An Approach for Processing Large and Non-Uniform Media Objects on MapReduce-based Clusters

Rainer Schmidt and Matthias Rella

Austrian Institute of Technology, Donau-City-Straße 1, 1220 Vienna, Austria
`firstname.lastname[at]ait.ac.at`

Abstract. Cloud computing enables us to create applications that take advantage of large computer infrastructures on demand. Data intensive computing frameworks leverage these technologies in order to generate and process large data sets on clusters of virtualized computers. MapReduce provides an highly scalable programming model in this context that has proven to be widely applicable for processing structured data. In this paper, we present an approach and implementation that utilizes this model for the processing of audiovisual content. The application is capable of analyzing and modifying large audiovisual files using multiple computer nodes in parallel and thereby able to dramatically reduce processing times. The paper discusses the programming model and its application to binary data. Moreover, we summarize key concepts of the implementation and provide a brief evaluation.

1 Introduction

Over the last years, major Internet companies have made their data centers available to public users via cloud services. New resource provisioning models (like PaaS, IaaS) evolved, allowing users to create and host applications that utilize virtually unlimited back-end infrastructures. Cloud computing provides a versatile technology that may be used just for storing data or to create applications that service massive user-requests. A popular web application that uses cloud resources to render videos for user-selected digital content is provided by Animoto [1]. This application gained attention in 2008 when it scaled from about 50 to 4000 instances in less than a week in order to deal with massively increasing user requests. Although supported by the enormous power of a cloud infrastructure, it is a grand challenge to build a web application of that dimension. Most notably, one has to cope with the complexity of coordinating the involved subsystems (like application servers, computing farms, and database systems) in order to achieve scalability and robustness.

In this paper, we target a more generic approach to support the processing of large volumes of digital content in cloud-based environments. We present

[1] http://animoto.com

research on applying a widely used programming model for data-intensive computations to the problem domain of audiovisual (AV) data. The paper describes a method and corresponding application for analyzing video archives based on the MapReduce programming model [1]. The application automates data compression and decomposition by utilizing native codec libraries and parallel processing based on file partitions. It can thereby take advantage of scalable computational environments in order to speed up execution times and throughput. Users are provided with full programmatic control over the processing logic (as opposed to the execution of wrapped 3rd party applications). The application utilizes Java-based abstractions for relevant concepts like containers, tracks, and frames allowing one to easily implement and incorporate custom application logic. We have implemented a set of examples that perform tasks such as applying filters (e.g. for face recognition) or performing text extraction on a collection of arbitrary input video files. A range of use-cases that support digital libraries and archives exist. Examples include error checking an repair of TV/video collections, the application of preservation actions, or the on-demand generation of access copies for different client devices.

2 Data-Intensive Computing

2.1 Execution Environments

In 2004, Dean and Ghemawat introduced MapReduce [1], a programming model and implementation that is capable of processing vast amounts of data on clusters of commodity computers. Data is shared using a distributed file system (GoogleFS) that provides a decentralized storage layer on top of local storage disks, called a shared nothing architecture. Other data-intensive computing environments include Apache Hadoop [2], Microsoft Dryad [5], and Nephele [11], which provide a range of languages, programming models, and runtime implementations for processing data on large storage and computing networks. A major benefit of utilizing such data processing environments compared to using low-level parallel libraries (like MPI) is the automated synchronization and handling of IO operations. Within their application scope, these systems can efficiently handle issues like workload distribution, coordination, and error handling. This in turn enables users to easily implement extremely robust applications that can be executed over thousands of simultaneously running nodes.

2.2 Processing Structured Data

Programming patterns like MapReduce and All-Pairs [6] are specifically designed for implementing applications that perform operations on files-based and structured content. These programming models provide a small set of abstractions allowing their users to express and execute data-intensive workloads. MapReduce provides a relatively simple programming model that is based on two interfaces;

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

one for distributing workload among the cluster nodes, and one for aggregating the results. The programming model has proven to be applicable to a range of problems that deal with the processing of textual data (e.g. graph processing and data mining). It has also been demonstrated that these programming models are applicable to types of scientific applications, like loosely-coupled and massively parallel problems, e.g. found in bioinformatics [4]. Data is typically accessible via an underlying distributed file system, which provide distributed and fault-tolerant storage. Database and data warehouse systems like HBase and HIVE [10] have been built on top of Hadoop's distributed file system (HDFS) providing distributed data stores. These systems relieve the users from the burden of creating applications based on the programming model primitives only and allow them to perform more familiar database operations based on a query language instead.

