HADOOP
AND OTHERS

INTRODUCTION TO DATA SCIENCE
TIM KRASKA
LAST LECTURE

• Cloud Computing
• HDFS
• MapReduce

THIS LECTURE

• MapReduce ctd
• Other large scale processing frameworks
• Small scale processing frameworks
• (NO SQL)
Input to the _______ is the sorted output of the mappers.

a) Reducer  
b) Mapper  
c) Shuffle  
d) All of the above
Data type: key-value records

Map function:

\[(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})\]

Reduce function:

\[(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})\]
def mapper(line):
    foreach word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
the quick brown fox
the fox ate the mouse
how now brown cow

the, 1
brown, 1
fox, 1

the, 2
fox, 1

how, 1
now, 1
brown, 1

ate, 1
mouse, 1

cow, 1

brown, 2
fox, 2
how, 1
now, 1
the, 3

ate, 1
mouse, 1
quick, 1

WORD COUNT WITH COMBINER
OTHER MAP/REDUCE PARAMETERS

• One or more Map tasks
• Zero or more Reduce tasks
• Zero or more Combiner tasks
• Shuffle / Partitioning function (distributed)
• Sort function (locally executed)
• Context for Map,Reduce Combiner
• Others (e.g., InputSplit)
• Configuration (more on that later)
MAP/REDUCE PROS

Distribution is completely transparent

• Not a single line of distributed programming (ease, correctness)

Automatic fault-tolerance

• Determinism enables running failed tasks somewhere else again
• Saved intermediate data enables just re-running failed reducers

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)
A FEW EXAMPLES
1. SEARCH

Input: (lineNumber, line) records
Output: lines matching a given pattern

Map: if(line matches pattern):
     output(line)

Reduce: identity function

• Alternative: no reducer (map-only job)
2. SORT

Input: (key, value) records
Output: same records, sorted by key

Map: identity function
Reduce: identity function

Trick: Pick partitioning function h such that $k_1 < k_2 \implies h(k_1) < h(k_2)$
What MapReduce tasks do you need to build an inverted index

A)  def Map (filename, text):
    foreach word in text.split(){
        output(word, filename)}

def Reduce(word, list(filename)):
    output(word, sort(filenames))

B)  def Map (filename, text):
    foreach word in text.split(){
        output(word, filename)}

def Combine(word, filenames):
    output(word, set(filenames))

def Reduce(word, filenames):
    output(word, sort(filenames))

C)  var globalHashMap = new HashMap on master-node
    def Map (filename, text):
        foreach word in text.split(){
            output(word, filename)}

    def Reduce(word, filenames):
        globalHashMap.add(word, sort(filenames))
INVERTED INDEX EXAMPLE

hamlet.txt
- to be or not to be

12th.txt
- be not afraid of greatness

afraid, (12th.txt)
- be, (12th.txt, hamlet.txt)
- greatness, (12th.txt)
- not, (12th.txt, hamlet.txt)
- of, (12th.txt)
- or, (hamlet.txt)
- to, (hamlet.txt)

be, (12th.txt, hamlet.txt)
- not, hamlet.txt

or, hamlet.txt
- be, hamlet.txt
- to, hamlet.txt
3. MOST POPULAR WORDS

Input: (filename, text) records
Output: top 100 words occurring in the most files drop rare words

Two-stage solution:

- **Job 1:**
  - Create inverted index, giving (word, list(file)) records
  - Important: do not remove duplicates

- **Job 2:**
  - Map each (word, list(file)) to (count, word)
  - Sort these records by count as in sort job

Optimizations:

- Map to (word, 1) instead of (word, file) in Job 1
- Count files in job 1’s reducer rather than job 2’s mapper
- Estimate count distribution in advance and drop rare words
HADOOP: THE 1st OPEN-SOURCE SYSTEM IMPLEMENTING THE MAPREDUCE PARADIGM
HADOOP ARCHITECTURE

Client \(\rightarrow\) Master Node

Master Node

\(\rightarrow\) Slave Node

Slave Node

Task Tracker

Task

HDFS

Data replicated across nodes
HADOOP ARCHITECTURE

Input

Map & Combine

Shuffle & Sort

Reduce

Output

the quick brown fox

the fox ate the mouse

how now brown cow

the, 1
brown, 1
fox, 1

the, 2
fox, 1

how, 1
now, 1
brown, 1

ate, 1
mouse, 1
cow, 1

brown, 2
fox, 2
how, 1
now, 1
the, 3

ate, 1
cow, 1
mouse, 1
quick, 1
For a very simple word count application on a cluster with 1000 nodes, each having two CPUs, 10 cores each, how many parallel MAP tasks (i.e., threads) per node should you use?

