MapReduce Runtime Environments

Design, Performance, Optimizations

Gilles Fedak
Many Figures imported from authors' works

Gilles.Fedak@inria.fr
INRIA/University of Lyon, France

Escola Regional de Alto Desempenho - Região Nordeste

Outline of Topics

1. What is MapReduce ?
   - The Programming Model
   - Principles and Design of Execution Runtimes
2. Performance Improvements and Optimizations
   - Heterogeneity
   - Stragglers and Outliers
   - Greener MapReduce
3. MapReduce beyond the Data Center
   - MapReduce beyond the Data Center
   - Special features and optimizations
   - Performance Evaluation
4. Conclusion

What is MapReduce ?

- Programming Model for data-intense applications
  - Proposed by Google in 2004
  - Simple, inspired by functionnal programming
    - programmer simply defines Map and Reduce tasks
  - Building block for other parallel programming tools
  - Strong open-source implementation: Hadoop
  - Highly scalable
    - Accommodate large scale clusters: faulty and unreliable resources

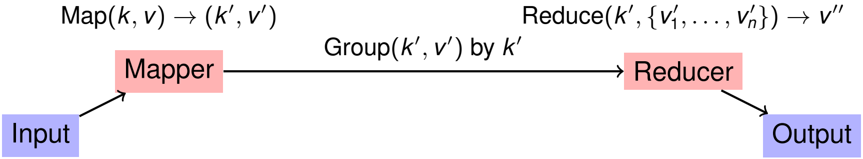
MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat, in **OSDI'04**: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

MapReduce in Lisp

- (map f (list $l_1, \ldots, l_n$))
  - (map square ‘(1 2 3 4))
  - (1 4 9 16)

- (+ 1 4 9 16)
  - (+1 (+4 (+9 16)))
  - 30

MapReduce *a la* Google



$\text{Map}(k, v) \rightarrow (k', v')$

$\text{Reduce}(k', \{v'_1, \dots, v'_n\}) \rightarrow v''$

$\text{Group}(k', v')$ by k'

Mapper

Reducer

Input

Output

- Map(*key*, *value*) is run on each item in the input data set
  - Emits a new (*key*, *val*) pair
- Reduce(*key*, *val*) is run for each unique *key* emitted by the Map
  - Emits final output

Exemple: Word Count

- Input: Large number of text documents
- Task: Compute word count across all the document
- Solution :
  - Mapper : For every word in document, emits `(word, 1)`
  - Reducer : Sum all occurrences of word and emits `(word, total_count)`

WordCount Example

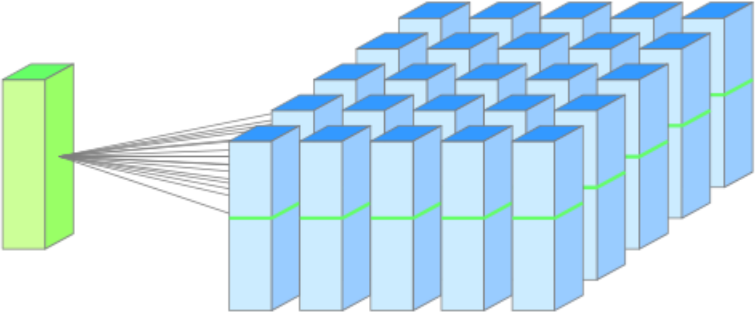
```
//Pseudo-code for the Map function
map(String key, String value):
  // key: document name,
  // value: document contents
  foreach word in split(value):
    EmitIntermediate(word,''1'');

//Peudo-code for the Reduce function
reduce(String key, Iterator value):
  // key: a word
  // values: a list of counts
  int word_count = 0;
  foreach v in values:
    word_count += ParseInt(v);
  Emit(key, AsString(word_count));
```

MapReduce Applications Skeleton

- Read large data set
- Map: extract relevant information from each data item
- Shuffle and Sort
- Reduce : aggregate, summarize, filter, or transform
- Store the result
- Iterate if necessary

MaReduce + Distributed Storage

- Powerfull when associated with a distributed storage system
  - GFS (Google FS), HDFS (Hadoop File System)



The Execution Runtime

- MapReduce runtime handles transparently:
  - Parallelization
  - Communications
  - Load balancing
  - I/O: network and disk
  - Machine failures
  - Laggers

Existing MapReduce Systems

- Google MapReduce
  - Proprietary and close source
  - Closely linked to GFS
  - Implemented in C++ with Python and Java Bindings
- Hadoop
  - Open source (Apache projects)
  - Cascading, Java, Ruby, Perl, Python, PHP, R, C++ bindings (and certainly more)
  - Many side projects: HDFS, PIG, Hive, Hbase, Zookeeper
  - Used by Yahoo!, Facebook, Amazon and the Google IBM research Cloud

Other Specialized Runtime Environments

- Hardware Platform
    - Multicore (Phoenix)
    - FPGA (FPMR)
    - GPU (Mars)
- Security
    - Data privacy (Airvat)
- MapReduce/Virtualization
    - EC2 Amazon
    - CAM

- Frameworks based on MapReduce
    - Iterative MapReduce (Twister)
    - Incremental MapReduce (Nephele, Online MR)
    - Languages (PigLatin)

File System for Data-intensive Computing

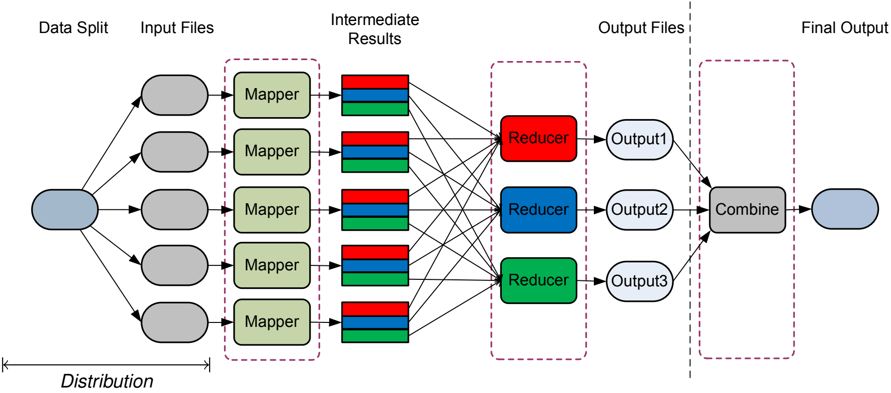
MapReduce is associated with parallel file systems

- GFS (Google File System) and HDFS (Hadoop File System)
- Parallel, Scalable, Fault tolerant file system
- Based on inexpensive commodity hardware (failures)
- Allows to mix storage and computation

GFS/HDFS specificities

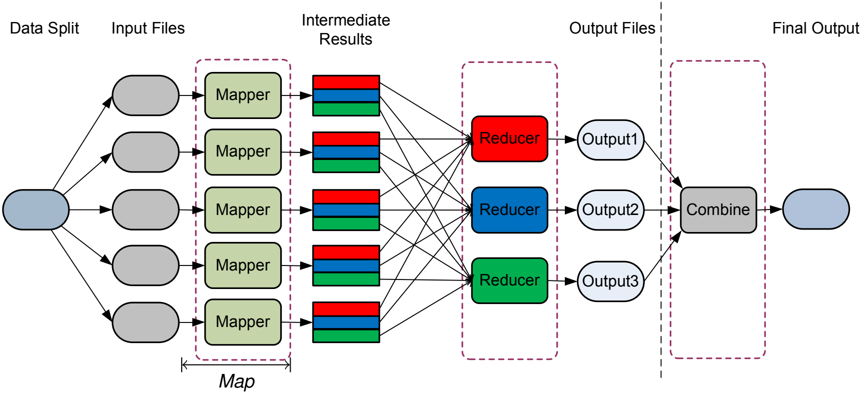
- Many files, large files ($> x$ GB)
- Two types of reads
  - Large stream reads (MapReduce)
  - Small random reads (usually batched together)
- Write Once (rarely modified), Read Many
- High sustained throughput (favored over latency)
- Consistent concurrent execution

Anatomy of a MapReduce Execution



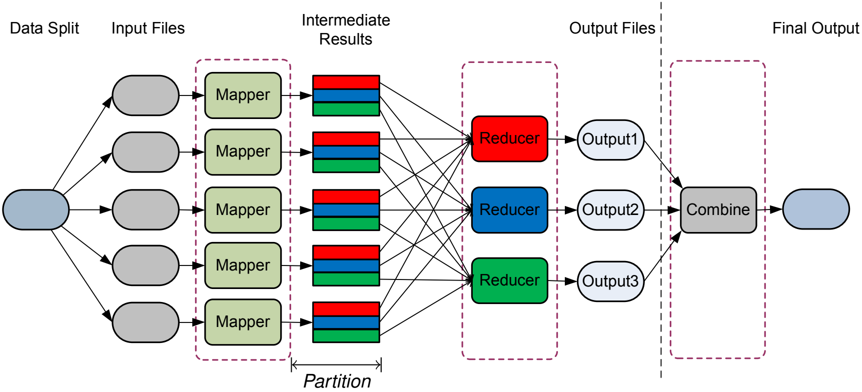
- Distribution :
    - Split the input file in chunks
    - Distribute the chunks to the DFS

Anatomy of a MapReduce Execution



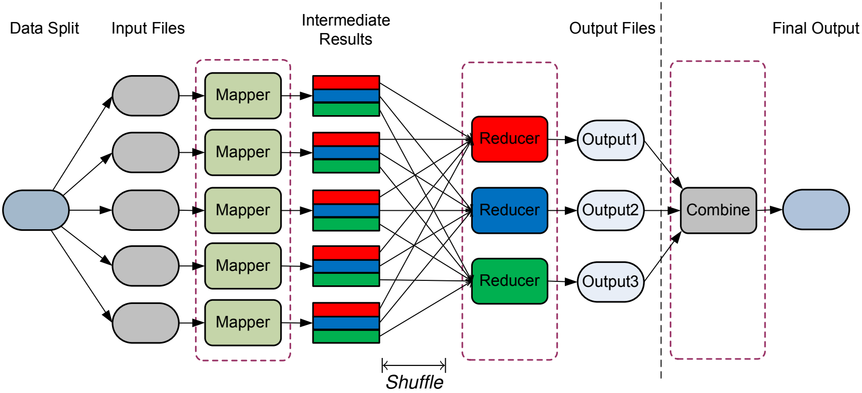
- Map :
  - Mappers execute the Map function on each chunks
  - Compute the intermediate results (i.e., list(k, v))
  - Optionally, a Combine function is executed to reduce the size of the IR

