MapReduce

Why MapReduce?

- Motivation: Large Scale Data Processing
- Want to process lots of data (> 1 TB)
- Want to parallelize across hundreds/thousands of CPUs
- … Want to make this easy
  - MPI has programming overhead

- MapReduce Idea: simple, highly scalable, generic parallelization model
- Automatic parallelization & distribution
- Fault-tolerant
- Clean abstraction for programmers
- status & monitoring tools

Who Uses MapReduce?

- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - “Web map” powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization

Overview

- **MapReduce**: the concept
- **Hadoop**: the implementation
- Query Languages for Hadoop
- **Spark**: the improvement
- MapReduce vs databases
- Conclusion

MapReduce: the concept

Credits:

- David Maier
- Google
- Shiva Teja Reddi Gopidi

Preamble: Merits of Functional Programming (FP)

- FP: input determines output – *and nothing else*
  - No other knowledge used (global variables!)
  - No other data modified (global variables!)
  - Every function invocation generates new data
- Opposite: procedural programming → side effects
  - Unforeseeable interference between parallel processes
    → difficult/impossible to ensure dterministic result
- (function, value set) must be monoid
- Advantage of FP: parallelization can be arranged automatically
  - can (automatically!) reorder or parallelize execution - data flow implicit

being side-effect free is a key poperty of SQL

Programming Model

- Goals: large data sets, processing distributed over 1,000s of nodes
  - Abstraction to express simple computations
  - Hide details of parallelization, data distribution, fault tolerance, load balancing - MapReduce engine performs all housekeeping
- Inspired by primitives from functional PLs like Lisp, Scheme, Haskell
- Input, output are sets of key/value pairs
- Users implement interface of two functions:

```
map (inKey, inValue) -> (outKey, intermediateValuelist )
```

aka „group by“ in SQL

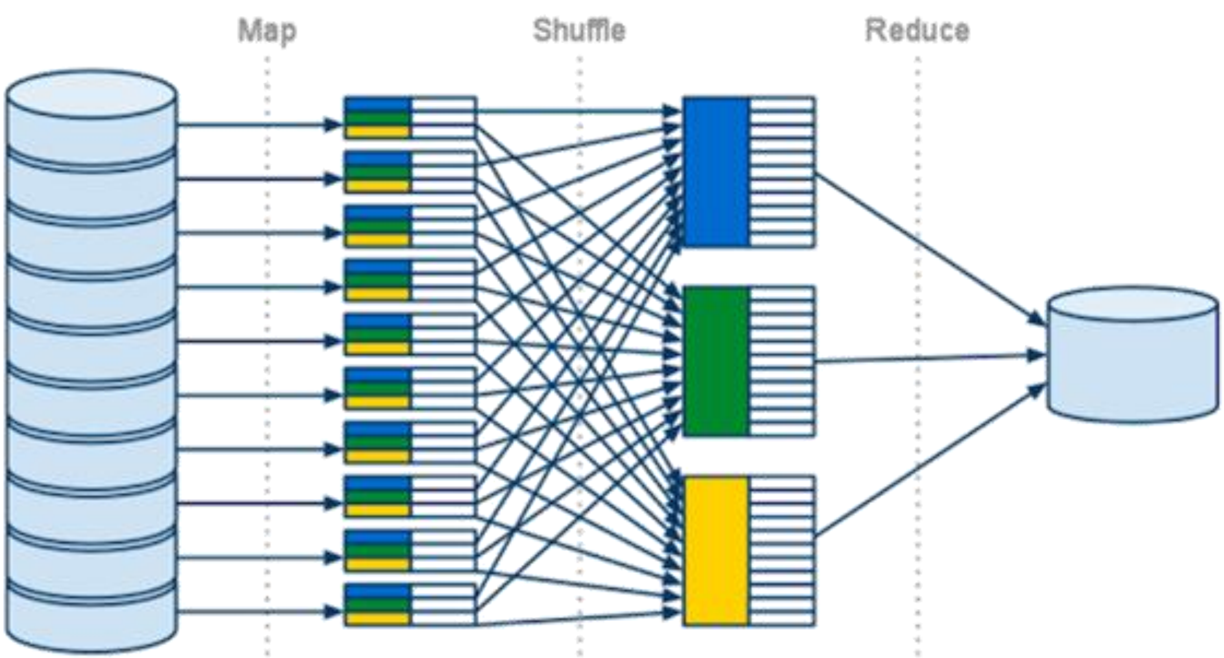
```
reduce(outKey, intermediateValuelist) -> outValuelist
```

aka aggregation in SQL

Ex 1: Count Word Occurrences

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

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

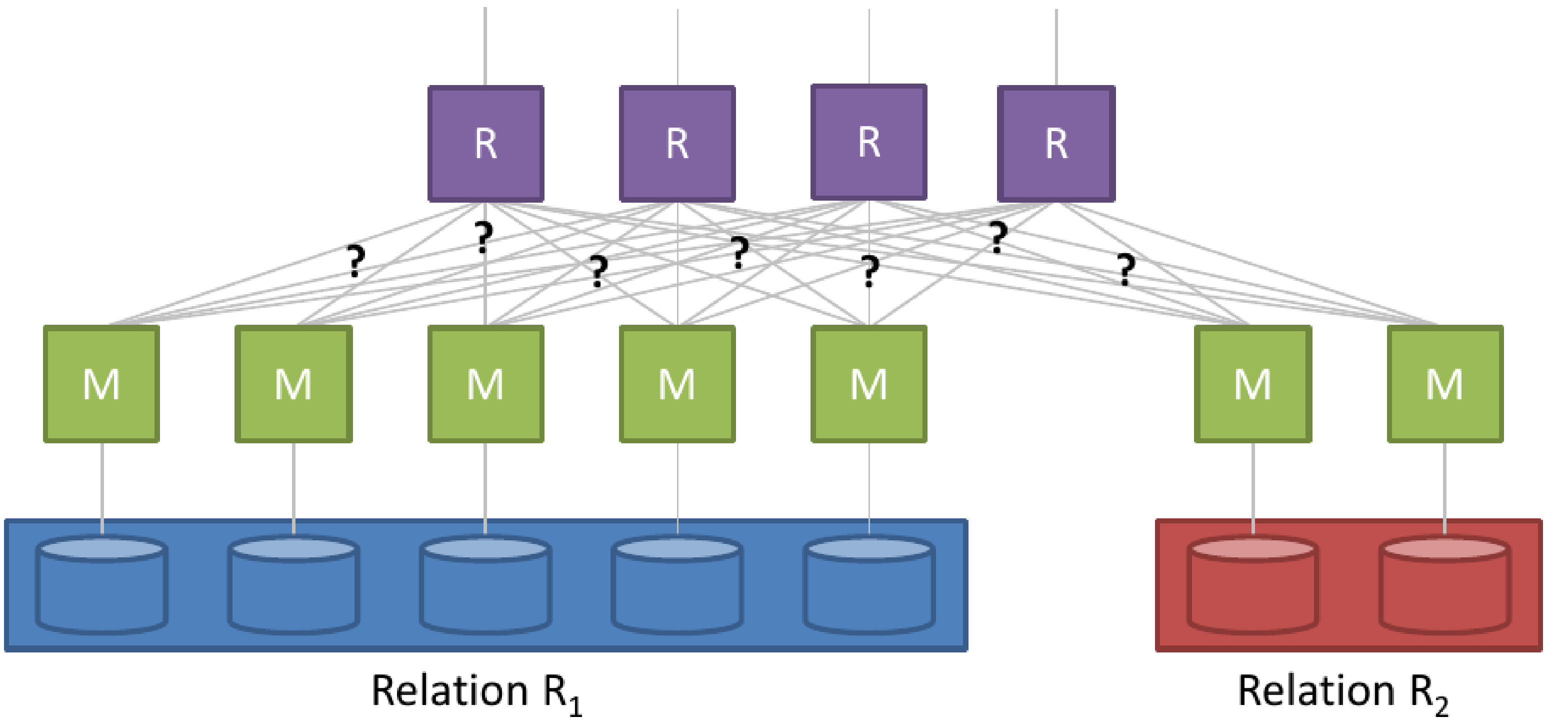


[image: Google]

