ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

APPLIED DATA ANALYSIS

School of Computer and Communication Sciences

Applied Data Analysis
Lecture Notes

Dr. Catasta Michele
Distributed Information Systems Laboratory (LSIR)
michele.catasta@epfl.ch

November, 2016

Contents

Scaling Up

Assumptions since now:

- All data fits on a single server
- All data fits in memory (pandas KAWAI!!)

It's a realistic assumption for **prototyping**, but not when moving to production.

Big Data Problem

Data is growing faster than computation speeds. CPU speed is stalling and there is a bottleneck in Storage. Therefore a single machine can not longer process or even store all the data. The only solution is to **ditribute** data over large clusters.
CONTINUE INTRO

How much data do you need?

THe answer depends on the question. But for many applications the answer is: **as much as we can get**. Big Data about people (text, web, social media, etc.) follow power law statistics. Most of the features occur only once or twice. Long tail is really important. Google knows that you're looking for something even if you checked on the web once or twice.
The number of features grows in proportion to the amount of data, *i.e.* doubling the dataset size roughly doubles the number of users we observe. Even one or two observations of a user improves predictions for them, so:

More data and bigger models => More revenue!

Hardware for Big Data

At the beginning, Google (and other Internet companies) used many low-end servers instead of expensive high-end servers. It is easier to add capacity and it is cheaper per CPU/disk. But there are problems:

- **Failures** (e.g. Google numbers)
  - 1-5% hard drives/year
  - 0.2% DIMMs/year (RAM)
- **Commodity Network** (1-20 Gb/s) speed vs RAM
  - Much more latency (100x – 100000x)
  - Lower throughput (100x-1000x)
- **Uneven Performance**
  - Inconsistent hardware skew
  - Variable network latency
  - External loads

These numbers are constantly changing thanks to new technology!

How do we split work across machines?

First example: *How do you count the number of occurences of each word in a document?* We use a hash table!
And if the document is really big? For example, the web. We can simply chunk the document in multiple documents and create a hash table on each machines. At the end, we aggregate all the hash tables. **But** the machine that aggregates is the bottleneck in term of performance. And if it crashes, it is even worse. Therefore, we can use **Divide-and-Conquer**. Each node is responsible of summing up a subset of the whole hash table. This is call **Map Reduce**. This works well when using a lot of different nodes.
ADD ALL THE IMAGES!!!
Each task is idem-potent. => The order of word count doesn't matter. Therefore, really easy to use without waiting the other nodes to finish.

What's hard about cluster computing?

How to divide work across machines?

- Must copnsider network, data locality (Always better on its own machine)
- Moving data may be **very** expensive

How to deal with failures?

- 1 server fails every 3 years => 10K nodes see 10 faults/day
- Even worse: stragglers. Node has not failed but is slow.

How to deal with failures? **Just launch another task!**
Large Parallel Database did not support fault tolerance. Therefore, if the query fails, you have to redo it since the beginning. If the query takes longer than 3 hours => we will experience one failure on a server during the query => query is useles. =(
How to deal with slow tasks? **Just launch another task!**
Map-Reduce is noe used by Google anymore (even if they invented it). Something more advanced exists. Why?
Because Memory is really cheap! RAM 1 cent/Mb.

Spark Computing Framework

Spark provides a programming abstraction and parallel runtime to hide this complexity. Spark does all the complex cluster computing stuff! How are Spark and MapReduce Different?
TABLE!!
Spark tries to do as much as possible in RAM. If it can't, it will use memory on disk. Therefore, it's way more faster than simple Map Reduce!
Persist data **in-memory**:

- Optimized for batch, data-parallel ML algorithms
- An efficient, general-purpose language for cluster processing of big data
- In-memory query processing (Spark SQL)

Practical Challenges with Hadoop:

- Very **low-level** programming model
- Very **little re-use** of Map-Reduce code between applications
- **Laborious** programming: design code, build jar, deploy on cluster
- **Relies heavily on Java reflection** to communicate with to-be-defined application code.

Practical Advantages of Spark:

- **High-level programming model**: can be used like SQL (Dataframe) or like a tuple store.
- **Interactivity**
- **Integrated UDFs** (User-Defined Functions)
- High-level model (**Scala Actors**) for distributed programming
- **Scala generics** instead of reflection: Spark code is generic over [Key, Value] types.

Fault Tolerance
Hadoop: Once computed, don't lose it (data replication on HDFS)
Spark: Remember **how** to recompute

Older Spark programming model

We use RDD (Resilient Distributed Dataset)

- Distributed array, n partitions
- Elements are lines of input
- Computed **on demand**
- Computer = (re)read from input

You can do multiple cool things (ADD CODE + PICTURES):

- Count the lines
- Count the lines and comments
- Cache stuff
- filter
- map
- flatMap
- Shuffle transformations
  - groupByKey
  - sortByKey
- Getting data out of RDDs

  - reduce
  - take
  - collect
  - saveAsTextFile

Spark DataFrames

Bringing the gap between your experience with Pandas and the need for distributed computing.
Important to understand what RDDs are and what they offer, but today most of the tasks can be accomplished with DataFrames (**higher lever of abstraction => less code**)

Spark's Machine Learning Toolkit

MLLib:

- Classification
  - Logistic Regression
  - Decision Trees
  - Random Forests
- Regression
  - Linear (with L1 or L2 regularization)
- Unsupervised
  - Alternating Least Squares
  - K-Means
  - SVD
- Optimizers
  - Optimization primitives (SGD, L-BGFS)

Spark Driver and Executors

- Driver runs user interaction, acts as master for batch jobs
- Driver hosts machine learning models
- Executors hold data aprtitions
- Tasks process data blocks
- Typically tasks/executors = number of hardware threads

Architectural Consequences

- **Simple programming**: Centralized model on driver, broadcast to other nodes
- Models must fit in single-machine memory, *i.e.* Spark supports **data parallelisme** but not **model parallelism**

- Heavy load on the driver. Model **update time grows** with number of nodes.

Other uses for MapReduce/Spark:
Non-ML applications
Data processing:

- Select columns
- Map functions over datasets
- Joins
- GroupBy and Aggregates
- ETL jobs

Other notable Spark tools:

- BlinkDB (approximate statistical queries)
- Graph operations (GraphX)
- Stream processing (Spark streaming)
- KeystoneML (Data Pipelines)