MapReduce and Big Data: an overview

Tobia Tesan

Abstract

In this document we discuss the MapReduce paradigm and its relationship with Big Data.

In Section 1 we give a general idea of what Big Data is and discuss its challenges and potential; in particular, in subsection 1.2 we give reasons why IoT should be considered closely related to Big Data.

In Section 2 we introduce the MapReduce programming paradigm and discuss its applicability, advantages and limitations; particularly, in 2.3.1 we highlight its relationship with parallel relational databases.

Finally, in Section 3, we briefly discuss the Hadoop implementation of MapReduce and give an example of a MapReduce program.

Contents

List of Figures

List of Listings

1 Big Data

Ronald Fisher is famously quoted as stating, in his Presidential Address to the First Indian Statistical Congress, that "to consult the statistician after an experiment is finished is often merely to ask him to conduct a post mortem examination", as he can at best "say what the experiment died of".

Yet the premise of big data is that valuable information can be extracted from data which is not the result of a carefully designed experiment – and it's the "bigness" of the data that makes this approach at once possible, necessary and challenging.

Big Data is a naturally emerging byproduct of digital interaction and not the result of a purposeful experiment – web server logs, clinical databases and in general large corpi (or corpuses) of text can be examples of big data.

As such, another key characteristic of big data is its being only partially structured.

It can also be characterized [LJ12] as noisy, dynamic, heterogeneous, interrelated, untrustworthy – and, of course, big in terms of sheer size.

1.1 The challenges of Big Data: Volume, Variety and Velocity

Famously, [GP11] summarizes the challenges of big data as "volume, variety and velocity".

It's indeed its size, or volume, that makes it valuable, potentially more than a true statistical sample of small size: as [LJ12] put it, "general statistics obtained from frequent patterns and correlation analysis usually overpower individual fluctuations and often disclose more reliable hidden patterns and knowledge".

Yet, its volume is a challenge – more precisely, its *increasing* volume in the face of non-increasing compute resources at the single core level [LJ12].

It's worth noting that the "end of Moore's law" is happening at the same time as the shift towards cloud storage and the third, simultaneus shift towards flash storage.

This requires that algorithms and computing models used to process big data be able to exploit parallelism and are fit to be deployed in an heterogeneous environment characterized by multi-tenancy and where elastic scalability is a requirement – this privileges high level, declarative approaches that lend themselves well to automatic balancing and optimization.

The variety of big data – i.e. heterogeneity of formats and potential incompleteness of records – makes designing appropriate algorithms harder than in the presence of well-structured, regular data.

Finally, its velocity requires a high acquisition rate and appropriate indexing techniques, so that the data can be queried in real time even when it is continuosly growing.

Filtering, indexing and pre-processing at the acquisition stage can help dealing with incompleteness and velocity, but engineers must find ways to do so while guaranteeing lack of information loss; the danger is real as, after all, the notion of aggregate data is fundamentally at odds with the one, given above, of big data.

1.2 Big Data and the Internet of Things

The nature of big data and IoT makes them ideal partners in a symbiotic relationship – one that is already flourishing.

IoT applications generate precisely the kind of digital data we have described in the preceding section, but on an unprecedented scale, as devices are ubiquitous and applications are potentially endless: data is no longer just a byproduct of "online" interaction and is instead continuously generated as things happen *in the physical world*: we could go as far as saying that IoT generates the quintessential form of big data.

To give an idea of the relevance of the phenomenon, 12 million RFID tags (used to capture data and track movement of objects in the physical world) were sold in 2011 [Dul] and Gartner estimates more than 8 billion IoT devices will be in use in 2017 [vdM17].

A consequence of this is an unprecedented potential to query data and uncover relationships not only in the digital domain, but among "real life" phenomena as well.