Ex 2: Distributed Grep

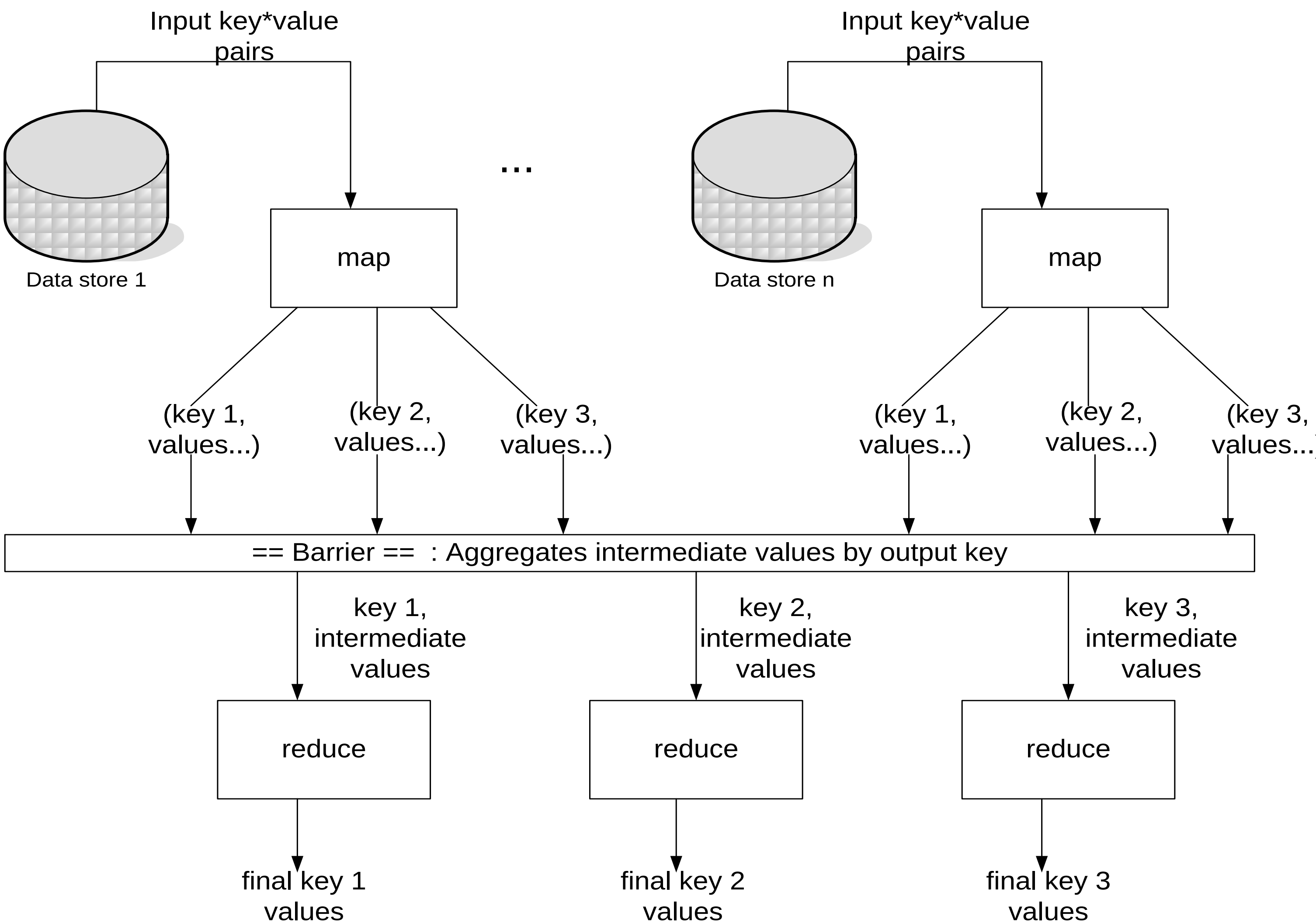
- map function emits line if matches given pattern
  - identity function that just copies supplied intermediate data to output
- Application 1: Count of URL Access Frequency
  - logs of web page requests → **map()** → <URL,1>
  - all values for same URL → **reduce()** → <URL, total count>
- Application 2: Inverted Index
  - Document → **map()** → sequence of <word, document ID> pairs
  - all pairs for a given word → **reduce()** sorts document IDs → <word, list(document ID)>
  - set of all output pairs = simple inverted index
  - easy to extend for word positions

Ex 3: Relational Join



- Map function M: "hash on key attribute": (?, tuple) → list(key, tuple)
- Reduce function R: "join on each k value": (key, list(tuple)) → list(tuple)

Map & Reduce



Input key*value pairs
Input key*value pairs
Data store 1
map
...
Data store n
map
(key 1, values...)
(key 2, values...)
(key 3, values...)
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
key 1, intermediate values
key 2, intermediate values
key 3, intermediate values
reduce
reduce
reduce
final key 1 values
final key 2 values
final key 3 values

Map Reduce Patent

- Google granted US Patent 7,650,331, January 2010
- **System and method for efficient large-scale data processing**
  A large-scale data processing system and method includes one or more application-independent map modules configured to read input data and to apply at least one application-specific map operation to the input data to produce intermediate data values, wherein the map operation is automatically parallelized across multiple processors in the parallel processing environment. A plurality of intermediate data structures are used to store the intermediate data values. One or more application-independent reduce modules are configured to retrieve the intermediate data values and to apply at least one application-specific reduce operation to the intermediate data values to provide output data.