2.3 Application to Binary Data

A range of use-cases for processing archived binary content exist, these include tasks like scaling, transcoding, and feature extraction of images, audio, and video content. An example for handling binary data in an MapReduce application for scientific data analysis is provided by Ekanayake et al. [3]. The authors describe a MapReduce application that must combine different features from generated binary histogram files. Apache Hadoop already provides basic support for processing binary data like for example dealing with compressed input/output files on the distributed file system. When handling large compressed text files (e.g. for performing log file analysis), it is important to extract and process only parts of the data in the map task. Specific binary formats (like Hadoop's SequenceFile) can provide an efficient intermediate representation for handling large data volumes on HDFS supporting serialization, compression, and splitting. However, these binary formats are designed to provide storage representations that efficiently encode large data sets. In order to process audiovisual data, it will be important to implement suitable data abstractions as well as storage handlers to directly access and decode natively binary data on the fly. Once these abstractions are in place, users should be able to utilize the available abstraction (like MapReduce) to easily implement data-intensive operation for processing audiovisual content.

3 Architecture and Application Design

3.1 Approach to the Problem

In the literature, we found existing work that describes a strategy for transcoding video data in a cloud computing scenario, called Split&Merge architecture [7]. The idea is to ensure fixed duration times for video encoding by scaling-up the number of cloud nodes used to process chunks of data depending on the size of the payload file. Here, we take a similar but more generic approach and focus

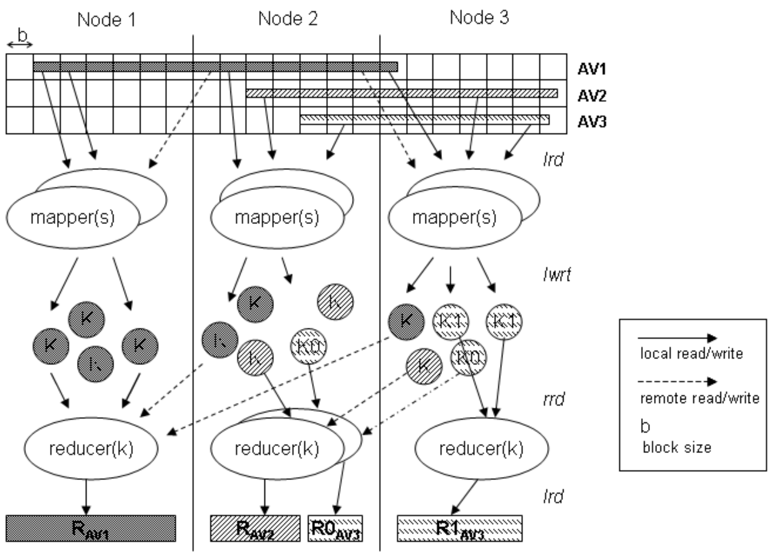


Fig. 1. A data flow for processing audiovisual data using the MapReduce model. This application takes m audiovisual (AV) input files as input and generates n AV output files. In a first phase, the MapReduce framework create a set of mappers for each input file and assigns them to file partitions (*splits*). The Mappers subdivide and process the input splits based on interpretable binary chunks (*records*). For each record, a mapper creates a set of intermediate output data streams which are grouped using keys (K), representing (parts of) the input file. Reducers remotely read the intermediate data streams and generate a result file for each key.

on its application to a widely used programming model (and corresponding execution environment). Designing an application around a standard programming model in general, provides a number of well known benefits like encapsulation, robustness, and portability. In large distributed systems like grids, clusters, or enterprise systems it is key to employ defined programming and deployment models in order to achieve speedup and scalability. The application presented in this paper has been implemented using the MapReduce programming model and Apache Hadoop as an execution environment.

3.2 Data Placement

Figure 1 shows a MapReduce application for processing a set of video files (AV1, AV2, AV3). The files are available on Hadoop's distributed file system (HDFS) which provides a shared storage network across the worker nodes (node1-3). Files that are stored on HDFS are automatically broken up into blocks (typically of 64MB) and stored (and replicated) on different data node based on a placement policy. During runtime, the execution environment facilitates the processing of payload data by assigning parts of the data (called splits) to worker nodes. Input splits are of the same size as storage blocks per default and Hadoop spawns one map task for each split on a worker node. It thereby tries to assign map tasks to workers that reside closely to the input splits in order to maximize local read operations. A number of input splits may be accessed remotely by map operations as the scheduling of map tasks is subject to load balancing.

