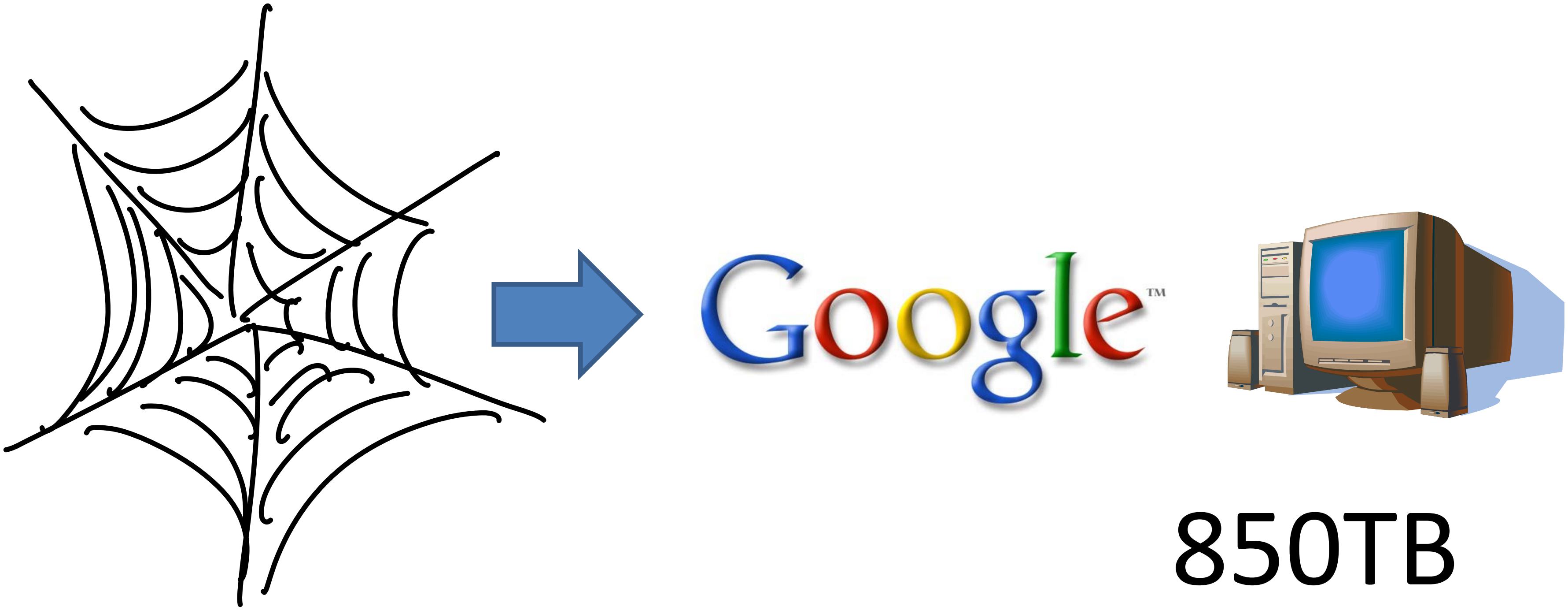
# Map-Reduce

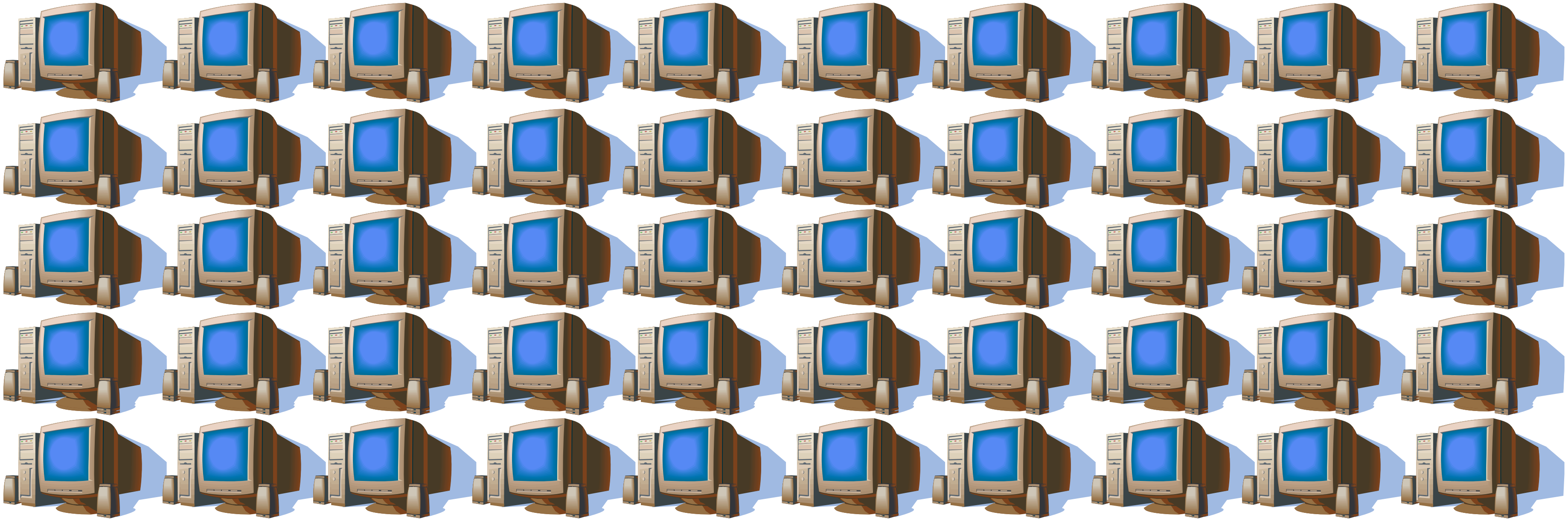
John Hughes

# The Problem




850TB
in 2006

# The Solution?



- Thousands of commodity computers networked together
- 1,000 computers → 850GB each
- How to make them work together?

# Early Days

- Hundreds of ad-hoc distributed algorithms
  - Complicated, hard to write
  - Must cope with fault-tolerance, load distribution, …

**MapReduce: Simplified Data Processing on Large Clusters**
*by Jeffrey Dean and Sanjay Ghemawat*

In Symposium on Operating Systems Design & Implementation (OSDI 2004)

# The Idea

- Many algorithms apply the *same* operation to a lot of data items, then *combine* results

- Cf map :: (a->b) -> [a] -> [b]
- Cf foldr :: (a->b->b) -> b -> [a] -> b
  - Called *reduce* in LISP

- Define a *higher-order function* to take care of distribution; let users just write the functions passed to map and reduce

# Pure functions are great!

- They can be *run anywhere* with the same result—easy to distribute
- They can be *reexecuted* on the same data to recreate results lost by crashes

# ”It’s map and reduce, but not as we know them Captain”

- Google map and reduce work on collections of *key-value pairs*

- map_reduce *mapper reducer* :: [(k,v)] -> [(k2,v2)]
  - *mapper* :: k -> v -> [(k2,v2)]
  - *reducer* :: k2 -> [v2] -> [(k2,v2)]

All the values with the same key are collected

Usually just 0 or 1

# Example: counting words

- Input: (file name, file contents)

