

Data acquisition, extraction, and storage

Distributed Computing with MapReduce and Beyond

Pierre Senellart



22 November 2024



Outline

Distributed Data Systems

Distributed Data Management

Distributed File Systems (GFS/HDFS)

MapReduce

Limitations of MapReduce

Alternative Distributed Computation Models



Distributed systems

A **distributed system** is an application that coordinates the actions of several computers to achieve a specific task.

This coordination is achieved by exchanging **messages** which are pieces of data that convey some information.

⇒ “shared-nothing” architecture → no shared memory, no shared disk.

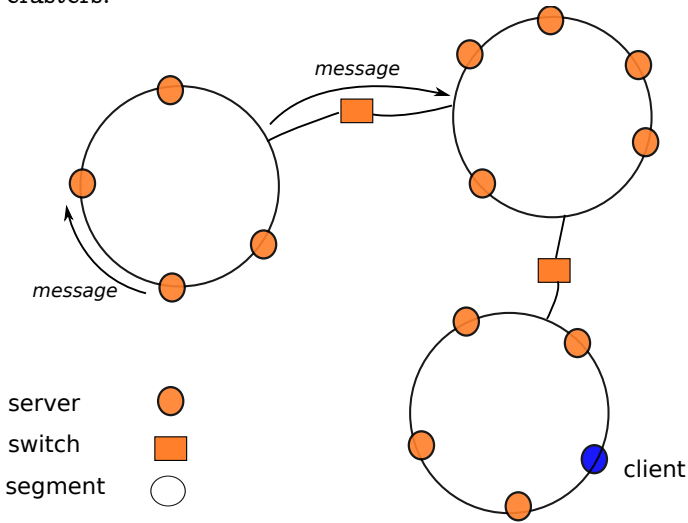
The system relies on a network that connects the computers and handles the routing of messages.

⇒ Local area networks (LAN), Peer to peer (P2P) networks...

Client (nodes) and **Server** (nodes) are communicating **software** components: we assimilate them with the machines they run on.

LAN-based infrastructure: clusters of machines

Three communication levels: “racks”, clusters, and groups of clusters.





Example: data centers

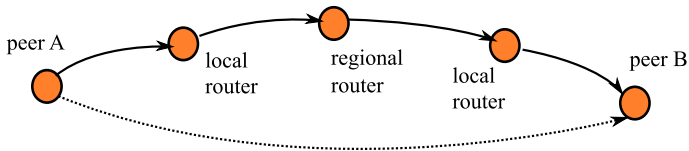
Typical setting of a Google data center.

1. ≈ 40 servers per rack;
2. ≈ 150 racks per data center (cluster);
3. $\approx 6,000$ servers per data center;
4. how many clusters? Google's secret, and constantly evolving ...

Rough estimate: 150-200 data centers? 1,000,000 servers?

P2P infrastructure: Internet-based communication

Nodes, or “peers” communicate with messages sent over the Internet network.



The physical route may consist of 10 or more forwarding messages, or “hops”.

Suggestion: use the traceroute utility to check the route between your laptop and a Web site of your choice.



Performance

| Type | Latency | Bandwidth |
|----------|------------------------------|----------------------------------|
| Disk | $\approx 5 \times 10^{-3}$ s | At best 200 MB/s |
| SSD | $\approx 10^{-4}$ s | 200–500 MB/s |
| LAN | $\approx 10^{-3}$ s | 100 Mb/s to 10Gb/s (single-rack) |
| Internet | 10–100 ms (highly variable) | A few MB/s (highly variable) |

Bottom line (1): it can be one order of magnitude faster to exchange main memory data between 2 machines in the same rack of a data center, than to read on the disk.

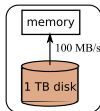
Bottom line (2): exchanging through the Internet is slow and unreliable with respect to LANs.

Distribution, why?

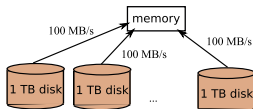
Sequential access. It takes a couple of hours to read a 1 TB disk.

Parallel access. With 100 disks, assuming that the disks work in parallel and sequentially: about 1 minute.

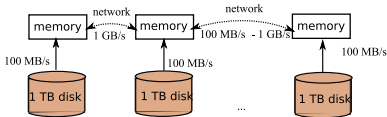
Distributed access. With 100 computers, each disposing of its own local disk: each CPU processes its own dataset.



a. Single CPU, single disk



b. Parallel read: single CPU, many disks



c. Distributed reads: an extendible set of servers

The latter solution is **scalable**, by adding new computing resources.