3.3 Data Decomposition

In order to support the parallel processing of binary input data it is important to provide suitable mechanisms to divide the data into parts that can be interpreted by the application. Most binary formats cannot be broken into chunks at arbitrary positions or do not support splitting at all. For processing sets of relatively small files one can overcome this problem by dividing the payload on a per-file basis [9]. Audiovisual content however tends to be large and its processing can easily become too resource demanding to be performed on a single processor in a reasonable time frame. It is therefore desirable to process single video files using multiple nodes in parallel in order to speed up the overall execution time. This is in particular important for applications that perform such operations on demand based on user requests, for example when triggered via a web interface.

Digital video images (frames) provide a natural unit for decomposing video materials into independently processable parts. Video file formats (like AVI, Flash, Quicktime) are however complex, containing a range of data streams like audio, video or text tracks. The different tracks are typically compressed based on a compression format (like mp3, h264, MJPEG) and must be decoded before being interpretable. One approach to obtain video frames from input splits is to migrate the data into an uncompressed format before processing it within a parallel application [8]. However, the migration itself is a resource and storage consuming process, which hinders the application of this approach for large volumes of content.

The application presented in this paper supports the parallel processing of video content directly from the compressed original formats. The idea is to split videos at key frame positions and perform the decompression of the data chunks in parallel within the MapReduce application. Video compression combines image compression and temporal motion compensation in order to reduce the amount of data required to encode video sequences. Key frames denote independently encoded frames that do not depend on other video data to be decoded. In order to split a media byte-stream into parts it is important to identify these key frame positions within the media container. Using the Hadoop MapReduce framework, we have implemented the required concepts (like *input format*, *record reader*, *compression codec*) that support the automated splitting and parallel processing of compressed media streams (section 4). This allows us to process heterogeneous collections of audiovisual content directly from the storage location without enforcing restrictions on the container and compression formats.

3.4 User Defined Functions

The MapReduce programming model allows users to write parallel applications by implementing the functions *map* and *reduce*. Data portions are automatically generated and passed between the map/reduce functions using a generic data model based on key-value pairs. This programming model allows users to easily implement parallel applications without having to deal with cumbersome parallelization strategies. The model has been widely used for analyzing and transforming large-scale data sets like text documents, database query results, or log files that can reside on different storage systems [2]. In order to enable the development of such user-defined function for processing audiovisual content, we have identified two major requirements: (1) the framework must provide a mechanism to generate meaningful records that can be processed within the map function, and (2) the framework must provide the required abstractions and mechanisms to analyze and transform the records.

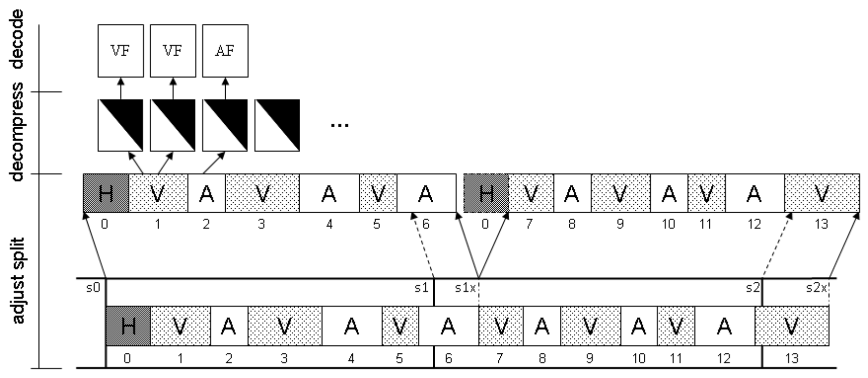


Fig. 2. A data pipeline for extracting records (audio/video frames) from splits of compressed input data. Split boarders (s1, s2) are adjusted (s1x, s2x) based on key frames discovered within the multiplexed data stream. The adjusted input splits together with the container's header information are fed into a decompressor. The extracted packets are finally decoded into interpretable binary records and passed to the map function.

4 Implementation

4.1 Software Stack

The application has been implemented based on Apache Hadoop's distributed file system (HDFS) and MapReduce environment (release 0.21.0). For handling media streams from Java, we utilize the Xuggler [3] open-source library, which provides an very stable wrapper around FFmpeg's *libav* libaries [4]. In previous

[3] http://www.xuggle.com/
[4] http://www.ffmpeg.org/

versions of this application, other Java media frameworks and bindings to native applications have been evaluated (including JMF, FMJ, Theora-Java, FOBS, Jffmpeg, FFMPEG-Java). It is in general desirable to base an applications for user-defined data processing on a high-level language like Java as this can greatly simplify the readability of the code. Due to dependencies to native libraries, the video processing application demands the installation of FFmpeg on every cluster node. Other dependencies may result from the incorporation of native special purpose libraries like e.g. for optical character recognition. It is important to note that virtualization and cloud technology provide a well suited model for efficiently deploying such environments within large data centers on demand. In previous work, we have instantiated a cluster with >150 video processing nodes on Amazon's utility cloud [5] based on a single virtual machine image having the entire software stack pre-installed.

