2013 IEEE International Conference on Cloud Computing Technology and Science

Block Sampling: Efficient Accurate Online Aggregation in MapReduce

Vasiliki Kalavri
KTH Royal Institute of Technology
Stockholm, Sweden
Email: kalavri@kth.se

Vaidas Brundza
KTH Royal Institute of Technology
Stockholm, Sweden
Email: vaidas@kth.se

Vladimir Vlassov
KTH Royal Institute of Technology
Stockholm, Sweden
Email: vladv@kth.se

***Abstract*—Large-scale data processing frameworks, such as Hadoop MapReduce, are widely used to analyze enormous amounts of data. However, processing is often time-consuming, preventing interactive analysis. One way to decrease response time is partial job execution, where an approximate, early result becomes available to the user, prior to job completion. The Hadoop Online Prototype (HOP) uses online aggregation to provide early results, by partially executing jobs on subsets of the input, using a simplistic progress metric. Due to its sequential nature, values are not objectively represented in the input subset, often resulting in poor approximations or "data bias".**

In this paper, we propose a block sampling technique for large-scale data processing, which can be used for fast and accurate partial job execution. Our implementation of the technique on top of HOP uniformly samples HDFS blocks and uses in-memory shuffling to reduce data bias. Our prototype significantly improves the accuracy of HOP's early results, while only introducing minimal overhead. We evaluate our technique using real-world datasets and applications and demonstrate that our system outperforms HOP in terms of accuracy. In particular, when estimating the average temperature of the studied dataset, our system provides high accuracy (less than 20% absolute error) after processing only 10% of the input, while HOP needs to process 70% of the input to yield comparable results.

***Keywords*—MapReduce; online aggregation; sampling; approximate results;**

I. Introduction

Real-time and near real-time large-scale data management and analysis have emerged as one of the main research challenges in computing. Modern large-scale analytics applications include processing web data, transaction and content-delivery logs, scientific and business data. Popular systems nowadays use massive parallelism and are usually deployed on clusters of inexpensive commodity hardware. However, even in such a highly parallel setting, analyzing big data sets takes a considerable amount of time. Many data analysis applications can tremendously benefit from using early approximate results, which can be available before job completion. Such applications can tolerate some inaccuracy in the results, while gaining significantly reduced response time. Example applications include search engines, estimation of Twitter trending topics, weather forecasts and recommendation systems.

Early accurate approximation techniques have been extensively studied in the context of relational databases. Popular works suggest using Wavelet transformations [1], [2], [3], [4], histogram-based approximations to return results with known-error bounds [5], [6] or sampling techniques [7], [8].

Hadoop [9], an open source implementation of Google's MapReduce [10], is the most popular and widely used big data processing framework. Due to its batch processing nature, though, it does not allow partial job execution. Even though existing research in approximate techniques for relational queries can serve as a starting point, applying such techniques to MapReduce-style processing frameworks is very challenging [11], [12], [13]. Apart from the aforementioned batch-style processing nature, the biggest challenge is posed by the unstructured data format that such systems need to analyze. Having sufficient knowledge on the input data structure, types and cardinalities of a query greatly facilitates the estimation of result accuracy. Unfortunately, this is not the case in Hadoop, where data is available as raw files stored in a distributed file system. Data is transformed into key-value pairs only after a job has been submitted for execution. Moreover, different applications can freely choose to interpret input data in different ways. MapReduce allows applications to specify arbitrary user-defined functions to be executed during the map and the reduce phase, thus making result estimation even harder.

These challenges have not discouraged researchers to try to integrate approximate result techniques in MapReduce. However, existing solutions are either far from matching the success of their database predecessors or are limited to specific operators and query templates [11], [12], [13]. A notable research work, MapReduce Online [14] integrates Online Aggregation into the Hadoop processing framework. MapReduce Online uses pipelining between operators, thus enabling partial job execution. However, input is read in a sequential manner and estimation accuracy is based on a simplistic job progress metric. Consequently, results are often highly inaccurate and are likely to exhibit data bias. Consider an application that computes the popularity of a search keyword over some specified period. It is often the case that some keywords, representing topics or events, become very popular over a short period of time or their popularity varies depending on the time of the day or day in the week. Two examples of keyword search distributions are shown in Figure 1, generated by Google Trends. If we estimate popularity using MapReduce Online, the result will be highly dependent on the selected period of time. For example, a partial job estimating the average popularity of the keyword "G1 league" of Figure 1(a), would sequentially process data from left to right, missing the values represented in the later spike, thus returning an inaccurate answer.

