Techniques and applications of early approximate results for big-data analytics

VAIDAS BRUNDZA

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2013
TRITA-ICT-EX-2013:185
Degree project in Distributed Systems

Techniques and applications of early approximate results for big-data analytics

Author: Vaidas Brundza
Supervisors: Vasiliki Kalavri, KTH, Sweden
Examiner: Associate Prof. Vladimir Vlassov, KTH, Sweden
Abstract

The amount of data processed by large-scale data processing frameworks is overwhelming. To improve the efficiency, such frameworks employ data parallelization and similar techniques. However, the expectations are growing: near real-time data analysis is desired. MapReduce is one of the most common large-scale data processing models in the area. Due to the batch processing nature of this framework, results are returned after job execution is finished. With the growth of data, batch operating environment is not always preferred: large number of applications can take advantage of early approximate results. It was first addressed by the online aggregation technique, applied to the relational databases. Recently it has been adapted for the MapReduce programming model, but with a focus to technical rather than data processing details.

In this thesis project we overview the techniques, which can enable early estimation of results. We propose several modifications of the MapReduce Online framework. We show that our proposed system design changes possess properties required for the accurate results estimation. We present an algorithm for data bias reduction and block-level sampling. Consequently, we describe the implementation of our proposed system design and evaluate it with a number of selected applications and datasets. With our system, a user can calculate the average temperature of the 100 GB weather dataset six times faster (in comparison to the complete job execution) with as low as 2% error.
Referat


I detta examensarbete vi översikt de tekniker, som kan möjliggöra tidig uppskattning av resultaten. Vi föreslår flera ändringar av MapReduce Online ram. Vi visar att våra föreslagna systemförändringar konstruktionsändringar besitter egenskaper som krävs för korrepta resultat uppskattning. Vi presenterar en algoritm för uppgifter partiskhet minskning och blocknivå provtagning. Därför beskriver vi genomförandet av vårt föreslagna system design och utvärdera den med ett antal utvalda applikationer och dataset. Vi visar att vårt utformat system kan återvända statistiskt djupgående resultat och kan anpassas av andra big-data-ramar. Med vårt system kan användaren beräkna den genomsnittliga temperaturen på 100 GB väder dataset sex gånger snabbare (i jämförelse med den fullständiga jobbkörning) med så lite som 2% fel.
Acknowledgment

I am deeply thankful to my supervisor Vasiliki Kalavri for a guidance and encouragement throughout this work. I would like to express my gratitude to examiner Vladimr Vlassov for co-supervising my work. Special acknowledgments go to everyone who supported me in this road of studies. I am very grateful to my family for their support and trust in me, without which I would not have done this work. Special mention goes to the ICT department, for a test environment and overall great atmosphere.

Stockholm, 03. July 2013

Vaidas Brundza
Contents

1 Introduction ........................................... 3
   1.1 Motivation ...................................... 3
   1.2 Contributions .................................. 4
   1.3 Structure of the thesis ......................... 5

2 Background ........................................... 7
   2.1 The MapReduce Programming Model ............. 7
   2.2 MapReduce Online ............................... 8
   2.3 HDFS ............................................. 10

3 Early approximate results enabling techniques ..... 13
   3.1 Histograms ...................................... 13
       3.1.1 Types of histograms ...................... 13
       3.1.2 Histogram-based approximation techniques 14
   3.2 Wavelets ....................................... 15
       3.2.1 One-dimensional Haar Wavelets ........ 16
       3.2.2 Multi-Dimensional Haar Wavelets ....... 17
       3.2.3 Wavelet synopses ....................... 18
       3.2.4 Approximate queries processing using wavelets synopses 19
   3.3 Samples ......................................... 21
       3.3.1 Uniform random samples .................. 21
       3.3.2 Block-level sampling ..................... 23
       3.3.3 Stratified samples ....................... 25
       3.3.4 Resampling techniques ................... 28
   3.4 Online aggregation ............................. 30
   3.5 Summary ......................................... 31

4 System design ......................................... 35
   4.1 Use-case scenarios ............................. 35
   4.2 Design objectives .............................. 37
   4.3 Architecture .................................... 38

5 Implementation ....................................... 44
   5.1 Data randomization ............................. 44
   5.2 Block-level sampling ............................ 45
       5.2.1 File sampling ............................. 45
## Contents

- **5.2.2 File processing** ............................................. 47
- **5.3 Results correctness** ........................................... 47
- **5.4 Summary** ..................................................... 49

### 6 Evaluation .................................................... 51

- **6.1 Test cluster setup** ........................................... 51
- **6.2 Experimental datasets** ....................................... 51
- **6.3 System performance evaluation** ................................ 52
  - **6.3.1 Early results snapshots influence to the overall systems performance** ........................................ 52
  - **6.3.2 The performance of bias reduction process** ........ 54
  - **6.3.3 The overhead of the sampling process** ............. 55
- **6.4 Evaluation of estimations accuracy** .......................... 55
  - **6.4.1 The average temperature estimation accuracy** ........ 55
    - **6.4.1.1 Aggregate log files dataset** ...................... 56
    - **6.4.1.2 Aggregate and sorted logs dataset** .......... 60
  - **6.4.2 Top 100 words count estimation accuracy** .......... 62

### 7 Related work ................................................ 66

### 8 Conclusions .................................................. 69

- **8.1 Future work** ............................................... 70
List of Figures

2.1 MapReduce processing pipeline ............................................. 7
2.2 The Architecture of the Hadoop MapReduce model .................. 9
2.3 The overview of MapReduce Online functional model ............... 10
2.4 The architecture of HDFS .................................................... 11

3.1 Classical equi-width (a) and equi-depth (b) histograms on continuous data ................................................................. 14
3.2 The non-standard decomposition of two-dimensional data. (a) Computing and distributing pairwise averages and differences. (b) Example decomposition of a 4 × 4 data array [10] ................................. 18
3.3 Possible strategies for processing query operators over the wavelet-coefficient domain [10] ......................................................... 19
3.4 Process of selection operation in relation (a) and wavelet-coefficient (b) domains [10] ............................................................... 20
3.5 Runtime phase of the dynamic sample selection architecture [7] ............................................................... 26
3.6 Example of a stratified sample associated with a group of columns [6] 28

4.1 The relevance of "G1 league" phrase in Google search over three months period ................................................................. 36
4.2 The relevance of "Hadoop" keyword in Google search over three months period ................................................................. 37
4.3 Job workflow in the Hadoop Online framework ....................... 38
4.4 Direct random sampling approach ........................................... 39
4.5 In-memory random data sampling .......................................... 40
4.6 The design of system with block-level sampling and bias reduction ............................................................... 41

5.1 UML diagram of newly introduced Hadoop classes .................. 48

6.1 The aggregate job execution time overhead dependency over the frequency of snapshot materialization. Used input dataset size: 100 GB. Baseline (no overhead) is execution time with no early snapshots materialization ............................................................... 53
6.2 The evaluation of three systems performance over various size inputs: our system (parameters: bias reduction enabled, 10% snapshot freq.), MapReduce Online (parameters: 10% snapshot freq.) and the Hadoop framework ............................................................... 54
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>The aggregate job execution time dependency over the effective input data sampling rate. Used input dataset size: 100 GB</td>
<td>56</td>
</tr>
<tr>
<td>6.4</td>
<td>The yearly average temperature estimations error range dependency on effective sampling rate</td>
<td>57</td>
</tr>
<tr>
<td>6.5</td>
<td>The error range of average temperature estimations of ten yearly data inputs, when data is not sorted. No sampling: base Hadoop Online</td>
<td>59</td>
</tr>
<tr>
<td>6.6</td>
<td>The error range of average temperature estimations of ten yearly data inputs, when data is not sorted and sampling rate is set to 10</td>
<td>59</td>
</tr>
<tr>
<td>6.7</td>
<td>The yearly average temperature estimations error range dependency on effective sampling rate over sorted log files</td>
<td>60</td>
</tr>
<tr>
<td>6.8</td>
<td>The error range of average temperature estimations of ten yearly data inputs, when data is sorted. No sampling: base Hadoop Online</td>
<td>61</td>
</tr>
<tr>
<td>6.9</td>
<td>The error range of average temperature estimations of ten yearly data inputs, when data is sorted and sampling rate is set to 10</td>
<td>61</td>
</tr>
<tr>
<td>6.10</td>
<td>The number of missed words in top100 words estimation dependency over the fraction of processed input data. No sampling: the standard MapReduce Online</td>
<td>63</td>
</tr>
<tr>
<td>6.11</td>
<td>The number of missed words in top100 words estimation dependency over the fraction of processed input data, when sampling rate is set to 10</td>
<td>63</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Complete Haar wavelet decomposition of vector $X$</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>The summary of introduced early results enabling techniques</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Related framework parameters</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Lines of code excluding comments</td>
<td>49</td>
</tr>
</tbody>
</table>
Real-time and near real-time large-scale data management and analysis have emerged as one of the main challenges in today’s Distributed Systems. Such systems are expected to be capable of processing large data inputs, growing up to Terabytes or larger, while meeting the given constraints. Modern large-scale analytics involves transforming user-clicks, transaction and content-delivery logs, scientific and business data, all of which might grow indefinitely and still require fast and relatively accurate processing. As the size of the input data grows, even the highly optimized systems can become a bottleneck, thus providing directions to Academia for further research. Due to the high requirements most of the nowadays systems are using massively parallel processing and are deployed on high-performance super computers, or more commonly, on clusters consisting of hundreds of commodity hardware.

Hadoop [1], an open source implementation of Google’s MapReduce [16], is widely known and the most used big-data processing framework up to date. This framework is typically used for performing batch-oriented jobs, which involves processing large amounts of data, however it has several limitations. An initial restriction appears due to the framework batch processing nature, as the system is limited in terms of jobs execution management. Thus, observation of partial running results is not possible in the Hadoop framework - results are available only after the job is complete. Even after employing largely parallel form of map-sort-reduce pipeline, execution of big-data processing jobs can take relatively long time. The need of addressing listed limitations has motivated researchers to develop several extensions for the Hadoop framework ([26],[14]). Our work focus on the early approximate result estimations, which can improve the frameworks manageability and reduce the execution time of various applications.

1.1 Motivation

Nowadays there are large number of tasks which can take advantage of early estimations. Consequently, some users are willing to do a trade-off between the accuracy
Introduction

of results and the execution process duration. Few well-known examples include estimation of trendy twitter topics and approximate weather forecasts. The effective web advertisement campaigns selection is also a very good illustration: usually it is sufficient to determine the top 10 common interests rather than having the exact results, most of which would be later discarded.

Users who want to retrieve faster, approximate job results within the MapReduce processing framework, have to write the specific application with an extra data processing step. The present MapReduce job have to address the size and properties of the underlying data distribution. If returned results are not satisfactory, application has to be rewritten and executed again. According to the user expectations and the underlying distribution of data, this process might have to be repeated several times, ultimately leading to significant overhead of data processing.

The integration of early accurate approximation techniques to relational database management systems (RDBM) is highly covered by Academia. Some scientific works suggest using Wavelet transformation ([38];[19]), others argue, that histogram-based approximations can return results with known-error bounds ([34];[24]). Finally, most of RDBMs employ some type of sampling techniques ([30];[37]). However, this is not the case when it comes to large-scale data processing frameworks - as of now, there are only handful of conducted researches, with no general examination of the available estimation techniques.

One of the notable recent researches attempts to integrate Online Aggregation technique into the Hadoop processing framework [23]. The results are of dual nature: it is able to execute a job online and return early estimations, however, these results are of limited use, as no attempt is done to ensure some statistical properties of the estimation. As a result, the Hadoop Online framework does not cover Online Aggregation specifications and its design benefits are not fully utilized, in terms of the early results accuracy.

1.2 Contributions

In this thesis we present our research on integration of early approximate results enabling techniques to big-data processing frameworks. One of the main goals of our work was to determine the answers to the following series of questions:

- Can we use an estimation technique to return fast yet accurate results?
- Which technique is more suitable for the large-data processing frameworks and why?
- What are the factors of estimation accuracy sensitivity?
1.3 Structure of the thesis

- What are the prerequisites for an effective estimation techniques integration into a batch-fashion big-data processing system, such as Hadoop MapReduce?

In order to get the answers to these questions, we studied most of the existing early estimation techniques and assessed their advantages and possible drawbacks in case of adaptation to the large-scale data analytics frameworks. We overviewed several real-world use-case scenarios, based on which we later determined the requirements of our system design.

