CS 347
Parallel and Distributed Data Processing
Spring 2016

Notes 11: MapReduce
Motivation

Distribution makes simple computations complex

Communication
Load balancing
Fault tolerance

Not all applications require relational data and transactions

What if we could automatically parallelize simple data processing?
Motivation

E.g., range-partition sort

```
Preprocessing & partition                      Additional processing
```

\[ R' \]

\[ R' \]

\[ R' \]

\[ k_0 \]

\[ k_1 \]

\[ k_2 \]
Motivation

E.g., asymmetric fragment + replicate join

Preprocessing & partition  →  Additional processing

partition

local join

result

union
Motivation

What if we didn’t have to think too hard about sites or fragments?

Goal
Provide a framework that hides the distribution from the developers, so they can focus on the fundamental nuggets of computation instead
Building a Text Index

Original MapReduce application

- **Loading**
  - Page stream
  - 1: ant, dog
  - 2: dog, cat
  - 3: ant, dog

- **Tokenizing**
  - (ant, 1)
  - (dog, 1)
  - (dog, 2)
  - (cat, 2)
  - (ant, 3)
  - (dog, 3)

- **Sorting**
  - (ant, 1)
  - (ant, 3)
  - (cat, 2)
  - (dog, 1)
  - (dog, 2)
  - (dog, 3)

- **Intermediate runs**
  - Disk

- **Flushing**
Building a Text Index

Intermediate runs

| ( ant, 1 ) |
| ( ant, 3 ) |
| ( cat, 2 ) |
| ( dog, 1 ) |
| ( dog, 2 ) |
| ( dog, 3 ) |

| ( ant, 5 ) |
| ( cat, 4 ) |
| ( dog, 4 ) |
| ( dog, 5 ) |
| ( eel, 6 ) |

Merge

| ( ant, 1 ) |
| ( ant, 3 ) |
| ( ant, 5 ) |
| ( cat, 2 ) |
| ( cat, 4 ) |
| ( dog, 1 ) |
| ( dog, 2 ) |
| ( dog, 3 ) |
| ( dog, 4 ) |
| ( dog, 5 ) |
| ( eel, 6 ) |

Final index

| ( ant: 1, 3, 5 ) |
| ( cat: 2, 4 ) |
| ( dog: 1, 2, 3, 4, 5 ) |
| ( eel: 6 ) |
Generalization

- Tokenizing
- Sorting
- Loading

MAP

Page stream

Loading

Tokenizing

Sorting

Intermediate runs

Disk

Flushing
Generalization

Intermediate runs

- (ant, 1)
- (ant, 3)
- (cat, 2)
- (dog, 1)
- (dog, 2)
- (dog, 3)
- (ant, 5)
- (cat, 4)
- (dog, 4)
- (dog, 5)
- (eel, 6)

Merge

- (ant, 1)
- (ant, 3)
- (ant, 5)
- (cat, 2)
- (cat, 4)
- (dog, 1)
- (dog, 2)
- (dog, 3)
- (dog, 4)
- (dog, 5)
- (eel, 6)

REDUCE

Final index

- (ant: 1, 3, 5)
- (cat: 2, 4)
- (dog: 1, 2, 3, 4, 5)
- (eel: 6)
Computational Model

Input

\[ D = \{ d_1, d_2, \ldots, d_n \} \]
\[ M(d_i) \rightarrow \{ [k_1, v_1], [k_2, v_2], \ldots \} \]
\[ R(k_i, \{ \text{values} \}) \rightarrow \text{result} \]

Let \( S = \{ [k, v] \mid [k, v] \in M(d) \text{ for some } d \in D \} \)

Let \( K = \{ k \mid [k, v] \in S, \text{ for any } v \} \)

Let \( G(k) = \{ v \mid [k, v] \in S \} \)

Output

\[ O = \{ [k, r] \mid k \in K, r = R(k, G(k)) \} \]
Example

Counting word occurrences

map(String doc, String value):
  for each word w in value
    emitIntermediate(w, “1”)

E.g., map(doc, “cat dog cat bat dog”) emits
[“cat”, “1”], [“dog”, “1”], [“cat”, “1”], [“bat”, “1”], [“dog”, “1”]
Example