4.2 Application Design

In the following, we provide an overview of the application design, and briefly describe some of the application's basic abstractions and their implementation.

AV Splittable Compression Codec One of the most critical issues when dealing with the parallel processing of large files is handling compression. A compression codec provides a helpful abstraction allowing one to easily read/write from/to compressed data streams. Hadoop provides codec implementations for a set of file compression codecs including gzip, bzip2, LZO, and DEFLATE as part of its API. It is however critical to consider if a file format supports *splitting* for processing it with MapReduce. Hadoop utilizes a specific interface called `SplittableCompressionCodec` to denote codecs that support the compression/decompression of streams at arbitrary positions. Codecs like bzip2 that implement this interface are highly valuable in this context as they support the partitioning and parallel processing of compressed input data. We have implemented *AV Splittable Compression Codec*, a class that supports the compression, decompression, and splitting of audiovisual files.

AV Input Stream In order to split a binary stream, it must be possible to detect positions where the data can be decomposed into blocks. This class implements a splittable input stream for compressed audiovisual content. As shown in figure 2, split boundaries must be repositioned to key frame positions by the codec in order to support decomposition of the data stream. Hence, during execution the reader must be advanced from an arbitrary position within the data stream to the next key frame position. This is done by utilizing a key frame index that is automatically generated from the container prior to the execution. In order to produce an interpretable data stream from the adjusted file split, the stream reader appends the container's header information (kept in memory) to each data portion. It is however not required to read the entire split into memory as the payload is directly read from HDFS.

[5] http://aws.amazon.com

Frame Record Reader Record readers are plugged into the input file format of a particular MapReduce job. They typically convert the data provided by the input stream into a set of key/value pairs (called records) that are processed within the map and reduce tasks. We utilize the concept of packets, which are logical data entities read and uncompressed from the input sequences. Packets are subsequently decoded (optionally error checked and resampled) into objects of a specific data type. For example, a *frame record reader* can utilize the above described concepts in order to obtain audio/video frames from an generic input split.

Output Generation *Record writers* provide the inverse concept to *record readers*. They write the delivered job outputs in the form of key/value pairs to the file system. Output files are produced per reduce task and might have to be merged in a postprocessing stage. To continue the example above, a *frame record writer* writes audio/video frames to an instance of *AV Output Stream* which can be obtained from *AV Splittable Compression Codec*. The codec implementation is customizable regarding the compression formats used to encode the diverse data tracks.

5 Evaluation

In the following, we provide an evaluation that investigates the impact of input file size and compression on the application's performance and scalability.

5.1 Experiment Setup

The evaluation has been conducted on a dedicated testing infrastructure that comprises a front-end and five worker nodes (Single Core 1.86GHz Intel CPU, 1.5GB RAM) connected through Gigabit Ethernet. For benchmarking, the video processing application was configured to decode every video frame of an input file and traverse the content for a given time period. Although the application supports a very large range of formats due to its bindings to FFmpeg, we have utilized a set of homogeneous input files in order to generate comparable results. The files (shown in table 5.1) differ in bitrate and duration only and utilize mp3 (48KHz) and MPEG4 (25fps) as compression formats and AVI as a container. The GOP (Group of Pictures) length basically determines the amount of successive pictures between two key frames, which also influences the achievable compression ratio. The application has been executed for each file sequentially as well as on 1-5 cluster nodes.

Encoding		Input File			
GOP length	bitrate	1800 sec	3600 sec	5400 sec	7200 sec
1	2380 kb/s	535 MB	1071 MB	1607 MB	2141 MB
10	436 kb/s	98 MB	196 MB	294 MB	393 MB
100	341 kb/s	76 MB	153 MB	230 MB	306 MB

Table 1. Payload data file sizes are depending on encoding and duration.