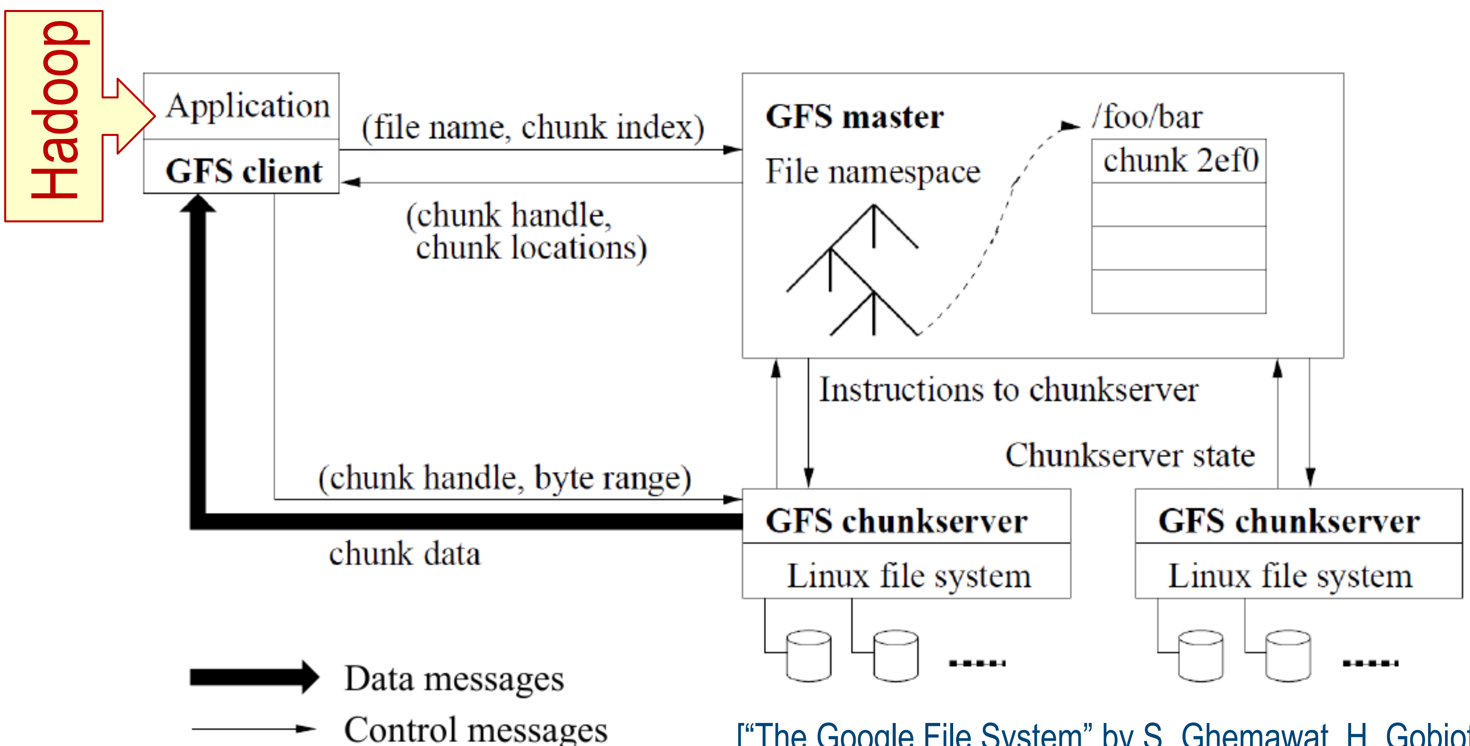
Hadoop: a MapReduce implementation

Credits:

- David Maier, U Wash
- Costin Raiciu
- "The Google File System" by S. Ghemawat, H. Gobioff, and S.-T. Leung, 2003
- https://hadoop.apache.org/docs/r1.0.4/hdfs_design.html

Hadoop Distributed File System

- HDFS = scalable, fault-tolerant file system
  - modeled after Google File System (GFS)
  - 64 MB blocks („chunks“)



[“The Google File System” by S. Ghemawat, H. Gobioff, and S.-T. Leung, 2003]

GFS

- Goals:
  - Many inexpensive commodity components – failures happen routinely
  - Optimized for small # of large files (ex: a few million of 100+ MB files)
- relies on local storage on each node
  - parallel file systems: typically dedicated I/O servers (ex: IBM GPFS)
- metadata (file-chunk mapping, replica locations, ...) in master node‘s RAM
  - Operation log on master‘s local disk, replicated to remotes → master crash recovery!
  - „Shadow masters“ for read-only access

HDFS differences?

- No random write; append only
- Implemented in Java, emphasizes platform independence
- terminology: namenode → master, block → chunk, ...

GFS Consistency

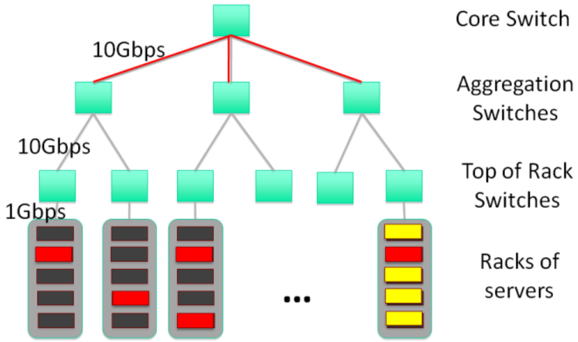
- Relaxed consistency model
  - tailored to Google’s highly distributed applications, simple & efficient to implement
- File namespace mutations are atomic
  - handled exclusively by master; locking guarantees atomicity & correctness
  - master’s log defines global total order of operations
- State of file region after data mutation
  - consistent: all clients always see same data, regardless of replica they read from
  - defined: consistent, plus all clients see the entire data mutation
  - undefined but consistent: result of concurrent successful mutations; all clients see same data, but may not reflect any one mutation
  - inconsistent: result of a failed mutation

GFS Consistency: Consequences

- Implications for applications
  - *better not distribute records across chunks!*
  - rely on appends rather than overwrites
  - application-level checksums, checkpointing, writing self-validating & self-identifying records
- Typical use cases (or "hacking around relaxed consistency")
  - writer generates file from beginning to end and then atomically renames it to a permanent name under which it is accessed
  - writer inserts periodical checkpoints, readers only read up to checkpoint
  - many writers concurrently append to file to merge results, reader skip occasional padding and repetition using checksums

Replica Placement

- Goals of placement policy
  - scalability, reliability and availability, maximize network bandwidth utilization
- Background: GFS clusters are highly distributed
  - 100s of chunkservers across many racks
  - accessed from 100s of clients from same or different racks
  - traffic between machines on different racks may cross many switches
  - bandwidth between racks typically lower than within rack



Replica Placement

- Goals of placement policy
  - scalability, reliability and availability, maximize network bandwidth utilization
- Background: GFS clusters are highly distributed
  - 100s of chunkservers across many racks
  - accessed from 100s of clients from same or different racks
  - traffic between machines on different racks may cross many switches
  - bandwidth between racks typically lower than within rack
- Selecting a chunkserver
  - place chunks on servers with below-average disk space utilization
  - place chunks on servers with low number of recent writes
  - spread chunks across racks (see above)

Hadoop Job Management Framework

- JobTracker = daemon service for submitting & tracking MapReduce jobs
- TaskTracker = slave node daemon in the cluster accepting tasks (Map, Reduce, & Shuffle operations) from a JobTracker

Discussion:

- Pro: replication & automated restart of failed tasks
  → highly reliable & available
- Con: 1 Job Tracker per Hadoop cluster, 1 Task Tracker per slave node
  → single point of failure

Optimizations / 1

- Problem:
  No reduce can start until map is complete
  → single slow disk controller can rate-limit whole process

- Solution:
  Master redundantly executes slow (“straggler”) map tasks;
  uses results of first copy to finish

- *Why is it safe to redundantly execute map tasks?*
  Wouldn’t this mess up the total computation?

