Problem Scope

Hadoop is a large-scale distributed batch processing infrastructure. While it can be used on a single machine, its true power lies in its ability to scale to hundreds or thousands of computers, each with several processor cores. Hadoop is also designed to efficiently distribute large amounts of work across a set of machines.

How large an amount of work? Orders of magnitude larger than many existing systems work with. Hundreds of gigabytes of data constitute the *low end* of Hadoop-scale. Actually Hadoop is built to process "web-scale" data on the order of hundreds of gigabytes to terabytes or petabytes. At this scale, it is likely that the input data set will not even fit on a single computer's hard drive, much less in memory. So Hadoop includes a distributed file system which breaks up input data and sends fractions of the original data to several machines in your cluster to hold. This results in the problem being processed in parallel using all of the machines in the cluster and computes output results as efficiently as possible.

Processor capabilities (Pentium to Multi core)

A **computer cluster** is a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system. Unlike grid computers, computer clusters have each node set to perform the same task, controlled and scheduled by software.

Grid computing is the use of widely distributed computer resources to reach a common goal. A computing grid can be thought of as a distributed system with non-interactive workloads that involve many files. Grid computing is distinguished from conventional high-performance computing systems such as cluster computing in that grid computers have each node set to perform a different task/application. Grid computers also tend to be more heterogeneous and geographically dispersed (thus not physically coupled) than cluster computers.

CHALLENGES AT LARGE SCALE

Performing large-scale computation is difficult. To work with this volume of data requires distributing parts of the problem to multiple machines to handle in parallel. Whenever multiple machines are used in cooperation with one another, the probability of failures rises. In a single-machine environment, failure is not something that program designers explicitly worry about very often: if the machine has crashed, then there is no way for the program to recover anyway.

In a distributed environment, however, partial failures are an expected and common occurrence. Networks can experience partial or total failure if switches and routers break down. Data may not arrive at a

particular point in time due to unexpected network congestion. Individual compute nodes may overheat, crash, experience hard drive failures, or run out of memory or disk space. Data may be corrupted, or maliciously or improperly transmitted. Multiple implementations or versions of client software may speak slightly different protocols from one another. Clocks may become desynchronized, lock files may not be released, parties involved in distributed atomic transactions may lose their network connections part-way through, etc. In each of these cases, the rest of the distributed system should be able to recover from the component failure or transient error condition and continue to make progress. Of course, actually providing such resilience is a major software engineering challenge.

Different distributed systems specifically address certain modes of failure, while worrying less about others. Hadoop provides no security model, nor safeguards against maliciously inserted data. For example, it cannot detect a man-in-the-middle attack between nodes. On the other hand, it is designed to handle hardware failure and data congestion issues very robustly. Other distributed systems make different trade-offs, as they intend to be used for problems with other requirements (e.g., high security).

In addition to worrying about these sorts of bugs and challenges, there is also the fact that the computer hardware has finite resources available to it. The major resources include:

- **Processor time**

- **Memory**
- **Hard drive space**
- **Network bandwidth**

Individual machines typically only have a few gigabytes of memory. If the input data set is several terabytes, then this would require a thousand or more machines to hold it in RAM -- and even then, no single machine would be able to process or address all of the data.

Hard drives are much larger; a single machine can now hold multiple terabytes of information on its hard drives. But intermediate data sets generated while performing a large-scale computation can easily fill up several times more space than what the original input data set had occupied. During this process, some of the hard drives employed by the system may become full, and the distributed system may need to route this data to other nodes which can store the overflow.

Finally, **bandwidth is a scarce resource** even on an internal network. While a set of nodes directly connected by a gigabit Ethernet may generally experience high throughput between them, if all of the machines were transmitting multi-gigabyte data sets, they can easily saturate the switch's bandwidth capacity. Additionally if the machines are spread across multiple racks, the bandwidth available for the data transfer would be much less. Furthermore RPC requests and other data transfer requests using this channel may be delayed or dropped.

To be successful, a large-scale distributed system must be able to manage the above mentioned resources efficiently. Furthermore, it must

allocate some of these resources toward maintaining the system as a whole, while devoting as much time as possible to the actual core computation.