978-0-7695-5095-4/13 $31.00 © 2013 IEEE
DOI 10.1109/CloudCom.2013.40

(a) Distribution of the Google search "G1 league"

(b) Distribution of the Google search "Hadoop"

Fig. 1: Popularity Variation Keyword Searches over a 3-month Period

In this paper, we present a statistically profound early estimation technique as a layer over MapReduce Online. We propose a simple, yet efficient random sampling technique implementation, which significantly improves the accuracy of Online Aggregation. In order to overcome the challenge of the unstructured nature of data, our implementation performs the sampling before data is sent to the mapper tasks, as soon as it is organized in blocks. Moreover, in order to avoid slow random disk accesses, we propose in-memory shuffling of data blocks, thus efficiently achieving random sampling and reducing data bias. We evaluate our implementation in terms of performance and accuracy, using real-world datasets of various sizes and distributions. We assess the accuracy of early estimations and study its dependency over our introduced block-level sampling technique parameters. We show that our system delivers highly accurate results, while matching MapReduce Online in performance.

The main contributions of this paper are as follows.

- A novel, efficient, in-memory block sampling technique for MapReduce applications.
- A method for integrating the block sampling technique with a distributed large-scale processing framework.
- An implementation of the block sampling technique for the Hadoop Online Prototype (HOP).
- An experimental evaluation of the proposed technique, focused on performance and accuracy, using real-world datasets and applications.

The rest of this paper is organized as follows. Section II gives the necessary background for the paper. Section III presents our block sampling technique and gives details on the design, architecture and implementation of our solution. In Section IV, we provide evaluation results and comparison with Hadoop and the HOP. Section V discusses related work. We discuss conclusions and future work in Section VI.

II. Background

In this section, we briefly review the MapReduce programming model, the MapReduce Online framework and the Online Aggregation technique.

A. The MapReduce Programming Model

MapReduce [10] is a programming model for large-scale parallel data processing. In the MapReduce programming model, one simply has to specify an input path in the distributed file system, two user-defined functions, map and reduce, and an output path. Data is read from the file system, organized in blocks and shipped to parallel map tasks, where they are parsed into key-value pairs. Each parallel map task processes one block and applies the used-defined map function on each key-value pair, producing new pairs as output. The output pairs are then grouped by key and are sent to parallel reduce tasks, which apply the reduce function on each group. The result of each reduce task produces one file in the distributed file system. MapReduce rapidly became popular, as it ensures efficient and reliable execution of tasks across large numbers of commodity machines and successfully manages to hide the complex details of parallelization, data distribution, fault tolerance and load balancing from the user.

B. MapReduce Online

MapReduce Online [14] is a modified version of Hadoop MapReduce, a popular open-source implementation of the MapReduce programming model. It supports Online Aggregation and stream processing, while also improving utilization and reducing response time. Traditional MapReduce implementations materialize the intermediate results of mappers and do not allow pipelining between the map and the reduce phases. This approach has the advantage of simple recovery in the case of failures, however, reducers cannot start executing tasks before all mappers have finished. This limitation lowers resource utilization and leads to inefficient execution for many applications. The main motivation of MapReduce Online is to overcome these problems, by allowing pipelining between operators, while preserving fault-tolerance guarantees.

C. Online Aggregation

Online Aggregation [15] is a technique enabling interactive access to a running aggregation query. In general, aggregate queries are executed in a batch-mode, i.e. when a query is submitted, no feedback is given during the query processing time. Consequently, the accumulated results are returned only after the aggregation process is completed. The technique enables partial query processing, without requiring prior knowledge of the query specifications, such as types of operators and data structures As a result, users are able to observe the progress of running queries and control their execution (e.g. stop query processing in case early results are acceptable). Due to the lack of knowledge on query and data characteristics, Online Aggregation relies on random sampling to provide early results. The system is then able to provide running confidence intervals along with an estimated query result. A number of estimators for several types of running confidence interval computations has been proposed in [16]. Though the Online Aggregation technique never made a big impact in the commercial database products, it becomes newly relevant due to the rising interest in fast large-scale data processing.

III. The Block Sampling Technique

In this Section, we describe the block sampling technique in detail. We discuss the design goals of our implementation and the integration of the proposed technique with the MapReduce Online framework.

A. Design Objectives

Providing early accurate query results in big data frameworks is a very challenging problem. Our vision is to provide a simple, yet efficient and robust solution, independent of the specific processing engine. Our design is based on the following principles:

- **Statistical Robustness:** The system should be able to provide a valid estimate of the final job results at any given time during the job execution. We consider an estimate valid, if it has been computed over a uniform random sample of the entire job input dataset. Statistical properties should be guaranteed independently of the input data format or its underlying distribution.
- **Framework Independence:** The solution should be generic, with a simple design, easy to integrate with any big data processing framework using a block-based storage or file system, such as HDFS. Thus, we have chosen to realize the implementation as a thin independent layer, on top of the MapReduce Online stack.
- **Application Transparency:** Existing applications should be able to benefit from the proposed technique, without any modification. Furthermore, users should be able to select whether the new functionality will be active or inactive during execution.
- **Efficiency:** The introduced modifications should impose minimal overhead when compared to the original system. This includes both the MapReduce processing and data collection phases.