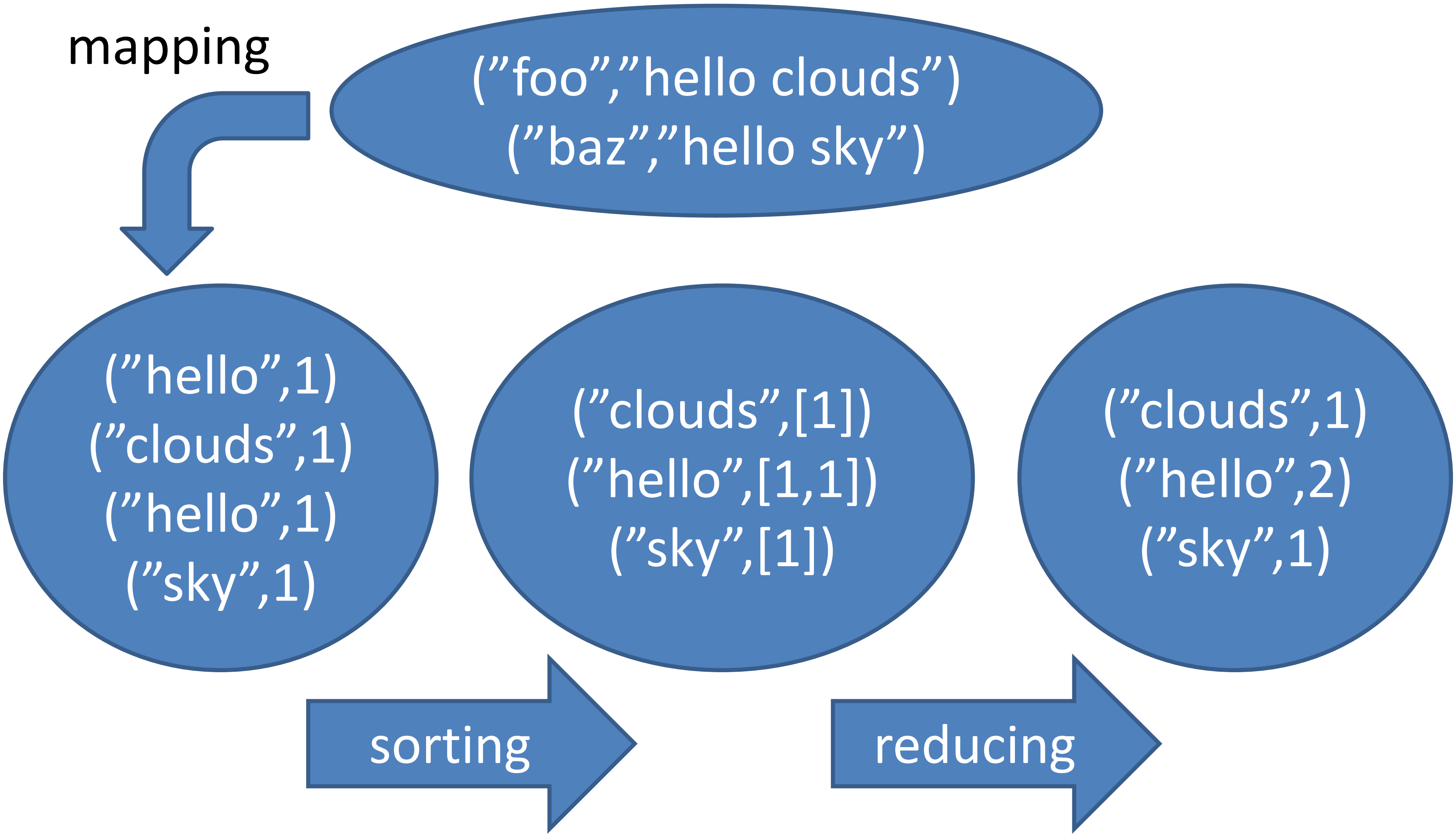
  ↓ mapper

- Intermediate pairs: (word, 1)

  ↓ reducer

- Final pairs: (word, total count)

# Example: counting words



# Map-reduce in Erlang

- A purely *sequential* version

```
map_reduce_seq(Map,Reduce,Input) ->
    Mapped = [{K2,V2}
              || {K,V} <- Input,
                 {K2,V2} <- Map(K,V)],
    reduce_seq(Reduce,Mapped).

reduce_seq(Reduce,KVs) ->
    [KV || {K,Vs} <- group(lists:sort(KVs)),
           KV <- Reduce(K,Vs)].
```

# Map-reduce in Erlang

- A purely *sequential* version

```
> group([{1,a},{1,b},{2,c},{3,d},{3,e}]).
[{1,[a,b]},{2,[c]},{3,[d,e]}]
```

```
m
                  {K2,V2} <-          ,V)],
    reduce_seq(Reduce,Map

reduce_seq(Reduce,KVs) ->
    [KV || {K,Vs} <- group(lists:sort(KVs)),
           KV <- Reduce(K,Vs)].
```

# Counting words

```
mapper(File,Body) ->
    [{string:to_lower(W),1} || W <- words(Body)].

reducer(Word,Occs) ->
    [{Word,lists:sum(Occs)}].

count_words(Files) ->
    map_reduce_seq(fun mapper/2, fun reducer/2,
        [{File,body(File)} || File <- Files].

body(File) ->
    {ok,Bin} = file:read_file(File),
    binary_to_list(Bin).
```

# Page Rank

```
mapper(Url,Html) ->
    Urls = find_urls(Url,Html),
    [{U,1} || U <- Urls].

reducer(Url,Ns) ->
    [{Url,lists:sum(Ns)}].

page_rank(Urls) ->
    map_reduce_seq(fun mapper/2, fun reducer/2,
        [{Url,fetch_url(Url)} || Url <- Urls]).
```

Saves memory in sequential map_reduce

Parallelises fetching in a parallel one

Why not fetch the URLs in the mapper?

# Page Rank

```
mapper(Url,ok) ->
    Html = fetch_url(Url),
    Urls = find_urls(Url,Html),
    [{U,1} || U <- Urls].

reducer(Url,Ns) ->
    [{Url,[lists:sum(Ns)]}].

page_rank(Urls) ->
    map_reduce_seq(fun mapper/2, fun reducer/2,
        [{Url,ok} || Url <- Urls]).
```

# Building an Index

```
mapper(Url,ok) ->
    Html = fetch_url(Url),
    Words = words(Html),
    [{W,Url} || W <- Words].

reducer(Word,Urlss) ->
    [{Word,Urlss}].

build_index(Urls) ->
    map_reduce_seq(fun mapper/2, fun reducer/2,
        [{Url,ok} || Url <- Urls]).
```

