Yokoyama, it is feasible to decompose the given space into cells, and to execute a *ak*NN query via the MapReduce framework in a distributed, parallel manner. Compared to some of the other work on kNN solutions in distributed environments, this study proposes a *kd-tree* [4] based approach that utilizes a variable number of neighboring cells for the cell merging process discussed below.

Figure 7: *ak*NN Processing (*k* = 2, Cell Based)




To briefly illustrate the methodology, 2-dimensional points with *x* and *y* axes are considered. The target space is decomposed into *2^n x 2^n* small cells, where the constant *n* determines the granularity of the decomposition. As the *k*-nearest neighbor points for a data point are normally located in close proximity, the assumption made is that most of the *k*NN objects are located in the nearby cells. Hence, the approach is based on classifying the data points into the corresponding cells and to compute candidate *k*NN points for each point. This process can easily be parallelized, and hence is suited for a MapReduce framework [1]. It has to be pointed out though that the approach may not be able to determine the *k*NN points in a single processing cycle and therefore, additional processing steps may be necessary (the crux of the issue is that data points in other nearby cells may belong to the *k*-nearest neighbors). To illustrate this problem Figure 7 depicts a *ak*NN processing scenario for *k* = 2. Figure 7 outlines that the query can locate 2 NN points for A by just considering the inside (boundary) of cell 0. In other words, the circle centered around A already covers the target 2 NN objects without having to go beyond the boundary of cell 0.

On the other hand, the circle for B overlaps with cells 1, 2, and 3, respectively. In such a scenario, there is a possibility to locate the 2 NN objects in 4 different cells. Ergo, in some circumstances, it may not be feasible to draw the circle for a point just based on the cell boundary. For point C, there is only 1 point available in cell 1, and hence cell 1 violates the *k = 2* requirement. Therefore, it is a necessary design choice for this study to prohibit scenarios where cells contain less than *k* points. This is accomplished by 1st identifying the number of points within each cell, and 2nd by merging a cell with < *k* points with a neighboring cell to assure that the number of points in the merged cell is >= *k*. At that point, the boundary circle can be drawn. The challenge with this approach is though that an additional counting cycle prior to the NN computation is required. The benefit of the approach is that during the counting phase, cells with no data points can be identified and hence, can be eliminated from the actual processing cycle. The entire processing cycle encompasses 4 steps

that are discussed below. The input dataset reflects a set of records that are formatted in a [id, x, y] fashion (the parameters *n* and *k* are initially specified by the user).

*MapReduce S1 Step: Acquiring Distribution and Cell Merging Information.* In this initial step, the entire space is decomposed into 2^*n* x 2^*n* cells and the number of points per cell is determined. The actual counting procedure is implemented as a MapReduce process. The *Map* function receives the formatted input and computes the cell number for each data point based on its coordinates. The format of the output record is a key-value pair [cell_id, data_point]. The *Reduce* function receives a set of records (with the same cell ids) and sums the value parts of the records. The output reflects a format [cell_id, sum].

To improve performance in Hadoop, an optional *Combiner* in the MapReduce process can be used. A Combiner is used to aggregate intermediate data within a Map task, a process that may reduce the size of the intermediate dataset. After the MapReduce S1 phase, cell merging is performed. For this study, a *kd-tree* tree structure [4] was implemented to merge neighboring cells. In a nutshell, the cell merging function generates the mapping information (how each cell id is matched up with a potentially new cell id in the merged cell decomposition scheme) that is used as additional input into the MapReduce S2 step.

*MapReduce S2 Step: akNN Computation.* In the 2nd step, the input records for each cell are collected and candidate *k*NN points (for each point in the cell region) are computed. The Map function receives the original data points and computes the corresponding cell id. The output records are formatted as [cell_id, id, coordinates], where the id represents the point id, and the coordinates reflect the actual coordinates of the point. The reduce function receives the records corresponding to a cell, formatted as [id, coordinates]. The Reduce function calculates the distance for each 2-point combo and computes the *k*NN points for each point in the cell. The output records are formatted as [id, coordinates, cell_id, *k*NN_list], where the id is used as the key, and the *k*NN_list reflects the list of the *k*NN points for point id (formatted as [*ho1; d1i;.... ; hoi; dki*], where *hoi represents* the *i*-th NN point and *dki* the corresponding distance.