Anatomy of a MapReduce Execution



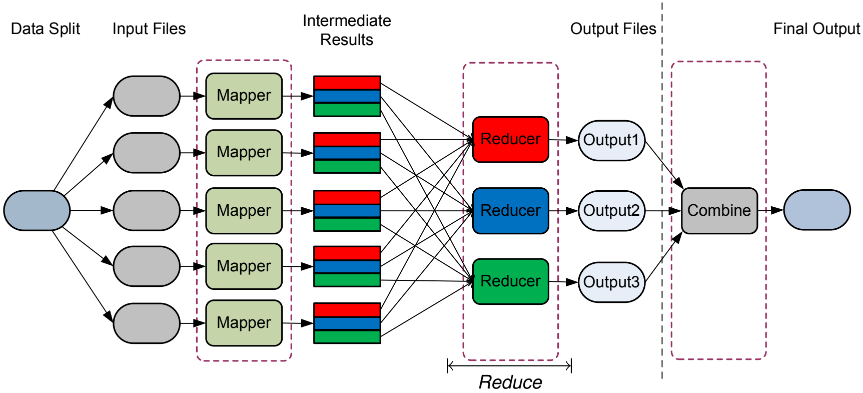
- Partition :
  - Intermediate results are sorted
  - IR are partitioned, each partition corresponds to a reducer
  - Possibly user-provided partition function

Anatomy of a MapReduce Execution



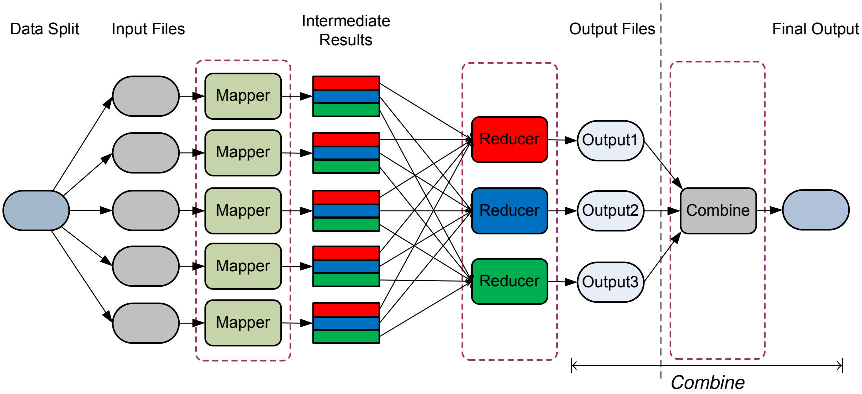
- Shuffle :
  - references to IR partitions are passed to their corresponding reducer
  - Reducers access to the IR partitions using *remore-read* RPC

Anatomy of a MapReduce Execution



- Reduce:
  - IR are sorted and grouped by their intermediate keys
  - Reducers execute the reduce function to the set of IR they have received
  - Reducers write the Output results

Anatomy of a MapReduce Execution



- Combine :
  - the output of the MapReduce computation is available as *R* output files
  - Optionally, output are combined in a single result or passed as a parameter for another MapReduce computation

Handling Failures

On very large cluster made of commodity servers, failures are not a rare event.

Fault-tolerant Execution

- Workers' failures are detected by heartbeat
- In case of machine crash :
  1. In progress Map and Reduce task are marked as *idle* and re-scheduled to another alive machine.
  2. Completed Map task are also rescheduled
  3. Any reducers are warned of Mappers' failure
- Result replica "disambiguation": only the first result is considered
- Simple FT scheme, yet it handles failure of large number of nodes.

Scheduling

Data Driven Scheduling

- Data-chunks replication
  - GFS divides files in 64MB chunks and stores 3 replica of each chunk on different machines
  - map task is scheduled first to a machine that hosts a replica of the input chunk
  - second, to the nearest machine, i.e. on the same network switch
- Over-partitioning of input file
  - #input chunks $>>$ #workers machines
  - # mappers $>$ # reducers

Speculative Execution

- Backup task to alleviate stragglers slowdown:
  - stragglers : *machine that takes unusual long time to complete one of the few last map or reduce task in a computation*
  - backup task : at the end of the computation, the scheduler replicates the remaining in-progress tasks.
  - the result is provided by the first task that completes

Speculative Execution in Hadoop

- When a node is idle, the scheduler selects a task
  1. failed tasks have the higher priority
  2. non-running tasks with data local to the node
  3. speculative tasks (i.e., backup task)
- Speculative task selection relies on a progress score between 0 and 1
  - Map task : progress score is the fraction of input read
  - Reduce task :
    - considers 3 phases : the copy phase, the sort phase, the reduce phase
    - in each phase, the score is the fraction of the data processed. Example : a task halfway of the reduce phase has score $1/3 + 1/3 + (1/3 * 1/2) = 5/6$
- straggler : tasks which are 0.2 less than the average progress score

Issues with Hadoop Scheduler and heterogeneous environment

Hadoop makes several assumptions :

- nodes are homogeneous
- tasks are homogeneous

which leads to bad decisions :

- Too many backups, thrashing shared resources like network bandwidth
- Wrong tasks backed up
- Backups may be placed on slow nodes
- Breaks when tasks start at different times

The LATE Scheduler

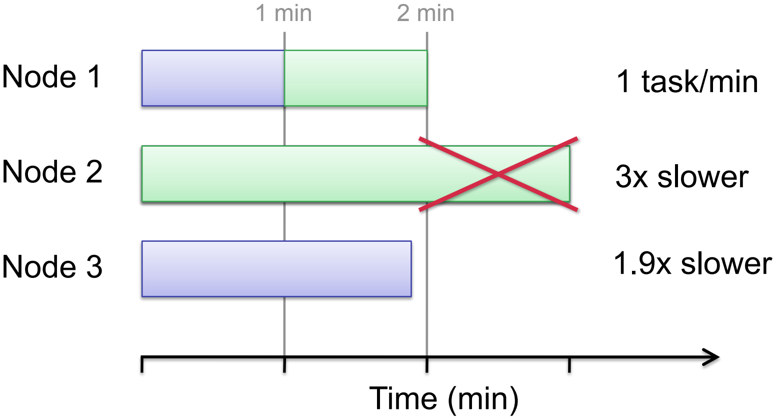
The LATE Scheduler (Longest Approximate Time to End)

- estimate task completion time and backup LATE task
- choke backup execution :
  - Limit the number of simultaneous backup tasks
  - Slowest nodes are not scheduled backup tasks
  - Only back up tasks that are sufficiently slow

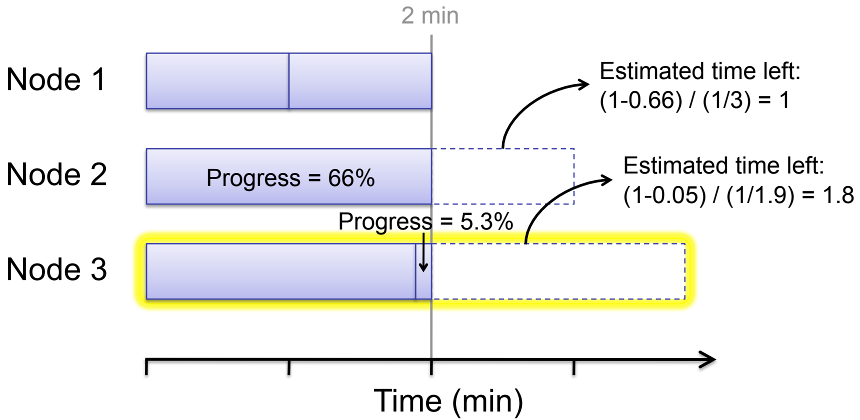
- Estimate task completion time
  - $progress\ rate = \frac{progress\ score}{execution\ time}$
  - $estimated\ time\ left = \frac{1 - progress\ score}{progress\ rate}$

Improving MapReduce Performance in Heterogeneous Environments. Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. and Stoica, I. in **OSDI'08**: Sixth Symposium on Operating System Design and Implementation, San Diego, CA, December, 2008.

Progress Rate Example



LATE Example



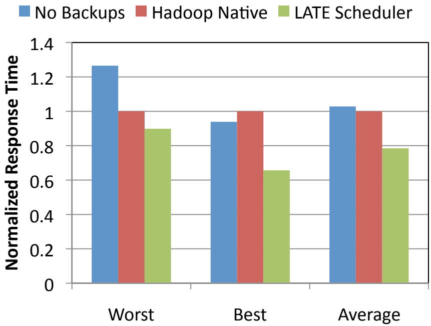
Estimated time left:
(1-0.66) / (1/3) = 1

Estimated time left:
(1-0.05) / (1/1.9) = 1.8

LATE Performance (EC2 Sort benchmark)

Heterogeneity

243VM/99 Hosts 1-7VM/Host

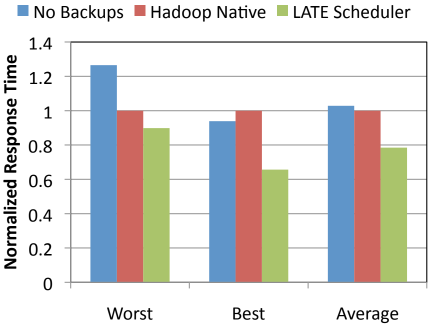


Results

Average 27% speedup over Hadoop, 31% over no backups

Stragglers

4 stragglers emulated by running `dd`

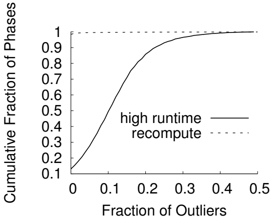


Results

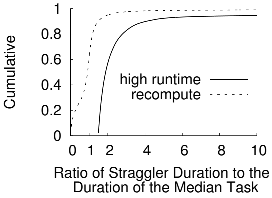
Average 58% speedup over Hadoop, 220% over no backups

Outliers

- Stragglers : Tasks that take $\geq$ 1.5 the median task in that phase
- Recompute : When a task output is lost, dependant tasks that wait until the output is regenerated



(a) What fraction of tasks in a phase are outliers?



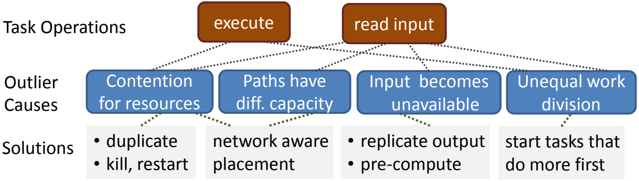
(b) How much longer do outliers take to finish?

Causes of outliers

- Data skew: few keys corresponds to a lot of data
- Network contention : 70% of the crossrack traffic induced by reduce
- Bad and Busy machines : half of recomputes happen on 5% of the machines

The Mantri Scheduler

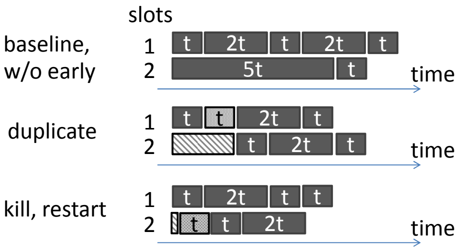
- *cause*- and *resource*- aware scheduling
- real-time monitoring of the tasks : define



Reining in the Outliers in Map-Reduce Clusters using Mantri in G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris **OSDI'10**: Sixth Symposium on Operating System Design and Implementation, Vancouver, Canada, December, 2010.