# Crawling the web

- Key-value pairs:
  - {Url,Body} if already crawled
  - {Url,undefined} if needs to be crawled

```
mapper(Url,undefined) ->
    Body = fetch_url(Url),
    [{Url,Body}] ++
      [{U,undefined} || U <- find_urls(Url,Body)];
mapper(Url,Body) ->
    [{Url,Body}].
```

# Crawling the web

- Reducer just selects the already-fetched body if there is one

```
reducer(Url,Bodies) ->
    case [B || B <- Bodies, B/=undefined] of
     [] ->
         [{Url,undefined}];
     [Body] ->
         [{Url,Body}]
    end.
```

# Crawling the web

- Crawl up to a fixed *depth* (since we don't have 850TB of RAM)

```
crawl(0,Pages) ->
    Pages;
crawl(D,Pages) ->
    crawl(D-1,
         map_reduce_seq(fun mapper/2, fun reducer/2,
                        Pages)).
```

- Repeated map-reduce is often useful

# Parallelising Map-Reduce

- Divide the input into M chunks, map in parallel
  - About 64MB per chunk is good!
  - Typically M ~ 200,000 on 2,000 machines (~13TB)

- Divide the intermediate pairs into R chunks, reduce in parallel
  - Typically R ~ 5,000

**Problem:** all {K,V} with the same key must end up in the same chunk!

# Chunking Reduce

- All pairs with the same key must end up in the same chunk
- *Map keys to chunk number: 0..R-1*
  - e.g. hash(Key) rem R

erlang:phash2(Key,R)

- Every mapper process generates inputs for all R reducer processes

# A Naïve Parallel Map-Reduce

```
map_reduce_par(Map,M,Reduce,R,Input) ->
    Parent = self(),
    Splits = split_into(M,Input),
    Mappers =
      [spawn_mapper(Parent,Map,R
       || Split <- Splits],
    Mappeds =
      [receive {Pid,L} -> L end || Pid
    Reducers =
      [spawn_reducer(Parent,Red
       || I <- lists:seq(0
    Reduceds =
      [receive {Pid,L} -> L end || Pid <- Reducers],
    lists:sort(lists:flatten(Reduceds)).
```

Spawn a

Mappers send

Spawn a reducer for

Combine and sort the results

# Mappers

```
spawn_mapper(Parent,Map,R,Split) ->
    spawn_link(fun() ->
      Mapped =
        %% tag each pair with its hash
        [{erlang:phash2(K2,R),{K2,V2}}
         || {K,V} <- Split,
            {K2,V2} <- Map(K,V)],
      Parent !
        %% group pairs by hash tag
        {self(),group(lists:sort(Mapped))}
    end).
```

# Reducers

```
spawn_reducer(Parent,Reduce,I,Mappeds) ->
    %% collect pairs destined for reducer I
    Inputs = [KV
              || Mapped <- Mappeds,
                 {J,KVs} <- Mapped,
                 I==J,
                 KV <- KVs],
    %% spawn a reducer just for those inputs
    spawn_link(fun() ->
      Parent !
        {self(),reduce_seq(Reduce,Inputs)}
    end).
```

# Results

- Despite naïvety, the examples presented run *more than twice as fast* on a 2-core laptop

# Why is this naïve?

- All processes run in one Erlang node—real map-reduce runs on a cluster
- We start *all* mappers and *all* reducers at the same time—would overload a real system
- All data passes through the ”master” process—needs far too much bandwidth

# Data Placement

- Data is kept in the *file system*, not in the master process
  - the master just tells workers where to find it
- Two kinds of files:
  - *replicated* on 3+ nodes, survive crashes
  - *local* on one node, lost on a crash
- Inputs & outputs to map-reduce are replicated, intermediate results are local
- Inputs & outputs are not collected in one place, they remain distributed

