Distributed High-Dimensional Index Creation using Hadoop, HDFS and C++

Gylfi Þór Guðmundsson, Laurent Amsaleg, Björn Þór Jónsson

Distributed High-Dimensional Index Creation using Hadoop, HDFS and C++

Gylfi Þór Gudmundsson
INRIA, Rennes, France
gylfi.gudmundsson@inria.fr

Laurent Amsaleg
IRISA–CNRS, Rennes, France
laurent.amsaleg@irisa.fr

Björn Þór Jónsson
Reykjavík University, Iceland
bjorn@ru.is

Abstract

This paper describes an initial study where the open-source Hadoop parallel and distributed run-time environment is used to speed-up the construction phase of a large high-dimensional index. This paper first discusses the typical practical problems developers may run into when porting their code to Hadoop. It then presents early experimental results showing that the performance gains are substantial when indexing large data sets.

1. Introduction

Over the last decade, the technology underlying Content-Based Image Retrieval Systems (CBIRS) has made tremendous progress. CBIRS can now manage collections having sizes that could not even be envisioned years back. Most systems can cope with several millions of images [9, 7] or billions of descriptors [10, 8], and some researchers address web scale problems [3, 1]. Modern CBIRS return query results in a snap and have very good recognition capabilities even though they run approximate k-nearest neighbors queries for speed.

Reducing the response times of systems at ever larger scales has been mainstream in the Computer Vision, Multimedia or Database literature. Searching is now truly fast: milliseconds when the data collections fit in main memory [7] and seconds when data must be fetched from disks [9]. It is unlikely we will observe dramatic response time improvements in the near future as there is not much slack remaining. In contrast, little performance progress has been made for *constructing* high-dimensional indices. Taking the raw collection of high dimensional descriptors and turning it into an index for subsequent ultra-fast searches is still a long, complex, costly, and resource-consuming task. Bulk-loading techniques help, but not that much. When the data collection is on the order of terabytes of high-dimensional descriptors, as is the case when indexing few tens of millions of real world images with SIFT [11], then indexing takes days or even weeks. Not only is the number of I/Os enormous, as reading terabytes and writing back once indexed is inherently slow, but the CPU load is gigantic as many distance calculations must be performed.

Turning to parallel and distributed computing is thus mandatory. Modern frameworks facilitating the parallel and distributed execution of massive tasks are becoming increasingly popular since the introduction of the Map-Reduce programming model, the Hadoop run-time environment as well as its HDFS distributed file system.

This paper is an initial study where Hadoop and HDFS are used to speed-up the creation of a high-dimensional index. Our goal is to gain knowledge on what is at stake when using Hadoop for index construction. This paper does not propose the ultimate distributed index creation algorithm, but rather uses a simple clustering-based indexing strategy as a vehicle for understanding how to distribute data, how to distribute tasks, how the network topology as well as the hardware available affects performance, etc.

Since Hadoop is Java-based, applications exhibit best performance when they are implemented in Java as they benefit from all the features of the framework. In contrast, centralized high-dimensional indexing algorithms are typically implemented in C++ for efficiency. We therefore faced the following dilemma: either completely re-implement the index creation in Java or try to connect the existing C++ code to the Hadoop framework. We decided to connect the C++ code to Hadoop for the following reasons: (i) high-dimensional index creation has access patterns to data that differ significantly from tutorial examples and we wanted to check the viability of going to Hadoop before investing in the re-implementation; (ii) the goal of our study was to get familiar with Hadoop, and to evaluate what distribution and parallelism bring, rather than coming up with a definitive algorithm right away; and (iii) Hadoop requires specific commands to deploy on a cluster of machines and we wanted to gain that knowledge as fast as possible, rather than spending time re-implementing the algorithm "in the dark", not knowing how to use the framework in practice.

We are well aware that connecting C++ and Hadoop raises many problems and that the performance gains are

far from what we could get otherwise. We believe that we are not the only ones to face such a dilemma and that approaching the problem from this perspective, which somehow balances the performance gains and the programming investment, is worth sharing with the multimedia community. This is the first contribution of this paper; the actual deployment and performance measurements being the other.