*MapReduce3 S3 Step: Update k-NN Points.* In the 3d step, the actual boundary circles are determined. In this step, depending on the data set, additional *k*-NN point processing scenarios may apply. The Map function basically receives the result of the MapReduce S2 step and for each point, the boundary circle is computed. Two different scenarios are possible here:

1. *The boundary circle does not overlap with other cells*: In this scenario, no further processing is required, and the output (key value pairs) is formatted as [cell_id, id, coordinates, kNN_list, true]. The cell_id reflects the key while the Boolean *true* denotes the fact that the processing cycle is completed. In other words, in the corresponding Reduce function, no further processing is necessary and only the record format is converted.
2. *The boundary circle overlaps with other cells*: In this scenario, there is a possibility that neighboring cells may contain *k*NN points, and hence an additional (check) process is required. To accomplish that, the [key, value] pairs are formatted as [cell_idx, id, coordinates, kNN_list, false]. The key cell_idx reflects the cell id of one of the overlapped cells while the Boolean *false* indicates a non-completed process. In scenarios where *n* cells overlap, *n* corresponding [key, value] pairs are used.

The Shuffle operation submits records with the same cell ids to the corresponding node, and the records are used as input into the Reduce function. As different types of records may exist as input, it is necessary to first classify the records and second to update the *k*NN points in scenarios where additional checks are required. The output records are formatted as [id, coordinates, cell_id, kNN_list], where the id field represents the key.

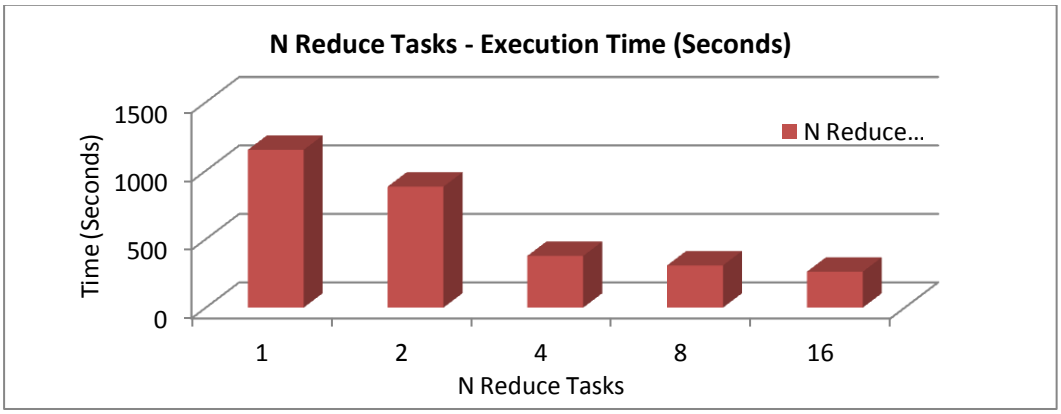
*MapReduce S4 Step*: This final processing step may be necessary as multiple update scenarios of *k*NN points per point are possible. In this step, if necessary, these *k*NN lists are fused, and a final *k*NN (result) list is generated. The Map function receives the results of the S3 step and outputs the *k*NN list. The output format of the records is [id, kNN_list], where the id represents the key. The Reduce function receives the records with the same keys and generates the integrated list

formatted as [id, kNN_list]. If multiple cells per point scenarios are possible, the Shuffle process clusters the *n* outputs and submits the data to the Reduce function. Ultimately, the *k*NN points per reference point are determined, and the processing cycle completes.

