Cloud Computing

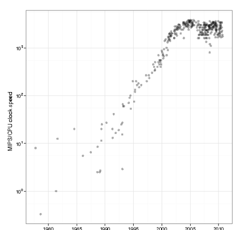
Ali Ghodsi
UC Berkeley, AMPLab
alig@cs.berkeley.edu

Background of Cloud Computing

- 1990: Heyday or parallel computing, multi-processors
  - Cannot make computers faster, they'll overheat, cannot make transistors smaller, etc.
  - Multiprocessors the only way to go
- But computers continued doubling in speed
  - Smaller transistors, better cooling, …

Multi-core Revolution

- 15-20 years later than predicted, we have hit the performance wall

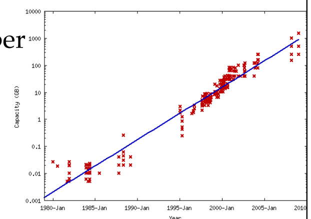


At the same time…

- Amount of stored data is exploding…

Data Deluge

- Billions of users connected through the net
  - WWW, FB, twitter, cell phones, …
  - 80% of the data on FB was produced last year
- Storage getting cheaper
  - Store evermore data



Solving the Impedance Mismatch

- Computers not getting faster,
  drowning in data
  - How to resolve the dilemma?
- Solution adopted by web-scale companies
  - Go massively *distributed* and *parallel*

Enter the World of Distributed Systems

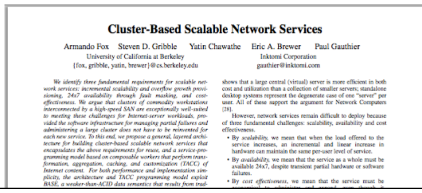
- Distributed Systems/Computing
  - *Loosely coupled* set of computers, communicating through message passing, solving a common goal
- Distributed computing is *challenging*
  - Dealing with *partial failures* (examples?)
  - Dealing with *asynchrony* (examples?)
- Disitributed Computing vs Parallel Computing?
  - distributed computing=parallel computing+partial failures

Dealing with Distribution

- Some tools to help distributed programming
  - Message Passing Interface (MPI)
  - Distributed Shared Memory (DSM)
  - Remote Procedure Calls (RPC)
  - RMI, WS, SOA
- Distributed programming still very hard

Nascent Cloud Computing

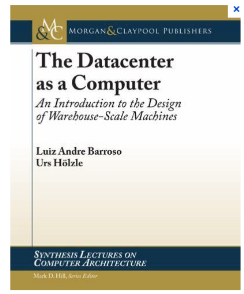
- Inktomi, founded by Eric Brewer/UCB
  - Pioneered most of the concepts of cloud computing
  - First step toward an **operating system for the datacenter**



Cluster-Based Scalable Network Services

Armando Fox Steven D. Gribble Yatin Chawathe Eric A. Brewer Paul Gauthier
University of California at Berkeley Inktomi Corporation
{fox, gribble, yatin, brewer}@cs.berkeley.edu gauthier@inktomi.com

The Datacenter is the new Computer



Datacenter OS

- If the datacenter is the new computer
  - what is it's **operating system**?
  - NB: not talking of a host OS

Classical Operating Systems

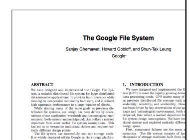
- Data sharing
  - IPC, files, pipes, …
- Programming Abstractions
  - Libraries (libc), system calls, …
- Multiplexing of resources
  - Time sharing, virtual memory, …

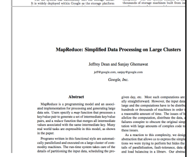
Datacenter Operating System

- Data sharing
  - Google File System, key/value stores
- Programming Abstractions
  - Google MapReduce, PIG, Hive, Spark
- Multiplexing of resources
  - Mesos, YARN, ZooKeeper, BookKeeper…

Google Cloud Infrastructure

- Google File System (GFS), 2003
  - Distributed File System for entire cluster
  - Single namespace
- Google MapReduce (MR), 2004
  - Runs queries/jobs on data
  - Manages work distribution & fault-tolerance
  - Colocated with file system
- Open source versions Hadoop DFS and Hadoop MR

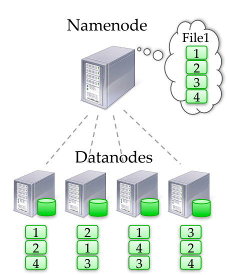

The Google File System
Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google


MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat
Google, Inc.

