Research at Google

Lecture: The Google MapReduce

http://research.google.com/archive/mapreduce.html

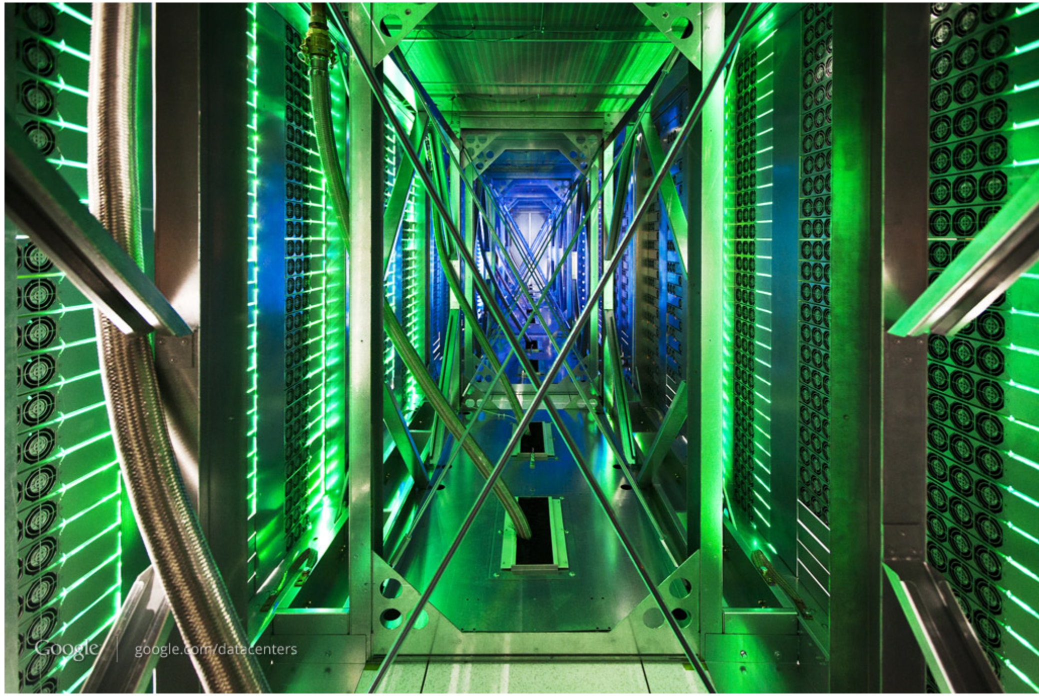
10/03/2014

Romain Jacotin

romain.jacotin@orange.fr

Agenda

- **Introduction**
- Programming model
- Implementation
- Refinements
- Performance
- Experience
- Conclusions



Introduction

Abstract

- MapReduce is a programming model and an associated implementation for processing and generating large data sets

- **Users specify a Map function** that processes a key/value pair to generate a set of intermediate key/value pairs, **and a Reduce function** that merges all intermediate values associated with the same intermediate key

- Programs written in this functional style are **automatically parallelized** and **executed on a large cluster of commodity machines**

- The run-time system take care of the details of partitioning of the input data, scheduling the program's execution

Introduction

MapReduce

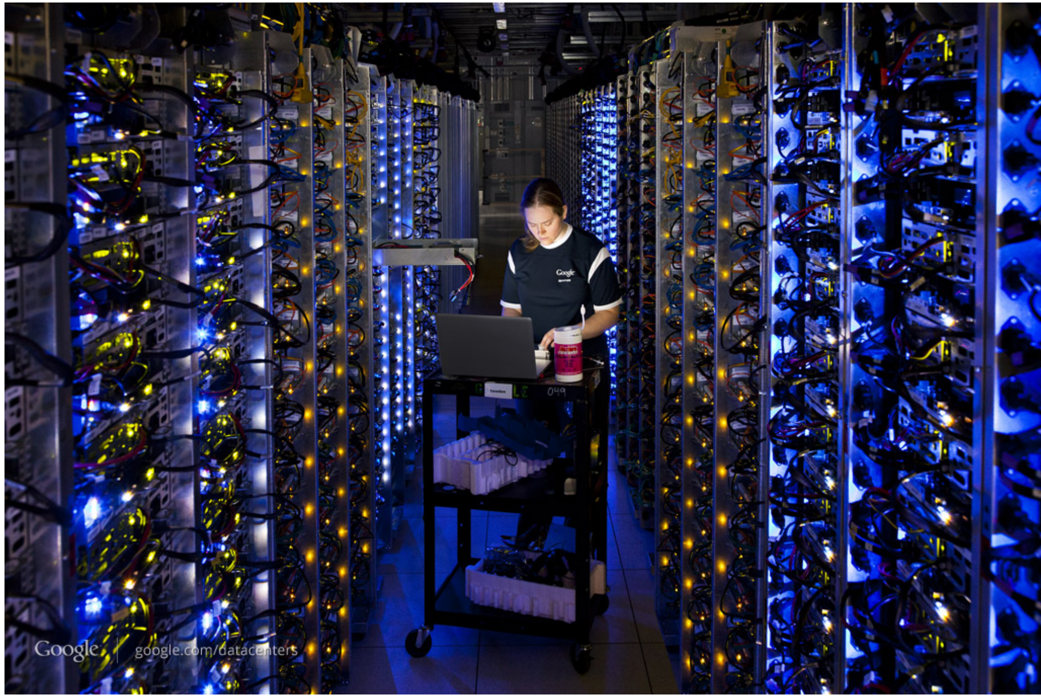
- Google design this abstraction to allow them to express the simple computations they want to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library

- This abstraction is inspired by the "***map***" and "***reduce***" primitives present in LISP

- **The major contribution of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations**