Synchronization between multiple machines remains the biggest challenge in distributed system design. If nodes in a distributed system can explicitly communicate with one another, then application designers must be cognizant of risks associated with such communication patterns. It becomes very easy to generate more remote procedure calls (RPCs) than the system can satisfy! **Performing multi-party data exchanges is also prone to deadlock or race conditions. Finally, the ability to continue computation in the face of failures becomes more challenging. For example, if 100 nodes are present in a system and one of them crashes, the other 99 nodes should be able to continue the computation, ideally with only a small penalty proportionate to the loss of 1% of the computing power. Of course, this will require re-computing any work lost on the unavailable node. Furthermore, if a complex communication network is overlaid on the distributed infrastructure, then determining how best to restart the lost computation and propagating this information about the change in network topology may be non trivial to implement.**

MOORE'S LAW

So why use a distributed system at all? They seem like more trouble than they're worth. And with the fast pace of computer hardware design, it seems inevitable that single-chip hardware will be able to "grow up" to handle the larger volumes of data. After all, Moore's Law (named after Gordon Moore, the founder of Intel) states that**the number of transistors that can be placed in a processor will double approximately every two years, for half the cost.** But trends in chip design are changing to face new realities. While we can still double the number of transistors per unit area at this pace, this does not necessarily result in faster single-threaded performance. New processors such as Intel Core 2 and Itanium 2 architectures now focus on embedding **many smaller CPUs** or "cores" onto the same physical device. This allows multiple threads to process twice as much data in parallel, but at the same speed at which they operated previously.

Even if hundreds or thousands of CPU cores are placed on a single machine, it would not be possible to deliver input data to these cores fast enough for processing. Individual hard drives can only sustain read speeds between 60-100 MB/second. These speeds have been increasing over time, but not at the same breakneck pace as processors. Optimistically assuming the upper limit of 100 MB/second, and assuming four independent I/O channels are available to the machine, that provides 400 MB of data every second. A 4 terabyte data set would thus take over 10,000 seconds to read--about three hours just to load the data! With 100 separate machines each with two I/O channels on the job, this drops to three minutes.

The Hadoop Approach

Hadoop is designed to efficiently process large volumes of information by connecting many commodity computers together to work in parallel. The theoretical 1000-CPU machine described earlier would cost a very large amount of money, far more than 1,000 single-CPU or 250 quad-core machines. Hadoop will tie these smaller and more reasonably priced machines together into a single cost-effective compute cluster.