We studied the Hadoop framework in depth and inspected its inner design structure. We show that it is possible to integrate statistically profound early estimation technique as a thin layer over the standard MapReduce Online framework. For evaluation part we prepared several datasets of various sizes and underlying data distributions. We assessed the accuracy of early estimations and its dependency over our introduced block-level sampling technique parameters. Furthermore, we evaluated the performance of our designed system. We observed a great boost in accuracy of two datasets estimations over the standard MapReduce Online framework. We show that usually our system delivers a considerable performance in terms of estimation accuracy and its statistical properties. With our system, a user can obtain an estimate of the average temperature of the 100 GB weather dataset up to six times faster (in comparison to the MapReduce job execution time) with as low as 2% error. Consequently, we strongly believe that our design, with some enhancements, could be integrated into the production level large-scale analysis frameworks.

1.3 Structure of the thesis

The remaining part of this thesis is structured as follows. In Chapter 2, we provide essential background details, which are required to follow our further work. In the following chapter (Chapter 3) we give an overview of the approximate early results enabling techniques: histograms (Section 3.1), wavelets (Section 3.2) and samples (Section 3.3). In Section 3.4 we present a concise description of the online aggregation technique, which is employed in our work. The subsequent Chapter 4 summarizes the design of our modifications and some of its use-case scenarios. We follow it with the implementation details (described in Chapter 5) and evaluation part, present in Chapter 6. The last chapter (Chapter 7) concludes our thesis work and provides insight to future research work.
Chapter 2

Background

2.1 The MapReduce Programming Model

MapReduce \[10\] is a programming model for large highly-parallelizable datasets processing. It is specifically designed to ensure the efficient and reliable execution of computations across large-scale clusters of commodity \[1\] machines. This framework is one of the main parts of open-source Apache Hadoop Project, an infrastructure for distributed computing \[11\]. MapReduce abstraction allows focusing purely on the problem solution via simple computations, while hiding the complex details of parallelization, data distribution, fault tolerance and load balancing addressed by the model. The architecture of MapReduce is inspired by the Map and Reduce primitives present in Lisp and many other functional languages \[16\]. These two primitives form a static data processing pipeline, showed in Figure 2.1.

In general, each job execution follows the several predefined steps in the MapReduce programming model. In the first step, input data are read as key/value pairs from the underlying data storage system. Usually the MapReduce model relies on distributed file systems. In the case of the Hadoop framework, data is commonly stored in the Hadoop Distributed File System (HDFS) \[9\], which is explained briefly in the

\[\text{Commonly available hardware available from multiple vendors}\]

\[\text{Figure 2.1: MapReduce processing pipeline}\]
following section. In the next step, parsed key/value pairs are grouped according to a key and passed to the parallel instances of the Map task. It is one of the two functions user has to define in order to execute the computations. The Map function is applied independently to every subset of records, expressed as key/value pairs. The output of the Reduce function can be generalized into several patterns: data reduction, data transformation and data increase. Data reduction is the most common use case of the MapReduce model - it allows reducing the input into some more compact results, e.g. machine learning algorithms [13] or the usual example of word count. The two other patterns are less common, however still can be used to solve the real-world problems, e.g. sorting very large input files such as the Terasort [31] or applying some of decompression algorithms. Next, the processed data is combined (the extra step, can be defined by the user), sorted and partitioned according to the framework defined partitioning function. This function ensures that all pairs with the same keys will be sent to and processed by the corresponding Reduce task. Likewise the Map function, the Reduce function is determined by the user for each MapReduce job and is applied to all in parallel. Each instance of the Reduce task creates a separate output file in the distributed file system, where its results are stored. Figure 2.2 illustrates the described stages of a MapReduce job [27].

One of the main advantages of the given MapReduce schema is its simplicity for the user, who needs to determine only the main Map and Reduce functions, leaving the handling of parallelization to the framework. However, this reduction in complexity has some drawbacks. First, the MapReduce framework lacks flexibility in terms of job and resources management. Each job must have only the single Map and Reduce functions. Furthermore, the Reduce function starts only after all parallel Map processes are finished, thus any results can be accessed only after the batch job is completed - no partial results are available. Moreover, if an algorithm requires multiple data processing steps (e.g. most of the machine learning algorithms), for each of the steps the separate Map and Reduce functions has to be defined and implemented as a separate job. This limitation can result in extra overhead due to using the distributed file system for results passing, as each Reduce function stores results there.

2.2 MapReduce Online

The Hadoop MapReduce Online [5] framework (HOP) is a modification of the MapReduce architecture that enables data pipelining between the Map and Reduce operators. The design of pipelines gives the following benefits to the MapReduce framework: a) improved system utilization b) wider range of possible MapReduce applications and c) the support of online aggregation technique - an extension over batch oriented processing of MapReduce programming model.
2.2 MapReduce Online

The performance improvement of the MapReduce framework is due to more efficient data movement between Mappers and Reducers. The Reducers in HOP are initialized as soon as job execution is started. In combination with data pipelining, it allows Mappers to push the data to Reducers through the initialized TCP channels as it is being produced. This process is further refined with an in-memory buffer, used by Map tasks to store the content of processed data. As soon as the buffer grows to a threshold size, a Mapper will register a file split to the TaskTracker. It is a node that accepts and monitors Map, Reduce and Combine operations and captures their output, required to track the job progress and a number of available task slots. In case a Reducer is not busy, the file split will be sent to it soon after the registration and removed from the buffer. However, if a Reduce task is not able to keep up with the production of Map tasks, the number of splits in a buffer will grow. In this case, each Mapper will merge its produced splits into one combined data split, which will be sent as soon as the Reducer is ready to process it. The evaluation results

\[\text{Transmission Control Protocol}\]
demonstrate that this modification can reduce the job execution time by up to 25%.

As Map tasks push the data to Reducers shortly after it was processed, the HOP
framework can support simplistic online aggregation by applying the Reduce function
to the received part of processed data. The online aggregation technique will be
further elaborated upon in Section 3.4. The output of a Reduce task can be seen as an
early results estimation of MapReduce jobs, or as authors call - a snapshot. We refer to
this solution as "simplistic" due to it providing only the superficial functionality of the
online aggregation technique. The current solution does not take into consideration
the possible bias in the processed data blocks and does not guarantee any statistical
properties. Furthermore, as the accuracy estimation is a relatively hard problem in
case of transparent queries, the HOP framework only monitors the job progress. A
progress score in the range of [0, 1] is assigned to each Map task and is sent to the
Reducer together with file splits. This score is used by Reduce tasks to determine
when a new snapshot should be materialized.

2.3 HDFS

The Hadoop Distributed File System (HDFS) is a system used by the Apache
Hadoop project and designed to run on commodity hardware. It is commonly used
for storing input and output files of the Hadoop MapReduce jobs. It is popular due
to suitability for very large datasets storage, with the guaranteed fault-tolerance and
high scalability.
2.3 HDFS

HDFS is based on the master/slave architecture. Each HDFS instance consists of the single NameNode and a number of DataNodes. The main responsibility of the NameNode is managing the file system: execution of the namespace operations (e.g. file opening or closing), mapping of the data blocks to DataNodes and storage of files metadata. The NameNode accounts for the main limitation of the HDFS file system - it determines the maximum number of separate files stored in HDFS. This limitation is present due to the NameNode storing file metadata in its memory. This design choice has been made due to the general assumption that HDFS will be used for storing small number of large files, rather than large number of small files.

DataNodes are responsible for storing blocks of data and serving them under request from the file system’s clients. The DataNode periodically communicates with the NameNode over the TCP/IP protocol to report that it is alive and is capable processing forward tasks from the NameNode. Furthermore, DataNodes can communicate between each other for the data replication and rebalancing. By default, the HDFS replication factor is set to three (separate replicas), however it can be specified by an application. The architecture of HDFS is shown in Figure 2.4. Large input data is stored as a number of blocks in the DataNodes. The typical block size varies between 64 - 128 MB and can be set in the HDFS configuration. The main advantage of using HDFS with the MapReduce programming model is its high throughput. Rather than moving data closer to where the application is running, the HDFS file system enables application to be aware of the data location and move itself closer to it. This allows reducing the network congestion and improving the overall throughput of the system.

![Figure 2.4: The architecture of HDFS](image)
Chapter 3

Early approximate results enabling techniques

In this chapter we overview the existing approximate early results enabling techniques for user defined queries and computations. Most of the surveyed techniques are used in nowadays Relational Databases or as a research topic for future updates. Furthermore, we evaluate their suitability to be used with big-data processing frameworks.

3.1 Histograms

The histogram is one of the most common ways to represent the distribution of data. Usually it is seen as a synopsis of a dataset, giving a simple to understand yet very informative graphical representation of it. Furthermore histograms can be used for data processing optimization, (e.g. for optimization of joins in a distributed environment), estimation of probability density and for other tasks. Nowadays they are widely spread among the commercial query processing systems for a graphical data representation. Few recent researches have also highlighted histograms suitability for the early results estimation. In particular, one research has been completed in Bell Laboratories, known as the AQUA approximate querying system. There, pre-computed histograms are used to estimate the result of join queries introducing just a relatively low storage space overhead.

3.1.1 Types of histograms

Histograms belong to the non-parametric group of statistical techniques. The main advantages of histograms over other statistical techniques are: a) the relatively low storage space requirements and b) accurate approximation of skewed distributions. These advantages make histograms an excellent technique for selectivity queries estimation, as described in [34]. Furthermore they claim that the accuracy of the
3 Early approximate results enabling techniques

Figure 3.1: Classical equi-width (a) and equi-depth (b) histograms on continuous data

estimation will depend on the type of histogram, more precisely the way values are grouped into its buckets.

During years of research multiple partitioning rules have been introduced for maintenance of histograms. Equi-depth and equi-width are the two most common types of histograms (example of each is displayed in Figure 3.1) used today. In equi-depth histograms all buckets are assigned the same number of values, while equi-width histograms have all buckets with an equal range of values. Each type of histograms have its strengths and shortcomings. Equivalent width histograms are relatively easy to compute and further maintain as new data is inserted. Furthermore, in case of large range of values in the initial dataset, it can result in a high degree of data compression. However the construction of such histograms requires a-prior knowledge of the dataset boundaries (minimum and maximum values) so that histogram buckets could be specified. In addition, this type of histograms may result in highly inaccurate estimations for some queries.

Alternatively to equi-width histograms, equi-depth histograms have a bounded worst-case estimation error. Each bucket in this type of histograms has the equivalent number of data points, resulting in the varying ranges of values. The worst-case error can be determined by selecting the amount of data points represented in each of the buckets. However, there is a price for the given guarantees, namely the construction cost of histogram. This process will require beforehand sorting of data, or complex quantile-computation algorithms will have to be used (as one described in [21]).

3.1.2 Histogram-based approximation techniques

There are two main histogram-based approximation techniques covered in scientific literature. One of them is the relational approximation technique [34]. The main concept of it is to "expand" in the histogram buckets present values until
3.2 Wavelets

Wavelets are mathematical tools for the efficient and theoretically profound decomposition of datasets. The result is a data transformation into the base approximation with additional detailed coefficients, used to reverse the process of decomposition. For the last few decades wavelet transformations were successfully applied in the area of signal and image processing [3]. In part inspired by this success, researchers have conducted multiple studies to determine the suitability of wavelets for the early results approximation [10], [19]. The general idea of these researches is straight-forward: apply the wavelet decomposition on the stored dataset or a continuous data stream.
and run a query on the obtained compact representation of the data. In a broad picture it can be seen as a 'lossy' data compression process.

### 3.2.1 One-dimensional Haar Wavelets

Following the previous works of the wavelets-based approximation techniques, we will focus on the Haar Wavelet Transform (HWT) process. This is one of the ground transformations, which uses recursive processing of the initial dataset. As we will show later, HWT process is relatively easy to compute, furthermore it can be used in a wide range of applications. In general, generated wavelets are classified into two types: one-dimensional and multi-dimensional. In this section we describe the one-dimensional data HWT process.