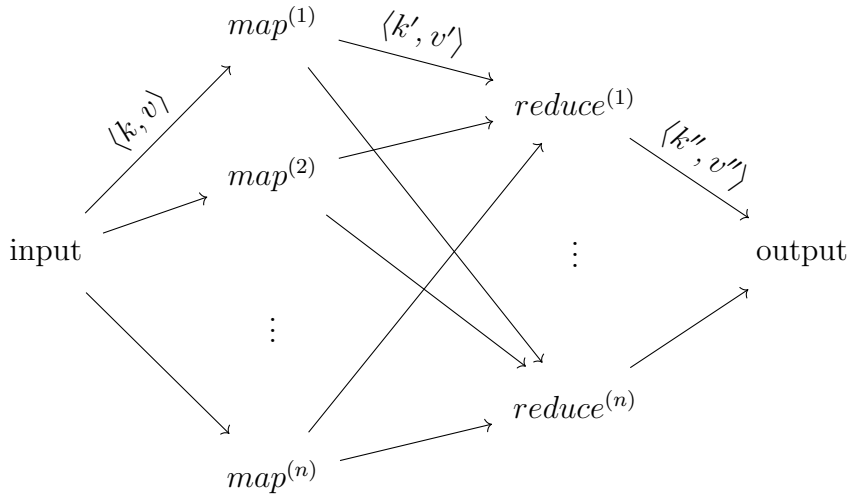

Figure 1: Basic flow of a MapReduce computation

2 MapReduce

MapReduce is a programming model that, for reasons that will soon be apparent, is extremely popular to process big data: its most popular implementations are by far Apache Hadoop, Disco and Spark, along – in terms of services powered and, likely, sheer number of nodes deployed – with the one internally used by Google.

MapReduce did in fact come to prominence in the late 2000s in the wake of the work of Dean and Ghemaway at Google [DG08]; the authors' purpose was to devise a mechanism to easily process and generate large data sets on commodity hardware for the company's internal needs; the main contribution of MapReduce was a simple interface in a functional programming style – inspired, in fact, by `map` and `reduce` primitives in LISP – that could be automatically parallelized.

The rationale that motivated the authors was the observation that the company's workloads were conceptually simple computations (often no more complex than an elementary SQL query) made cumbersome by the need to handle large input data, parallelize and distribute the computation.

With MapReduce, the run-time system takes care of partitioning input data, scheduling execution, handling failures and managing communication.

The MapReduce paradigm fits naturally well with distributed file systems, and is in fact often cited in conjunction with the work on the Google File System by the same authors [GGL03].

2.1 How MapReduce works

A MapReduce program is defined in terms of two functions `map` : $\langle k, v \rangle \to \langle k', v' \rangle$ and `reduce` : $\langle k', v' \rangle \to \langle k'', v'' \rangle$.

The MapReduce process is initated by a master node.

The input data, stored on a distributed filesystem, is first partitioned into m `m-splits` for processing by m map instances.

The master node maps each `m-split` to m map instances $map^{(1...m)}$ that yield m sets of $\langle k', v' \rangle$ pairs, each with arbitrary cardinality; the output is buffered to a local disk upon completion.

Each set is then mapped on n reduce instances $reduce^{(1...n)}$, which the master node initiates on worker nodes.

The reduce worker first reads *all* the intermediate data, it sorts the data and iteratively applies `reduce` : $\langle k', v' \rangle \rightarrow \langle k'', v'' \rangle$ over every sorted $\langle k', v' \rangle$ pair.

Finally, the output from the various reduce jobs is written to a single output file.

This computation flow is summed up in Figure 1.

Its worth noting that the algorithm described above does not require defining a schema for the input data, which can partitioned, for example, on a per-line basis, and is thus well-suited to the kind of semi-structured big data that we've discussed in the preceding section.

2.2 Benefits of MapReduce

MapReduce operates well in environments where bandwidth is a relatively scarce resource, as scheduling can be optimized for locality, i.e. the master node can try to initiate each worker instance on a node containing – or in the proximitiy of a node containing – a replica of the input data.

MapReduce offers a high level of resiliency to large-scale worker failures: upon failure of a node that causes loss of the intermediate `map` output or failure to carry a `reduce` job to completion, additional copies of the computations can be rescheduled on surviving nodes.