Counting word occurrences

reduce(String key, Iterator values):
    int result = 0
    for each v in values result += parseInt(v)
    emit(asString(result))

E.g., reduce("dog", { "1", "1", "1", "1" }) emits "4"
References

*MapReduce: Simplified Data Processing on Large Clusters.* J. Dean and S. Ghemawat, OSDI 2004

*Pig Latin: A Not-So-Foreign Language for Data Processing.* Olston et al, SIGMOD 2008
Processing Model

User Program

Master

(1) fork

(1) fork

(1) fork

(2) assign map

(2) assign reduce

worker

worker

worker

worker

worker

worker

(3) read

(4) local write

(5) remote read

(6) write

split 0

split 1

split 2

split 3

split 4

Input files

Map phase

Intermediate files (on local disks)

Reduce phase

Output files

output file 0

output file 1
Can vary the number of mappers to tune performance

Reduce tasks bounded by number of intermediate files produced by each map worker
Implementation Issues

Combine function
File system
Partitions
Latency
Failures
Backup tasks
Ordering of results
Pipelines
Combine Function

Combine is a *local reduce* step applied before distribution:

["cat", "1"], ["cat", "1"], ["cat", "1"] → worker

["dog", "1"], ["dog", "1"] → worker

["cat", "3"] → worker

["dog", "2"] → worker
File System

All data transfers are through a distributed file system; high-throughput network is essential.

Map workers must be able to access any split of the input; input is a distributed file.

Reduce worker must be able to access local disks on map workers.

Any worker must be able to write its part of the output; output is left in a distributed file.
Partitions

How many splits and workers are best?

- **How many splits?**
- **How many workers?**

**Best to have many splits per worker:** improves load balance; if worker fails, easier to spread its task.

**Should workers be assigned to splits near them?**

**Similar questions for reduce workers**
Latency

What takes time?
   Reading the input
   Mapping
   Shuffling
   Reducing

Map and reduce are separate phases
   Latency determined by slowest task
   Reduce task (data) skew can significantly increase latency

Map and shuffle can overlap
   Once some mappers are done, shuffle may commence
Failures

Distributed implementation should produce same output as a non-faulty sequential execution

General strategy
Master detects worker failures, gets task re-done by another worker
Failures

Distributed implementation should produce same output as a non-faulty sequential execution

General strategy
Master detects worker failures, gets task re-done by another worker
Backup Tasks

Stragglers are workers that take unusually long finish their tasks. They can delay overall completion.

When overall processing is close to completion, the master may schedule backup execution for remaining tasks. Must be able to eliminate redundant results.
Ordering of Results

Results are in key order
Example

Sorting records

Map: extract k, output [k, record]

Reduce: Do nothing

One or two records for k = 6?
Pipelines

The output of one MapReduce can become the input for another

Example

Identifying top-k terms

Stage 1: translate data to canonical form
Stage 2: term count
Stage 3: sort by frequency
Stage 4: extract top-k
MapReduce Implementations

Google MapReduce
Proprietary system

Hadoop
Open-source MapReduce framework
Also, a toolkit
HDFS: filesystem for Hadoop
HBase: database for Hadoop

Also, an ecosystem
Tools
Recipes
Developer community
MapReduce Advantages

Model easy to use, hides details of parallelization and fault recovery

Many problems can be expressed in the MapReduce framework

Scales to thousands of machines
MapReduce Disadvantages

1-input, 2-stage data flow is rigid, hard to adapt to other scenarios

Custom code is needed for even the most common operations
  E.g., projections and filtering

The opaque nature of map/reduce functions impedes optimization
Questions

Can MapReduce support traditional database operators?
   E.g., how can we perform joins?

