Data Science at Scale: MapReduce & Spark

CSE481DS Data Science Capstone
Mike Merrill

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING
Group Reflection on Process & Validity
Lecture: Distributed Computing for Data Science
Agenda

- **Commodity Computing**
  Computing with thousands of failures a day

- **Map Reduce**
  Plumbing for billions of data points

- **Spark**
  The best tool for a nasty problem

- **Data Engineering in Practice**
  What to look forward to

- **Lab!**
Large-scale Computing

- **Large-scale computing for data mining problems on commodity hardware**

- **Challenges:**
  - How do you distribute computation?
  - How can we make it easy to write distributed programs?
  - Machines fail:
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to lose 1/day
    - With 1M machines 1,000 machines fail every day!

Tim Althoff, UW CSE481DS: Data Science Capstone, http://www.cs.washington.edu/cse481ds
Storage Infrastructure

- **Problem:**
  - If nodes fail, how to store data persistently?

- **Answer:**
  - **Distributed File System**
    - Provides global file namespace

- **Typical usage pattern:**
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common
# Distributed File System

- **Reliable distributed file system**
- Data kept in “chunks” spread across machines
- Each chunk **replicated** on different machines
  - Seamless recovery from disk or machine failure

---

Chunk servers also serve as compute servers

Bring computation directly to the data!
An Issue and a Solution

- **Issue:**
  Copying data over a network takes time

- **Idea:**
  - Bring computation to data
  - Store files multiple times for reliability

- **Map Reduce** address these problems
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - MapReduce
    - Spark
MapReduce is a style of programming designed for:

1. Easy parallel programming
2. Invisible management of hardware and software failures
3. Easy management of very-large-scale data

It has several implementations, including Hadoop, Spark (used in this class), Flink, and the original Google implementation just called “MapReduce”
MapReduce: Overview

3 steps of MapReduce

- **Map:**
  - Apply a user-written *Map function* to each input element
    - *Mapper* applies the Map function to a single element
      - Many mappers grouped in a *Map task* (the unit of parallelism)
    - The output of the Map function is a set of 0, 1, or more *key-value pairs*.

- **Group by key:** Sort and shuffle
  - System sorts all the key-value pairs by key, and outputs key-(list of values) pairs

- **Reduce:**
  - User-written *Reduce function* is applied to each key-(list of values)

Outline stays the same, **Map** and **Reduce** change to fit the problem
**Map-Reduce: A diagram**

**MAP:**
Read input and produces a set of key-value pairs

**Group by key:**
Collect all pairs with same key (Hash merge, Shuffle, Sort, Partition)

**Reduce:**
Collect all values belonging to the key and output

```
Input

Intermediate
k1:v k1:v k2:v k1:v k3:v k4:v k4:v k5:v k4:v k1:v k3:v

Group by Key

Grouped
k1:v,v,v,v k2:v k3:v,v k4:v,v,v k5:v

Output
```

Tim Althoff, UW CSE481DS: Data Science Capstone, http://www.cs.washington.edu/cse481ds
Example: Word Counting

Example MapReduce task:
- We have a huge text document
- Count the number of times each distinct word appears in the file

Many applications of this:
- Analyze web server logs to find popular URLs
- Statistical machine translation:
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/machine partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need."

**MapReduce: Word Counting**

<table>
<thead>
<tr>
<th>Big document</th>
<th>Provided by the programmer</th>
<th>Provided by the programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(key, value)</td>
<td><strong>Map:</strong> Read input and produces a set of key-value pairs</td>
<td><strong>Group by key:</strong> Collect all pairs with same key</td>
</tr>
<tr>
<td>(key, value)</td>
<td>(The, 1) (crew, 1) (of, 1) (the, 1) (space, 1) (shuttle, 1) (Endeavor, 1) (recently, 1) ....</td>
<td>(crew, 1) (crew, 1) (space, 1) (shuttle, 1) (recently, 1) ....</td>
</tr>
<tr>
<td>(key, value)</td>
<td>(crew, 2) (space, 1) (the, 3) (shuttle, 1) (recently, 1) ....</td>
<td>(key, value)</td>
</tr>
</tbody>
</table>

**Reduce:** Collect all values belonging to the key and output
Word Count Using MapReduce

\[
\text{map} (\text{key}, \text{value}) : \\
\text{# key: document name; value: text of the document} \\
\text{for each word w in value:} \\
\quad \text{emit}(w, 1)
\]

\[
\text{reduce} (\text{key}, \text{values}) : \\
\text{# key: a word; value: an iterator over counts} \\
\quad \text{result} = 0 \\
\quad \text{for each count v in values:} \\
\qquad \text{result} += v \\
\quad \text{emit} (\text{key}, \text{result})
\]
Map-Reduce: In Parallel

