CONSTRUCTOR UNIVERSITY

MapReduce

Instructor: Peter Baumann

email: pbaumann@constructor.university

tel: -3178

office: room 88, Research 1

MapReduce

- Goals: large data sets, distributed processing
  - Hide details of parallelization, data distribution, fault tolerance, load balancing
  - Inspired by functional PLs: Lisp, Scheme, Haskell, ...
  - Functional programming: no side effects → automatic parallelization
- MapReduce programming model:
  - sets of key/value pairs
  - Developer implements interface of two (side-effect free) functions:

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

← aka „group by“ in SQL

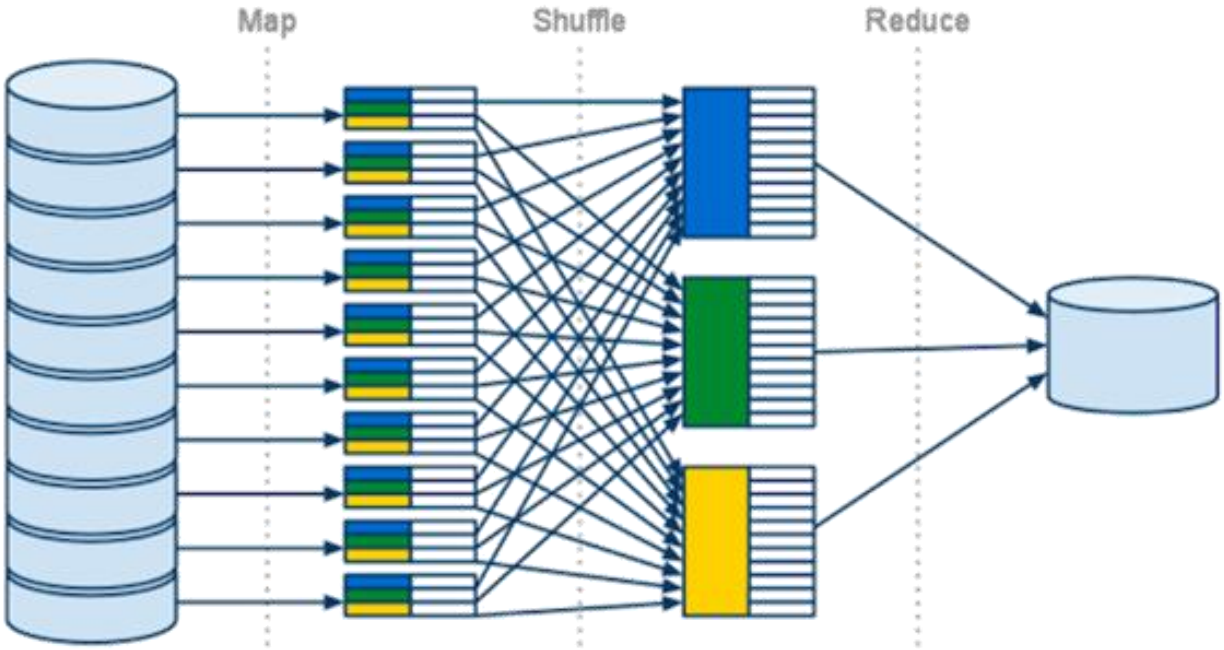
```
reduce(outKey, intermediateValuelist) -> outValuelist
```

← 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]

