"MapReduce"

[(big) data processing 'at scale']

Computational machinery to process large volumes of data

- Modern databases store data as key/value pairs, resulting in explosive growth when it comes to number of rows and file sizes.
- Traditional, sequential, single machine oriented access does NOT work at all - what is needed is a massively parallel way to process the data.

Note that we are talking about SIMD form of parallelism.

Aside: functional constructs

Before we look at 'MapReduce' etc., let us take a side trip and look at some functional programming features.

In functional programming, functions are first-class objects that can be passed into a function as arguments; a function can also be returned from a function as output. JavaScript, Python etc. are languages that provide functional programming facilities.

In Python, the built-in functions map(), filter() and reduce(), all accept a function as input.

lambda

A lambda operator lets us define anonymous, JIT functions, eg.

```
>>> f = lambda x, y : x + y # f is a var that receives an
anonymous fn assignment
```

```
>>> f(1,1)
2
```

map

```
r = map(func, seq)
```

map() applies the function func, to a sequence (eg. a list) seq.

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)
>>> print Fahrenheit
[102.56, 97.700000000000003, 99.140000000000001,
100.039999999999999]
>>> Centigrade = map(lambda x: (float(5)/9)*(x-32), Fahrenheit)
>>> print Centigrade
[39.200000000000003, 36.5, 37.300000000000004,
37.799999999999997]
>>>
```

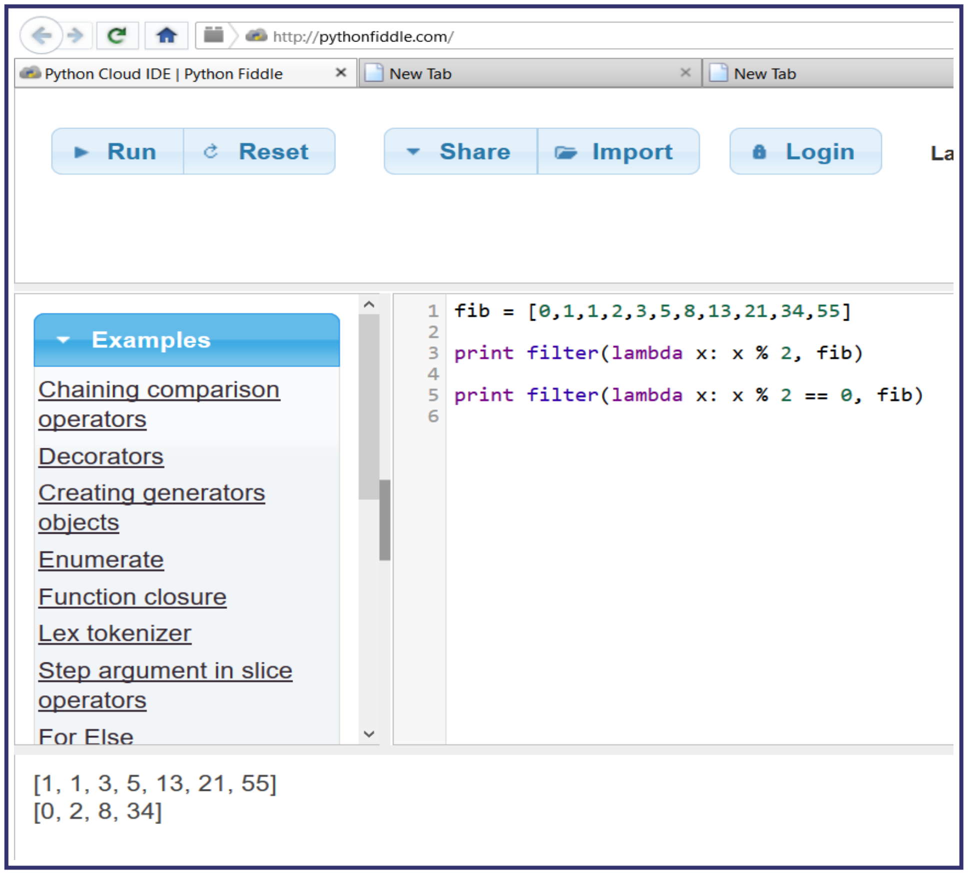
filter

```
r = filter(func, l)
```

filter() filters (outputs) all the elements of a list l, for which the passed-in function func returns True.

```
>>> fib = [0,1,1,2,3,5,8,13,21,34,55]

>>> # for an odd number x, x%2 is 1, ie. True; so the following
filters odd numbers
>>> result = filter(lambda x: x % 2, fib)
>>> print result
[1, 1, 3, 5, 13, 21, 55] # odd nums
>>> # to filter out even numbers, make it true that the mod has
no remainder, ie. x%2==0
>>> result = filter(lambda x: x % 2 == 0, fib)
>>> print result
[0, 2, 8, 34] # only evens
>>>
```

Here is a snapshot of the above two filter() calls, running in pythonfiddle:



```python
fib = [0,1,1,2,3,5,8,13,21,34,55]

print filter(lambda x: x % 2, fib)

print filter(lambda x: x % 2 == 0, fib)
```

```
[1, 1, 3, 5, 13, 21, 55]
[0, 2, 8, 34]
```

reduce

```python
r = reduce(func, l)
```

reduce() applies func repeatedly to the elements of l, to generate a single value and output it.