Resource Aware Restart

- Restart Outliers tasks on better machines
- Consider two options : *kill & restart* or *duplicate*



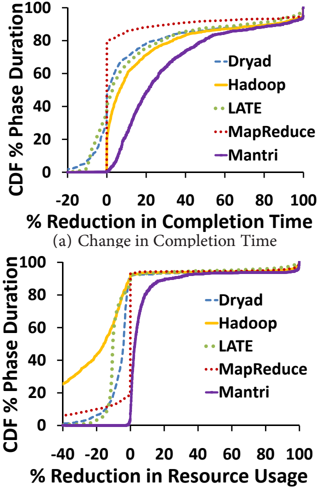
- restart or duplicate if $P(t_{new} < t_{rem})$ is high
- restart if the remaining time is large $t_{rem} > E(t_{new}) + \Delta$
- duplicates if it decreases the total amount of resources used $P(t_{rem} > t_{new}\frac{(c+1)}{c}) > \delta$, where $c < 3$ is the number of copies and $\delta = 0.25$

Network Aware Placement

- Schedule tasks so that it minimizes network congestion
- Local approximation of optimal placement
  - do not track bandwidth changes
  - do not require global co-ordination
- Placement of a reduce phase with n tasks, running on a cluster with r racks, $I_{n,r}$ the input data matrix.
  - Let d_u^i, d_v^i the data to be uploaded and downloaded, and b_u^i, b_d^i the corresponding available bandwidth
  - the placement of is $arg\ min(max(\frac{d_u^i}{b_v^i}, \frac{d_v^i}{b_d^i}))$

Mantri Performances

- Speed up the job median by 55%
- Network aware data placement reduces the completion time of typical reduce phase of 31%
- Network aware placement of tasks speeds up half of the reduce phases by $>$ 60%



(a) Change in Completion Time

(b) Change in Resource Usage

Towards Greener Data Center

- Sustainable Computing Infrastructures which minimizes the impact on the environment
  - Reduction of energy consumption
  - Reduction of Green House Gaz emission
  - Improve the use of renewable energy (Smart Grids, Co-location, etc...)

Solar Bay Apple Maiden Data Center (20MW,) 00 acres : 400 000 m2. 20 MW peak

MapReduce over the Internet

Computing on Volunteers' PCs (a la SETI@Home)

- Potential of aggregate computing power and storage
  - Average PC : 1TFlops, 4BG RAM, 1TByte disc
  - 1 billion PCs, 100 millions GPUs + tablets + smartphone + STB
- hardware and electrical power are paid for by consumers
- Self-maintained by the volunteers

But ...

- High number of resources
- Volatility
- Low performance
- Owned by volunteer

- Scalable but mainly for embarrassingly parallel applications with few I/O requirements
- Challenge is to broaden the application domain

MapReduce over the Internet

Computing on Volunteers' PCs (a la SETI@Home)

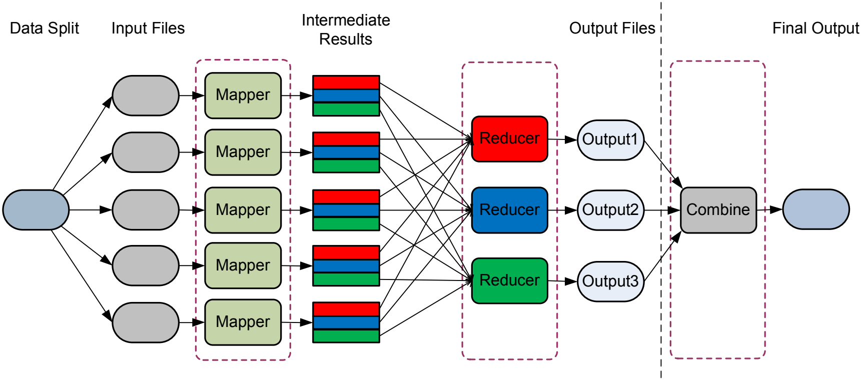
- Potential of aggregate computing power and storage
  - Average PC : 1TFlops, 4BG RAM, 1TByte disc
  - 1 billion PCs, 100 millions GPUs + tablets + smartphone + STB
- hardware and electrical power are paid for by consumers
- Self-maintained by the volunteers

But ...

- High number of resources
- Volatility
- Low performance
- Owned by volunteer

- Scalable but mainly for embarrassingly parallel applications with few I/O requirements
- Challenge is to broaden the application domain

Challenge of MapReduce over the Internet



- no shared file system nor direct communication between hosts
- Faults and hosts churn

- Needs Data Replica Management
- Result Certification of Intermediate Data
- Collective Operation (scatter + gather/reduction)

Prototype of MapReduce over BitDew

BitDew : a Programmable Environment for Large Scale Data Management

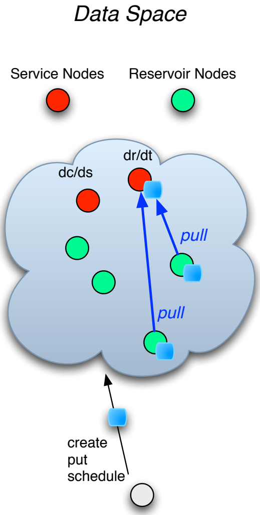
- provides an API and a runtime environment which integrates several P2P technologies in a consistent way
- relies on metadata (*Data Attributes*) to drive transparently data management operation : replication, fault-tolerance, distribution, placement, life-cycle.

Towards MapReduce for Desktop Grids B. Tang, M. Moca, S. Chevalier, H. He, G. Fedak, in Proceedings of the Fifth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing **3GPCIC**, Fukuoka, Japan, Novembre 2010

Anatomy of a BitDew Application

- Clients
  1. creates `Data` and `Attributes`
  2. `put` file into data slot created
  3. `schedule` the data with their corresponding attribute

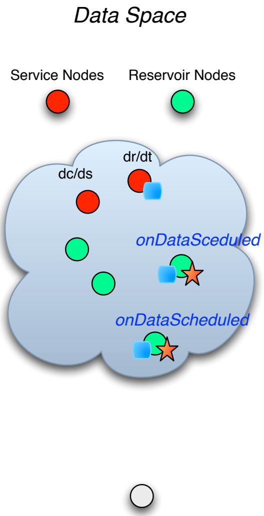
- Reservoir
  - are notified when data are locally scheduled `onDataScheduled` or deleted `onDataDeleted`
  - programmers install callbacks on these events
- Clients and Reservoirs can both create data and install callbacks



Anatomy of a BitDew Application

- Clients
  1. creates `Data` and `Attributes`
  2. `put` file into data slot created
  3. `schedule` the data with their corresponding attribute

- Reservoir
  - are notified when data are locally scheduled `onDataScheduled` or deleted `onDataDeleted`
  - programmers install callbacks on these events
- Clients and Reservoirs can both create data and install callbacks



Data Attributes

REPLICA : indicates how many occurrences of data should be available at the same time in the system

RESILIENCE : controls the resilience of data in presence of machine crash

LIFETIME : is a duration, absolute or relative to the existence of other data, which indicates when a datum is obsolete

AFFINITY : drives movement of data according to dependency rules

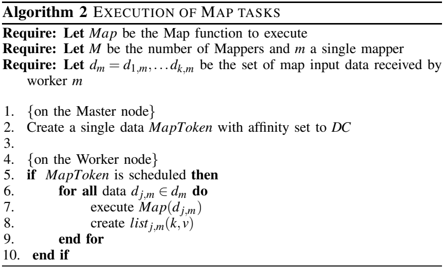
TRANSFER PROTOCOL : gives the runtime environment hints about the file transfer protocol appropriate to distribute the data

DISTRIBUTION : which indicates the maximum number of pieces of *Data* with the same *Attribute* should be sent to particular node.

Implementing MapReduce over BitDew (1/2)

Initialisation by the Master

1. Data slicing : Master creates `DataCollection` $DC = d_1, \ldots, d_n$, list of data chuncks sharing the same `MapInput` attribute.
2. Creates reducer tokens each one with its own `TokenReducX` attribute (affinity set to the Reducer).
3. Creates and pin collector tokens with `TokenCollector` attribute (affinity set to the Collector Token).



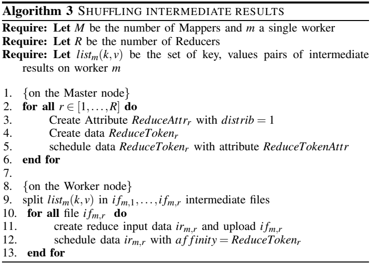
Algorithm 2 EXECUTION OF MAP TASKS

```
Require: Let $Map$ be the Map function to execute
Require: Let $M$ be the number of Mappers and $m$ a single mapper
Require: Let $d_m = d_{1,m}, \ldots d_{k,m}$ be the set of map input data received by
    worker $m$

{on the Master node}
Create a single data $MapToken$ with affinity set to $DC$

{on the Worker node}
if $MapToken$ is scheduled then
    for all data $d_{j,m} \in d_m$ do
        execute $Map(d_{j,m})$
        create $list_{j,m}(k,v)$
    end for
end if
```

Execution of Map Tasks

1. When a Worker receives MapToken it launches the corresponding MapTask and computes the intermediate values $list_{j,m}(k,v)$.

Implementing MapReduce over BitDew (2/2)



Algorithm 3 SHUFFLING INTERMEDIATE RESULTS

Require: Let M be the number of Mappers and m a single worker
Require: Let R be the number of Reducers
Require: Let $list_m(k,v)$ be the set of key, values pairs of intermediate results on worker m

```
{on the Master node}
for all $r \in [1,\ldots,R]$ do
    Create Attribute $ReduceAttr_r$ with $distrib = 1$
    Create data $ReduceToken_r$
    schedule data $ReduceToken_r$ with attribute $ReduceTokenAttr$
end for

{on the Worker node}
split $list_m(k,v)$ in $if_{m,1},\ldots,if_{m,r}$ intermediate files
for all file $if_{m,r}$ do
    create reduce input data $ir_{m,r}$ and upload $if_{m,r}$
    schedule data $ir_{m,r}$ with $affinity = ReduceToken_r$
end for
```

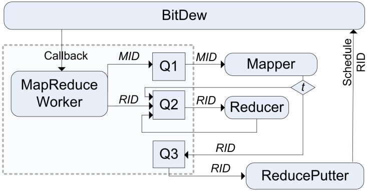
Shuffling Intermediate Results

1. Worker partitions intermediate results according to the partition function and the number of reducers and creates `ReduceInput` with the attribute `ReducTokenR`.

Execution of Reduce tasks

1. When a reducer receives ReduceInput $ir_{m,r}$ files it lauches the corresponding launch ReduceTask $Reduce(ir_{m,r})$.
2. When all the ir_r have been processed, the reducer combines the result in a single final result O_r.
3. eventually, the final result can be sent back to the master using the `TokenCollector` attribute.

Special features and optimizations (1/2)



Latency hiding Implement a multi-thredead worker to ovelap communication and computation. The number of maximum concurrent Map and Reduce tasks can be configured, as well as the minimum number of tasks in the queue before computations can start.

