Sampling with Incremental MapReduce

Marc Schäfer, Johannes Schildgen, Stefan Deßloch

Heterogeneous Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
{m_schaef,schildgen,dessloch}@cs.uni-kl.de

Abstract: The goal of this paper is to increase the computation speed of MapReduce jobs by reducing the accuracy of the result. Often, the timely processing is more important than the precision of the result. Hadoop has no built-in functionality for such an approximation technique, so the user has to implement sampling techniques manually. We introduce an automatic system for computing arithmetic approximations. The sampling is based on techniques from statistics and the extrapolation is done generically. This system is also extended by an incremental component which enables the reuse of already computed results to enlarge the sampling size. This can be used iteratively to further increase the sampling size and also the precision of the approximation. We present a transparent incremental sampling approach, so the developed components can be integrated in the Hadoop framework in a non-invasive manner.

1 Introduction

Over the last ten years, MapReduce [DG08] has become an often-used programming model for analyzing Big Data. Hadoop[1] is an open-source implementation of the MapReduce framework and supports executing jobs on large clusters. Different from traditional relational database systems, MapReduce focusses on the three characteristics (”The 3 Vs”) of Big Data, namely volume, velocity and variety [BL12]. Thus, efficient computations on very large, fast changing and heterogeneous data are an important goal. One benefit of MapReduce is that it scales. So, it is well-suited for using the KIWI approach (”Kill It With Iron”): If a computation is too slow, one can simply upgrade to better hardware (”Scale Up”) or add more machines to a cluster (”Scale Out”).

In this paper, we focus on a third dimension additional to resources and time, namely computation accuracy. The dependencies of the dimensions can be depicted in a time-resources-accuracy triangle. It says that one cannot make Big-Data analyses in short time with few resources and perfect accuracy. The area of the triangle is constant. Thus, if one wants to be accurate and fast, more resources are needed (KIWI approach). If a hundred-percent accuracy is not mandatory, a job can run fast and without upgrading the hardware.

On the one hand, most work regarding Big-Data analysis, i.e. frameworks and algorithms are 100% precise. On the other hand, these approaches often give up the ACID properties

[1] http://hadoop.apache.org

and claim: Eventual consistency ("BASE") is enough. So, let us add this: For many computations, a ninety-percent accuracy is enough. One example: Who cares if the number of your friends' friends' friends in a social network is displayed as 1,000,000 instead of 1,100,000?

Some people extend the definition of Big Data by a fourth "V": *veracity* [Nor13]. This means, the data sources differ in their quality. Data may be inaccurate, outdated or just wrong. So, in many cases, Big-Data analyses are already inaccurate. When using sampling, the accuracy of the result decreases again, but the computation time improves. Sampling means, only a part of the data is analyzed and the results are extrapolated in the end.

Within this work, we extended the Marimba framework (see section 4.1) by a sampling component to execute existing Hadoop jobs with just a few changes in the user's jobs and no changes in the Hadoop framework. We use the so-called *Overwrite Installation* for making sampling jobs incremental. So, when increasing the size of samples, results from a former computation on a small sample size can be reused. For our framework, we use common sampling techniques which are presented in the following subsection.

In the following, some related work is presented. These are existing frameworks which also add sampling to MapReduce and techniques our approach is based on. In section 3, we explain commonly-used sampling techniques and present our approach for a Hadoop-based sampling framework. Section 4 focuses on making sampling incremental to increase the accuracy of a computation without starting from scratch. The sections 5 and 6 contain a performance evaluation and a conclusion.

2 Related Work

Hadoop RandomSampler There are different approaches for doing sampling in MapReduce. First of all, there is no built-in sampling in Hadoop. But for getting an idea of how the input data looks like, a *RandomSampler*[Whi09, p. 226] can be used. With that, Hadoop reads a given percentage of the input data (key-value pairs) which is stored on different machines and collects the keys of these. This is done before the Map phase and the result is a *key set*. This key set is a subset of all input keys and can be used to define key intervals for Map-output partitions. A *TotalOrderPartitioner* produces partitions with a better distribution than without sampling, so every Reducer has an equal amount of work. An example: Input keys are URLs of websites. Some of them start with "mail.", some with "mobile.", but more than 99% of all keys start with "www.". If the number of Reducers is three, a TotalOrderPartitioner would create three partitions, e.g. $[a-i]$, $[j-r]$ and $[s-z]$. But as there is no uniform distribution, 99% of the Map outputs will be sent to the third Reducer. With Random Sampling, the partitions are of the same size, e.g. $[a - www.k]$, $[www.k - www.q]$ and $[www.r - z]$. This kind of sampling can be used when Reducers need their keys in the right order. As this is not needed for most MapReduce jobs, a simple *HashPartitioner* ensures a good distribution by performing modulo hashing. Nevertheless, different from our approach, this sampling technique fails to increase computation performance.