All phases are distributed with many tasks doing the work
MapReduce Pattern

Input

key-value pairs

Mappers

Reducers

Output
MapReduce: Environment

MapReduce environment takes care of:

- **Partitioning** the input data
- **Scheduling** the program’s execution across a set of machines
- Performing the *group by key* step
  - In practice this is the bottleneck
- Handling machine *failures*
- Managing required inter-machine communication
Dealing with Failures

- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle and rescheduled
  - Reduce workers are notified when map task is rescheduled on another worker

- **Reduce worker failure**
  - Only in-progress tasks are reset to idle and the reduce task is restarted
Spark
Problems with MapReduce

- **Two major limitations of MapReduce:**
  - Difficulty of programming directly in MR
    - Many problems aren’t easily described as map-reduce
  - Performance bottlenecks, or batch not fitting the use cases
    - Persistence to disk typically slower than in-memory work

- **In short, MR doesn’t compose well for large applications**
  - Many times one needs to chain multiple map-reduce steps
MapReduce uses two “ranks” of tasks: One for Map the second for Reduce
- Data flows from the first rank to the second

Data-Flow Systems generalize this in two ways:
1. Allow any number of tasks/ranks
2. Allow functions other than Map and Reduce
- As long as data flow is in one direction only, we can have the blocking property and allow recovery of tasks rather than whole jobs
Spark: Most Popular Data-Flow System

- Expressive computing system, not limited to the map-reduce model

- Additions to MapReduce model:
  - Fast data sharing
    - Avoids saving intermediate results to disk
    - Caches data for repetitive queries (e.g. for machine learning)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce

- Compatible with Hadoop
Spark: Overview

- Open source software (Apache Foundation)
- Supports Java, Scala and Python

- **Key construct/idea:** Resilient Distributed Dataset (RDD)

- **Higher-level APIs:** DataFrames & DataSets
  - Introduced in more recent versions of Spark
  - Different APIs for aggregate data, which allowed to introduce SQL support
Spark: RDD

Key concept *Resilient Distributed Dataset* (RDD)

- Partitioned collection of records
  - Generalizes (key-value) pairs
- Spread across the cluster, Read-only
- Caching dataset in memory
  - Different storage levels available
  - Fallback to disk possible

- RDDs can be created from Hadoop, or by transforming other RDDs (you can stack RDDs)
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
Spark RDD Operations

- **Transformations** build RDDs through deterministic operations on other RDDs:
  - Transformations include \( \text{map}, \text{filter}, \text{join}, \text{union}, \text{intersection}, \text{distinct} \)
  - **Lazy evaluation**: Nothing computed until an action requires it

- **Actions** to return value or export data:
  - Actions include \( \text{count}, \text{collect}, \text{reduce}, \text{save} \)
  - Actions can be applied to RDDs; actions force calculations and return values
Task Scheduler: General DAGs

- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles
Dataframe & Dataset

- **Dataframe:**
  - Unlike an RDD, data organized into named columns, e.g. a *table in a relational database*.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction.

- **Dataset:**
  - Extension of Dataframe API which provides **type-safe, object-oriented programming interface** (compile-time error detection).

Both built on Spark SQL engine. Both can be converted back to an RDD.
Spark vs. Hadoop MapReduce

- **Performance:** Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to perform well; if there are other resource-demanding services or can’t fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- **Ease of use:** Spark is easier to program (higher-level APIs)
- **Data processing:** Spark is more general
Data Engineering in Practice
Data Analytics Software Stack

Spark Streaming
- Stream processing

GraphX
- Graph computation

MLlib
- User-friendly machine learning

SparkSQL
- SQL API

Hive

Storm

MPI

Spark
- Fast memory-optimized execution engine (Python/Java/Scala APIs)

Tachyon
- Distributed Memory-Centric Storage System

Hadoop Distributed File System (HDFS)

Mesos
- Cluster resource manager, multi-tenancy
Some other technologies to keep an eye on...

**Pros:** Map Reduce with Pandas API
**Cons:** Unstable, Not much support

**Pros:** Map Reduce via SQL, integrations
**Cons:** Not as flexible

**Pros:** Loads of integrations, promises a lot
**Cons:** New-ish player, lots of development ahead
5 min break 😊
Thank you for sharing your feedback with us!

Lab Part: Intro to Spark