Introduction to Map-Reduce Computing

Dawid Weiss

Institute of Computing Science
Poznań University of Technology

2008

SORT 1 TB OF DATA

SORT 1 TB OF DATA

- how?

SORT 1 TB OF DATA

- how?
- how long?

SORT 1 TB OF DATA

- how?
- how long?
- why me?

Estimate:

- read 100MB/s, write 100MB/s
- no disk seeks, instant sort

Estimate:

- read 100MB/s, write 100MB/s
- no disk seeks, instant sort
- 341 minutes → 5.6 hours

The terabyte benchmark winner:

Estimate:

- read 100MB/s, write 100MB/s
- no disk seeks, instant sort
- 341 minutes → 5.6 hours

The terabyte benchmark winner:

- 209 seconds (3.48 minutes)

Estimate:

- read 100MB/s, write 100MB/s
- no disk seeks, instant sort
- 341 minutes → 5.6 hours

The terabyte benchmark winner:

- 209 seconds (3.48 minutes)
- 910 nodes x (4 dual-core processors, 4 disks, 8 GB memory)

Speeding-up computations

Moore's Law aside, we have had:

- Vector-data processing
  special compilers, vector data structures
- Transputers
  OCCAML, message passing, etc.
- Multiple-processor machines
  threading, software or hardware mediated synchronization
- Massively parallel computing
  multi-node computing, network issues, failure recovery
- GPU computing
  local, hardware-based acceleration of specific operations

The overhead of custom parallelization

Parallelization is difficult:

- Job distribution, balancing and scheduling.
- Failure detection and recovery.
- Multi-threading and synchronization issues.
- Job progress, status tracking, recovery.

The overhead of custom parallelization

Parallelization is difficult:

- Job distribution, balancing and scheduling.
- Failure detection and recovery.
- Multi-threading and synchronization issues.
- Job progress, status tracking, recovery.

Simplicity of the original computation is usually lost.

Different types of scaling problems

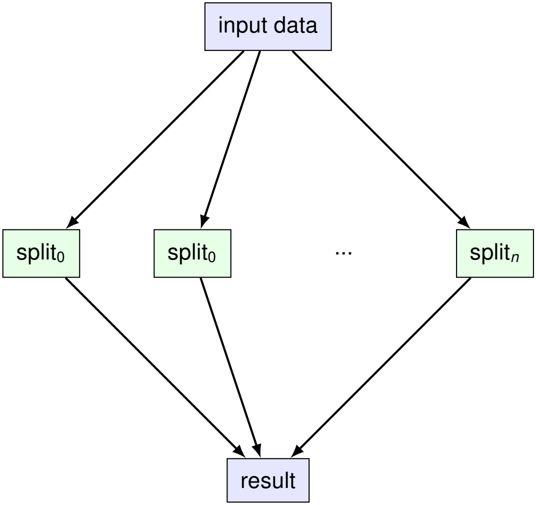
Algorithm complexity:

- adding new CPUs $\rightarrow$ at most linear speedup,
- growing communication/ synchronization overhead (and complexity).

Input/ output data size:

- large instances are disk-bound,
- splitting the data and processing in chunks can provide **significant** speedups.