EARL The *Early Accurate Result Library* (EARL) [LZZ12] is a modification of the Hadoop framework to run MapReduce jobs on a subset of the data. This subset is randomly created in a *Sampling Stage*, the first of three phases. In the second phase, *resamples* are created which are subsets of the given sample. Each of these resamples is used as input for an individual MapReduce computation. The outputs of these are compared to each other to determine the accuracy. This is done in the third phase, called *Accuracy Estimation*. EARL changes Hadoop very deeply. It forces MapReduce jobs not to stop after they are finished and to accept more input data by increasing the input-sample size. This is done until a desired accuracy is reached.

EARL provides two strategies for reading a subset of the input, namely *Pre-Map sampling* and *Post-Map sampling*. As their names imply, the first strategy is to skip most of the input tuples and only perform the Map function on p% of the input. In the second strategy, the Map function is computed on the full dataset, but if it wants to emit data, this is only done with a probability of p%. So, most of the Map output is discarded. Pre-Map sampling is faster than Post-Map sampling because the Map function is called less often, but it leads to a lower accuracy because it either processes a segment completely or not at all. In both Pre-Map and Post-Map sampling, the Reducers only get a subset of the complete intermediate-key/value pairs. In the second approach, they are more random. EARL does not extrapolate the Reduce outputs automatically. Instead, the user has to provide a *Correct* function. So, if for example a $p\%$ sample of a large text is read to count the occurrences of terms in the text, the Correct function has to multiply the count values in the end by $100/p$ to estimate the real quantities. In our framework, one does not need to provide a Correct function and our framework does not modify the Hadoop core.

Sampling for Hadoop Other works focuses only on sampling the input. Our approach also extrapolates the results. In [GC12], Grover and Carey tested how to increase the performance of Hadoop when doing *predicate sampling*, i.e. finding n items in a given dataset which fulfill a predicate. The algorithm stops when enough items are found. The randomness of the sampling set is not important for this algorithm. So, the results can not be used to estimate properties of the full dataset well.

Apache's Pig provides a *sample* operator[2] for shrinking a dataset. The following Pig script will produce a 10% sample: `a = LOAD 'data.csv'; s = SAMPLE a 0.1`

Sampling in Relational DBMS In [OR90], Olken and Rotem suggest sampling as a DBMS operator. This leads to changes in the query language, optimizers and access paths. It is described how B+ trees and hashing algorithms have to be modified for accessing only a specific percentage of the data. The objectives are estimating the results of aggregate functions with the usage of sampling techniques. In [PBJC11], these techniques are built into MapReduce. However, only simple aggregate functions after a grouping are supported. In [PJ05], bootstrapping is used for relational databases and again, the approximation method which is used there only supports simple aggregates. In [Fis11], an interactive data-visualization tool is presented that incrementally delivers more accurate

[2] https://pig.apache.org/docs/r0.11.1/basic.html#sample

results to a given query. Instead of returning an inaccurate result value to the user, the result is presented as a range of possible values. The tool can be used to produce many kinds of histograms based on GROUP BY queries.

3 Sampling with Hadoop

Sampling is used to draw a subset of a population. Afterwards the characteristics of the population are estimated based on the characteristics of the subset. The goal is to get a representative subset in order to get more accurate extrapolations. There are different techniques for choosing a subset [Sud76]. In the following, three of them are described (see Figure 1).