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 Key Components

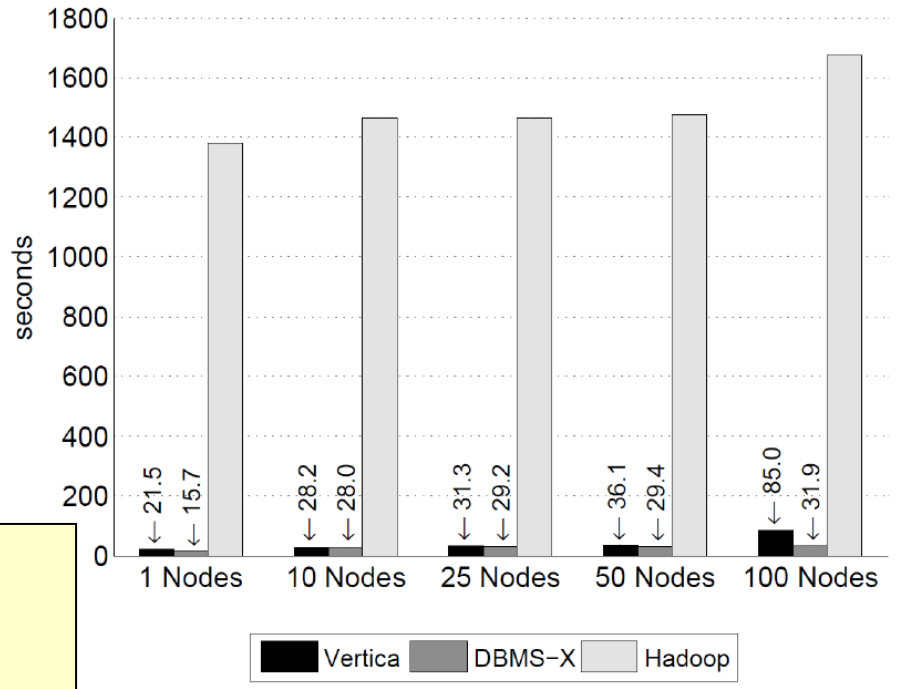
- Hadoop Job Management Framework
  - JobTracker = daemon service for submitting & tracking MapReduce jobs
  - TaskTracker = slave node daemon in the cluster accepting tasks (Map, Reduce, & Shuffle operations) from a JobTracker
- Hadoop File System (HDFS) = scalable, fault-tolerant file system
  - modeled after Google File System (GFS)
  - programs request data as 64 MB blocks („chunks“) from server, Hadoop ships
- *Data processing, not management*

Query Languages for MapReduce

- MapReduce powerful, but slow and fairly low-level
  - algorithms need cumbersome rewriting = special-skill programming
  - common “job patterns”, like SQL join?

```
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
```



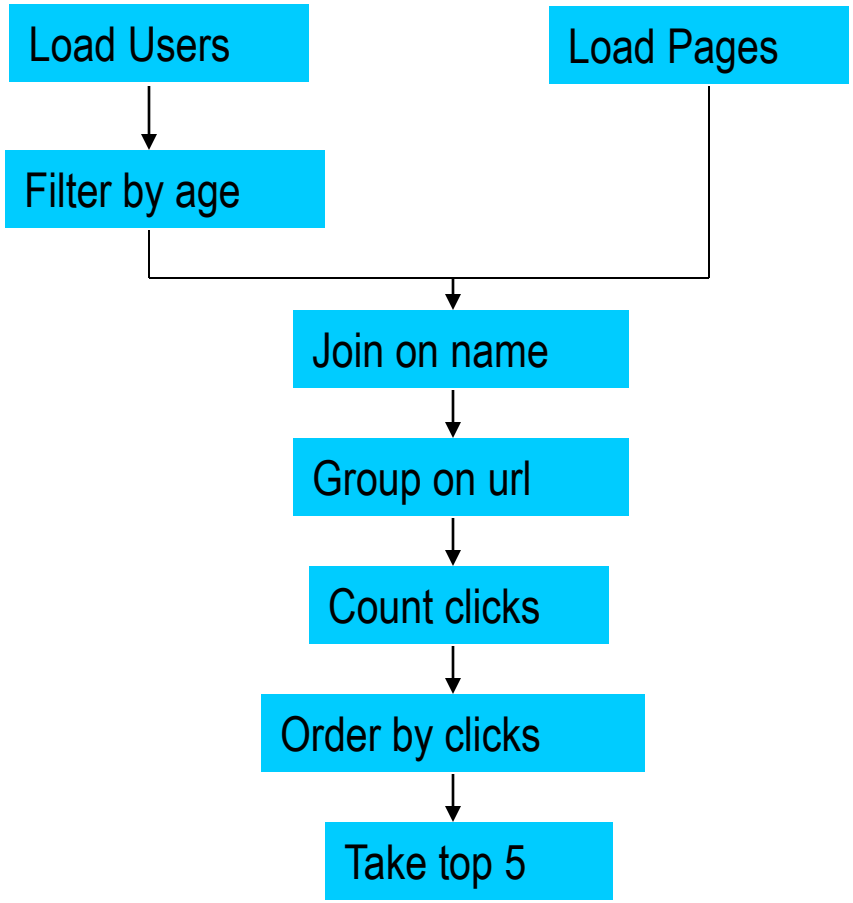
[A. Pavlo et al.: A Comparison of Approaches to Large-Scale Data Analysis]

Pig

- Pig = declarative query language
  - Yahoo! Research
- Features:
  - sequences of MapReduce jobs
  - 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]