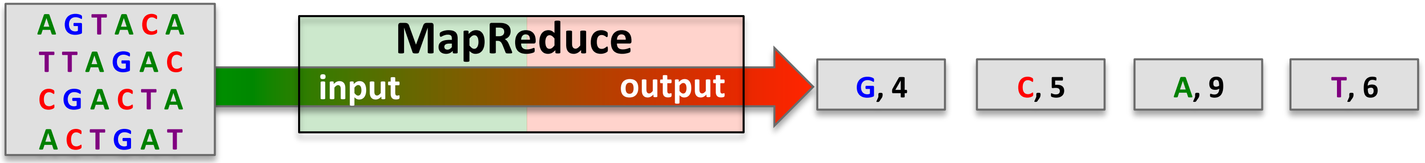
Agenda

- Introduction
- **Programming model**
- Implementation
- Refinements
- Performance
- Experience
- Conclusions

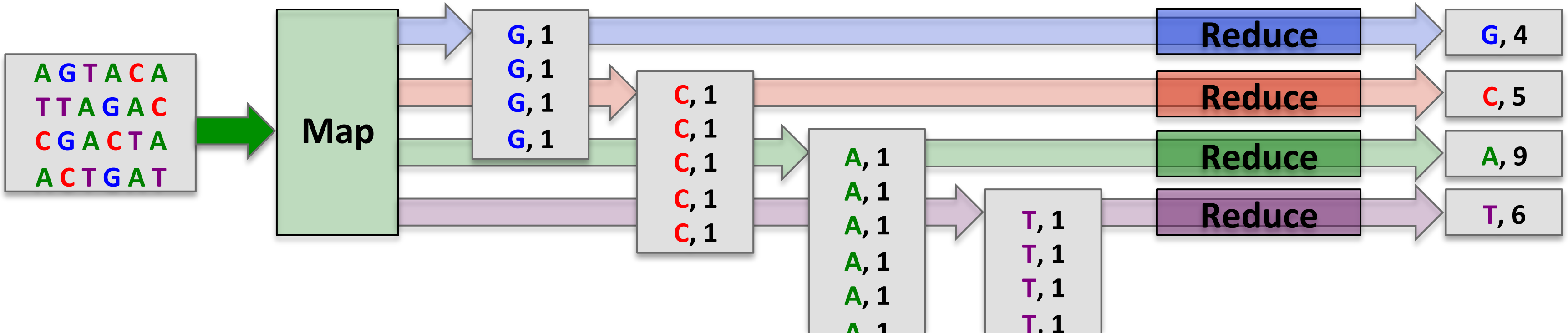


Programming model

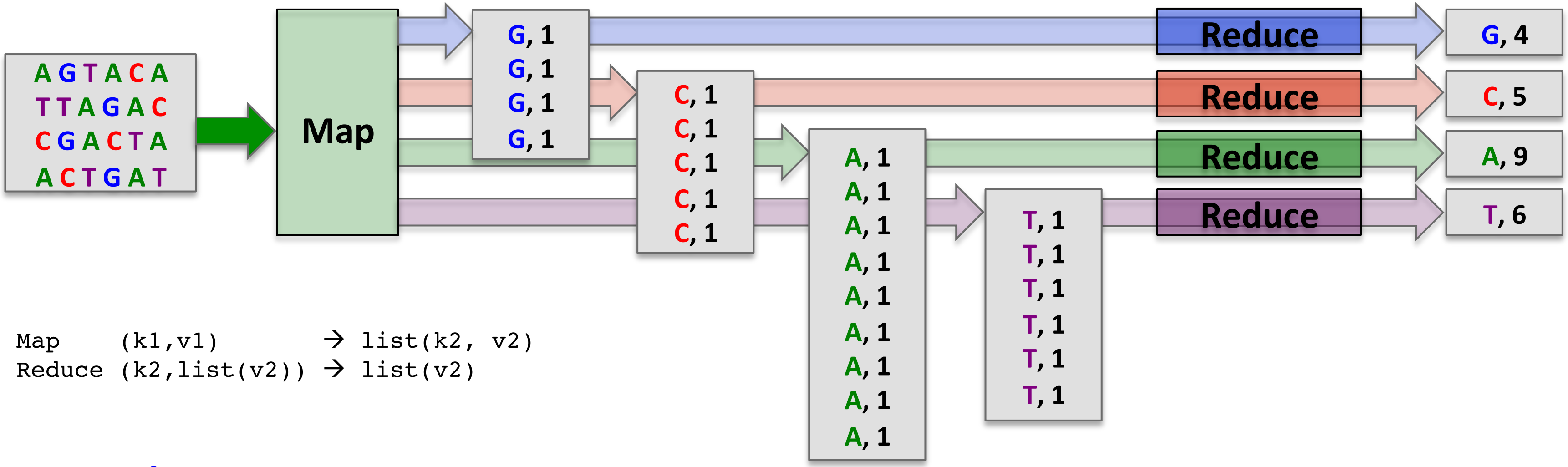
- The computation takes a set of input key/value pairs, and produces a set of output key/value pairs



- The user of the MapReduce library expresses the computation as two functions: **Map** and **Reduce**
  - **Map function takes an input pair and produces a set of intermediate key/value pairs.** MapReduce library groups together all intermediate values associated with the same intermediate key `I` and passes them to the Reduce function
  - **Reduce function accepts intermediate key `I` and a set of values for that key. It merges together these values to form a possibly smaller set of values.** Typically just zero or one output value is produced per Reduce invocation



Programming model



```
Map    (k1,v1)         → list(k2, v2)
Reduce (k2,list(v2)) → list(v2)
```

Examples:

- Distributed Grep, Count of URL access frequency, Reverse Web-Link graph, Term-Vector per Host, Inverted index, Distributed sort, …