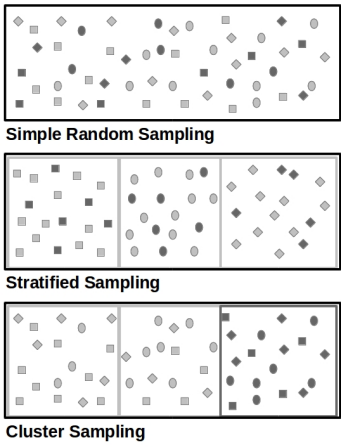

Figure 1: Three Sampling Techniques

Simple Random Sampling When using *Simple Random Sampling*, all elements are chosen with the same probability of $p\%$. According to the *Law of Large Numbers*, the subset of a population of size *n* will have the size $n \cdot p/100$.

Stratified Sampling In the *Stratified-Sampling* approach, the population is divided into sub-populations. On each of these, sub-population sampling is done. To improve the accuracy, the dividing characteristics should correlate with the focused characteristic. Additionally, the sub-populations should be homogeneous.

Cluster Sampling This approach also divides the population in subpopulations (clusters), but this division is done *naturally*, based on for example location, etc. Afterwards one or more clusters are randomly chosen and are completely used as the subset for the extrapolation. There is the risk, that the extrapolation might be biased because of homogeneous clusters. It is possible to reduce that risk by using a *Simple Random Sampling* on the chosen clusters.

Our goal is to leave the MapReduce framework intact. So we first show how to change an existing Hadoop job to improve the performance at an expense of accuracy. Later in this section, we present a framework which can be used on top of Hadoop to execute existing Hadoop jobs with the usage of sampling [Sch14]. With that, the user should not have to change her Map and Reduce functions depending on whether the whole input or only a subset is analyzed.

3.1 Restrictions

There are three kinds of MapReduce jobs which must be treated differently when enabling sampling techniques on them.

Sampling only Top-k queries, computations of MIN, MAX or AVG, and the determination of a set of items with certain properties are some examples for jobs with no need for an extrapolation. The result of a computation on a subset of the input can be used as a final result. Inaccuracies arise through missing hits: Maybe another skipped value is higher than the computed maximum and maybe some skipped items fulfill the search criteria as well. The average function does not need to be extrapolated because it is a division of a sum and a count. The extrapolation would be canceled out.

Sampling and Extrapolation Counting and computing sums are popular tasks in MapReduce jobs. If these operations are performed on a subset, the result has to be multiplied by $100/p$ where p is the sampling percentage. Other tasks which produce complex objects or compute the final results by a multiplication need different extrapolation functions. For the latter one the result r has to be corrected like this: $r^{100/p}$.

Sampling not supported Iterative algorithms, graph transformations and more do not support sampling because the output computed on a subset is useless. They always have to be executed on the full input.
In our work, we focused on the *Sampling and Extrapolation* class of jobs. It stands to reason that *Sampling only* jobs are supported too. Here, the extrapolation can be seen as the identity function.

3.2 Subset selection

As it is done in EARL (see section 2), a user can implement a Pre-Map or a Post-Map sampling by either changing the *RecordReader* of the used *InputFormat* to skip $(100-p)\%$ of the items. Or one adds some lines to the user-defined Map function to skip input or output key-value pairs with a probability of $p\%$.

Our goal is an automated subset selection. Therefore, we propose a *SamplingInputFormat* which delegates all function calls to the actual user-defined InputFormat and is able to skip items in the input splits (*item sampling* / for texts: *line sampling*), or to skip splits as a whole (*split sampling*). A split is a chunk of items which are stored on one machine and processed by one Mapper node. When using text files which are stored in the Hadoop Distributed File System (HDFS), a split usually is a 64 MB large text snippet. The lines in this snippet are called *items* and are the input for one Map call.

A selection based on the set of items corresponds to the *Simple Random Sampling* technique. Thereby, a potential uniform coverage is given, since the selection is done on the smallest usable unit. The disadvantage of this technique is that the RecordReader has to read every item and discard most of them.

A less time-consuming approach is a selection on the set of splits. This approach corresponds to *Cluster Sampling* where the splits are the naturally-grouped subsets based on locality. This approach discards whole splits, which is significantly less computation than in the previous approach. If the data is not uniformly distributed, the split approach could lead to biased results.

A user can balance the pros and cons of both approaches by combining them. Our frame-

work provides the configuration of two percentage values for both item and split selection, so one can perform a 10% sampling by skipping every second split and in each split doing a 20% item sampling.