**Hadoop akNN MapReduce Benchmarks**

For this study, several large-dataset benchmark studies were conducted, focusing on Hadoop MapReduce performance for solving *akNN* related data projects. The Hadoop (1.0.2) cluster used for the study consisted of 12 Ubuntu 12.04 server nodes that were all equipped with Intel Xeon 3060 processors (12GB RAM, 4M Cache, 2.40GHz, Dual Core). Each node was configured with 4 1TB hard disks in a JBOD (Just a Bunch Of Disks) setup. The interconnect consisted of a 10Gbit (switched) network. The benchmark data set consisted of 4,000,000, 8,000,000, 16,000,000, and 32,000,000 reference points, respectively. For this study, the Hadoop replica factor was set to 1, while the HDFS block size was left at the default value of 64MB. The Hadoop cluster utilized the FIFO scheduler. The (above discussed) granularity *n* was set to 8. In this report, the results for the 8,000,000 reference point dataset runs are elaborated on. For the 1st set of benchmarks, the number of Reduce tasks was scaled from 1 to 16, while *k* was set to 4, and the number of Map tasks was held constant at 8 (see Figure 8).

Figure 8: *akNN* MapReduce Performance - Varying Number of Reduce Tasks



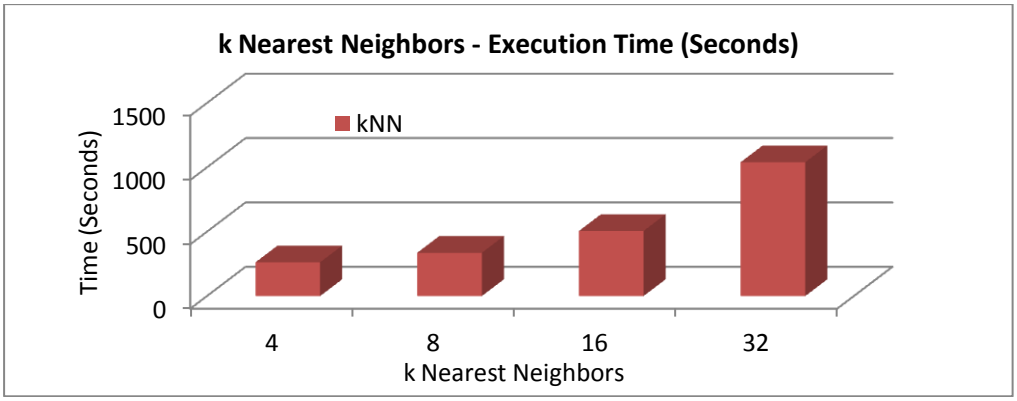

Note: The results depicted in Figure 2 reflect average execution time over 25 test runs per benchmark set. For all the benchmark sets, the CV determined over all the individual test runs was less than 3%.

As Figure 8 outlines, the average total execution time decreases as the number of Reduce tasks is increased (total execution time equals to the sum of the steps S1, S2, and if necessary, S3, and S4). With 16 Reduce tasks, the 8,000,000 data points are processed in approximately 262 seconds on the Hadoop cluster described above. The largest performance increase (delta of approximately 505 seconds) was measured while increasing the number of Reduce tasks from 2 to 4. The study showed that with the data set at hand, only approximately 29.3% of the data points require executing the MapReduce S3 step discussed above. In other words, due to the rather high data point density in the target space (8,000,000 reference points), over 70% of the data points only require executing steps S1 and S2, respectively. Further, the study revealed that while increasing the number of worker threads, the execution time for step S1 is increased. This is due to the fact that the input size per Reducer task diminishes, and as the data processing cost is rather low, increasing the number of Reducer tasks actually adds overhead into the aggregate processing cycle.

For the 2nd set of benchmark runs, the number of Map and Reduce tasks was set to 8 and 24, respectively, while the number of *k* Nearest Neighbors was scaled from 4 to 32. As Figure 9

depicts, the average total execution time increases as *k* is scaled up. The study disclosed that while scaling *k*, the processing cost for the MapReduce steps S3 and S4 increases significantly. The cost increase is mainly due to the increase of (larger size) intermediate records that have to be processed, as well as due to the increased number of data points that require steps S3 and S4 to be processed. To illustrate, increasing *k* from 16 to 32 resulted in an average record size increase of approximately 416 bytes, while at the same time an additional 10.2% of the data points required executing steps S3 and S4. From an average total execution time perspective, the delta between the *k*=16 and *k*=32 was approximately 534 seconds, while the delta for the *k*=8 and *k*=16 runs was only approximately 169 seconds. To summarize, while processing the *akNN* MapReduce framework in parallel, it is possible to reduce the total execution time rather considerably (see Figure 8).

