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

1 Scaling Up

For the moment, we always made one big assumption: **All data fits on a single server**. Even more, all data fits in the memory. It's a realistic assumption for *prototyping.* But it becomes totally obsolete when moving to production. In this chapter, we will introduce the Big Data Problem as well as Spark.

1.1 Big Data Problem

Nowadays, Internet is the biggest source of data. And it is growing faster and faster. Even if computers are really powerful, Data Scientist encounters one problem: The Big Data Problem. Indeed, **Data is growing faster than computation speeds**.
We can see, for example, that CPU speed is stalling and that there is a bottleneck in storage. Therefore a single machine can no longer process or even store all the data. The only solution is to **ditribute** data over large clusters.

1.1.1 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 the power law statistics. Indeed, most of the features occur only once or twice. Therefore the distribution is a long tail distribution. And the long tail is really important. For example, 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. Therefore, even one or two observations of a user improves predictions for them, so:

More data and bigger models => More revenue!

1.1.2 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!

1.1.3 How do we split work across machines? (Map-Reduce)

How do you count the number of occurences of each word in a document?

We can use a **hash table**. It's an easy tool to use for these kind of task. But what happens if the document is really big? (For example, the web) We can simple **chunk** the document into 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 this machine crashes, it's a disaster!

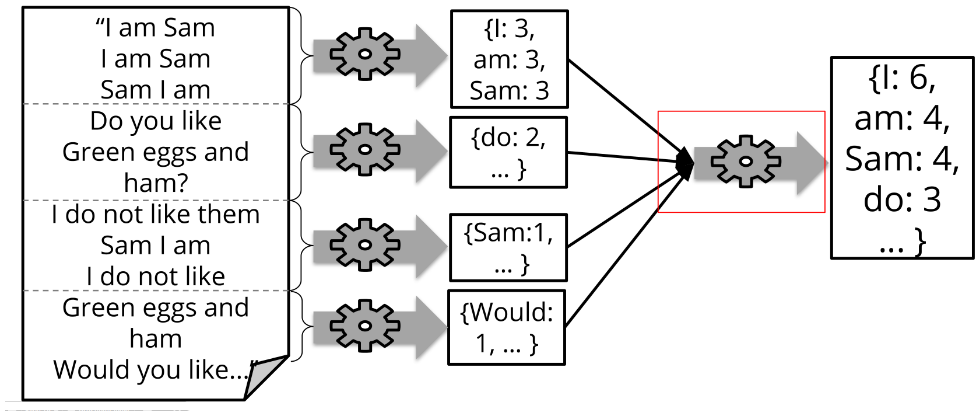

Figure 1.1: Usage of hash tables on multiple machines. We see that the bottleneck machine is the machine that aggregates the hash tables into one.

Therefore, we can use a more clever algorithm: **Divide-and-Conquer**. Each node is responsible os summing up a subset of the whole hash table.

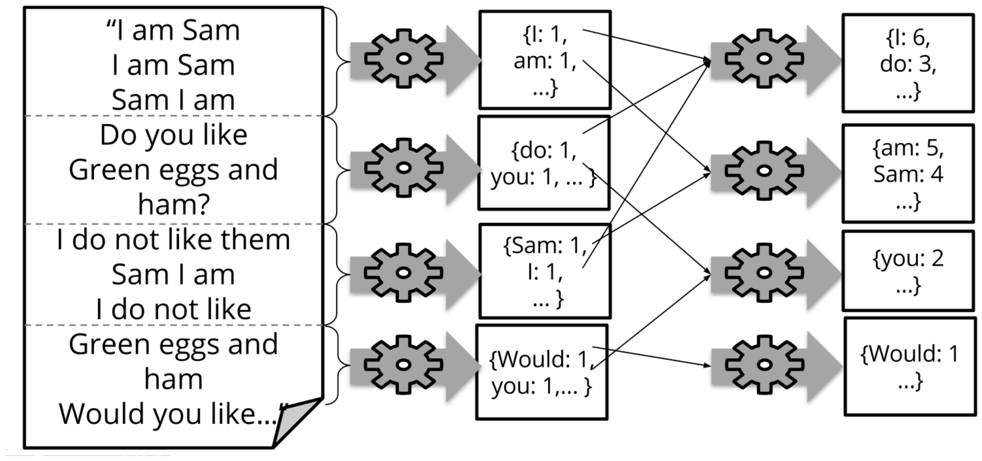

Figure 1.2: Usage of hash tables on multiple machines with Divide-and-Conquer technique. We don't have any bottleneck this time.