Performance of data-centric distr. systems

1. disk transfer rate is a bottleneck for large scale data management; parallelization and distribution of the data on many machines is a means to eliminate this bottleneck;
2. *write once, read many*: a distributed storage system is appropriate for large files that are written once and then repeatedly scanned;
3. *data locality*: bandwidth is a scarce resource, and program should be “pushed” near the data they must access to.

A distr. system also gives an opportunity to reinforce the security of data with **replication**.



Outline

Distributed Data Systems

Distributed Data Management

Distributed File Systems (GFS/HDFS)

MapReduce

Limitations of MapReduce

Alternative Distributed Computation Models



History and development of GFS

Google File System, a paper published in 2003 by Google Labs at OSDI.

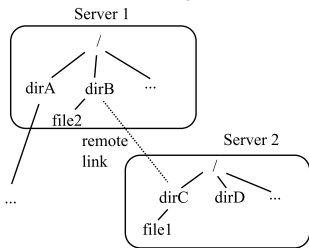
Explains the design and architecture of a distributed system apt at serving very large data files; internally used by Google for storing documents collected from the Web.

Open Source versions have been developed at once: Hadoop File System (HDFS), and Kosmos File System (KFS).

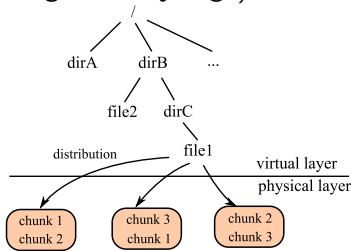
The problem

Why do we need a distributed file system in the first place?

Fact: standard NFS (left part) does not meet scalability requirements (what if file1 gets really big?).



A traditional network file system

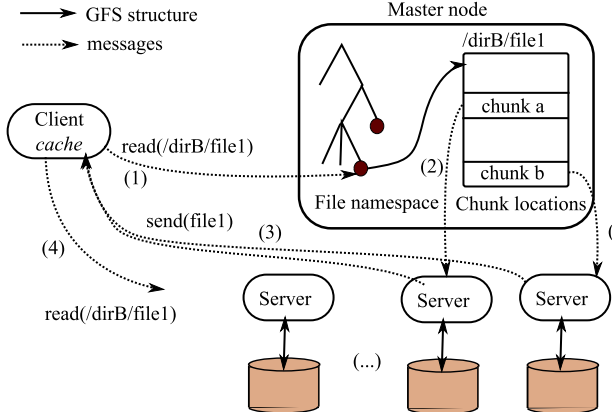


A large scale distributed file system

Right part: GFS/HDFS storage, based on (i) a virtual file namespace, and (ii) partitioning of files in “chunks”.

Architecture

A **Master node** performs administrative tasks, while **servers** store “chunks” and send them to Client nodes.



The Client maintains a **cache** with chunks locations, and directly communicates with servers.

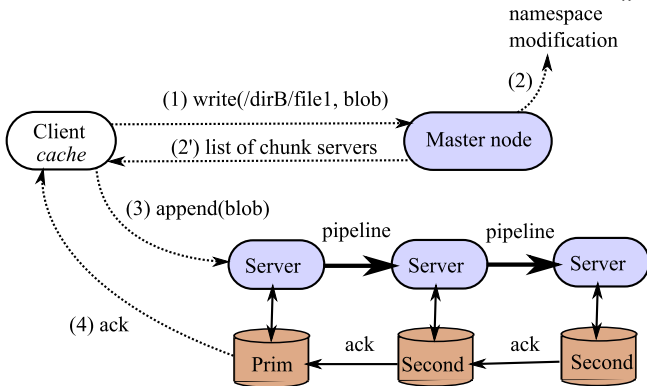


Technical details

- The architecture works best for very large files (e.g., several Gigabytes), divided in large (64-128 MBs) chunks.
⇒ this limits the metadata information served by the Master.
- Each server implements recovery and replication techniques (default: 3 replicas).
- (**Availability**) The Master sends heartbeat messages to servers, and initiates a replacement when a failure occurs.
- (**Scalability**) The Master is a potential single point of failure; its protection relies on distributed recovery techniques for all changes that affect the file namespace.

Workflow of a *write()* operation (simplified)

The following figure shows a non-concurrent *append()* operation.



Write (append) in GFS (simplified to non-concurrent operations)

In case of concurrent appends to a chunk, the primary replica assigns serial numbers to the mutation, and coordinates the secondary replicas.