Google File System (GFS) Hadoop Distributed File System (HDFS)

GFS/HDFS Architecture

- Files split into blocks
- Blocks replicated across several *datanodes*
- *Namenode* stores metadata (file names, locations, etc)



GFS/HDFS Insights

- *Petabyte* storage
  - Large block sizes (128 MB)
  - Less metadata about blocks enables centralized architecture
  - Big blocks allow high throughput sequential reads/ writes
- Data *striped* on hundreds/thousands of servers
  - Scan 100 TB on 1 node @ 50 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

GFS/HDFS Insights (2)

- *Failures* will be the norm
  - Mean time between failures for 1 node = 3 years
  - Mean time between failures for 1000 nodes = 1 day
- Use *commodity* hardware
  - Failures are the norm anyway, buy cheaper hardware
- No complicated consistency models
  - Single writer, append-only data

MapReduce

MapReduce Model

- Data type: key-value **records**
- **Map** function:

  $(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$
- Group all identical K_{inter} values and pass to reducer
- **Reduce** function:

  $(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$

Example: Word Count

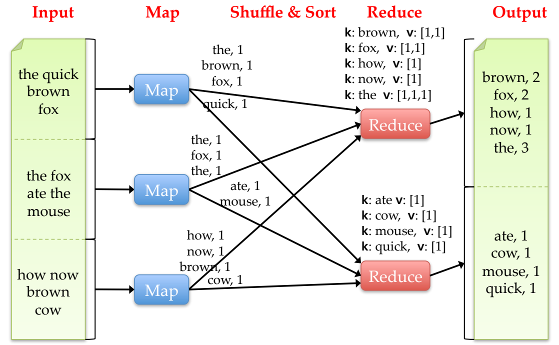
Input: key is filename, value is a line in input file

```
def mapper(file, line):
    foreach word in line.split():
        output(word, 1)
```

Intermediate: key is a word, value is 1

```
def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution



What is MapReduce Used For?

- At **Google**:
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At **Yahoo!**:
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail
- At **Facebook**:
  - Data mining
  - Ad optimization
  - Spam detection

MapReduce Model Insights

- Restricted model
  - Same **fine-grained operation** (m & r) repeated on big data
  - Operations must be **deterministic**
  - Operations must have **no side effects**
  - Only communication is through the shuffle
  - Operation (m & r) output saved (on disk)

MapReduce pros

- Distribution is completely **transparent**
  - Not a single line of distributed programming
- Automatic **fault-tolerance**
  - Determinism enables running failed tasks somewhere else again
  - Saved intermediate data enables just re-running failed reducers

MapReduce pros

- Automatic **scaling**
  - As operations as side-effect free, they can be distributed to any number of machines dynamically
- Automatic **load-balancing**
  - Move tasks and speculatively execute duplicate copies of slow tasks (*stragglers)*

MapReduce cons

- Restricted programming model
  - Not always natural to express problems in
  - Low-level coding necessary
  - Little support for iterative jobs
  - High-latency (batch processing)
- Addressed by follow-up research
  - **Pig** and **Hive** for high-level coding
  - **Spark** for iterative and low-latency jobs

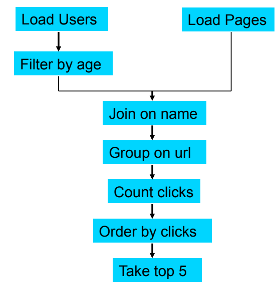
PIG & Hive

Pig

- High-level language:
  - Expresses sequences of MapReduce jobs
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc)
  - Easy to plug in Java functions
- Started at Yahoo! Research
  - Runs about 50% of Yahoo!'s jobs

An Example Problem

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited pages by users aged 18-25.



Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

In MapReduce

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

In Pig Latin

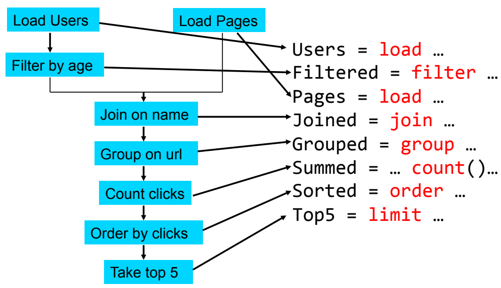
```
Users    = load 'users' as (name, age);
Filtered = filter Users by
               age >= 18 and age <= 25;
Pages    = load 'pages' as (user, url);
Joined   = join Filtered by name, Pages by user;
Grouped  = group Joined by url;
Summed   = foreach Grouped generate group,
              count(Joined) as clicks;
Sorted   = order Summed by clicks desc;
Top5     = limit Sorted 5;