Agenda

- Introduction
- Programming model
- **Implementation**
- Refinements
- Performance
- Experience
- Conclusions

Implementation

Large clusters of COTS x86 servers connected together with switched Ethernet

- Dual-processor x86 running Linux, 2-4 GB RAM
- FastEthernet NIC or GigabitEthernet NIC
- Hundreds or thousands of machines (machines failure are common)
- Storage = inexpensive IDE disks (DAS) and a distributed file system (GFS = Google File System)

- Users submit jobs to a scheduling system: each job consist of a set of tasks, and is mapped by the scheduler to a set of available machines within the cluster

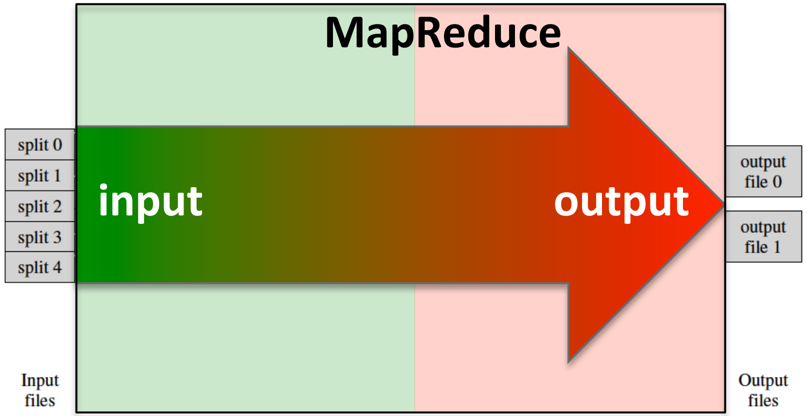
Implementation

Execution overview *(from the User point of view)*

- **Map** invocations are distributed across multiple machines by partitioning the input data into a set of M splits
  - Number of splits **M is automatically calculated based on the input data size**
- **Reduce** invocations are distributed by partitioning the intermediate key space into **R** pieces using a partition function: ***hash(key) mod R***
  - Number of partitions **R and the partitioning function are specified by the user**

Let's run MapReduce with **R = 2** on those input files with my C++ program that contain my custom **Map** and **Reduce** functions :

```
public: virtual void Map(const MapInput &input) {…}
public: virtual void Reduce(ReduceInput* input) {…}
```



MAP

- input files splits in 5 pieces: **M = 5**

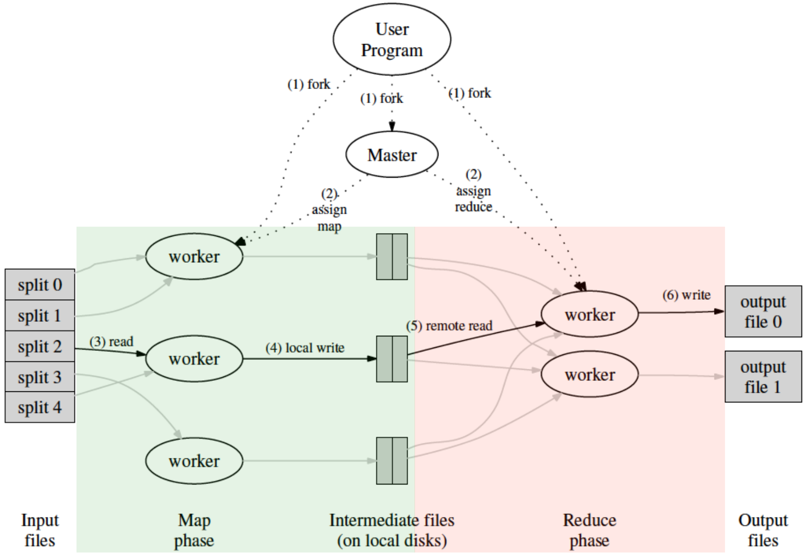
REDUCE

- 2 partitions: **R = 2**

Implementation

Execution overview *(from the **Cluster** point of view)*

- **Map** invocations are distributed across multiple machines by partitioning the input data into a set of M splits
  - Number of splits **M is automatically calculated based on the total input data size**
- **Reduce** invocations are distributed by partitioning the intermediate key space into **R** pieces using a partition function: ***hash(key) mod R***
  - Number of partitions **R and the partitioning function are specified by the user**



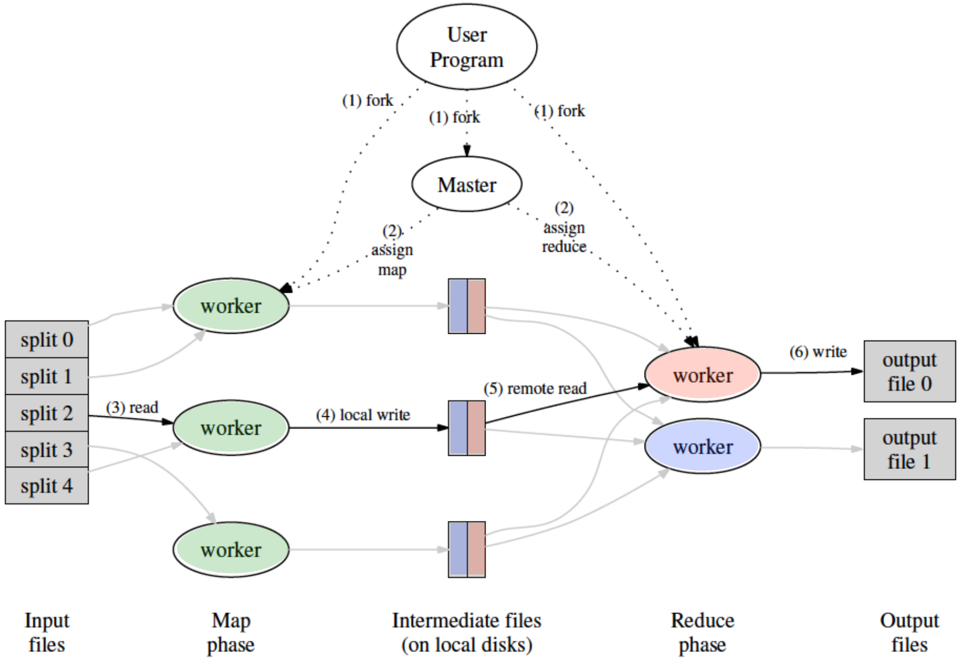
MAP
- input files splits in 5 pieces: **M = 5**
- 5 Map tasks with 3 workers