```
>>> reduce(lambda x,y: x+y, [47,11,42,13])
113
```

Out of map(), filter() and reduce(), map() and reduce() together have become a potent data processing combination ("MapReduce"!) in recent years, particularly when applied in a parallel fashion to Big Data.

MapReduce

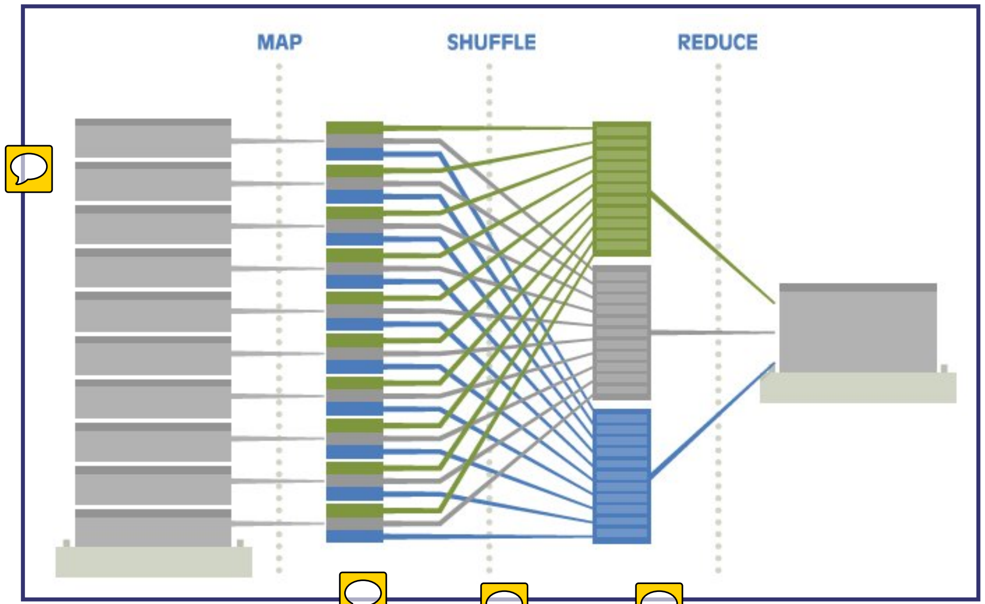
MapReduce is a programming paradigm invented at Google, one which has become wildly popular since it is designed to be applied to Big Data in NoSQL DBs, in data and disk parallel fashion - resulting in **dramatic** processing gains.

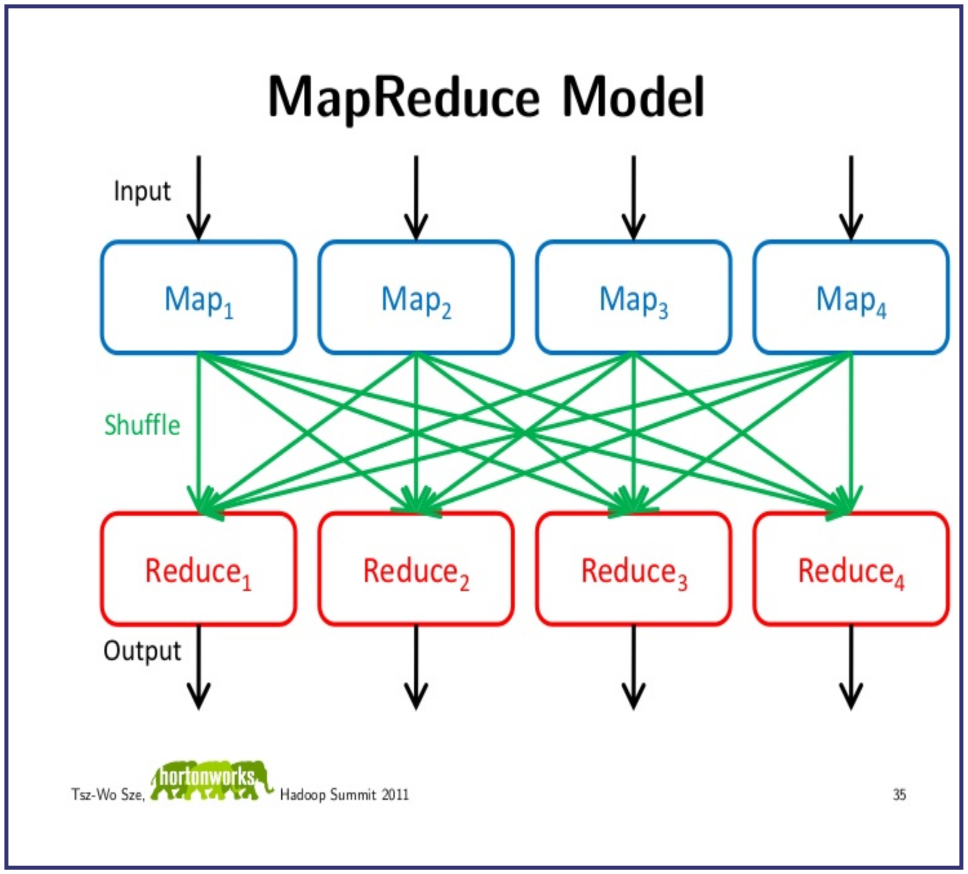
MapReduce works like this:

- [big] data is split into segments, held in a compute cluster
- a mapper task is run in parallel on all the segments (ie. in each cluster, therefore on each segment); each mapper produces output in the form of (key,value) pairs
- related key/value pairs from all mappers are forwarded to a shuffler (there are multiple shufflers); each shuffler consolidates its values into a list
- shufflers forward their keys and lists, to reducer tasks; each reducer processes its list of values and emits a single value (for its key)

The cluster user (programmer) only needs to supply a mapper task and a reducer task, the rest is automatically handled!

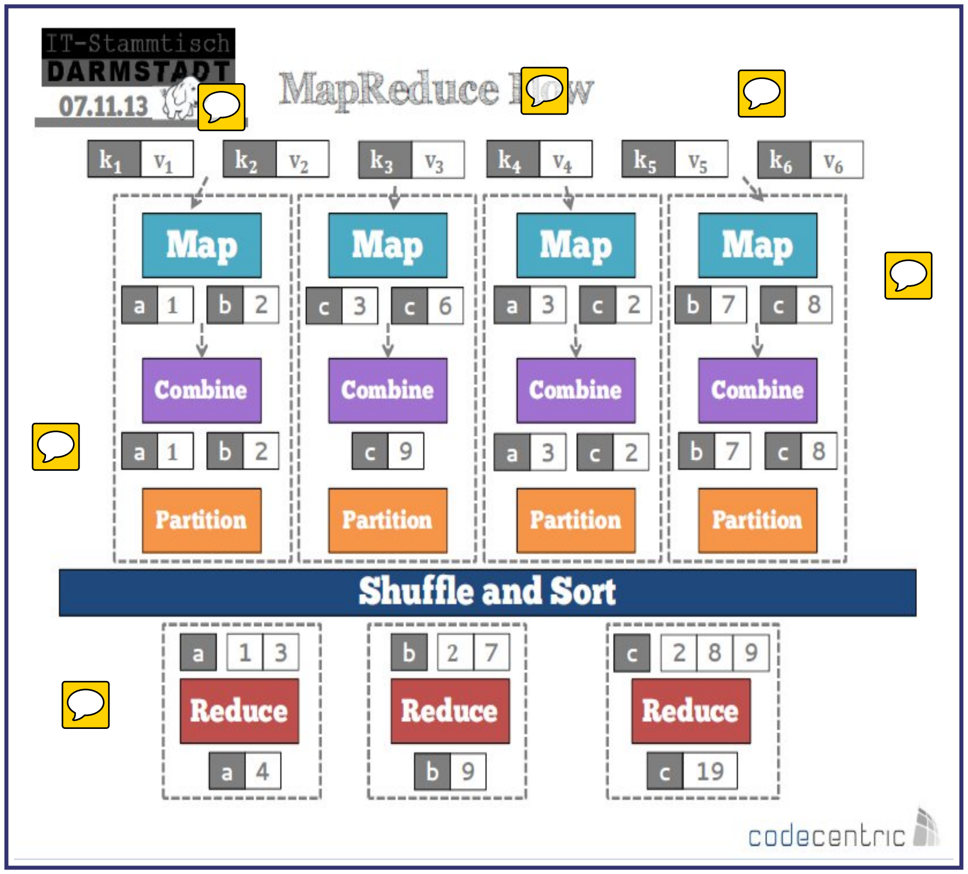
The following diagrams illustrate this elegant, embarrassingly simple idea.

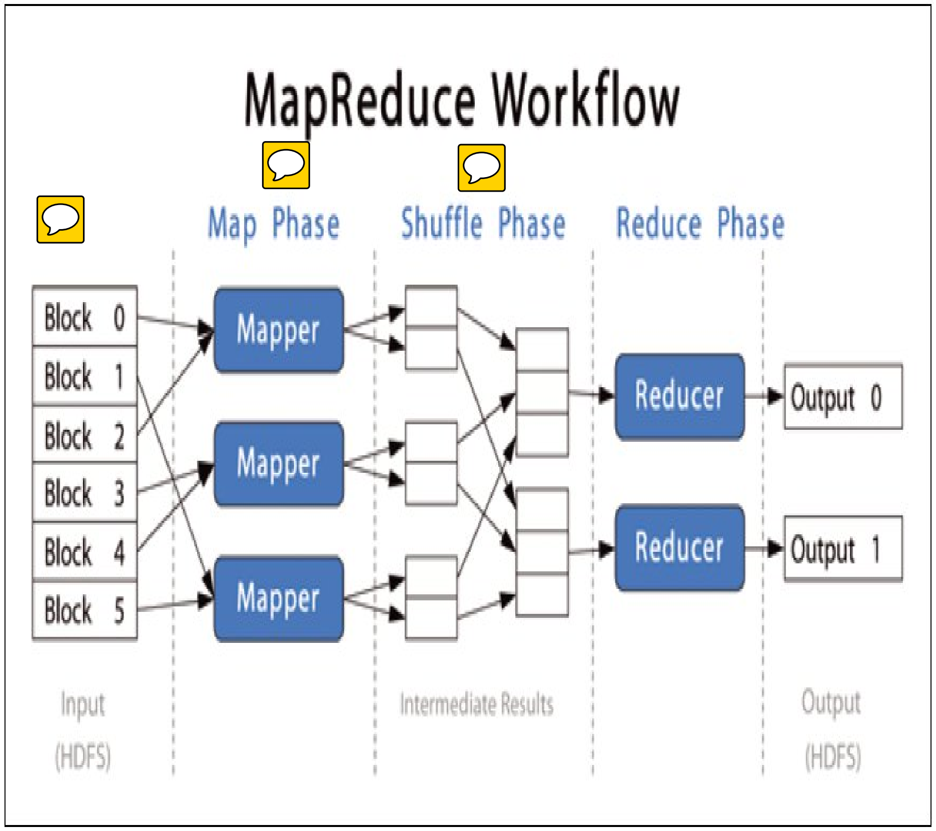


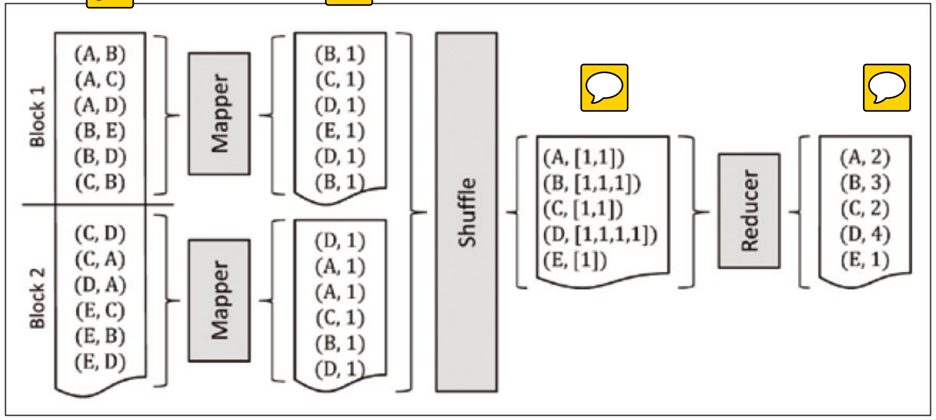


MapReduce Model

Tsz-Wo Sze, hortonworks Hadoop Summit 2011 35


IT-Stammtisch
DARMSTADT
07.11.13
MapReduce Flow
k_1 v_1
k_2 v_2
k_3 v_3
k_4 v_4
k_5 v_5
k_6 v_6
Map
a 1 b 2
c 3 c 6
a 3 c 2
b 7 c 8
Combine
a 1 b 2
c 9
a 3 c 2
b 7 c 8
Partition
Shuffle and Sort
a 1 3
b 2 7
c 2 8 9
Reduce
a 4
b 9
c 19
codecentric


MapReduce Workflow
Map Phase
Shuffle Phase
Reduce Phase
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Mapper
Mapper
Mapper
Reducer
Reducer
Output 0
Output 1
Input
(HDFS)
Intermediate Results
Output
(HDFS)


Block 1
(A, B)
(A, C)
(A, D)
(B, E)
(B, D)
(C, B)
Block 2
(C, D)
(C, A)
(D, A)
(E, C)
(E, B)
(E, D)
Mapper
Mapper
(B, 1)
(C, 1)
(D, 1)
(E, 1)
(D, 1)
(B, 1)
(D, 1)
(A, 1)
(A, 1)
(C, 1)
(B, 1)
(D, 1)
Shuffle
(A, [1,1])
(B, [1,1,1])
(C, [1,1])
(D, [1,1,1,1])
(E, [1])
Reducer
(A, 2)
(B, 3)
(C, 2)
(D, 4)
(E, 1)

Summary: "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."

GFS

Since MapReduce involves accessing (reading, writing) distributed (in clusters) data in parallel, there needs to be a high-performance, distributed file system that goes along with it - Google created GFS to fill this need.

GFS abstracts details of network file access so that remote reads/writes and local reads/writes are handled (in code) identically.

GFS differs from other distributed file systems such as (Sun's) NFS, in that the file system is implemented as a process in each machine's OS; striping is used to split each file and store the resulting chunks on several 'chunkservers', details of which are handled by a single master.

Hadoop: backstory

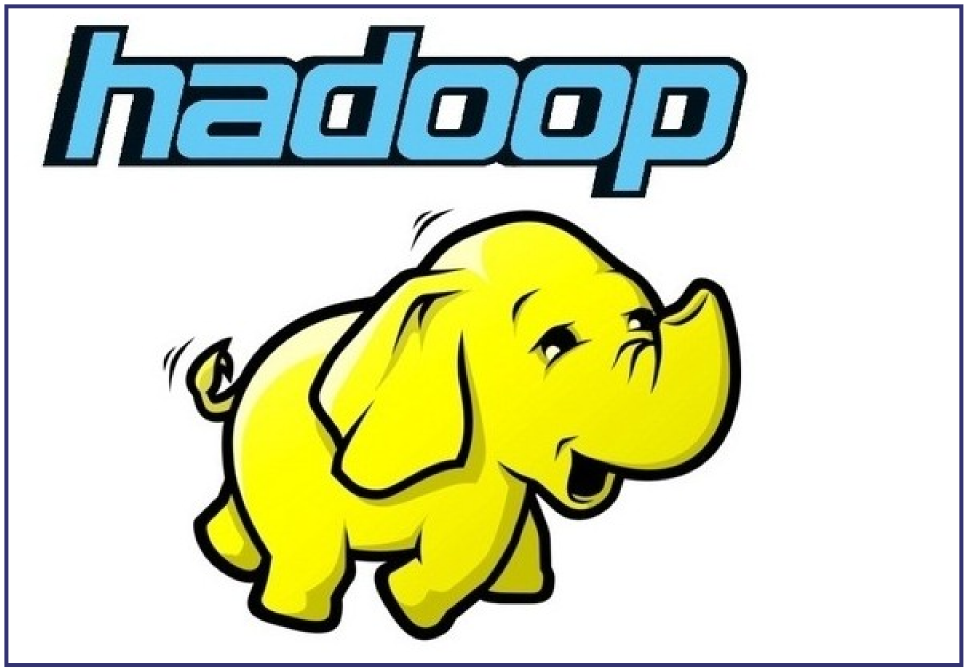
Doug Cutting and Mike Cafarella started 'Nutch' (a search engine project)in 2002.

Nutch had severe scalability problems.

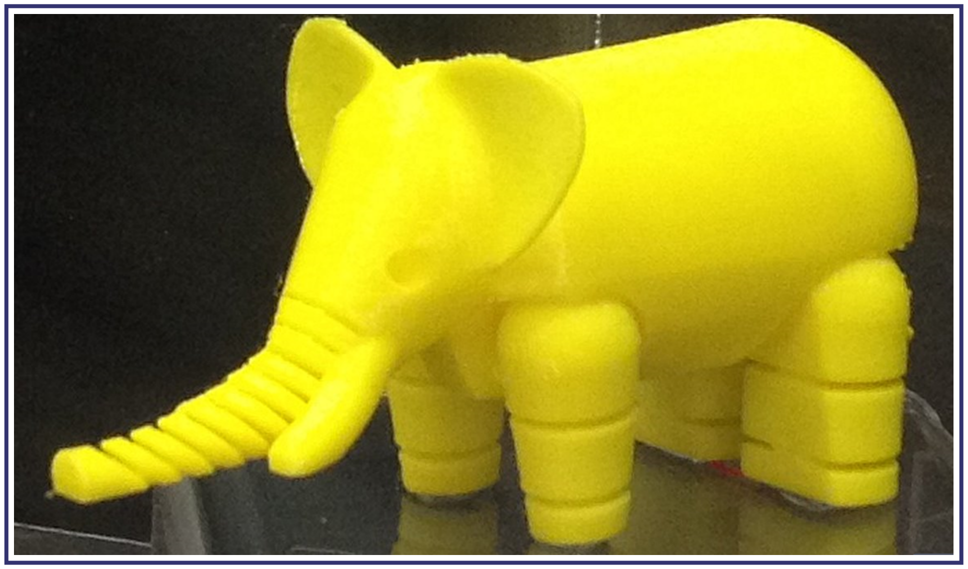
Even as Doug and Mike were struggling, Google published a paper on GFS, and a year later, another on MapReduce [links to the papers are in previous slides, you can find them here as well]; Doug and Mike gave up on their own work-in-progress, and implemented BOTH! Result: "Hadoop"..

Why "Hadoop"? Doug's son's toy elephant's name was "Hadoop" :) Read more here.

Here he is!

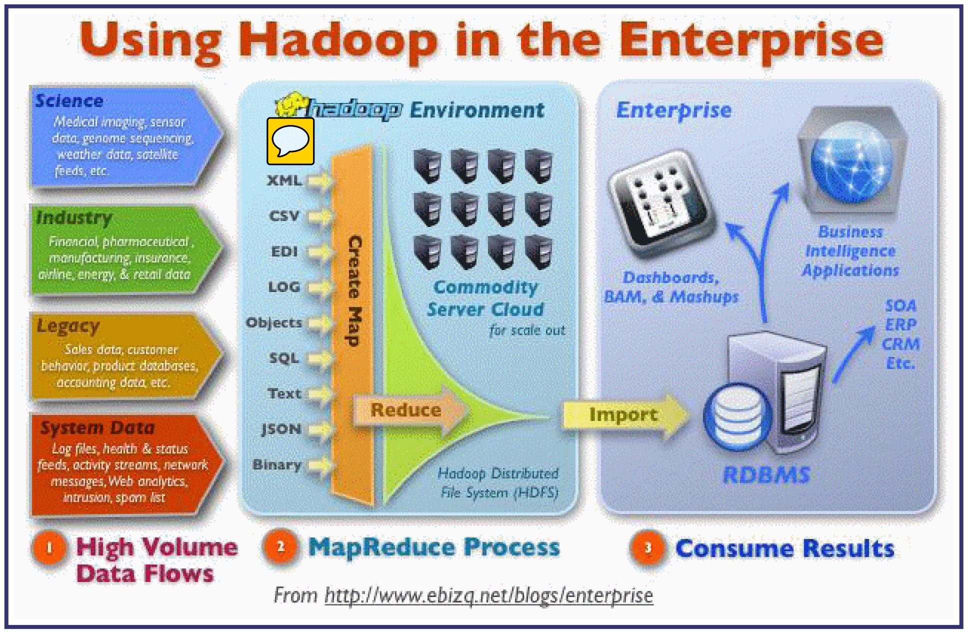


And in 3D too?!



Hadoop, HDFS

Hadoop is modeled after the MapReduce paradigm, and is utilized identically (by having users run mappers and reducers on (big) data).


Using Hadoop in the Enterprise
Science
Medical imaging, sensor data, genome sequencing, weather data, satellite feeds, etc.
Industry
Financial, pharmaceutical, manufacturing, insurance, airline, energy, & retail data
Legacy
Sales data, customer behavior, product databases, accounting data, etc.
System Data
Log files, health & status feeds, activity streams, network messages, Web analytics, intrusion, spam list
hadoop Environment
XML
CSV
EDI
LOG
Objects
SQL
Text
JSON
Binary
Create Map
Commodity Server Cloud
for scale out
Reduce
Hadoop Distributed File System (HDFS)
Import
Enterprise
Dashboards, BAM, & Mashups
Business Intelligence Applications
SOA
ERP
CRM
Etc.
RDBMS
1 High Volume Data Flows
2 MapReduce Process
3 Consume Results
From http://www.ebizq.net/blogs/enterprise

HDFS is modeled after Google's GFS, but with some important differences [read the paper to find out if you are interested] - as for similarities, there is a single master NameNode, and multiple DataNodes.

MR examples

WordCount is the 'Hello World' of Hadoop [counts the number of occurrences of each word in a given input set].

Following is the Java code for WordCount - note that it has both the mapper and reducer specified in it:

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
public class WordCount {

  public static class TokenizerMapper
       extends Mapper{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new
StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
  public static class IntSumReducer
       extends Reducer {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable values,
                       Context context
                       ) throws IOException, InterruptedException
{
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Below are the results of running this on a small two-file dataset.