COMPARISON TO EXISTING TECHNIQUES

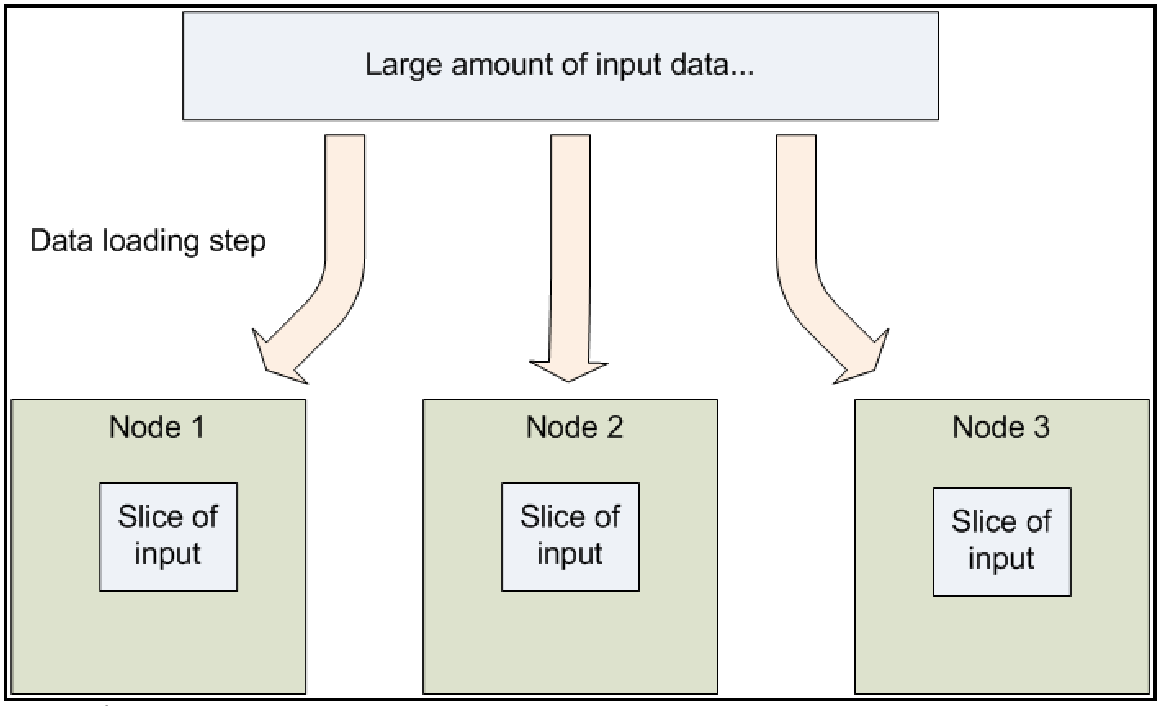
Performing computation on large volumes of data has been done before, usually in a distributed setting. What makes Hadoop unique is its **simplified programming model** which allows the user to quickly write and test distributed systems, and its **efficient, automatic distribution of data and work across machines** and in turn utilizing the underlying parallelism of the CPU cores.

Grid scheduling of computers can be done with existing systems such as Condor. But Condor does not automatically distribute data: a separate SAN must be managed in addition to the compute cluster. Furthermore, collaboration between multiple compute nodes must be managed with a communication system such as MPI (Message Passing Interface). This programming model is challenging to work with and can lead to the introduction of subtle errors.

DATA DISTRIBUTION

In a Hadoop cluster, data is distributed to all the nodes of the cluster as it is being loaded in. The Hadoop Distributed File System (HDFS) will split large data files into chunks which are managed by different nodes in the cluster. In addition to this each chunk is replicated across several machines, so that a single machine failure does not result in any data being unavailable. An active monitoring system then re-replicates the data in response to system failures which can result in partial storage. Even though the file chunks are replicated and distributed across several machines, they form a single namespace, so their contents are universally accessible.

Data is conceptually **record-oriented** in the Hadoop programming framework. Individual input files are broken into lines or into other formats specific to the application logic. Each process running on a node in the cluster then processes a subset of these records. The Hadoop framework then schedules these processes in proximity to the location of data/records using knowledge from the distributed file system. Since files are spread across the distributed file system as chunks, each compute process running on a node operates on a subset of the data. Which data operated on by a node is chosen based on its locality to the node: most data is read from the local disk straight into the CPU, alleviating strain on network bandwidth and preventing unnecessary network transfers. This strategy of **moving computation to the data**,



instead of moving the data to the computation allows Hadoop to achieve high data locality which in turn results in high performance.

Figure 1.1: Data is distributed across nodes at load time.

MAPREDUCE: ISOLATED PROCESSES

Hadoop limits the amount of communication which can be performed by the processes, as each individual record is processed by a task in isolation from one another. While this sounds like a major limitation at first, it makes the whole framework much more reliable. Hadoop will not run just any program and distribute it across a cluster. **Programs must be written to conform to a particular programming model, named "MapReduce."**

In MapReduce, records are processed in isolation by tasks called *Mappers*. The output from the Mappers is then brought together into a second set of tasks called *Reducers*, where results from different mappers can be merged together.