The paper is structured as follows. Section 2 presents the Map-reduce framework, Hadoop and HDFS. Section 3 describes the centralized version of the index-creation part of the extended-Cluster Pruning algorithm which gets distributed using Hadoop. The way the algorithm is distributed is presented Section 4. Section 5 then explains what practical problems application developers run into when connecting their code to Hadoop. Section 6 gives experimental results and Section 7 concludes the paper.

2. Background

2.1. Map-Reduce

Map-Reduce is a programming model for processing extremely large datasets. It applies to a distributed context where independent nodes run tasks, typically in a cluster of standard, inexpensive hardware. The run-time environment for Map-Reduce transparently handles the partitioning of the input data, schedules the execution of tasks across the machines and manages the communications between processing nodes when sending and/or receiving the records to process. Furthermore, the run-time deals with node failures as it is able to restart aborted tasks on nodes, possibly on replicated data in case of unavailability. The run-time tries to save as much network bandwidth as possible by increasing the likelihood of running computations on the data that is locally stored on nodes and by paying attention to the topology of the network.

To program an application, the user must give the implementation of the `Map` and the `Reduce` functions, which the run-time environment will call when appropriate. The execution flow of typical Map-Reduce applications is as follows. First, the data to process is loaded into a specific file systems where blocks of data are distributed onto multiple nodes [4]. This is a transparent process.

When the application starts, the run-time creates M instances of the `Map` function. Each `Map` function gets from the file system a chunk of data to process, independently of others. The input data is made of (key, value) records. Each `Map` iteratively processes input (key, value) records from its chunk and produces intermediate output (key, value) records. Output data is typically stored in RAM, to local disks if necessary. When the output data is big enough or when a chunk is entirely processed, the run-time sorts (key, value) records according to the 'key' field, then partitions the sorted records into R sets which constitute the input data of the R instances of the `Reduce` functions that are then created by the system. The run-time thus sends from `Map`-tasks to `Reduce`-tasks loads of data; it tries to limit the network consumption as much as it can. Note that `Map` and `Reduce` tasks might run simultaneously.

Each `Reduce` function thus receives sorted data from multiple sources and merges data items before processing them. This produces some final results that are handed out to the distributed file system before eventually reaching the user. Map-Reduce was originally developed by Google [2]. It is a very simple yet extremely effective programming model adopted by many large scale applications designers.

2.2. Hadoop and HDFS

Hadoop [14] is a Java-based open-source version of the Map-Reduce framework. It includes an implementation of a distributed file system called HDFS. HDFS provides data location awareness to the Hadoop run-time environment. In turn, Map-Reduce applications built with Hadoop try to run their tasks on the node where the data is. HDFS supports data replication, both for performance and fault tolerance. The HDFS file system is designed for large files (their size is typically a multiple of 64MB) across multiple machines and it is assumed that most access patterns to data are large sequential reads and/or writes; that assumption offers the opportunity for specific optimizations. Note that random accesses are possible, but they may perform poorly. One specific machine is responsible for managing the mapping between file blocks and their location within the distributed system. Fault tolerance is achieved by replication of file blocks to multiple machines, defaulting to 3. Another machine is typically responsible for scheduling all the tasks and monitoring the execution flow.

3. Extended Cluster Pruning

This section gives a brief overview on the centralized version of the high-dimensional indexing scheme we use in this paper. In the next section we describe how it is distributed and implemented within the Hadoop framework.

3.1. Centralized eCP

We decided to build on top of the extended Cluster Pruning (eCP) algorithm [5] because it is quite representative of the core principles underpinning many of the quantification-based high-dimensional indexing algorithms that perform very well [13, 6]. Overall, eCP is very related to the well-known k-means approach. As k-means, eCP is an unstructured quantifier, thus coping quite well with the true distribution of data [12]. eCP is designed to be I/O friendly

as it is assumed that the database size is too large to fit in memory and must reside on secondary storage.