Optimizations / 2

- Problem:
  excessive data transport between map() and reduce() workers
- Approach:
  "Combiner" functions can run on same machine as a mapper
  - "mini-reduce phase" followed by "final" reduce phase
  - saves bandwidth
- *Under what conditions is it sound to use a combiner?*

Discussion

- MapReduce concept:
  - One-input two-stage data flow extremely rigid
  - Most suitable for independent data
    - *Good: word count*
    - *Not optimal: join, graphs, arrays, ...*
  - HDFS assumes shared-nothing & locality, but datacenters often run SANs
  - (Well-known) algorithms need cumbersome rewriting = special-skill programming
    - *Query frontends: Pig Latin, Hive, etc.*
  - map(), reduce() procedural Java code
    → hard to optimize
- Hadoop implementation:
  - All intermediate data communicated via disk
  - Task scheduler: central point of failure
  - HDFS not standards conformant (eg, POSIX)

Query Languages for MapReduce

Credits:

- Matei Zaharia

Motivation

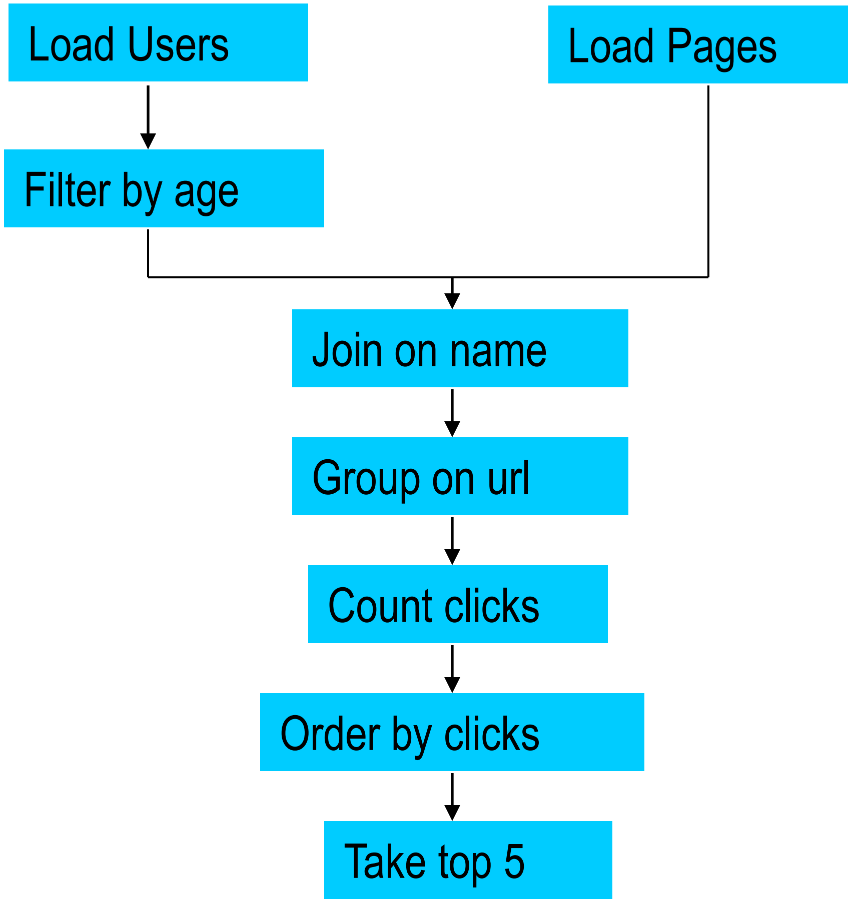
- MapReduce is powerful
  - many algorithms can be expressed as a series of MR jobs
- But fairly low-level
  - must think about keys, values, partitioning, etc.
- Can we capture common “job patterns”?
  - Like eg SQL does

Pig

- Started at Yahoo! Research
  - Runs about 50% of Yahoo!'s jobs
- Features:
  - Expresses sequences of MapReduce jobs
  - Data model: nested “bags” of items
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc)
  - Easy to plug in Java functions

Example Problem

- user data in one file
- website data in another
- find top 5 most visited pages
- by users aged 18-25



[http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt]

In MapReduce

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobC ontrol;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.sub string(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1 " + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring( firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so w e know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.to String();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
```

```java
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputColle ctor<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', first Comma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
```

```java
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.add InputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf( MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMap per.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.se tOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRE xample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFi leOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputF ormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
```

[http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt]

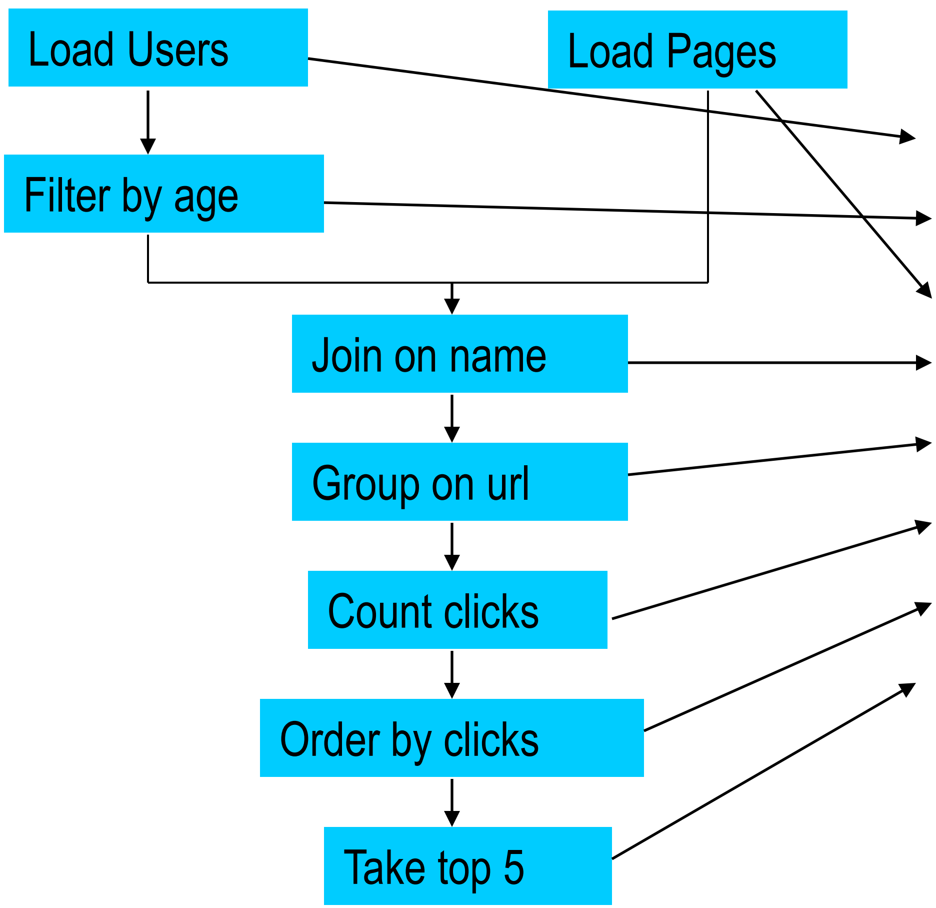
In Pig Latin

