MapReduce: Simplified Data Processing on Large Clusters

Ahmed Elbagoury
Dr.Iman Elghandour
Alexandria Univeristy

Agenda

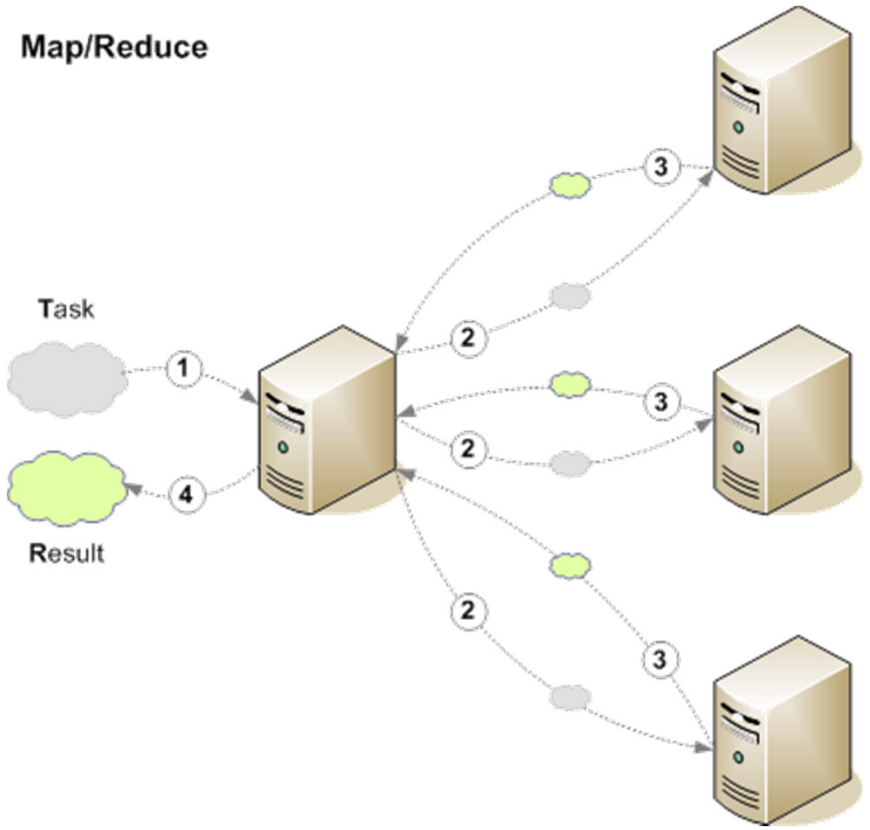
- Motivation
- Programming Model
- Examples
- Implementation
- Refinements
- Performance
- Hadoop

Motivation

- Google has implemented hundreds of special-purpose computations that process large amounts of raw data (crawled documents, web request logs, etc.,)
- The input data is usually large and the computations have to be distributed across hundreds or thousands of machines.
- The issues of how to parallelize the computation, distribute the data, and handle failures require large amounts of complex code to deal with these issues.

Motivation

There is a need for a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations.



Agenda

- Motivation
- Programming Model
- Examples
- Implementation
- Refinements
- Performance
- Hadoop

Programming Model

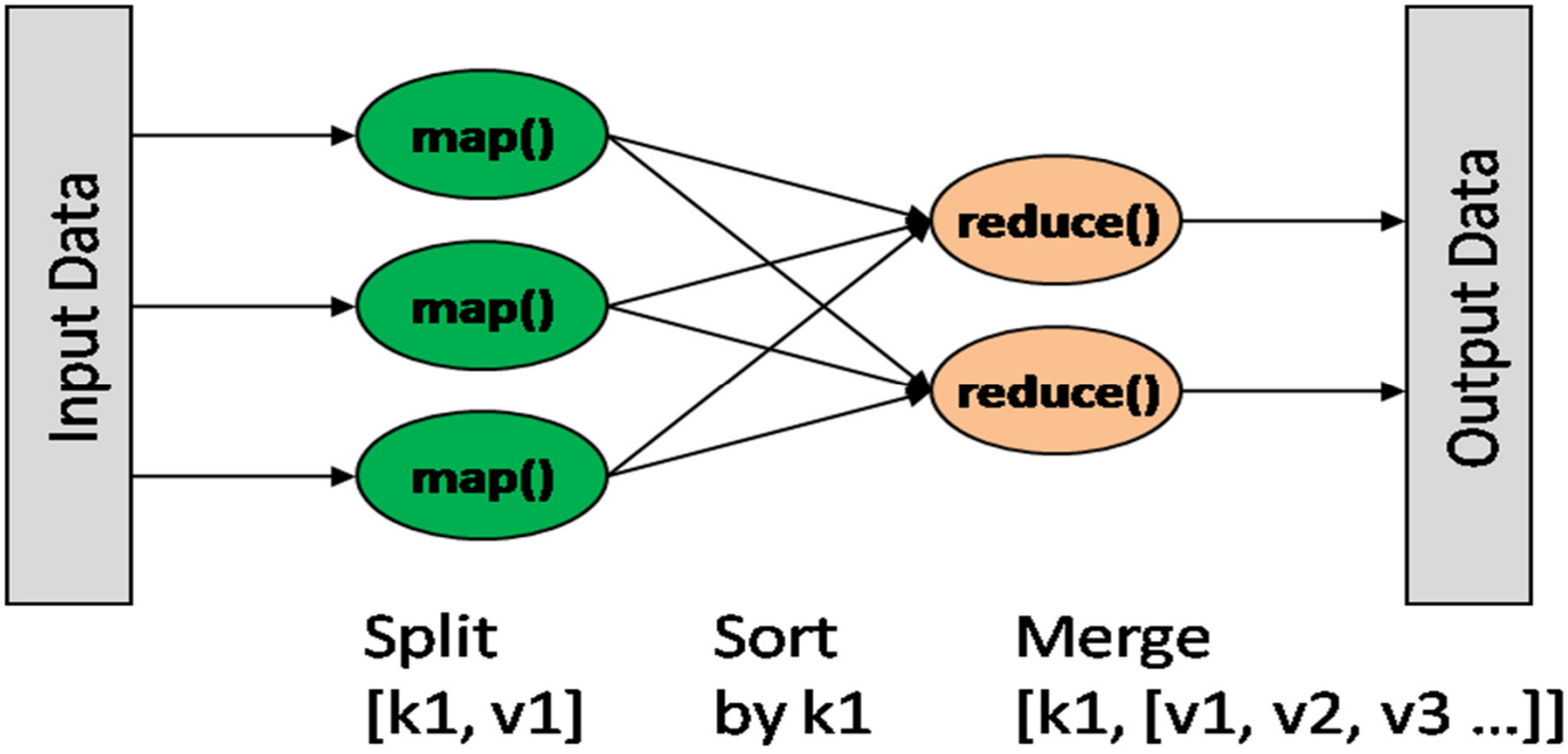
The computations involved applying a map operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately.