Overview of Index Creation. eCP starts by randomly selecting C points from the data collection. They are used as representatives of the C clusters eCP will eventually build. Then the remaining points from the data collections are read, one after the other, and assigned to the closest cluster representative. When the data collection is large, the representatives are organized in a multi-level hierarchy. This accelerates the assignment step as finding the representative that is closest to a point then has logarithmic complexity rather than linear complexity. Once all the raw collection has been processed, then eCP has created C clusters as well as a tree of representatives, all stored sequentially on disks. The tree structure is very small compared to the data. The tree is a hierarchy: the points used at each level of the tree are representatives of the points stored at the level below.

Overview of the Search. When searching, the query point is compared to the nodes in the tree structure to eventually find the closest cluster representative. Then the corresponding cluster is accessed from the disk, fetched into memory, and the distances between the query point and all the points in that cluster are computed to get the k-nearest neighbors. The search is approximate as some of the true nearest neighbors may be outside the cluster under scrutiny.

Details of Index Creation. This study focuses on the efficiency of the index creation which is the most I/O intensive phase, much more intensive than the search phase. Not only does index creation need to read the entire data collection from secondary storage, and then eventually write everything back to disks, but it must also perform a huge number of CPU intensive distance calculations to cluster vectors. The index is created in a bulk manner: all vectors to index are known before the process starts and the index tree structure, as well as the bottom leafs of the tree, are created in one go.

The creation starts by building its in-memory index tree by picking cluster representatives from the raw collection. Then it allocates a buffer, called *in-buff*, for reading the raw data collection in large pieces. It then iterates through the raw collection via this buffer, filling it with many not-yet-indexed vectors. The index is used to quickly identify the cluster representative that is the closest to each vector in *in-buff*, representing the cluster that vector must be assigned to. Once all vectors in *in-buff* have been processed, then *in-buff* is sorted on increasing values of the cluster representative identifiers. It then creates a new temporary data file on disk, and flushes *in-buff* into that file before closing it. It then reads another large piece from the raw collection into *in-buff* and loops.

After having processed all vectors from the raw collection, a phase merging that sorted data is initiated. It eventually creates a single large file where all the data points assigned to one cluster are stored in a continuous manner, sequentially. During this phase, all the temporary files are opened simultaneously and iteratively read. The merge process simply reads data from all files and appends to the final file the data that belongs to the currently built cluster.

In terms of access patterns, the index construction performs large sequential reads to fill *in-buff* and large sequential writes when creating each temporary file. When creating the final file, it performs many small random reads to get data from all the sorted temporary files and large sequential writes for the final file.

4. Distributing the Index Creation of eCP

The index creation process of eCP can be split into three main phases:

Phase #1: Creation of the index tree. During this phase, cluster representatives are picked from the collection and organized in a in-memory tree.

Phase #2: Vectors are assigned to cluster in rounds of executions, each round creating a temporary file where vectors are sorted per representative.

Phase #3: The sorted temporary files are merged into a unique final file.

Obviously, phases #2 and #3 are good candidates for being Hadoop-ed. It is rather trivial to foresee that chopping the entire data collection into independent parts assigned to physically distinct computing nodes is going to speed up the whole process. Phase #2 is particularly intensive as the whole data collection must be read, and vector-to-cluster assignment requires a lot of CPU for computing distances. This is to contrast with Phase #3 which is almost solely I/O intensive but involves very little CPU, since merging is a matter of comparisons and hence is cheap. For these reasons, Phase #2 is the prime candidate for parallelization and distribution.

In contrast, making Phase #1 parallel is very complicated because cluster representatives are randomly picked from the data collection and all the nodes participating in the distributed creation of the index must use the same set of representatives. Otherwise, no consistency would exist between what each node assigns and the final merge phase would make no sense. It is instead very easy to pick representatives in a centralized manner, before starting the distributed computation, and to send them to every node. When a node receives this set of representatives, it starts by building the index tree in memory and can then proceed with the assignment to clusters of the points it must deal with.

In this paper, we are not considering at all the search phase of eCP, which has not been distributed. Note also that

the quality of the approximate searches is not at all impacted by the distributed index creation: the final index is exactly identical, regardless of whether it has been generated in a centralized or distributed manner. The result quality and efficiency of the search is studied in detail in [5] .