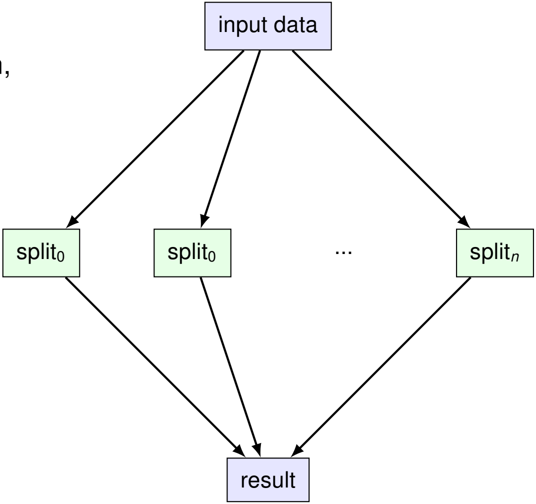
Optimization strategy



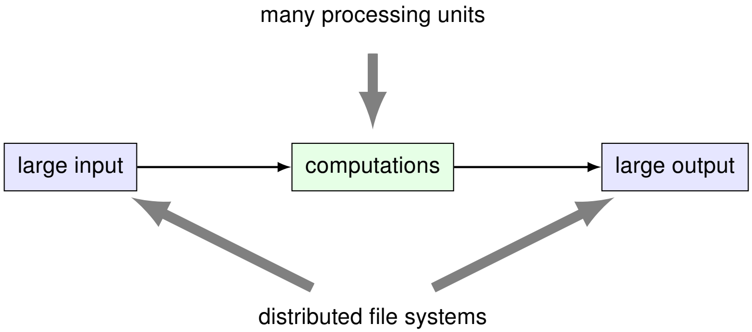
Optimization strategy

Note that:

- chunks must be independent,
- parallelization easy,
- not a universal solution,
- lots of reusable code.



Massive distributed processing with data splits



Assumptions:

- Computations are very simple and chunk-independent.
- Data instances huge.
- Input can be fragmented into “splits”.
- Focus on **computations**, forget the rest.

Quiz

Why is this computation model very good?

Quiz

Why is this computation model very good?

It worked for Google. It must be good.

Examples of MDP problems

- Searching or scanning (grep).
- Counting (URL accesses, patterns).
- Indexing problems (reverse link, inverted indices).
- Sorting problems (merge sort).
- Data clustering problems.

MAP-REDUCE: the power of distributed computing for everyone.

Map Reduce

Map Reduce (Jeffrey Dean, Sanjay Ghemawat; Google Inc.)

A technique of automatic parallelization of computations by enforcing a **restricted programming model**, derived from functional languages.

- Inspiration: functional languages (Lisp).
- Data$\rightarrow$data computation flow, no updates.

- Hide the messy details, keep the programmer happy.
- Scalability, robustness and fault-tolerance at the framework level.

SOUNDS TOO BEAUTIFUL TO BE TRUE?

The programming model

```
map(key_in, value_in)
   → [(key_tmp, value_tmp), (key2_tmp, value2_tmp), ...]
```

```
reduce(key, [value₁, value₂, ...])
   → [(key_output, value_output), ...]
```

The programming model

```
map(key_in, value_in)
  → [(key_tmp, value_tmp), (key2_tmp, value2_tmp), ...]
```

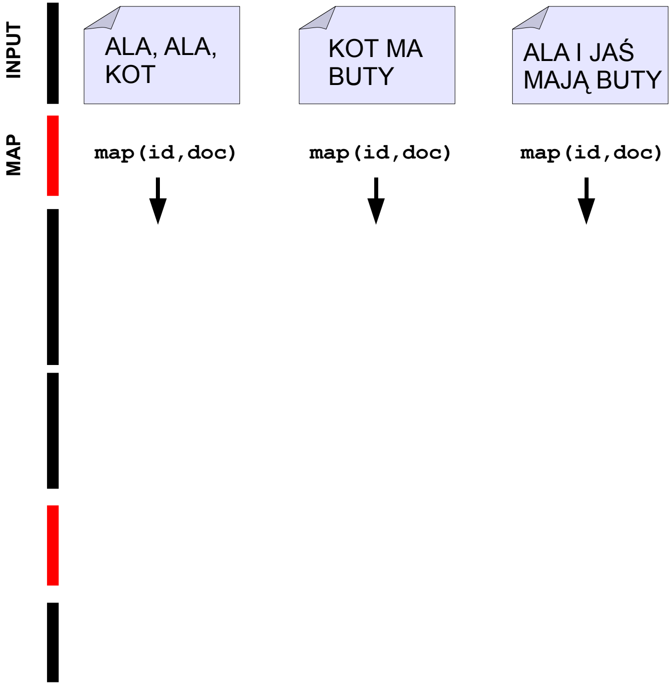
reduce(key, [value_1, value_2, ...])
  → [(key_output, value_output), ...]