Figure 9: *akNN* MapReduce Performance - Varying Number of k Nearest Neighbors




Note: The results depicted in Figure 3 reflect average execution time over 25 test runs per benchmark set. For all the benchmark sets, the CV determined over all the individual test runs was less than 3%.

## Summary & Conclusions

The additional studies executed on the Hadoop cluster showed that a rather large data set (high space density) is required for the discussed *ak*NN methodology to operate efficiently and effectively. Scaling *k* in the benchmark sets disclosed a rather substantial processing cost increase that is mainly due to executing additional, larger MapReduce records in the S3 and S4 processing cycles. In a nutshell though, Hadoop's MapReduce and HDFS infrastructure provides straightforward and robust techniques that operate on commodity (inexpensive) HW, and does provide high data availability features to the user community. For (very) large, unstructured datasets, Hadoop provides the IT performance infrastructure necessary to deliver results in a timely fashion.

## References

1. *Takuya Yokoyama, Yoshiharu Ishikawa, Yu Suzuki, "Processing All k-Nearest Neighbor Queries in Hadoop" Nagoya University, Japan, 2011*
2. *F. N. Afrati, J. D. Ullman, "Optimizing joins in a map-reduce environment", EDBT, 2010.*
3. *R. Vernica, M. J. Carey, C. Li, "Efficient parallel set-similarity joins using MapReduce", SIGMOD, 2010*
4. *Bruce F. Naylor, "A Tutorial on Binary Space Partitioning Trees", Spatial Labs, 2010*
5. *White Tom, "Hadoop: The Definitive Guide", 3rd Edition, Storage and Analysis at Internet Scale, O'Reilly Media, May 2012*
6. *Heger, D., "Quantifying IT Stability - 2nd Edition, Fundcraft Publication, 2010*
7. *K. Shvachko, H. Kuang, S. Radia, R. Chansler, "The Hadoop distributed file system", In Proc. of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST), 2010.*
8. *Hadoop: http://hadoop.apache.org/*
9. *HDFS: http://hadoop.apache.org/hdfs/*
10. *Hbase: http://hbase.apache.org/*

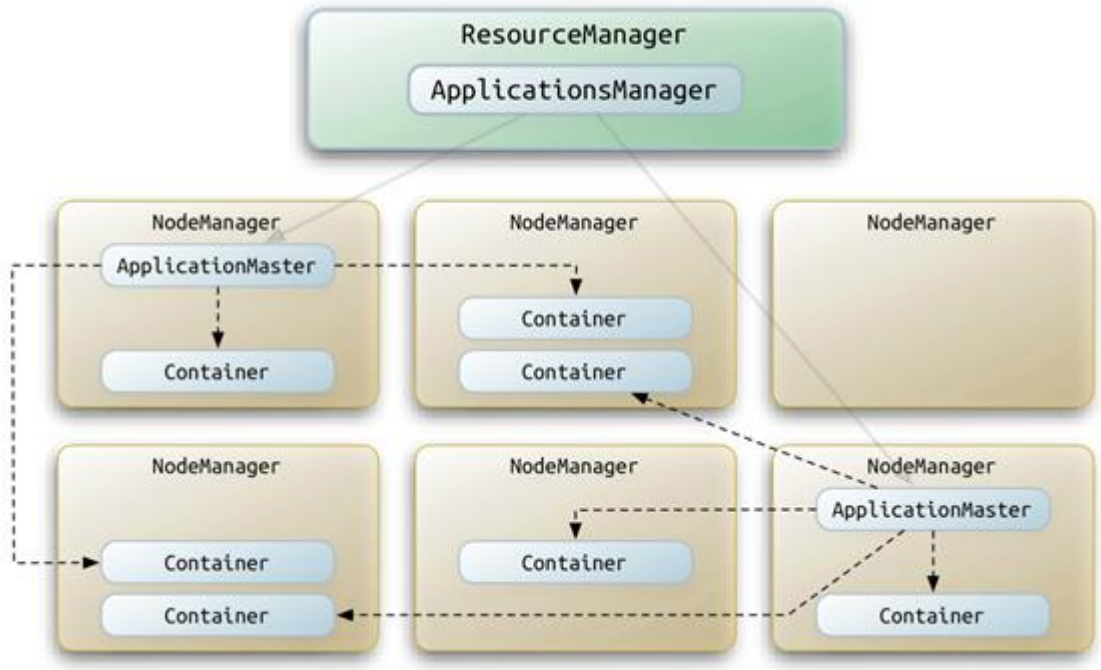
**Appendix A: The MapReduce MRv2 (YARN) Framework & HDFS NameNode Considerations**