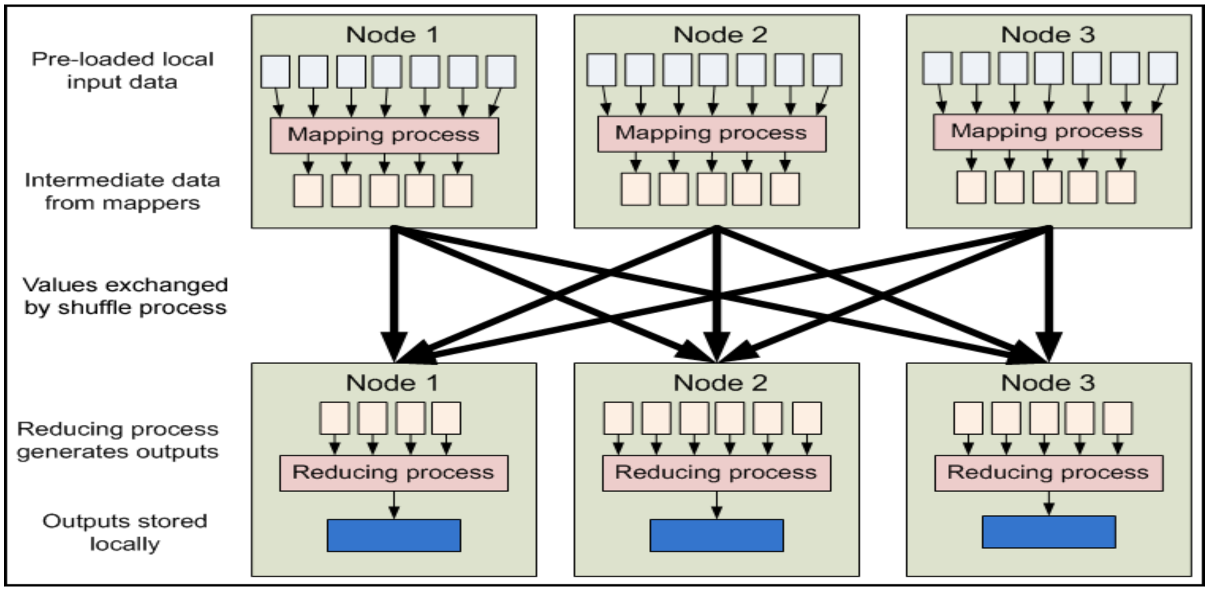


Figure 1.2: Mapping and reducing tasks run on nodes where individual records of data are already present.

Separate nodes in a Hadoop cluster still communicate with one another. However, in contrast to more conventional distributed systems where application developers explicitly marshal byte streams from node to node over sockets or through MPI buffers, communication in Hadoop is performed *implicitly*. Pieces of data can be tagged with key names which inform Hadoop how to send related bits of information to a common destination node. Hadoop internally manages all of the data transfer and cluster topology issues.

By restricting the communication between nodes, Hadoop makes the distributed system much more reliable. Individual node failures can be worked around by restarting tasks on other machines. Since user-level tasks do not communicate explicitly with one another, no messages need to be exchanged by user programs, nor do nodes need to roll back to pre-arranged checkpoints to partially restart the computation. The other workers continue to operate as though nothing went wrong, leaving the challenging aspects of partially restarting the program to the underlying Hadoop layer.

FLAT SCALABILITY

One of the major benefits of using Hadoop in contrast to other distributed systems is its flat scalability curve. Executing Hadoop on a limited amount of data on a small number of nodes may not demonstrate particularly stellar performance as the overhead involved in starting Hadoop programs is relatively high. Other parallel/distributed programming paradigms such as MPI (Message Passing Interface) may perform much better on two, four, or perhaps a dozen machines. Though the effort of coordinating work among a small number of machines may be better-performed by such systems, the price paid in performance and engineering effort (when adding more hardware as a result of increasing data volumes) increases non-linearly.

A program written in distributed frameworks other than Hadoop may require large amounts of refactoring when scaling from ten to one hundred or one thousand machines. This may involve having the program be rewritten several times; fundamental elements of its design may also put an upper bound on the scale to which the application can grow.

Hadoop, however, is specifically designed to have a very flat scalability curve. After a Hadoop program is written and functioning on ten nodes, very little--if any--work is required for that same program to run on a much larger amount of hardware. Orders of magnitude of growth can be managed with little re-work required for your applications. The underlying Hadoop platform will manage the data and hardware resources and provide dependable performance growth proportionate to the number of machines available.