MapReduce vs. Pig Latin

```
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;
```

```
        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> {
```

```
        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);
```

```
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';
```

```
        // Prepend an index to the value so we 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.toString();
            if (value.charAt(0) == '1')
first.add(value.substring(1));
            else second.add(value.substring(1));
```

```
            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");
    lp.setInputFormat(TextInputFormat.class);
```

```
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.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]

Hive

- Relational database built on Hadoop
  - Facebook, now Apache
- Common relational features:
  - table partitioning, complex data types, sampling
  - some query optimization
- Ex:

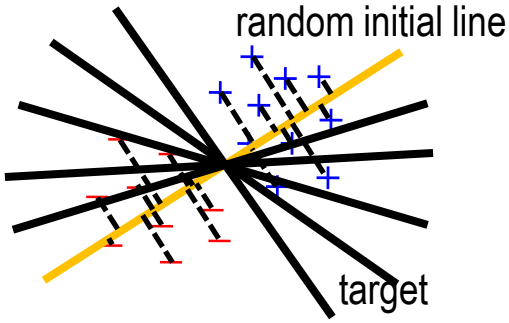
```
SELECT word, count(1) AS count
FROM  (SELECT explode(split(line, '\s')) AS word
       FROM docs) temp
GROUP BY word
ORDER BY word
```

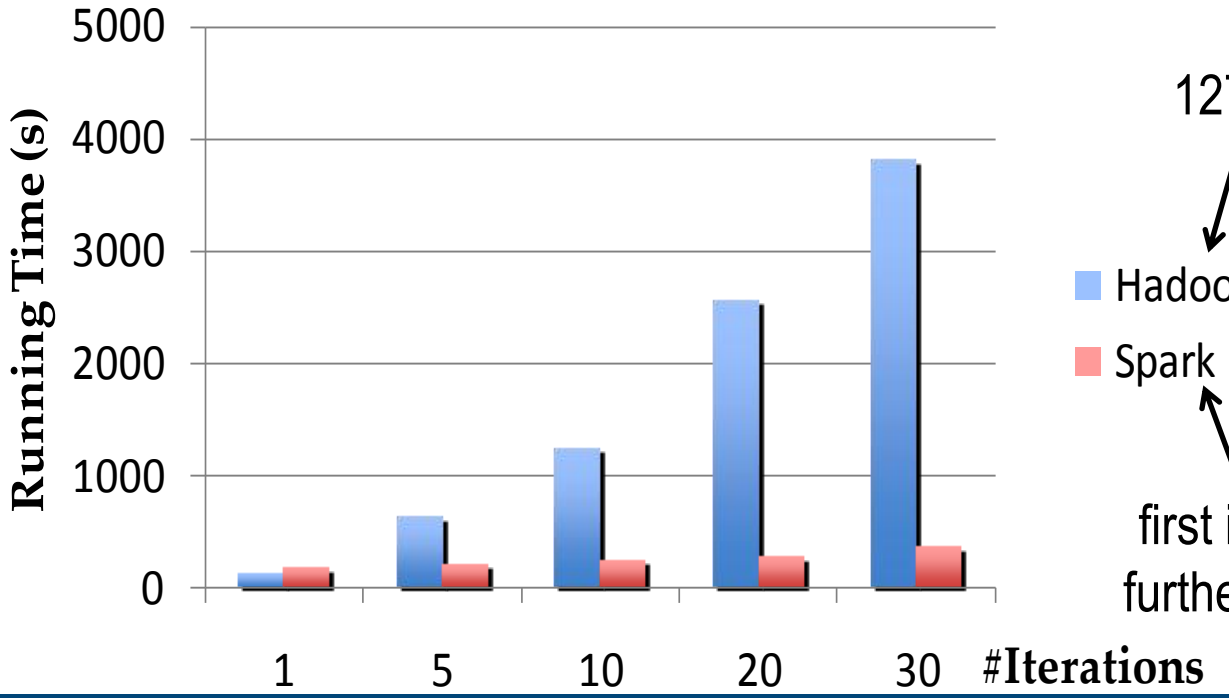
Spark: improving Hadoop

- After initial Hadoop hype, shortcomings perceived
  - Difficulty of use, efficiency, tool integration, ...
- Spark = cluster-computing framework by Berkeley AMPLab
  - Now Apache
- MapReduce, but:
  - Disk-based comm → in-memory comm
  - Java → Scala
  - Resilient Distributed Datasets (RDDs)
    - *Objects split across cluster*
    - *Remember sequence of transformations → can recompute on failure*
- *Data processing, not management*

Ex: Logistic Regression Performance

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





127 s / iteration

Hadoop

Spark

first iteration 174 s
further iterations 6 s

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, …