Data and Algorithms of the Web: MapReduce

Mauro Sozio

March 15, 2015

Outline

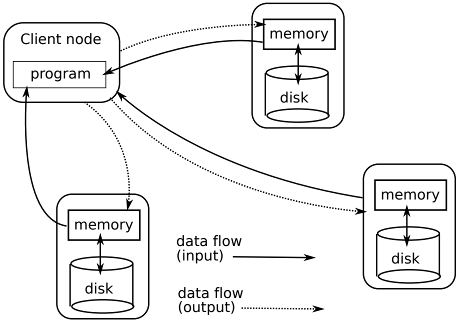
Outline

Data analysis at a large scale

- Very large data collections (TB to PB) stored on distributed filesystems:
  - Query logs
  - Search engine indexes
  - Sensor data
- Need efficient ways for analyzing, reformatting, processing them
- In particular, we want:
  - Parallelization of computation (benefiting of the processing power of all nodes in a cluster)
  - Resilience to failure

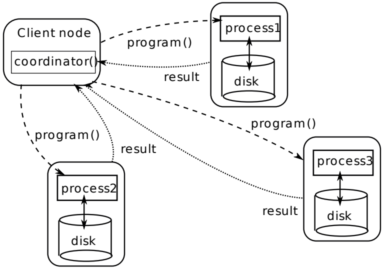
Centralized computing with distributed data storage

Run the program at client node, get data from the distributed system.



Downsides: Significant amount of data is transferred. Cluster computing resources are not used.

Pushing the program near the data



- MapReduce: A programming model (inspired by standard functional programming operators) to facilitate the development and execution of distributed tasks.
- Published by Google Labs in 2004 at OSDI [DG04]. Widely used since then, open-source implementation in Hadoop.

MapReduce in Brief

- The programmer defines the program logic as two functions:

Map	transforms the input into key-value pairs to process
Reduce	aggregates the list of values for each key

- The MapReduce environment takes in charge distribution aspects
- A complex program can be decomposed as a succession of Map and Reduce tasks
- Higher-level languages (Pig, Hive, etc.) help with writing distributed applications

Outline

Three operations on key-value pairs

1. User-defined: $\mathit{map} : (K, V) \rightarrow \text{list}(K', V')$

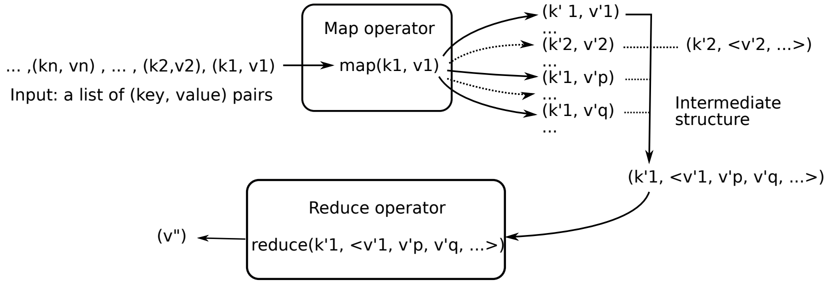
```
function map(docOffset, document)
  while (document.hasNext())
    output (document.nextWord(), 1)
```

2. Seamless to the programmer: $\mathit{shuffle} : \text{list}(K', V') \rightarrow \text{list}(K', \text{list}(V'))$ regroups all intermediate pairs on the key
3. User-defined: $\mathit{reduce} : (K', \text{list}(V')) \rightarrow \text{list}(K'', V'')$

```
function reduce(word, parSums)
  tot=0
  while (parSums.hasNext())
    tot += parSums.next()
  output (word, tot)
```

Job workflow in MapReduce

Important: each pair, at each phase, is processed independently from the other pairs.


... ,(kn, vn) , ... , (k2,v2), (k1, v1)
Input: a list of (key, value) pairs
Map operator
map(k1, v1)
(k' 1, v'1)
...
(k'2, v'2)
...
(k'1, v'p)
...
(k'1, v'q)
...
(k'2, <v'2, ...>)
Intermediate structure
(k'1, <v'1, v'p, v'q, ...>)
Reduce operator
reduce(k'1, <v'1, v'p, v'q, ...>)
(v")

Network and distribution are transparently managed by the MapReduce environment.