B. System Architecture

Before describing our system architecture, we first briefly review the MapReduce Online framework execution workflow. Like in standard Hadoop, in MapReduce Online, data is stored in HDFS [17] and is read in blocks, which are sent to map tasks wrapped into Input Splits. Before applying the map function, mappers parse the Input Splits into key-value records. The structure of the input data is described by the InputFormat method. This method selects a valid data reader, which reads an input file sequentially, parses it and returns the input records to the map tasks. The output of each map task is stored in an in-memory buffer and is periodically sent to the reduce tasks. The reduce tasks collect and merge partial results sent by mappers and the reduce function is applied to each record in the merged results file. Based on the initial job configuration, the output of a reduce task is materialized as part of the final job results or as a snapshot.

Standard map tasks access one data block and process it sequentially, one record at a time. In order to obtain a uniform random data sample, we chose to slightly modify the read operation and have each call to the record reader return a random record from the spill file. The implementation challenge here comes from the fact that common distributed file systems, like HDFS, are highly optimized for sequential reads, while random accesses are very expensive. Furthermore, the record reader should have sufficient prior information about the spill file, i.e., how many records each data block stores, which is not available at this stage of the workflow. In order to avoid expensive random disk accesses, we chose to process file splits in memory as shown in Figure 2. Each map task retrieves a number of record blocks from several data splits (1). The accessed record blocks are stored in memory (2), shuffled in-place in order to reduce possible data bias (3) and then sent to the map function (4). This way, block-level sampling reduces the I/O related overhead, which is inevitable when accessing multiple files in the distributed file system. We are convinced that the shuffling phase is also necessary, as it reduces the bias of collected data and guarantee the statistical properties of partial results (taken at any time during the execution). Finally, the input data structure remains consistent, so there are no MapReduce applications based constraints.

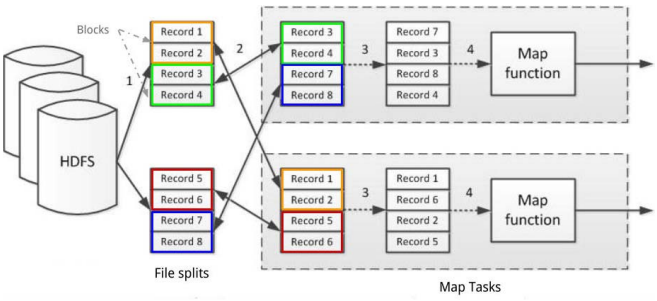


Fig. 2: The Block Sampling Technique Steps

C. Implementation

We have implemented a prototype of the proposed block sampling technique on top of HOP, that we call HOP-S. The source code is publicly available [1]. In this section, we discuss implementation details. Likewise the described design and implementation process can be split into two phases: bias reduction and sampling.

1) Bias Reduction: Data randomization is a necessary step in order to reduce the overall bias of the data and ensure that the randomness property is guaranteed at any time during the execution. We have implemented the data randomization phase as part of the initial reading process, after running several tests and verifying that it is an efficient solution. The functionality can be enabled by setting up a newly introduced MapReduce daemon parameter. The list of supplementary Hadoop framework parameters is given in Table I.

The bias reduction process is done in the following steps:

1) **Access phase.** During the initial step, the map tasks retrieve the file URI from the HDFS NameNode and access the specified file. By default, a file is read as a stream of bytes and parsed into lines of text (based on specified separator characters), text is then parsed into records and finally sent to the map user-defined

[1] https://github.com/vasia/HOP-S

TABLE I: Newly Introduced Configuration Parameters

Parameter	Description
io.file.shuffle (boolean)	Input records shuffling method for data bias reduction.
io.split.maxsubsplit (int)	Set the number of split files from which block-level sample will be created. Default value = 4.
io.split.insort (boolean)	If enabled, split files are shuffled before sampling method is applied.

function. We have introduced a new data collection method, which processes the blocks of the entire input split and stores them in the local task's memory. Further processing is delayed until the next described shuffling phase is completed.

2) **Shuffling phase.** The input data is stored in a custom in-memory data structure. Its design focuses on optimizing data randomization: a number of bytes (corresponding to a single line of text) retrieved from HDFS is stored as a tuple, with an additional, randomly generated, integer value in range of $[0...2^{31} - 1]$. The randomly generated integer value facilitates the shuffling: stored data is sorted according to the assigned integer value, thus returning randomly shuffled input data. Our tests showed that 512 MB of memory allocated to TaskTracker child processes is sufficient for the default 64 MB block size of HDFS.
3) **Processing phase.** Even though storing the shuffled data to disk would enable us to reuse the initial Hadoop data reading function, this step would introduce unnecessary I/O overhead. Therefore, we have developed an additional read method, which is used when the data randomization is enabled. It skips the data access and pre-processing steps and instead serves the line of text from local memory.