This technique is called **Map Reduce**. The first part is called map because we create a map for each part of the big document. Then we reduce it into a simpler hash table. This works well when using a lot of different nodes. In addition, each task is **idem-potent**. It means that the order of the word count doesn't matter. Thus, it is really easy to user without waiting the other nodes to finish.

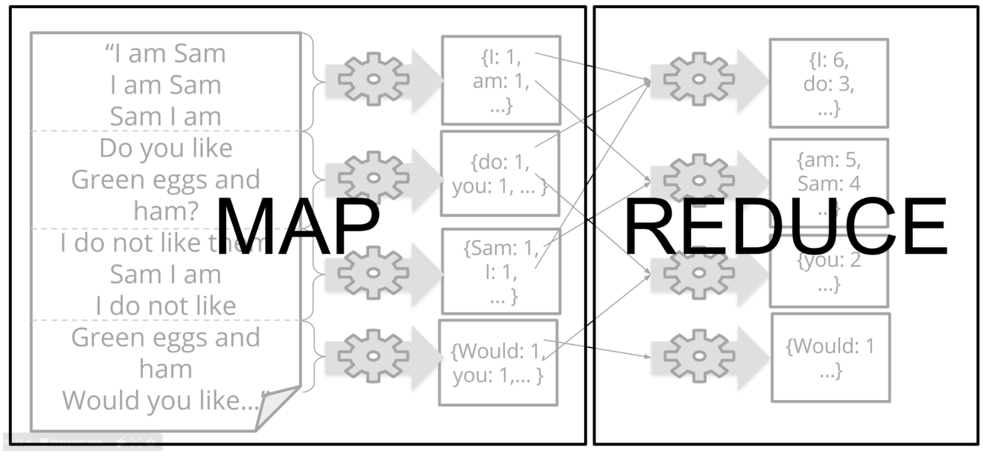

Figure 1.3: Map Reduce illustrated on the Divide-and-Conquer algorithm in Figure 1.2

1.1.3.1 What's hard about cluster computing?

How to divide work across machines?

- We Must consider network, data locality. (It's always better on its own machine.)
- Moving data may be **very** expensive!

How to deal with failures?

- 1 server fails every 3 years. Therefore, 10K nodes will give 10 faults/day.
- Even worse: stragglers. The node has not failed but is slow.

How to deal with failures? **Just launch another task!**

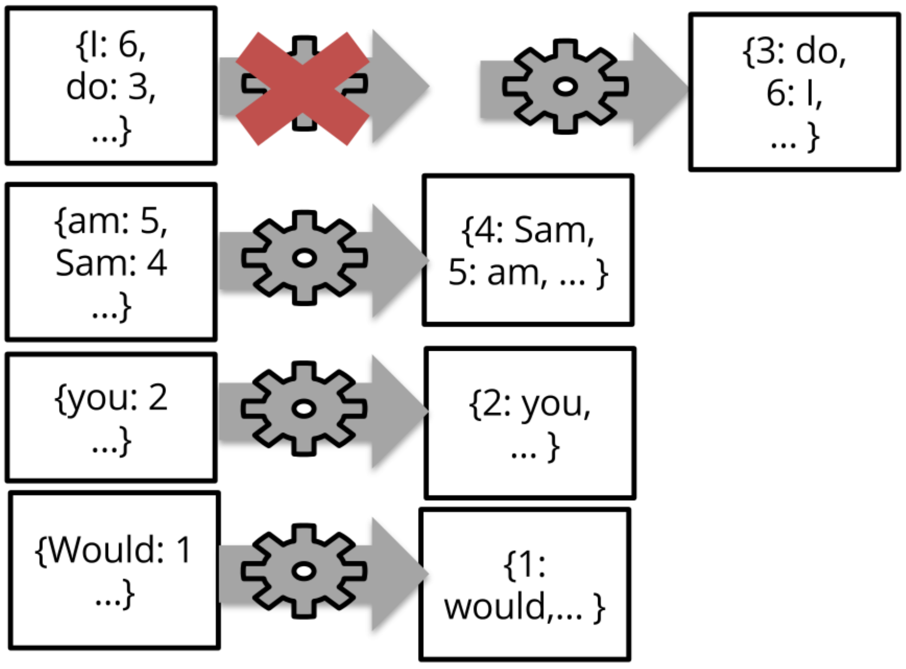

Figure 1.4: To deal with failure, we can just launch another task.