```
Input:
$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World
$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

```
Run:
$ bin/hadoop jar wc.jar  WordCount  /user/joe/wordcount/input
/user/joe/wordcount/output

Output:
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000`
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

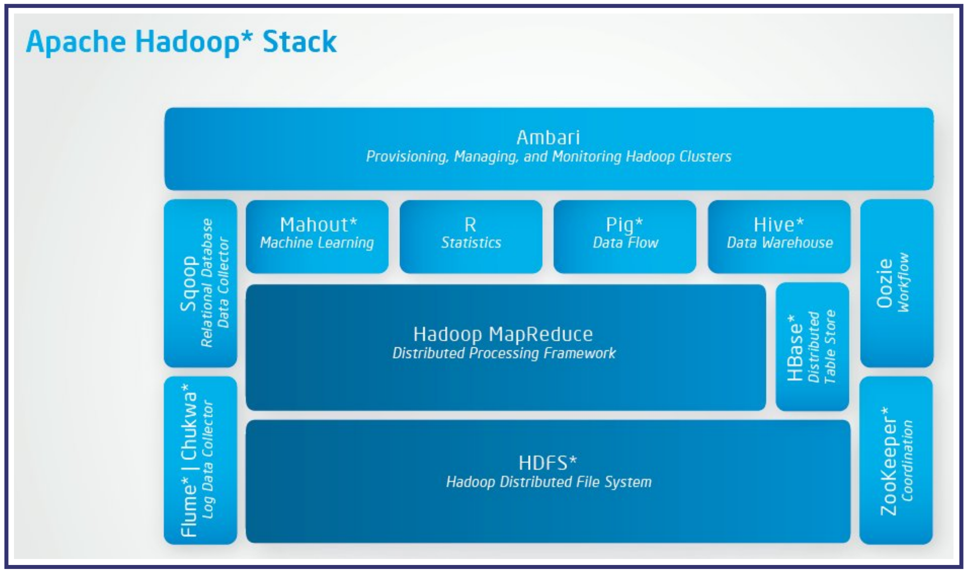
You can get more details here.

MR examples [cont'd]

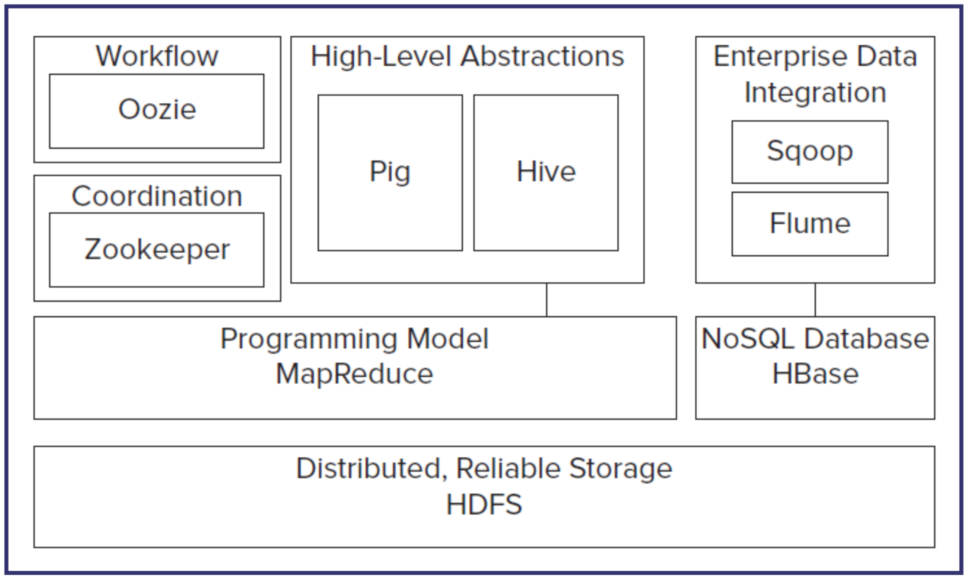
```
#mapper:
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)