Programming Model

The computation takes a set of input key/value pairs, and produces a set of output key/value pairs.(we have to represent the problem in this form!)

Programming Model

The **user** of the MapReduce library expresses the computation as two functions: Map and Reduce



Programming Model

- Map function:

  takes an input pair and produces a set of intermediate key/value pairs.

  The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.

- Reduce function

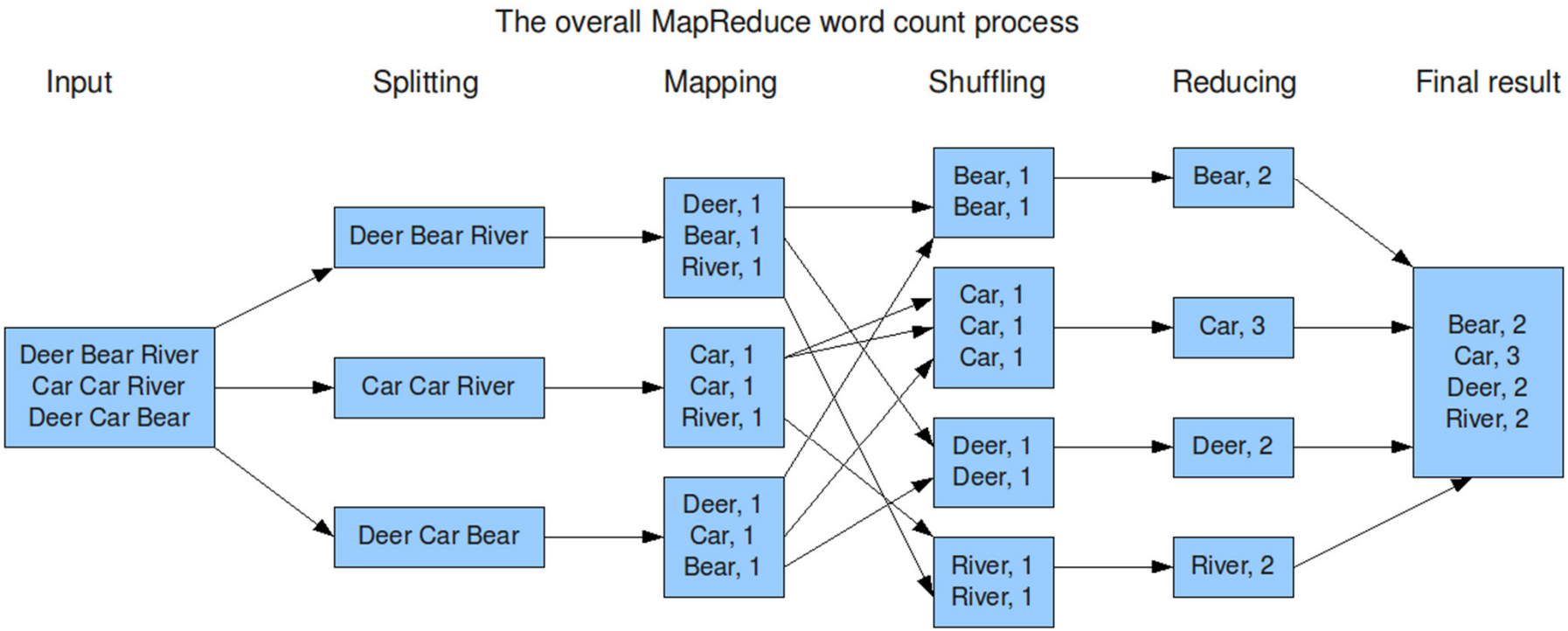
  Accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values.