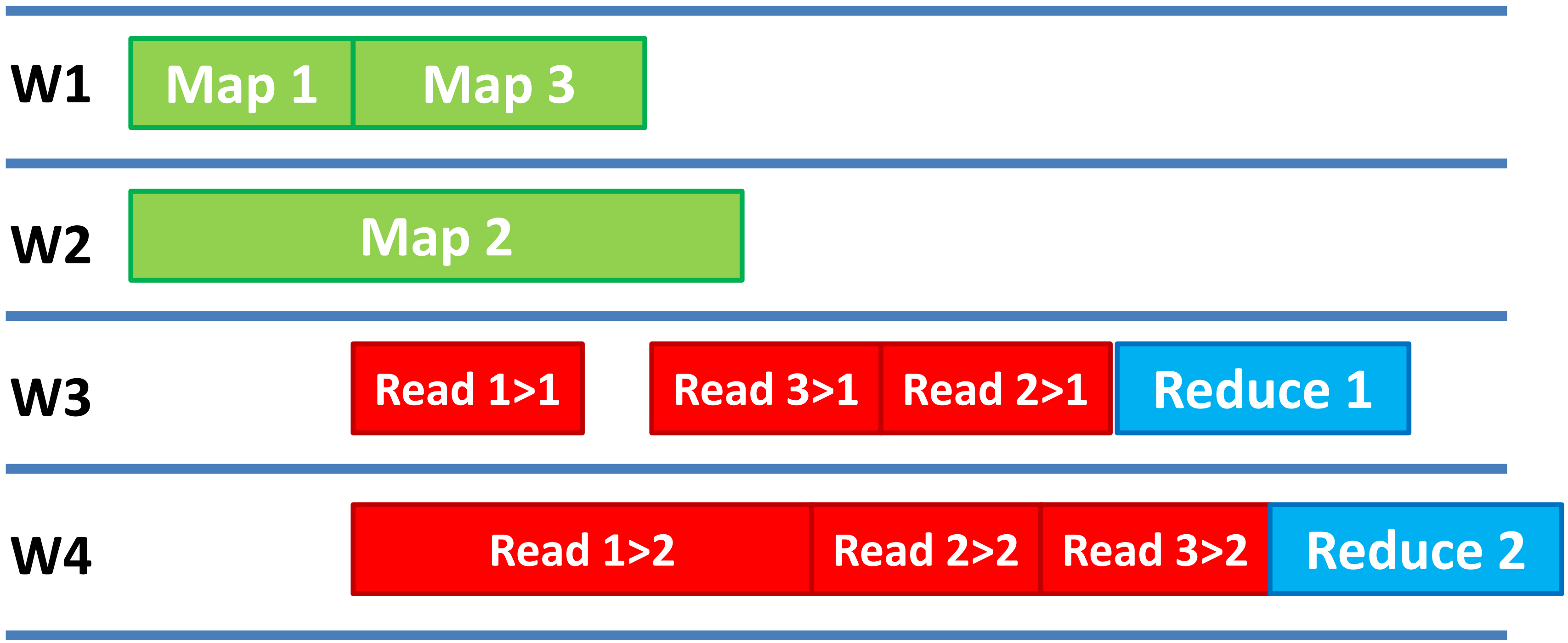
# Intermediate values

- Each mapper generates R local files, containing the data intended for each reducer
  - Optionally reduces each file locally
- Each reducer reads a file from each mapper, by rpc to the node where it is stored
- Mapper results on nodes which crash are regenerated on another node

# Master process

- Spawns a limited number of workers
- Sends mapper and reducer jobs to workers, sending new jobs as soon as old ones finish
- Places jobs close to their data if possible
- Tells reducers to start fetching each mapper output as soon as it is available