5. C++/Hadoop and Reality

As was said in the introduction, we decided to connect our existing index creation code implemented in C++ to Hadoop instead of re-implementing the whole process in Java. While this proved to work well, with great performance gains (see Section 6), it has a number of problems, which are discussed below.

Reading Binary Data. The high-dimensional vectors that are extracted from images and must be indexed are typically stored in a binary format. This is for efficiency: binary data is compact, ready to be used in distance calculations and no conversion is needed. This is problematic, however, when using Hadoop. Hadoop is Java-based, but it provides programmers with interfaces for running C++ code. Unfortunately, there is no interface for reading binary records from C++, as only a text-based record reading interface is provided. It is clearly unacceptable to convert all vectors to text to inject data into the Map-Reduce pipeline as this would take so much space on disks for large scale settings. Instead of passing to Hadoop the data to index, we therefore decided to pass to Hadoop text data that is subsequently used by our C++ code to know what to index.

It works as follows. First, we decide beforehand what will be the size of the data chunk each `Map`-task will process. Then we produce a series of records keeping track of the resulting offsets each `Map`-task will have to use to read its part from the raw data collection. Then the run-time of Hadoop starts `Map`-tasks, each receiving an offset record. Each `Map`-task then calls the C++ code that reads its own portion of the data file before assigning vectors to clusters.

The main drawback of not passing the data through the Map-Reduce pipeline is that this does limit Hadoop's ability to start `Map`-tasks where the data resides in HDFS, which in turn puts much more load on the underlying network. HDFS will, however, still try to minimize the cost of remote reads by reading locally if possible, then from machines on the same rack, then from more remote sources.

HDFS-block Sizes vs. Data Chunk Sizes for Hadoop. For performance, HDFS distributes the blocks (typically 64MB) of each file on different machines. This offers opportunities for true parallelism because multiple machines can simultaneously deliver the data from a single file. While this is totally transparent when running tasks within Hadoop, it is more complicated otherwise. It is difficult to know precisely what are the offsets of HDFS-block boundaries in order to give these offsets to the C++ tasks. It is therefore possible that two C++ tasks reading data indeed access the same HDFS-block, which potentially hurts performance.

No Data Filtering. Originally, the Map-Reduce model was designed to process large volumes of textual data, such as log files. In this case, `Map`-tasks typically serve as a severe filter reducing the amount of data that gets passed along, i.e. copied, to the `Reduce`-tasks (the word-counting example often used to demonstrate Hadoop is like this). For this reason, the framework controls the number of `Map`-tasks to start, which explains why there are typically far more `Map`-tasks than `Reduce`-tasks. The index creation workload breaks this model as the volume of data that travels through the system is not decreasing at all; the number of vectors does not change between the start and the end of the indexing process. This puts a lot of pressure on the network traffic as well as on the disks as all data must be eventually pushed back to disks again.

No Implicit Sorting. As the data to index is not inside the Map-Reduce pipeline, the run-time can not sort what is produced by `Map`-tasks. We sort vectors from C++.

No `Reduce`-Tasks. Hadoop can fire `Reduce`-tasks as soon as enough data has been produced by `Map`-tasks, allowing for possible simultaneous execution. Here, again, as we are outside the Hadoop pipeline, `Reduce`-tasks can not be easily defined. Even if they could, they would run into the problem of reading binary data.

Merging Outside Hadoop. Merging the intermediate data files produced by the `Map`-tasks is done outside the Hadoop framework, as a separate process that is started once the last `Map`-task completes. Merging consumes only little CPU as it solely compares representative identifiers and copies data to the final file. There is thus very little to gain from making the merge phase parallel inside Hadoop. Note that the I/O load is reduced if there are only a few files to merge since this reduces the number of random reads.

Large vs. Small Tasks. A rule of thumb to get good performance with Map-Reduce is to break jobs into as small tasks as possible. This gives the framework maximum flexibility for placing tasks on nodes and to keep nodes constantly busy. With indexing, each `Map`-task launches C++ code creating the index memory tree. This takes a fixed amount of time. Then vector assignment starts. If we follow that rule of thumb, then we have to pay that fixed cost very often, which hurts performance. Therefore, it is better to run big enough `Map`-tasks to marginalize the tree construction cost.