Agenda

- Motivation
- Programming Model
- Examples
- Implementation
- Refinements
- Performance
- Hadoop

Examples

Word Count



More examples

- Distributed Grep
- Count of URL Access Frequency
- Reverse Web-Link Graph
- Term-Vector per Host
- Inverted Index

Agenda

- Motivation
- Programming Model
- Examples
- Implementation
- Refinements
- Performance
- Hadoop

Implementation

- Many different implementations of the MapReduce interface are possible.
- The right choice depends on the environment (small shared-memory, a large NUMA multi-processor, larger collection of networked machines).
- We will explain an implementation targeted to the computing environment in wide use at Google.

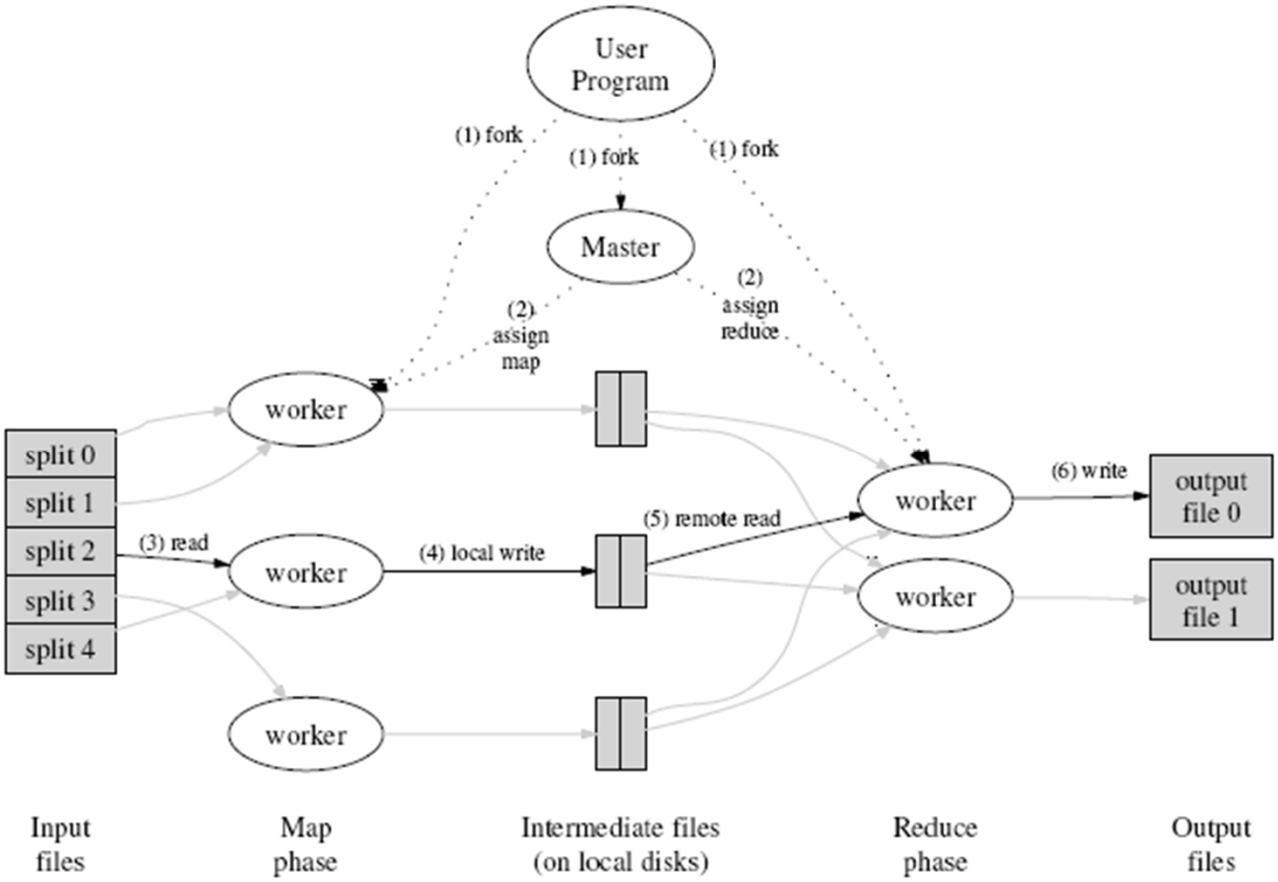
Computing environment in wide use at Google

- Dual-processor x86 processors running Linux, with 2-4 GB
- Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level
- A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.
- Storage is provided by inexpensive IDE disks attached directly to individual machines.
- Users submit jobs to a scheduling system. Each job consists of a set of tasks.

Execution Overview

- The Map invocations are distributed across multiple machines by partitioning the input data into a set of M splits.
- Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R).
- The number of partitions (R) and the partitioning function are specified by the user (can we automate it ?!)

Execution Overview



Execution Overview-notes

- When a reduce worker is notified by the master about the locations of the buffered pairs, it uses **remote procedure calls** to read the buffered data from the local disks of the map workers.
- When a reduce worker has read all intermediate data, it sorts it by th intermediate keys so that all occurrences of the same key are grouped together. As many different keys map to the same reduce task(external sort).

Execution Overview-notes

- The output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user).
- Users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

Fault tolerance

Worker Failure

- The master pings every worker periodically.
- If no response is received from a worker in a certain amount of time, the master marks the worker as failed.
- Any map tasks completed by the worker are reset back to their initial idle state.
- Any map task or reduce task in progress on a failed worker is also reset to idle.