Some terminology: job,task,mapper,reducer

- A MapReduce job is a unit of work that the client node seeks to perform. It consists mainly of the input data, the MapReduce program and configuration information.
- A MapReduce program might contain one or several jobs.
- Each job might consist of several *map* and *reduce* tasks.
- Mapper and Reducer: any node that executes either map or reduce tasks, respectively.

Example: term count in MapReduce (input)

URL	Document
u_1	the jaguar is a new world mammal of the felidae family.
u_2	for jaguar, atari was keen to use a 68k family device.
u_3	mac os x jaguar is available at a price of us $199 for apple's new "family pack".
u_4	one such ruling family to incorporate the jaguar into their name is jaguar paw.
u_5	it is a big cat.

Example: term count in MapReduce

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	1
...	

map output
shuffle input

Example: term count in MapReduce

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	1
…	

map output
shuffle input

term	count
jaguar	1,1,1,1
mammal	1
family	1,1,1
available	1
…	

shuffle output
reduce input

Example: term count in MapReduce

term	count
jaguar	1
mammal	1
family	1
jaguar	1
available	1
jaguar	1
family	1
family	1
jaguar	1
. . .	

map output
shuffle input

term	count
jaguar	1,1,1,1
mammal	1
family	1,1,1
available	1
. . .	

shuffle output
reduce input

term	count
jaguar	4
mammal	1
family	3
available	1
. . .	

final output

More on the *map* function

```
function map(docOffset, document)
  while (document.hasNext())
    output (document.nextWord(), 1)
```

where docOffset is the physical address of document (handled by MapReduce). Why we output 1? Input is a *data stream* with *sequential access* (accessed through an Iterator in Java).

More efficient: output partial sums rather than 1's (keep some data in main memory).

Best: use a combiner (more on this later).

More on the *reduce* function

```
function reduce(word, parSums)
  tot=0
  while (parSums.hasNext())
    tot += parSums.next()
  output (word, tot)
```

Each reduce task consists of aggregating partial sums. Note that reduce tasks are independent from each other and can be executed in parallel.

If user requires more than one reducer, then reduce tasks are executed in parallel. E.g. two reducers, each taking care of 50% of words (ideal case).

Otherwise no parallelism in the reduce tasks.

A MapReduce cluster

Nodes inside a MapReduce cluster can be classified as follows:

- A jobtracker acts as a master node. It coordinates all the jobs run on the system, scheduling tasks to run on tasktrackers. It also keeps a record of the overall progress of each job. If a task fails, the job tracker can reschedule it.
- Several tasktrackers run the computation itself, i.e., *map* and *reduce* tasks and send progress report to the jobtracker.
- Tasktrackers usually also act as data nodes of a distributed filesystem (e.g., HDFS)

+ a client node where the application is launched.

Processing a MapReduce job

- the input is divided into fixed-size pieces called splits (default size:64MB).
- MapReduce assigns one map task for each split, which runs the user-defined map function (assignment is based on the data locality principle).
- Map output is stored on a local disk (not HDFS), as it can be thrown away after reducers processed it.

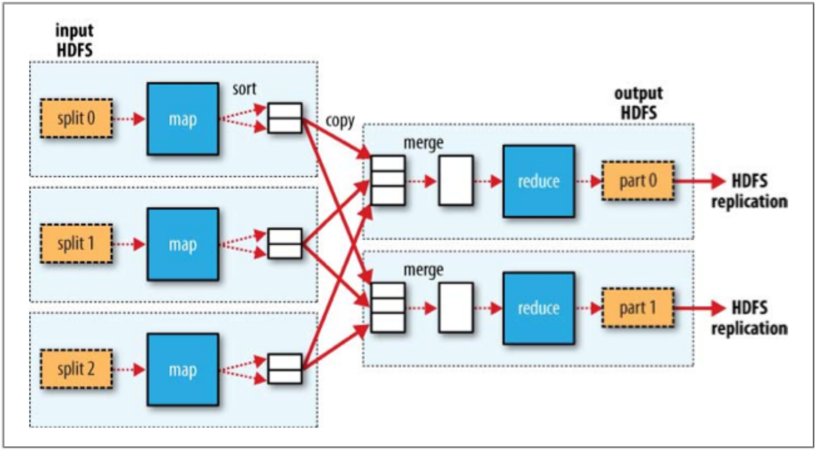
Remarks:

- Data locality no longer hold for reducers, since they read from the mappers.
- Split size can be changed by the user. Small splits implies more parallelism, however, too many splits entails too much overhead in managing them.

Facts about mappers and reducers

- The number of mappers is function of the number of map tasks and the number of available nodes in the cluster.
- Assignment of data splits to mappers tries optimizing data locality: the mapper node in charge of a split is, if possible, one that stores a replica of this split.
- The number of reducers is set by the user.
- Map output is assigned to reducers by hashing of the key, usually uniformly at random to balance the load. No data locality possible.

Distributed execution of a MapReduce job.



Failure management

Since tasks are distributed over thousands of machines, faulty machines are relatively common. Starting the job from the beginning is not a valid option.

The jobtracker periodically checks the availability and reachability of the tasktrackers (heartbeats) and whether *map* or *reduce* jobs make any progress

1. if a reducer fails, its task is reassigned to another tasktracker; this usually requires restarting map tasks as well.
2. if a mapper fails, its task is reassigned to another tasktracker
3. if the jobtracker fails, the whole job should be re-executed

Adapting algorithms into MapReduce

- Prefer simple *map* and *reduce* functions
- try to balance computational load across the machines (reducers). Easier when computing matrix-vector multiplications.
- iterative algorithms: prefer small number of iterations. The output of each iteration must be stored in the HDFS $\rightarrow$ large I/O overhead per iteration.

A given application may have:

- many mappers and reducers (parallelism in HDFS I/O and computing).
- only one reducer.
- zero reducers (e.g. input preprocessing, filtering, . . .).

Outline

Combiners

- A mapper task can produce a large number of pairs with the same key (e.g. (jaguar,1)), which need to be sent over the network to reducers: costly.
- A combiner combine these pairs into a single key-value pair

Example

(jaguar,1), (jaguar, 1), (jaguar, 1), (jaguar, 2)$\rightarrow$(jaguar, 5)

- *combiner* : list$(V')\rightarrow V'$ function executed (possibly several times) to combine the values for a given key, on *a mapper node*.
- No guarantee that the *combiner* is called
- Easy case: the combiner is the same as the *reduce* function. Possible when the aggregate function computed by *reduce* is commutative and associative.

Compression

- Data transfers over the network:
  - From datanodes to mapper nodes (usually reduced using data locality)
  - From mappers to reducers
  - From reducers to datanodes to store the final output
- Each of these can benefit from data compression
- Tradeoff between volume of data transfer and (de)compression time
- Usually, compressing *map* outputs using a fast compressor increases efficiency

Optimizing the *shuffle* operation

- Sorting of pairs (needed to assign data to reducers) can be costly.
- Sorting much more efficient in memory than on disk
- Increasing the amount of memory available for *shuffle* operations can greatly increase the performance
- . . . at the downside of less memory available for *map* and *reduce* tasks.

Outline

Hadoop

- Open-source software, Java-based, managed by the Apache foundation, for large-scale distributed storage and computing
- Originally developed for Apache Nutch (open-source Web search engine), a part of Apache Lucene (text indexing platform)
- Open-source implementation of GFS and Google's MapReduce
- Yahoo!: a main contributor of the development of Hadoop
- Hadoop components:
  - Hadoop filesystem (HDFS)
  - MapReduce
  - Pig (data exploration), Hive (data warehousing): higher-level languages for describing MapReduce applications
  - HBase: column-oriented distributed DBMS
  - ZooKeeper: coordination service for distributed applications