The number of dimensions in wavelet depends on the structure of input dataset. In general, we can consider the count of separate columns in the initial dataset as the number of dimensions in a resulting transformation. Suppose we have the input data, presented as a vector \( X \) with \( N = 8 \) data values \( X = [4, 4, 6, 0, 1, 3, 3, 7] \). The HWT of this vector can be computed as follows. First, we perform a pairwise averaging, to get a 'lower-resolution' image of the data with the following values: \([4, 3, 2, 5]\). That is, the first value of the given image is an average of two initial values in the original dataset (correspondingly 4 and 4), second value is average of the 3rd and 4th values and so on. As some may notice, part of the initial information is lost in this process. To address it, additional detail coefficients have to be computed. In case of Haar wavelets, these coefficients are the difference between the second of the averaged values and the computed pairwise average \([7]\). In our example it can be calculated as following: \([4-4, 0-3, 3-2, 7-5]\). As a result, no information has been lost in Haar wavelet construction process - original data can be easily reconstructed from the 'lower-resolution' representation of the data and the given detail coefficients. The full transformation, listed in Table 3.1, is achieved by applying the same steps recursively till result is a single value: representation of an overall average of the initial data values.

The Haar wavelet transform \( W_x \) of \( X \), is the single average coefficient, followed by the detail coefficients in the ascending resolution order. In our example \( W_x = [7/2, 0, -1/7, 3/2, 0, -3, 1, 2] \). The main advantage of Haar wavelet transformation

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Averages</th>
<th>Detail Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[2, 2, 0, 2, 3, 5, 4, 4]</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>[2, 1, 4, 4]</td>
<td>[0, -1, -1, 0]</td>
</tr>
<tr>
<td>1</td>
<td>[3/2, 4]</td>
<td>[1/2, 0]</td>
</tr>
<tr>
<td>0</td>
<td>[1/4]</td>
<td>[-5/4]</td>
</tr>
</tbody>
</table>

| Table 3.1: Complete Haar wavelet decomposition of vector X |
3.2 Wavelets

over the original dataset is a more compact representation, following from the fact that similar values in a dataset will result in the relatively small detail coefficients. These coefficients can be eliminated (treated as zeros) from the wavelet transform. Consequently, it will result in compaction of the data. Furthermore, since only small detail coefficients are eliminated, this process will introduce relatively low error in case of original data reconstruction, resulting in a very effective form of the 'lossy' data compression [34].

3.2.2 Multi-Dimensional Haar Wavelets

There are the two main HWT extensions used to transform the multi-dimensional (e.g. multi-column) data, namely the standard and nonstandard Haar transform. Both of them generalize the process of the one-dimensional transformation, described above, with small changes and are used in various applications, including approximation of query results.

The standard decomposition is a basic transform of a single-dimensional case. In the first step the data is ordered according to its dimensions (e.g. 1, 2, ..., d) and later the one-dimensional HWT is applied separately for each data "row" of a single dimension. Overall, this process takes d one-dimensional transforms along array of cells in a single dimension, where d is a total number of dimensions. However, this approach is not very efficient I/O wise. To address this bottleneck, several I/O efficient transform algorithms were introduced in [38]. The main idea of these algorithms is splitting the standard HWT decomposition into the hyperplanes (in chosen order of dimensions), such that the complete transform for the each hyperplane can be carried entirely in-memory, thus reducing the overall number of I/O operations. The resulting decomposition can be employed for the approximation of OLAP data cubes.

The nonstandard decomposition, on the other hand, employs the recursive approach. In short, the nonstandard HWT alternates between dimensions during steps of a pairwise averaging and differencing: a single step is performed for each one-dimensional row of values along the dimension n, for each of n = 1...d dimensions. These steps are then repeated recursively on a single quadrant which contains the averages from every dimension. One of the ways to visualize this process is to think of \(2 \times 2 \times ... \times 2(= 2^d)\) hyper-box being shifted across the data array, as a pairwise averaging and differencing is performed, distributing the results to the locations of the HWT array, with averages stored in the "lower-left" quadrant of the transform array and, finally, applying the same steps recursively on that quadrant [10]. The steps of the non-standard decomposition and a simple example are displayed in Figure 3.2.

\(^1\)Input/Output
In Figure 3.2(b) we show an example of the non-standard HWT process. Initially we have a $4 \times 4$ data array (3.2(b.1)) for which we will apply the decomposition process. The results of initial averaging and differencing step are displayed in Figure 3.2(b.2). The averages of each $2 \times 2$ hyper-box are stored to the 'lower-left' quadrant in the wavelet transform array, while the three remaining quadrants accumulate the detail coefficients of transform, as shown in 3.2(b.3). Further, the same process is recursively applied to the 'lower-left' quadrant of wavelet transform array, resulting in the final wavelet transform with an average value of 2.5 and a number of detail coefficients, demonstrated in Figure 3.2(b.4).

### 3.2.3 Wavelet synopses

The goal of employing the Haar wavelet transform technique is to build a compact representation of the original dataset (e.g. vector X in earlier example). This kind of data representation in literature is usually called a synopsis, or in this case the wavelet synopsis. Usually the number of values represented in synopsis is much lower than in the target dataset. However, in exchange to data compression, the integrity of the initial data is reduced, thus lowering the accuracy of the restored dataset.

The essential step in data compression is a coefficient thresholding process, which allows determining and storing the "best" coefficients of the Haar wavelet decomposition in order to minimize the error of the approximation. The traditional approach to
pick coefficients randomly can introduce several important drawbacks, such as bias in the reconstructed data and high variance of data approximation quality. Most of the works on wavelet-based datasets reduction and approximation are based on the conventional thresholding scheme that greedily retains only the largest HWT coefficients in an absolute normalized value. This rule is proven to be optimal for minimizing the overall mean squared error in the data compression [36].

### 3.2.4 Approximate queries processing using wavelets synopses

The wavelets synopses allow to perform fast approximate processing of both aggregate and non-aggregate SQL-like queries. The "render-then-process" is one of the processing strategies suitable for wavelet synopses. It consists of several steps. Initially the execution engine determines synopses over which the query will be executed. Next, each selected wavelet-based synopsis is rendered (we denote this process as render \( W(T_k) \)) into the approximate dataset for a given relation. Lastly, all operators of the query are processed over the resulting approximate sets of data. However, this strategy is not very efficient: the number of data values in the approximate sets can be similar to the number of data values in the original relation sets. This means that the execution times of the query can be relatively similar. As a result, "render-then-process" strategy can be inefficient in terms of early results.

The inefficiency issue can be addressed by modifying the query algebra of the execution engine. Change of the query execution domain is an essential adjustment: engine should use wavelet-coefficient domain instead of relational data domain, used initially. The motivation behind query algebra modification is usually compact size of the wavelet-based synopses. As a result, the cost of query execution can be drastically decreased. The main change is in the algorithms of query operators \( op \) (such as select and average), which take wavelet coefficients as an input and produce a result set of the output coefficients. Both semantics of query processing are displayed in Figure 3.3. In this illustration the preceding strategy takes 'right-then-down'
Chakrabarti et al. [10] describe several algorithms of various SQL-like operators to be used with the wavelet-based synopses. We will give a brief overview of the select (selection) operator algorithm. The select operator filters out the sub-set of the wavelet coefficients in the initial synopsis $W_T$, which do not fall into the $k$-dimensional selection range, and thus do not contribute to the final value of the query. The process of this operator application is illustrated in Figure 3.4. The algorithm of select operator process over the wavelet-coefficient domain takes several steps. In the initial stage, the algorithm selects coefficients which overlap with a $k$-dimensional selection rectangle. These coefficients are then modified to have a valid sign information and reflect the boundaries of the selection rectangle. To be more exact, coefficient sign information have to be modified in case the selection range along a single dimension $D_i$ is completely contained in a single sign-vector range (either low(−) or high(+)). In either case, the sign-vector is modified to contain in the selection present single sign and the coefficient’s sign-change is set to its leftmost boundary value [10].

As a result, the wavelet-based approximation techniques can be well integrated into the relational databases. However, there are several limitations we have to consider. First, similarly to the histogram-based techniques, they require the pre-processing step, which has to be done offline, ahead of any query execution. Due to the recursive nature, pre-processing of large datasets can take relatively long time. To emphasize the efficiency, the query execution engine of big-data processing frameworks has to be modified: queries have to be translated to operate in wavelet-coefficient domain. Furthermore the returned coefficients have to be post-processed to estimate the final answer of issued queries. In case of the Hadoop framework, MapReduce applications would also require some modification, i.e. the Map function has to be applied on wavelet-coefficients.

Nevertheless, we strongly believe, that this technique might be a future direction for a big-data processing frameworks. It has several highly desirable features, such as a ‘lossy’ data compression and compatibility with most of SQL-like queries. Combined
with an adaptive selection of wavelet-coefficients, the wavelet-based estimation tech-
niques can be the foundation of very effective and interactive database system models. However, this would require extensive research in several topics (e.g. improvement of wavelet-transform algorithms or the returned results post-processing efficiency). Due to the limited scope and time restrictions of this project, we opt to delay the further design and integration research for a future work.

3.3 Samples

A sample represents a subset of data values selected from the initial dataset via some stochastic process, usually referred to as sampling. Sampling is arguably the longest studied method for early results approximation in databases, with the earliest paper being published in 1984 [33]. Nowadays most of the relational DBMS by default support at least one of the sampling methods, shortly examined in [30].

The main reason of long sampling techniques research history is its simplicity. The idea is quite straightforward: given a collection of records and a query with an unknown result we aim to estimate, a compact representation is obtained by selecting a number of the initial dataset records according to some strategy. Next, some of statistics, such as variance and standard deviation, can be computed over the generated sample. Lastly, the gathered statistics are used to estimate the results of a given query and its accuracy.

3.3.1 Uniform random samples

The random records selection is one of commonly used strategies. In this case, if the probability density function $p(q)$ of the collected sample is uniform, then sample creation process is called uniform random sampling, while its output - uniform random sample. It is the most common type of samples used in research studies and commercial products. There are several reasons behind its popularity. First, the uniform random sampling has a wide range of estimators used to estimate the answer to an extensive variety of arbitrary queries. Such ubiquity is due sample being a partial representation of the initial dataset, thus the same query-algebra can be applied without any additional modifications. Furthermore, the sampling-based estimation is not sensitive to the number of dimensions (e.g. columns in columnar storage) in the initial dataset. However the main reason of this technique’s efficiency is its efficiency. Sampling does not rely on a pre-constructed model, which has to be pre-processed before arbitrary queries execution. Often a sample can be generated after the user defined query has been present (namely, online processing), without introducing a noticeable delay. Further in text we will refer to uniform random samples as random samples.
There are the two types of random sampling algorithms, namely sampling with replacements and sampling without replacements. The main difference between them is the content of the resulting sample. The sampling with replacement algorithms sample each value independently, meaning that the single value in the original dataset can be represented several times in the generated sample. Mathematically speaking, this means that covariance between the values in a sample is equal to zero.

On the contrary, in case of the sampling algorithms without replacements, a single value in the initial dataset can be represent only once in its sample, meaning that the covariance between the values is not zero anymore. As a result, it can complicate the computation of sample statistics. However, in case of big-data, the difference between both types of algorithms becomes relatively insignificant due to the covariance converging to zero as the size of dataset increases.

During the years of research a number of sampling algorithms has been present. Most of them require beforehand knowledge of some properties of the initial dataset, such as the total number of values or minimum and maximum values in the dataset \[ 18, 25 \]. This requires at least a single scan over the whole dataset before the sampling algorithm can be applied, which impacts the overall efficiency of the process. Even worse, dataset might be of indeterminate length e.g. when data is being streamed to the server, then it would not be possible to determine the size of the dataset. The problem has been addressed by introducing the new family of sampling techniques, characterized as the reservoir algorithms. In general, a reservoir algorithm creates a sample with only a single sequential pass over the dataset without prior knowledge of its size. This property is invaluable for large-data processing frameworks, such as BlinkDB \[ 6 \], where it is used for online creation of the uniform random sample. Jeffrey S. Vitter in \[ 37 \] has introduced the three highly optimized reservoir algorithms for random samples without replacements selection. We will overview one of them further in this paragraph.

Essentially, the idea behind any reservoir algorithm is to choose a sample of size \( \geq n \), from which a random sample of size \( n \) can be obtained. The process starts similarly - by placing the first \( n \) records of the initial dataset into a 'reservoir'. The 'later' records are processed sequentially and are either selected as candidates to be placed or are skipped. At the end of the sequential processing, random sample must be selected from the obtained values. However, it still might be relatively large, thus highly optimized algorithms tend to focus on keeping the size of the 'reservoir' same as the required sample size \( n \). See below for an example Algorithm \[ \]  

There are several important parts in the given example. First, the initial number of records is placed into the 'reservoir' which is the same size as the one of the expected random sample. The remaining part of the dataset is scanned sequentially and for each read value the random index between 0 and number of processed records is generated. If it is lower than the size of reservoir, previously stored record with the
3.3 Samples

Algorithm 1 Simplified pseudo-code of reservoir random sampling algorithm

```
for j -> reservoirSize do
    readNextRecord(C[j]);
end for
procRecords ← reservoirSize;
while not eof do
    procRecords++;
    randPos ← (int) (procRecords × random());
    if randPos < reservoirSize then
        readNextRecord(C[randPos]);
    else
        skipRecords(1);
    end if
end while
```

same index is overwritten by the newly read record (line 8 in the pseudo-code 1). Otherwise, the record is skipped and the processing is continued, till the end of file (eof).