The loss of a master node is handled in the original paper through the "ostrich algorithm" – i.e. the possibility is ignored due to its improbability [DG08].

However, an implementation of MapReduce can easily include some form of backup or redundancy for the master node.

A distributed, non-faulting execution of a MapReduce program yields the same output as a local, sequential execution of the composition of the `map` and `reduce` function, which makes it easy to reason about Mapreduce programs, debug and test them locally.

2.3 Limitations of MapReduce

In the previous section we have detailed how declarative, high level approaches are to be favoured for deployment in the cloud; moreover, high level primitives are also essential for composing and building complex anaytical pipelines [LJ12].

MapReduce is still a relatively low level idiom that requires developers to write custom programs which – while devoid of the complexity of handling distribution and file access manually – are hard to maintain and reuse [TSJ+09].

In this respect high level languages like Hive or Pig Latin, used by Apache Pig, appear very promising: Pig Latin is a higher level procedural language, whereas Hive provides a SQL-like query language called HiveQL which supports many familiar forms of SQL statements, including `SELECT`, `PROJECT`, `JOIN`, `AGGREGATE`, `UNION` as well as sub-queries in the `FROM` clause.

	RDBMS	MapReduce
Volume	Gigabytes	Petabytes
Mode of operation	Interactive and batch	Batch
Mode of access	R/W-many	WORM
Transaction	ACID	No transaction support

Table 1: Comparison of RDBMSs and MapReduce, from [Whi12]

Another potentially significant drawback is that deploying MapReduce on commodity nodes used for processing *and* storage has potentially serious consequences on the energy efficiency of MapReduce clusters.

Leverich and Kozyrakis [LK10] have highlighted how the filesystem component of MapReduce frameworks effectively precludes scale-down of clusters, as even idle nodes remain powered on all the time to ensure data availablility.

The authors have experimented modifications to the Hadoop scheduler and data layout that enabled them to achieve up to 50% energy savings in trade for performance, and suggested that an energy-efficient Hadoop cluster should contain a dynamic power controller which is able to intelligently optimize usage in order to assign and guarantee different service level agreements for different jobs.

2.3.1 MapReduce and parallel DBMSs

MapReduce finds itself competing in areas that, until the early 2000s, were the domain of traditional (i.e. SQL) parallel DBMSs.

However, MapReduce and SQL-based products are best considered complementary technologies suited for complementary classes of problems [SAD+10].

MapReduce is a fundamentally batch paradigm [Whi12], best suited to implement an extract-transform-load system for enormous quantities of unstructured or semistructured data rather than for fast querying of data.

Besides the edge that MapReduce has in handling semistructured data – which could be, for example, represented in traditional database systems with a wide table with many nullable attributes to accommodate multiple record types [SAD+10] – not having to define a schema makes MapReduced better suited for one-off complex data transformations.

PDBMSs also exhibit significantly higher TOC and setup costs that make MapReduce-based systems attractive [SAD+10] for users with limited budgets.

Stonebraker [SAD+10] however notes that PDBMSs are still the appropriate choice for query-intensive, interactive applications.

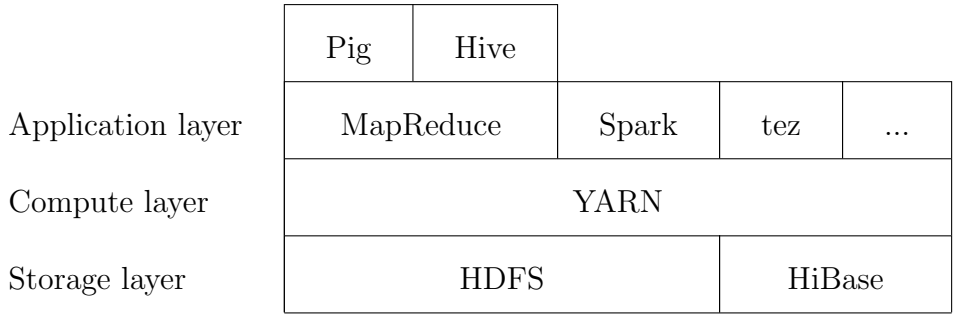

Figure 2: The YARN stack of technologies (from [Whi12])

3 Hadoop

Hadoop is an extremely popular implementation of MapReduce that was originally implemented as a component of Apache Nutch, directly inspired by the GFS and MapReduce papers soon after they were published by Google engineers; Hadoop was spun off Apache Nutch and became its own project in 2006 [Whi12].