Jobs of the class *Sampling only* don't need any further modifications or considerations. Item sampling, split sampling and the combination of both are all Pre-Map approaches. The user-defined Map function is called for the subset of items the *SamplingInputFormat* provides. The Map and Reduce functions can stay unchanged and no extrapolation is needed.

3.3 Extrapolation

For jobs of the class *Sampling and Extrapolation*, the final output has to be corrected to estimate the result of a computation on the whole data. Approaches presented in section 2 rely on a user-defined Correct function. This function is executed *Post-Reduce*. We want to introduce a generic *Pre-Reduce* extrapolation method. The benefit of that is that Map and Reduce can be left unchanged and that no Correct function is needed.

Example (Post-Reduce / Pre-Reduce Extrapolation) A MapReduce job is used to compute the number of website visits per day by analyzing log files. 5% of the input is sampled and passed through the Map function. Using *Post-Reduce Extrapolation*, the Reduce function sums up all visit counts for one day. After that, a Correct function has to multiply the numbers by 20. With *Pre-Reduce Extrapolation*, before calling the Reduce function, each intermediate key-value pair is copied 19 times so it reaches the Reducer twenty times. The output in both approaches is the same.

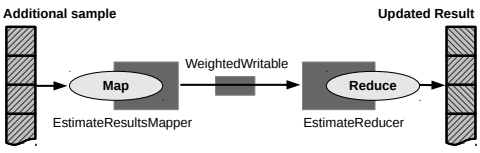

Figure 2: The extrapolation based on the old result and a new sample.

At a first glance, the Pre-Reduce approach sounds expensive because the amount of data shipped to the Reducers increases drastically. We solved this by copying the data just virtually by encapsulating the user-defined Map-output type and the Reduce function (see Figure 2). A *WeightedWritable* object contains one Map-output value and a weight value (in the example above it would be 20). The user-defined Reducer is replaced by an *EstimateReducer* that delegates each function call to the actual Reducer but with replacing the Iterable object that contains all values by a *MultiIterable*. This is programmed to return each value n times when calling the `next()` function.

With this approach, a problem arises for a non-integer n since an element cannot be reused for example 0.6 times. A simple solution would be to round the value, but then the accuracy of the result is too low. For example, when sampling a 62.5% subset, $n = 1.6$. Rounding would lead to $n = 2$, which would lead to a simulated dataset of 125% of the original dataset size. We solved this problem by iterating over each element $\lfloor n \rfloor$ times and with a probability $n - \lfloor n \rfloor$ once more. So, if $n = 2.6$, each element is either processed two or three times. The probabilities for that are 40% respectively 60%. As it is shown later, this approach leads to good results.

4 Incremental Sampling

The previous section showed how to extend Hadoop with a generic sampling approach. But it is hard to determine which sampling percentage to choose to deliver a sufficient accuracy. One is not able to tell the accuracy of a computation without comparing the result to the result of the full computation. EARL (see section 2) estimates the accuracy by sampling multiple times and comparing the results. We use an approach that is often used in iterative graph algorithms like PageRank [PBMW99]. If the change of the results towards a former iteration is below a threshold, no further iteration is needed. This means for sampling, one starts with a small sample size and increases this size step by step until the results converge. To increase the precision of an already computed estimation, it is not necessary to discard the former result. Instead, it is sufficient to make a new computation and combine its results with the former one. Our incremental approach is based on the Marimba framework.

4.1 Marimba Framework

A Hadoop-based framework, called Marimba [SJHD14] can be used to create self-maintainable MapReduce jobs. Based on the two strategies *Overwrite* and *Increment Installation* presented in [JPYD11], Marimba executes a MapReduce computation only on the changes in the base data since the former computation and aggregates the results to the former results. If this aggregate operation is a simple plus operation and if the output format supports in-place additions, the *Increment Installation* can be used. HBase has a support for this strategy, so if for example a Reduce output key/value pair is `('high', 5)`, the current value for the key `high` is read, it is increased by five and written back. As this read operation is not a sequential but a random read, the Increment Installation is very expensive if there are many changes in the input since the former computation. Therefore, the *Overwrite Installation* can be used. Here, the full former result is read and aggregated to the Map output records of the changed data. In the end, the final result overwrites the former one. A benefit of Marimba is that the user can simply execute a normal Hadoop job with only making a few changes. First, the Map-output values have to form an Abelian group. This can be reached by replacing the Hadoop type *LongWritable* with the invertible Marimba type *LongAbelian*. Second, the user has to define a *Deserializer* to reuse the former result.