Fault tolerance

Worker Failure

- Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible.
- Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

Worker Failure-notes

- MapReduce is resilient to large-scale worker failures.
- During one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes.
- The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

Master Failure

- The failure can be handled by writing periodic checkpoints of the master data structures (and start a new copy)
- Given that there is only a single master, its failure is unlikely; therefore current implementation aborts the MapReduce computation if the master fails.
- Clients can check for this condition and retry the MapReduce operation if they desire.

Locality

- Network bandwidth is a relatively scarce resource.
- GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines.
- The MapReduce master attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data.

Task Granularity

- Map phase is divided into M pieces and the reduce phase into R pieces
- Ideally, M and R should be much larger than the number of worker machines.
- There are practical bounds on how large M and R can be in
  - The master must make O(M + R) scheduling decisions and keeps O(M $*$ R) state in memory.
  - R is often constrained by users because the output of each reduce task ends up in a separate output file.

Task Granularity

- Map phase is divided into M pieces and the reduce phase into R pieces
- Ideally, M and R should be much larger than the number of worker machines.
- There are practical bounds on how large M and R can be in
  - The master must make O(M + R) scheduling decisions and keeps O(M $*$ R) state in memory.
  - R is often constrained by users because the output of each reduce task ends up in a separate output file.

Backup Tasks

- Straggler: a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation.
- Stragglers can arise for a whole host of reasons:
  - A machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s!
  - Scheduling many tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth.

Backup Tasks

- When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks.
- The task is marked as completed whenever either the primary or the backup execution completes.
- A sort program takes 44% longer to complete when the backup task mechanism is disabled.

Agenda

- Motivation
- Programming Model
- Examples
- Implementation
- Refinements
- Performance
- Hadoop

Partitioning Function

- A default partitioning function is provided that uses hashing (e.g. “hash(key)mod R”).
- This tends to result in fairly well-balanced partitions.
- Sometimes it is useful to partition data by some other function of the key:
  - Using hash(Hostname(urlkey)) mod R” as the partitioning function causes all URLs from the same host to end up in the same output file.

Refinements

- Order Guarantees.
  - support efficient random access lookups by key.
  - Users of the output find it convenient to have the data sorted.
- Side effects
  - While producing auxiliary files as additional outputs from map and/or reduce operators.
  - We rely on the application writer to make such side-effects atomic.

Refinements

Combiner functions

- Word counting: hundreds or thousands of records of the form <the, 1>.
- All of these counts will be sent over the network to a single reduce task
- The Combiner function is executed on each machine that performs a map task.
- The same code is used to implement both the combiner and the reduce functions.
- The output of a reduce function is written to the final output file.
- The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Refinements

Skipping bad records

- Sometimes there are bugs in user code that cause the Map or Reduce functions to cras deterministically on certain records.
- Sometimes it is not feasible to fix a bug; perhaps the bug is in a third-party library for which source code is unavailable.
- Sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set.
- An optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress

Refinements

Status information: Status pages show the progress of the computation

- How many tasks have been completed
- How many are in progress.
- Bytes of input.
- Bytes of intermediate data.
- Bytes of output, processing rates, etc.
- Links to the standard error and standard output files generated by each task.

Refinements

- **Status information**:

The user can use this data to

- Predict how long the computation will take
- whether or not more resources should be added to the computation.
- To figure out when the computation is much slower than expected.

Refinements

Counters:

- user code may want to count total number of words processed or the number of German documents indexed.
- increments the counter appropriately in the Map and/or Reduce function.
- The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response)
- The master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting.

Agenda

- Motivation
- Programming Model
- Examples
- Implementation
- Refinements
- Performance
- Hadoop

Performance

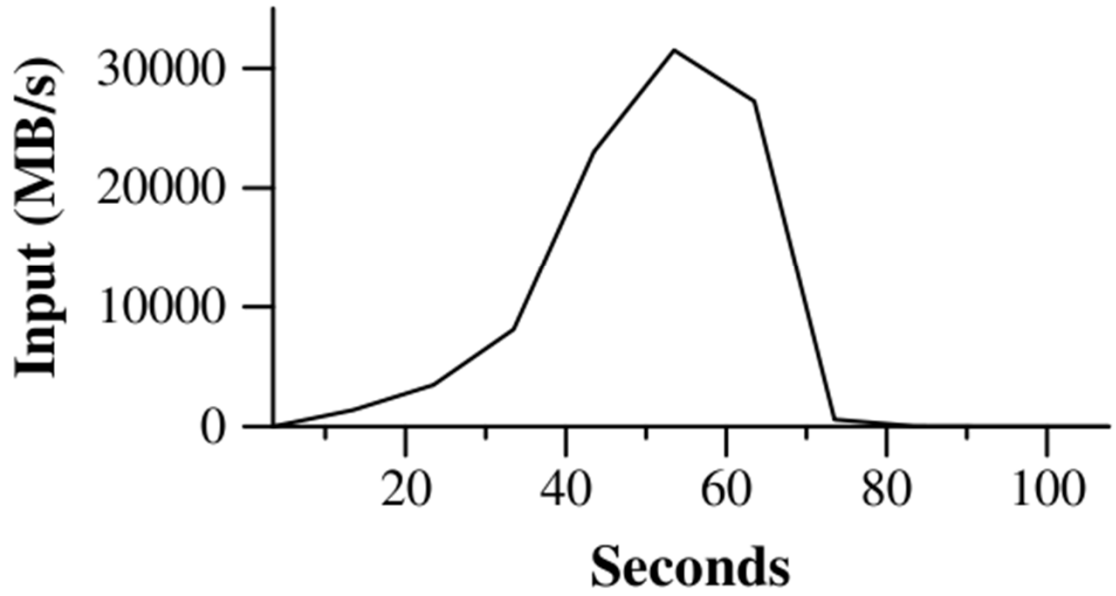
Cluster Configuration

- Cluster consisted of approximately 1800 machines
- Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled,
- 4GB of memory.
- Two 160GB IDE disks

Performance

- Grep
  - The grep program scans through 10^10 100-byte records, searching for a relatively rare three-character pattern
  - The input is split into approximately 64MB pieces (M = 15000), and the entire output is placed in one file (R = 1).