Since its beginnings, Hadoop has incorporated a MapReduce implementation and a distributed filesystem called HDFS [Whi12];

Moreover, since its second version, Hadoop incorporates Apache YARN for cluster management; YARN provides low-level APIs for requesting and working with cluster resources.

In turn, these APIs are leveraged by higher level distributed computing frameworks – such as MapReduce, Spark... – that run as YARN applications on top of the YARN cluster [Whi12]; the MapReduce implementation running on YARN in Hadoop $\geq 2.x$ mantains API compatibility with so-called Mapreduce 1 API.

Three layers – Application Layer, Cluster Compute Layer and Storage Layer – are thus determined, visible in Figure 2.

The participants in a YARN compute cluster are the Resource Manager that exposes the computing resources to clients and coordinates several Node Managers, one per cluster node; several containerized application processes are run under each Node Manager, as illustrated in Figure 3.

Application processes are not necessarily symmetrical nor independent from each other – for example, in MapReduce jobs, there is a so-called "application master" process, which has coordination responsibilities towards other processes in the same job.

3.1 An example MapReduce program: education in Italy

We will now give an example of a MapReduce program using Hadoop.

Let us note that the data we will manipulate are indeed distant from the definition of "big data" and the operations we will carry out on them are equivalent to a one-line SQL query – however, they shall make for a relatively straightfoward example.

More complex programs that process *actual* big data will not deviate from the paradigm that the example serves to illustrate – since, naturally, they will be subject to the same flow of control imposed by the framework – but may include more complex logic in their `reduce` and `map` functions.

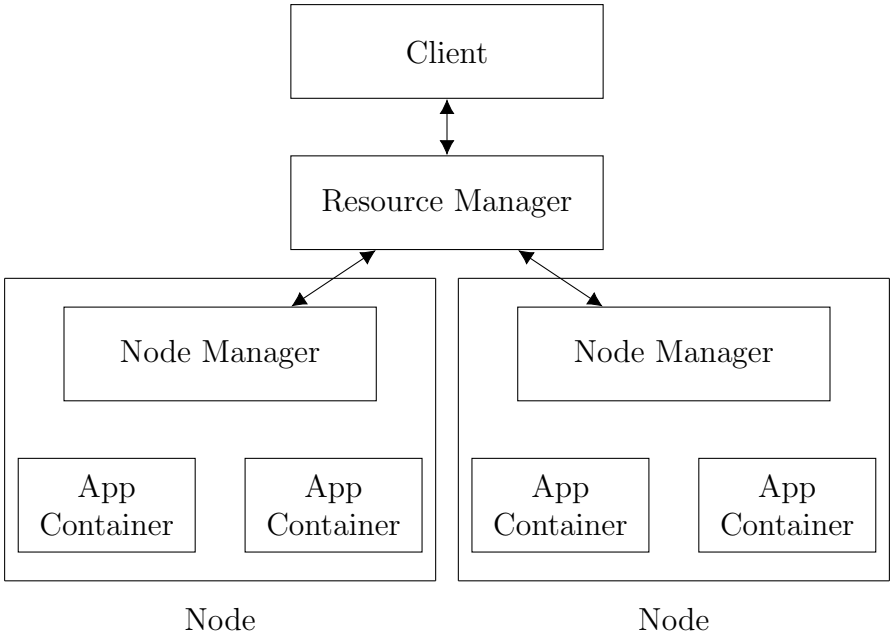

Figure 3: Architecture of a multi-node Hadoop cluster

We have retrieved from the i.Stat datawarehouse [IST], made available by ISTAT, a 112.7MB dataset containing the census of individuals per type and level of degree obtained[1], per gender, per age group in each geographical region.