Outline

Distributed Data Systems

MapReduce

Introduction

The MapReduce Computing Model

MapReduce Optimization

MapReduce in Hadoop

Limitations of MapReduce

Alternative Distributed Computation Models

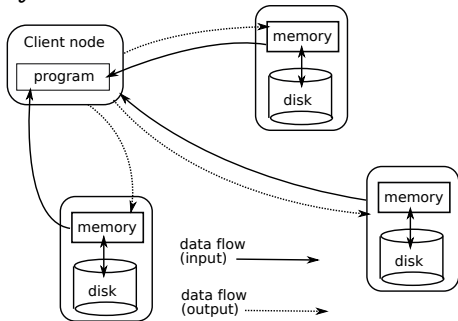


Data analysis at very large scale

- **Very large** data collections (dozen of TBs to EB) stored on distributed filesystems:
 - Crawl data
 - 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

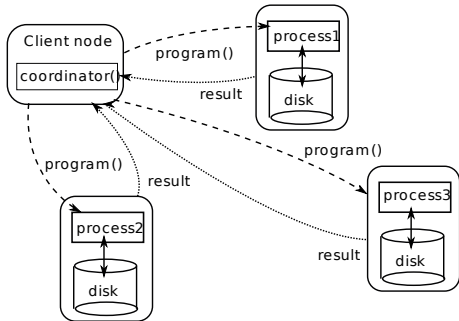
Centralized computing with distributed data storage

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

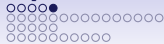


Downsides: important data flows, no use of the cluster computing resources.

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 (Cascading, Hive, etc.) help with writing distributed applications



Outline

Distributed Data Systems

MapReduce

Introduction

The MapReduce Computing Model

MapReduce Optimization

MapReduce in Hadoop

Limitations of MapReduce

Alternative Distributed Computation Models



Three operations on key-value pairs

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

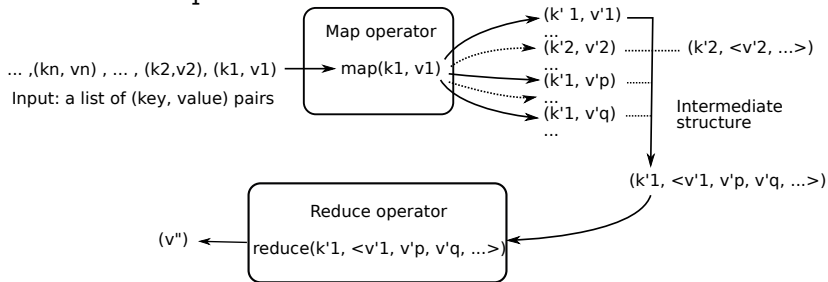
```
function map(uri, document)
  foreach distinct term in document
    output (term, count(term, document))
```

2. Fixed behavior: $shuffle : list(K', V') \rightarrow list(K', list(V'))$
regroups all intermediate pairs on the key
3. User-defined: $reduce : (K', list(V')) \rightarrow list(K'', V'')$

```
function reduce(term, counts)
  output (term, sum(counts))
```

Job workflow in MapReduce

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



Network and distribution are transparently managed by the MapReduce environment.



Example: term count in MapReduce (input)

| URL | Document |
|-----|----------|
|-----|----------|

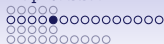
| | |
|-------|---|
| u_1 | the jaguar is a new world mammal of the felidae family. |
|-------|---|

| | |
|-------|--|
| 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". |
|-------|--|

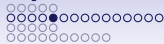
| | |
|-------|---|
| u_4 | one such ruling family to incorporate the jaguar into their name is jaguar paw. |
|-------|---|

| | |
|-------|------------------|
| 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 | 2 |
| ... | |
| <hr/> | |
| <i>map</i> output | |
| <i>shuffle</i> 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 | 2 |
| ... | |

map output

shuffle input

| term | count |
|-----------------------|---------|
| jaguar | 1,1,1,2 |
| mammal | 1 |
| family | 1,1,1 |
| available | 1 |
| ... | |
| <i>shuffle</i> output | |
| <i>reduce</i> 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 | 2 |
| ... | |

map output
shuffle input

| term | count |
|-----------|---------|
| jaguar | 1,1,1,2 |
| mammal | 1 |
| family | 1,1,1 |
| available | 1 |
| ... | |

shuffle output
reduce input

| term | count |
|-----------|-------|
| jaguar | 5 |
| mammal | 1 |
| family | 3 |
| available | 1 |
| ... | |

final output

Example: simplification of the *map*

```
function map(uri, document)
  foreach distinct term in document
    output (term, count(term, document))
```

can actually be further simplified:

```
function map(uri, document)
  foreach term in document
    output (term, 1)
```

since all counts are aggregated.