4.2 Reusing Former Sampling Results

A weighting of the inputs is necessary since every input can have a different size. One example: After a one-percent sampling, another percent is sampled and the results are combined. As both results are too different, another percent should be sampled. Now, the former result has to be double-weighted and the new data single-weighted. The end result is then based on three percent of the input data. Executing multiple incremental one-percent jobs is much faster than classical non-incremental jobs. As a drawback, it leads

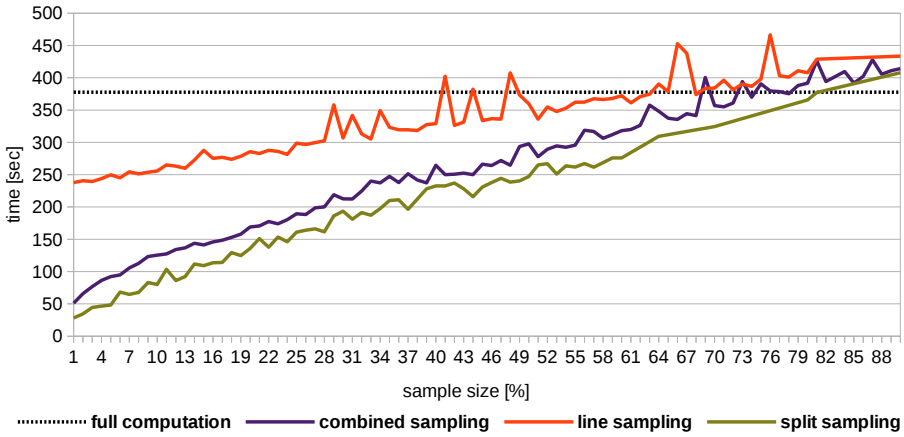

Figure 3: The three sampling approaches in comparison.

to a shade lower accuracy because of overlaps in the input samples. Incremental sampling corresponds to *sampling with replacement*, classical approaches to *sampling without replacement*. Note that the maximum number of replacements is limited by the number of iterations.

An incremental sampling job needs to read two inputs: a former result and an additional sample. To reuse the already computed result, another Map function is needed to reverse the extrapolation from the former iteration. This function converts Reduce output key/value pairs to Reduce input key/value pairs. In many MapReduce jobs, the input and output type are equal. Then an identity mapping is sufficient. As mentioned in the previous chapter, the *WeightedWritable* class is used as intermediate-value class. The weights for the reused results depends on the former sample size.

Our incremental-sampling framework uses the Overwrite Installation approach [JPYD11]. The incremental computation is divided into two steps. The first step is a normal MapReduce sampling job for a new subset of the input data. The result of this job together with a former result are the inputs for the second job. This second job is the incremental part which computes the updated result. For this job, the user needs to develop a Map function which reads the data and the old results and converts these into key/value pairs. Also, a Reduce function is needed to write the new result. Weighting is automatically done after the user-defined Map function through the intermediate-value class *WeightedWritable* which contains the value together with the weighting factor. A special Reduce function is needed to process this intermediate results. This function computes the final Reduce outputs based on the input values and the weighting factor.

5 Evaluation

We tested our sampling framework on a five-node Hadoop cluster. Each node consists of a Quadcore 2.53 GHz CPU and 4 GB of main memory. They are connected via Gigabit Ethernet.

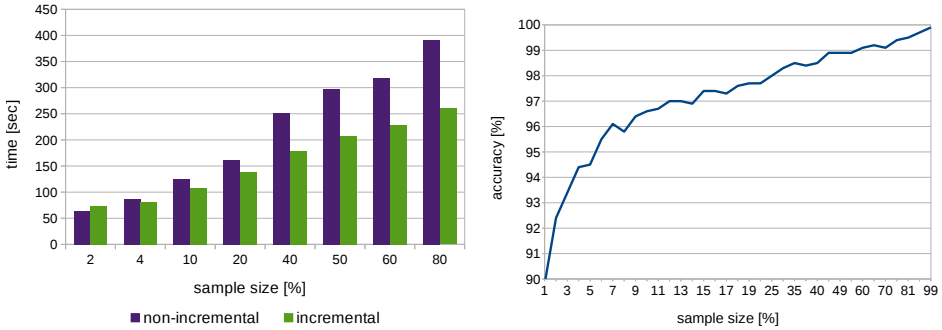

Figure 4: (a) Non-incremental vs. incremental sampling (b) Accuracy of a computation