However, the random sampling has few drawbacks. Most notably, the random sampling in general is sensitive to data skew and outliers. For example, we have a dataset consisting of values: ⟨5, 8, 3, 9, 14, 17, 10, 10⟩. Then any estimate for the sum of the dataset values will have an arbitrary low accuracy. In case the sample would miss the record with value of 10, then as a result, the sample would underestimate the final sum of dataset and in all other cases it would be overestimated by approximately a factor of 1/p where p is the data sampling rate (p × 100)%. Furthermore, random samples are not very suitable for highly selective queries answer approximation, which depends upon only minor number of values from the large dataset. To address listed shortcomings, some advanced sampling techniques have been introduced in the field. We will cover few the most relevant in the further paragraphs.

3.3.2 Block-level sampling

The block-level sampling is quite similar to the uniform random sampling techniques. The main difference between them is a process of record collection. Rather than drawing one random value at each step (it is a common practice of the random sampling algorithms), the block-level sampling algorithms draw a number, namely block, of values.

The considerable advantage of the block-level sampling is its superior efficiency over the uniform random sampling. We will use an example to explain it. Assume we

\[\text{An outlier is a data value numerically distant from the rest of the values}\]
have a large dataset stored in the persistent storage (e.g. HDDs), which we want to sample. If the uniform random sampling algorithm would be applied, in the end it will require either a scan of the whole dataset (as explained in the reservoir algorithms section), or the high number (same as the wanted size of sample) of random accesses to the persistent storage. If we take the top enterprise class HDD based storage systems, each random access\(^3\) can take around 3 (ms) to complete, thus limiting the number of drawn values to about 10000 records/minute/disk. Consequently, the random sampling is efficient only when applied to the relatively small datasets (when whole dataset can be store in the memory) or when a small sample is required. The block-level sampling minimizes this drawback by retrieving a block of values after the single random access, thus effectively raising the rate of sample collection to \(\text{blockSize} \times 10000\) records/minute/disk.

To some extent, the block-level sample is the same as the uniform random sample, which enables in both cases using the same estimators. As a result, the block-level sampling technique is quite widespread in the relational storage products. However, there are only few research works, which address the properties of the block-level sample. One of the reasons behind it is the block-level sampling properties dependency on the layout of the stored dataset. In case the layout of a dataset is random, the block-level sample is as good as the uniform random sample. However, other extreme is when the values of layout are completely correlated (biased), making block-level sample prone to the significant error if used to estimate the answer to a given query. Usually, the layout of actual datasets is somewhere in between, so the block-level sampling can be relevant in certain cases. One of the past research work \([12]\) addresses the problem of histograms construction from the block-level samples. More recent work \([11]\) has also addressed the problem of the number of distinct-values estimation. We will overview it next.

The naive approach for the distinct-value estimation from the block-level sample is to consider it as the uniform random sample and apply the estimators which perform well with uniform random samples. However, this approach can return highly inaccurate results. Consider the block-level sample which has multiple occurrences of a single value. By observing it, we have a fact about a value being present in a block, but it is not an indicator, that this particular value is frequent in the initial dataset. To generalize, the occurrence of the same value across blocks is a good indicator of its abundance, while multiplicity within a single block is a misleading indicator. This has been addressed by several algorithms, which take into account the possible correlation of the block-level sample.

One of them is the \(\text{COLLAPSE} \ [11]\) algorithm, which focuses on the estimated results accuracy (in form of bias and error of the estimator) improvement when the block-level sample is used with the uniform random sampling based estimators. The idea of it is quite simple - rather than using the whole block-level sample, the algorithm processes it to obtain an 'appropriate subset', which would have

\(^3\)We assume that random access will require the seek operation.
3.3 Samples

properties similar to the ones of uniform random sample. The algorithm is described as Algorithm 2.

Algorithm 2 Distinct-value estimation with block-level samples

Input \( q \): Block-level sampling fraction.

1. Sampling step: Take a block-level sample \( S_{blk} \) with sampling fraction \( q \).

2. Collapse step: In \( S_{blk} \), collapse all multiple occurrences of a value within a block into one occurrence. Result is a sample \( S_{coll} \).

3. Estimation step: Use \( S_{coll} \) with an existing estimator as if it were a uniform-random sample with sampling fraction \( q \).

Essentially, the given algorithm removes multiple occurrence of the same value within a block, thus minimizing the probability of overestimating the frequency of a single value in a dataset, which translates into the higher accuracy of the distinct-value count estimation.

The purpose of the previous algorithm overview is to demonstrate how relatively simple it is to process the block-level sample to be used as the uniform random sample. This property makes block-level sampling as a good candidate to be used for big-data analytics. Consider the HDFS file system: there, a dataset is stored in a number of data nodes, which are running on the commodity hardware. Usually the size of stored data exceeds the amount of memory available for data processing, making the random access time of storage disks one of the main bottlenecks for the sampling techniques. In this case the block-level sampling can neglect the impact of it.

3.3.3 Stratified samples

The stratified sample \cite{28} consists of several independent subgroups (stratums) of the sampled values. The strata are mutually exclusive: any value can be assigned only to a single stratum. This property is very valuable in case of the multi-relational or highly skewed datasets. To display stratification advantages, we will use an example with a small dataset:

\[\langle 8, 8, 14, 9, 10, 10, 10, 9, 10^10 \rangle\]

We can arrange these values into the frequent and infrequent value groups. Assume that to be frequent, the value should appear at least twice in the initial dataset. This way, the frequent group will include values \( \langle 8, 9, 10 \rangle \) while \( \langle 14, 10^{10} \rangle \) will belong to the infrequent values group. These two groups can be sampled independently to get the stratified sample, containing uniform-random sample of the frequent values group.
and every value of the infrequent one. As a result, the final sample will over-represent the rare group of values, as following:

\( \langle 8, 9, 9, 14, 10^\rangle \)

Note, that this sample will provide more accurate estimation (if compared to uniform random sample) of the various queries results (e.g. average or sum). In research works the stratified samples are closely related to the multi-columnar storage systems, which are able to store and process the multi-dimensional datasets. Most of the works focus on join, group-by and similar queries, which require the values from several data columns. In this case, using the uniform random samples of each accessed column can return a low quality approximation. One of the reasons is the low number of values in the resulting sample (e.g. when two columns are joined) even if the initial join selectivity is relatively high. Furthermore, the sample combined from several uniform random samples is no longer uniform random. Stratified samples, on the other hand, can provide relatively accurate estimations. However, it requires a pre-processing step and additional storage space, which can grow indefinitely due to a wide range of possible group-by queries. As a result, it is important to find the optimal (the lowest cost) stratified samples creation and maintenance strategy. We will overview several systems which employ the stratified sampling techniques.

The Aqua system [5] introduces the one-per-relation join synopsis heuristic, with an aim to minimize the average relative error over a set of join-queries. The system is designed to be used on top of commercial DBMS and maintain the synopses of the stored data. These synopses later are used to estimate an answer to queries. The main contribution of the paper is a demonstration that a small number of computed stratified samples can be used to obtain random samples for most of joins in the database schema. The strategy of samples creation is based on the shared error bounds property: for COUNT, SUM and AVG, commonly-used Hoeffding and
Chebyshev bounds are both inversely proportional to $\sqrt{n}$, where $n$ is the number of tuples in the join sample \[^{[5]}\]. The goal is to cover the largest fraction of the foreign-key join and sole relation queries for a given amount of memory. As a result, they propose the three heuristic space-allocation strategies. The maintenance of join synopses is based on a relatively simple algorithm: if the new value $\tau$ corresponding to relation $u$ is added to the uniform random sample $S_u$, then the corresponding maximum key join tuple $\tau \Join \tau_2 \Join ... \Join \tau_k$ is stored to the join synopsis $I(S_u)$. The computation of the join tuple will take up to $k-1$ lookups. If the size of resulting join synopsis exceeds its target size, then tuple $\tau'$ is randomly selected and removed from $S_u$ along with matching join-tuple from $I(S_u)$. Similar steps are performed in case of tuple $\tau'$ removal from the relation $u$. Evaluation of the Aqua system indicates that this approach is quite effective in aggregate join queries estimation.

Another group of query results approximation techniques, called as dynamic sample selection \[^{[16]}\], focuses on the data analysis efficiency. In this work authors introduce the dynamic sample selection strategy, which focuses on selecting the most appropriate small sample from a larger pre-processed sample. Usually such a sample is collected by augmenting the uniform random sample with a small number of additional tuples. This technique is identified as the small group sampling \[^{[7]}\]. It is designed for estimating aggregation queries with "group-by" clauses. Small group sampling uses the large uniform sample for the overall estimations and some small-sample tables for the infrequent values, which are stored without down-sampling. The appropriate sample is selected during the runtime, based on incoming query predicates. The overview of this process is displayed in Figure 3.5.

The selection is done according to the meta-data, stored along with the pre-processed samples. It includes the list of columns covered by the small-sample tables (which are nothing else, than a stratified samples). In addition, there can be a requirement to adjust (modify) the initial query, so it could take into consideration the difference in sampling rates of the selected samples. For instance, in \[^{[39]}\] authors describe the approximate relational query algebra which returns monotonically improving results as more of the input data is processed. This step allows the system to use the multi-level hierarchy of samples, e.g. sample 100% of rare values, 10% of relatively frequent values and 1% of very frequent values. As a result, there can be a large number of samples stored in the system, from which the most appropriate one will be dynamically selected during the query execution.

The recently introduced BlinkDB \[^{[6]}\] approximate query engine combines some of the functionality from both previous systems, and is optimized to be run on large volumes of data. The main ideas of this engine are: a) an adaptive optimization framework, employed for both uniform random and multi-dimensional stratified samples creation and maintenance over time and b) a complex dynamic sample selection strategy. The stratified sample creation process in BlinkDB system is formulated as an optimization problem: given a requested queries history and the available storage size, the system has to choose the collection of stratified samples which would give a good coverage
3 Early approximate results enabling techniques

Figure 3.6: Example of a stratified sample associated with a group of columns [6]

for the similar future queries. To address this problem, authors assume, that the data columns used as a query predicates (such as WHERE, GROUP BY and HAVING) do not change over time. After the optimization process, when the subset of the most relevant columns is selected, its represent multi-column stratified sample is created as showed in the Figure 3.6.

As visualized, the stratified sample will include all records of a given set of columns, up to the defined maximum frequency value. Very frequent values will be down-sampled: all appearances above the maximum frequency will be ignored in the stratified sample. However, to ensure the maximum accuracy of the estimation, stratified samples include the effective sampling rate of the frequent values (in form of meta-data). As soon as both stratified and uniform random samples are collected, the BlinkDB system is ready to run interactive queries on it, with a user specified time or relative error constraints. This is a dynamic sample selection heuristic, based on query execution of very small subsamples (which fit into the memory of the system). The gathered results along with query’s statistics (the underlying distribution of its input, selectivity and complexity) are used by the BlinkDB system to extrapolate the response time and relative error dependency over the size of samples and construct the Error Latency Profile [31] of a query. This profile is then used to dynamically select the most relevant sample and its size for the given query (with or without additional user constraints). As a result, the BlinkDB query engine can execute most of the aggregate queries with additional selectivity predicates over massive datasets (e.g. several Terabytes) in a very short time span (matter of seconds).

3.3.4 Resampling techniques

Re-sampling is a powerful statistical method used for several important tasks, such as an estimation of wide range of statistics or sample-based data model validation. The re-sampling process involves creating multiple subsamples from the initial sample, each of which can be analyzed in the same way to determine the variation of statistics. Jackknifing and bootstrapping are the two most commonly used resampling techniques. Both of them are used to determine the statistics of an original estimator, which is retrieved from the initial sample. Jackknifing [35] originally has been introduced to
3.3 Samples

Algorithm 3 Standard Deviation estimation of an estimator using bootstrapping method [16]

\begin{verbatim}
initSample ← getSample(data);
initEstimator ← estimateParam(initSample);

for j → bootSamples do
    subSample ← drawSubsample(initSample);
    subParams[j] ← estimateParam(subSample);
end for

sampleMean ← avg(subParams);

for subParam → subParams do
    varSum += (pow(subParam − sampleMean, 2));
end for

sampleVariance ← (1/(bootSamples − 1)) * varSum;

sampleStdDev ← sqrt(sampleVariance);

return (initEstimator, sampleStdDev);
\end{verbatim}

estimate the bias and variance of an estimator. The results of this technique are very consistent in case of smooth model of statistics (e.g. sample means, sample variance, maximum likelihood). However, the jackknifing process does require \( n − 1 \) of re-samples, each with one withdrawn value, where \( n \) is a size of the initial sample.