Might be less efficient though (we may need a **combiner**, see further)



A MapReduce cluster

Nodes inside a MapReduce cluster are decomposed as follows:

- A **jobtracker** acts as a master node; MapReduce jobs are submitted to it
- Several **tasktrackers** run the computation itself, i.e., *map* and *reduce* tasks
- A given tasktracker may run several tasks in parallel
- Tasktrackers usually also act as **data nodes** of a distributed filesystem (e.g., GFS, HDFS)

+ a client node where the application is launched.



Processing a MapReduce job

A MapReduce **job** takes care of the distribution, synchronization and failure handling. Specifically:

- the input is split into M groups; each group is assigned to a **mapper** (assignment is based on the data locality principle)
- each mapper processes a group and stores the intermediate pairs locally
- grouped instances are assigned to **reducers** thanks to a hash function
- (*shuffle*) intermediate pairs are sorted on their key by the reducer
- one obtains grouped instances, submitted to the *reduce* function

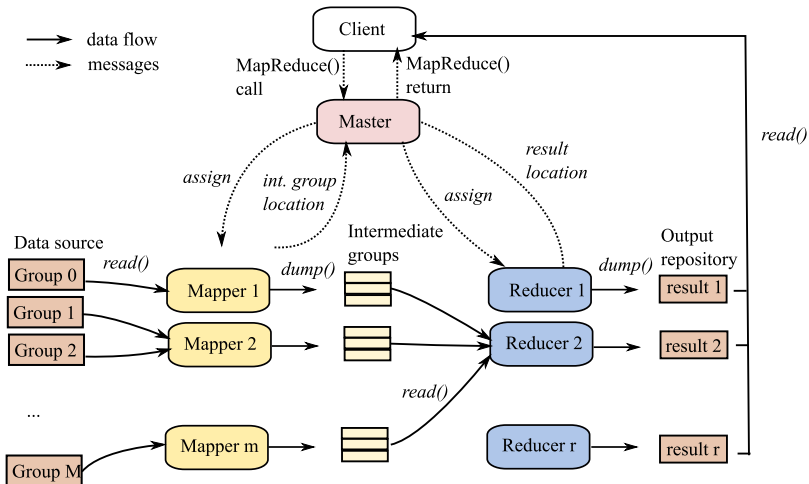
Remark: the data locality does no longer hold for the *reduce* phase, since it reads from the mappers.



Assignment to reducer and mappers

- Each mapper task processes **a fixed amount of data (split)**, usually set to the distributed filesystem block size (e.g., 64 MB)
- The number of mapper nodes is function of the number of mapper tasks and the number of available nodes in the cluster: each mapper nodes can process (in parallel and sequentially) **several mapper tasks**
- Assignment to mapper tries optimizing **data locality**: the mapper node in charge of a split is, if possible, one that stores a replica of this split (or if not possible, a node of the same rack)
- The number of reducer tasks is **set by the user**
- Assignment to reducers is done through a hashing of the key, usually **uniformly at random**; no data locality possible

Distributed execution of a MapReduce job.





Processing the term count example

Let the input consists of documents, say, one billion 100-terms documents of approximately 1 KB each.

Assume the split operation distributes these documents in groups of 64 MB: each group consists of 64,000 documents. Therefore $M = \lceil 1\,000\,000\,000 / 64\,000 \rceil \approx 16\,000$ groups.

If there are 1 000 mapper nodes, each node processes on average 16 splits.

If there are 1 000 reducers, each reducer r_i processes all key-value pairs for terms t such that $hash(t) = i$ ($1 \leq i \leq 1\,000$)



Processing the term count example (2)

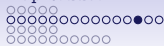
Assume that $hash('call') = hash('mine') = hash('blog') = i = 100$. We focus on three Mappers m_p , m_q and m_r :

1. $G_i^p = (\langle \dots, ('mine', 1), \dots, ('call', 1), \dots, ('mine', 1), \dots, ('blog', 1) \dots \rangle$
2. $G_i^q = (\langle \dots, ('call', 1), \dots, ('blog', 1), \dots \rangle$
3. $G_i^r = (\langle \dots, ('blog', 1), \dots, ('mine', 1), \dots, ('blog', 1), \dots \rangle$

r_i reads G_i^p , G_i^q and G_i^r from the three Mappers, sorts their unioned content, and groups the pairs with a common key:

$\dots, ('blog', \langle 1, 1, 1, 1 \rangle), \dots, ('call', \langle 1, 1 \rangle), \dots,$
 $('mine', \langle 1, 1, 1 \rangle)$

Our *reduce* function is then applied by r_i to each element of this list. The output is $('blog', 4)$, $('call', 2)$ and $('mine', 3)$

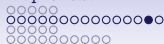


Failure management

In case of failure, because the tasks are distributed over hundreds or thousands of machines, the chances that a problem occurs somewhere are much larger; starting the job from the beginning is not a valid option.

The Master 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**, intermediate groups are sent over from mappers
2. if a mapper fails, its task is **reassigned to another tasktracker**
3. if the jobtracker fails, **the whole job should be re-initiated**

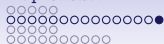


Joins in MapReduce

Two datasets, A and B that we need to join for a MapReduce task

- If one of the dataset is small, it can be **sent over fully** to each tasktracker and exploited inside the *map* (and possibly *reduce*) functions
- Otherwise, each dataset should be **grouped according to the join key**, and the result of the join can be computing in the *reduce* function

Not very convenient to express in MapReduce. Much easier in higher-level languages.



Using MapReduce for solving a problem

- Prefer:
 - **Simple** *map* and *reduce* functions
 - Mapper tasks processing **large data chunks** (at least the size of distributed filesystem blocks)
- A given application may have:
 - **A chain of *map* functions** (input processing, filtering, extraction...)
 - A sequence of **several *map-reduce* jobs**
 - **No *reduce* task** when everything can be expressed in the *map* (zero reducers, or the identity reducer function)
- Not the right tool for everything (see further)



Outline

Distributed Data Systems

MapReduce

Introduction

The MapReduce Computing Model

MapReduce Optimization

MapReduce in Hadoop

Limitations of MapReduce

Alternative Distributed Computation Models



Combiners

- A mapper task can produce a large number of pairs with the same key
- They need to be sent over the network to the reducer:
costly
- It is often possible to **combine** these pairs into a single key-value pair
(jaguar,1), (jaguar, 1), (jaguar, 1), (jaguar, 2) → (jaguar, 5)
- *combiner* : $\text{list}(V') \rightarrow V'$ 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 α computed by *reduce* is **distributive**: $\alpha(k_1, \alpha(k_2, k_3)) = \alpha(k_1, k_2, k_3)$



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**
- **Trade-off** 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 on each reducer, to compute the groups:
costly operation
- 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 (but usually not much needed)



Speculative execution

- The MapReduce jobtracker tries detecting tasks that take longer than usual (e.g., because of hardware problems)
- When detected, such a task is **speculatively** executed on another tasktracker, without killing the existing task
- Eventually, when one of the attempts succeeds, the other one is killed



Outline

Distributed Data Systems

MapReduce

Introduction

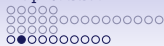
The MapReduce Computing Model

MapReduce Optimization

MapReduce in Hadoop

Limitations of MapReduce

Alternative Distributed Computation Models



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



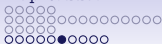
Hadoop programming interfaces

- 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., Hive, Cascading)
- Advanced features only available in the Java API, but the streaming API is good enough for most uses

Python *map* for the term count example

```
import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print('%s\t%s' % (word, 1))
```



Python *reduce* for the term count example

```
import sys

c = dict()

for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    count = int(count)
    if word in c:
        c[word] += count
    else:
        c[word] = count

for w in c:
    print('%s\t%s' % (w, c[w]))
```



Command line to run the streaming API

```
hadoop jar hadoop-streaming-*.jar \  
-files mapper.py, reducer.py \  
-mapper mapper.py \  
-reducer reducer.py \  
-input 'input_directory/*' \  
-output output_directory
```



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 local 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)



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 EMR
 - Google Cloud Dataproc
 - Cloudera CDH
 - 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

Distributed Data Systems

MapReduce

Limitations of MapReduce

Alternative Distributed Computation Models



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
- **No transactions.** No concurrency control at all, completely unsuitable for transactional applications [PPR⁺09].



MapReduce limitations (2/2)

- **Relatively low-level.** See Hive, Spark, etc. for higher-level languages
- **No structure.** Implies lack of indexing, difficult to optimize, etc. [DS87]
- **Hard to tune.** Number of reducers? Compression? Memory available at each node? etc.