```
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';
```

Translation to MapReduce

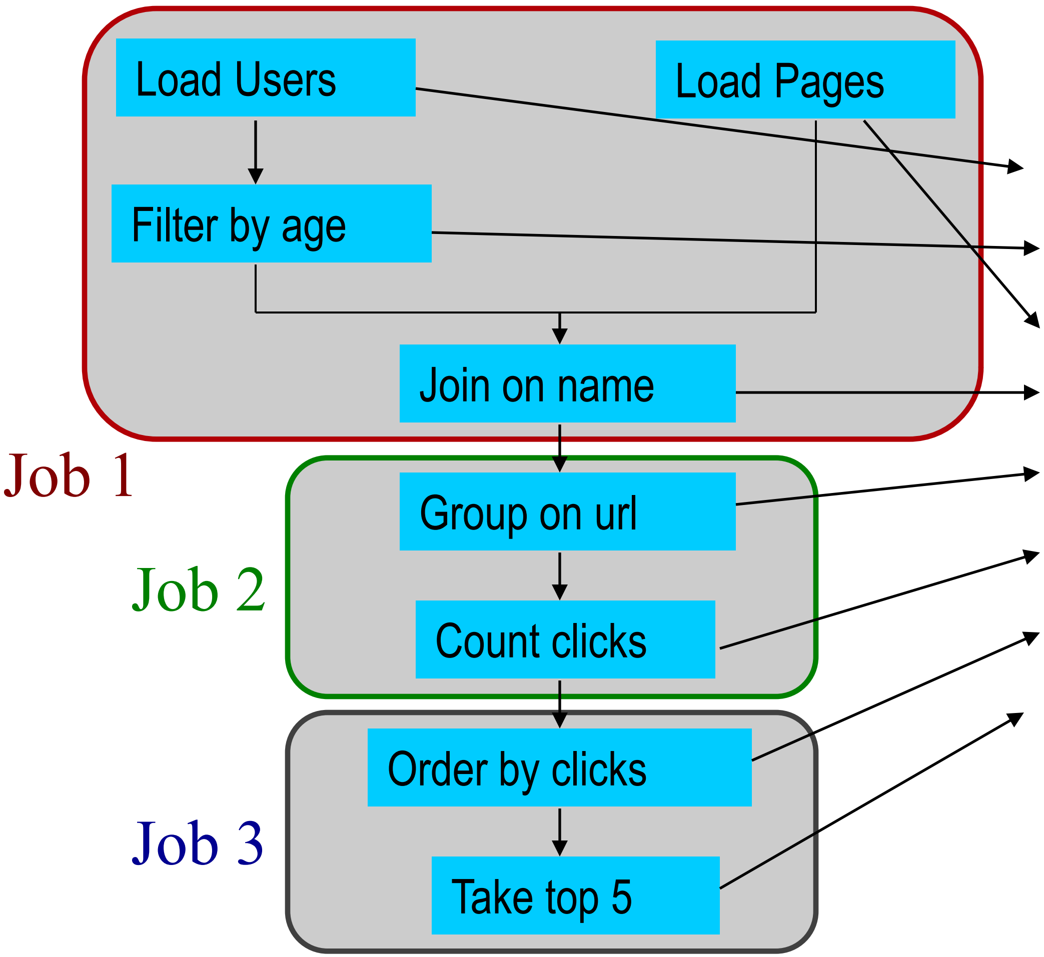
Quite natural translation of job components into Pig Latin:




```
Users = load …
Filtered = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … count()…
Sorted = order …
Top5 = limit …
```

Translation to MapReduce

Quite natural translation of job components into Pig Latin:




```
Users = load …
Filtered = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … count()…
Sorted = order …
Top5 = limit …
```

Hive

- Relational database built on Hadoop
  - table schemas, SQL-like query language

