Tools & Models for Data Science

Big Data Part One: An Introduction to MapReduce

Chris Jermaine & Risa Myers

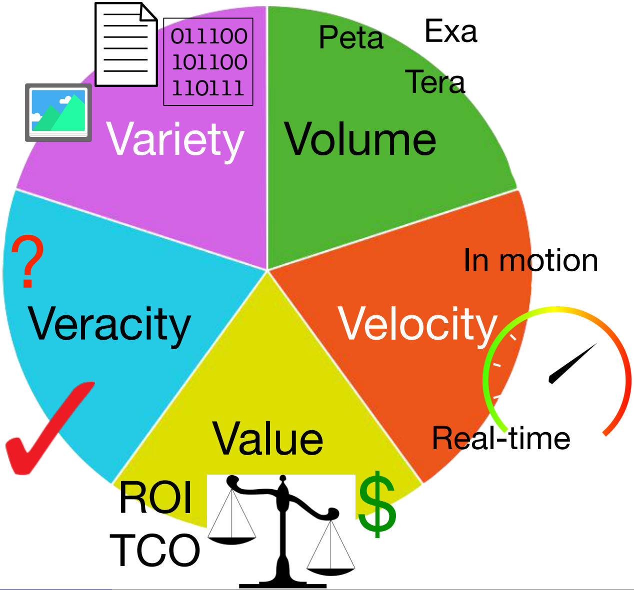
Rice University

15 Years Ago...

- Say you had a big data set, wanted platform to analyze it
- What is "big"?
  - Too large to fit in RAM of an expensive server machine
  - "Big Data" termed appeared in late 90's
- Is 5GB in 2002, a couple of terabytes in 2018
- You might spend \$20 - 30K on a server

The 5 V's of "Big" Data

1. Volume
2. Variety
3. Velocity
4. Veracity
5. Value



7 V's

1. Volume
2. Variety
3. Velocity
4. Veracity
5. Value
6. Variability
7. Visualization

How to Analyze a "Big" Dataset

- You might write your own sofware
- Costly, time consuming
  - A $10M software feature might eat up most of the IT budget for a single firm
- Requires expertise not always found in house
- Risky: high potential for failure

Or, You Might Buy a DB System

- Costs a LOT of money
- Performance often unpredictable, or just flat out poor
- Software can be insanely complicated to use correctly
- Software stack too big/deep, not possible to unbundle
  - If you are doing analysis, ACID not important
  - And yet, you pay for it (money, complexity, performance)
- Difficult to put un- or semi-structured data into an SQL DB

Plus, Many People Just Don't Like SQL

- People uncomfortable with declarative programming
  - We love it!
  - But users don't really know what's happening under the hood
  - Makes many programmers uncomfortable
- Also, not easy/natural to specify important computations
  - Especially data mining and machine learning
  - Not to mention High Performance Computing-style computations (Analytics)

By Early-Mid 2000's...

- The Internet companies (Google, Yahoo, etc.)...
  - ...had some of the largest databases in the world
  - But they never used classical SQL databases for webscale
- How'd they do it?
  - Many ways...
  - But paradigm with most widespread impact was MapReduce
  - First described in a 2004 academic paper, appeared in Operating Systems Design and Implementation (OSDI)
  - "MapReduce: Simplified Data Processing on Large Clusters"

Stage is set for a New Paradigm

- Existing approaches don't work
  - Custom software is too expensive
  - RDBMSs are too rigid
- Data volume has exploded
- Commodity hardware has become cheap
- Cost is proportional to workload
- Welcome MapReduce!

What Is MapReduce?

- A simple data processing paradigm
- Leverages a cluster of commodity machines
  - For distributed processing
  - For fault tolerance
  - Requires a distributed file system
  - See “The Google File System” by Ghemawat, et al. 2003
- Well suited to calculations that can be computed in independent chunks

What Is Novel about MapReduce?