REDUCE
- 2 partitions: **R = 2**
- 2 Reduce tasks with 2 workers

Implementation

Execution overview

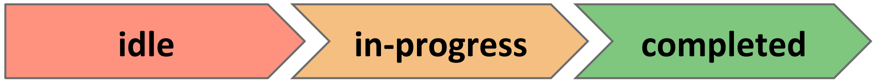
1) First the MapReduce library in the “user program” splits the input files into M pieces (16 MB to 64 MB per piece, controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2) One of the copies of the program is special: the Master. The rest are workers that are assigned work by the master. There are M Map tasks and R Reduce tasks to assign: the Master pick idle workers and assigns each one a task.
3) A worker who is assigned a Map task reads the content of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory
4) Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these local file are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5) Master notified Reduce tasks about the intermediate files locations. Reduce worker read and sort all intermediate data so that all occurrences of the same key are grouped together.
6) Reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user’s Reduce function. The output of the Reduce function is appended to a final output file for this Reduce partition.
7) When all Map tasks and Reduce tasks have been completed, the Master wakes up the “User Program”



Implementation

Master Data Structures *(for each job and each associated Map and Reduce sub-tasks)*

- **Task state**



- **Worker machine identity affected to of each tasks**

Task	**Worker** (x86 Linux server)
Map task for split0	worker000042.dcmoon.internal.google.com
Map task for split1	worker000051.dcmoon.internal.google.com
Map task for split2	Worker000033.dcmoon.internal.google.com
Map task for split3	worker001664.dcmoon.internal.google.com
Map task for split4	worker001789.dcmoon.internal.google.com
Reduce task for partition0	worker001515.dcmoon.internal.google.com
Reduce task for partition1	worker001806.dcmoon.internal.google.com

- **Locations and sizes of R intermediate file regions produced by each Map task**
  - Map tasks update regularly the Master concerning intermediate file size locations and sizes
  - Master then pushed incrementally the information to workers that have in-progress Reduce tasks

Implementation

Fault Tolerance

- MapReduce library tolerate machine failures (= worker failure) gracefully

- **Worker failure**
  - Master “pings” every worker periodically: if no response from worker in a certain amount of time, then Master marks worker as failed
  - Any Map task or Reduce task **in-progress** on a failed worker is reset to **idle** and become eligible for re-scheduling
  - Similarly, any **completed** Map tasks that is reset back to their initial **idle** state become eligible for scheduling on other workers
  - When a Map task is re-scheduling after a failed worker, all workers executing associated Reduce tasks are notified of the re-execution.

Implementation

Fault Tolerance

- **Master failure**
  - Master write periodic checkpoints of Master data structures
  - If Master dies, a new copy can be started from the last check pointed state
  - Google current implementation (2004) aborts the MapReduce computation if the Master fails … ☹
  - Clients can check for this condition and retry the MapReduce operation if they desire …

Implementation

Fault Tolerance

- **Semantics in the presence of failures**
  - If Map and Reduce functions are deterministic functions then one input give always the same output for a task: as a consequence **distributed MapReduce implementation always produce the same output, even with failed-rescheduled tasks**
  - **MapReduce rely on atomic commits of Map and Reduce task outputs** to achieve this property:
    - **Each in-progress task writes its output to private temporary files**. A Reduce task produces one such file, and a Map task produces R such files (one per Reduce tasks).
    - When a Map task completed, worker sends a message to the Master and includes the names of the R temporary files
    - When a Reduce task completes, worker **atomically renames** its temporary output file to the final output file

See backup tasks later… ;-)

  - *If the Master receives a completion message for an already completed Map task, it ignores the message*
    - ***The first of duplicate Map tasks that ends gets the win***
  - *MapReduce rely on the atomic renames operation of the GFS to guarantee that a final Reduce output file contains just the data produced by only ONE execution of the task*
    - ***The first of duplicate Reduce tasks that ends gets the win***

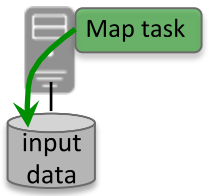
Implementation

Locality

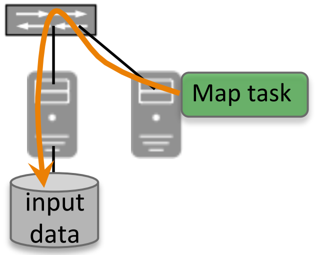
- Network bandwidth is a relatively scarce resource

- MapReduce conserve network bandwidth:
  - Master try to choose a Map worker that is also the GFS chunkserver responsible of the chunk replica of the input data (then Map task runs on the same chunkserver that stores input data)
  - Or Master attempts to schedule a Map task near a replica (worker and GFS chunkserver on the same switch)
  - Map worker doesn't use GFS to store intermediate file, but store them on local disks

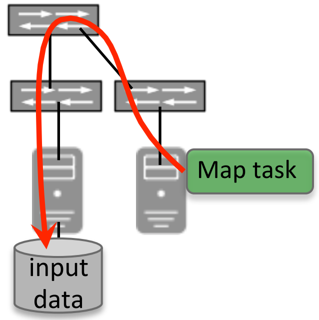
First choice (no network impact)