The bootstrap re-sampling method [17] can be seen as a generalization of jackknifing technique. The main function of this method is an estimation of a standard error over the wide range of data distributions, including some falling under the non-smooth functional model. The bootstrapping process is based on drawing the number of subsamples with replacement from the initial sample and collecting its statistics for a further bootstrap estimate. The bootstrap standard deviation estimation algorithm of the initial estimator is present in Algorithm 3.

The necessary number of subsamples is one of the main differences between jackknife and bootstrap techniques. While the jackknife method requires a fixed number of re-samples, the number of re-samples for the bootstrapping technique varies and can be much lower than that of jackknife. Furthermore, the number of required re-samples can be reduced by applying the Monte Carlo standard approximation technique [35]. As a result, bootstrapping is the most common nowadays used re-sampling technique.

The EARL framework (EARL) [26] employs the bootstrapping technique as a part of a non-parametric extension for the Hadoop framework. The goal of this framework is to provide accurate estimations of an arbitrary MapReduce job results with a reduced time and resource constraints. The approximation model is based on bootstrap combination with a newly introduced delta maintenance re-sampling technique. In order to reduce the number and size of sub-samples, EARL bootstrap
3 Early approximate results enabling techniques

Variance estimation is based on the Monte Carlo approximation, which claims that the number of subsample can be set to \( \frac{1}{2\epsilon^2} \), where \( \epsilon \) is a desired error of the approximation with respect to the original estimator of bootstrap\(^{[17]}\). However, experimental results given in\(^{[26]}\) show that this number can be further reduced. For its assessment, EARL framework uses the SampleSize and BootstrapEstimation multi-phase algorithm. This algorithm measures the difference of error variation between two candidate collections of very small subsamples. If the difference is small enough (smaller than the desired error accuracy), the number of subsamples is set the same as the number of subsamples in the smallest collection of two used in a process.

The resampling technique employed in the EARL framework uses the delta maintenance method. This method enables the initial subsamples re-usage, in case its size is not sufficient for the accurate bootstrap results approximation. Its functionality is based on the observation that a larger sample \( S' \) can be obtained by aggregating a previous sample \( S \) and additional sample \( \Delta S \), drawn from the initial dataset. The process is incremental, as the results computed from sample \( S \) can be later re-used in results of \( S' \) computation.

The listed optimizations required several modifications of the Hadoop framework. First, the pipelining has been introduced between Map and Reduce processes. Furthermore, the inter task communication have been enabled through the initialized pipelines: Reducers are able to send messages (i.e. sample expansion or job termination) to the Map processes.

As a result, EARL framework is able to boost the performance of the Hadoop framework up to four times in a scenario with relatively large dataset (> 100 GB). However, there are cases when this modification can result in an overhead over the execution time retrieved from the original Hadoop framework (e.g. when the size of dataset is relatively small (< 1 GB)).

3.4 Online aggregation

Online aggregation\(^{[23]}\) is a technique enabling an interactive access to the running aggregation queries. 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 duration of job execution and lack of partial results are the main issues addressed by the online aggregation technique. This technique enables online query processing without requirement to provide its prior specifications. As a result, users are able to observe the progress of running queries and control its execution (e.g. stop query processing in case early results are acceptable). Furthermore, the results update frequency can be configured, i.e. set the refresh rate of results during the query execution. The online aggregation technique design relies on running
aggregates estimation, considered as "correct" regardless of the access order of data records. However, statistically meaningful estimates of the query results are available only if records can be retrieved in a random order, i.e. collected data can be seen as a random sample at any given time during the query processing. Consequently, a system is able to provide running confidence intervals (proximity to the final query result) along with an estimated query results. A number of estimators, useful for several types of running confidence interval computations were proposed in \cite{22} These estimators are designed for relational databases and the common single-table or multi-table aggregate queries, i.e. AVG, COUNT, SUM, VARIANCE and STDEV.

Though the online aggregation technique has never made an impact on the commercial DB products, it becomes newly relevant due to rising interest in effective very large-scale data processing. In \cite{31} authors introduce the system model and integration of online aggregation technique into the MapReduce system. Hyracks framework has been selected for an implementation. There are several significant differences between this system design and the one used in MapReduce Online \cite{14}. First, the proposed model is suitable for statistical analysis, i.e. allows predicting the final result of a Hyracks job at any point of time during the execution. Furthermore authors discuss and provide the theoretical foundation of a Bayesian framework with the designed model. Following it, the system is able to estimate the final results and its confidence bounds. In order to reduce estimation bias, the system takes into account the "inspection paradox" \cite{15}, which refers to the possible correlation between block processing time and its aggregate value. In other words, some of the blocks can have more data and take longer time to be processed, while others can take less time but their aggregate values might have a small impact to the overall results, leading to biased estimates. The Bayesian estimation deals with it by taking into account the processing time of each block. However, this approach is quite complex regarding the implementation, as it requires several processing steps along with time synchronization, which can be a considerable problem in a distributed environment. Authors claim, that they have implemented several models for aggregate queries (SUM, COUNT, AVG, VAR and STDEV), however, there are no implementation details provided. Furthermore, the description of the system design does not consider data randomization and possible bias reduction.

3.5 Summary

In this chapter we have analyzed the three main groups of approximate results enabling techniques: histogram-based, wavelet-based and sample-based. Each group has its advantages and shortcomings. The summary of our investigated approximate results enabling techniques is present in Table 3.2.
The histogram-based estimation techniques can perform well (in terms of results accuracy) independently of the input data distribution and do not introduce large space overhead. Wavelets, on the other hand, can provide very accurate estimations, as well as data compression. Furthermore, they can estimate the answer of nearly all SQL-like queries. In section 3.2 we describe more details, which make wavelet-techniques a robust tool for early estimations of the final results. However, both techniques have several important limitations. First, in both cases the data pre-processing phase is essential, before the further analysis can be applied. Furthermore, the efficient integration would require fundamental changes in the query processing engine: queries should be translated to be applied over histogram or wavelet-coefficient domains, respectively. As a result, we leave further investigation of these techniques out of this thesis scope.

This leaves us with the sample-based estimation techniques. The assurance of statistical properties and availability of distinct sampling strategies and estimators are just few of their advantages. For a further research we have selected the uniform random sampling technique with some additional modifications, i.e. block-level samples. The main reason of this choice is techniques suitability for online processing: a sample can be obtained during the job execution, without a-prior data processing. Furthermore, the uniform random sampling can be closely integrated into the Hadoop framework and benefit from the pipelines design in HOP. Consequently, we have selected to adapt the latter framework. The further description of our system architecture is presented in the following chapter.
<table>
<thead>
<tr>
<th>Technique</th>
<th>Histograms</th>
<th>Wavelets</th>
<th>Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advantages</td>
<td>High accuracy estimation in case of data skew or outliers. Low storage space requirements</td>
<td>'Lossy' input data compression - high accuracy with relatively low storage overhead</td>
<td>Universality, simplicity, relatively small processing overhead</td>
</tr>
<tr>
<td>Pre-processing</td>
<td>Equi-width histograms require prior knowledge of min/max values in a dataset; equi-depth: dataset values have to be sorted or costly function applied</td>
<td>Wavelet transform process execution and 'best' coefficients selection</td>
<td>Varies: some techniques can be applied online, others (e.g. stratified sampling) require a pre-processing phase</td>
</tr>
<tr>
<td>Estimation</td>
<td>Equi-depth histograms have a bounded worst-case estimation error</td>
<td>Based on the stored Wavelet decomposition coefficients: if all coefficients are stored, returns the precise answer</td>
<td>Varies based on queries, data set properties and used estimators. Overall: relatively high</td>
</tr>
<tr>
<td>guarantees</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query applicability</td>
<td>Most of aggregate queries for single- and multi-dimensional datasets)</td>
<td>Almost all SQL-like queries</td>
<td>Almost all SQL-like queries</td>
</tr>
<tr>
<td>Additional</td>
<td>Algebraic translation of initial queries</td>
<td>Selection of query processing strategy: can require query translation</td>
<td>Selection of sampling techniques according to underlying data distribution and targeted queries</td>
</tr>
<tr>
<td>requirements</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 3.2: The summary of introduced early results enabling techniques*
In this chapter we describe the design of our modifications in the MapReduce Online framework. We start by describing several real-world use-case scenarios which we want to address in our designed modifications. Next, with reference to described scenarios and some non-functional requirements, we define the design goals for our work. Finally, we provide an overview of design choices we made and the complete system.

4.1 Use-case scenarios

In order to determine the system design and its objectives, we start this chapter by covering several use-case scenarios. We intend, that our design would be able to address each of below described use cases.

For our first scenario we chose a weather sensors data processing application. The sensors data is available from the National Climatic Data Center\footnote{\url{http://www.ncdc.noaa.gov/}}. Each log contains yearly measurement records stored in ASCII format - one line per record. The selected application measures average annual temperature. It is necessary measurement for comparison of climate changes, used in the evaluation of various phenomena influence. Each weather sensor generates a new record and stores it in the log file every hour. If these files would be read sequentially, Mappers would first access January data, then February and subsequent months. As a result, the early estimation of average temperature would be biased: for the most part in the northern hemisphere, temperatures during winter months (e.g. January and February) are lower than during summer months (e.g. July and August), so the difference in estimates would be apparent. As a result, our design should address the possible data bias over the process sequence of the files. Furthermore, the accuracy of our designed systems returned estimations should converge faster than of the ones returned by the framework, which relies on the sequential big-data processing.
One common group of MapReduce applications is responsible for a prompt evaluation of the trend topics in selected areas. The benefit of this knowledge is immense: it can be used to establish efficient publicity campaigns, choose the most suitable advertisements or for other goals. Some may assume that single day data log might not take long to process, however a recent Twitter post proves it can be a different scenario: every three days they have been receiving over 1 billion of tweets. Processing similar volume of data can take considerable amount of time, whereas the accuracy of results is highly related to its accumulation rate. For our scenario, we acquired the certain keywords (“g1leage” - an online event in China) search trend over the last 90 day’s period. It is displayed in Figure 4.1. The numbers on Y-axis define the interest rate of the search (100 - maximum, 0 - minimum).

Suppose we have a raw search requests data and want to determine how popular this particular pair of keywords was during the last three months period. We can split its distribution into the three parts: no relevance (equal to 0), low relevance (between 1 and 50) and high relevance (more than 50). In the case of the sequential data access, the high relevance area would be reached at the end, hence lowering the accuracy of an early estimation returned by the MapReduce application. Our designed system should be able to retrieve and process some records of each area at any part of the execution. In addition, if the user is not satisfied with the returned early estimations, he should be able to get the exact result in a timely manner. Consequently, our design should not introduce large overhead over the base framework, which is highly optimized for a batch-processing of big data.

For a final scenario we chose a case, when our design should have only a limited advantage over the unmodified Hadoop Online framework. We have selected a
4.2 Design objectives

The early estimation of final results within big-data frameworks is a challenging problem due to hidden issues and various requirements. Based on our use-case scenarios and some non-functional requirements, our system design should address following challenges:

**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 as valid, if it has statistical properties that can be used with some of estimators available in statistics theory for an accurate results estimation. Statistical properties should be guaranteed independently of the input data format or its underlying distribution.

**Simplicity:** We aim to implement the functionality as a discrete intermediate layer for the MapReduce architecture. The motivation comes from accessible variety of big-data processing frameworks and their extensions. We intend our solution to be independent of the framework, thus the design of it should be simple to integrate with any given system.

Figure 4.2: The relevance of "Hadoop" keyword in Google search over three months period

frequent search keyword: "Hadoop". Its Google search relevance results are illustrated in Figure 4.2. The bottoms in this graph correspond to the weekend time, when the relevance of "Hadoop" keyword drops by up to 50%. The distribution has an evident pattern, so the acceptable accuracy estimate can be received after processing one week records and scaling it up.