Grep

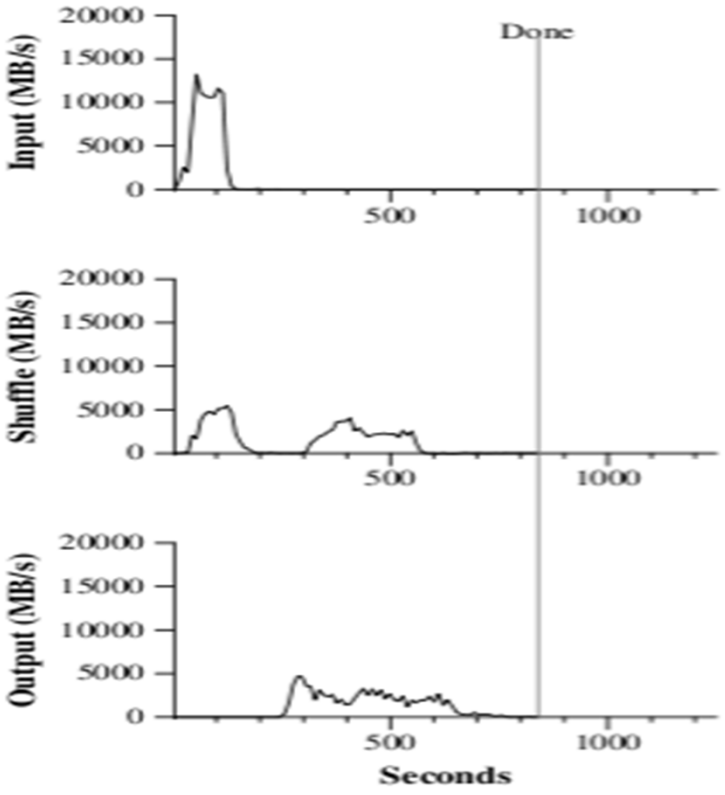


The entire computation takes approximately 150 seconds from start to finish.

Sort

- The sort program sorts 10^{10} 100-byte records (approximately 1 terabyte of data)
- Map function extracts a 10-byte sorting key from a text line and emits the key and the original text line as the intermediate key/value pair.
- Identity function is used as the Reduce operator.
- The input data is split into 64MB pieces (M = 15000).
- The sorted output is partitioned into 4000 files (R = 4000).

Sort-normal execution

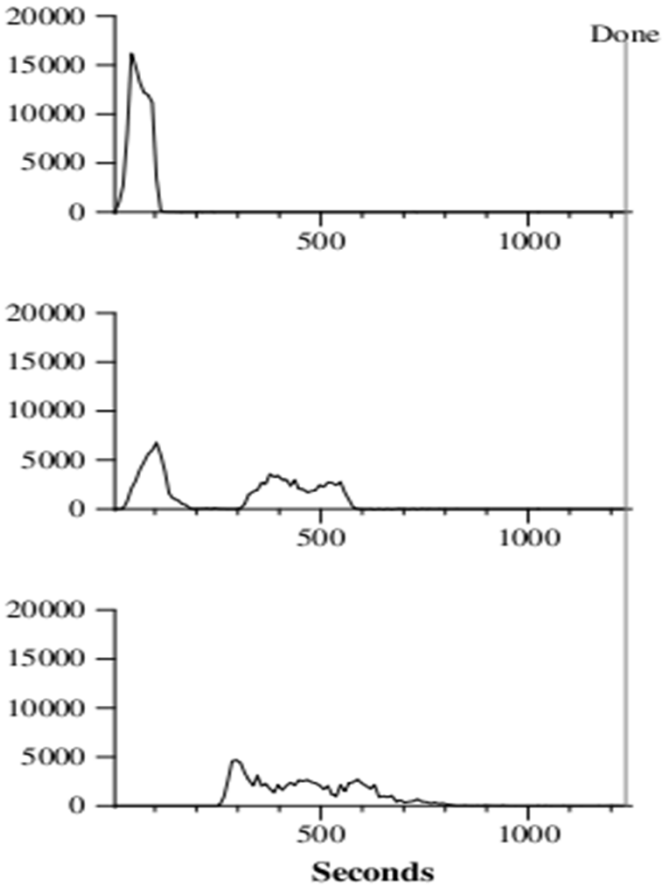


(a) Normal execution

Sort

- The shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time).
- There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data.