We will extract the total, nation-wide number of individuals for each type of degree and rank the various types of degrees by numerosity, with the aim of gaining some insight on what are the most represented specializations[2].

The dataset is in CSV format, uncommonly using a pipe (|) as separator. A snippet of the dataset is visible in Figure 4.

We will use, in fact, two jobs for a total of 2 mappers and 2 reducers: one job (Listings 2 and 3) counts the number of individuals per degree, using the degree as output key[3], while the second one (Listings 4 and 5) ranks degrees by popularity.

The jobs are chained together rather crudely in the main method (Listing 1) – the first job writes its output to a temporary output directory which is then used as input for the second one.

A more sophisticated way of daisy-chaining jobs is offered by JobControl; alternately, Apache Oozie can be used, which runs as a service in the cluster [Whi12].

In Figure 5 we can see the resulting output, trimmed for length.

From it we can take away the reassuring notion that telecommunications engineers outnumber specialists in subtropical agriculture.

[1]Only the highest degree obtained by each individual is counted in the census

[2]In practice, we will find that the results are inconclusive due to irregular input data – particularly, in some regions large numbers of individuals are simply reported to have a "Master's Degree" or "Laurea" with no mention of the field.

[3]Note that the degree is the 13th field of a record, whereas the number of recorded individuals holding that degree for the given region/age/gender group is the 18th.

```java
public class RankDegrees {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: RankDegrees <input path> <output path>");
            System.exit(-1);
        }

        Job job = new Job();

        job.setJarByClass(RankDegrees.class);
        job.setJobName("Compute total count for each degree");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        java.nio.file.Path tempnio = Files.createTempDirectory("top");
        Path temp = new Path(tempnio.toString() + "/output");
        FileOutputFormat.setOutputPath(job, temp);

        job.setMapperClass(CountMapper.class);
        job.setReducerClass(CountReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        Boolean status = job.waitForCompletion(true);

        // This is a rather crude way of chaining jobs
        if (status) {
            Job job2 = new Job();

            job2.setJarByClass(RankDegrees.class);
            job2.setJobName("Rank degrees by total count");

            FileInputFormat.addInputPath(job2, temp);
            FileOutputFormat.setOutputPath(job2, new Path(args[1]));

            job2.setMapperClass(RankMapper.class);
            job2.setReducerClass(RankReducer.class);

            job2.setMapOutputKeyClass(IntWritable.class);
            job2.setMapOutputValueClass(Text.class);

            job2.setOutputKeyClass(Text.class);
            job2.setOutputValueClass(IntWritable.class);
            System.exit(job2.waitForCompletion(true) ? 0 : 1);
        } else {
            System.exit(1);
        }
    }
}
```

Listing 1: RankDegrees class with main method

```
public class CountMapper
        extends
            Mapper<LongWritable, Text, Text, IntWritable> {
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {

        if (key.get() > 0) {
            String line = value.toString();
            String[] parts = line.split(Pattern.quote("|"));
            if (parts.length >= 18) {
                String degree = parts[13].trim();
                int number = Integer.parseInt(parts[18].trim());
                if (!degree.equals("\"total\""))
                    context.write(new Text(degree), new IntWritable(↩
    number));
            }
        }
    }
}
```

Listing 2: CountMapper class

```
public class CountReducer
        extends
            Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
            Context context)
            throws IOException, InterruptedException {
        int total = 0;
        for (IntWritable value : values) {
            total = total + value.get();
        }
        context.write(key, new IntWritable(total));
    }
}
```

Listing 3: CountReducer class

```
public class RankMapper
        extends
            Mapper<LongWritable, Text, IntWritable, Text> {
    @Override
    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        context.write(new IntWritable(1), value);
    }
}
```