# A possible schedule



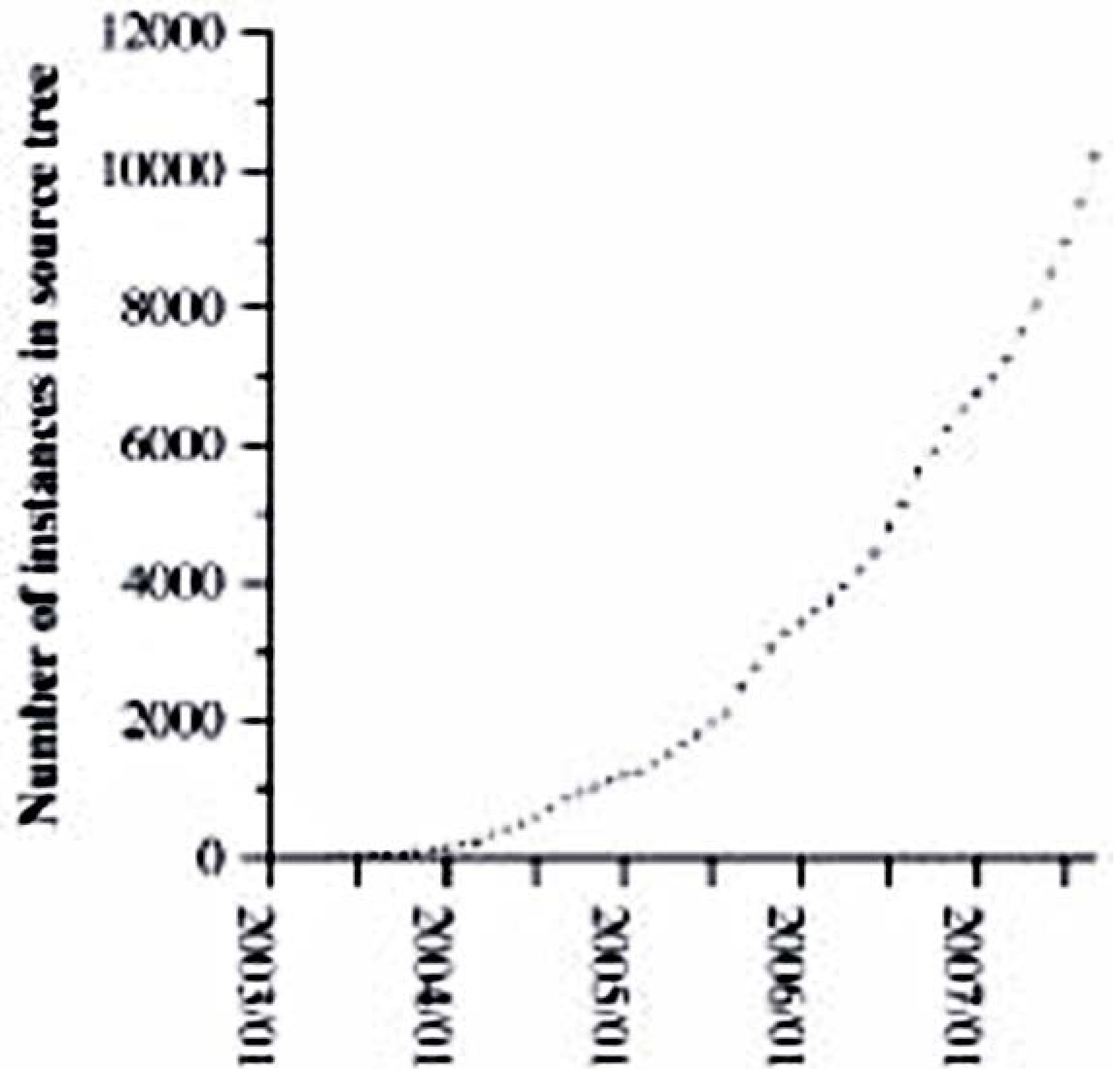

Each reduce worker starts to read map output as soon as possible

# Fault tolerance

- Running jobs on nodes that fail are restarted on others (Need to detect failure, of course)
- Completed maps are rerun on new nodes
  - because their results may be needed
- Completed reduce jobs leave their output in *replicated* files—no need to rerun
- Close to the end, remaining jobs are replicated
  - Some machines are just slow

“During one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines and continued to make forward progress, eventually completing the MapReduce operation.”

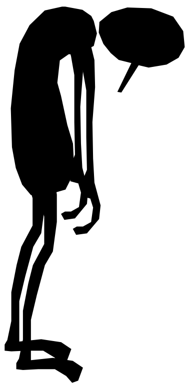
# Usage



12000
10000
8000
6000
4000
2000
0
Number of instances in source tree
2003/01
2004/01
2005/01
2006/01
2007/01

# Google web search indexing

**Before**



3800

LOC

**After**

700

LOC

# Experience

“Programmers find the system easy to use: more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google’s clusters every day, processing a total of more than twenty petabytes of data per day.”

*From MapReduce: Simplified Data Processing on Large Clusters by Jeffrey Dean and Sanjay Ghemawat, CACM 2008*

# Applications

- large-scale machine learning
- clustering for Google News and Froogle
- extracting data to produce reports of popular queries
  - e.g. Google Zeitgeist and Google Trends
- processing of satellite imagery
- language model processing for statistical machine translation
- large-scale graph computations.
- Apache Hadoop

# Map-Reduce in Erlang

- Functional programming concepts underlie map-reduce (although Google use C++)
- Erlang is very suitable for implementing it
- Nokia Disco—www.discoproject.org
  - Used to analyze tens of TB on over 100 machines
  - Multiple masters
- Riak MapReduce
  - Improves locality in applications of the Riak no-SQL key-value store