Over the long run, this scenario does not give noticeable advantage to the system with data sampling technique, as estimation is relatively independent of the processed weeks count. However, there is a visible trend bias over the period of week (i.e. workdays and weekend). Users might not be aware of possible bias and can compute largely inaccurate estimates. As a result, it should be addressed in our designed system.
4 System design

**Integrity:** The modified system should be able to execute any application which is supported by the unmodified system. Furthermore, users should be able to select whether the new functionality will be active or inactive during the execution of applications. The job applications should require little to no modifications when intent to be used with the new functionality.

**Performance:** The design of the modification should take into account the possible overhead and try to reduce it to have a minimal effect to the overall performance of the system. This includes both MapReduce and data collection phases. Furthermore, the designed system should address properties of the underlying data storage system, to ensure an efficient use of its features.

4.3 Architecture

The job execution workflow within the MapReduce Online framework is visualized in Figure 4.3. To explain it, we will describe the flow of one MapReduce job execution. Map and Reduce are the two main tasks in Hadoop Online framework. At the initial stage, after the job is submitted for the execution, the initialized Map tasks access the storage system and retrieve input data. The distributed file systems (i.e. HDFS) are a common selection for Hadoop Online framework. It stores datasets in the form of blocks, which are returned to Map tasks wrapped as the Input Splits. In order to apply the Map function, Mappers have to parse the Input Splits into separate \(<key, value>\) records. The structure of input data is established by the `InputFormat` function. This function selects a valid data reader, which will read an input file sequentially, parse it and return the input records to Map tasks, where the application defined Map function will be applied. The output of Map tasks is stored in an in-memory buffer and is periodically sent to the Reduce tasks. The processed records are transferred through data pipelines, initialized by using open TCP sockets (between Map and Reduce tasks). Reduce tasks collect and merge partial results.
4.3 Architecture

Figure 4.4: Direct random sampling approach

sent by Mappers. Consequently, the Reduce function is applied to each record in the merged results file. Based on the initial job configuration, the output of Reduce tasks is materialized as the final job results or as a snapshot. Further, we describe and evaluate several alternative designs, which take into account our set goals and can be applied to the above described system.

After evaluating techniques in our conducted research, we chose to focus on sampling techniques, due to the following advantages: they do not require additional preprocessing stage and can be integrated with most of the large-data processing frameworks. The straight-forward uniform random sampling design for the Hadoop Online framework is illustrated in Figure 4.4. This approach requires only a slight modification of Map tasks in the Hadoop Online framework. Standard Map tasks during execution access a data block and process it sequentially, one record at a time. Uniform random data sample can be obtained with a slight modification of a read operation: each call to the record reader should return a random record from the spill file. The described process is marked as (2) in Figure 4.4. However, the performance of such a system would be reduced noticeably in comparison to the initial system design. In fact, common distributed file systems (e.g. HDFS) are highly optimized for sequential reads, while random accesses can take relatively long time. Furthermore, the record reader should have the prior knowledge of the spill file, i.e., how many records each data block stores. Such a requirement increases the complexity of implementation, as HDFS does not include it in files metadata. Considering our set design objectives and experience, we believe that this solution is insufficient to deliver expected results, so it was not investigated further.

The evaluation of previously proposed design and further research have led to the conclusion: it is better that file splits are processed in memory, after they have been loaded from HDFS. This process is illustrated in Figure 4.5. In this scenario, the record reader parses the entire file split and stores every record in the local memory of the Map task 4.5 (1). Next, stored records are randomly shuffled and
sequentially returned to the Map function \[4.5(2)\]. The strong point of this approach is its simplicity: Map functions can use a sequential access to the parsed records, while, from statistical point of view, this process is equivalent to a random sampling. In addition, this modification can reduce the bias of the input data. For example, file split can contain a day long log of user-clicks. If processed sequentially, the returned average clicks per-hour estimation would be biased towards the periods of high/low user activity in the server. However, shuffling the data before the further process would address this issue. The present solution satisfies the set design goals, thus it has been the starting point for our implementation. However, the evaluation of built system exposed one factor of the process we did not address. The common MapReduce job processes a sizeable data input (in range of Gigabytes or even Terabytes), which is stored as a large number of data blocks. During the job execution each Mapper can sequentially process several data-blocks. This design does not consider described scenario: it assumes execution of one file split per Mapper. As a result, in the case of multiple split files, the collection of processed records is not fully random and still can be biased, so the accuracy of early estimations might suffer. For example, there can be six Map processes and more than six input file splits. In this case some of splits will be processed later than others, thus the data bias might be still present in an early estimation (before the later results are available). The Hadoop Online framework runs Map tasks sequentially on a number of spawned Java\textsuperscript{TM} Virtual Machines (JVM). In other words, those machines are re-used for multiple Map tasks processing in a sequence. After analyzing the former scenario, we have switched to our next and the final system design.

The final design takes into account the possibly high number of data splits to the available Mappers ratio of the submitted MapReduce job. We chose to introduce the block-level sampling technique, which enables distribution of input data: during job execution, each Map process can retrieve a number of record blocks from several data splits \[4.6(1)\]. Further processing steps are the same as in our former design: accessed record blocks are stored in memory (Figure \[4.6(2)\]) and shuffled to reduce possible data bias. The discussed solution well addresses our set design requirements. First,
4.3 Architecture

Figure 4.6: The design of system with block-level sampling and bias reduction

Block-level sampling can be implemented as an additional layer with no important Mapper-related changes. Furthermore, 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 can reduce 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. The structure of our proposed design is illustrated in 4.6.

However, our proposed system design does not address the estimation accuracy problem. We chose to leave this job to the user due to the following reasons. First, the Hadoop framework has very limited information about queries which will be run during the job execution, i.e., queries are transparent to the system. In addition, the style of code in the MapReduce applications can vary greatly, making query identification a difficult problem. It can be addressed by introducing a built-in collection of most common operators, which would change the user defined Map and Reduce functions. However, this step would require some low-level modifications in the Hadoop framework. Furthermore, we are aware of a growing trend in a multi-job execution graphs, run on top of Hadoop (e.g. the Pig platform [2]). Due to the lack of user interface, the best option would be to return estimation accuracy as an additional data in materialized results. We are convinced that these results are relevant only in the single-job case, while the multi-job execution would contribute far less: the second and further Map tasks would have to address accuracy values in the processing phase, e.g. adjust the estimation or branch it to several values (e.g. min, current and max). Consequently, our design focuses on the statistically profound online aggregation, which can provide relatively accurate early estimations. Users then can apply some of the statistical techniques, e.g. delta method, or observe the difference among several early estimations (e.g. after 10% and 20% of input data processed) and scale the estimation accordingly. Next, we give and explain
several use-case scenarios which highlight the advantages of our proposed system functionality.
Chapter 5

Implementation

In this section we discuss the proposed design implementation details. As both Hadoop and Hadoop Online frameworks are written in Java language, we have also selected it for the implementation task. At the beginning we have closely investigated the MapReduce Online framework source code (it is based on the Hadoop 0.19 version) to familiarize with the structure of the project and its inner processes during the execution. One of the main goals of the code study was to determine the least interfering integration path, so the initial functionality of Hadoop framework would be preserved. Likewise the described design and implementation process can be split into two phases: bias reduction, namely the shuffling phase and the sampling phase. The details of its fulfillment are further elaborated in consecutive subsections.

5.1 Data randomization

Data randomization is a necessary step to reduce the overall bias of the data and ensure that the randomness property is presented at any time during the execution. We have decided to piggy-back the data randomization on top of the initial reading process, as several tests showed that in our case it is the most efficient solution. The functionality can be enabled by setting up newly introduced MapReduce daemon parameter. The list of supplementary Hadoop framework parameters is given in Table 5.1. The bias reduction process takes several steps, listed below:

1. **Data access phase.** In the initial step, the Map process retrieves the file URI from the NameNode and accesses it. By default, a file is read as a stream of bytes and parsed into lines of text (based on specified separator characters). Next, the text is parsed into records and sent to a Map process. We have introduced a new data collection method, which process blocks of the entire input file and stores them to the local task memory. The further processing (i.e. the Map task) is delayed until the below described shuffling phase is done.
5.2 Block-level sampling

2. **Shuffling phase.** The input data (in form of lines) is stored in our implemented data structure. The design of it focuses on data randomization efficiency: 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 \([0...2^{31} - 1]\). By default, Java does not have tuple data structure, so we had to implement one for this project. The addition of an integer value is necessary for the shuffling process: stored data is sorted according to the assigned integer value. After evaluation of several sorting methods, we have chosen the combination of quicksort and heap-sort algorithms. In our standalone tests it proved to be the most efficient for a given problem. Consequently, a sort process returns randomly shuffled input data. Short notice: we make an assumption that our framework will be set up on systems with sufficient amount of memory to store the whole HDFS data block. Our tests showed that 512 MB of memory allocated to TaskTracker child processes (Map tasks) is sufficient for the 64 MB size (the default HDFS parameter) input data block processing.

3. **Data process phase.** Storing the shuffled data to disk would enable us to reuse the initial Hadoop data reading function. However, since the shuffling phase is not based on an external sort algorithm, 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 (which are done in the data access phase) and instead returns the line of text (number of bytes) from the local memory, as showed in step (3) of Figure 4.6.

5.2 Block-level sampling

The block-level sampling requires several modifications of the Map process, mostly related to the storage system access. By default, HDFS provides several random access methods, but it is a costly operation and should be used deliberately. To ensure the best result, we have followed the guidelines of the file access in HDFS, related to the random file reading \([40]\). Sampling process is configured by few parameters, which we list in Table 5.1. However, users also have to specify the particular file input format, designed to work with block-level samples. The main processes and their operation details are presented further in this section.

5.2.1 File sampling

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, i.e. sample the data. The initial file information is acquired by issuing the request to
### Table 5.1: Related framework parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>io.file.shuffle</code> (boolean)</td>
<td>Input records shuffling method for data bias reduction</td>
</tr>
<tr>
<td><code>io.split.maxsubsplit</code> (int)</td>
<td>Set the number of split files from which block-level sample will be created. Default value = 4. <strong>Note:</strong> works only with RandomInput formats.</td>
</tr>
<tr>
<td><code>io.split.insort</code> (boolean)</td>
<td>If enabled, split files are shuffled before sampling method is applied.</td>
</tr>
<tr>
<td><code>mapred.child.java.opts</code> (-opts)</td>
<td>Set the memory size of MapReduce child processes. E.g.: (-Xmx512m) (sets max. available memory to 512 MB)</td>
</tr>
</tbody>
</table>

the HDFS. By default, the data acquisition process would proceed to Map function. In contrast, our introduced RandomFileInputFormat stores each file’s information in the local memory. Based on it, initial data splits are divided into the number of equally-sized data blocks, which will be processed by separate Map tasks. This approach requires minimum number of random accesses (one-per-block) lowering the cost of HDFS I/O operations (mostly due to the relatively small number of seek operations) [40]. We determine the significance of I/O overhead in the evaluation part.

The actual sampling rate, i.e. the number of splits each block will be divided into, can be determined in the MapReduce job configuration. The default value is four data fragments per initial file split. In case the data block cannot be split into equal parts, sampling process will determine the optimal sizes for each block ensuring that Mappers will have 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 `io.split.maxsubsplit` parameter to 6, most of Map task will process six blocks of data from six separate input splits. Furthermore, the list of input splits, obtained in the data access phase, can be shuffled (`io.split.insort` parameter) before the sampling process is initialized. This step can improve the randomness property of block-level samples, in other words, accuracy of estimations can be enhanced. Finally, due to the equal size of split parts, our algorithm improves the balance of the workload between the Map tasks (e.g. when the size of input files varies and is lower than set HDFS block size) and addresses the "inspection paradox", described in Section 3.4.

Following process steps involve storing Map tasks output to the local buffers and later processing it in Reduce tasks. This part of the job process is identical to the one of MapReduce Online, described in Section 2.2. Subsequently, subject to the job settings, the Reduce task will take a snapshot of aggregated values and materialize it.
5.3 Results correctness

to HDFS, from where it can be collected by the user.

The FileSplit data structure, used in Hadoop and Hadoop Online frameworks, does not support methods required for block-level samples processing. By following the existing code structure, we have developed RandomFileSplit format, sufficient for storing the complete block-level samples information, including URI’s of the input splits, data offsets and block lengths. Consequently, these details are used to query the Namenode of HDFS in the data access phase. As a result, Map tasks can sequentially open several input data streams, consumed further in the processing phase. However, the default data reader does not support multiple input data streams, thus we had to address it.