store Top5 into 'top5sites';
```

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

Translation to MapReduce

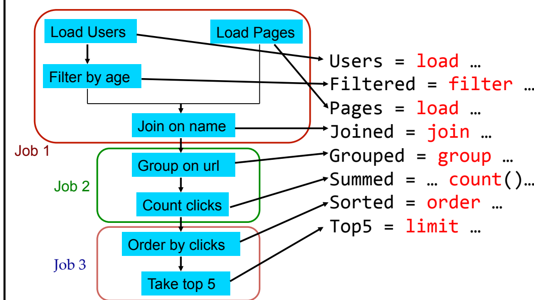
Notice how naturally the components of the job translate into Pig Latin.



Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

Translation to MapReduce

Notice how naturally the components of the job translate into Pig Latin.



Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt

Hive

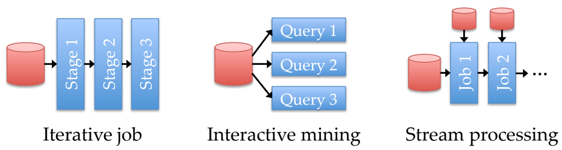
- Relational database built on Hadoop
  - Maintains table schemas
  - SQL-like query language (which can also call Hadoop Streaming scripts)
  - Supports table partitioning, complex data types, sampling, some query optimization
- Developed at Facebook
  - Used for most Facebook jobs

Spark

Spark Motivation

Complex jobs, interactive queries and online processing all need one thing that MR lacks:

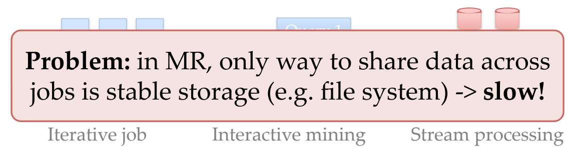
Efficient primitives for **data sharing**



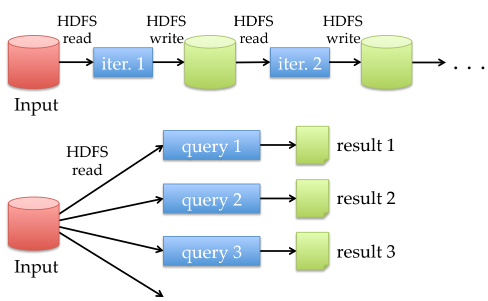
Spark Motivation

Complex jobs, interactive queries and online processing all need one thing that MR lacks:

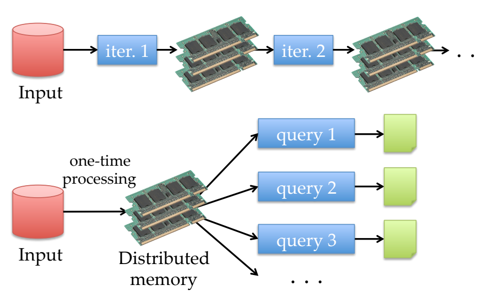
Efficient primitives for **data sharing**



Examples



Goal: In-Memory Data Sharing



10-100× faster than network and disk

Solution: Resilient Distributed Datasets (RDDs)

- Partitioned collections of records that can be stored in memory across the cluster
- Manipulated through a diverse set of transformations (*map*, *filter*, *join*, etc)
- Fault recovery without costly replication
  - Remember the series of transformations that built an RDD (its *lineage*) to *recompute* lost data
- **www.spark-project.org**

Mesos

Background

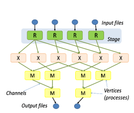
•Rapid innovation in cluster computing frameworks

Hadoop MapReduce

Google Pregel

Pig

RAILS



Dryad

CIEL

S4 distributed stream computing platform

Google Percolator

MPI2

Problem

•Rapid innovation in cluster computing frameworks

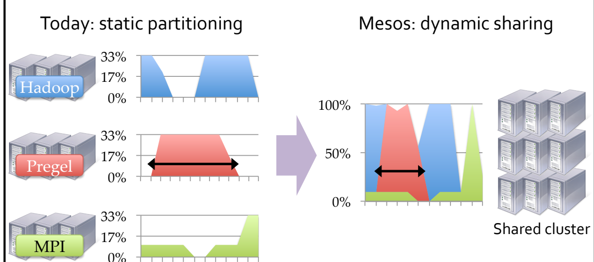
•No single framework optimal for all applications

•Want to run multiple frameworks in a single cluster

» …to *maximize utilization*