Fault-tolerance In Desktop Grid, computing resources have high failure rates, failures can happen during the computation, either execution of Map or Reduce tasks, or during the communication, that is file upload and download. We simply toggle the resilient flag on the attributes associated to `MapInput` data and `ReduceToken` token. Because intermediate results $ir_{j,r}, j \in [1, m]$ have the `affinity` set to *ReduceToken*$_r$, they will be automatically downloaded by the node on which the *ReduceToken* data is rescheduled.

MapReduce/BitDew

- Barrier-free computation Because of fault tolerance, several replica of the same data can exist in the system, in particular, intermediate results can be duplicated. To tolerate the replication of intermediate results Reducer nodes detects that several versions of the same intermediate file exist in the queue and only process the first one. Early reduction combined with the replication of intermediate results allowed us to remove the barrier between Map and Reduce tasks.
- 2-level scheduler First, the placement of data on host is ensured by the BitDew scheduler, which is mainly guided by the attribute properties given to data. Second, workers periodically report to the MR-scheduler, running on the master node the state of their ongoing computation. The master node can then determine if there are more nodes available than tasks to execute which can avoid the lagger effect.
- Distributed Result Checking Because intermediate results might be too large to be sent back to the server, result certification mechanism cannot be centralized as it is currently implemented in existing Desktop Grid systems. Solution is to create several replicats of Mapinput and Reducer and select correct results by majority voting.

BitDew Collective Communication

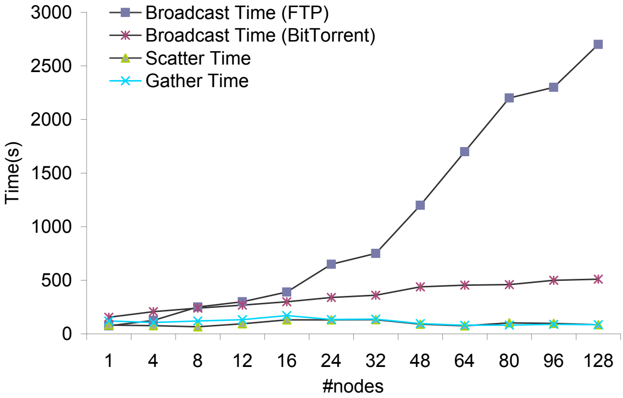


Figure: Execution time of the Broadcast, Scatter, Gather collective for a 2.7GB file to a varying number of nodes

MapReduce Evaluation

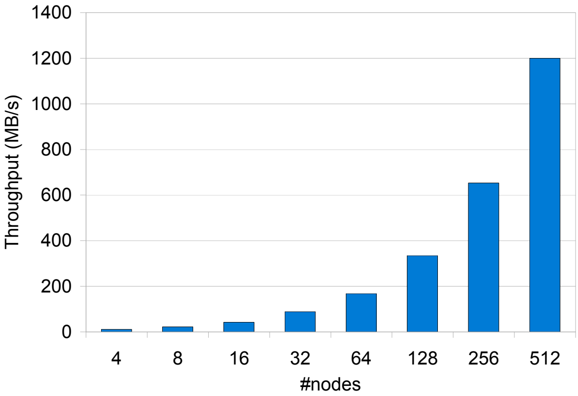


Figure: Scalability evaluation on the WordCount application: the *y* axis presents the throughput in MB/s and the *x* axis the number of nodes varying from 1 to 512.

Fault-Tolerance Scenario

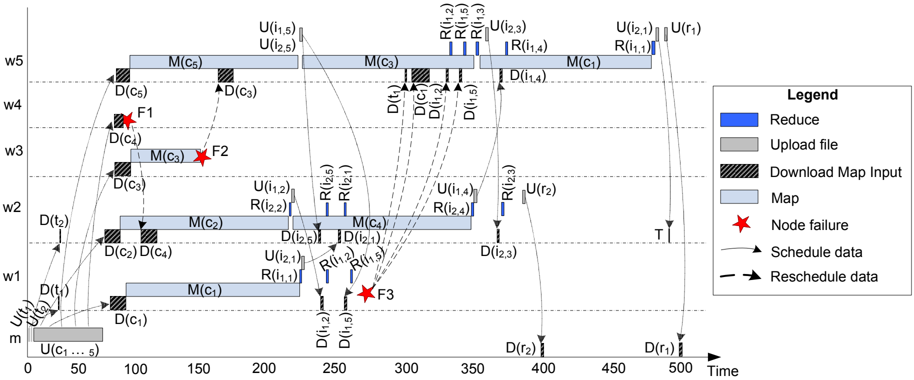


Figure: Fault-tolerance scenario (5 mappers, 2 reducers): crashes are injected during the download of a file (F1), the execution of the map task (F2) and during the execution of the reduce task on the last intermediate result (F3).

Conclusion

- MapReduce Runtimes
  - Attractive programming model for data-intense computing
  - Many research issues concerning execution runtimes (heterogeneity, stragglers, scheduling)
  - Many others : security, QoS, iterative MapReduce,
- MapReduce beyond the Data Center :
  - could enable data-intensive application on Desktop Grid
  - created a first fully functional prototype specifically designed for Internet MapReduce
  - features multi-protocols file transfer, 2-levels data scheduling, automatic replication and transparent data placement, distributed result checking, barrier free computation etc...
- Want to learn more ?
  - the MapReduce Workshop@HPDC (http://graal.ens-lyon.fr/mapreduce)
  - *Desktop Grid Computing* ed. C. Cerin and G. Fedak – CRC Press
  - our websites : http://www.bitdew.net and http://www.xtremweb-hep.org

MapReduce and Hadoop

Luis Rodero-Merino[*], Gilles Fedak[†], and Adrian Muresan[‡]

Avalon Group, INRIA-École Supérieure de Lyon,
46 allée d'Italie, F-69364 Lyon Cedex 7, Lyon, France.

February 8, 2011

1 What is MapReduce?

MapReduce is a solution to address the analysis of huge amounts of data, even in the order of petabytes. The analysis is performed in two steps which are denoted (not surprisingly) *Map* and *Reduce*. The power of MapReduce comes from the fact that in each step the task can be split between different nodes that will run independently and in parallel. MapReduce implementations are usually fault tolerant: if any node fails, the task can be reassigned to some other node. Also, MapReduce shows very good scalability, MapReduce executions can be run on several thousand nodes with the corresponding gain in speed. This makes MapReduce an ideal solution to run distributed tasks on commodity hardware.

MapReduce was introduced in [2]. There, the authors explain how MapReduce emerged as a common solution to address different computation problems, all of them involving the analysis of huge amounts of data and easily parallelizable. The solution provides a framework to simplify the programming of those tasks, that at the same time takes care of the addition and removal of hosts, data transfer (in optimal ways), gathering of results, coordination of task executors, execution planning, status reports, etc. As the authors themselves state, this work was inspired by the *map* and *reduce* primitives present in some functional languages.

1.1 How MapReduce Works

As mentioned before the algorithm has two basic steps: Map and Reduce. At each step a special function defined by the user is run: one function for the Map step, or Map function, and another for the Reduce step, or Reduce function.

[*] email: `luis.rodero-merino@ens-lyon.fr`

[†] email: `gilles.fedak@inria.fr`

[‡] email: `adrian.muresan@ens-lyon.fr`

Figure 1.1 shows an overview of MapReduce running on different nodes. The input data is split into M parts (M is defined by the user), and each part is sent to one of the nodes running the Map function (Map nodes). The Map function input is a key K and a value V, the user must define how these keys and values must be extracted from the data. The Map function "emits" several new keys and values associated, possibly in domains different from those of K and V. As it can be seen in the figure, the values associated to each key by the Map function can be different in different nodes. The framework needs to temporarily store those associations (keys and values) as they are the input for the Reduce processing step.

The Reduce step starts once the Map function has finished. As in the case of the Map function, the Reduce function is run in several nodes. The Reduce function has as input the keys and values generated by the Map function. This set is split into R parts (as M, R is defined by the user). The keys are distributed among the Reduce nodes, so if for example key $k1$ is assigned to some Reduce node, that node will retrieve all the key-value pairs where the key is $k1$ from all the intermediate results created by the Map nodes. See for example how in Figure 1.1 keys $k1$ and $k2$ are assigned to the Reduce node on the left, and how all Map nodes send the values associated to those keys to that node (and only to it). When all values from all keys assigned to that node have been received, the Reduce node sorts them, resulting a list of ordered values. Then, the Reduce function is called for each key in that reduce node, passing as input the key and its list of sorted values. The output of the Reduce function will be another list of values associated to that key (from the same domain of the intermediate values). Finally, all keys and values are gathered from the Reduce nodes and given to the user.

Both Map and Reduce nodes can be run in the same physical machine. Also, depending on the amount of Map and Reduce nodes available, each node can run one or more Map or Reduce tasks. Finally, there is a *Master node* that coordinates the transfer of data, the call to the Map and Reduce function on the corresponding nodes, etc.

1.2 MapReduce as a Cloud Solution

MapReduce defines an approach to deal with certain computational jobs that demand high processing power and operates over big sets of data. It can be implemented in a variety of ways, and depending on its features each implementation can be considered as a cloud solution or not.

To be considered as a proper cloud system, a MapReduce implementation must release users from infrastructure (software and hardware) management concerns, allowing them to focus on their own problem. Such MapReduce implementations can be deemed as a PaaS system. They provide a framework and runtime where developers just upload their code (and input data). The MapReduce implementation will take care of handling that code execution (spawning of map and reduce tasks, recovery from failures, load balancing among nodes...) and retrieving the results. In this regard, it follows the same philosophy of other

PaaS solutions, freeing developers from the tedious and repetitive tasks related with the platform management (software stack and hosting machines). Hadoop can be considered as a PaaS cloud system: once it is properly set in a production environment, developers can use it as the runtime platform for their data processing tasks. They only have to program their corresponding map and reduce functions and deploy them on the Hadoop cluster. They are not concerned about distributing the load, moving code close to data, potential node failures, etc Hadoop can be considered as a PaaS cloud system: once it is properly set in a production environment, developers can use it as the runtime platform for their data processing tasks. They only have to program their corresponding map and reduce functions and deploy them on the Hadoop cluster. They are not concerned about distributing the load, moving code close to data, potential node failures, etc..

1.3 MapReduce Applied to Large Parallel Data Analysis

Although MapReduce was originally developed for use by web enterprises in large data-centers, this technique has gained a lot of attention from the scientific community and corporates for its applicability in large parallel data analysis (including geographic, high energy physics, genomics, business intelligence etc. . .). In this section, we will survey how the MapReduce programming model can be applied to a broad variety of applications ranging from Web and Cloud applications to high performance computing. Surveying this ever growing application ecosystem, we will also outline the strength of MapReduce implementations, and in particular Hadoop, as well as its shortcoming and the alternative systems that are being developed by the open source community and the academic researchers.