2) Block-level sampling: The block-level sampling requires several modifications of the map phase, mostly related with the storage system access. The processes of file sampling and processing are presented in detail next.

By default, each mapper retrieves and processes a single data block. Our aim is to force a map task to fetch and process data from multiple HDFS data blocks, thus emulating data sampling. The initial file information is acquired by issuing the request to HDFS. We introduce the RandomFileInputFormat which stores each file's information in local memory. Initial data splits are then divided into equally-sized blocks, to be processed by the separate map tasks. This approach requires a minimum number of random accesses (one-per-block), reducing the cost of HDFS I/O operations. We experimentally investigate the overhead of I/O operations in Section IV.

The number of blocks each split is divided into is determined in the MapReduce job configuration. In case the data split cannot be divided into equal-sized parts, the sampling process will determine the optimal sizes for each block ensuring that mappers receive a comparable workload, in terms of input size. Each map task will be responsible for processing a number of blocks from separate splits, equal to the number of parts each split is divided to. For example, if the user sets the *io.split.maxsubsplit* parameter to 6, map tasks will process 6 blocks of data from 6 separate input splits. Furthermore, the list of input splits, obtained in the data access phase, can be shuffled by setting the *io.split.insort* parameter, before the sampling process starts.

We have additionally developed a RandomFileSplit format, necessary for storing the complete block-level samples information, including URIs of the input splits, data offsets and block lengths. This information is used to query the the HDFS NameNode during the data access phase, thus enabling map tasks to sequentially open several input data streams.

Finally, we have implemented two additional InputFormat methods, in order to enable processing of the block-level samples, which are produced during the sampling phase. There are two main differences between the default and our introduced formats. First, our methods are able to access multiple input splits and read the blocks of data from each of them. This process is cyclic: as one data stream is consumed (certain number of bytes is read), a next one is initiated, until the whole sample is processed. Furthermore, block-level samples can start or end in the middle of a text line of the initial dataset, since the sampling phase relies solely on the stored file's meta-data. This issue is addressed with a simple rule: a task that receives a data block with the beginning of text line will process the entire line, otherwise, it will skip that fragment of the line.

IV. Evaluation

We have evaluated the block sampling technique and our implementation on top of the Hadoop Online Prototype, HOP-S. This section presents the results of our evaluaton.

A. Evaluation Environment and Datasets

Our setup consists of an OpenStack cluster, deployed on top of 11 Dell PowerEdge servers, each with 2 x Intel Xeon X5660 CPUs (24 cores in total), 40 GB of memory and 2 TB of storage. We ran all experiments using 8 large-instance virtual machines, each having 4 virtual CPUs, 8 GB of memory and 90 GB of disk space. Nodes run Linux Ubuntu 12.04.2 LTS OS and have 1.7.0_14 version JavaTM SE Runtime Environment installed. We configured Hadoop, HOP and HOP-S to use up to 17 map tasks and 5 reduce tasks per job, HDFS block size of 64MB and set the data replication factor set to 2.

For our experiments, we retrieved and pre-processed a number of varying size datasets. For the performance evaluation, we acquired weather data for several decades, available from the National Climatic Data Center ftp server [2]. Data is present for each year separately, (available years 1901 to 2013) and contains log files from a number of different weather stations. The logs were compressed and stored separately, per station. As a pre-processing step, we merged each weather station log files into a single yearly weather dataset. The size of aggregated log varies from 2 to 10 GB, depending on the year of measurement. In total, we acquired 21 aggregate log files consisting of 100 GB of data.

For the accuracy evaluation experiments, we prepared several datasets with different data distributions of size between 11 and 25 GB. The first dataset is an extract of the previously mentioned 100GB weather dataset and consists of 10 arbitrarily selected files of 25 GB size in total. The yearly

[2] ftp://ftp3.ncdc.noaa.gov/pub/data/noaa/

log files of the each weather station are merged sequentially, one after the other. For our second experiment, we reuse this dataset, after sorting it according to the date and time of the measurement, in order to create a dataset that would allow us to test our technique's bias reduction. Finally, we use a dataset consisting of the complete Project Gutenberg e-books catalog [3]. It contains 30615 e-books in .txt format, merged into a single 11 GB size file. Overall, we closely selected each of the described datasets in order to cover the wide range of data distributions and evaluate the accuracy of estimations returned by the evaluated systems.

B. Results

First, we present experiments to measure performance and investigate possible sources of overhead. Next, we evaluate the estimation accuracy of the proposed technique. We try to identify sensitivity factors and how they influence the results' accuracy.