Second choice (same switch)



Worst case (different switches)



Implementation

Task Granularity

- Ideally M and R should be much larger than the number of worker machines
  - Improves dynamic load-balancing
  - Speeds up recovery when a worker fails

- Practical bounds
  - Master must make O(M + R) scheduling decisions
  - Master must keep O(M x R) state in memory

- In practice (2004)
  - User choose M so that each individual task is roughly 16 MB to 64 MB of input data
  - User make R a small multiple of the number of worker machines
  - Typical MapReduce computations with M = 200 000 and R = 5 000 using 2 000 worker machines

Implementation

Backup Tasks

- **Problem**: a straggler is a machine that takes an unusually long time to complete one of the last few Map or Reduce tasks in the computation: bad disk, competition for CPU, memory, local disk, …

- **Solution**: When a MapReduce operation is close to completion, the Master schedules backup executions of the remaining ***in-progress*** tasks *(the task is marked as completed whenever either the primary or the backup execution completes)*

- Mechanism tuned to increase computational resources used by the MapReduce operation by no more than a few 1% of backup tasks increase over the total of M + R tasks
  - *Example: Backup tasks threshold for M = 200 000 and R = 5 000 is (200 000 + 5 000) / 100 = 2 050*

- **This significantly reduces the time to complete large MapReduce operation ! ☺**

- *See performance benchmark later …* **;-)**

Agenda

- Introduction
- Programming model
- Implementation
- **Refinements**
- Performance
- Experience
- Conclusions

Refinements

Partition function

- Users specify the number of reduce tasks/output files they desire (= R)
- Data gets partitioned across these tasks using a partition function on the intermediate key
  - Default partitioning function is **hash(key) mod R** that gives fairly well-balanced partitions

- User can provide custom partitioning function for special cases
  - Example: Output keys are URLs and you want all entries for a single host to end up in the same output file

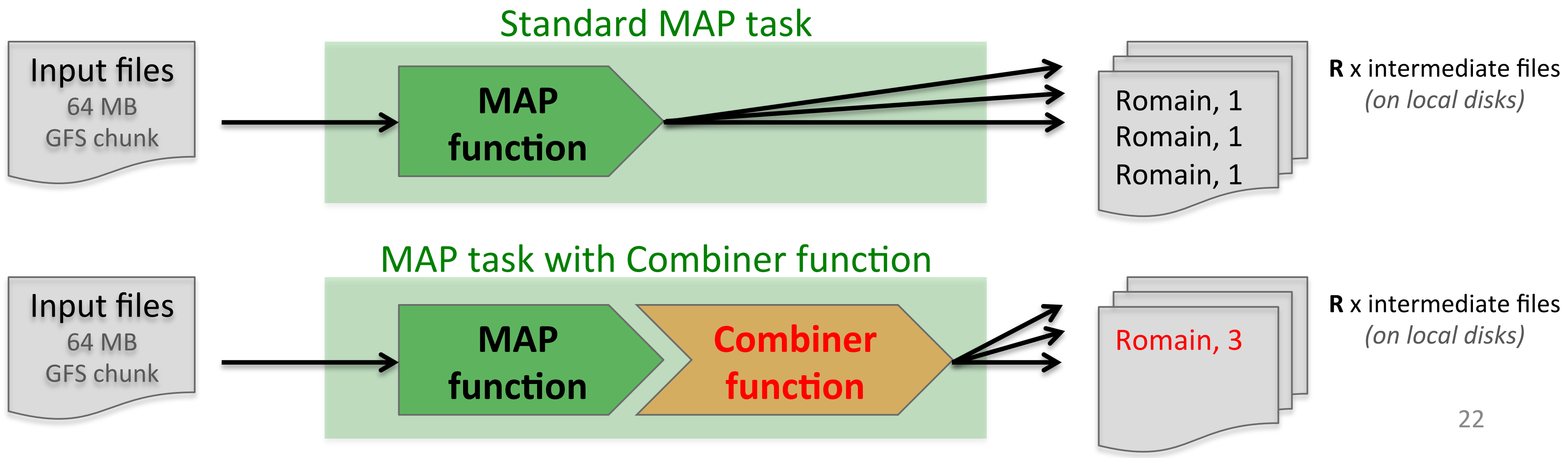
Ordering guarantees

- Within a partition the intermediate key/value pair are processed in increasing key order
  - Make it easy to to generate a sorted output file per partition
  - Efficient random access lookups by key on output file

Refinements

Combiner function

- User is allow to specify an optional combiner function that does partial merging of this data before it is sent over the network
- Combiner function is executed on each Map task (typically the same code is used to implement both the combiner and the reduce function)



Refinements

Input and Output types

- MapReduce library provides support for reading input data in several formats
  - "text" mode input type treats each line as a key/value pair (offset in the file/contents of the line)
  - Sequence of key/value pairs sorted by key
  - …
- A reader does not necessarily need to provide data read from a file
  - From a database (BigTable, Megastore ?, Dremel ?, Spanner ?, …)
  - From data structured mapped in memory,
  - …
- MapReduce also support a set of output types for producing data in different formats
- User can add support for new input or output types

Refinements

Skipping Bad records

- **Problem**: bugs in user code could cause the Map or Reduce functions to crash deterministically on certain records : such bugs prevent a MapReduce operation from completing …

- **Solution**: Sometimes it is acceptable to ignore few records (statistical analysis on large data set) There is an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records on order to make forward progress
  - When master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

Refinements

Local execution

- **Problem**: debugging MapReduce problems in a distributed system can be tricky …
- **Solution**: an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on a local machine
  - To help facilitate debugging, profiling, and small-scale testing (GDB, …)
  - Computation can be limited to a particular map tasks