Figure 3 shows the runtimes of the three sampling approaches. All tests are based on a word-count job on 3200 English texts of a size of 1.2 GB. As every split is read in the line-based sampling, it is slower than split-based sampling. The combination of both approaches[3] lies in between. When the sampling size is more than 50%, a full computation is faster than using sampling because the amount of intermediate data includes a few overhead and has to be shipped over the network.

In Figure 4a, a $(p/2)\%$ sample was doubled to $p\%$. When using incremental sampling, the old result can be reused which leads to a better performance than non-incremental sampling for p greater than two percent. We compared the results of the word-count sampling job with the actual count values and observed an accuracy of more than 90% for the words that occurred more than 5000 times (see Figure 4b).

Our assumption and the time-resources-accuracy triangle in the beginning of this paper were confirmed: For a sampling size of 10%, we saw that our incremental sampling approach is three times faster than a full computation at the expense of 3% inaccuracy.

6 Conclusion

The goal of the presented framework is to provide user-friendly sampling and extrapolation components for Hadoop which also enable an incremental sampling approach. These components can be used for sampling of arithmetic jobs. We made experiments on those jobs and we found out that there is a large speed-up when using sampling. This speed-up is even higher when using incremental sampling where former results can be reused to increase the sample size. Different from other sampling approaches, our framework does not need to modify the Hadoop framework. A user can simply enable input-data sampling and an automatic extrapolation of the results for existing MapReduce jobs with only a few modifications.

In future work, our sampling approaches can be used in an interactive data-visualization tool. Like in [Fis11], a user can see the results of a query in a chart. When using a small sample size, the chart can be displayed very fast. With the usage of incremental sampling,

[3] For a sampling size $s \in (0, 1]$, $\sqrt{s}$ of the splits are read, and within a split, $\sqrt{s}$ of the lines. For example, a 25% sampling ($s = 0.25$) is divided into 50% of splits and 50% of lines.

it can become more accurate over time. Currently, our framework does not give accuracy guarantees. We are currently working on a new version which displays the accuracy to the user.

References

[BL12] Mark A Beyer and Douglas Laney. The importance of'big data': a definition. *Stamford, CT: Gartner*, 2012.

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

[Fis11] Danyel Fisher. Incremental, approximate database queries and uncertainty for exploratory visualization. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 73–80. IEEE, 2011.

[GC12] Raman Grover and Michael J Carey. Extending map-reduce for efficient predicate-based sampling. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 486–497. IEEE, 2012.

[JPYD11] Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental recomputations in MapReduce. In *Proceedings of the third international workshop on Cloud data management*, CloudDB '11, pages 7–14, New York, NY, USA, 2011. ACM.

[LZZ12] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. *CoRR*, abs/1207.0142, 2012.

[Nor13] K Normandeau. Beyond Volume, Variety and Velocity is the Issue of Big Data Veracity. *Inside Big Data*, 2013.

[OR90] Frank Olken and Doron Rotem. Random sampling from database files: A survey. In *Statistical and Scientific Database Management*, pages 92–111. Springer, 1990.

[PBJC11] Niketan Pansare, Vinayak R Borkar, Chris Jermaine, and Tyson Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow*, 4(11):1135–1145, 2011.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. 1999.

[PJ05] Abhijit Pol and Christopher Jermaine. Relational confidence bounds are easy with the bootstrap. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 587–598. ACM, 2005.

[Sch14] Marc Schäfer. Samping mit inkrementellem MapReduce. Master's thesis, Technische Universität Kaiserslautern, 2014.

[SJHD14] Johannes Schildgen, Thomas Jörg, Manuel Hoffmann, and Stefan Deßloch. Marimba: A Framework for Making MapReduce Jobs Incremental. *IEEE International Congress on Big Data*, 2014.

[Sud76] Seymour Sudman. *Applied sampling*. Academic Press New York, 1976.

[Whi09] Tom White. *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.", 2009.