5.2.2 File processing

The MapReduce Online framework can process various data formats. In order to apply the Map function, the input data have to be parsed into records - key-value pairs. This process is defined in an InputFormat interface. There are several commonly used methods which implement InputFormat interface, i.e. TextInputFormat and KeyValueTextInputFormat. The earlier format defines position in the file as a key and whole line as a value, whereas KeyValueTextInputFormat can split the line into the key-value pair, according to the parameters set in application. The input format can be defined in the MapReduce job properties, otherwise TextInputFormat will be used by default. However, as these formats are designed to work with a single file split, they will not be able to parse block-level samples.

We have implemented two additional InputFormat methods, which function similarly to the above described ones and are able to process the block-level samples, created in the sampling phase. There are two main differences between the initial and our introduced formats. First, our methods are able to access multiple input splits and read the block of data from each of it. This process is cyclic: as one data stream is consumed (certain number of bytes is read), next one is initiated, until the whole sample is processed. Furthermore, block-level samples can start or end in the middle of the text line of initial dataset, as the sampling phase relies solely on stored files meta-data. It is addressed with our defined rule: a task which receives a data block with the beginning of text line, will process the entire line, otherwise it will skip that fragment of the line. These modifications are present in RandTextInputFormat and RandKeyValueTextInputFormat classes.

5.3 Results correctness

In this subsection, we discuss the correctness of our system in terms of its output. The traditional MapReduce implementation can support both single-job and multi-job
executions. Latter one consists of several separate MapReduce jobs, usually generated by some high-level query system, such as Pig platform.

The single-job execution correctness is straightforward. The running confidence interval depends on the number of read records, which increases during the execution. The large-sample confidence interval (relevant to a large datasets) is based on the following relation: the interval of estimation value $[Y_n - \epsilon_n, Y_n + \epsilon_n]$ decreases proportionally to $\sim 1/\sqrt{n}$, where $n$ is the number of randomly read records. As our designed framework utilizes the block-level sampling and data shuffling techniques, we can assume that Map tasks retrieve and process records in random order. These records are later sent to Reducers and finally, materialized. Due to the randomization, the described process can be considered as a statistical estimator of the final results. Furthermore, at the end of job execution, if it was not canceled before, whole input data will be processed by Map tasks (independent of sampling and shuffling settings). Consequently, the final result will be materialized by Reduce task and stored in HDFS, similar to the unmodified Hadoop framework.
5.4 Summary

<table>
<thead>
<tr>
<th>Contribution</th>
<th># of Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOP modifications</td>
<td>665</td>
</tr>
<tr>
<td>Test applications</td>
<td>450</td>
</tr>
</tbody>
</table>

Table 5.2: Lines of code excluding comments

It is also simple to address the multi-job execution correctness. We identified that single-job execution is correct. Multi-job execution can be seen as a collection of single-job executions with shared input/output data. As the results of a separate job are correct, the overall results of the multi-job execution are correct. For example, imagine we have a job consisting of the two separate MapReduce applications. In this case, the first MapReduce job will be executed in the same way, as described in the single-job correctness paragraph above. Mappers of the second MapReduce application will start input data processing by sampling the early results of the first execution (i.e. sampling the 1st jobs output sample). As a result, the early estimations of the second job might have lower accuracy (e.g. due to the lower selectivity), however the correctness of results is still guaranteed. Eventually, after the first job finished the processing, Map tasks of the second MapReduce job will obtain the complete input data, so further it can be seen as a single-job execution. Consequently, multi-job execution, consisting of 2 separate jobs will produce the correct results. Following this example, we can show that multi-job executions with more than two jobs will also return the correct results.

5.4 Summary

In this chapter we described the implementation of our designed MapReduce Online framework (HOP) adjustments, listed in Section 4.3. We have listed the main methods, its performed functions and correlation during the job execution. The introduced functionality mainly relies on classes displayed in Figure 5.1. To ensure straight-out integration, we have modified a number of present HOP classes and closely followed the Hadoop project programming style. Our contribution, expressed as a number of lines of code, is listed in Table 5.2. There we have also listed a number of code lines written for the evaluation part. It includes MapReduce jobs and datasets formatting applications. In the next chapter we present its details and evaluate the performance of our developed system.
Chapter 6

Evaluation

In this chapter we evaluate the effectiveness of our designed system. First we provide the performance evaluation depending on various system properties. Later we evaluate the estimation accuracy dependency over our present early estimation techniques’ parameters.

6.1 Test cluster setup

For our systems evaluation we got an access to the OpenStack cluster, hosted by SICS (Swedish Institute of Computer Science). This cluster is running on 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 created eight (1 master and 7 slaves) large-instance virtual machines cluster on top of the OpenStack infrastructure. Each node in a cluster had assigned 4 virtual CPUs, 8 GB of memory and 90 GB of disk space. Nodes had Linux Ubuntu 12.04.2 LTS Operating System based on 3.2.0-38 Linux kernel and a running 1.7.0_14 version Java™ SE Runtime Environment. There we deployed our modified and the Hadoop Online frameworks. Common settings we adjusted before deployment are listed below:

- Each job can use up to 17 Map and 5 Reduce tasks
- HDFS block size set to 64 MB
- Data replication factor set to 2

6.2 Experimental datasets

For our systems evaluation, we retrieved and pre-processed a number of varying size datasets. For the performance evaluation we prepared weather sensors data of several decades, available from the National Climatic Data Center ftp server\(^1\). Data is

\(^1\)ftp://ftp3.ncdc.noaa.gov/pub/data/noaa/
present for each year separately (currently available years 1901 to 2013) and contains log files from varying number of weather stations. The present logs were compressed and stored separately (one per station). As we were aware of the common Hadoop framework usage scenarios and HDFS design, 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 data.

For accuracy evaluation we prepared several datasets with different data distributions of size between 11 and 25 GB. One of the datasets is an extract of the previously mentioned 100GB weather data and consists of 10 files with 25 GB total size. The structure of it is as follows: the yearly log files of the each weather station are merged sequentially, i.e. one after the other. As a result, we can assume that data distribution is similar to one of our present use-case scenarios, displayed in Figure 4.2. We decided to reuse this data for our second experimental dataset, with one important adjustment - arrange data according to the date and time of the measurement (i.e. sort it). This step returns a log file, which is similar to the ones obtained and processed nowadays in large-scale data analysis systems. Furthermore, it introduces additional difficulty for obtaining early accurate estimations of final results: weather temperature data can be biased according to the time of measurement, e.g. winter or summer. The last dataset consists of the complete Project Gutenberg\footnote{http://www.gutenberg.org/} e-books catalog (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 evaluated systems.

6.3 System performance evaluation

6.3.1 Early results snapshots influence to the overall systems performance

Our system is implemented on top of the standard MapReduce Online framework. It allows materializing the early snapshots of aggregate results present by Reducers. This functionality is necessary for early estimations and the possibility to terminate the incomplete MapReduce job without loss of results (depends on the job configuration). However, in case approximate results are not of acceptable quality, the user can wait till the whole data input will be processed and the final results will be returned. Consequently, we evaluate the influence of snapshot materialization frequency (i.e. how often estimations will be materialized into HDFS) on the overall performance of the execution. For this test, we increased the maximum memory size available.
Our measured performance results are displayed in Figure 6.1. Each value was obtained by averaging the results of 3 to 5 executions over an extended period of time, to reduce the influence of cluster performance fluctuations (e.g. due to other jobs, executed in parallel). Overall, the results show that there is a moderate execution overhead (up to 35%) in the case of high number of snapshot. This overhead 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 snapshot materialization 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 will later show in the accuracy evaluation section, early estimations of various aggregate MapReduce jobs can have a relatively high accuracy. Consequently, the user can terminate the job execution earlier.
6 Evaluation

Figure 6.2: The evaluation of three systems performance over various size inputs: our system (parameters: bias reduction enabled, 10% snapshot freq.), MapReduce Online (parameters: 10% snapshot freq.) and the Hadoop framework

6.3.2 The performance of bias reduction process

In this subsection, we evaluate the performance of our modified system with enabled bias reduction process and compare it to the other two large-scale processing frameworks. For this task, we prepared seven different size (5.5 to 100 GB) datasets 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 (with relatively small deviation), which are present in Figure 6.2. This figure shows the MapReduce job execution times dependency over the size of input data. We demonstrated the evaluation results of three systems: Hadoop (ver. 0.19), MapReduce Online and our modified with the enabled bias reduction. 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 (snapshot freq. was set to 10% in all tests). The bias reduction process performance depends on the systems parameters: while input data size is relatively small, the bias reduction process gives little to no overhead to the overall execution time. However, in the case of large inputs (consisting of 10 - 21 separate datasets), the overhead noticeably increased (up to 20% over the execution time with no bias reduction). Such ambiguous results can be explained by our choice of system parameters, namely the relatively low
number of Map tasks, which means that the additional processing time is aggregated
over a large number of sequentially processed data blocks (i.e. each JVM executes a
large number of Map tasks sequentially). We strongly believe, that shuffling phase is
the main source of displayed overhead: it arranges input data block records by the
assigned random prefix. However, there is additional overhead which is not reflected
in the present figure: before sorting, all records must be stored in the local memory
of the processing nodes. The amount of required memory depends on size of the
data block, stored in the HDFS file system. As a rule of thumb, we recommend to
increase the maximum memory amount by $4 \times \text{blockSize}$.

6.3.3 The overhead of the sampling process

In this test we evaluated the sampling process influence to the overall job execution
performance. As the MapReduce programming paradigm focus on large-scale data
processing, we selected the largest, 100 GB size input dataset, which was used
for the average temperature computation. We performed a number of tests with
various block-level sampling rates. In order to reduce the result correlation with
clusters’ load, we repeated all runs multiple times on different dates and averaged the
retrieved results. For illustration, we also measured the standard MapReduce Online
framework execution time. Figure 6.3 shows the dependency of the job execution time
over the block-level sampling rate. 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 such overhead reasons: sample generation
process, block-level samples movement across the nodes and the increased number of
random accesses to the HDFS file system. Due to our introduced modifications design,
the number (in the HOP system) of random accesses increases $n$ times, where $n$ is the
data sampling rate. Overall, our designed sampling process increases the execution
time of MapReduce jobs by around 16 percent (in comparison to MapReduce Online
framework results) in case of 10 block-level samples per Map process, however, we
believe that it is a reasonable cost in case of aggregate jobs, due to relatively accurate
early estimations, as we demonstrate in the next section.

6.4 Evaluation of estimations accuracy

6.4.1 The average temperature estimation accuracy

In this section, we evaluate and compare the accuracy of early average temperature
estimations, which are returned by our and Hadoop Online systems. We use two pre-
viously described weather datasets with different distributions of values across them.
The evaluation results are present in the following subsections.
6.4.1.1 Aggregate log files dataset

For our first experiment, we used ten yearly aggregate logs of weather stations, in total 25 GB of data. It is consumed by our implemented MapReduce application, which measures the average yearly temperature. This application can be executed with either of frameworks: Hadoop, Hadoop Online or our modified one. 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 the complete functionality of our framework: data sorting and sampling phases. Due to the extensive amount of results, in Figure 6.4 we demonstrate the gathered estimation results of 2 randomly selected yearly log files.

This figure presents the absolute error range variation subject to the number of block-level samples given for each of Map tasks. In the graph depicted boxes define 25th/75th percentiles while ends of lines show min/max values of the measured absolute error. Each of the displayed line-box combinations consist of 8 values (present as diamond-shaped symbols) obtained after 10 - 80% of input data is processed. We observe that the default setting of 4 block-level samples per Map task is too conservative: few of initial estimations (i.e. till Reducers process 20 percent of input data) are fairly inaccurate, with an absolute error rising up to 45%, in
6.4 Evaluation of estimations accuracy

Figure 6.4: The yearly average temperature estimations error range dependency on effective sampling rate

Comparison to the final result. To some extent similar results are obtained both with 6 and 8 samples per Map task. The reason of this observation comes from the probability theory - with a lower number of block-samples 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 more accurate from the beginning (at 10 and 20% of processed input file). At later stages of the job execution, the estimations become relatively accurate independently of the number of processed blocks. Consequently, for a further evaluation of early results accuracy we selected a case with 10 block-samples, as it gives fairly good accuracy and has a lower overhead in comparison to a higher number (e.g. 12 or 16) of blocks per process.