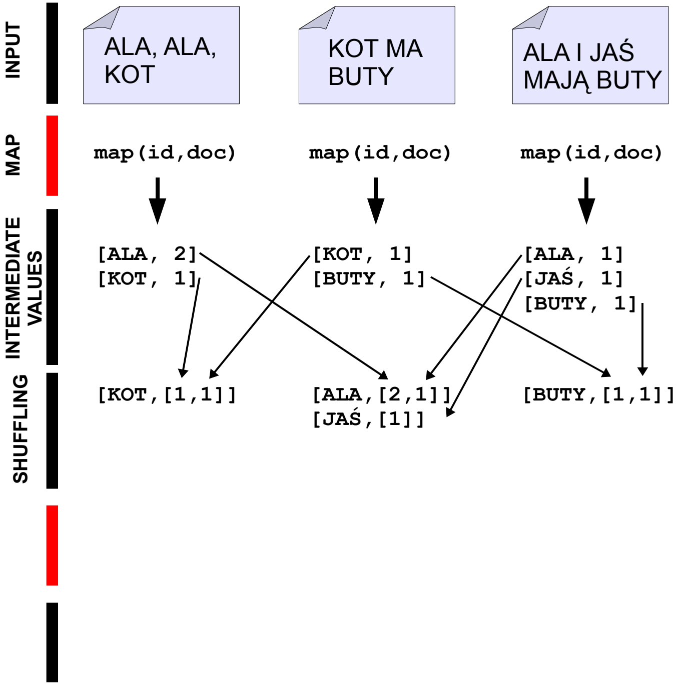
Small print: keys are sorted for the reducer.

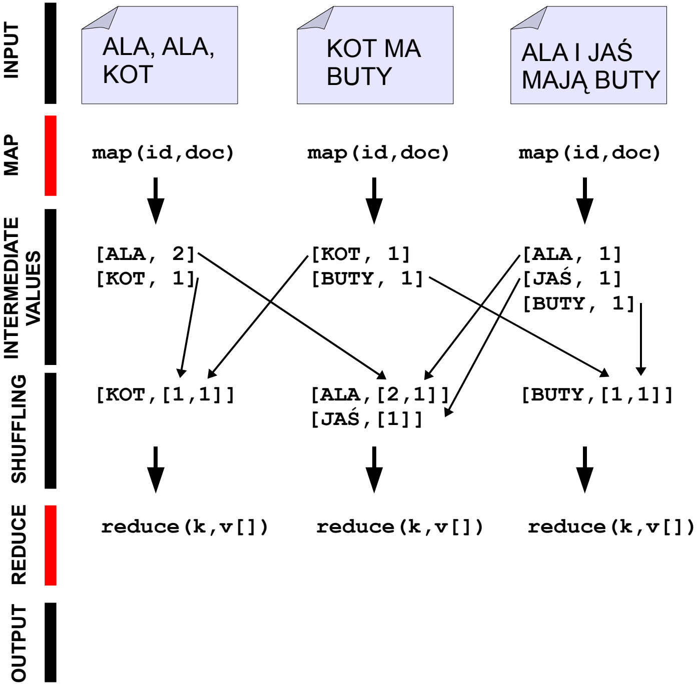
Your assignment:

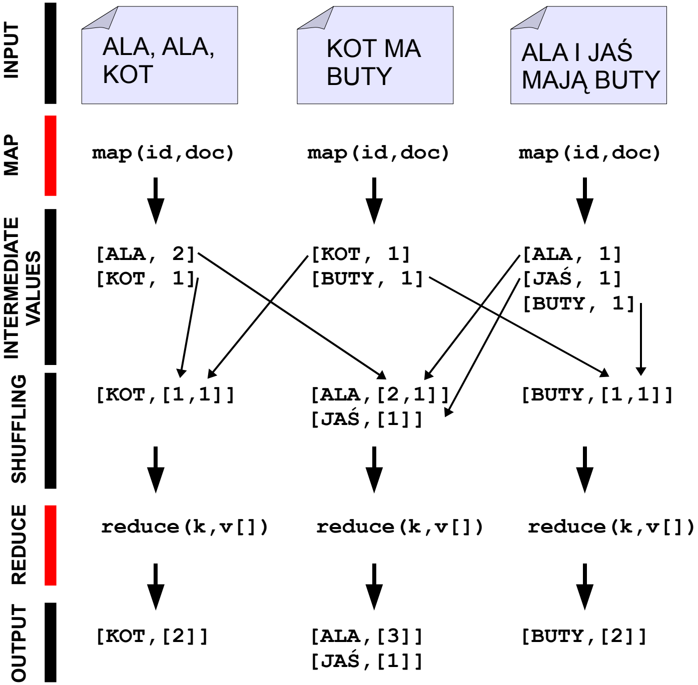
Word frequency in Wikipedia

(approx. 13 GB of text)


INPUT
MAP
ALA, ALA, KOT
KOT MA BUTY
ALA I JAŚ MAJĄ BUTY
map(id,doc)
map(id,doc)
map(id,doc)


INPUT
MAP
INTERMEDIATE VALUES
SHUFFLING
ALA, ALA, KOT
KOT MA BUTY
ALA I JAŚ MAJĄ BUTY
map(id,doc)
map(id,doc)
map(id,doc)
[ALA, 2]
[KOT, 1]
[KOT, 1]
[BUTY, 1]
[ALA, 1]
[JAŚ, 1]
[BUTY, 1]
[KOT,[1,1]]
[ALA,[2,1]]
[JAŚ,[1]]
[BUTY,[1,1]]


INPUT
MAP
INTERMEDIATE VALUES
SHUFFLING
REDUCE
OUTPUT
ALA, ALA, KOT
KOT MA BUTY
ALA I JAŚ MAJĄ BUTY
map(id,doc)
map(id,doc)
map(id,doc)
[ALA, 2]
[KOT, 1]
[KOT, 1]
[BUTY, 1]
[ALA, 1]
[JAŚ, 1]
[BUTY, 1]
[KOT,[1,1]]
[ALA,[2,1]]
[JAŚ,[1]]
[BUTY,[1,1]]
reduce(k,v[])
reduce(k,v[])
reduce(k,v[])


INPUT
MAP
INTERMEDIATE VALUES
SHUFFLING
REDUCE
OUTPUT
ALA, ALA, KOT
KOT MA BUTY
ALA I JAŚ MAJĄ BUTY
map(id,doc)
map(id,doc)
map(id,doc)
[ALA, 2]
[KOT, 1]
[KOT, 1]
[BUTY, 1]
[ALA, 1]
[JAŚ, 1]
[BUTY, 1]
[KOT,[1,1]]
[ALA,[2,1]]
[JAŚ,[1]]
[BUTY,[1,1]]
reduce(k,v[])
reduce(k,v[])
reduce(k,v[])
[KOT,[2]]
[ALA,[3]]
[JAŚ,[1]]
[BUTY,[2]]

Example: word counting [Dean and Ghemawat, 2004]

Map function

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w, "1");
```

Reduce function

```
reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
      result += ParseInt(v);
      Emit(AsString(result));
```

WHERE IS THE FUN PART?

WHERE IS THE FUN PART?

We don’t care about sorting.
We don’t care about shuffling.
We don’t care about communication protocols.
We care about the data and the task.

Some questions

- What should be my key/ value?
  Depends. Examples later.
- How many maps and reduces?
  How much parallelism? Is single result required?
- Can I have multiple maps and a single reduce?
  Yes and no. Decompose. Keep it simple.

Even more questions

Can I disable intermediate sorting? Can I change input/ output data format? Can I. . .

Everything can be tweaked. Don't bother unless really necessary.

Examples of applying map-reduce

- Distributed `grep`.