# reducer:
#!/usr/bin/env python

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

The Hadoop 'ecosysem'

Because the core Hadoop system has been so popular, a whole bunch of associated projects have resulted, leading to a thriving 'ecosystem'. - don't feel overwhelmed though, you don't need to learn to use all these all at once!



Another view:


Workflow
Oozie
Coordination
Zookeeper
High-Level Abstractions
Pig
Hive
Enterprise Data Integration
Sqoop
Flume
Programming Model MapReduce
NoSQL Database HBase
Distributed, Reliable Storage HDFS

Hadoop: Hive

Hive provides a SQL-like scripting language called HQL. "Better than SQL" - eg. no need to create relational table schemas and populate with data.

"Hive translates most queries to MapReduce jobs, thereby exploiting the scalability of Hadoop, while presenting a familiar SQL abstraction."

Below is the WordCount task expressed in HiveQL - just 8 lines, compared to the standard MR Java code (shown earlier) of 53 lines!

```
CREATE TABLE docs (line STRING);
LOAD DATA INPATH 'docs' OVERWRITE INTO TABLE docs;
CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
  (SELECT explode(split(line, '\s')) AS word FROM docs) w
```

```
GROUP BY word
ORDER BY word;
```

Facebook was a major early contributor.

Hadoop: Pig

"Pig provides an engine for executing data flows in parallel on Hadoop. It includes a language, Pig Latin, for expressing these data flows. Pig Latin includes operators for many of the traditional data operations (join, sort, filter, etc.), as well as the ability for users to develop their own functions for reading, processing, and writing data."

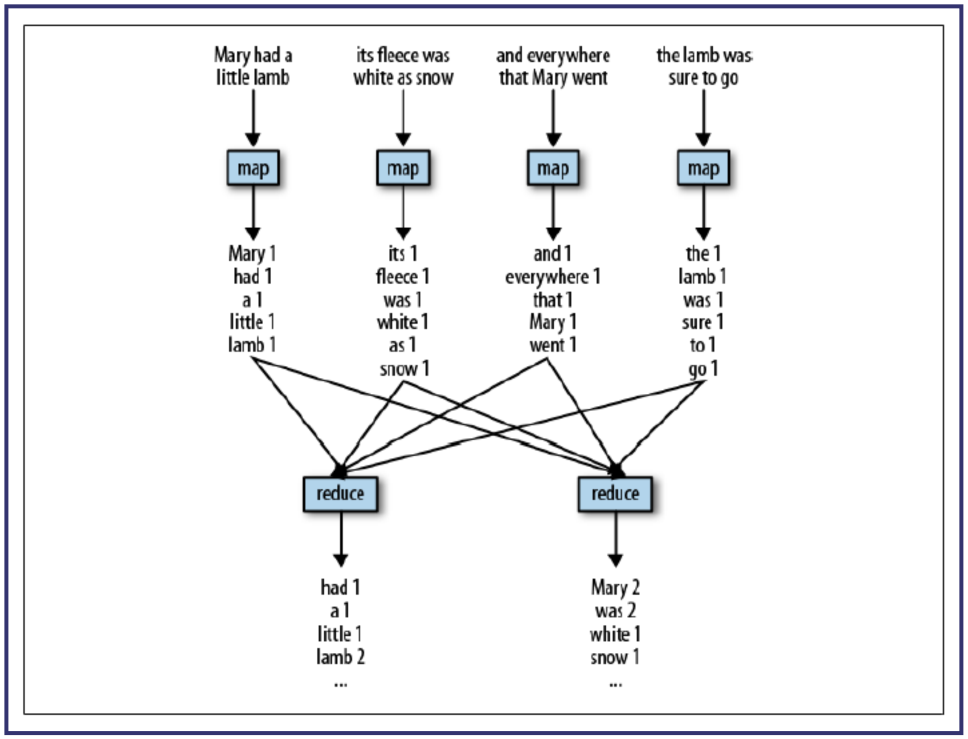
Pig Latin scripts are compiled into MR jobs that are then run on the cluster.

Here is a Pig Latin script for doing word counting, on the first stanza of Mary Had A Little Lamb:

```
-- Load input from the file named Mary, and call the single
-- field in the record 'line'.
input = load 'mary' as (line);
-- TOKENIZE splits the line into a field for each word.
-- flatten will take the collection of records returned by
-- TOKENIZE and produce a separate record for each one, calling
the single
-- field in the record word.
words = foreach input generate flatten(TOKENIZE(line)) as word;
-- Now group them together by each word.
grpd = group words by word;
-- Count them.
cntd = foreach grpd generate group, COUNT(words);
-- Print out the results.
dump cntd;
```

Note that with Pig Latin, there is no explicit spec of mapping and reducing phases.

For completeness, here is how we picture the underlying MR job running:

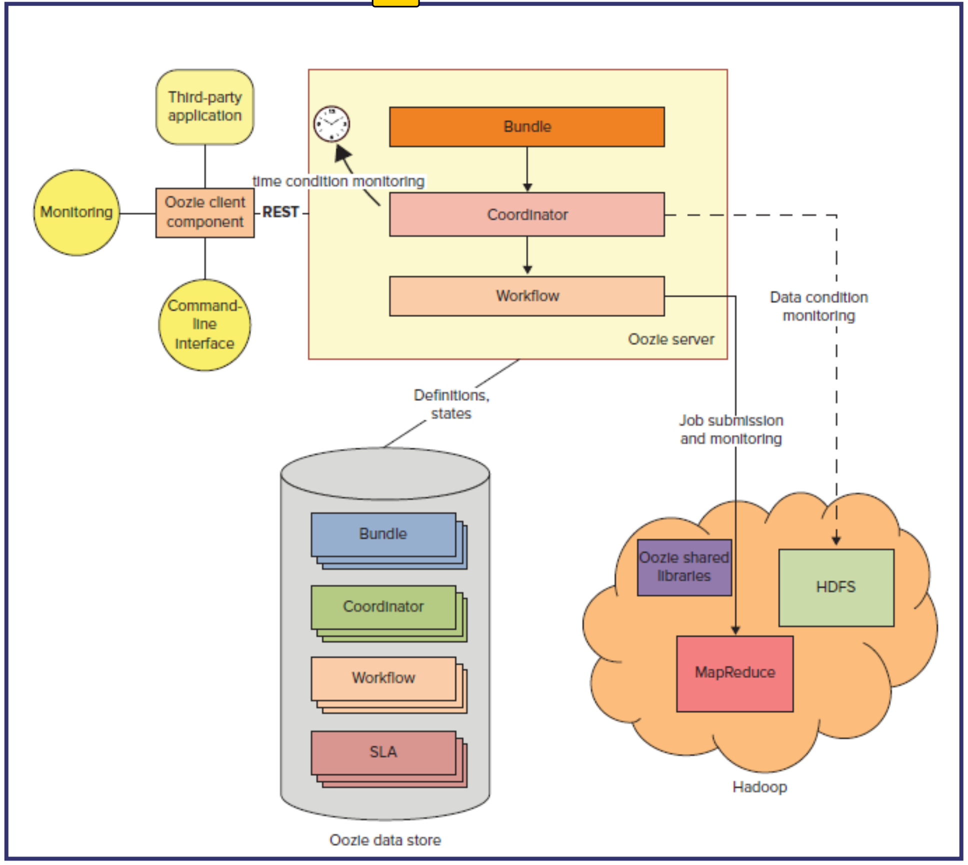

Mary had a little lamb
its fleece was white as snow
and everywhere that Mary went
the lamb was sure to go
map
map
map
map
Mary 1 had 1 a 1 little 1 lamb 1
its 1 fleece 1 was 1 white 1 as 1 snow 1
and 1 everywhere 1 that 1 Mary 1 went 1
the 1 lamb 1 was 1 sure 1 to 1 go 1
reduce
reduce
had 1 a 1 little 1 lamb 2 ...
Mary 2 was 2 white 1 snow 1 ...