- Novelty is in scalability and fault tolerance
- The functional model already existed
- Consider the map and reduce functions in Python
- Map:

```
items = [ 1, 2, 3, 4, 5 ]
mod2 = list(map(lambda x:  x % 2, items))
        [ 1, 0, 1, 0, 1 ]
```

- Reduce:

```
from functools imoport reduce
mySum = reduce((lambda x,y:  x + y), items)
15
```

MapReduce

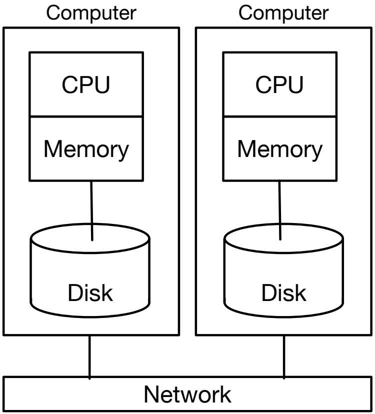
- To process a data set:
  - You have two pieces of user-supplied code
  - A Map code
  - And a Reduce code

MapReduce

- These are run in a huge shared-nothing compute cluster
- Using three data processing phases
  1. A Map phase
  2. A Shuffle phase
  3. And a Reduce phase

First: What Is Shared-Nothing?

- Store/analyze data on a large number of commodity machines
  - Local, non-shared storage attached to each of them
  - Only link is via a LAN
  - Shared nothing refers to no sharing of RAM, storage
  - Note: Network Attached Storage (NAS) is common now, “pure” shared-nothing rarer



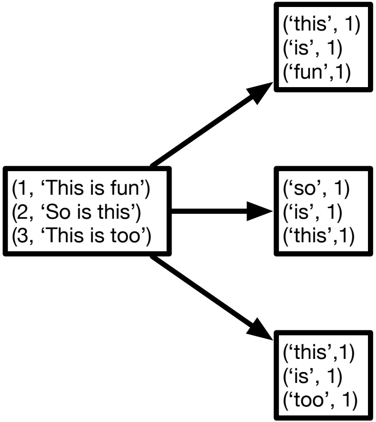
Benefits of Shared-Nothing

- Inexpensive, built out of commodity components
- Compute resources scales nearly linearly with money
- Contrast to shared RAM machine with uniform memory access
- Easier to program than Shared Memory systems

MapReduce: The Map Phase

- Input data are stored in a huge file
  - Contains a simple list of pairs of type $(key1, value1)$
- Have a UDF (user defined function) of the form $Map(key1, value1)$
  - Outputs a list of pairs of the form $(key2, value2)$
- During the Map phase of the MapReduce computation
  - The Map function is called for every record in the input
  - Instances of Map run in parallel all over the cluster

Task: Compute the number of occurrences of each token

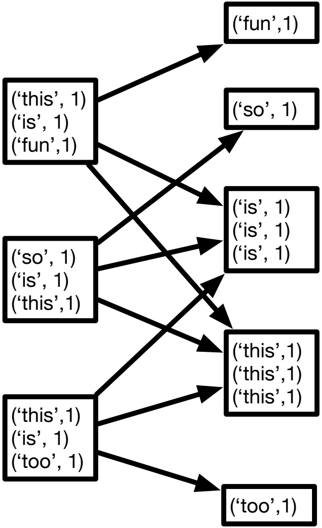


Example: Word Count

- Large text corpus
- Want to count number of occurences of each word
- Ex output: ('The', 1832321), ('An', 1732432), etc.
- To power the Map phase, MapReduce software automatically:
  - Breaks the corpus into large number of $(lineNo, text)$ pairs
  - Distributes the Map UDF
  - Distributes the fragments
  - Ensures that the UDF is run on all the fragments

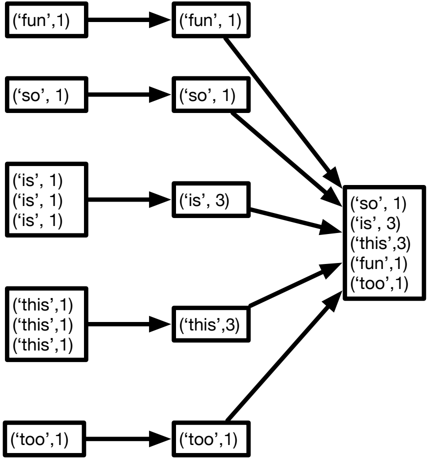
MapReduce: The Shuffle Phase

- Accepts all of the $(key2, value2)$ pairs from the Map phase
  - And it groups them together
- After grouping, all of the pairs
  - From all over the cluster having the same $key2$ value
  - Are merged into a single $(key2, itemize\langle value2\rangle)$ pair
- Called a “Shuffle”...
  - Because this is where a potential all-to-all data transfer happens
  - Can be expensive
  - Often implemented using a hash function to map keys to nodes transfer happens