1) Performance Evaluation: We first evaluate the influence of the snapshot materialization frequency, namely how often estimations will be materialized into HDFS, on the overall system performance. This is an overhead present in both HOP-S and HOP and it is usually nullified by the gains provided by pipelining [14]. Note that both bias reduction and sampling are disabled in this experiment. For this test, we set the maximum memory size available for map and reduce tasks to 512 MB per child. Our results are displayed in Figure 3(a). Each value is obtained by averaging the results of 3 to 5 executions over an extended period of time, to reduce the influence of cluster performance fluctuations. Overall, the results show that there is a moderate execution overhead (up to 35%) in the case of frequent snapshots. This overhead mainly occurs due to the reduce tasks being unable to process the output of map tasks during the period of snapshot materialization to the HDFS file system. Based on our observations, we recommend to set the snapshot materialization parameter to every 25% of processed input, as it has less overhead (about 15%) in comparison to executions with higher snapshot frequency settings. However, as we show in the next section, even earlier estimations of various MapReduce jobs can have high accuracy.

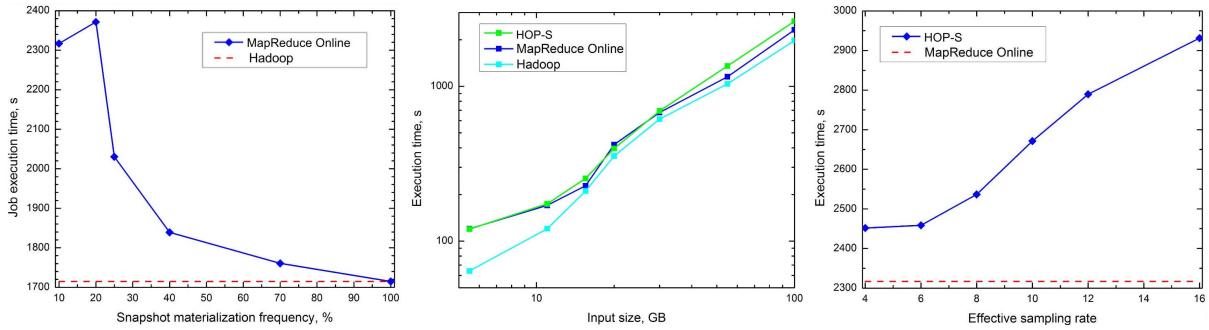
For the second experiment, we enable bias reduction and compare the performance of HOP-S to Hadoop and MapReduce Online. For this task, we prepared seven different size datasets, varying from 5,5GB to 100GB, which were used in the aggregate average temperature job execution. In order to minimize the influence of clusters performance variation, we ran multiple executions over each dataset and averaged the results, which are shown in Figure 3(b). We demonstrate the evaluation results of three systems: Hadoop (ver. 0.19), MapReduce Online and HOP-S with bias reduction enabled. The measured performance difference between the standard Hadoop framework and the other two frameworks is insignificant, after taking into account the overhead of frequent early result snapshots. Note that the snapshot frequency is set to 10% for all tests. We further investigate how the bias reduction process performance depends on system parameters. While the input data size is relatively small, bias reduction has little to no overhead. However, in the case of large inputs, the overhead noticeably increases, up to 20% over execution time with no bias reduction. Such results can be explained by the choice of system parameters, namely the relatively low number of map tasks, resulting in additional processing time being aggregated over a large number of sequentially processed data blocks each JVM executes a large number of map tasks sequentially). We strongly believe, though, that the shuffling phase is the main source of the inspected overhead, when input data records are arranged by the assigned random prefix. However, there is another source of potential overhead, not reflected in the present figure. Before sorting, all records are stored in the local memory of the processing nodes. The amount of required memory depends on the size of the HDFS data blocks. As a rule of thumb, we recommend increasing the available memory of each node to 4 times the blockSize.

In the last experiment, we measure the overhead of the sampling process to the overall job execution time. We use the 100 GB size input dataset, on which we ran an average temperature computation MapReduce application, using HOP-S and also the MapReduce Online framework. We performed several tests varying the block-level sampling rate. Results are shown in Figure 3. We observe that there is a noticeable overhead in comparison to job execution time over the MapReduce Online framework, which is growing with the number of block-level samples. We identify the following overhead sources: the sample generation process, the block-level samples movement across the nodes and the increased number of random accesses to HDFS, as there is a linear dependency between the number of additional random accesses introduced and the chosen sampling rate.

2) Accuracy Evaluation: In this section, we discuss experiments which evaluate the precision of early results, as produced by MapReduce Online, with and without using our proposed technique. We define accuracy as the absolute error between the early returned result of a partially executed job and the result returned after processing the entire dataset. We use the two previously described weather datasets with different distributions of values.