Sort-Effect of Backup Tasks

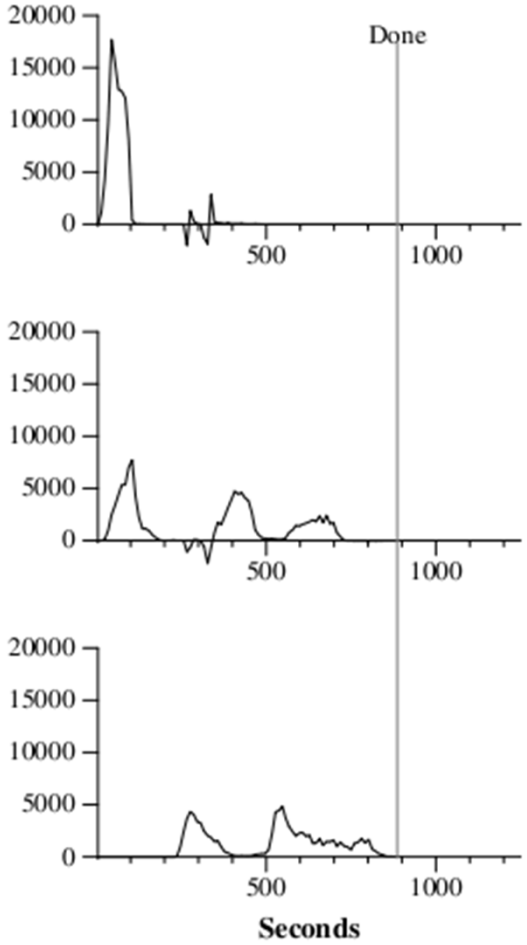


(b) No backup tasks

Sort-Effect of Backup Tasks

- There is a very long tail where hardly any write activity occurs.
- After 960 seconds, all except 5 of the reduce tasks are completed.
- The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

Machine Failures



(c) 200 tasks killed

Machine Failures

- The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly.
- The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

Agenda

- Motivation
- Programming Model
- Examples
- Implementation
- Refinements
- Performance
- Hadoop

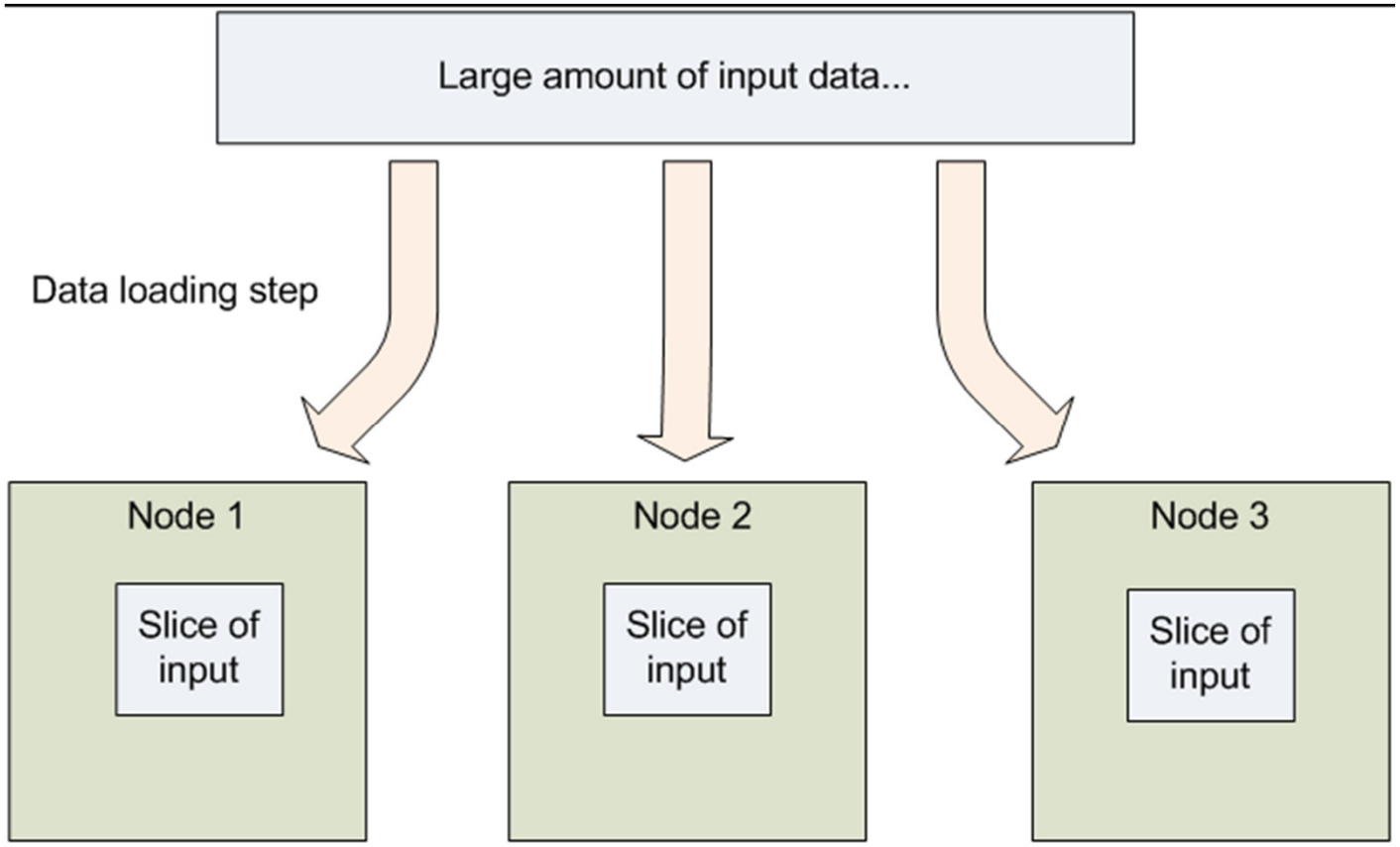
Hadoop

- 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.