a) 20

b) 2 * 20

c) 60 – 100

d) More than 100
For a very simple word count application on a cluster with 1000 nodes, each having two CPUs, 10 cores each, how many parallel reduce tasks per node should you use?

a) The same number as map tasks
b) $2 \times 1.75 \times 20$ per node
c) $2 \times 0.95 \times 20$ per node
d) Needs to be fine tuned so that the output is a multiple of a block size
e) Needs to be fine tuned so that a reduce task takes between 5 and 10 minutes
Iterative Algorithms in MapReduce
Example KMeans

Select $K$ random data points $\{s_1, s_2, \ldots, s_K\}$ as centroids $c_j$. Until clustering converges or other stopping criterion{

For each data point $x_i$:

Assign $x_i$ to the closest centroid such that $dist(x_i, c_j)$ is minimal.

For each cluster $c_j$, update the centroids

$c_j = \mu(c_j)$

How do you express K-Means in the Map/Reduce paradigm?
Iterative Algorithms in MapReduce
Example KMeans

\textbf{Map1}(\text{filename}, \text{data}) := \text{emit data as (r-id, features)}

centroids[] = \text{read-centroids from disk}
\textit{Configure map2 job with centroids[]}
\textbf{Map2}(\text{r-id, features}) :=
\hspace{1em} \text{compare features (i.e., coordinates) with centroids}
\hspace{1em} \text{return (Closest-Centroid-ID, features)}

\textbf{Reduce}(\text{Centroid-ID, List[features]}) :=
\hspace{1em} \text{average features (i.e., coordinates) and emit (Centroid-ID, New-Coordinates)}

\text{Write new centroids to disk}
\text{Check if converged, if not do Map2 and Reduce again}
WHAT DO YOU THINK WERE THE REACTION OF THE DATABASE COMMUNITY?
I JUST DON'T CARE
MR VS.
DATABASES
HADOOP VS. RDBMS

Comparison of 3 systems

- Hadoop
- Vertica (a column-oriented database)
- DBMS-X (a row-oriented database)
  - rhymes with “schmoracle”

Qualitative

- Programming model, ease of setup, features, etc.

Quantitative

- Data loading, different types of queries

Pavlo et al. 2009
Grep Task

- Find 3-byte pattern in 100-byte record
  - 1 match per 10,000 records

- Data set:
  - 10-byte unique key, 90-byte value
  - 1TB spread across 25, 50, or 100 nodes
  - 10 billion records

- Original MR Paper (Dean et al. 2004)
Grep Task Loading Results

![Grep Task Loading Results Chart]

- **Hadoop**
- **Vertica**
- **DBMS-X**

- **25x40GB**
- **50x20GB**
- **100x10GB**

<table>
<thead>
<tr>
<th>Dataset Size</th>
<th>Hadoop</th>
<th>Vertica</th>
<th>DBMS-X</th>
</tr>
</thead>
<tbody>
<tr>
<td>25x40GB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50x20GB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100x10GB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Grep Task Execution Results

- **25x40GB**: Hadoop > Vertica > DBMS-X
- **50x20GB**: Hadoop > Vertica > DBMS-X
- **100x10GB**: Hadoop > Vertica > DBMS-X
SELECTION TASK

SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X

1 GB / node
Analytical Tasks

- Simple web processing schema

- Data set:
  - 600k HTML Documents (6GB/node)
  - 155 million UserVisit records (20GB/node)
  - 18 million Rankings records (1GB/node)
Aggregate Task

- Simple query to find adRevenue by IP prefix

\[
\text{SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)}
\]

\[
\text{FROM userVisits}
\]

\[
\text{GROUP BY SUBSTR(sourceIP, 1, 7)}
\]
Aggregate Task Results

- Hadoop
- Vertica
- DBMS-X

- 25 nodes
- 50 nodes
- 100 nodes
Join Task

- Find the sourceIP that generated the most adRevenue along with its average pageRank.