Unsurprisingly the first applications which have demonstrated the potential of MapReduce have been proposed by Google and other concurrent Web companies. To understand how MapReduce is employed, let's first examine a classical algorithm used by Google to sort the web pages by their estimated relevance. The principle of the PageRank algorithm [10], proposed in 1999, is to compute for each document a rank value based on the weights associated to the hyperlinks which points to that document. After crawling the web, PageRank extracts the links from the web documents and constructs a gigantic graph representing the web. After that, the algorithm computes iteratively the PageRank values until a convergence is detected. One can imagine how big is the graph representing the Web considering that the web is now more than a trillion of unique URLs. Several times a day, PageRank has to sort PetaBytes of data which shows how crucial is it to have scalable and reliable infrastructure for data-intense computing. MapReduce is not only used to run PageRank but various core business applications : commercial advertisements, comparative trends in search keywords as well as day to day monitoring such as log analysis. As a proof of this pervasiveness, the open-source Hadoop system is now used by the key Internet companies such as Facebook, Baidu, Adobe, Ebay, Twitter, Hulu, LinkedLn, Yahoo[1].

However, when conducting large data analysis, one may experience the fact that the simplicity of the MapReduce model, which relies on a low semantic interface, has the drawback of making the development of complex applications a slow and tedious task. This is why higher level languages have been proposed on top of MapReduce (DryadLINQ [16], Pig Latin [9], Sawzall[11]). For instance, Pig Latin allows developers to program their data processing applications using a declarative syntax has an expressiveness close to the SQL language. The Pig compiler transforms the user's queries in a sequence of MapReduce statements and performs several optimizations on the generated code. The advantage for the user is to benefit from the high degree of intrinsic parallelism and to execute the program on parallel infrastructures using the Hadoop framework. Conversely, the MapReduce programming model is finding a place in the database world. For example, CouchDB[6] is a schema-free distributed database which has been designed to store unstructured or semi-structured documents such as blog posts, email or tweets for instance. Instead of using SQL queries, developers extract the relevant information from the database by writing Map and Reduce tasks.

The scientific community is a traditional consumer of high performance parallel computing. As soon as the model appeared few years ago, some scientists tried to apply the model to their research applications. The first noticeable experiments have focused on machine learning [1], where ten classical algorithms (linear regression, k-means, logistic regression, naive Bayes, gaussian, neural network, backpropagation and more) were parallelized with almost linear speedup when executed on multicores machine. Following this pioneer work, various similar and successful attempts have been made to *mapreduce* their algorithms such as Monte-Carlo simulation, matrix multiplication, pair-wise computation, image processing, genetic sequence search and many more. Today MapReduce appears to be the language of choice to extract meaning from very large scientific datasets. The combination of strong open-source solution like Hadoop with the large scalability of Cloud infrastructure is offering a promising solution for the scientists to solve the data deluge challenge they are facing. However, these experiments have also outlined some shortcoming with the Hadoop system. One of the key feature of scientific applications is their iterative nature because the applications are often described as workflows of Map and Reduce tasks which iterates until a solution has been reached. The Twister environment[3] developed at the Indiana University is a runtime execution system dedicated to iterative cpu/data intensive application with optimizes the data scheduling and the reuse of intermediate data between the iterations.

Although Hadoop and Elastic MapReduce by Amazon are the reference implementation of MapReduce for clusters and for the Cloud, there is a need for implementation dedicated for other types of computing infrastructures. The open source community and the researchers in computer science have proposed no-

[1] The complete list of known Hadoop users can be found on the Haddop web page (`http://hadoop.apache.org`)

ticeable alternative to Hadoop. [5] describes an implementation of MapReduce on top of MPI, the message passing communication, which allows MapReduce to be executed on Supercomputer or cluster with high performance network. Phoenix [12] is an implementation of MapReduce for multicores and multiprocessors systems while [4] describes an implementation for GPU systems. In [8] and [13], Desktop Grid versions of MapReduce are proposed which would allow to use existing PCs in a LAN or provided by Internet users when they are idle to run MapReduce applications.

MapReduce is now a well established technology in the Internet industry and has found many early adopters in the scientific community. Surfing on the wave of the NoSQL database, it would not be surprising to see a broader adoption by all the major IT enterprises and why not database vendors. We can anticipate that Hadoop and MapReduce are going to play a central role for the next future in data manipulation and analysis.

2 About Hadoop

Hadoop[2] is a project of the Apache Software Foundation (ASF) that consists of several subprojects for scalable distributed computing. Its origins lay in the Nutch project[3], that tries to build an open-source web search engine. Nutch developers found the same problems that Google had: the crawling and analysis of web-pages (at web scale) requires huge amounts of computation over immense sets of data. When the MapReduce paper [2] was published Nutch developers implemented the same ideas for their own search engine. Later on, Yahoo hired some Nutch developers to start the Hadoop project as a split of the Nutch project under the ASF umbrella.

Hadoop is not just a framework to run MapReduce tasks. Instead, Hadoop is formed by three core subprojects, each one having a well-defined functionality:

- *Hadoop Common.* Set of utilities used by the rest of Hadoop projects. Also, it provides the libraries and scripts needed to start a Hadoop cluster, as for example utilities to handle Hadoop's filesystem (see paragraph below), MapReduce libraries...

- *Hadoop Data File System* (HDFS). A file system for the management of big data sets (in the order of terabytes or even petabytes) in distributed environments. Understanding how HDFS works it is necessary to successfully operate Hadoop's MapReduce, and helps to understand the kind of analysis works Hadoop is oriented to. Thus, a brief description to HDFS is provided in Section 2.1).

- *Hadoop Map/Reduce.* Hadoop's MapReduce framework, explained in Section 3.

[2] http://hadoop.apache.org
[3] http://nutch.sourceforge.net

Also, there are several subprojects based on Hadoop's MapReduce and/or HDFS, like for example:

- *Hive*. It is a "*data warehouse infrastructure*" based on Hadoop.
- *HBase*. Column-oriented database, is part of the *NoSQL* set of solutions for structured data that have appeared recently.
- *Chukwa*

We encourage the interested reader to further explore those projects purpose and fundamentals.

2.1 Hadoop Distributed File System

In any Hadoop installation (sometimes called "*Hadoop cluster*") HDFS is the main building block, the pillar that all other Hadoop services such as MapReduce use as basis[4]. Thus, it is necessary to understand how HDFS works before addressing MapReduce functionality. This is only a brief description of HDFS, more info is provided at Hadoop web site and in Hadoop books [7, 15, 14].

HDFS is a distributed file system with emphasis on high throughput and failure tolerance. It is intended to be run on cluster made with commodity hardware (it does not require expensive hardware to run efficiently), to handle large data sets. It implements its own namespace, where data is organized in files. Each file is split into blocks, and each block is copied to different hosts in the cluster (typically three copies of each block are created). Having several blocks of data allows for better reliability. Also, HDFS provides *data location awareness*, which allows to assign computing tasks (such as those required by MapReduce) to those nodes that are "closer" to the data. Thus, services and applications using HDFS tend to move code to those nodes where data is stored, instead of moving data to executor nodes. This make sense as Hadoop is oriented to computations that involve the processing of huge amounts of data.

Consistency is managed by HDFS. Some simplifications are assumed that make HDFS differ from other distributed file systems. Data in HDFS are written only once (it cannot be rewritten), once the file is written and closed its contents cannot be changed. This limits the set of services or applications that can run on top of HDFS, but on the other hand fits very well the requirements of the applications Hadoop is designed for.

The HDFS architecture is based on two types of nodes: the *MasterNode* and the *DataNode*:

1. *NameNode*. There is only one NameNode process, that takes care of managing the HDFS. It provides a namespace to identify files, and keeps a mapping between each file in that namespace and the data blocks that contain the file contents. There will be different replicas of each data

[4] In fact, Hadoop MapReduce can run on the local filesystem. However HDFS features make it ideal to be used along with MapReduce works

block, and each replica will be stored in different nodes of the Hadoop cluster. The NameNode is aware of the location of all the replicas of each data block, and decides where new replicas must be hosted. Whenever a data access must be performed by some process, it calls the NameNode to know where the data is located. Also, the NameNode ensures that the configured number of replicas of each data block is always available. The NameNode is a single point of failure for Hadoop. There can be other secondary NameNodes running in the same cluster, but they only keep track of the actions of the primary MasterNode to replace it in case it fails. If no master NameNode is available, then all Hadoop services will not be able to run.

2. *DataNode.* Typically, there is one DataNode for each host storing data in the cluster. The DataNode stores HDFS data blocks (along with its metadata), serves this data to other processes running in the host and sends the data to other DataNodes. DataNodes are coordinated by the NameNode. Each DataNode sends *heartbeat* signals to the NameNode. Whenever the NameNode does not receive heartbeats from some DataNode, it will make new replicas of the data blocks stored by that failed DataNode. The goal is to make sure that at all times the configure amount of data replicas are available.

3 Hadoop Implementation of MapReduce

In Hadoop's jargon, a *job* is a MapReduce computation, comprised of its input data, its Map and Reduce functions and its configuration. The Map and Reduce functions will be run as *tasks*. Usually, each function will be run by several tasks in different nodes, each task processing one part of the input data.

Hadoop's MapReduce service is provided by two types of nodes: *JobTrackers* and *TaskTrackers*:

1. *JobTracker.* This node monitors the execution of MapReduce *jobs*. As explained above, each MapReduce job is split into several Map and Reduce *tasks*. The JobTracker coordinates all these tasks and commands where they must be executed. Ideally, this will happen in the same host where data is already located or at least in a host in the same rack. The JobTracker uses the NameNode to know where data is stored and sends tasks to the known TaskTrackers. The JobTracker is a point of failure for the Hadoop MapReduce service, if it fails no MapReduce jobs will be run.

2. *TaskTracker.* Receives tasks assignments from the JobTracker. Each TaskTracker process attends task execution petitions from the JobTracker, and notifies it when some task is finished (successfully or not). A TaskTracker has a defined number of tasks it can attend at the same time (*slots*). Each TaskTracker sends periodically a *heartbeat* signal to the JobTracker carrying the number of available slots in that TaskTracker. This way, the

JobTracker is aware of the amount of slots available at each host. This information is necessary to plan task assignments. If after a certain time the JobTracker does not receive any heartbeat from a certain TaskTracker, the JobTracker will assume that node is down and will reassign the tasks sent to that node to some other TaskTracker.

Usually, MapReduce will be run on top of the HDFS, so nodes will perform I/O operations on that file system. Other data sources can be used instead (Amazon S3, HTTP, FTP... or even the local filesystem) but then the system will not be aware of data location, which is required to plan where tasks must be run to minimize data traffic.

The JobTracker can used different scheduling algorithms to assign tasks to TaskTracker depending on their free slots.

4 Installing Hadoop

First thing to do when installing Hadoop is which kind of setup is going to be installed. Hadoop :

- *Local* A local setup runs Hadoop in your local machine as a single process, and it is oriented to beginners that want to try and evaluate Hadoop. By default, Hadoop is ready to operate in local mode.

- *Pseudo-distributed* All nodes run in the local machine, but in separate processes.