6. Experiments

6.1. Data Collection and Indexing Set-up

We created an image collection composed of 100,000 random pictures downloaded from Flickr that are very diverse in contents. All images have been resized to 512 pixels on their longer edge. We computed the SIFT local description of these images using the code by Lowe [11]. This results in more than 110 million vectors of dimension 128, for a total of almost 13.6GB of data.

The parameters driving the index creation have been set according to the conclusions of [5]: the in-memory index tree has 3 levels, each cluster contains 992 vectors on average, and there are 111,424 clusters.

6.2. Hardware

We run our experiments on the Grid5000 using the clusters in Orsay, France. To quote: "Grid5000 is a scientific instrument for the study of large scale parallel and distributed systems. It aims at providing a highly reconfigurable, and monitorable experimental platform to its users."[1] The hardware is actually two clusters, GDX with 310 machines and netGDX with 30. The specifications are as follows: each machine has two 2.0Ghz or 2.4Ghz AMD Opteron CPUs with 1MB cache, 2GB RAM and 80GB SATA disks. GDX is organized in 16 racks of 18 machines where each machine connects by a 1Gbps link to one of the 14 Cisco Catalyst 3750 or one of the 2 HP Procurve switch. In addition there is one rack of 22 machines that interconnect with only 100Mbps links to a HP Procurve switch, for a total of 310 machines and 17 switches. The switches are then interconnected by a Cisco Catalyst 6509 by 2x or 3x aggregated links. The 30 machines of the netGDX cluster connect directly by 1Gbps links to the Catalyst 6509 unit.

The clusters operate on a declarative basis as users requests exclusive access to a certain amount of hardware for a predefined amount of time but cannot specify what hardware. The infrastructure then grants access to machines, but does not particularly pay attention to minimizing network distances between granted machines. It is thus not uncommon to end up with machines spread all over the local network. This is a potential bottleneck, especially if other applications running on the cluster heavily use the network.

6.3. Hadoop Settings

Three machines have a special role when deploying Hadoop: the Primary and Secondary Name Nodes and the Job Tracker. The remaining machines run tasks and serve HDFS data blocks. For an experiment involving N CPUs, Hadoop must thus be deployed on $(N + 6)$ machines.

The data to index is copied into the HDFS system before launching experiments. Each machine in the Grid5000 cluster has 2 CPUs. Hadoop is thus set to run at most two tasks per node. Since each CPU has only 1GB of RAM, in which the in-memory index tree as well as the system must fit, we set *in-buff* to be only 60MB to avoid swapping and thrashing. Overall, this gives 236 tasks to run to complete index construction, each processing one 60MB chunk.

[1] See https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home.

Table 1. Scaling index creation with Hadoop.

Machines	CPUs	Tasks	Time	Ratio
1	1	1	473m 46s	100%
23	40	236	19m 13s	4.1%
62	118	236	7m 31s	1.6%
62	118	236	21m 10s	4.5%
13	20	20	26m 56s	5.7%

6.4. Baseline and Scalability

The goal of the first experiment was to determine the performance gains when distributing the creation of the index. We thus started by running the indexing process on a single machine of the Grid5000 cluster to get the baseline, and then ran the indexing process on multiple machines. The results are reported Table 1. For this experiment, the HDFS-block size was set to 33MB and the replication factor to 1 (data is not replicated).

Table 1 shows that it takes almost 8 hours to create the index on one CPU inside one machine. We then deployed Hadoop on 40 CPUs to do the indexing jobs (and 3 machines thus 6 CPUs for managing the run-time). These 40 CPUs thus run the 236 tasks in about 20 minutes, which is only 4.1% of the baseline. Observing the execution, we saw that some machines ran 6 indexing tasks while others only 5; speed-up is thus not optimal as some machines remained idle while others were busy. Note also that the in-memory index tree had to be constructed 236 times.

We then changed the set-up to deploy on 62 machines, thus using 118 CPUs for index creation, asking each CPU to run only two tasks. One run was extremely fast: the complete index was built in less then 8 minutes, which is 1.6% of the baseline. Other runs had fluctuating performance, the worst one needing about 21 minutes to complete. This can be explained by networking and disks problems as some rounds of experiments were competing for networking and disks resources with the 278 other machines running different experiments. Still, going from 8 hours to 8 minutes shows that great gains can be expected.