# Reading: one of

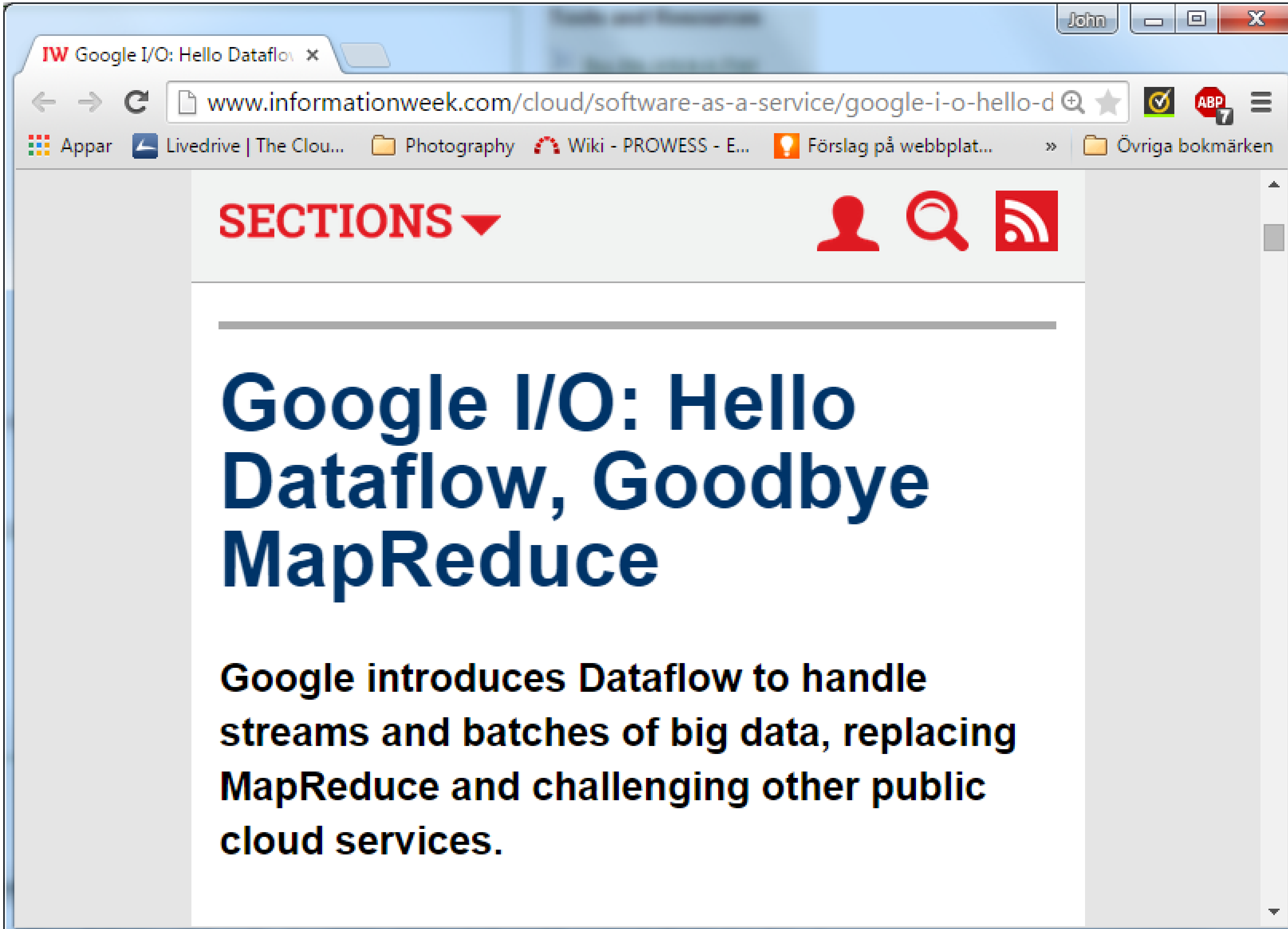
- The original OSDI 2004 paper (see earlier)

- *MapReduce: simplified data processing on large clusters,* Jeffrey Dean and Sanjay Ghemawat

  In Communications of the ACM - 50th anniversary issue: 1958 – 2008, Volume 51 Issue 1, January 2008