Can MapReduce be made more declarative?
Supporting Database Operators

Simple idea
Each operator is a MapReduce stage

How to do?
  Selection
  Projection
  Group by and aggregation
Joins

Reduce-side join
Shuffle puts all values for the same key at the same reducer

Mapper
  Input: tuples from \( R \); tuples from \( S \)
  Output: \([ \text{join value}, (R|S, \text{tuple}) ]\)

Reducer
  Local join of all \( R \) tuples with all \( S \) tuples
Joins

Map-side join
Like a hash-join, but every mapper has a copy of the hash table

Mapper
  Read hash table of \( R \)
  Input: tuples of \( S \)
  Output: tuples of \( S \) joined with tuples of \( R \)

Reducer
  Pass through
Joins

Comparison
Reduce-side join shuffles all the data
Map-side join requires one table to be small
Joins

Semi-joins

One idea

Stage 1: Extract the join keys of $R$; reduce-side join with $S$
Stage 2: Map-side join result of stage 1 with $R$
Pig & Pig Latin

A layer on top of MapReduce (Hadoop)
   Pig is the system
   Pig Latin is the query language

Pig Latin is a hybrid between
   High-level *declarative* query language in the spirit of SQL
   Low-level procedural programming using MapReduce
Example

Data
Info about web pages: \text{urls}(\text{url}, \text{category}, \text{score})

Problem
Find, for each sufficiently large category, the average score of high-scoring pages in that category
Example

SQL:

```
SELECT category, AVG(score)
FROM urls
WHERE score > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6
```

Pig Latin:

```
high_score_urls = FILTER urls BY score > 0.2;
groups = GROUP high_score_urls BY category;
large_groups = FILTER groups
             BY COUNT(high_score_urls) > 10^6;
output = FOREACH large_groups GENERATE category,
         AVG(high_score_urls.score);
```
Example

```
high_score_urls = FILTER urls BY score > 0.2;
```

<table>
<thead>
<tr>
<th>url</th>
<th>category</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>z.cnn.com</td>
<td>.com</td>
<td>0.9</td>
</tr>
<tr>
<td>y.yale.edu</td>
<td>.edu</td>
<td>0.5</td>
</tr>
<tr>
<td>w.uc.edu</td>
<td>.edu</td>
<td>0.1</td>
</tr>
<tr>
<td>x.nyt.com</td>
<td>.com</td>
<td>0.8</td>
</tr>
<tr>
<td>y.ut.edu</td>
<td>.edu</td>
<td>0.6</td>
</tr>
<tr>
<td>w.wh.gov</td>
<td>.gov</td>
<td>0.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>url</th>
<th>category</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>z.cnn.com</td>
<td>.com</td>
<td>0.9</td>
</tr>
<tr>
<td>y.yale.edu</td>
<td>.edu</td>
<td>0.5</td>
</tr>
<tr>
<td>x.nyt.com</td>
<td>.com</td>
<td>0.8</td>
</tr>
<tr>
<td>y.ut.edu</td>
<td>.edu</td>
<td>0.6</td>
</tr>
<tr>
<td>w.wh.gov</td>
<td>.gov</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Example

groups = GROUP high_score_urls BY category;

<table>
<thead>
<tr>
<th>url</th>
<th>category</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>z.cnn.com</td>
<td>.com</td>
<td>0.9</td>
</tr>
<tr>
<td>y.yale.edu</td>
<td>.edu</td>
<td>0.5</td>
</tr>
<tr>
<td>x.nyt.com</td>
<td>.com</td>
<td>0.8</td>
</tr>
<tr>
<td>y.ut.edu</td>
<td>.edu</td>
<td>0.6</td>
</tr>
<tr>
<td>w.wh.gov</td>
<td>.gov</td>
<td>0.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>category</th>
<th>high_score_urls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.com</td>
<td>{ ( z.cnn.com, .com, 0.9 ) ( x.nyt.com, .com, 0.8 ) }</td>
</tr>
<tr>
<td>.edu</td>
<td>{ ( y.yale.edu, .edu, 0.5 ) ( y.ut.edu, .edu, 0.6 ) }</td>
</tr>
<tr>
<td>.gov</td>
<td>{ ( w.wh.gov, .gov, 0.7 ) }</td>
</tr>
</tbody>
</table>
Example

large_groups = FILTER groups BY COUNT(high_score_urls) > 1;