Refinements

Status information

- Master runs an internal HTTP server and exports a set of status pages for human consumption
  - Progress of the computation
  - How many tasks completed/in progress
  - Bytes of input, bytes of intermediate data, bytes of output
  - Processing rates
  - …
- Pages also contain link to the standard error and standard output files generated for each task
- User use this data to predict how long the computation will take, and whether or not more resources should be added to the computation
- Top level status page
  - Which workers have failed
  - Which Map and Reduce tasks were processing when they failed

Refinements

Counters

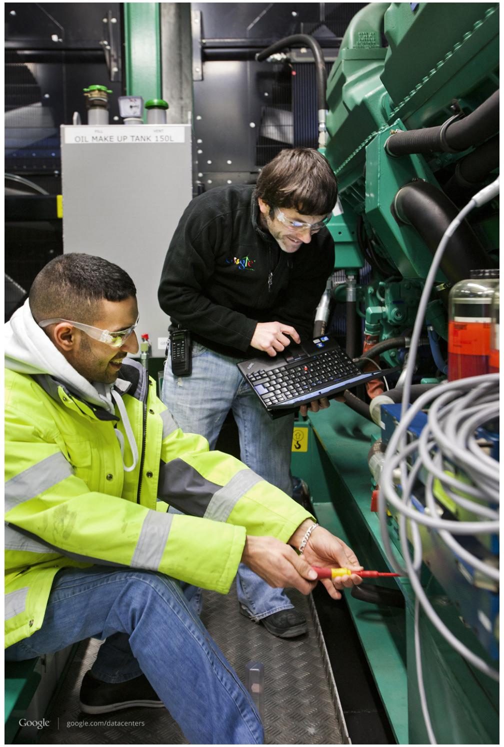
- MapReduce library provides a counter facility to count occurences of various events
  - User code may want to count total numbers of words processed or other customs counters
- User code creates a custom named counter object and then increments the counter appropriately in the Map and/or Reduce function
- Counter values from individual worker machines are periodically propagated to the master (piggybacked on the Heartbeat response). Master aggregates the counter from successful Map/ Reduce tasks
- MapReduce default counter values
  - Number of input key/value pairs processed
  - Number of output key/value pairs processed
  - …

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

Agenda

- Introduction
- Programming model
- Implementation
- Refinements
- **Performance**
- Experience
- Conclusions



Performance

Cluster configuration

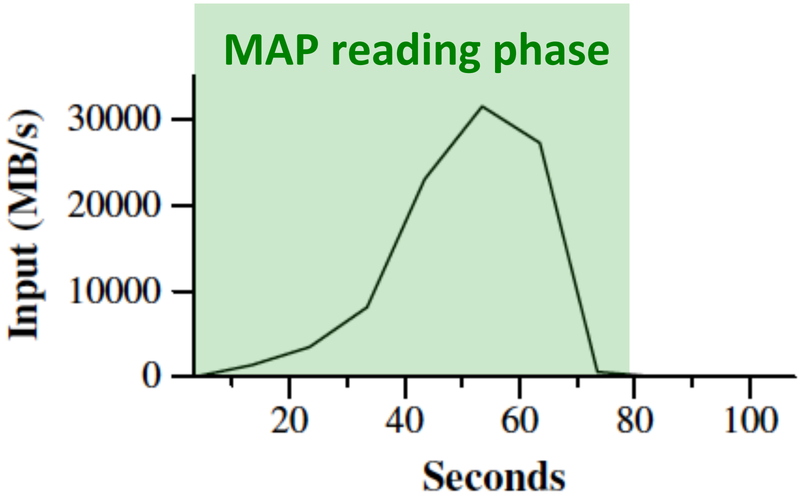
- **1 800 machines**
  - 2 x 2GHz Intel Xeon processors with Hyper-Threading enabled, 4 GB RAM, 2 x 160 GB IDE disks, Gigabit NIC

- **Terra-Search**
  - Searching a rare three-character pattern through 10^10 100-byte records
  - Input split into 64MB (M = 15 000 and R = 1)
  - Pattern occurs in 92 337 records
- **Terra-Sort**
  - Sorts 10^10 100-byte records (= 1 terabyte of data)
  - M = 15.000 and R = 4.000

Performance (Terra-Search)