"Pig Latin is a dataflow language. This means it allows users to describe how data from one or more inputs should be read, processed, and then stored to one or more outputs in parallel. These data flows can be simple linear flows like the word count example given previously. They can also be complex workflows that include points where multiple inputs are joined, and where data is split into multiple streams to be processed by different operators. To be mathematically precise, a Pig Latin script describes a directed acyclic graph (DAG), where the edges are data flows and the nodes are operators that process the data."

Yahoo! Research was a major developer of Pig.

Hadoop: Oozie

"A scalable workflow system, Oozie is integrated into the Hadoop stack, and is used to coordinate execution of multiple MapReduce jobs. It is capable of managing a significant amount of complexity, basing execution on external events that include timing and presence of required data."


Third-party application
Monitoring
Oozie client component
Command-line interface
REST
time condition monitoring
Bundle
Coordinator
Workflow
Oozie server
Data condition monitoring
Definitions, states
Job submission and monitoring
Bundle
Coordinator
Workflow
SLA
Oozie data store
Oozie shared libraries
HDFS
MapReduce
Hadoop

Hadoop: Musketeer

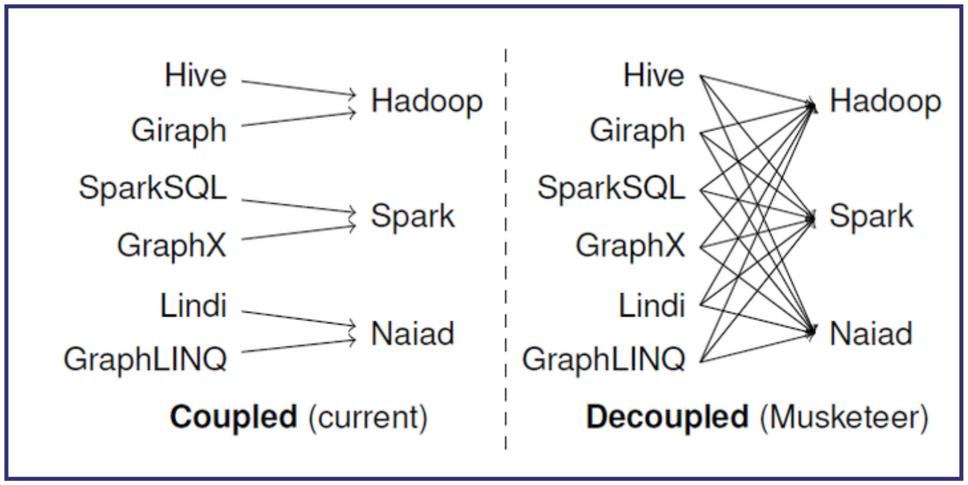
Currently, front end workflows (eg. written using Hive) are *coupled* with back-end engines (such as Hadoop), making them less usable than if these could be decoupled.

Musketeer is an experimental approach to do the decoupling. Three benefits:

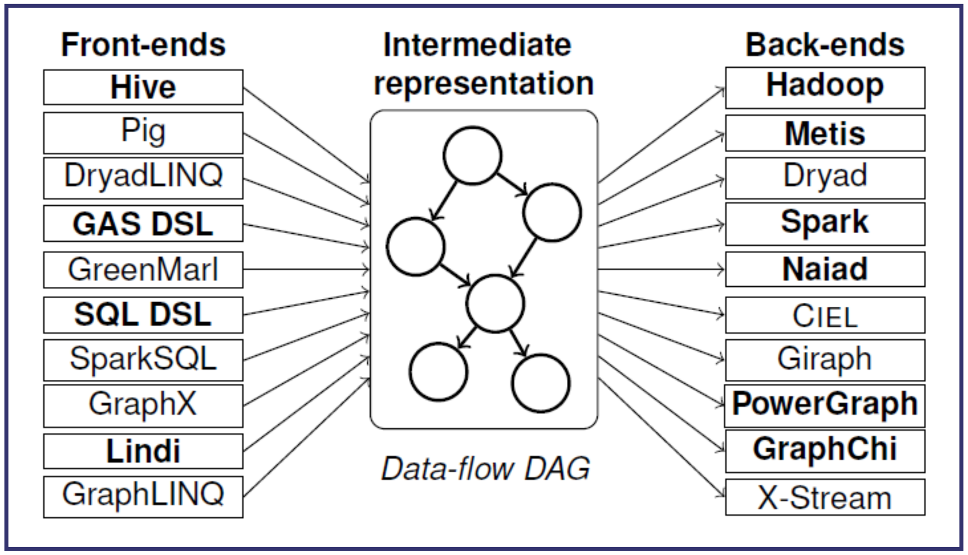
1. Users write their workflow once, in a way they choose, but can easily execute it on alternative systems;

2. Multiple sub-components of a workflow can be executed on different back-end systems; and

3. Existing workflows can easily be ported to new systems.



Hive
Giraph
SparkSQL
GraphX
Lindi
GraphLINQ
Hadoop
Spark
Naiad
Coupled (current)
Hive
Giraph
SparkSQL
GraphX
Lindi
GraphLINQ
Hadoop
Spark
Naiad
Decoupled (Musketeer)



Front-ends
Hive
Pig
DryadLINQ
GAS DSL
GreenMarl
SQL DSL
SparkSQL
GraphX
Lindi
GraphLINQ
Intermediate representation
Data-flow DAG
Back-ends
Hadoop
Metis
Dryad
Spark
Naiad
CIEL
Giraph
PowerGraph
GraphChi
X-Stream

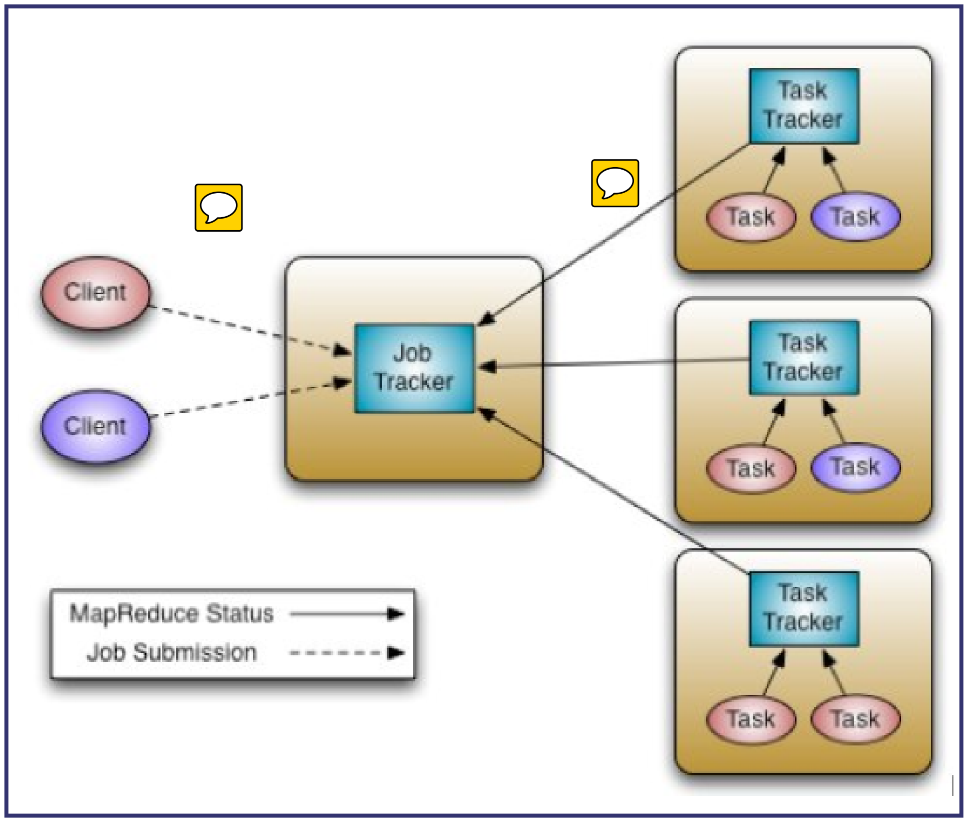
MRv2: YARN

YARN is "MapReduce v2".

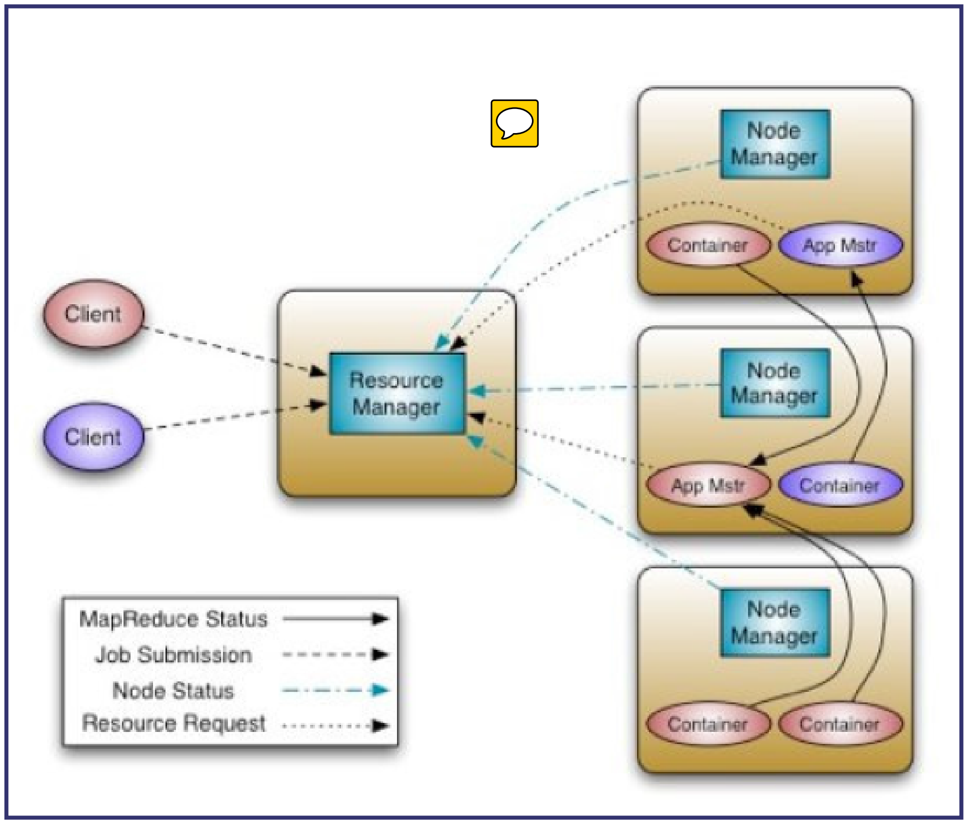
The first version of MR/Hadoop was 'batch oriented', meaning that static, distributed data was processed via mapping, shuffling and reducing steps.