```
map:    (--, line) -> (line)
reduce: identity
```

- Reverse Web link graph.

```
map:    (source-url, html-content) ->
               (target-url, source-url)
reduce: (target-url, [source-urls]) ->
               (target-url, concat(source-urls))
```

- Inverted index of documents.

```
map:    (doc-id, content) -> (word, doc-id)
reduce: (word, [doc-ids]) -> (word, concat(doc-ids))
```

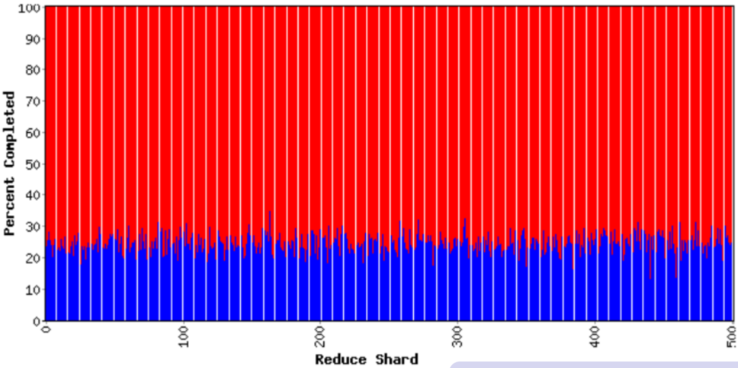
- More complex tasks achieved by combining Map-Reduce jobs (the indexing process at Google – more than 20 MR tasks).

MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 31 min 34 sec

1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	133837.8	136929.6



Counters

Variable	Minute
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.1
Output (MB/s)	1238.8
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	51738599
mr-merge-	51738599

Example reduce phase (indexing at Google).

More interesting examples

the ever-interesting context of

BURAK

Keys, values, map/ reduce function?

- Input keys: plain text documents
- Map function:

```
map(doc, --):
    foreach (String [] words : to_stems(sentence_split(doc))) {
        for (i = 0; i < words.length; i++) {
            if (words[i] == "burak") {
                for (j = max(0, i - w); j <= min(words.length - 1, i + w); j++) {
                    if (j != i && not_stopword(words[j]))
                        emit(words[j], 1)
                }
            }
        }
    }
```

- Reduce function: simple count
- Subsequent job: re-sort by count.

- Rzeczpospolita corpus (190379 articles, approx. 300 MB)
- Buzz corpus (1201458 RSS posts, approx. 180 MB)

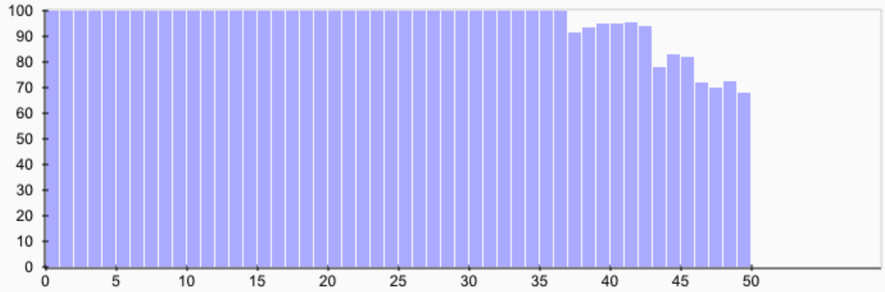
Processing times: approx. 3 minutes.

User: dweiss
Job Name: context-1
Job File: hdfs://hpc1:40010/tmp/hadoop-dweiss/mapred/system/job_200811051400_0001/job.xml
Status: Running
Started at: Wed Nov 05 14:03:46 CET 2008
Running for: 1mins, 15sec

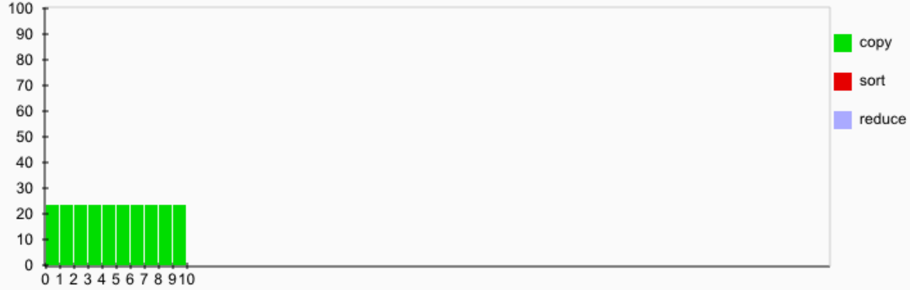
Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	70.68%	50	4	17	29	0	0 / 0
reduce	12.00%	10	0	10	0	0	0 / 0

	Counter	Map	Reduce	Total
File Systems	HDFS bytes read	182,853,817	0	182,853,817
	Local bytes written	8,684	0	8,684
Job Counters	Launched reduce tasks	0	0	10
	Rack-local map tasks	0	0	22
	Launched map tasks	0	0	46
	Data-local map tasks	0	0	24
Map-Reduce Framework	Combine output records	28	0	28
	Map input records	111,735	0	111,735
	Map output bytes	19,264	0	19,264
	Map input bytes	167,331,567	0	167,331,567
	Map output records	1,376	0	1,376
	Combine input records	1,376	0	1,376

Map Completion Graph - close



Reduce Completion Graph - close



Rzeczpospolita (c-1)		Rzeczpospolita (c-2)		Buzz (c-2)	
498	cukrowych	425	cukrowych	28	cukrowych
132	kg	235	zł	18	plantatorzy
93	ton	159	kg	15	cukrowego
84	zł	91	ton	6	plantatorami
67	skupu	62	skupu	6	zekai
64	uprawy	58	plantatorom	6	tahir
58	plantatorom	56	uprawy	6	ankarze
56	cukrowego	53	preferencyjnych	6	produkcji
53	zbiory	53	zbiory	6	ziemniaków
51	plantatorów	47	plantatorów	5	upraw
46	cukrowe	42	mln	4	skutki
31	plony	41	cukrowe	4	plantatorów
27	tys	37	cukru	4	limitu
26	cebula	37	cukrowego	4	podlascy
24	czerwone	32	ziemniaków	4	sfermentowanymi
19	ziemniaków	30	cebula	4	ciężkie
18	artur	30	plony	4	raka
17	plantatorzy	30	ceny	3	artur
16	ćwikłowych	28	marchew	3	plantacje
16	zbioru	26	cena	3	lotniczych

Context of “wojna” (selected)

Rzeczpospolita (c-2)		Buzz (c-2)	
2310	światowej	1972	światowej
576	zimnej	330	drugiej
550	czeczenii	180	zimnej
536	domowej	150	gwiezdnych
385	zatoce	41	futbolowej
308	światową	39	futbolowa
158	bośni	33	gruzji
146	jugosławii	30	cenowej
132	afganistanie	25	polsko-polska
125	wietnamie	25	prezydent
125	terroryzmem	19	rosyjsko-gruzińskiej

Other analytical tasks

Other analytical tasks

- HTTP log analysis

Other analytical tasks

- HTTP log analysis
- Shallow linguistic patterns

Other analytical tasks

- HTTP log analysis
- Shallow linguistic patterns
- Lech Poznań vs. Legia Warszawa, annual mentions in RzP

Other analytical tasks

- HTTP log analysis
- Shallow linguistic patterns
- Lech Poznań vs. Legia Warszawa, annual mentions in RzP
- Doda vs. Jola Rutowicz?

Other analytical tasks

- HTTP log analysis
- Shallow linguistic patterns
- Lech Poznań vs. Legia Warszawa, annual mentions in RzP
- Doda vs. Jola Rutowicz?
- ...

M-R bottlenecks

Parallelism and speed up due to:

- All intermediate values are **independent**. Adding CPUs $\rightarrow$ nearly linear speedup.
- Distributed input $\rightarrow$ fewer I/O bottlenecks.
- No limit on data size (list iterators).

Bottlenecks:

- Reduce phase cannot start until map is finished.
- Computation not over until the last reducer is finished.
- A lot of data shuffling.

FAULT TOLERANCE AND OPTIMIZATIONS

Fault tolerance and optimizations

Facts:

- Some nodes **will fail**.

Fault tolerance and optimizations