As of MapReduce 2.0 (MRv2), the MapReduce framework has been completely redesigned (see Figure A1). The essential idea of MRv2's YARN (Yet Another Resource Negotiator) architecture is to decompose the 2 principal responsibilities of the old JobTracker into a (1) resource management and (2) a job scheduling/monitoring entity. The new design is based on a global ResourceManager (RM) and a per-application ApplicationMaster (AM) daemon. Starting with MRv2, the RM and the per-node NodeManager (NM) reflect the new data-computation framework. The RM replaces the functionalities that used to be provided by the JobTracker and represents the ultimate resource arbitration authority (among all the applications). The AM (a framework specific library) negotiates any resource specific tasks with the RM and operates in conjunction with the NM to execute and monitor the tasks.

The RM consists of 2 components, the scheduler and the ApplicationsManager (AppMg). The scheduler (such as Capacity or Fair) is responsible for allocating resources to the various running applications. The scheduler only performs scheduling tasks and is not involved in monitoring or tracking the application status. The scheduler performs the scheduling functions based on the resource requirements of the applications (via the abstract notion of a resource Container that incorporates resource usage elements such as CPU, memory, disk, or network. The AppMg is responsible for accepting job submissions, negotiating the first container for execution of the application specific AM and provides the services for restarting the AM container in failure situations. The NM represents the per system framework agent responsible for the containers, monitoring their resource usage (cpu, memory, disk, network) and reporting to the RM (scheduler). The per application AM is in charge of negotiating the appropriate resource containers (in conjunction with the scheduler), tracking their status and monitoring the progress.

MRV2 maintains API compatibility with previous stable releases (hadoop-0.20.205). This implies that all Map-Reduce jobs should still run unchanged on top of MRv2 with just a recompile.

Figure A1: MRv2 Framework




Note: Figure courtesy of Apache

One significant issue with the original HDFS design is that the NameNode basically reflects a single point of failure. If the NameNode service or the server hosting the NameNode service is unavailable, HDFS is down for the entire cluster. The Secondary NameNode discussed in this paper (that periodically snapshots the NameNode) cannot be considered an actual HA solution.

To address the issue, several options are available today. MapR, one of the major Hadoop distributors, designed/developed a distributed NameNode architecture where the HDFS metadata is scattered across the cluster (in what is labeled as containers). The concept of the NameNode is basically eliminated in the design via a distributed metadata architecture. An actual Container Location Database (CLDB) is used in the MapR design to facilitate the NameNode functionalities.

Starting with Hadoop 2.0.2, HDFS incorporates a HDFS High Availability (HA) feature that provides the option of configuring 2 redundant NameNodes in the same cluster in an active/passive setup. The 2 nodes are referred to as the Active NameNode and the Standby NameNode. Unlike the Secondary NameNode already discussed in this paper, the Standby NameNode is a hot standby, allowing for fast failover scenarios to a new NameNode in the case that the node becomes unavailable or a graceful administrator-initiated failover for maintenance purposes is required. In a nutshell, the Active NameNode logs all changes to a directory that is accessible by the Standby NameNode and is used to keep the passive node up-to-date.