```
SELECT word, count(1) AS count
FROM  (SELECT explode(split(line, '\s')) AS word
       FROM docs) temp
GROUP BY word
ORDER BY word
```

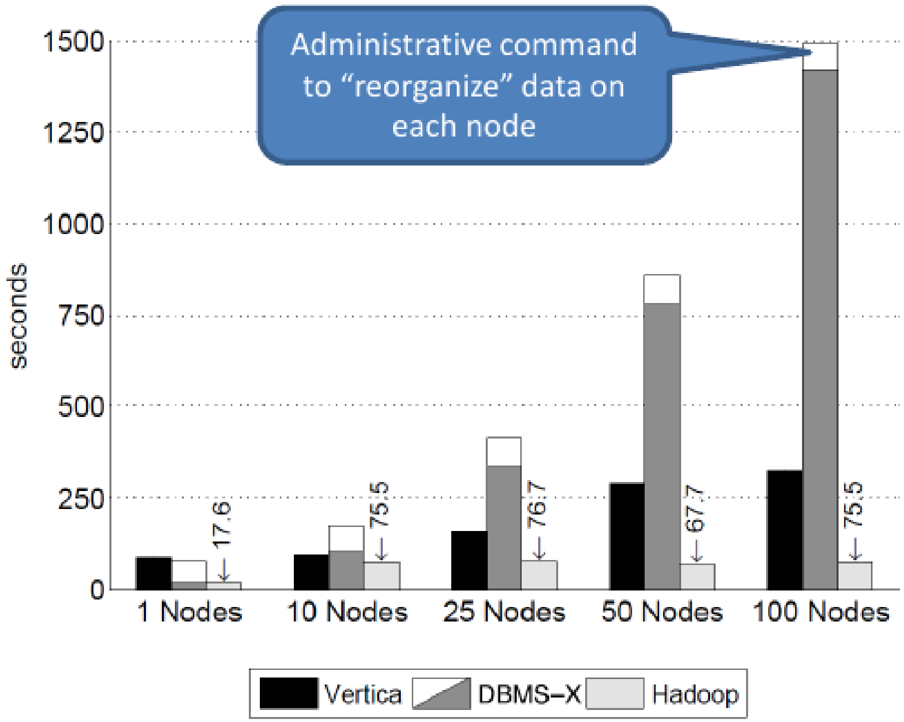
  - can call Hadoop Streaming scripts
- Common relational features:
  - table partitioning,complex data types, sampling
  - some query optimization
- Developed at Facebook, now Apache
  - Today: „data warehouse infrastructure“

MapReduce vs (Relational) Databases

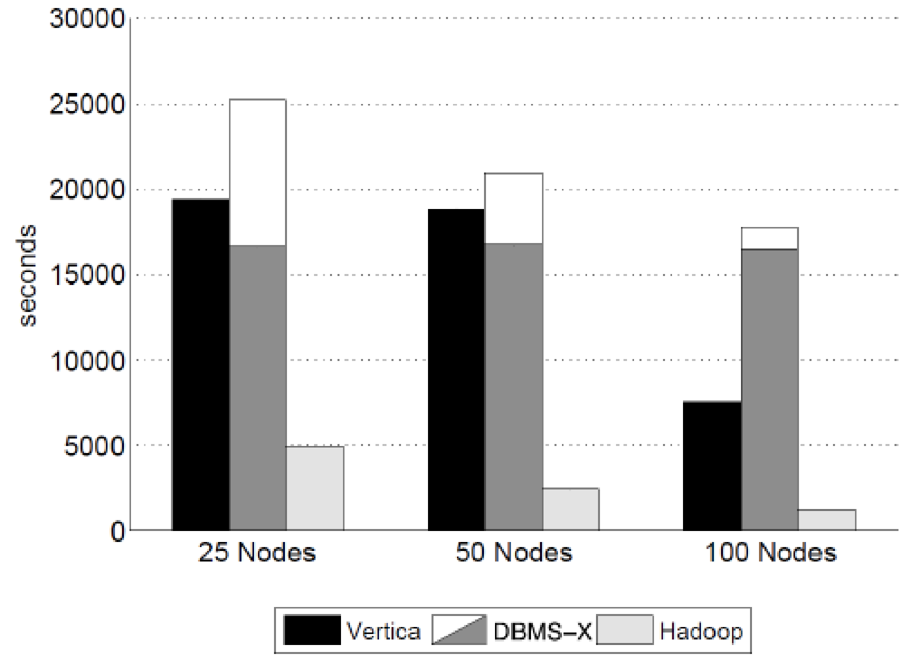
Credits: David Maier

Grep Task: Load Times

535 MB/node



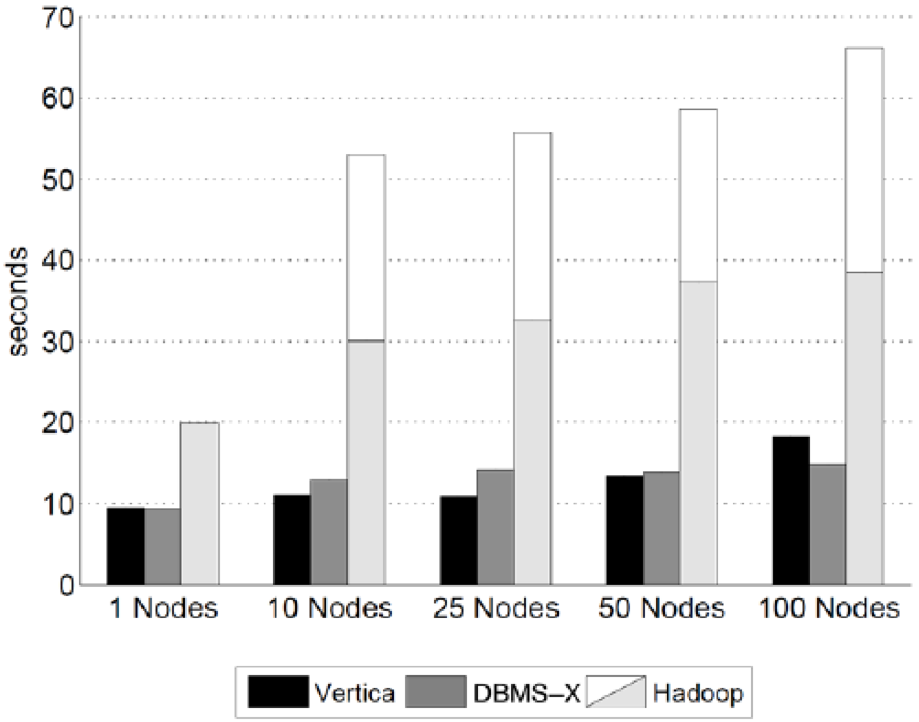
1 TB/cluster



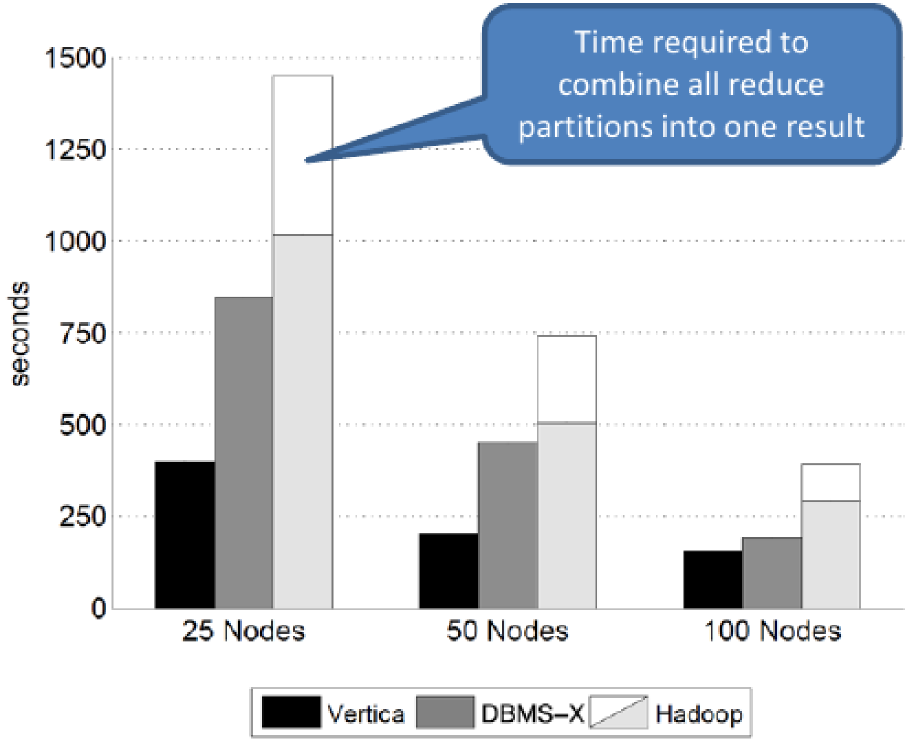
["A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004]

Grep Task: Execution Times

535 MB/node



1 TB/cluster



[“A Comparison of Approaches to Large-Scale Data Analysis” by A. Pavlo et al., 2004]

MapReduce Criticism

- Efficiency
  - master makes O(M + R) scheduling decisions
  - master stores O(M * R) states in memory
- "Why not use a parallel DBMS instead?"
  - map/reduce is a "giant step backwards"
  - no schema, no indexes, no high-level language
  - not novel at all
  - does not provide features of traditional DBMS
  - incompatible with DBMS tools

Analytics Tasks

```
CREATE TABLE Documents (
   url VARCHAR(100)
   PRIMARY KEY,
   contents TEXT );
```

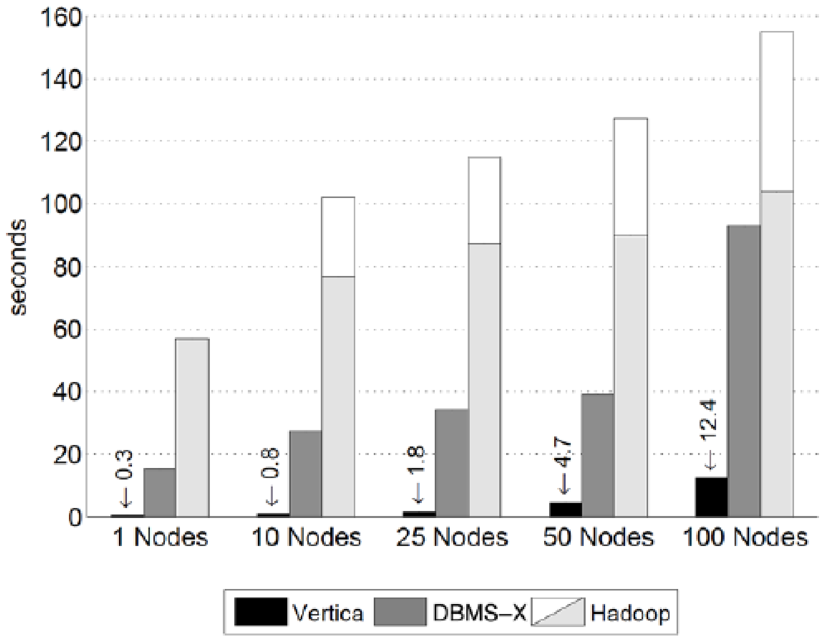
```
CREATE TABLE Rankings (
   pageURL VARCHAR(100)
   PRIMARY KEY,
   pageRank INT,
   avgDuration INT );
```

```
CREATE TABLE UserVisits (
   sourceIP VARCHAR(16),
   destURL VARCHAR(100),
   visitDate DATE,
   adRevenue FLOAT,
   userAgent VARCHAR(64),
   countryCode VARCHAR(3),
   languageCode
   VARCHAR(3),
   searchWord VARCHAR(32),
   duration INT );
```

- Data set
  - 600K unique HTML documents
  - 155M user visit records (20 GB/node)
  - 18M ranking records (1 GB/node)

["A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004]

Select Task



- SQL Query:

```
SELECT pageURL, pageRank
FROM Rankings
WHERE pageRank > X
```

- Relational DBMS
  - use index on pageRank column
  - Relative performance degrades as number of nodes increases
- Hadoop start-up cost increase with cluster size

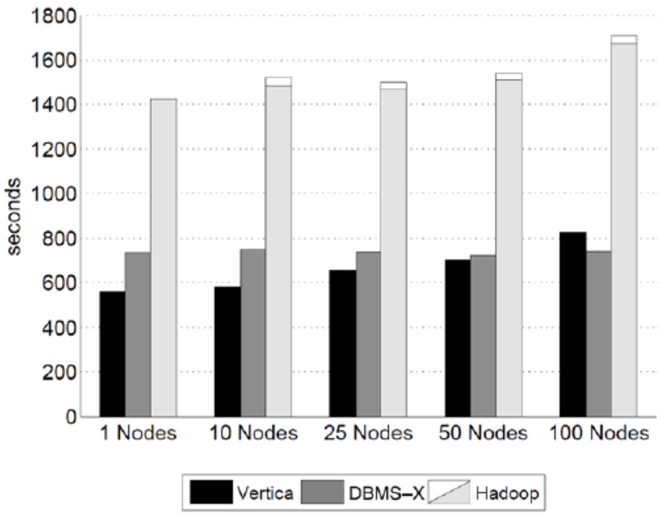
[“A Comparison of Approaches to Large-Scale Data Analysis” by A. Pavlo et al., 2004]

Aggregation Task

"total ad revenue for each source IP, based on user visits table"

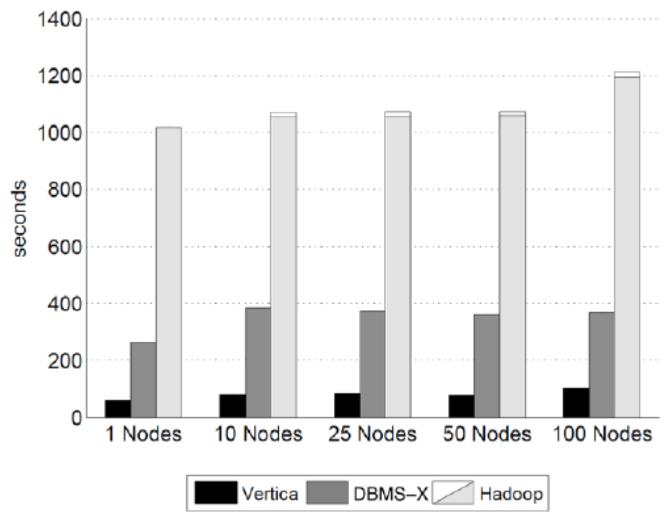
Variant 1: 2.5M groups

```
SELECT sourceIP, SUM(adRevenue)
FROM UserVisits
GROUP BY sourceIP
```



Variant 2: 2,000 groups

```
SELECT SUBSTR(sourceIP, 1, 7),
SUM(adRevenue)
FROM UserVisits
GROUP BY SUBSTR(sourceIP, 1, 7)
```



["A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004]

Join Task

SQL Query:

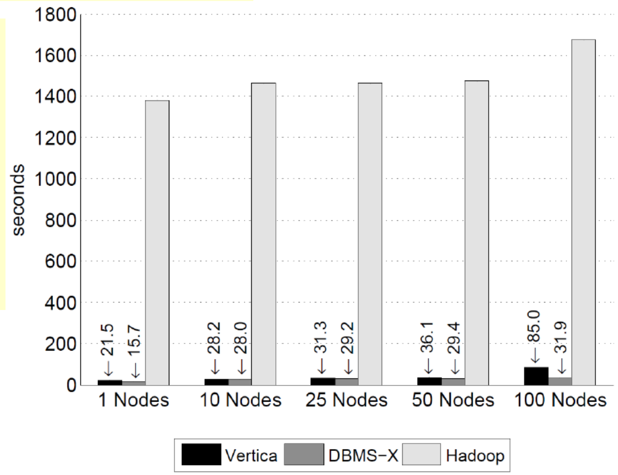
```
SELECT INTO Temp
  UV.sourceIP,
  AVG(R.pageRank) AS avgPageRank,
  SUM(UV.adRevenue) AS totalRevenue
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL
  AND UV.visitDate BETWEEN
      DATE('2000-01-15') AND
      DATE('2000-01-22')
GROUP BY UV.sourceIP
```

```
SELECT sourceIP,
avgPageRank,
totalRevenue
FROM Temp
ORDER BY totalRevenue
         DESC LIMIT 1
```

MapReduce program:

- filter records outside date range, join with rankings file
- compute total ad revenue and average page rank based on source IP
- produce largest total ad revenue record

- Phases run in strict sequential order



["A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004]

Summary: MapReduce vs Parallel (R)DBMS

- MapReduce: No schema, no index, no high-level language
  - faster loading vs. faster execution
  - easier prototyping vs. easier maintenance
- Fault tolerance
  - restart of single worker vs. restart of transaction
- Installation and tool support
  - easy to setup map/reduce vs. challenging to configure parallel DBMS
  - no tools for tuning vs. tools for automatic performance tuning
- Performance per node
  - results seem to indicate that parallel DBMS achieve same performance as map/reduce in smaller clusters

In a nutshell:
- (R)DBMSs: efficiency, QoS
- MapReduce: cluster scalability

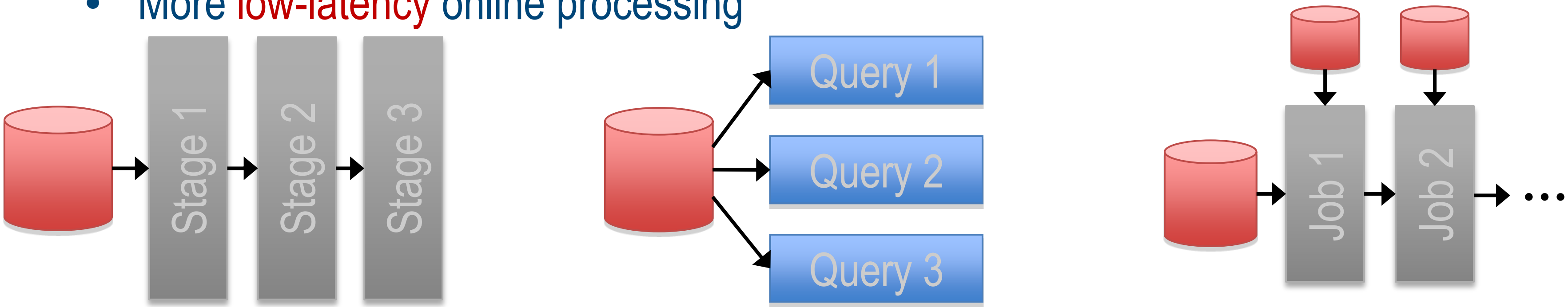
Spark: improving Hadoop

Credits:

- Matei Zaharia

Motivation

- MapReduce aiming at “big data” analysis on large, unreliable clusters
  - After initial hype, shortcomings perceived:
    ease of use (programming!), efficiency, tool integration, ...
- …as soon as organizations started using it widely, users wanted more:
  - More complex, multi-stage applications
  - More interactive queries
  - More low-latency online processing



Iterative job | Interactive mining | Stream processing

Spark vs Hadoop

- Spark = cluster-computing framework by Berkeley AMPLab
  - Now Apache
- Inherits HDFS, MapReduce from Hadoop
- But:
  - Disk-based comm → in-memory comm
  - Java → Scala

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 series of transformations that built RDD (its *lineage*)
  - Can recompute lost data based on input files

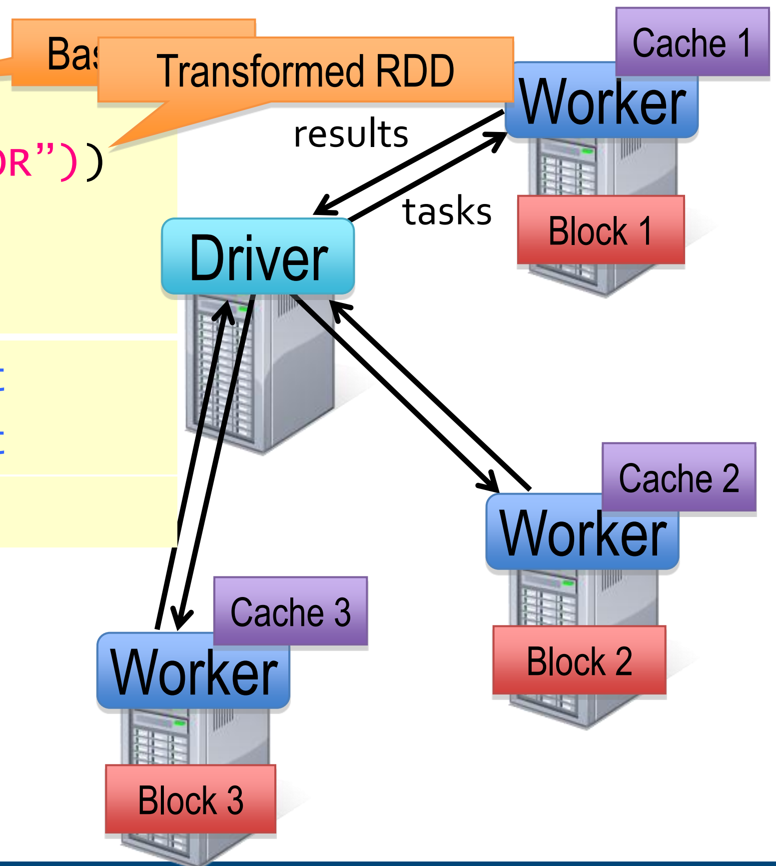
Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
messages.cache()
```

```
messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
```

```
. . .
```