YARN (Yet Another Resource Negotiator) on the other hand makes non-MR applications (eg. graph processing, iterative modeling) possible (but is fully backwards compatible with v.1.0, ie. can run MapReduce jobs), and offers better scalability and cluster utilization (compared to MRv1). It also makes it possible to create (near) real-time applications.

MRv1:


Client
Client
Job Tracker
Task Tracker
Task
Task
Task Tracker
Task
Task
Task Tracker
Task
Task
MapReduce Status
Job Submission

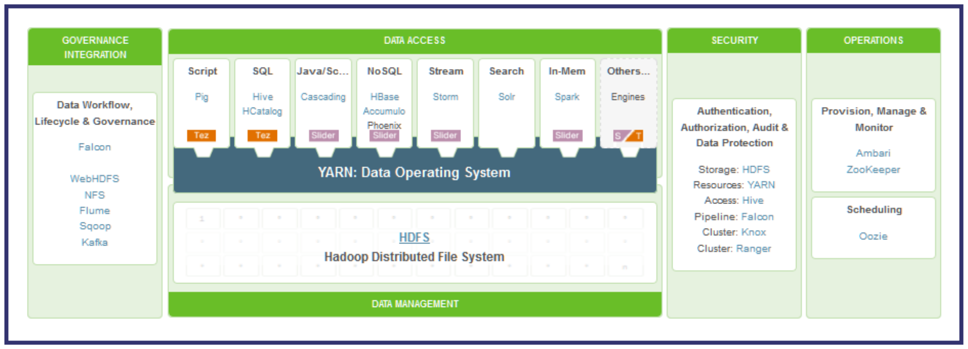
MRv2, ie. YARN:


Client
Client
Resource Manager
Node Manager
Container
App Mstr
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

DBs most often used with Hadoop

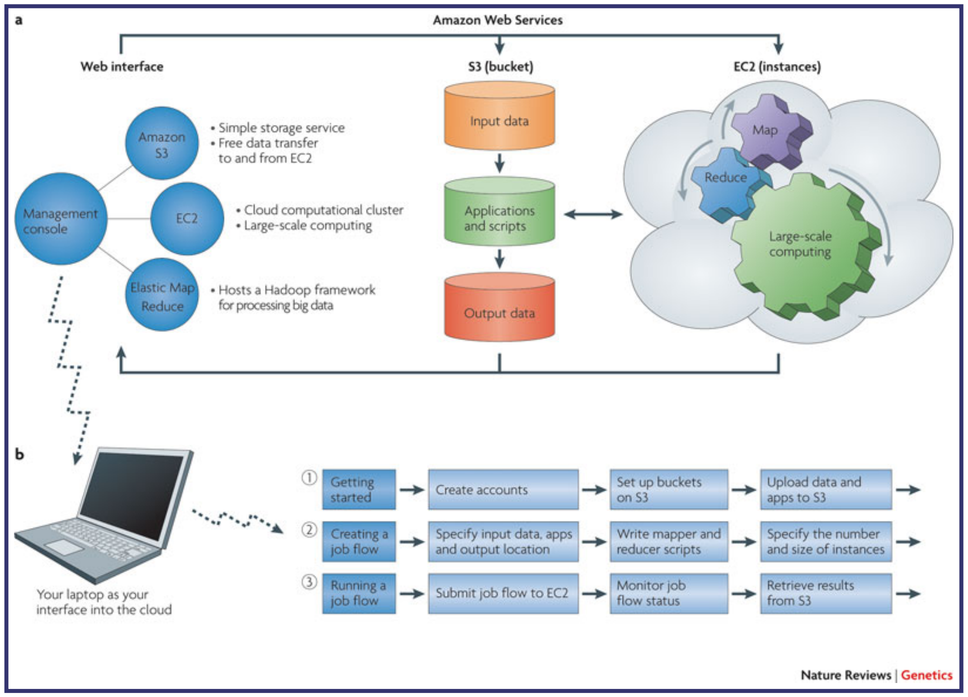
The following databases are most commonly used inside a Hadoop cluster:

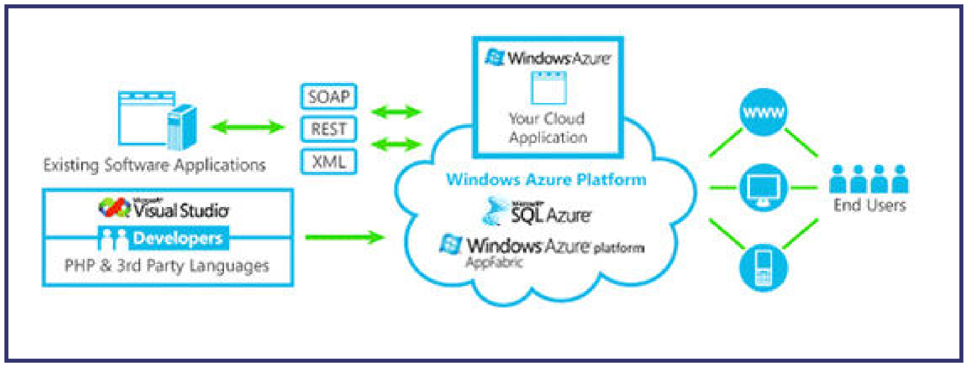
- MongoDB
- Cassandra
- HBase
- Hive
- Spark
- Blur
- Accumulo
- Memcached
- Solr
- Giraph


GOVERNANCE INTEGRATION
Data Workflow, Lifecycle & Governance
Falcon
WebHDFS
NFS
Flume
Sqoop
Kafka
DATA ACCESS
Script
Pig
Tez
SQL
Hive
HCatalog
Tez
Java/Sc...
Cascading
Slider
NoSQL
HBase
Accumulo
Phoenix
Slider
Stream
Storm
Slider
Search
Solr
In-Mem
Spark
Slider
Others...
Engines
S
T
YARN: Data Operating System
HDFS
Hadoop Distributed File System
DATA MANAGEMENT
SECURITY
Authentication, Authorization, Audit & Data Protection
Storage: HDFS
Resources: YARN
Access: Hive
Pipeline: Falcon
Cluster: Knox
Cluster: Ranger
OPERATIONS
Provision, Manage & Monitor
Ambari
ZooKeeper
Scheduling
Oozie

Cloud infrastructure

Amazon's EC2/S3, Microsoft's Azure, etc. offer a 'cloud platform' on which to build apps and services.


a
Amazon Web Services
Web interface
S3 (bucket)
EC2 (instances)
Amazon S3
• Simple storage service
• Free data transfer to and from EC2
Management console
EC2
• Cloud computational cluster
• Large-scale computing
Elastic Map Reduce
• Hosts a Hadoop framework for processing big data
Input data
Applications and scripts
Output data
Map
Reduce
Large-scale computing
b
Your laptop as your interface into the cloud
① Getting started → Create accounts → Set up buckets on S3 → Upload data and apps to S3 →
② Creating a job flow → Specify input data, apps and output location → Write mapper and reducer scripts → Specify the number and size of instances →
③ Running a job flow → Submit job flow to EC2 → Monitor job flow status → Retrieve results from S3 →
Nature Reviews | Genetics



They offer 'elastic scaling' (of resources), guaranteed uptime, speedy access, etc.

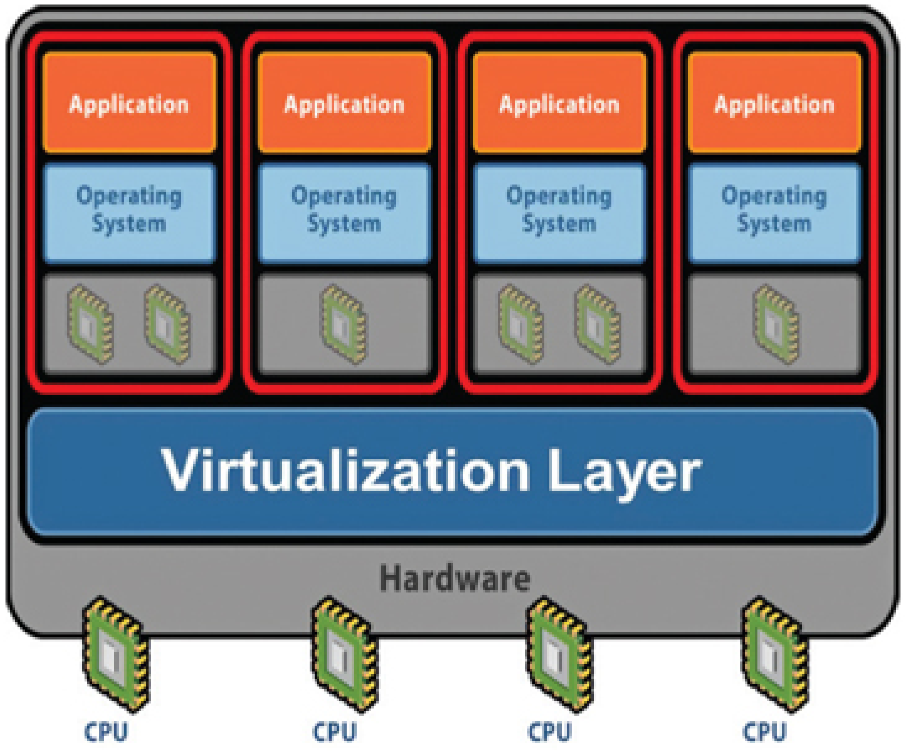
Your own Hadoop cluster: inside EC2, Azure..