groups

<table>
<thead>
<tr>
<th>category</th>
<th>high_score_urls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.com</td>
<td>{ ( z.cnn.com, .com, 0.9 ) ( x.nyt.com, .com, 0.8 ) }</td>
</tr>
<tr>
<td>.edu</td>
<td>{ ( y.yale.edu, .edu, 0.5 ) ( y.ut.edu, .edu, 0.6 ) }</td>
</tr>
<tr>
<td>.gov</td>
<td>{ ( w.wh.gov, .gov, 0.7 ) }</td>
</tr>
</tbody>
</table>

large_groups

<table>
<thead>
<tr>
<th>category</th>
<th>high_score_urls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.com</td>
<td>{ ( z.cnn.com, .com, 0.9 ) ( x.nyt.com, .com, 0.8 ) }</td>
</tr>
<tr>
<td>.edu</td>
<td>{ ( y.yale.edu, .edu, 0.5 ) ( y.ut.edu, .edu, 0.6 ) }</td>
</tr>
</tbody>
</table>
Example

output = FOREACH large_groups GENERATE category, AVG(high_score_urls.score);

<table>
<thead>
<tr>
<th>category</th>
<th>high_score_urls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.com</td>
<td>{ ( z.cnn.com, .com, 0.9 ) ( x.nyt.com, .com, 0.8 )}</td>
</tr>
<tr>
<td>.edu</td>
<td>{ ( y.yale.edu, .edu, 0.5 ) ( y.ut.edu, .edu, 0.6 )}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>category</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>.com</td>
<td>0.85</td>
</tr>
<tr>
<td>.edu</td>
<td>0.55</td>
</tr>
</tbody>
</table>
Pig Features

Similar to specifying a query execution plan
   Easy for developers to understand and control processing flow

Supports a flexible, fully nested data model

Supports user-defined functions

Can operate over plain input files without any schema information

Provides debugging environment, useful for large data sets
Execution Control

Good or bad?

Example

spam_urls = FILTER urls BY isSpam(url);
culprit_urls = FILTER spam_urls BY score > 0.8;

Should system re-order filters?
User Defined Functions

Example

groups = GROUP urls BY category;
output = FOREACH groups GENERATE category, top10(groups.url);

groups

<table>
<thead>
<tr>
<th>category</th>
<th>urls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.gov</td>
<td>{ ( x.fbi.gov, .gov, 0.7 ) ... }</td>
</tr>
<tr>
<td>.edu</td>
<td>{ ( y.yale.edu, .edu, 0.5 ) ... }</td>
</tr>
<tr>
<td>.com</td>
<td>{ ( z.cnn.com, .com, 0.9 ) ... }</td>
</tr>
</tbody>
</table>

output

<table>
<thead>
<tr>
<th>category</th>
<th>urls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.gov</td>
<td>{ ( fbi.gov ) ( cia.gov ) ... }</td>
</tr>
<tr>
<td>.edu</td>
<td>{ ( yale.edu ) ... }</td>
</tr>
<tr>
<td>.com</td>
<td>{ ( cnn.com ) ( ibm.com ) ... }</td>
</tr>
</tbody>
</table>
Data Model

Atom

‘alice’

Tuple

(‘alice’, ‘lakers’)

Bag

{ (‘alice’, ‘lakers’), (‘alice’, (‘iphone’, ‘apple’)) }

Map

[ ‘interests’ → { (‘lakers’), (‘phone’) }, ‘age’ → 20 ]

Bags can only hold tuples, so \{ 1, 2, 3 \} is stored as \{(1), (2), (3)\}
## Expressions

\[
t = (\text{Alice}, \{ ('lakers', 1), ('iPod', 2) \} , [\text{age} \rightarrow 20])
\]

Let fields of tuple \( t \) be called \( f_1, f_2, f_3 \)