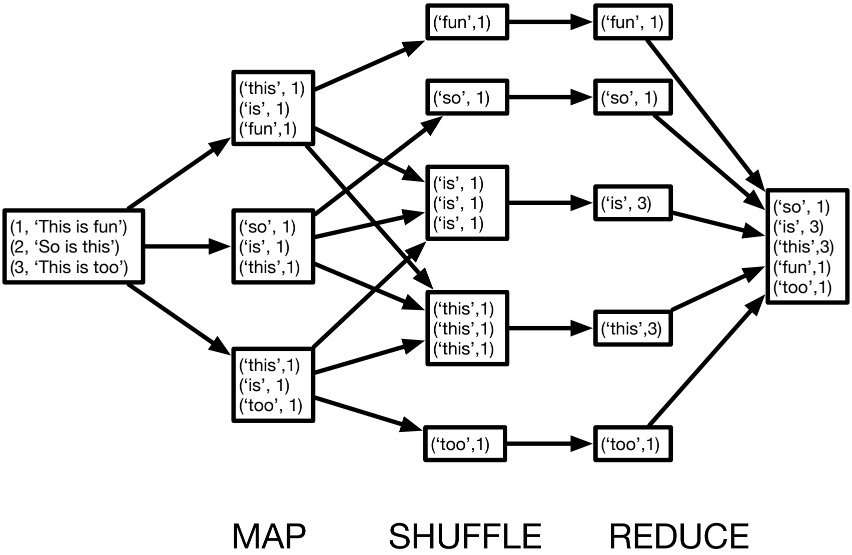


MapReduce: The Reduce Phase

- Have a user-supplied function of the form
  - $Reduce(key2, itemize\langle value2 \rangle)$
  - Outputs a list of $value3$ objects
- In the Reduce phase of the MapReduce computation
  - $Reduce$ function is called for every $key2$ value output by the Shuffle
  - Instances of $Reduce$ run in parallel all over the compute cluster
  - The output of all of those instances is collected
  - Put in a (potentially) huge output file



MapReduce: All Phases



MapReduce is a Compute Paradigm

- It is not a data storage paradigm
  - But must read/write data from some storage system
- So MapReduce is strongly linked with the idea of a distributed file system (DFS)
  - Allows data to be stored/accessed across machines in a network
  - Abstracts away differences between local and remote data
  - Uses the same API to read/write data
  - Regardless of network location

Distributed File Systems for MR

- DFSs have been around for a long time
  - First widely used DFS was Sun's NFS, first introduced in 1985
  - Intended to be allow file access across networks as if file was local
- Unlike classical DFS...
  - MapReduce DFS sits on top of each machine's OS
  - Lives in "user space"
  - The OS is not aware of the DFS
  - You can't mount it anywhere
- Why on top of, not in the OS?
  - Heterogeneity no problem (each machine can run a different OS)
  - Easily portable (JVM)
- Even as MapReduce becomes less popular, MR DFS lives on!

MapReduce vs. HPC

- MapReduce pros
  - MUCH lower programmer burden than HPC
  - No synchronization, all parallelism is implicit
  - Data and task placement automatic
  - Built-in fault tolerance
  - Works with (almost!) arbitrarily-sized data
  - Out-of-core execution is no problem

MapReduce vs. HPC

- MapReduce cons
  - Standard softwares are JVM-based
  - Not suitable for communication-heavy tasks...
  - ...only communication is via the shuffle
  - Assumes BIG data... always reads/writes data from DFS

Wrap up

- How can we use what we learned today?
- What do we know now that we didn't know before?