Hadoop programming interfaces

- Different APIs to write Hadoop programs:
  - A rich Java API (main way to write Hadoop programs)
  - A Streaming API that can be used to write *map* and *reduce* functions in any programming language (using standard inputs and outputs)
  - A C++ API (Hadoop Pipes)
  - With a higher-language level (e.g., Pig, Hive)
- Advanced features only available in the Java API
- Two different Java APIs depending on the Hadoop version; presenting the "old" one

Testing and executing a Hadoop job

- Required environment:
  - JDK on client
  - JRE on all Hadoop nodes
  - Hadoop distribution (HDFS + MapReduce) on client and all Hadoop nodes
  - SSH servers on each tasktracker, SSH client on jobtracker (used to control the execution of tasktrackers)
  - An IDE (e.g., Eclipse + plugin) on client
- Three different execution modes:

local	One mapper, one reducer, run locally from the same JVM as the client
pseudo-distributed	mappers and reducers are launched on a single machine, but communicate over the network
distributed	over a cluster for real runs

Debugging MapReduce

- Easiest: debugging in local mode
- Web interface with status information about the job
- Standard output and error channels saved on each node, accessible through the Web interface
- Counters can be used to track side information across a MapReduce job (e.g., number of invalid input records)
- Remote debugging possible but complicated to set up (impossible to know in advance where a *map* or *reduce* task will be executed)
- IsolationRunner allows to run in isolation part of the MapReduce job

Hadoop in the cloud

- Possibly to set up one's own Hadoop cluster
- But often easier to use clusters in the cloud that support MapReduce:
  - Amazon EC2
  - Cloudera
  - etc.
- Not always easy to know the cluster's configuration (in terms of racks, etc.) when on the cloud, which hurts data locality in MapReduce

Outline

MapReduce limitations (1/2)

- High latency. Launching a MapReduce job has a high overhead, and *reduce* functions are only called after all *map* functions succeed, not suitable for applications needing a quick result.
- Batch processing only. MapReduce excels at processing a large collection, not at retrieving individual items from a collection.
- Write-once, read-many mode. No real possibility of updating a dataset using MapReduce, it should be regenerated from scratch

MapReduce limitations (2/2)

- Relatively low-level. Ongoing efforts for more high-level languages: Scope [CJL+08], Pig [ORS+08, GNC+09], Hive [TSJ+09], Cascading `http://www.cascading.org/`
- No structure. Implies lack of indexing, difficult to optimize, etc. [DS87]
- Hard to tune. Number of reducers? Compression? Memory available at each node? etc.
- Recent projects: Pregel [MAB+10] for graph data processing (Giraph open-source implementation), Spark [ZCF+10] (maybe the most efficient but perhaps still difficult to use).

Resources

- Original description of the MapReduce framework [DG04]
- Hadoop distribution and documentation available at `http://hadoop.apache.org/`
- Documentation for Pig is available at `http://wiki.apache.org/pig/`
- Excellent textbook on Hadoop [Whi09]

References I

Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou.
SCOPE: easy and efficient parallel processing of massive data sets.
Proc. Intl. Conf. on Very Large Databases (VLDB), 1(2):1265–1276, 2008.

Jeffrey Dean and Sanjay Ghemawat.
MAPREDUCE: Simplified Data Processing on Large Clusters.
In *Intl. Symp. on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

D. DeWitt and M. Stonebraker.
MAPREDUCE, a major Step Backward.
DatabaseColumn blog, 1987.
http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

References II

Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava.
Building a HighLevel Dataflow System on top of MAPREDUCE: The PIG Experience.
Proceedings of the VLDB Endowment (PVLDB), 2(2):1414–1425, 2009.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski.
Pregel: a system for large-scale graph processing.
In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

References III

- Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins.
  Pig latin: a not-so-foreign language for data processing.
  In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 1099–1110, 2008.
- Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy.
  Hive - A Warehousing Solution Over a Map-Reduce Framework.
  Proceedings of the VLDB Endowment (PVLDB), 2(2):1626–1629, 2009.
- Tom White.
  Hadoop: The Definitive Guide.
  O'Reilly, Sebastopol, CA, USA, 2009.

References IV

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica.
Spark: cluster computing with working sets.
In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.