For the first experiment, we use a 25GB dataset of ten yearly aggregate logs of weather stations. We ran the MapReduce application that measures the average yearly temperature on Hadoop, HOP and HOP-S. First, we evaluate the variation of estimation accuracy over the number of block-level samples processed by each map task. The estimations were materialized and stored to HDFS every 10% of the processed input data. For this test we enabled both data sorting and sampling. Figure 4(a) shows the aggregated estimation results of 2 randomly selected yearly log files. The graph illustrates the absolute error range variation over the number of block-level samples given for each map task. Each box defines the 25th/75th percentiles, while the drawn line shows the minimum and maximum values of the measured absolute error. Each column consists of 8 values obtained after 10 up to 80% of input data is processed. We observe that the default setting of 4 block-level samples per map task is too conservative. When a low number of block-samples is processed by each map task, there is a higher probability that one or few of yearly datasets will be present to a lesser extent, thus the estimation accuracy might suffer. On the other hand, results obtained in case of 10 to 16 blocks per map are very accurate even after processing just 10% of

[3] http://www.gutenberg.org/



(a) Snapshot Frequency Overhead (b) Bias Reduction Overhead (c) Sampling Overhead

Fig. 3: Evaluation Results: Performance

the input. At later stages of the job execution, the estimations become relatively accurate independently of the number of processed blocks. Figure 4(b) shows the results for the same experiment ran on the sorted weather data. The main difference is the values distribution of the data. If this dataset is read sequentially (as in MapReduce Online), winter months will be processed first. In comparison to the unsorted weather data, there are only slight variations in the maximum values of the error range. However, the accuracy of estimations converges promptly. After 20% of processed input, estimations have very low errors. Based on these results, we set 10 samples per block for the rest of the experiments.

Figure 5(a) shows the results for the yearly average temperature calculation application, over varying-size processed inputs. The gray boxes correspond to execution of the application on the Hadoop Online system, while the colored boxes correspond to executions of the application on HOP-S. For Hadoop Online, some of the values have 100% error rate. This means that the system did not provide any estimation for a particular year. Also, all ten estimations were available only after 40% of the input data was processed. Furthermore, we notice that the system mostly processes blocks of one or few input files at a time, therefore resulting in a maximum error value between 40 to 70 percent of the processed input data which does not change. Overall, we conclude that Hadoop Online does not provide statistically meaningful results. Some of the early estimations might be reasonably accurate even at early stages of processing, while others required the whole input data to be processed in order to return a fairly accurate estimation. On the other hand, HOP-S gives promising results. Even after processing only 10% of the input, the maximum value of the absolute error is around 30%, with 25th/75th percentiles being less than 20 %. Furthermore, accuracy steadily increases with the amount of processed input. We also notice that even at 10% of processed input, average temperature estimations are available for all 10 years, whereas the Hadoop Online framework required 40% of input to do the same. Similar results were observed when running the same experiment over logs of the weather data, sorted by date. The results are illustrated in Figure 5(b).

In a final experiment, we evaluated the estimation accuracy of a top-100 words application MapReduce job, for which we used the e-books dataset described in Section IV-A. One important difference from the previous datasets is that it has a Zipfian distribution: a small number of words occur very frequently, while many others occur rarely. We measured the number of missed words (out of the top-100 final words) over the part of processed input data and observed that even after 10% of the processed input, both MapReduce Online and our designed system give fairly precise results, with up to 7 misses. As the segment of the processed input data grows, the number of missed words is reduced. We observed close to no difference in estimations of the HOP and our system. Due to the Zipfian distribution, the block-level sampling technique does not provide tangible benefit over sequential processing, however, it still offers a great advantage over traditional batch-processing, since complete processing is not necessary.

V. Related Work

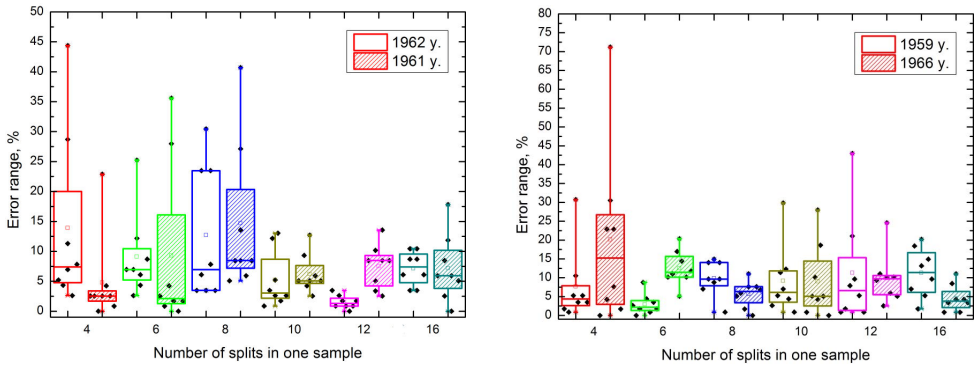
A. Large-Scale Parallel Data Processing Systems

Adapting the approximation techniques, previously used in databases, to large-scale, distributed processing systems is not straight-forward. Major challenges include the unstructured nature of data, the shared-nothing distributed environments such systems are deployed on and the need to support more complex analysis operations than simple aggregations.

Apart from MapReduce Online, which we briefly cover in Section II, there have been several other important contributions towards the direction of providing fast, approximate yet accurate results in large-scale MapReduce-like systems.