Large Parallel Databases did not support fault tolerance until recently. Therefore, if the query fails you have to redo it since the beginning. For example, if the query takes longer than 3 hours you will experience one failure on a server during the query. Thus, the query is useless.
How to deal with slow tasks? **Just launch another task!**

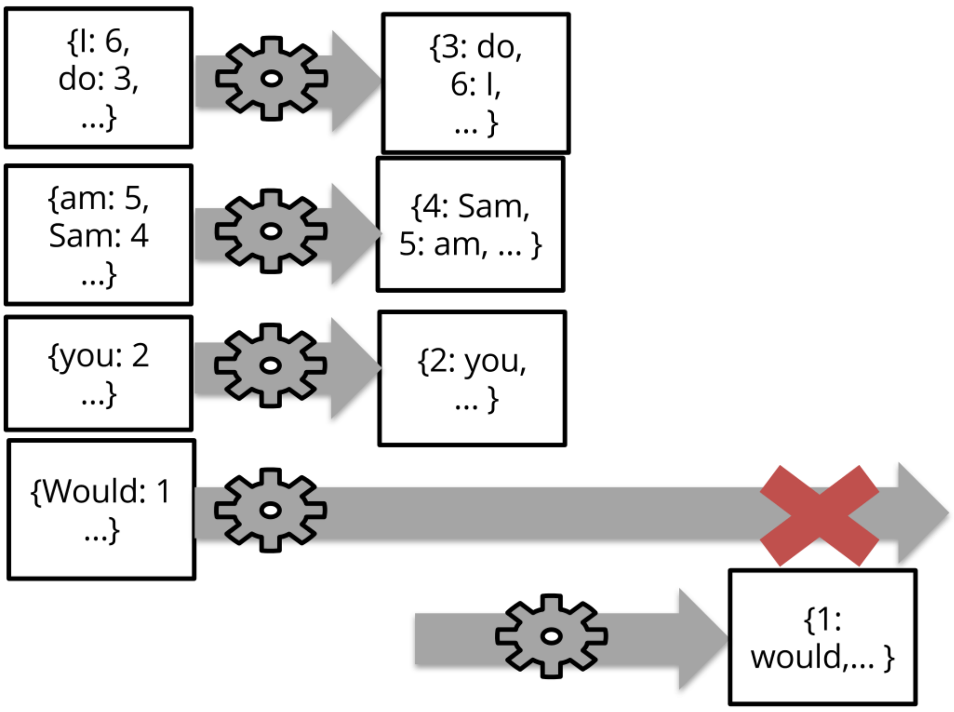

Figure 1.5: To deal with slow tasks, we can just launch another task.

Map-Reduce is not used by Google anymore, even if they invented it. More advance techniques exist nowadays. Why? Because Memory is really cheap! (RAM: 1 cent/Mb)

1.2 Spark Computing Framework

Spark provides a programming abstraction and parallel runtime to hide all of these complexity. Table 1 shows the main differences between Spark and Hadoop Map-Reduce. The big advantage with Spark is that it deals itself with all the complex clustering between the nodes.

	Hadoop Map Reduce	Apache Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc.
Ease of use	Java program	Scala and Python shells

Table 1: Main differences between Spark and Hadoop

One of the big advantage of Spark is that it uses in-memory storage. This can speed-up the computation time by a factor of 100 on the Logistic Regression for example. The good thing with Spark is that if the RAM is full, then it will use the memory on disk. We can see some interesting fact about having 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)

Spark is not only good because it's faster than Hadoop. For example, we can see that some practical Challenges with Hadoop are:

- 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.

And some practical Advantages of Spark are:

- **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.

One last advantage of Spark is the way it deal with **Fault tolerance**. Hadoop uses data replication on HDFS. Therefore, once it's computed, you should not loose it. On the other hand, Spark remember **how** to recompute everything. For example, it will remember which nodes failed and which road he used to compute everything.

1.2.1 Read more

If you want to read more on Haddop, you can go on Apache Hadoop website. For Spark, you can go on Aparche Spark website. A good article comparing Spark and Hadoop can be found on Xplenty. You can always find more information about Spark and Hadoop on Internet. But the thing you should know is that even if Spark is much faster and much easier to use, Hadoop is not ready yet for the *elephant's graveyard*.

2 Handling Text