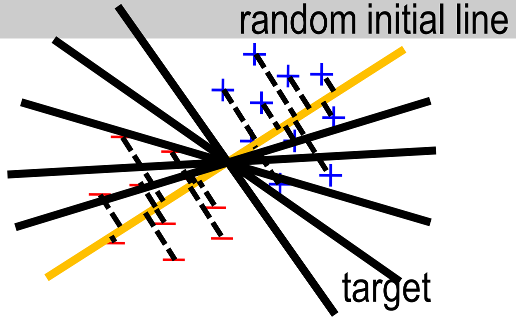


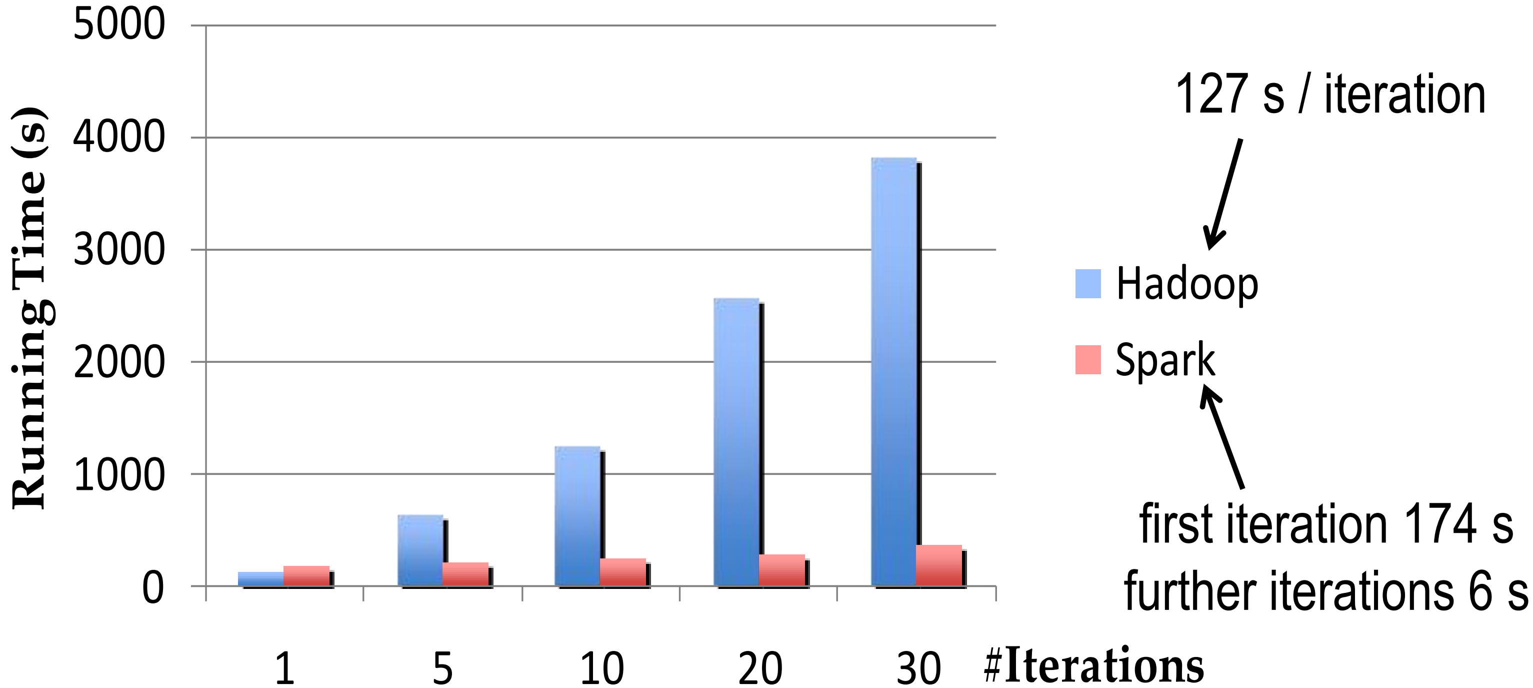
Scala programming language

1 TB data in 5-7 sec (vs 170 sec on disk)

Ex: Logistic Regression Performance

- Find best line separating two sets of points
- 29 GB dataset
- 20x EC2 m1.xlarge 4-core machines
- Result:





Conclusion

Conclusion

- MapReduce = specialized (synchronous) distributed processing paradigm
  - Optimized for horizontal scaling in commodity clusters (!), fault tolerance
  - Efficiency? Hardware, energy, ... (see [0], [1], [2], [3] etc.)
    - *"Adding more compute servers did not yield significant improvement"* [src]
  - Well suited for sets, less so for highly connected data (graphs, arrays)
  - Need to rewrite algorithms
- Apache Hadoop = MapReduce implementation (HDFS, Java)
- Apache Spark = improved MapReduce implementation (HDFS, DSS, Scala)
- Query languages on top of MapReduce
  - HLQLs: Pig, Hive, JAQL, ASSET, …