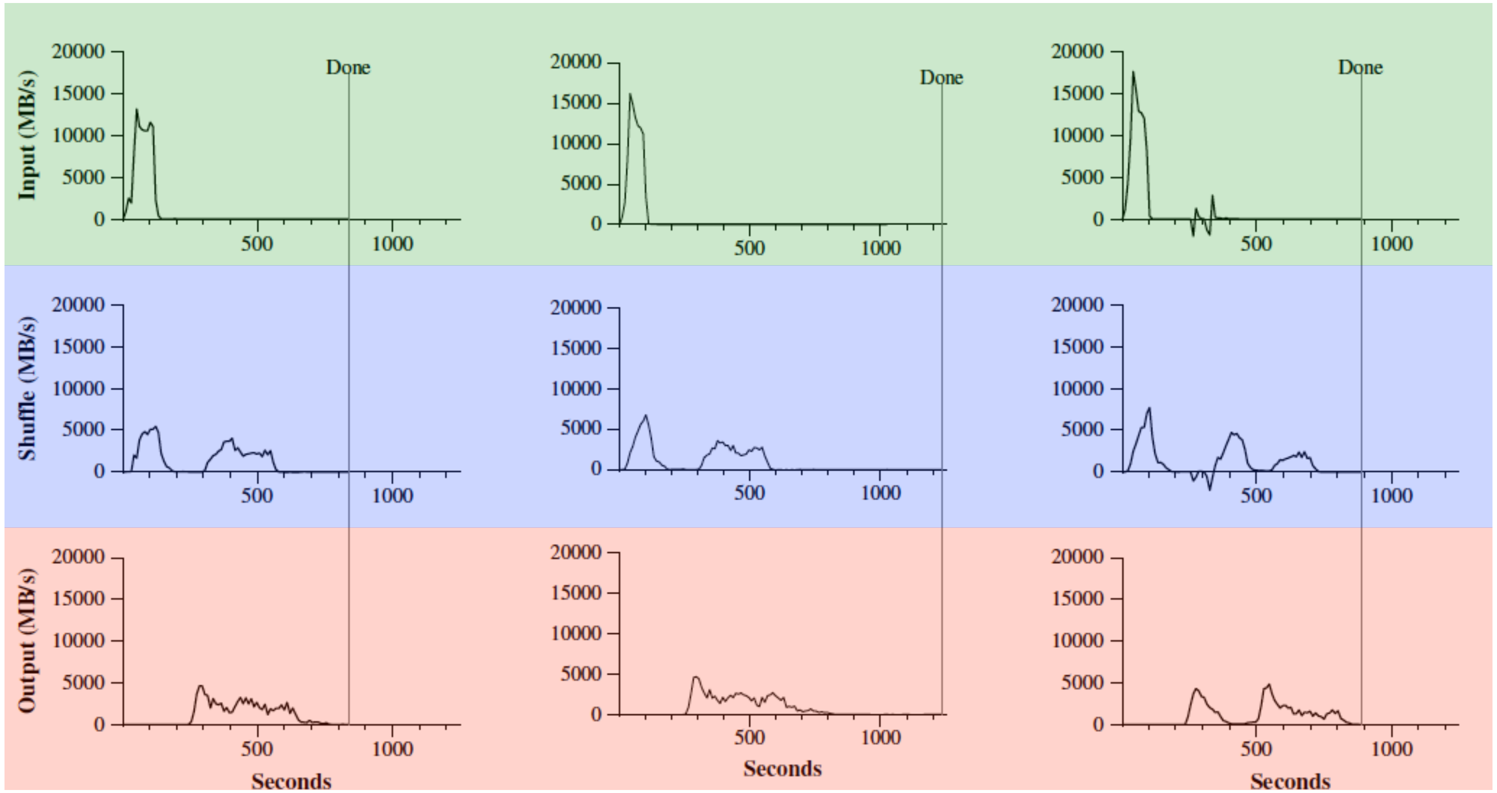
Y-axis show the rate at wich input data is scanned

- Rate peaks at over **30 GB/s whith 1 764 workers** assigned
- Entire computation = **150 seconds** including 60 seconds of startup overhead:
  - Worker program propagation over network
  - GFS delay to open 1.000 GFS files



Data transfer rate over time

Performance (Terra-Sort)



Normal execution **No backup tasks** **200 tasks killed**

Performance (Terra-Sort)

Normal execution = 891 seconds

No Backup tasks = 1 283 seconds

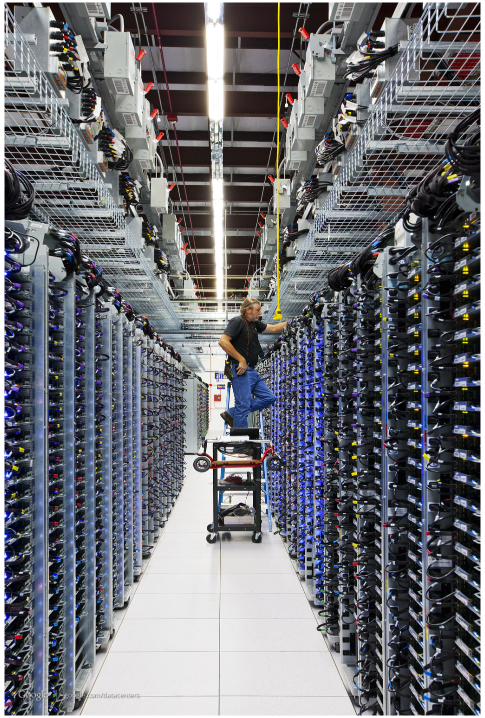
- After 960 seconds, all except 5 of reduce tasks are completed
- Slow machines (= stragglers) don't finish until 300 seconds later
- An increase of 44% in elapsed time

200 tasks killed = 933 seconds

- 200 tasks killed out of 1746 worker processes (underlying scheduler immediately restarted new worker processes on these machines)
- An increase of 5% over the normal execution time

Agenda

- Introduction
- Programming model
- Implementation
- Refinements
- Performance
- **Experience**
- Conclusions

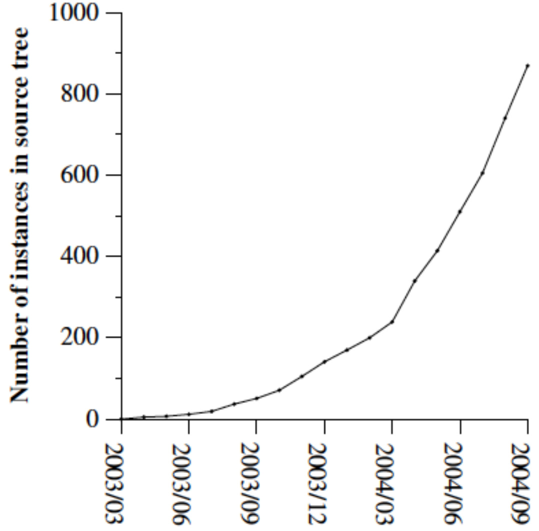


Experience (Feb 2003 – Aug 2004)

Usage

- Large scale machine learning problems
- Clustering for Google News and Froogle products
- Extraction of data to produce reports of popular queries
- Extraction of properties of web pages
- Large-scale graph computations