One hassle-free way to set up a Hadoop compute cluster is to do so inside EC2 or Azure.

Note - setting these up can be quite tedious (and is thankfully a one shot thing!).

VM infrastructure

A virtual machine (VM) is a piece of software that runs on a host machine, to enable creating self-contained 'virtual' machines inside the host - these virtual machines can then serve as platforms on which to run programs and services.



Virtualization software allows applications that previously ran on separate computers to run on one server machine.

Your own Hadoop cluster: inside a virtual machine

Another way (not cloud based) to experiment with Hadoop is to download implementations meant for virtual machines, and load them into the VMs.

Here are a couple you can try [these are the most used ones, compared to the ones listed below]: HortonWorks' Hadoop Sandbox and MapR's MapR Sandbox.

In addition, you can also experiment with Oracle's 'Big Data Lite' VM, Cloudera's CDH VM, IBM's BigInsights QSE VM and Talend's Big Data Sandbox VM [this one packages an existing VM along with a custom platform and sample data].

There is also a Blaze VM for streaming processing SQL.

You might enjoy reading this blog post from VMWare's CTO, on their virtualizing Hadoop.

Beyond MR: Spark

Spark makes Big Data real-time and interactive - it is an in-memory data processing engine (so it is FAST), specifically meant for iterative processing of data.

Better efficiency: general execution graphs, in-memory data storage.

Being used at Yahoo!, Intel, Adobe, Quantifind, Conviva, Ooyala, Bizo, etc.

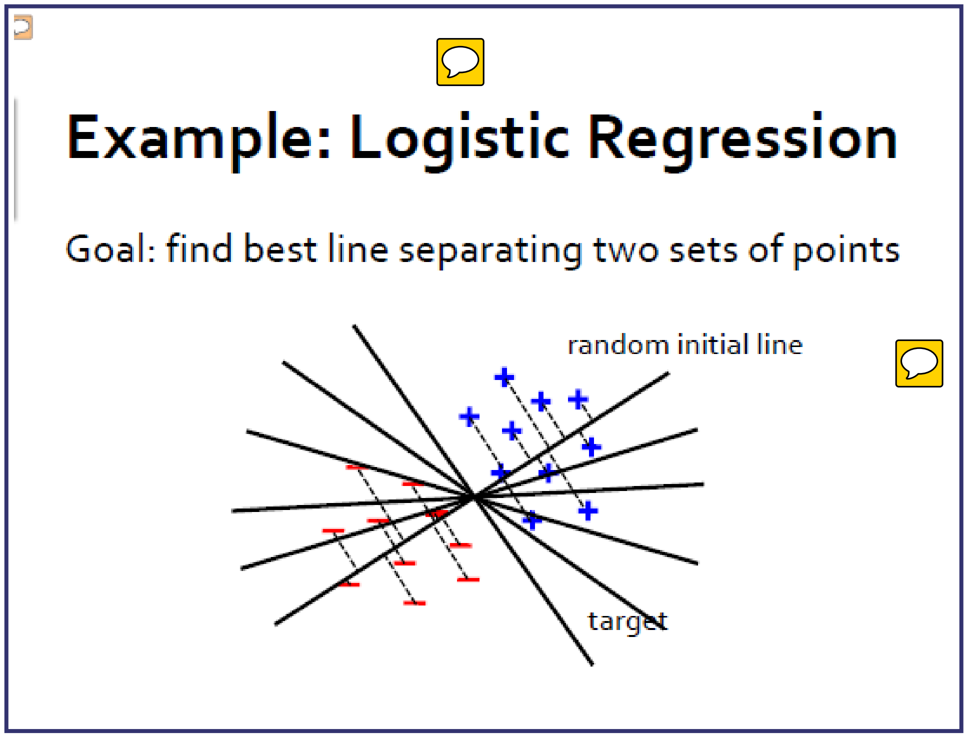
Query lang is SparkSQL (used to be called Shark; Shark itself was an alternative to Hive).

MR could not deal with complex (multi-pass) processing, interactive (ad-hoc) queries or real-time (stream) processing. Spark addresses all these.

Big idea: resilient distributed datasets (RDDs)

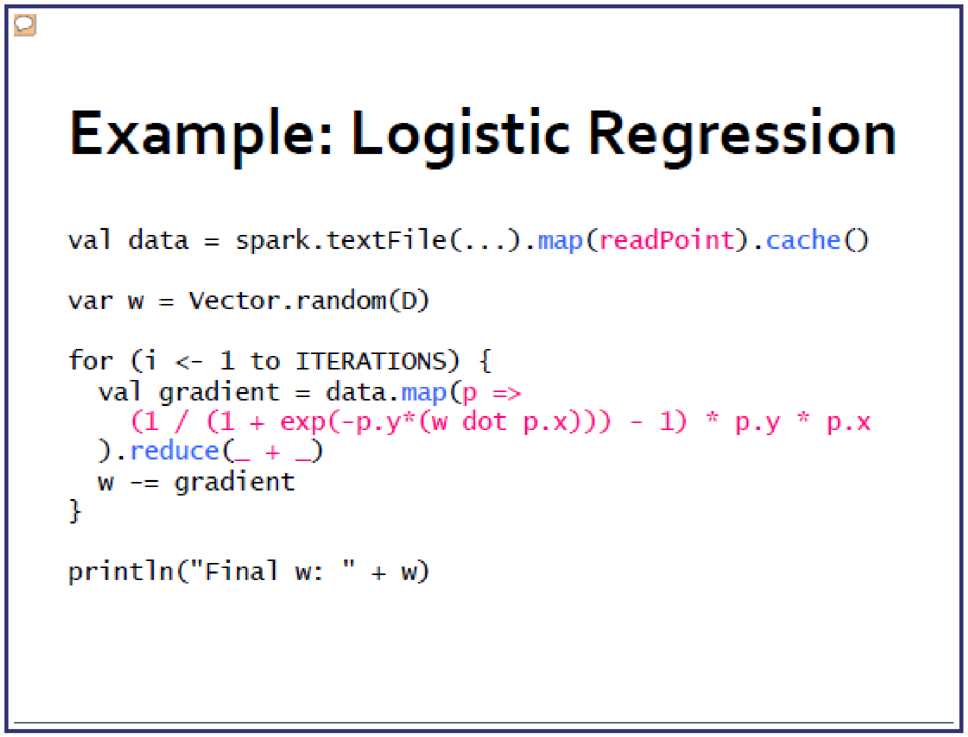
- distributed collections of objects that can be cached in memory across cluster
- manipulated through parallel operators
- automatically recomputed on failure

Impressive performance during iterated computations!



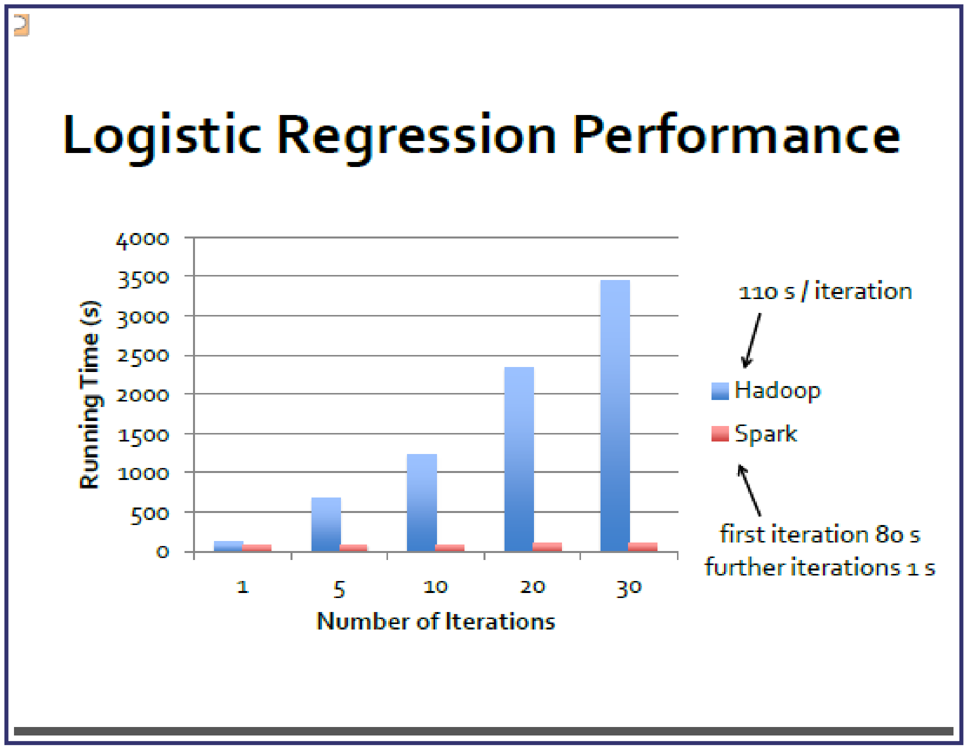
Example: Logistic Regression

Goal: find best line separating two sets of points



Example: Logistic Regression

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```



APIs exist in Python, Java, etc..

```
# Python:
lines = sc.textFile(...) lines.filter(lambda x: "ERROR" in
x).count()

// Java:
JavaRDD lines = sc.textFile(...); lines.filter(new Function() {
```

```
Boolean call(String s) { return s.contains("error"); }
}).count();
```

Spark's modular architecture has been instrumental in enabling the following add-on functionalities:

- Spark Streaming
- Spark SQL
- Spark MLib
- Spark GraphX

Beyond MR: Flink

Similar to MR, Flink is a parallel data processing platform.