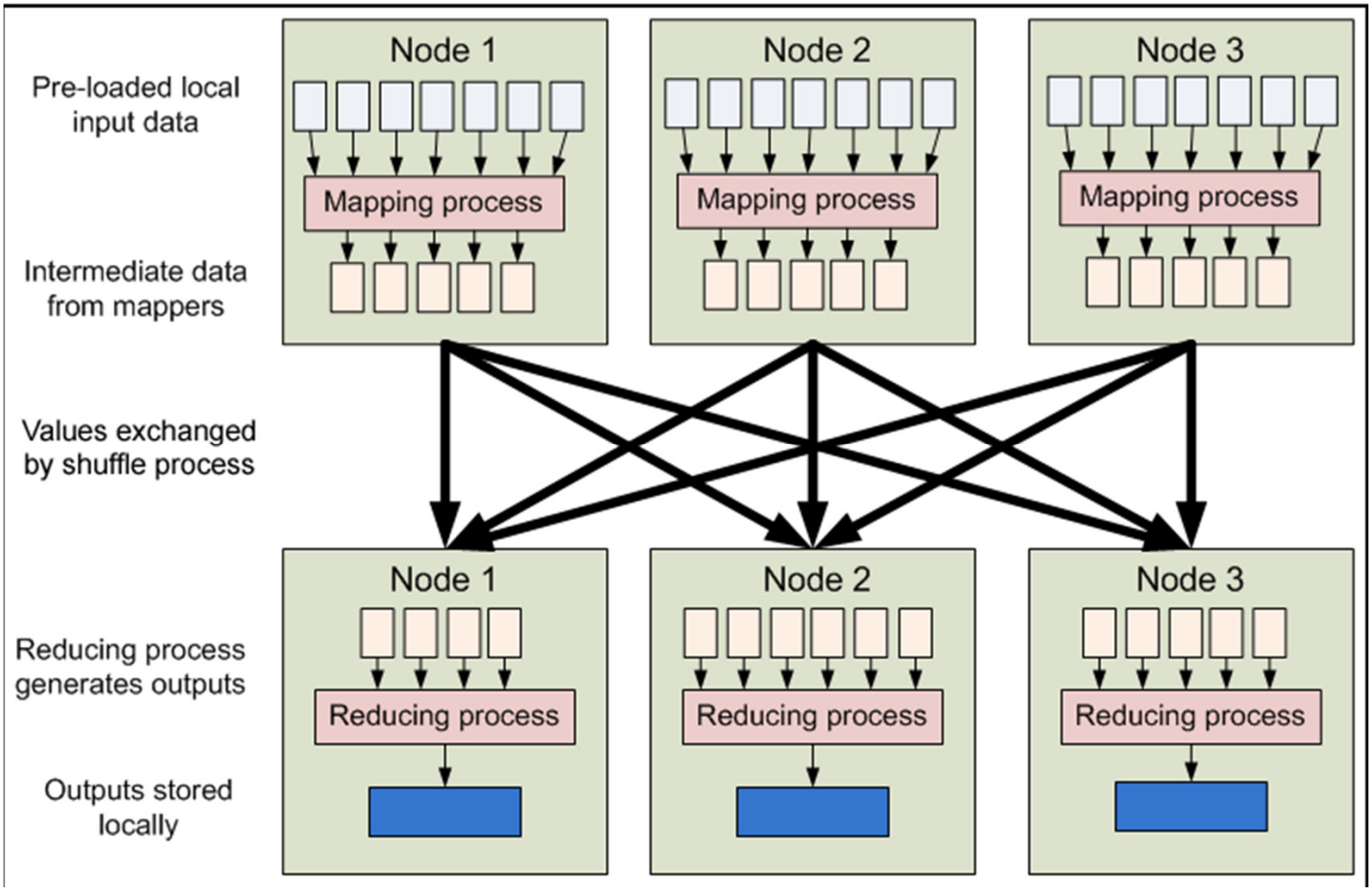
Data distribution

- 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.
- Each chunk is replicated across several machines, so that a single machine failure does not result in any data being unavailable.

Data distribution



Mapreduce-isolated processes



The Hadoop Distributed File System

- HDFS is designed to store a very large amount of information (terabytes or petabytes).
- HDFS should store data reliably.
- HDFS should provide fast, scalable access to this information.
- It should be possible to serve a larger number of clients by simply adding more machines to the cluster.
- HDFS should integrate well with Hadoop MapReduce, allowing data to be read and computed upon locally when possible.

The Hadoop Distributed File System

- HDFS is optimized to provide streaming read performance, this comes at the expense of random seek times to arbitrary positions in files.
- Data will be written to the HDFS once and then read several times; updates to files after they have already been closed are not supported.
- Due to the large size of files, and the sequential nature of reads, the system does not provide a mechanism for local caching of data.
- Individual machines are assumed to fail on a frequent basis, both permanently and intermittently.

The Hadoop Distributed File System

- HDFS is a block-structured file system: individual files are broken into blocks of a fixed size.
- These blocks are stored across a cluster of one or more machines with data storage capacity.
- Individual machines in the cluster are referred to as DataNodes.
- DFS combats the problem of machine failure by replicating each block across a number of machines (3, by default).