What you should remember on distributed computing

MapReduce is a simple model for **batch processing** of very large collections.

⇒ **good** for data analytics; **not good** for point queries (high latency).

The systems brings **robustness against failure** of a component and **transparent distribution and scalability**.

⇒ more expressive languages required



Outline

Distributed Data Systems

MapReduce

Limitations of MapReduce

Alternative Distributed Computation Models

Apache Hive: SQL Analytics on MapReduce

Apache Storm: Real-Time Computation

Apache Spark: More Complex Workflows

Pregel, Apache Giraph, GraphLab: Think as a Vertex



Apache Hive

- Data warehousing on top of Hadoop
- SQL-like language to express high-level queries, esp., aggregate and analytics
- Queries translated into Map-Reduce jobs (or Spark jobs, see further)
- Geared towards analytics vs transformation of datasets



Term count in Hive

```
CREATE TABLE doc(text STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\n'
STORED AS TEXTFILE;

LOAD DATA INPATH 'hdfs://input.txt'
  INTO TABLE doc;

SELECT word, COUNT(*)
FROM doc
  LATERAL VIEW EXPLODE(SPLIT(text, '␣')) temp
  AS word
GROUP BY word;
```



Outline

Distributed Data Systems

MapReduce

Limitations of MapReduce

Alternative Distributed Computation Models

Apache Hive: SQL Analytics on MapReduce

Apache Storm: Real-Time Computation

Apache Spark: More Complex Workflows

Pregel, Apache Giraph, GraphLab: Think as a Vertex



Apache Storm

- Real-time analytics, in opposition to the batch model of MapReduce
- Achieves very reasonable latency
- Process data in a streaming fashion
- Based on the notion of event producer (spout) and manipulation (bolt)
- Written in Clojure (Lisp) + Java, jobs usually written in Java



Term count in Storm

```
Config config = new Config();
config.put("inputFile", args[0]);
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("line-reader-spout",
                new LineReaderSpout());
builder.setBolt("word-spitter", new WordSplitterBolt()).
    shuffleGrouping("line-reader-spout");
builder.setBolt("word-counter", new WordCounterBolt()).
    shuffleGrouping("word-spitter");
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("HelloStorm", config,
                      builder.createTopology());
```

Only the driver, the spouts and the bolts also need to be defined!



Outline

Distributed Data Systems

MapReduce

Limitations of MapReduce

Alternative Distributed Computation Models

Apache Hive: SQL Analytics on MapReduce

Apache Storm: Real-Time Computation

Apache Spark: More Complex Workflows

Pregel, Apache Giraph, GraphLab: Think as a Vertex



Apache Spark

- High-level language with a dataflow of operators
- Complex programs can be written in Scala, Java, or Python
- Jobs are not translated to MapReduce, but executed directly in a distributed fashion
- Based on **RDDs** (Resilient Distributed Dataset) that can be HDFS files, HBase tables, or the result of applying a succession of operators on these
- Contrarily to MapReduce, local workers have the ability to keep data in memory in a succession of tasks
- Extensions allowing to perform streaming data processing, to process Hive SQL queries
- MLLib: machine learning library on top of Spark



Term count in Spark (Python)

```
file = spark.textFile("hdfs://input.txt")
counts = file \
    .flatMap(lambda line: line.split("_")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://output.txt")
```



Outline

Distributed Data Systems

MapReduce

Limitations of MapReduce

Alternative Distributed Computation Models

Apache Hive: SQL Analytics on MapReduce

Apache Storm: Real-Time Computation

Apache Spark: More Complex Workflows




Pregel, Apache Giraph, GraphLab: Think as a Vertex



Graph Computation Frameworks

- Parallel computation on graph-like data (Web graph, social networks, transportation networks, etc.)
- Pregel: original system by Google
- Apache Giraph: open-source clone of Pregel
- GraphLab: similar goals, different architecture
- One writes **vertex programs**: each vertex receives messages and sends messages to its neighbours
- Pregel and Giraph are based on the **bulk synchronous parallel** paradigm: synchronization barrier once vertex programs are executed on every vertex
- GraphLab uses an **asynchronous** model

References I

-  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/>.
-  Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker.
A comparison of approaches to large-scale data analysis.
In *Proc. ACM Intl. Conf. on the Management of Data (SIGMOD)*, pages 165–178, 2009.