"Apache Flink's programming model is based on concepts of the MapReduce programming model but generalizes it in several ways. Flink offers Map and Reduce functions but also additional transformations like Join, CoGroup, Filter, and Iterations. These transformations can be assembled in arbitrary data flows including multiple sources, sinks, and branching and merging flows. Flink's data model is more generic than MapReduce's key-value pair model and allows to use any Java (or Scala) data types. Keys can be defined on these data types in a flexible manner.

Consequently, Flink's programming model is a super set of the MapReduce programming model. It allows to define many programs in a much more convenient and concise way.

I also want to point out that it is possible to embed unmodified Hadoop functions (Input/OutputFormats, Mapper, Reducers) in Flink programs and execute them jointly with native Flink functions."

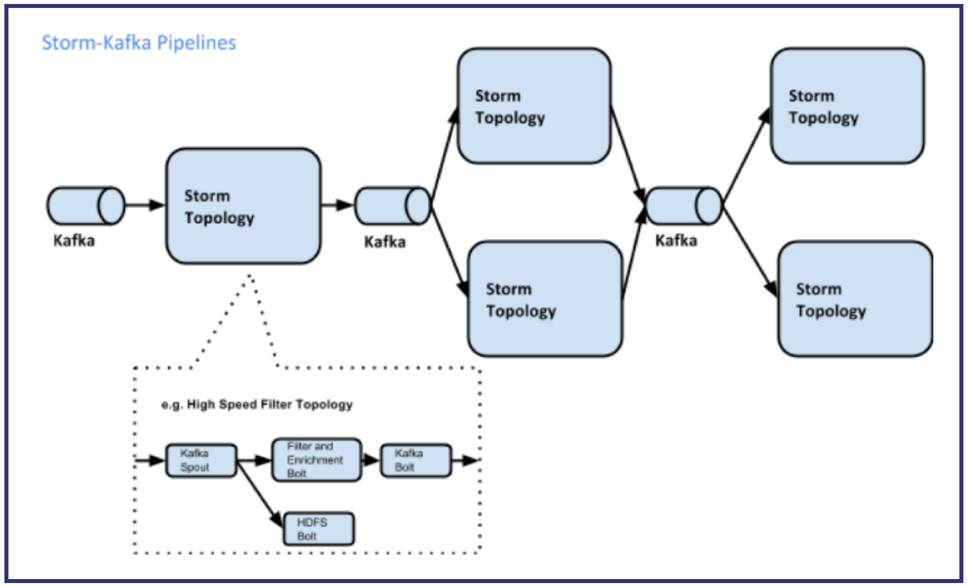
Beyond MR: Storm

"Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language, and is a lot of fun to use!

Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate."

Complementing Storm, Kafka is a distributed pub-sub real-time messaging system that provides strong durability and

fault tolerance guarantees.



BSP: MR alternative

The Bulk Synchronous Parallel (BSP) model is an alternative to MR.

A BSP computation is executed on a set of processors which are connected in a communication network but work independently by themselves. The BSP computation consists of a sequence of iterations, called supersteps. In each superstep, three actions occur:

- (i) concurrent computation performed LOCALLY by a set of processors. Each processor has its own local memory and uses local variables to independently complete its computations. This is the asynchronous part.
- (ii) communication, during which processors send and receive messages (exchange/access data).
- (iii) synchronization which is achieved by setting a barrier - when a processor completes its part of computation and

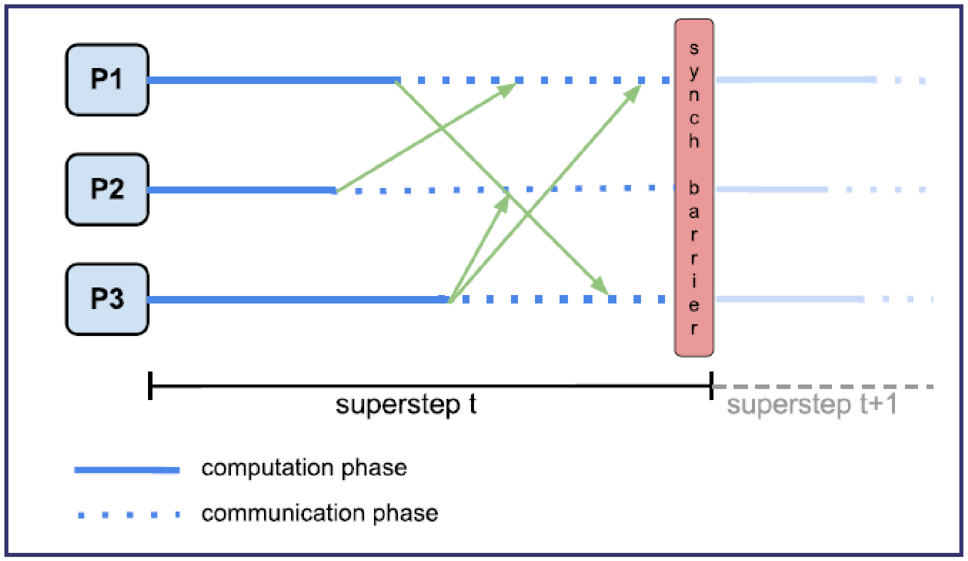
communication, it reaches this barrier and waits for the other processors to finish.

In other words, there are three steps (phases) in each superstep:

Local computation: every processor performs computations using data stored in local memory - independent of what happens at other processors; a processor can contain several processes (threads)

Communication: exchange of data between processes (put and get); one-sided communication

Barrier synchronization: all processes wait until everyone has finished the communication step The following figure illustrates the actions applied in one superstep.


P1
P2
P3
synch barrier
superstep t
superstep t+1
computation phase
communication phase

BSP -> Pregel

"Think like a vertex"..

User-defined vertex update ops (that can happen in parallel), local edge updates.

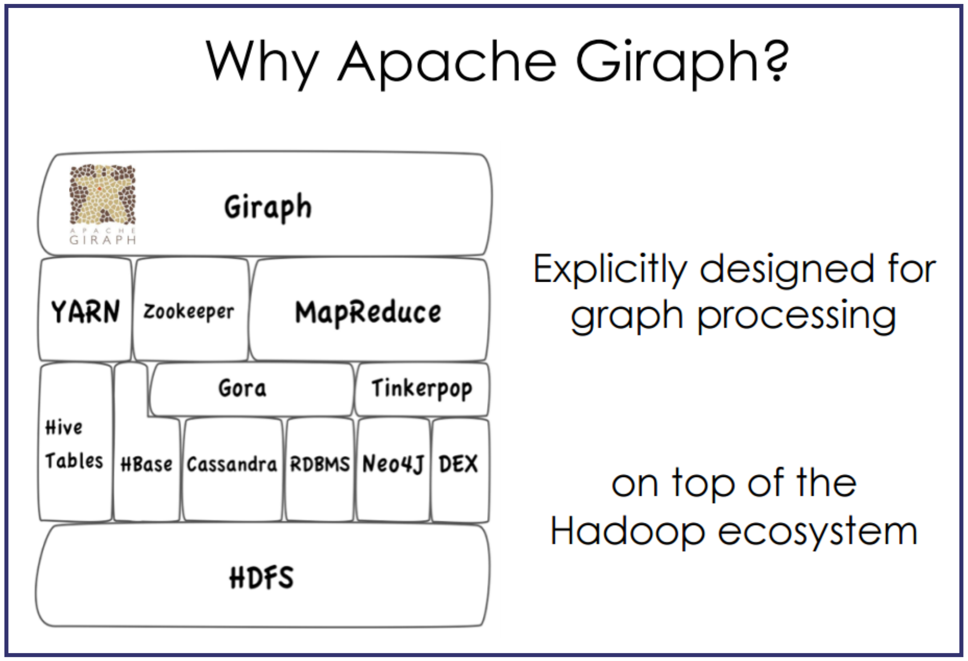
Google's implementation of BSP is called Pregel.

Trivia: why did they name it Pregel? This blog post has the answer :)

Pregel -> Giraph

Giraph is an open source version of Pregel, so is Hama, Golden Orb, Stanford GPS.

Specifically designed for iterative graph computations (and nothing else!).



HAMA: large scale graph processing

Apache HAMA is a general-purpose Bulk Synchronous Parallel (BSP) computing engine on top of Hadoop. It provides a parallel processing framework for massive iterative algorithms (including ones for scientific computing, ie. 'HPC' applications).

HAMA performs a series of supersteps based on BSP - it is suitable for iterative computation, since it is possible that input data which can be saved in memory, is able to get transfered between supersteps (unlike MR).

HAMA's vertex-centric graph computing model is suggestive of MapReduce in that users focus on a local action, processing each item independently, and the system (HAMA runtime) composes these actions to run over a large dataset.

But HAMA is not merely a graph computing engine - instead it is a general purpose BSP platform, so on top of it, any arbitrary computation (graph processing, machine learning, MRQL, matrix algorithms, network algorithms..) can be implemented; in contrast, Giraph is ONLY for graph computing.

Graph MR example: 1T edges!

Here is a paper on the processing of one TRILLION (!) edges at Facebook. Three applications (that run on Facebook's friendships graph) are presented:

- label propagation
- PageRank
- 'friends of friends' score

Please read the paper to learn the details.

Also, here is a writeup on Facebook's use of Giraph (to enable graph searching by users).