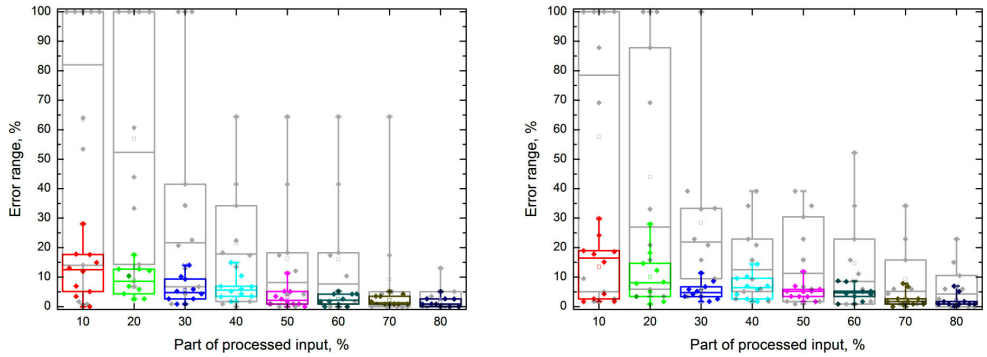
The EARL library [13] is an extension to the Hadoop framework and focuses on accurate estimation of final results, while providing reliable error estimations. It uses the bootstrap technique, which is applied to a single pre-computed sample. Consequently, numerous subsamples are extracted and used to compute the estimate. The accuracy of estimation can be improved with an expansion of the initial sample size and increase in number of used subsamples. EARL allows estimations for arbitrary work-flows and requires only minimal changes to the MapReduce framework. However, efficiency is very dependent on the job and the provided dataset properties and distribution. The bootstrap technique often proves to be very expensive and would result in even slower execution than running the complete job without sampling.

The BlinkDB [11] approximate query engine creates and uses pre-computed samples of various sizes to provide fast answers. It relies on two types of samples: large uniform random and smaller multi-dimensional stratified. Queries are



(a) Unsorted Weather Data (b) Sorted Weather Data (by date)

Fig. 4: Evaluation Results: Error Variation over Sampling Rate



(a) Unsorted Weather Data (b) Sorted Weather Data (by date)

Fig. 5: Evaluation Results: Error Range of Average Temperature Estimations

evaluated on a number of selected samples and an initial estimate is produced. BlinkDB is based on the predictable query column sets (QCS) model, so it assumes that a constant set of data columns, used for group or filter predicates, exist in the stored datasets. As a result, the system can estimate results of standard aggregate queries easily. More complex queries, including arbitrary joins, are currently not supported. In case the accuracy or time constraints of a query are not met, larger or smaller samples can be selected. The accuracy of various queries estimation depends on the composition and sizes of stored samples. However, samples creation (primarily stratified) is an expensive task and can take considerable amount of time. Consequently, the processing of newly arrived data to be included in the preceding samples can be delayed. On the other hand, our designed system can return continuously improving accuracy estimates without the additional pre-processing or the requirement to previously store samples. Further, it does not require pre-assumptions made in the predictable QCS model.

Another important work builds Online Aggregation [12] for MapReduce jobs, using the Hyracks execution engine [18]. The authors argue that Online Aggregation is newly relevant for the Cloud Computing cost model, as it can save computations and therefore money. They aim to adjust the classic work of databases in a MapReduce environment and mainly focus on facing the challenges that rise because of the shared-nothing cluster environment. In such a setup, where failures are also quite frequent, it is hard to guarantee the statistical properties of the partially processed input. They propose an operational model and a Bayesian framework for providing estimations and confidence bounds for the early returned results. However, in order to guarantee such properties, the system only supports applications conforming to a specialized interface and limited to the set of common aggregate functions.

Finally, Facebook's Peregrine [19] is a distributed low-latency approximate query engine, built on top of Hive [20] and HDFS. It supports a subset of operators and provides approximate implementations for some aggregate functions. A user has to explicitly use the approximate functions in their query and can get terminate it before the execution is complete. After termination, information is provided on the number of scanned records and possible failures that occurred during execution. In order to provide fast results, Peregrine uses one-pass approximate algorithms and an in-memory serving tree framework for computing aggregations.

VI. Conclusions and Future Work

The amount of data organizations and businesses store and process everyday is increasing with tremendous rates. In order to analyze data efficiently and at a low cost, the academic and industry communities have relied on data-parallelism and have developed distributed, shared-nothing processing architectures and frameworks, like MapReduce. However, even with these highly distributed solutions, query latency is still very high. Data analysts often have to wait for several minutes or even

hours to acquire a result. In many cases, however, quite accurate answers can be returned after only processing a small subset of the available data and great value can be extracted by only partial results. Several analysis applications can tolerate approximate answers to queries and highly benefit from lower latency. Such a functionality can also be used for rapid prototyping or pattern discovering in huge datasets.