- *Cluster* For “real” works in production environments, the cluster setup must be chosen instead. This setup builds a Hadoop cluster where nodes are distributed across physical hosts. As explained before, a Hadoop cluster will have one NameNode, one JobTracker, and several DataNodes and TaskTrackers. The NameNode and the JobTracker will usually run in their own machines, while the rest of hosts will typically run one DataNode and one TaskTracker each. This is depicted in Figure 4.

This section will explain how to install a minimum but complete Hadoop cluster with a NameNode, a DataNode, a JobTracker and a TaskTracker. All of them will be running in the same host, so this would be in fact a pseudo-distributed installation by Hadoop terminology. But instead of starting all processes at once we will start them one by one as if they were started in different hosts, and *each node will have its own set of configuration files* instead of sharing them as in typical pseudo-distributed installations of Hadoop. The goal of this approach is to help to understand how the process of configuring and starting each type of node in a cluster environment would be done, by allowing the reader to start all nodes in separated steps without requiring having different machines available. Once those concepts are clear, moving this to a *Cluster* installation should be easy. Hence, from now on we will use the term *cluster* to refer to both the typical *Cluster* installation and the setup explained here.

4.1 Required Software

Hadoop has been extensively used in Linux-based environments. Running it on top of Windows systems is possible but has not been thoroughly tested as yet. Thus, we will assume that physical hosts run Linux.

Also, Hadoop requires a Java Runtime Environment (JRE) to work, version 6. Sun JRE is advised. We will assume that Java binaries (e.g. `java` command to start java programs) are available from the command line.

Communication with Hadoop nodes will be done through ssh, so the `ssh` utility must be installed and sshd daemons must be running on all cluster hosts. Also, the `rsync` utility is likewise needed[5].

```
$ sudo apt-get install ssh
$ sudo apt-get install rsync
```

Also, Hadoop nodes must be able to use ssh without having to set the password. This can be done as follows:

```
$ ssh-keygen -t dsa -P ’’ -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

This will allow our own user to ssh to the local machine without the need to introduce the password. If hadoop is going to be run by other users, them the contents of their `$HOME/.ssh/id_dsa.pub` files must be added to their `$HOME/.ssh/authorized_keys`.

Hadoop software can be downloaded from Hadoop website[6]. Look for the latest release. It will be a compressed `.tar.gz` file, uncompress it with the `tar` utility. For example, assuming you downloaded the `hadoop-0.20.2.tar.gz` file:

```
$ tar -zxvf hadoop-0.20.2.tar.gz
```

This will create the `hadoop-0.20.2` folder. From now on, we will assume that the absolute path to that folder will be stored in the `$HADOOP_HOME` environment variable:

```
$ export HADOOP_HOME=/path_to_hadoop_folder/hadoop-0.20.2
```

WARNING: for the scripts to run properly, it is advised not to use spaces in the `$HADOOP_HOME` path.

The main binary is in `$HADOOP_HOME/bin/hadoop`. This is a script that first loads configuration settings by calling to another script in the same folder, `$HADOOP_HOME/bin/hadoop-config.sh`. And in fact, all scripts that start the different Hadoop services call to that same common configuration script. In `hadoop-config.sh`, the `hadoop-env.sh` script is called in turn. In that file at least the `$JAVA_HOME` environment variable must be set to the path the

[5] `rsync` is used by Hadoop for data replication among nodes. But users do not have to care about this as this is done transparently. Hence we will not refer further to this tool

[6] `http://hadoop.apache.org/common/releases.html`

folder where the JRE was installed (in Ubuntu, this would be /usr/lib/jvm/java-6-sun). Truth is, most of Hadoop nodes configuration is through the *.xml files in the $HADOOP_HOME/conf folder (or whatever configuration folder the user sets when calling the script). But being aware of the existence of these scripts is useful for example if we want to pass certain parameters to the java virtual machines running the nodes.

The hadoop binary can be used to run MapReduce tasks as a single process (local setup), without the need of a name server service.

The following step is to start running a pseudo-distributed installation of hadoop, by starting a NameNode, a DataNode, a JobTracker and a TaskTracker in the localhost.

Hadoop has a single script to run its nodes as daemon processes, $HADOOP_HOME/bin/hadoop-daemon.sh, that calls to hadoop. Another script, $HADOOP_HOME/bin/hadoop-daemons.sh is used to call hadoop-daemon.sh in a list of remote nodes (ssh connectivity with those remote nodes must be available), these list is by default read from the $HADOOP_HOME/conf/slaves file. Finally, this is used by $HADOOP_HOME/bin/start-dfs.sh and $HADOOP_HOME/bin/start-mapred.sh, commodity scripts to start the HDFS and MapReduce services at once. A typical pseudo-distributed installation would use the $HADOOP_HOME/bin/start-all.sh script, that in turn calls the start-dfs.sh and start-mapred.sh scripts. But, as commented before, we will start nodes manually for a better understanding of the startup process. Thus, we will only make use of the hadoop script.

4.2 Installation of a HDFS service

First, we will start a HDFS service with its NameNode and one DataNode. The NameNode needs to use a folder to store the HDFS metadata it handles, (i.e., the HDFS filesystem). So the initial step is to "format the filesystem" in that folder. The folder to format is, by default /tmp/hadoop/dfs/name. This can be changed by setting the dfs.name.dir parameter, which is defined in the $HADOOP_HOME/conf/hdfs-site.xml file. The value can be an absolute or a relative path. The following code shows an example of the contents of hdfs-site.xml to use ./data :

```
<configuration>
  <property>
    <name>dfs.name.dir</name>
    <value>./data</value>
  </property>
</configuration>
```

Now we only need to run the hadoop script (remember that $HADOOP_HOME and $JAVA_HOME must be set!):

```
$ cd $HADOOP_HOME
$ bin/hadoop namenode -format
```

This would prepare the folder `./data`, relative to the folder from where the script was called, to contain the HDFS data. Each time the NameNode is started, it will check in that same parameter where the metadata is stored.

Once the data folder is ready the NameNode can be run. Nodes are started also by the `$HADOOP_HOME/bin/hadoop` script. The NameNode URI, where it will listen for connections, is set in the `core-site.xml` file in the configuration folder (by default `$HADOOP_HOME/conf`, but this can be changed when calling to the script), by the `fs.default.name` property. For example, if we want the NameNode to listen for connections in port 9000 of the localhost the contents of that file would be:

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

When the `core-site.xml` file is ready we *must* set the `$HADOOP_HOME` variable to the path where Hadoop was installed (the folder created when the corresponding `hadoop-*-.tar.gz` file was decompressed). Then we can start the NameNode by running:

```
$ cd $HADOOP_HOME
$ bin/hdfs namenode
```

The following step is to start a DataNode. The network address the DataNode will listen to is given by the `dfs.datanode.address`, that is set in the `hdfs-site.xml`. Of course, different DataNodes in the same Hadoop cluster must have the same value for the `fs.default.name` in their `core-site.xml` file, but different values in the `dfs.datanode.address` parameter. In general, nodes in the same Hadoop cluster will have the same `core-site.xml` file but different `hdfs-site.xml` files. If the DataNode is started from the same Hadoop installation than the NameNode (i.e. from the same folder in the same host), and to avoid mixing their configurations, we can copy the contents of the `conf` directory to a new folder to store the DataNode configuration:

```
$ cd $HADOOP_HOME
$ cp -r conf confDataNode
```

Usually this will not be necessary in real Hadoop clusters as the NameNode will not run in the same host than other nodes. But in this case and for the sake of simplicity we are

And then the `confDataNode/hdfs-site.xml` file would be edited as needed. Besides the `dfs.datanode.address`, in the `hdfs-site.xml` file we can use the `dfs.data.dir` parameter to configure in which folder the DataNode must store data. So the resulting `hdfs-site.xml` file could look like this:

```
<configuration>
  <property>
    <name>dfs.data.dir</name>
    <value>./dataNodeStore</value>
  </property>

  <property>
    <name>dfs.datanode.address</name>
    <value>localhost:8990</value>
  </property>
</configuration>
```

Finally, to start the DataNode hadoop script is called again:

```
$ cd $HADOOP_HOME
$ bin/hadoop --config ./confDataNode datanode
```

Note how hadoop is instructed to read the configuration not from the default directory but from confDataNode (from now on, when calling to the bin/hadoop script, it will be assumed that it is called from the $HADOOP_HOME folder).

Now that there is a HDFS system running, the user can store and retrieve data from it (almost) like in any other filesystem by using the hadoop util.

Also, it is possible to browse the filesystem through a web interface that the NameNode offers. It can be checked in localhost:50070. To change this address the dfs.http.address in the hdfs-site.xml file can be used.

It is beyond the scope of this text to explain all the possible configuration parameters of the HDFS system. All these parameters can be found in the $HADOOP_HOME/src/hdfs/hdfs-default.xml file, along with an explanation of what are they intended for. The user should not modify this file, but use it as reference. Similarly, there is a $HADOOP_HOME/src/core/core-default.xml file with the common configuration parameters of the Hadoop nodes.

To browse and operate the Hadoop filesystem the hadoop utility can be used with the fs flag. If you run

```
$ bin/hadoop fs
```

the script will show all the HDFS contents manipulation options available from the script. As before, $HADOOP_HOME and $JAVA_HOME must be set conveniently. All these operations are performed by contacting the NameNode, whose address is read again from the $HADOOP_HOME/conf/core-site.xml file. To change this, you can apply a different configuration file (by using the -conf flag) or use the -fs flag.

To add contents to the HDFS, hadoop -put must be used. For example, if we had our input data in folder /var/input, running

```
$ bin/hadoop fs -put /var/input /
```

would create the /input folder in the HDFS system, with the same contents than in the local filesystem. Thus, executing now:

```
$ bin/hadoop fs -ls /input
```

would show the folder contents.

4.3 Installation of a MapReduce Service

Once there is a HDFS service running, we can proceed to start a small MapReduce service that uses the former to store and read data.

First, the JobTracker must be started. Configuring it will be straightforward. First, as before, we will create a folder that it will use as configuration:

```
$ cd $HADOOP_HOME
$ cp -r conf confJobTracker
```

Now, the JobTracker has to use the same core-site.xml than the NameNode (mainly to know the address the NameNode is listening for connections at). The other parameter needed is the own JobTracker address. This must be set in the $HADOOP_HOME/confJobTracker/mapred-site.xml file with the mapred.job.tracker parameter (for example, it could be set to localhost:2244 or any other port the user wishes). Once this is set, the JobTracker can be started with the hadoop tool (once more, $JAVA_HOME must be set):

```
$ bin/hadoop --config ./confJobTracker jobtracker
```

And finally, a TaskTracker node must be started. For its minimum configuration, a TaskTracker only needs the same configuration files than the JobTracker so we could use the same configuration folder. However, we would create a different configuration folder for it as in typical deployments TaskTracker would have some configuration of their own.