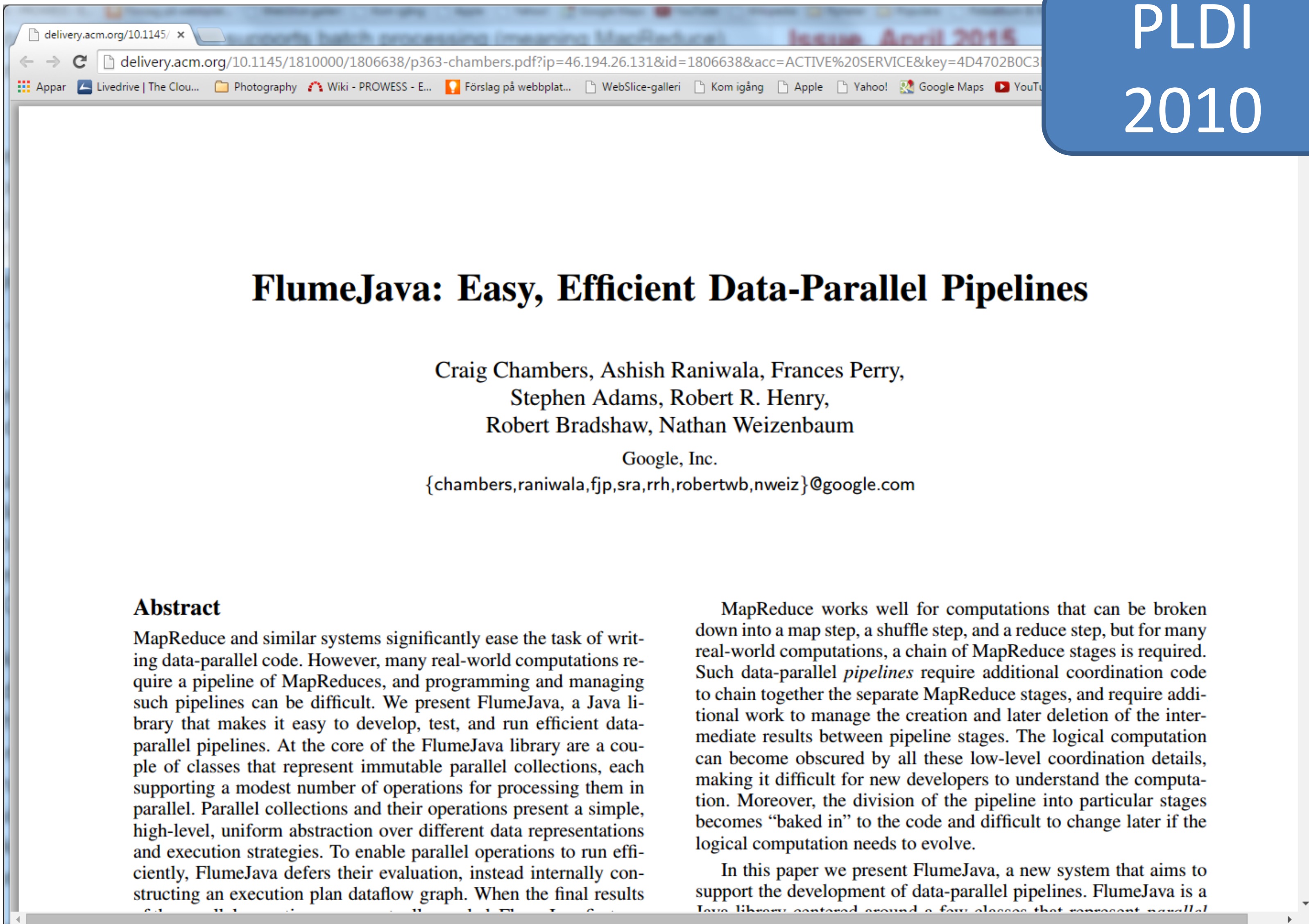
  – A shorter summary, some more up-to-date info

# You may have seen...



# What is it?

PLDI 2010



# FlumeJava: Easy, Efficient Data-Parallel Pipelines


Craig Chambers, Ashish Raniwala, Frances Perry,
Stephen Adams, Robert R. Henry,
Robert Bradshaw, Nathan Weizenbaum

Google, Inc.
{chambers,raniwala,fjp,sra,rrh,robertwb,nweiz}@google.com



## Abstract

MapReduce and similar systems significantly ease the task of writing data-parallel code. However, many real-world computations require a pipeline of MapReduces, and programming and managing such pipelines can be difficult. We present FlumeJava, a Java library that makes it easy to develop, test, and run efficient data-parallel pipelines. At the core of the FlumeJava library are a couple of classes that represent immutable parallel collections, each supporting a modest number of operations for processing them in parallel. Parallel collections and their operations present a simple, high-level, uniform abstraction over different data representations and execution strategies. To enable parallel operations to run efficiently, FlumeJava defers their evaluation, instead internally constructing an execution plan dataflow graph. When the final results


MapReduce works well for computations that can be broken down into a map step, a shuffle step, and a reduce step, but for many real-world computations, a chain of MapReduce stages is required. Such data-parallel *pipelines* require additional coordination code to chain together the separate MapReduce stages, and require additional work to manage the creation and later deletion of the intermediate results between pipeline stages. The logical computation can become obscured by all these low-level coordination details, making it difficult for new developers to understand the computation. Moreover, the division of the pipeline into particular stages becomes “baked in” to the code and difficult to change later if the logical computation needs to evolve.

In this paper we present FlumeJava, a new system that aims to support the development of data-parallel pipelines. FlumeJava is a

# What is it?

- A datatype of *immutable parallel collections*
  - which can be distributed over a data centre
  - or consist of *streaming data*
- An API including *map*, *reduce, filter*, *group…* that apply *pure functions* to collections
- An *optimising on-the-fly compiler* that converts FlumeJava pipelines to a sequence of MapReduce jobs…
- A *higher-level interface* built on top of MapReduce