In Figure 6.5 we present the dependency of the yearly average temperature estimations error over a segment of processed input in the standard Hadoop Online system. One may notice, that some of the values have 100% error rate. This value of the error rate means that the system did not provide any estimation for for a particular year. All ten estimations were available only after 40% of the input data was processed. Furthermore, we noticed some tendency in the execution: the system mostly process blocks of one or few input files at a time. One can easily observe this in the graph by checking the maximum error value between 40 to 70 percent of processed input data - it does not change (for one of the input data files). Overall,
we can summarize 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 will require the whole input data to be processed to return a fairly accurate estimation. Consequently, in statistics available estimators are useless, due to the obvious sequential nature of the Hadoop Online framework processing.

On the other hand, our system gives promising results, displayed in Figure 6.6. Even at 10% of processed input, the maximum value of the absolute error is around 30%, with 25th/75th percentile being less than 20 percent. Furthermore, there is a quite noticeable results accuracy convergence, with a maximum error being around 10 percent when half of the input data is processed by Reduce tasks. This is clearly illustrated by values of 25th/75th percentile boxes. To some extent, we can reason that the distribution of data (described in Section 6.2) also influences the accuracy of estimations: depending on the data read sequence, the final result might be underestimated or overestimated. One may have noticed that even at 10% of processed input, average temperature estimations are available for each of ten years, whereas the Hadoop Online framework required 40% of input to do the same. According to our observations, we can consider Map tasks input data as random samples. Therefore, our system allows applying some of statistical estimators over the returned early approximate results.
6.4 Evaluation of estimations accuracy

Figure 6.5: The error range of average temperature estimations of ten yearly data inputs, when data is not sorted. No sampling: base Hadoop Online

Figure 6.6: The error range of average temperature estimations of ten yearly data inputs, when data is not sorted and sampling rate is set to 10
6 Evaluation

6.4.1.2 Aggregate and sorted logs dataset

For this test we used sorted datasets of weather stations. The main difference from previously used input data is its distribution of values: if read sequentially, initially winter months will be processed, then spring months and so on. For the test runs we used the same parameters as in earlier experiments.

We start by evaluating the variation of estimations accuracy according to the effective block-level sampling rate. Due to excessive amount of collected data, we randomly selected 2 years out of 10 used in our tests. The results are shown in Figure 6.7. In comparison to the unsorted weather stations data, there are slight variations in the maximum values of error range: the initial estimations, collected after 10% of input data was processed, can return fairly inaccurate estimations. This outcome is mainly due to the following reasons: randomization (e.g. some month data can be present in more/less samples than others) and a lower selectivity of block-level samples. As a result, the initial estimation might largely overestimate or underestimate the final result. However, the accuracy of estimations converges promptly: most of estimations beyond 20% of processed input returns estimations with lower than 20 percent absolute error. Lastly, we once again observe that the default value of 4 block-samples per Map is too conservative and should be adjusted by the user in MapReduce jobs configuration.

Figure 6.7: The yearly average temperature estimations error range dependency on effective sampling rate over sorted log files
6.4 Evaluation of estimations accuracy

Figure 6.8: The error range of average temperature estimations of ten yearly data inputs, when data is sorted. No sampling: base Hadoop Online

Figure 6.9: The error range of average temperature estimations of ten yearly data inputs, when data is sorted and sampling rate is set to 10
The error range of estimations obtained from the Hadoop Online framework is displayed in Figure 6.8. The collected results prove our proposition of the HOP sequential processing nature, as some of early estimations are very accurate upon the initial estimation, while others have high error even when 60% of input data is processed by Reduce tasks. Finally, all 10 yearly average temperature estimations are available only after 40% of input data is processed - same as in previous test case.

In comparison, our systems estimation results are present in Figure 6.9. In order to ensure the similar test scenario, we again selected the 10 blocks per Map task sampling rate. Due to the random nature of our used sampling method, one can observe some slight error peaks in later stages of the process. However, the overall error rate decreases rapidly, so even at the early stages of processing estimations are fairly accurate. In addition, we observe an evident convergence of the estimation accuracy, illustrated by height and values of 25\textsuperscript{th}/75\textsuperscript{th} percentile boxes. The results are very promising and demonstrate large improvement in accuracy over the standard Hadoop Online framework.

### 6.4.2 Top 100 words count estimation accuracy

In this test we evaluate the estimation accuracy of top 100 words obtained after executing the MapReduce job. We used an input dataset of 11 GB size, combined from a large set of e-books. One particular difference from previous datasets is its underlying Zipfian data distribution: a small number of words occur very frequently, while many other occur rarely. To some extent, trendy topics have a similar perspective - a high peak in otherwise relatively even distribution (of topics). Thus, we strongly believe, that our test results can be interpreted in various situations (e.g. one given in Section 4.1). The \texttt{TopKWordCount} is an application of our choice for this test. It is available from the Hadoop example jobs archive and allows retrieving the $K$ most occurring words in the input dataset. It is similar to the Top\textit{K} query, which is frequently used in DBMSs.

The results of executions are displayed in Figure 6.10 and Figure 6.11. They demonstrate the number of missed words (out of the top100 final words) over the part of processed input data. Our systems setup includes 5 Reducers, so in total we retrieved five result lists of 100 the most frequent words. Consequently, in our figures there are five values for each of the processed input parts. Even after 10\% of processed input both MapReduce Online (Figure 6.10) and our designed system (Figure 6.11) return superior results, with respectively 5 and 7 (or less) misses. As a segment of the processed input data grows, the number of missed words in the returned estimations is reduced. We observe close to no difference in estimations of the MapReduce framework and our designed system. Small variations can be explained by the fluctuations of some evenly popular words (e.g. 100\textsuperscript{th} and 101\textsuperscript{th} word in the final list order). Due to the Zipfian distribution, our block-level sampling
6.4 Evaluation of estimations accuracy

Figure 6.10: The number of missed words in top100 words estimation dependency over the fraction of processed input data. No sampling: the standard MapReduce Online

Figure 6.11: The number of missed words in top100 words estimation dependency over the fraction of processed input data, when sampling rate is set to 10
technique (we set 10 block-samples per Map task) does not provide tangible benefit over sequential processing. However, our intention with this test was to showcase the usability of early results - in most cases the estimation obtained after 10% of input data is sufficient. As a result, the processing cost of MapReduce job can be notably reduced.
Related work

Prior work on the early results estimation has mostly relied on solutions for relational database management systems. Large-scale processing frameworks use the similar approach to RDBM’s approach for data processing, yet they are not compatible with the approximation techniques used in RDBMs. We present a number of solutions in the approximate techniques section and here will focus on relevant work in the big-data analytics area.

The EARL framework is an extension to the Hadoop framework and focuses on accurate estimation of the final results. 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. Sampling is performed dynamically: data is collected in Map tasks and sent to Reducers through established pipelines. Although this system is based on sampling and can give an accurate estimation of the final results, its efficiency is very dependent on the executed MapReduce job and its processed dataset properties. There are some cases, when the estimation process takes more time than the complete MapReduce job. Furthermore, this framework still relies on a batch processing nature - user has limited control of the process, partial results are not available.

Apache S4 is a parallel large-scale data stream processing platform. It utilizes the workflow-based programming model and enables dynamical estimation of some properties of an incoming data stream, without additional pre-processing or data storage. Platform uses the inbound processing: incoming data is consumed by the processing elements (PE), followed by results publishing. To avoid disk I/O bottleneck, S4 framework exclusively uses the local memory in each processing node. Common processing task incorporates number of PE with data being routed through the established data pipelines. The frequency of output events is determined in the job settings and can be delivered in various formats. To some extent, our designed system is similar to Apache S4: both use inbound processing, local memory for data and are able to produce fast approximate results. However, Apache S4 is highly
optimized for the data streams processing and is not able to process batch data. As a result, users are not able to estimate some properties from existing datasets when data stream processing is not possible or excessive task. Furthermore, some of streamed data can be lost due to insufficient amount of memory, as there is no backup storage available. Our system, on other hand, can be adapted to the available amount of memory by setting up the size of data blocks stored in the HDFS.

BlinkDB \cite{BlinkDB} approximate query engine stores pre-computed samples of various sizes on disk. It relies on the two types of samples: large uniform random and smaller multi-dimensional stratified. The issued queries are evaluated on a number of selected samples and an initial estimate is produced. BlinkDB is based on the predictable query column sets (QCS) model, i.e. it assumes that in stored datasets exist the constant sets of data columns used for group or filter predicates. As a result, the system can estimate results of standard aggregate queries (e.g. COUNT, AVG, SUM and QUANTILE). The remaining SQL-like queries, including arbitrary joins, currently are not supported. In case the accuracy or time constraints of a query are not met, larger or smaller samples can be selected. However, this does not imply the support of continuous estimation: query will be executed only on a single, determined size sample. 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 other hand, our designed system can return continuously improving accuracy estimates without the additional pre-processing or samples storage. Further, it does not require pre-assumptions made in the predictable QCS model.
Real time big-data analytics have emerged as one of the most challenging goals in the existing data analysis systems. Nowadays large-scale processing systems, like the Hadoop framework, focus on the efficient processing paradigms and executions over the massive clusters, consisting of hundreds (or even thousands) of dedicated machines. However, as the amount of generated data grows daily, even highly optimized data-processing frameworks, running on large clusters, might require an extensive amount of time to return the final answer, opposite of users’ expectations. One of the solutions, largely covered in Relational Databases area, is the early estimation of results. However, there is only handful of researches which cover some of the estimation techniques integration into the big-data processing frameworks.

In this thesis, we studied the existing approximate early results enabling techniques and analyzed their expected advantages and shortcomings in case of adaption to the large-scale programming models, like MapReduce. We determined that wavelets and sampling are the two very promising techniques in terms of their present estimation functionality. However, the wavelet-based technique requires an in-depth research over the necessary pre-processing steps compatibility and efficient integration with the large-scale processing frameworks. Furthermore, in order to operate, it must known what type of queries are executed during the job process. This information is not available in the Hadoop MapReduce framework: the queries are transparent to the system. On the other hand, sampling can be used with almost all SQL-like queries and requires no prior information related to it. Consequently, we concluded that sampling is the most relevant technique for integration with our selected the MapReduce Online framework (HOP).

We presented a number of use-case and experimental scenarios, which covers a wide range of real-world large-scale data processing applications. We determined the main factors which can influence the estimation accuracy, namely bias of the input datasets, consecutive data processing and the data "inspection paradox", which can occur in the large-scale data processing frameworks. Subsequently, we did a research on the inner processes of the MapReduce Online framework and the HDFS file system and determined the non-functional requirements. After taking into consideration both
8 Conclusions

groups of perquisites, we introduced and investigated several options of the sampling
techniques integration into HOP design. Several of the issues we had to address are:
relatively slow random accesses in the HDFS file system, batch-processing nature
of MapReduce programming paradigm and the sequential data processing. We
concluded that the block-level sampling with an additional bias reduction technique
gives the most benefits (e.g. statistically profound estimation) with a relatively low
performance overhead. We selected the block-level sampling to reduce the impact
of slower random data accesses in HDFS to the overall performance. Due to the
highly optimized sequential processing of the Hadoop MapReduce framework, we
chose to pre-process input data before the Map stage: the data bias reduction is
performed during the reading phase. As a result, Map tasks still can access data
sequentially, with no additional overhead. Furthermore, our system addresses the
"inspection paradox" by balancing the workload of the Map tasks during the job
execution.

The evaluation of our designed system revealed the 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 acceptable accuracy of the estimations.
We demonstrated, that our system can estimate the average temperature of 25 GB
weather dataset with as low as 2% error, up to 6 times faster than a complete job
execution time. However, the sampling technique is not effective in every case: we
showed, that in case of Zipfian data distribution, both our system and the MapReduce
Online framework can return similar results. Nevertheless, we displayed that the
eyear 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. In conclusion,
this work shows promising results of the early estimation techniques adaptation to
the large-scale processing frameworks and should be continued further in order to
obtain the complete production-level solution.

8.1 Future work

There are several open questions for further improvements of our designed system.
First, an additional research should be done to determine the feasible design alterna-
tives of various statistical estimators’ integration into the MapReduce programming
paradigm, which is the foundation of our framework. An investigation of automatic
sampling process adaption over the properties of the system (e.g. block size, available
memory, HDFS properties) is also an interesting topic for a future research. The
idea would be to fine tune the sampling process to reach the best possible perfor-
mance in the each system separately. Lastly, based on our research, we believe that
wavelet-based early approximation techniques can greatly contribute to the future
large-scale processing frameworks, especially for the non-batch processing oriented
jobs (i.e. multiple-jobs execution).
Bibliography


Declaration

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

Stockholm, 03. July 2013

-----------------------------
Vaidas Brundza