```
$ cd $HADOOP_HOME
$ cp -r confJobTracker confTaskTracker
```

Then, the TaskTracker is started by running:

```
$ bin/hadoop --config ./confTaskTracker tasktracker
```

Keep in mind nonetheless that both the JobTracker and the TaskTracker are highly configurable. All the configuration options can be checked in $HADOOP_HOME/src/mapred/mapred-default.xml.

4.4 Sending a MapReduce Job

To conclude this Section we will program a small MapReduce example that will be run in the Hadoop cluster we have just built. The goal is to describe how MapReduce jobs must be sent to the cluster, and how to insert input data and retrieve output data from the HDFS.

The example to run is a well-known MapReduce program to count the number of occurrences of words. It uses as input a set of files that contain the text to analyze, and it gives as output a list containing all the words found and the times each word appears in the files.

First, we will insert the input files in the `/wordcount/input` folder of the HDFS filesystem. We will use as input the `.html` files in the `$HADOOP_HOME/docs` folder (and, even once more, remember that `$JAVA_HOME` and `$HADOOP_HOME` must be set):

```
$ cd $HADOOP_HOME
$ bin/hadoop fs -mkdir /wc/input
$ bin/hadoop fs -put docs/*html /wc/input
```

The output folder `/wordcount/output` where we plan to store the results will be automatically created by the code. If this was not the case, we would create it as follows:

```
$ bin/hadoop fd -mkdir /wc/output
```

Now, there are several versions of the wordcount program available. As we aim in this section not to explain how to program MapReduce jobs but only how to deploy them on Hadoop clusters we will not describe now how to program the example. More complex MapReduce examples will be described in Section 5. But now we suggest the reader to use the code available in the Hadoop manual[7]. Following the instructions in that manual, the user will get a `wordcount.jar` file with the code to be run. Once this file is available, for example in the user home dir, and assuming the class with the `main()` method is `org.myorg.WC`, we would deploy it on our Hadoop cluster by running:

```
$ bin/hadoop jar ~/wordcount.jar org.myorg.WC /wc/input /wc/output
```

Note that the configuration used is the default one, i.e., the one in `$HADOOP_HOME/conf`. This is enough, as the script only needs to know where the NameNode is located (which was defined in `core-site.xml`) to ask for the JobTracker and send the MapReduce job. Once the job is finished, we can extract the results from the HDFS:

```
$ bin/hadoop fs -ls /wc/output
Found 1 items
-rw-r--r-- 3 user supergroup 527728 2010-10-10 12:01 /wc/output/part-r-00000
$ bin/hadoop fs -get /wc/output/part-r-00000 wordCountResults.txt
```

Results will be stored in the local file `wordCountResults.txt`.

[7] `http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html`

5 Programming a Non-Trivial MapReduce Example with Hadoop

The current section illustrates the Hadoop API by means of a practical example. We have chosen a an application that uses geo-location. The purpose of the example is to group Wikipedia[8] articles by their geographic coordinates and present the result to the output. The input data is in the form of a .CVS file offered by DBPedia[9], containing article titles and geographical coordinates[10].

The map-reduce workflow has the following fixed steps:

1. **Map step.** The input data is distributed to the mapper nodes in the hadoop deployment. Each mapper node consumes the input data in raw format and outputs a set of key-value pairs that serve as input for the next step. There is no communication between the mapper nodes. In fact, all mapper nodes are identical and perform the same operation on the input data. The input data makes the only difference between one mapper node and another.

2. **Combiner step.** This is an optional step that acts as a local reduce for each mapper node. Its goal is to reduce the network traffic by doing a mapper-local reduction of values that belong to the same keys. The combiner is actually identical to the reducer, but may not be always usable. If the reduce operation is both associative and commutative, then a mapper-local reduce will not influence the value of the final result and this means that the combiner is usable.

3. **Shuffle step.** This step uses the previously-generated key-value pairs as input. During this step, all values with the same key are being sent to the same reducer node. At the end of the step the reducers have keys associated to them and a list with all the values belonging to a key, for each of the keys they process. The shuffle step is done automatically by the framework and is the only communication step in the whole workflow.

4. **Reduce step.** During the reduce step, each reducer aggregates the values associated to the keys that it is suppose to process. At the end of the computation, the result is outputted locally. Like the mappers, the reducers do not communicate with one-another.

Given the fact that all mappers and reducers are created equally, they can also be replaced easily. Because there is no explicit communication being done it follows that recovery from failure can be done automatically. In fact, the Hadoop framework itself is responsible for restarting tasks that have previously failed. This make the platform itself resilient to failure. To aid task restarting,

[8] `http:\\www.wikipedia.org`

[9] `http://dbpedia.org`

[10] `http://downloads.dbpedia.org/3.3/en/pagelinks_en.csv.bz2`

it is advisable that neither the mapper nor the reducer have any side effects or explicit communication.

From an implementation point of view, there are three main functionalities that need implementing: the mapping operation, the reduction operation and the job configuration.

5.1 The mapping operation

In order for the Hadoop framework to recognize mapper and reducer implementation, the Hadoop API offers a base class for them: $org.apache.hadoop.mapred.MapReduceBase$.

The mapper class also needs to implement the $Mapper < K1, V1, K2, V2 >$ interface and provide functionality for mapping input key-value pairs of type $K1, V1$ to intermediary key-value pairs of type $K2, V2$. Input pairs are generated by an input formatter, originating from the $InputFormat$ base class. Each entry given by the formatter, called $InputSplit$, generates one set of input key-value pairs. For each of these input pairs, the framework automatically calls the $map(K1, V1, OutputCollector < K2, V2 >, Reporter)$ method. Each input pair can generate one or several intermediary pairs that are collected in the $OutputCollector < K2, V2 >$ instance. The produced intermediary pairs, do not need to be of the same type as the input pairs. The *Reporter* instance is a utility that helps in keeping track of job progress.

In the current example, the input data is in text form as a .CVS file. The input formatter produces one input entry at a time, corresponding to each line in the input file. As a result, the input value is of type $Text$. The input key is of no influence to the rest of the map-reduce process and the $LongWritable$ was used, which is a class implementing $Comparable$ and $Writable$ for $Long$ values.

The intermediary keys are groups of the form $(< rounded - latitude >, < rounded - longitude >)$ and are of type $Text$. Therefore, the intermediary keys represent the geographic location of a Wikipedia article.

The intermediary value retains the type of the input value and is obtained by prepending the value of the key to the input value.

The current example produces only one intermediary key-value pairs for each input pair since each entry in the input file corresponds to only one Wikipedia article.

In what follows, we detail the implementation of the current example.

```
public class GeoLocationMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, Text> {
// the input key-value pairs are of type LongWritable and Text
// the intermediary key-value pairs are both of type Text

public static String GEO_RSS_URI = "http://www.georss.org/georss/point";

// in these members we store the intermediary key-value pairs
private Text geoLocationKey = new Text();
private Text geoLocationName = new Text();
```

```
// this method is run for each input item give by the input formatter
public void map(LongWritable key, Text value,
OutputCollector<Text, Text> outputCollector, Reporter reporter)
throws IOException {
// this represents one data item
String dataRow = value.toString();

// since these are tab seperated files lets tokenize on tab
StringTokenizer dataTokenizer = new StringTokenizer(dataRow, "\t");

String articleName = dataTokenizer.nextToken();
String pointType = dataTokenizer.nextToken();
String geoPoint = dataTokenizer.nextToken();

// we know that this data row is a GEO RSS type point.
if (GEO_RSS_URI.equals(pointType)) {
// now we process the GEO point data.
StringTokenizer tokenizer = new StringTokenizer(geoPoint, " ");

// obtain the latitude and longitude
String stringLat = tokenizer.nextToken();
String stringLong = tokenizer.nextToken();

double doubleLat = Double.parseDouble(stringLat);
double doubleLong = Double.parseDouble(stringLong);

// round the coordinates to long values
long roundedLat = Math.round(doubleLat);
long roundedLong = Math.round(doubleLong);

// construct the intermediary key
String locationKey = "(" + String.valueOf(roundedLat) + ","
+ String.valueOf(roundedLong) + ")";

// construct the intermediary value
String locationName = URLDecoder.decode(articleName, "UTF-8");
locationName = locationName.replace("_", " ");
locationName = locationName + ":(" + doubleLat + "," + doubleLong + ")";

// store the intermediary key-value pair
geoLocationKey.set(locationKey);
geoLocationName.set(locationName);

// collect the intermediary key-value pair
outputCollector.collect(geoLocationKey, geoLocationName);
}
}
}
```

5.2 The reduction operation

As in the case of the mapping operation, the reducer must also extend the *MapReduceBase* class. The reducer must implement the $Reducer < K2, V2, K3, V3 >$ interface. Its role is to transform a set of intermediary key-value pairs of type $K2, V2$, to a smaller set of key-value pairs of type $K3, V3$. All intermediary values that share the same key are grouped into a collection. For each of these groups, the $reduce(K2, Iterator < V2 >, OutputCollector < K3, V3 >, Reporter)$ method is automatically called. Typically, the collection of values that are associated to the same key, are reduced into zero or one final value. In contrast to the mapper, the output keys and values need to be of the same type as the intermediary keys and values. Like the mapper, the reducer can also report its progress by means of the *Reporter* instance.

In our current example, the intermediary and output key-value pairs are all of type $Text$. The reduce operation itself is actually a concatenation of all the article titles that have been found in the same geographic location.

The articles are automatically grouped by geographic location since this is what the intermediary key represents.

In the following we will detail the source code that implements the reduce operation.

```
public class GeoLocationReducer extends MapReduceBase implements
Reducer<Text, Text, Text, Text> {
// the intermediary and output key-value pairs are of type Text

// these members store the output key-value pair
private Text outputKey = new Text();
private Text outputValue = new Text();

// this method is called automatically for each intermediary key
// and its corresponding list of values
public void reduce(Text geoLocationKey, Iterator<Text> geoLocationValues,
OutputCollector<Text, Text> results, Reporter reporter)
throws IOException {
// in this case the reducer just creates a list so that the data can be used later
String outputText = "";

// the reducer receives a list of intermediary values for each key
// here we iterate through the list and construct the reduced value
while (geoLocationValues.hasNext()) {
Text locationName = geoLocationValues.next();
outputText = outputText + locationName.toString() + " ,";
}

// prepare the output key-value pair
outputKey.set(geoLocationKey.toString());
outputValue.set(outputText);

// collect the output key-value pair
results.collect(outputKey, outputValue);
}
}
```

5.3 The job configuration

To be able to orchestrate job execution, the Hadoop runtime needs to know some configuration parameters. The most basic of these parameters are the classes that implement the mapper and reducer functionalities and what formatter the runtime should use for parsing the input.

For our current example, we have detailed the implementation of these classes in the previous subsections.

The code behind our example's configuration can be found below.