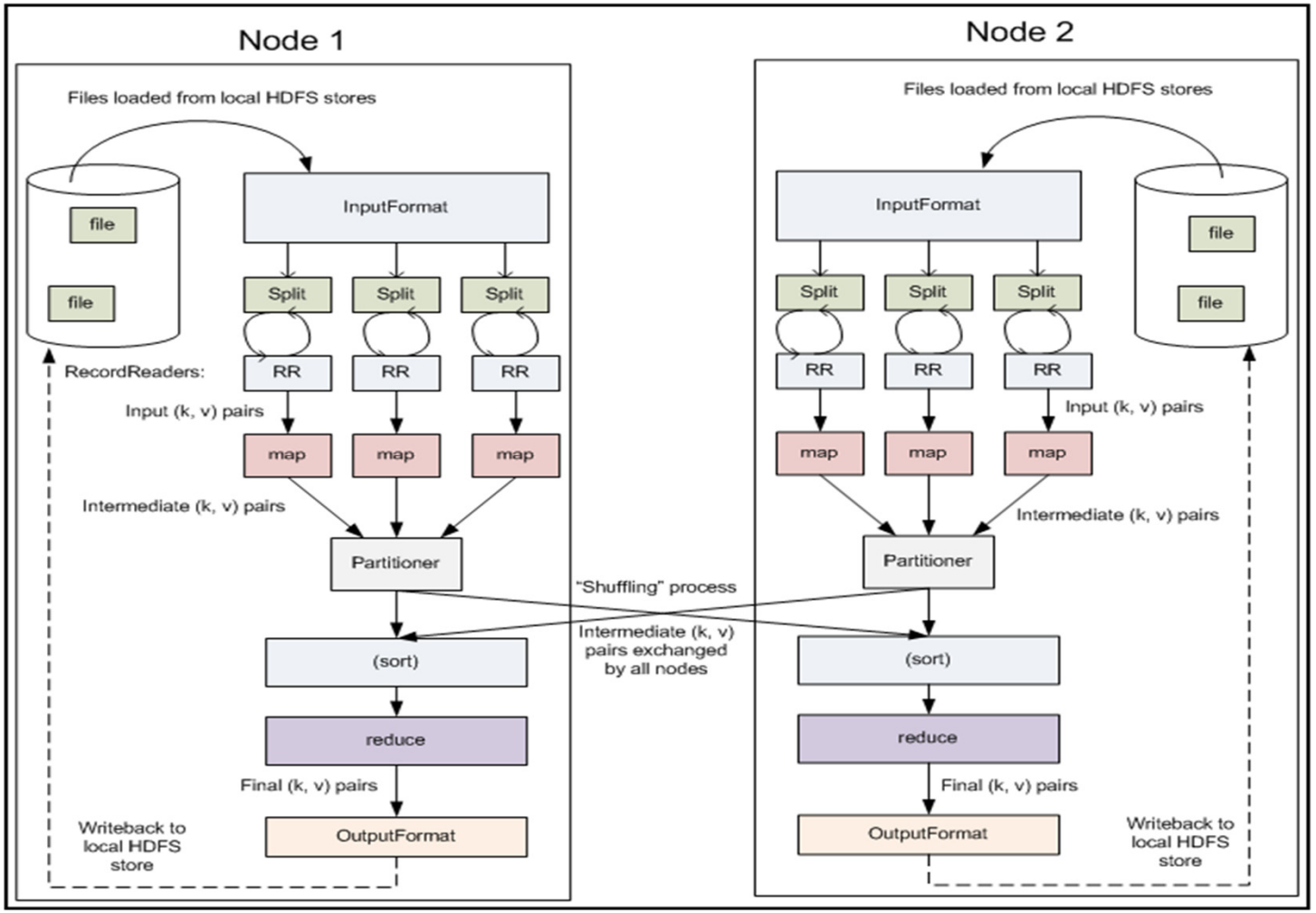
NameNode

- The metadata must be stored reliably.
- The metadata structures (e.g., the names of files and directories) can be modified by a large number of clients concurrently. It is important that this information is never desynchronized.
- The metadata is handled by a single machine, called the NameNode
- Because of the relatively low amount of metadata per file (file names, permissions, and the locations of each block of each file), all of this information can be stored in the main memory of the NameNode machine, allowing fast access to the metadata.

NameNode

- To open a file, a client contacts the NameNode and retrieves a list of locations (DataNodes)for the blocks that comprise the file.
- Clients then read file data directly from the DataNode servers, possibly in parallel.
- The NameNode is not directly involved in this bulk data transfer, keeping its overhead to a minimum.

Mapreduce-a closer look



ZooKeeper

- ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.
- The ZooKeeper service is intended to run on a set of several machines, which prevents the loss of individual nodes from bringing down the cluster.

ZooKeeper

- Manage configuration across nodes
- Implement reliable messaging
- Implement redundant services
- Synchronize process execution

Pig

- Pig is a platform for analyzing large data sets.
- Pig's language, Pig Latin, lets you specify a sequence of data transformations such as merging data sets, filtering, and applying functions to records or groups of records.
- Pig Latin programs are compiled into Map/Reduce jobs, and executed using Hadoop.

Pig

- The highest abstraction layer in Pig is a query language interface, whereby users express data analysis tasks as queries, in the style of SQL or Relational Algebra.
- Queries apply a function to every record in a set, or group records according to some criterion.

Thank You

Question ?