Table 2. Effects of replication factor.

Replication Factor	Time	Ratio
1	25m 20s	5.3%
2	14m 49s	3.1%
3	12m 09s	2.6%
4	12m 11s	2.6%
10	10m 06s	2.1%
20	12m 14s	2.6%

We also ran an experiment where we tried to make all indexing tasks complete at the same time, leaving no machine idle while others were still running. We deployed on 13 machines, reserving 20 CPUs for running the index creation, and manually forced each CPU to read 12 chunks of 60MB, which covers the entire data collection. The results are the ones given by the last line of Table 1. The index is built after 27 minutes, or 5.7% of the baseline time, which is close to the optimal which is 5.0% (1/20).

6.5. Effect of Replication

The first experiment had no data replication and therefore index creation tasks were likely to compete for reading data coming from the same HDFS-blocks. Increasing the replication factor of course consumes more disk space but also allows the run-time to reduce disk competition as identical blocks can be sent to requesting machines from different sources. Increasing the replication factor also increases the likelihood for an indexing task to get the data it needs from its local disk.

To study the effects of varying the replication factor, we reused the setting where Hadoop is deployed on 62 machines or 118 CPUs. Table 2 reports the impact of the replication factor on the response time for completing the creation of the index. Note that the whole set of experiments turned out to be performed while the Grid5000 cluster was heavily loaded with other tasks. With a replication factor of 1, then it takes about 25mn (21 in the first experiment). Increasing the replication factor results in reduced response time; there is less disk competition. But the improvement reaches a plateau as soon as the network becomes the bottleneck. In our case, it is not useful to massively replicate HDFS blocks hoping to nicely distribute the data-feeding load across machines as the network is quickly saturated. Note Hadoop does not actively try to balance the block placement. In practice, it might be possible that all the copies of the same block end-up on a unique machine... Yet, taking into account the network topology as well as its usage, since it is shared, is key to performance. We ran other experiments varying the HDFS-block size; a similar trend was observed.

7. Conclusion and Lessons Learned

This paper presents an initial study where the creation of a high-dimensional index is made parallel and distributed by using the Hadoop framework. Early experimental results show substantial performance gains, despite the fact that the Hadoop framework is loosely coupled to the C++ based index creation. Two main lessons can be drawn from this work: (i) it is key to invest time, energy and manpower to re-implement the code in Java in order to benefit from all the features of Hadoop—although our results are already impressive, even better performance gains will be observed if the index creation is re-implemented in Java; and (ii) special care must be taken to account for the networking topology to prevent message exchanges from becoming the new bottleneck, when parallelism fixes the CPU bottleneck and HDFS the I/O bottleneck.

References

[1] M. Batko, F. Falchi, C. Lucchese, D. Novak, R. Perego, F. Rabitti, J. Sedmidubský, and P. Zezula. Building a web-scale image similarity search system. *MTAP*, 2010.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *CACM*, 2008.

[3] M. Douze, H. Jégou, H. Singh, L. Amsaleg, and C. Schmid. Evaluation of gist descriptors for web-scale image search. In *CIVR*, 2009.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, 2003.

[5] G. Gudmundsson, B. Jónsson, and L. Amsaleg. A large-scale performance study of cluster-based high-dimensional indexing. In *VLS-MCMR Workshop with ACM MM*, 2010.

[6] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 2011.

[7] H. Jégou, F. Perronnin, M. Douze, J. Sánchez, P. Pérez, and C. Schmid. Aggregating local image descriptors into compact codes. *TPAMI*, 2011.

[8] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, 2011.

[9] H. Lejsek, F. Amundsson, B. Jónsson, and L. Amsaleg. NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *TPAMI*, 2009.

[10] H. Lejsek, B. Jónsson, and L. Amsaleg. NV-Tree: nearest neighbors at the billion scale. In *ICMR*, 2011.

[11] D. Lowe. Distinctive image features from scale invariant keypoints. *IJCV*, 2004.

[12] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 2010.

[13] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.

[14] T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2010.