```
public class GeoLocationJob {

// the program entry point
public static void main(String[] args) throws Exception {
// store job configuration
JobConf conf = new JobConf(GeoLocationJob.class);
conf.setJobName("geolocationgroup");

// set the classes of the output key-value pairs
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);

// set the mapper and reducer class
conf.setMapperClass(GeoLocationMapper.class);
conf.setReducerClass(GeoLocationReducer.class);

// set the input formatter
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

// the input and output files are taken
// from command line arguments
FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
// launch the job
JobClient.runJob(conf);
}
}
```

Acknowledgments The authors would like to thank *Subbu Nagarajan* for his open-source, freely-available and usable examples[11] for the Hadoop framework, which have been an inspiration for the current work.

References

[1] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. *Advances in Neural Information Processing Systems*, (19):281–288, 2007.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, January 2008.

[3] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a Runtime for Iterative MapReduce. In *1st Workshop on MapReduce Programming Model and its Applications*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[4] Bingsheng He, Wenbin Fang, Qiong Lo, Naga K. Govindaraju, , and Tuyong Want. A MapReduce Framework on Graphics Processors. In *Parallel Architectures and Compilation Techniques (PACT)*, Toronto, Canada, 2008.

[5] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards Efficient MapReduce Using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting.* Springer, Sep. 2009.

[6] http://couchdb.apache.org.

[7] Chuck Lam. *Hadoop in Action.* Manning Publications, December 2010.

[8] Heshan Lin, Jeremy Archuleta, Xiaosong Ma, and Wuchun Feng. MOON: MapReduce On Opportunistic eNvironments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, Chicago, 2010.

[9] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[11] `http://code.google.com/p/hadoop-map-reduce-examples/`

[10] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[11] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[12] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA)*, Phoenix, AZ, 2007.

[13] Bing Tang, Mircea Moca, Stphane Chevalier, Haiwu He, and Gilles Fedak. Towards MapReduce for Desktop Grid Computing. In *Fifth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'10)*, Fukuoka, Japan, November 2010.

[14] Jason Venner. *Pro Hadoop*. Apress, June 2009.

[15] Tom White. *Hadoop the Definitive Guide*. O'Reilly, 2nd edition, October 2010.

[16] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

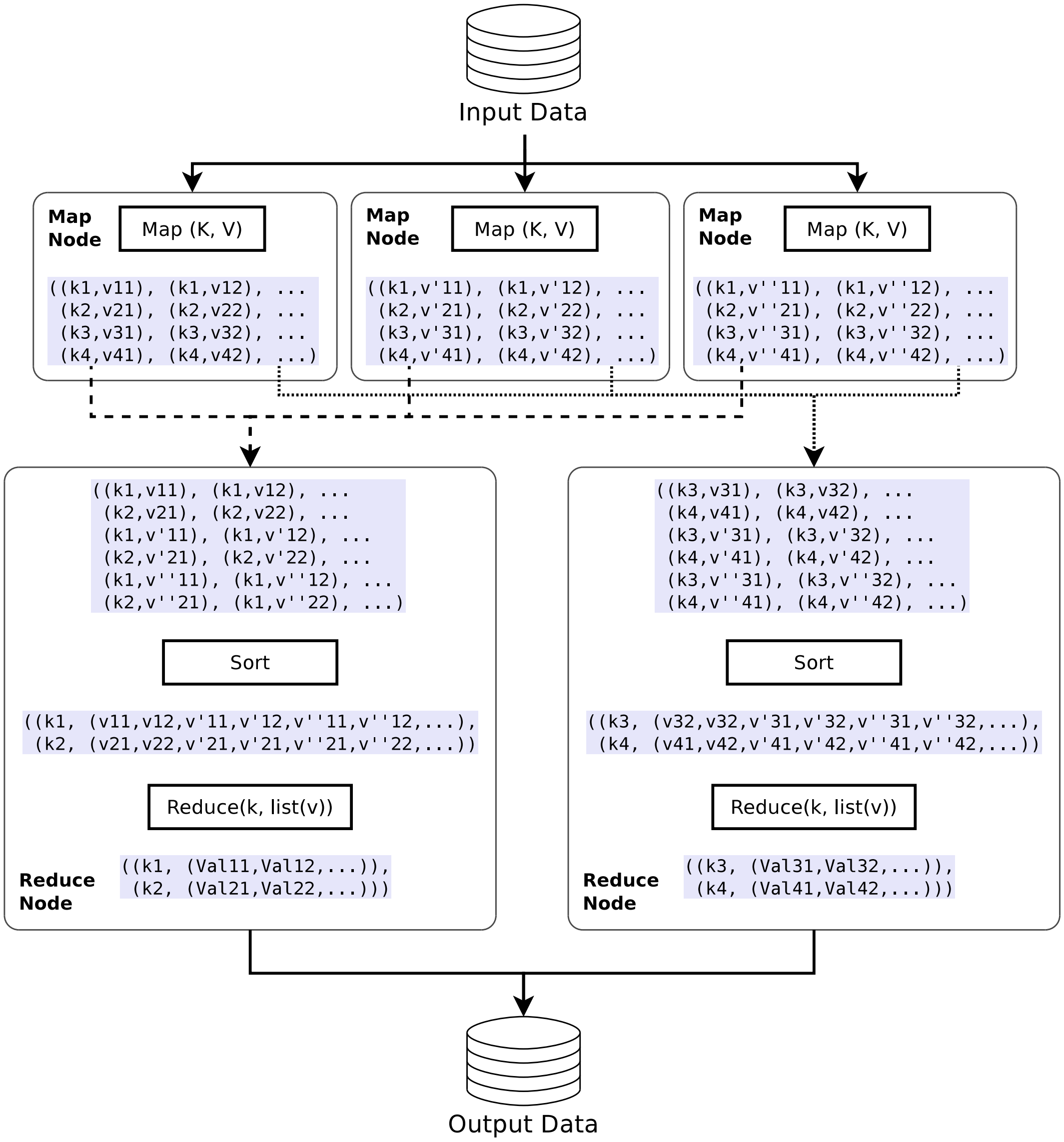

Input Data
Map Node
Map (K, V)
((k1,v11), (k1,v12), ...
(k2,v21), (k2,v22), ...
(k3,v31), (k3,v32), ...
(k4,v41), (k4,v42), ...)
Map Node
Map (K, V)
((k1,v'11), (k1,v'12), ...
(k2,v'21), (k2,v'22), ...
(k3,v'31), (k3,v'32), ...
(k4,v'41), (k4,v'42), ...)
Map Node
Map (K, V)
((k1,v''11), (k1,v''12), ...
(k2,v''21), (k2,v''22), ...
(k3,v''31), (k3,v''32), ...
(k4,v''41), (k4,v''42), ...)
((k1,v11), (k1,v12), ...
(k2,v21), (k2,v22), ...
(k1,v'11), (k1,v'12), ...
(k2,v'21), (k2,v'22), ...
(k1,v''11), (k1,v''12), ...
(k2,v''21), (k1,v''22), ...)
Sort
((k1, (v11,v12,v'11,v'12,v''11,v''12,...),
(k2, (v21,v22,v'21,v'21,v''21,v''22,...))
Reduce(k, list(v))
Reduce Node
((k1, (Val11,Val12,...)),
(k2, (Val21,Val22,...)))
((k3,v31), (k3,v32), ...
(k4,v41), (k4,v42), ...
(k3,v'31), (k3,v'32), ...
(k4,v'41), (k4,v'42), ...
(k3,v''31), (k3,v''32), ...
(k4,v''41), (k4,v''42), ...)
Sort
((k3, (v32,v32,v'31,v'32,v''31,v''32,...),
(k4, (v41,v42,v'41,v'42,v''41,v''42,...))
Reduce(k, list(v))
Reduce Node
((k3, (Val31,Val32,...)),
(k4, (Val41,Val42,...)))
Output Data

Figure 1: Overview of Map-Reduce Sequence

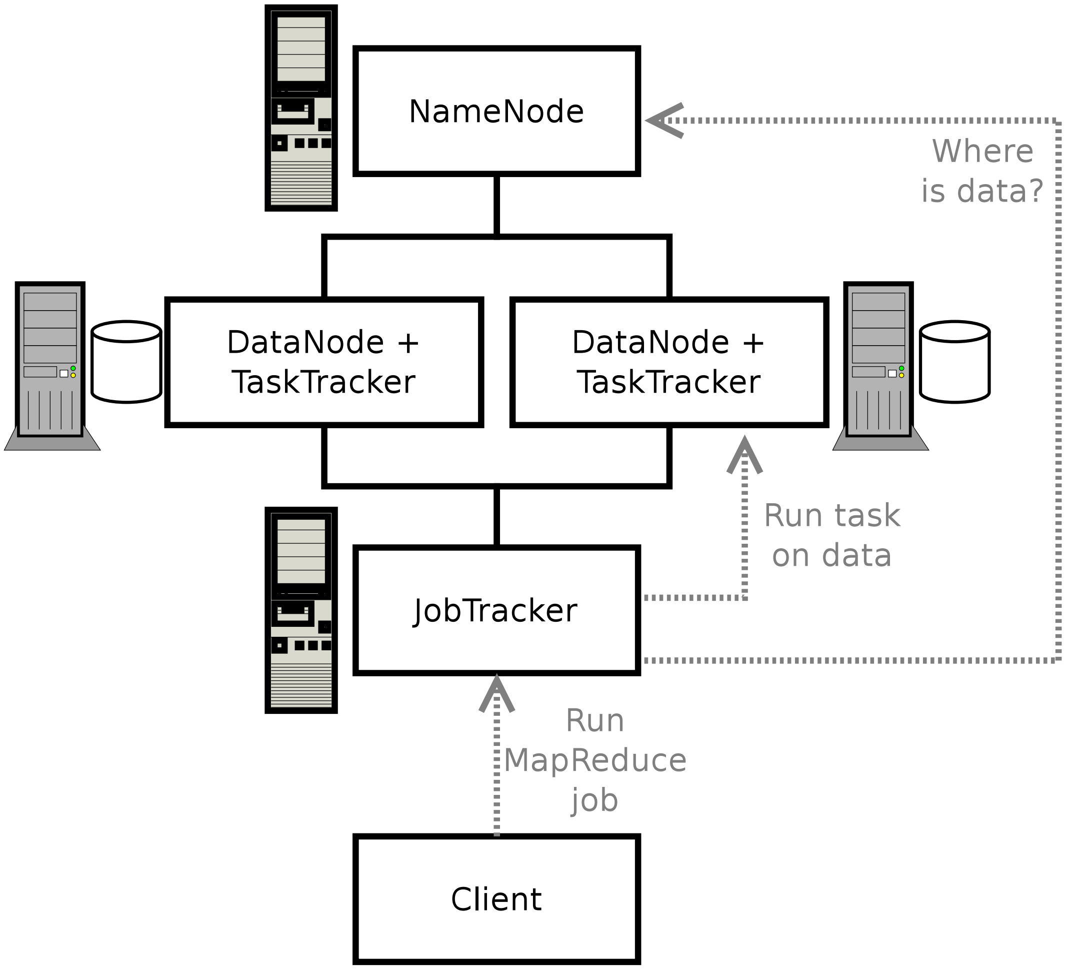


Figure 2: Hadoop Node Types