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Example</th>
<th>Value for ( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>‘bob’</td>
<td>Independent of ( t )</td>
</tr>
<tr>
<td>Field by position</td>
<td>$0</td>
<td>‘alice’</td>
</tr>
<tr>
<td>Field by name</td>
<td>( f_3 )</td>
<td>{\text{age} \rightarrow 20}</td>
</tr>
<tr>
<td>Projection</td>
<td>( f_2.$0 )</td>
<td>{('lakers'), ('iPod')}</td>
</tr>
<tr>
<td>Map Lookup</td>
<td>( f_3#\text{age} )</td>
<td>20</td>
</tr>
<tr>
<td>Function Evaluation</td>
<td>( \text{SUM}(f_2.$1) )</td>
<td>( 1 + 2 = 3 )</td>
</tr>
<tr>
<td>Conditional Expression</td>
<td>( f_3#\text{age} &gt; 18? )</td>
<td>‘adult’</td>
</tr>
<tr>
<td></td>
<td>‘adult’ : ‘minor’</td>
<td>( \text{‘adult’} )</td>
</tr>
<tr>
<td>Flattening</td>
<td>( \text{FLATTEN}(f_2) )</td>
<td>{‘lakers’, 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘iPod’, 2</td>
</tr>
</tbody>
</table>
Loading Data

queries = LOAD 'query_log.txt'
USING myLoad()
AS (user, queryString, timestamp);
For Each and Flatten

Basic form:
expanded_queries = FOREACH queries GENERATE
    user, expandQuery(queryString);

Removing one level of nesting:
expanded_queries = FOREACH queries GENERATE
    user, FLATTEN(expandQuery(queryString));

Each tuple is processed independently → good for parallelism
For Each and Flatten

queries
(user, queryString, timestamp)

(‘alice’, ‘lakers’, 1)
(‘bob’, ‘iphone’, 3)

FOREACH queries GENERATE user, expandQuery(queryString)

Without flattening

With flattening

(‘alice’, ‘lakers rumors’)
(‘alice’, ‘lakers news’)
(‘bob’, ‘iphone 6’)
(‘bob’, ‘iphone apps’)

queries
('alice', 'lakers', 1)
('bob', 'iphone', 3)
Filter

real_queries = 
  FILTER queries BY user <> 'bot';

real_queries = 
  FILTER queries BY NOT isBot(user);
Group

grouped_revenues = GROUP revenue BY queryString;

query_revenues = FOREACH grouped_revenues
    GENERATE queryString, 
    SUM(revenue.amount) AS totalRevenue;
Co-Group and Join

results(queryString, url, rank)
revenue(queryString, adSlot, amount)

grouped_data = COGROUP results BY queryString, 
    revenue BY queryString;
url_revenues = FOREACH grouped_data GENERATE 
    FLATTEN(distributeRevenue(results, revenue));
Co-Group and Join

results:
(queryString, url, rank)

(revenue:
(queryString, adSlot, amount)

(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)

(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)

COGROUP

grouped_data: (group, results, revenue)

(lakers,
(lakers, nba.com, 1)
(lakers, espn.com, 2)
),
(lakers, top, 50)
(lakers, side, 20)

(kings,
(kings, nhl.com, 1)
(kings, nba.com, 2)
),
(kings, top, 30)
(kings, side, 10)

JOIN

(lakers, nba.com, 1, top, 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)

...
Co-Group and Join

Join operator

\[
\text{join\_result} = \text{JOIN results BY queryString, revenue BY queryString};
\]

Shorthand for

\[
\text{temp\_var} = \text{COGROUP results BY queryString, revenue BY queryString};
\]

\[
\text{join\_result} = \text{FOREACH temp\_var GENERATE FLATTEN(results), FLATTEN(revenue)};
\]

Co-group more flexible than join
MapReduce in Pig Latin

map_result = FOREACH input GENERATE FLATTEN(map(*));

key_groups = GROUP map_result BY $0;

output = FOREACH key_groups GENERATE reduce(*);
Storing Data

STORE query_revenues INTO 'output.csv' USING myStore();

Output file

Custom serializer
Why Not Use a DBMS?

Many database systems are highly optimized
  Loading data is hard (e.g., need schema)
  May come with system overhead (e.g., transactions)

MapReduce is
  Scalable
  Easy to deploy/use/extend
  Free
  Suitable for diverse large, parallelizable computation
    E.g., building inverted indices or thumbnailing images

One idea
  Use DBMS for data input/output
Summary

MapReduce
  Motivation
  Model
  Implementation details
  Advantages and disadvantages
  Database operators
Pig & Pig Latin
  Features
  Model
  Operators