Listing 4: RankMapper class

```
public class RankReducer
        extends
            Reducer<IntWritable, Text, Text, IntWritable> {
    @Override
    public void reduce(IntWritable key, Iterable<Text> values,
            Context context)
            throws IOException, InterruptedException {

        class Pair {
            public String _s;
            public int _n;
            Pair(String s, int n) {
                _s = s;
                _n = n;
            }
        }

        TreeMap<Integer, Pair> map = new TreeMap<Integer, Pair>();

        for (Text value : values) {
            String line = value.toString();
            String[] parts = line.split(Pattern.quote("\t"));
            if (parts.length >= 2) {
                Integer number = Integer.parseInt(parts[1].trim());
                String stuff = parts[0].trim();
                map.put(number, new Pair(stuff, number));
            }
        }

        for (Pair pair : map.descendingMap().values()) {
            context.write(new Text(pair._s), new IntWritable(pair._n));
        }
    }
}
```

Listing 5: RankReducer class

```
"ITTER107"|"Territory"|...|"Gender"|...|"Certificate or degree obtained"|...|Value|

"ITG"|"Isole"|...|"males"|...|"diploma of vocational institute for marketing, tourism and advertising"    \
                                                                                                |...|12235|
"ITG"|"Isole"|...|"males"|...|"enterostomal therapy for professional nurses"|...|395|
"ITG"|"Isole"|...|"males"|...|"laboratory technician of agricultural and food industries"|...|16|
"ITG"|"Isole"|...|"males"|...|"sociopolitical field"|...|525|
"ITG"|"Isole"|...|"males"|...|"other degrees of the sports field"|...||1321|
"ITG"|"Isole"|...|"males"|...|"herbal sciences"|...|103|
"ITG"|"Isole"|...|"males"|...|"environmental sciences"|...|54|
"ITG"|"Isole"|...|"males"|...|"occupational therapy (qualifying course for occupational therapist health c\
                                                                                    are profession)"|...|72|
"ITG"|"Isole"|...|"males"|...|"electrical engineering"|...|280|
"ITG"|"Isole"|...|"males"|...|"agricultural field"|...||1245|
"ITG"|"Isole"|...|"males"|...|"business and economics tele-teacher"|...|56|
"ITG"|"Isole"|...|"males"|...|"diploma of vocational institute for trade and industry"|...|49150|
"ITG"|"Isole"|...|"males"|...|"science of legal services for employment"|...|45|
"ITG"|"Isole"|...|"females"|...|"degree in interpretation and translation advanced school"|...|23|
"ITG"|"Isole"|...|"females"|...|"academic degree of school of fine arts (afam first level)"|...|441|
"ITG"|"Isole"|...|"females"|...|"environmental sciences"|...|16|
"ITG"|"Isole"|...|"total"|...|"total"|...|3033984|
```

Figure 4: Snippet of input file, with fields removed

```
"diploma of  lower secondary education"                        201610085
"diploma of upper secondary education (4-5 years)"             167711824
"primary school certificate"                                   136087601
"diploma of technical institute"                                83185853
"no formal education"                                           59288383
.
.
"telecommunications,  electronics and computer engineering"     1490593
.
.
"academic degree of national academy of dramatic art [...]"        3825
"industrial relations"                                             3770
"tropical/subtropical agricultural sciences"                       3666
```

Figure 5: Job output

References

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Dul] Tamara Dull. Big data and the internet of things: Two sides of the same coin? [Online].

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[GP11] Yvonne Genovese and S Prentice. Pattern-based strategy: getting value from big data. *Gartner Special Report G*, 214032:2011, 2011.

[IST] ISTAT. Banca Dati i.Stat. `http://dati.istat.it/`. Accessed: 2017-05-10.

[LJ12] Alexandros Labrinidis and Hosagrahar V Jagadish. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, 5(12):2032–2033, 2012.

[LK10] Jacob Leverich and Christos Kozyrakis. On the energy (in) efficiency of hadoop clusters. *ACM SIGOPS Operating Systems Review*, 44(1):61–65, 2010.

[SAD+10] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[vdM17] Rob van der Meulen. Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016, February 2017. [Online; posted 7-February-2017].

[Whi12] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.