- Implementations:
  - **DBMSs** – Complex SQL using temporary table.
  - **MapReduce** – Three separate MR programs.
Join Task Results

- **Hadoop**
- **Vertica**
- **DBMS-X**

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Hadoop</th>
<th>Vertica</th>
<th>DBMS-X</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td></td>
<td>32.0</td>
<td>29.2</td>
</tr>
<tr>
<td>50</td>
<td>35.4</td>
<td>29.4</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>55.0</td>
<td>31.9</td>
<td></td>
</tr>
</tbody>
</table>
PROBLEMS WITH THIS ANALYSIS?

Other ways to avoid sequential scans?
Fault-tolerance in large clusters?
Tasks that cannot be expressed as queries?
Google’s Response: Cluster Size

- Largest known database installations:
  - *Greenplum* – 96 nodes – 4.5 PB (eBay) [1]
  - *Teradata* – 72 nodes – 2+ PB (eBay) [1]

- Largest known MR installations:
  - *Hadoop* – 3658 nodes – 1 PB (Yahoo) [2]
  - *Hive* – 600+ nodes – 2.5 PB (Facebook) [3]

[1] eBay’s two enormous data warehouses – April 30th, 2009

http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html

Concluding Remarks

- What can MapReduce learn from Databases?
  - Declarative languages are a good thing.
  - Schemas are important.

- What can Databases learn from MapReduce?
  - Query fault-tolerance.
  - Support for in situ data.
  - Embrace open-source.
APACHE 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

https://pig.apache.org/

Similar to Google’s (internal) Sawzall project
EXAMPLE PROBLEM

Given *user data* in one file, and *website data* in another, find the *top 5 most visited pages by users aged 18-25*

- Load Users
- Filter by age
- Load Pages
- Join on name
- Group on url
- Count clicks
- Order by clicks
- Take top 5

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:

- **Load Users**
- **Filter by age**
- **Join on name**
- **Group on url**
- **Count clicks**
- **Order by clicks**
- **Take top 5**

Job 1:

- **Users = load ...**
- **Filtered = filter ...**
- **Pages = load ...**
- **Joined = join ...**
- **Grouped = group ...**
- **Summed = ... count()...**
- **Sorted = order ...**
- **Top5 = limit ...**

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
**APACHE 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 many Facebook jobs

Now used by many others

- Netflix, Amazon, …

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

Efficient primitives for data sharing

Iterative job

Interactive mining

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

Efficient primitives for data sharing

Problem: in MR, the only way to share data across jobs is using stable storage (e.g. file system) → slow!
Oppportunity: DRAM is getting cheaper ➔ use main memory for intermediate results instead of disks
GOAL: IN-MEMORY DATA SHARING

Input → iter. 1 → iter. 2 → ... → one-time processing → Distributed memory

query 1 → query 2 → query 3 → ...

10-100 × faster than network and disk
Resilient distributed datasets (RDDs)

• Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
• Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
• Can be cached across parallel operations

Parallel operations on RDDs

• Reduce, collect, count, save, …

Restricted shared variables

• Accumulators, broadcast variables
Example: Log Mining

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

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

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count

Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)
```
RDDS IN MORE DETAIL

An RDD is an immutable, partitioned, logical collection of records
  • Need not be materialized, but rather contains information to rebuild a dataset from stable storage

Partitioning can be based on a key in each record (using hash or range partitioning)

Built using bulk transformations on other RDDs

Can be cached for future reuse
# RDD OPERATIONS

<table>
<thead>
<tr>
<th>Transformations  (define a new RDD)</th>
<th>Parallel operations  (return a result to driver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
</tr>
<tr>
<td>filter</td>
<td>collect</td>
</tr>
<tr>
<td>sample</td>
<td>count</td>
</tr>
<tr>
<td>union</td>
<td>save</td>
</tr>
<tr>
<td>groupByKey</td>
<td>lookupKey</td>
</tr>
<tr>
<td>reduceByKey</td>
<td>...</td>
</tr>
<tr>
<td>join</td>
<td></td>
</tr>
<tr>
<td>cache</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
RDDs maintain *lineage* information that can be used to reconstruct lost partitions

**Ex:** `cachedMsgs = textFile(...).filter(_.contains("error")) .map(_.split('t')(2)) .cache()`
BENEFITS OF RDD MODEL

Consistency is easy due to immutability

Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)

Locality-aware scheduling of tasks on partitions

Despite being restricted, model seems applicable to a broad variety of applications
### RDDS VS DISTRIBUTED SHARED MEMORY

<table>
<thead>
<tr>
<th>Concern</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Bulk transformations</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using speculative execution</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (but runtime aims for transparency)</td>
</tr>
</tbody>
</table>