MapReduce instances over time



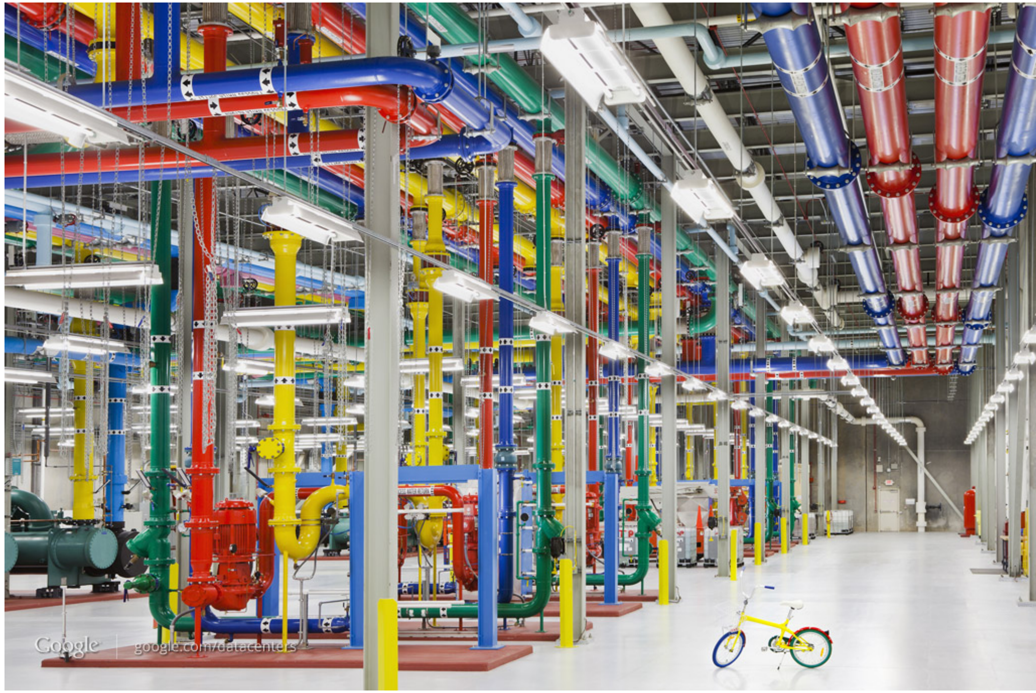
MapReduce jobs run in august 2004

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB

Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Agenda

- Introduction
- Programming model
- Implementation
- Refinements
- Performance
- Experience
- **Conclusions**



Conclusions (2004)

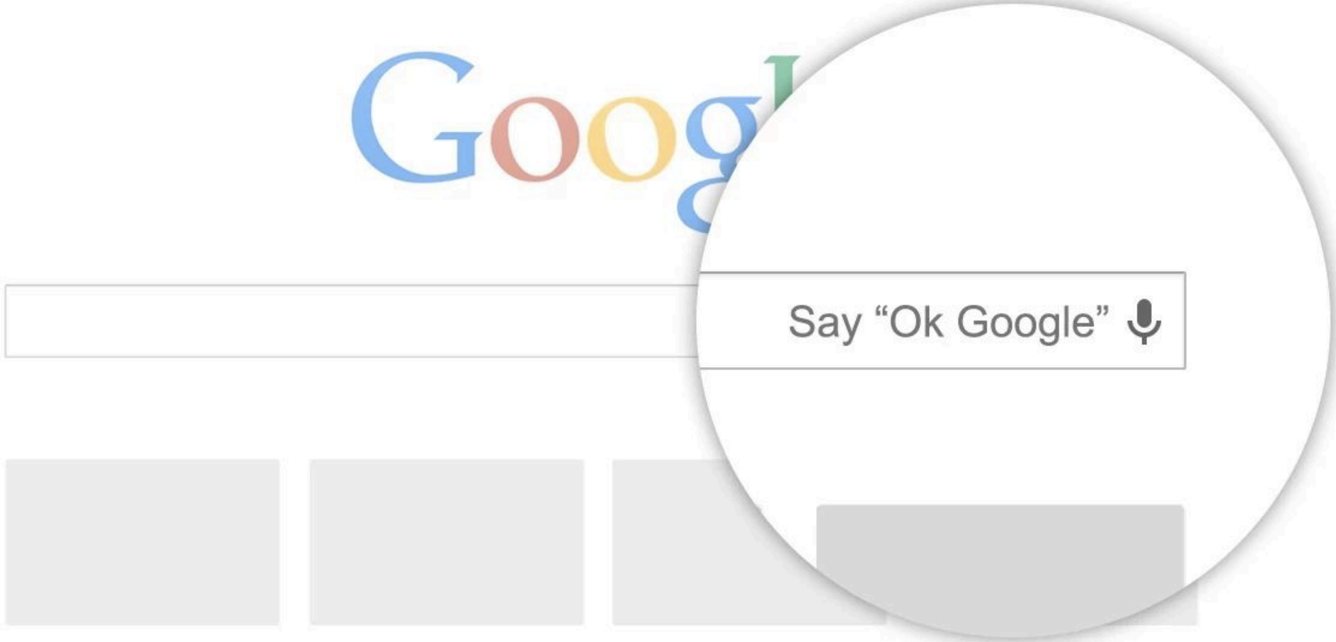
Success

- MapReduce model is easy to use without experience with distributed systems
- Can address large variety of problem (web search, sorting, data mining, machine learning, …)
- Scale to large clusters (thousands of machine)

Things learned

- Restricting programming model make it easy to parallelize and to be fault-tolerant
- **Network bandwidth is a scarce resource** (optimizations to reduce amount of data sent across the network)
- Redundant execution to reduce impact of slow machines AND to handle failures and data loss

Any questions



Romain Jacotin

romain.jacotin@orange.fr