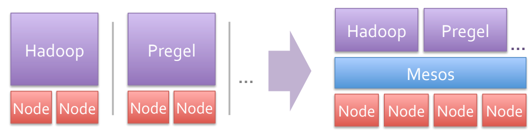
» …to *share data* between frameworks

Where We Want to Go



Solution

•Mesos is a common resource sharing layer over which diverse frameworks can run



Other Benefits of Mesos

•Run multiple instances of the *same* framework
  – Isolate production and experimental jobs
  – Run multiple versions of the framework concurrently

•Build *specialized frameworks* targeting particular problem domains
  – Better performance than general-purpose abstractions

Mesos Goals

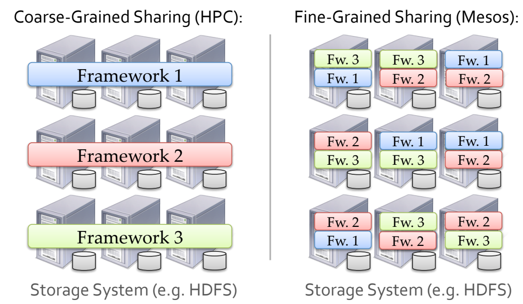
- **High utilization** of resources
- **Support diverse frameworks** (current & future)
- **Scalability** to 10,000's of nodes
- **Reliability** in face of failures

Resulting design: Small microkernel-like core that pushes scheduling logic to frameworks

Design Elements

- Fine-grained sharing:
  - Allocation at the level of *tasks* within a job
  - Improves utilization, latency, and data locality
- Resource offers:
  - Simple, scalable application-controlled scheduling mechanism

Element 1: Fine-Grained Sharing



+ Improved utilization, responsiveness, data locality

Element 2: Resource Offers

- Option: Global scheduler
  - Frameworks express needs in a specification language, global scheduler matches them to resources

+ Can make optimal decisions

•– Complex: language must support all framework needs

– Difficult to scale and to make robust

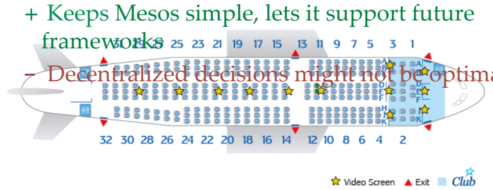
– Future frameworks may have unanticipated needs

Element 2: Resource Offers

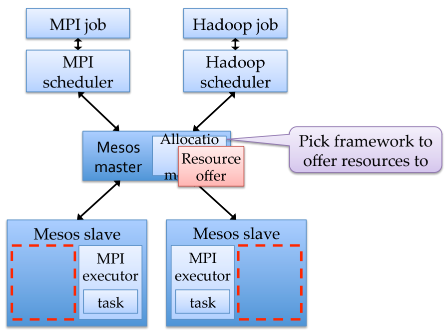
- Mesos: Resource offers
  - Offer available resources to frameworks, let them pick which resources to use and which tasks to launch

+ Keeps Mesos simple, lets it support future frameworks

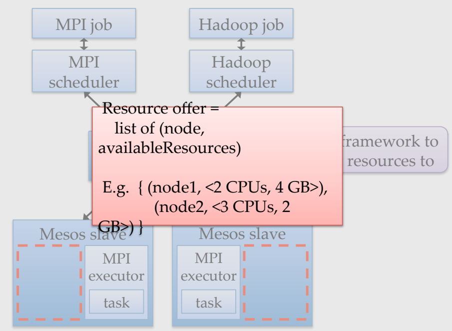
– Decentralized decisions might not be optimal



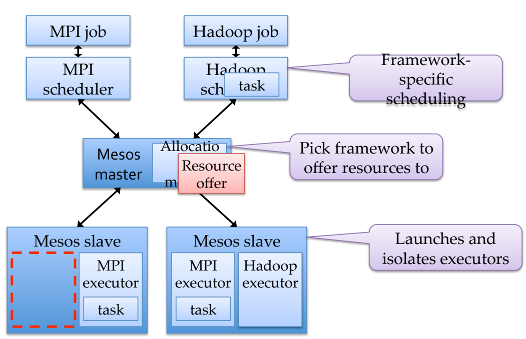
Mesos Architecture



Mesos Architecture



Mesos Architecture



Summary

- Cloud computing/datacenters are the new computer
  - Emerging "operating system" appearing
- Pieces of the OS
  - High-throughput filesystems (GFS/HDFS)
  - Job frameworks (MapReduce, Pregel)
  - High-level query languages (Pig, Hive)