Query result approximation is not a novel idea. There has been extensive research on the topic from the database community in the past. The data explosion that we are experiencing today, leaves us no choice but to reconsider approximation techniques, in the context of large-scale MapReduce-style systems, in order to reduce query response times. However, adoption of existing techniques is not straight-forward and proves to be very challenging. In the MapReduce world, data is not organized in tables or properly structured, but is often schema-less and stored in raw files. Moreover, analysis applications are usually much more complex than simple aggregations and can use arbitrary user-defined functions.

In this paper, we present the block-level sampling technique, which can provide a random sample of the provided dataset, without requiring pre-processing or additional storage space. We integrated block-level sampling with the MapReduce Online framework and we show that, together with an additional bias reduction technique, it can provide accurate estimations of results, without requiring a-priori knowledge of the query. In order to achieve that, data is sampled before the map stage and is shuffled in-memory, in order to introduce randomization. As a result, map tasks still can access data sequentially, avoiding the overhead of random disk accesses.

The evaluation of HOP-S shows superb results over the standard MapReduce Online framework, in terms of the early aggregate jobs estimations accuracy. Consequently, the execution time of most aggregate applications can be reduced noticeably, while still maintaining the high accuracy of the estimations. We demonstrate, that our system can estimate the average temperature of 100GB weather dataset with as low as 2% error, up to 6 times faster than a complete job execution time. However, we also show that the benefit varies and is highly dependent on the data distribution. For example, in the case of Zipfian distribution, MapReduce Online can return quite accurate results, even with sampling disabled. Nevertheless, we display that early estimations of the most frequent values in such datasets can be very accurate, thus the complete process of the input data is not always necessary. An interesting alternative metric that is subject of our future work, is the time needed to achieve a certain accuracy level.

Several open and interesting issues remain to be explored in the context of our work. First, we would like to explore the feasibility of integrating statistical estimators into our system, in order to provide error bounds or similar useful feedback to users. We are also interested in trying to automate the sampling process and optimize the strategy, based on system configuration parameters, such as block size and available memory. The automatic process would fine tune the sampling process to reach the best possible performance for the provided system configuration. Another direction would be to investigate alternative sampling techniques, or even wavelet-based early approximation techniques and explore the possibility for integration with large-scale processing frameworks.

ACKNOWLEDGMENT

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043.

REFERENCES

[1] E. Aboufadel and S. Schlicker, *Discovering wavelets*. Wiley-Interscience, 2011.

[2] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim, "Approximate query processing using wavelets," in *Proceedings of the 26th International Conference on Very Large Data Bases*. Citeseer, 2000, pp. 111–122.

[3] M. Garofalakis, "Wavelet-based approximation techniques in database systems," *IEEE Signal processing magazine*, no. 54, November 2006.

[4] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin, *Wavelets for computer graphics - Theory and applications*. Morgan Kaufmann, 1996.

[5] V. Poosala, V. Ganti, and Y. E. Ioannidis, "Approximate query answering using histograms," in *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 1999.

[6] Y. Ioannidis and V. Poosala, "Histogram-based approximation of set-valued query answers," in *Proc. of the 25th VLDB conference*, 1999.

[7] F. Olken and D. Rotem, "Random sampling from databases: a survey," *Statistics and Computing*, vol. 5, no. 1, pp. 25–42, 1995.

[8] J. S. Vitter, "Random sampling with a reservoir," vol. 11, no. 1. ACM, 1985, pp. 37–57.

[9] "The apache hadoop project page," http://hadoop.apache.org/, 2013, last visited on 1 May, 2013.

[10] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] S. Agarwal, A. Panda, B. Mozafari, S. Madden, and I. Stoica, "Blinkdb: Queries with bounded errors and bounded response times on very large data," in *ACM EuroSys 2013*, 2013.

[12] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie, "Online aggregation for large mapreduce jobs," vol. 4, no. 11, 2011, pp. 1135–1145.

[13] N. Laptev, K. Zeng, and C. Zaniolo, "Early accurate results for advanced analytics on mapreduce," vol. 5, no. 10. VLDB Endowment, 2012, pp. 1028–1039.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010, pp. 21–21.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *ACM SIGMOD Record*, vol. 26, no. 2. ACM, 1997, pp. 171–182.

[16] P. J. Haas, "Large-sample and deterministic confidence intervals for online aggregation," in *Scientific and Statistical Database Management, 1997. Proceedings., Ninth International Conference on*. IEEE, 1997, pp. 51–62.

[17] D. Borthakur, "The hadoop distributed file system: Architecture and design," http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf, 2013, last visited on 2 May, 2013.

[18] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 1151–1162.

[19] R. Murthy and R. Goel, "Peregrine: Low-latency queries on hive warehouse data," *XRDS*, vol. 19, no. 1, pp. 40–43, Sep. 2012.

[20] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.