Facts:

- Some nodes **will fail**.

Fault tolerance and optimizations

Facts:

- Some nodes **will fail**.
- Some nodes are faster than others.
- Some failures ar silent killers
  (memory corruption, bad blocks).

Fault tolerance and optimizations

Facts:

- Some nodes **will fail**.
- Some nodes are faster than others.
- Some failures ar silent killers
  (memory corruption, bad blocks).

Map-reduce controller node can:

- distribute multiple copies of each map/reduce job (corrupted nodes problem),

Fault tolerance and optimizations

Facts:

- Some nodes **will fail**.
- Some nodes are faster than others.
- Some failures ar silent killers
  (memory corruption, bad blocks).

Map-reduce controller node can:

- distribute multiple copies of each map/reduce job (corrupted nodes problem),
- isolate and exclude key/value that causes repeatable errors (corrupted computation problem),

Fault tolerance and optimizations

Facts:

- Some nodes **will fail**.
- Some nodes are faster than others.
- Some failures ar silent killers (memory corruption, bad blocks).

Map-reduce controller node can:

- distribute multiple copies of each map/reduce job (corrupted nodes problem),
- isolate and exclude key/value that causes repeatable errors (corrupted computation problem),
- speculatively execute multiple copies of each map/reduce job (slow tail node problem).

Data locality and bandwidth optimizations

Map-reduce controller node should:

- cooperate with DFS and assign map/ reduce jobs to where the data already is (node, rack, data center),

Data locality and bandwidth optimizations

Map-reduce controller node should:

- cooperate with DFS and assign map/ reduce jobs to where the data already is (node, rack, data center),
- possibly apply an “early-reduce” phase (combiner functions) to limit network traffic.

Data locality and bandwidth optimizations

Map-reduce controller node should:

- cooperate with DFS and assign map/ reduce jobs to where the data already is (node, rack, data center),
- possibly apply an “early-reduce” phase (combiner functions) to limit network traffic.

Data locality and bandwidth optimizations

Map-reduce controller node should:

- cooperate with DFS and assign map/ reduce jobs to where the data already is (node, rack, data center),
- possibly apply an “early-reduce” phase (combiner functions) to limit network traffic.

You, as a programmer, should:

- split your task into chunks of sensible size; more chunks $\longrightarrow$ more load-balancing options,

Data locality and bandwidth optimizations

Map-reduce controller node should:

- cooperate with DFS and assign map/ reduce jobs to where the data already is (node, rack, data center),
- possibly apply an “early-reduce” phase (combiner functions) to limit network traffic.

You, as a programmer, should:

- split your task into chunks of sensible size; more chunks $\longrightarrow$ more load-balancing options,
- make simple map and reduce operations; chain them if necessary.

Applicability of Map-Reduce

- Processing of large, record-based data.

Applicability of Map-Reduce

- Processing of large, record-based data.
- Load-testing framework (?).

Applicability of Map-Reduce

- Processing of large, record-based data.
- Load-testing framework (?).
- (Expensive) heating device.

Conclusions

- An interesting programming paradigm.
- Paralellism, scalability, fault-tolerance.

- Open source implementation: Hadoop.
- “Play big” using services like Amazon’s EC/S3.

References

- Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation, OSDI '2004*, pages 137–150
- lucene (2007). Apache lucene. On-line: `http://lucene.apache.org/`
- nutch (2007). Apache nutch. On-line: `http://lucene.apache.org/nutch/`
- hadoop (2007). Apache hadoop. On-line: `http://lucene.apache.org/hadoop/`