5.2 Results and Improvements

The left part of table 2 shows performance results that have been obtained using a static input split size that corresponds to the file system's block size (i.e. the default configuration). The results show execution times that increase horizontally (with growing duration) as well as vertically (with growing compression rate). Here, a higher compression rate of the payload data has a significantly negative impact on the application throughput. This effect however is caused by an imbalanced workload distribution, as the content is split and distributed using data chunks of a fixed size. This strategy however provides only a reasonably fair workload distribution if every frame is encoded as a key frame (GOP length=1). Compression algorithms like motion compensation disrupt this even density of information within the byte stream. Hence, the size of a byte stream does not provide an adequate measure for the workload it imposes on the application. For video content, it is therefore important to balance the workload (i.e. the uncompressed frames) based on GOPs rather than fixed chunk sizes. We have implemented a simple algorithm that adjusts the split size based on the average GOP, block, and frame size, in order to achieve a better workload distribution. The results in the right part of table 2 were obtained using the dynamic input split adaption algorithm. Here, we see an overall improved throughput rate that is independent of the volumes of content. Also, higher compression rates slightly improve the throughput due to the smaller input file size. Figure 3 shows the application performance on different numbers of nodes. In the tested setting, the application showed almost linear speedup, allowing one to efficiently reduce response times by increasing the number of worker nodes, e.g. important when processing content on demand and/or under SLA constraints.

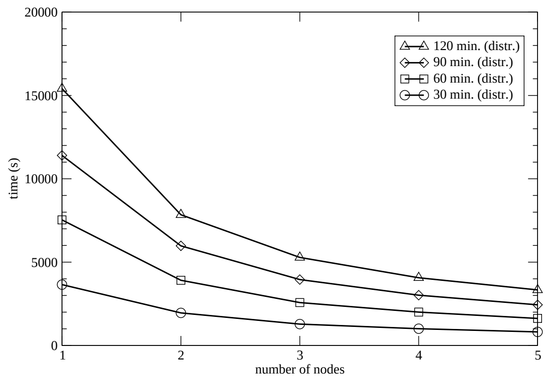
GOP	1800 sec	3600 sec	5400 sec	7200 sec
1	921/49	1720/52	2623/51	3480/51
10	3686/12	3853/23	3890/35	4400/41
100	4910/9	4923/18	4911/27	4944/36

GOP	1800 sec	3600 sec	5400 sec	7200 sec
1	814/55	1624/55	2441/55	3333/54
10	772/58	1499/60	2236/60	2988/60
100	754/60	1440/62	2119/64	2830/64

Table 2. Execution time [sec.] and throughput [frames per second] on 5 nodes with static split size (left) and dynamic split size adaption (right)

6 Conclusion

We have presented an application for the parallel processing of binary media objects based on the MapReduce programming model. The implementation relies on Apache Hadoop and implements the required concepts to generate meaningful records like audio/video frames, which can be processed using common Java abstractions and user-defined logic. Furthermore, we provide insights on interpreting the compressed payload data, as this is highly important in order to assess and balance the application's workload. We motivate the employment of this approach in order to achieve minimal response times for Internet-accessible applications that maintain audiovisual content.



#n	Avg. Speedup	Avg. Efficiency
1	0,99	99,8%
2	1,90	95,2%
3	2,89	96,2%
4	3,74	93,6%
5	4,62	92,4%

Fig. 3. Application performance for different input files on 1-5 worker nodes.

References

1. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51, 107–113 (January 2008)
2. Dean, J., Ghemawat, S.: Mapreduce: a flexible data processing tool. Commun. ACM 53(1), 72–77 (2010)
3. Ekanayake, J., Pallickara, S., Fox, G.: Mapreduce for data intensive scientific analyses. In: eScience, 2008. eScience '08. IEEE Fourth International Conference on. pp. 277 –284 (2008)
4. Gunarathne, T., Wu, T.L., Qiu, J., Fox, G.: Cloud computing paradigms for pleasingly parallel biomedical applications. In: Proc. of the 19th ACM Int. Symposium on High Performance Distributed Computing. pp. 460–469. HPDC '10 (2010)
5. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proc. of the 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems 2007. pp. 59–72 (2007)
6. Moretti, C., Bui, H., Hollingsworth, K., Rich, B., Flynn, P., Thain, D.: All-pairs: An abstraction for data-intensive computing on campus grids. IEEE Transactions on Parallel and Distributed Systems 21, 33–46 (2010)
7. Pereira, R., Azambuja, M., Breitman, K., Endler, M.: An architecture for distributed high performance video processing in the cloud. In: Proc. of the 2010 IEEE 3rd International Conference on Cloud Computing. pp. 482–489. CLOUD '10 (2010)
8. Schmidt, R., Rella, M.: Considering data locality for parallel video processing. ERCIM News 2010(83) (2010)
9. Schmidt, R., Sadilek, C., King, R.: A service for data-intensive computations on virtual clusters. Intensive Applications and Services, International Conference on 0, 28–33 (2009)
10. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., Murthy, R.: Hive - a petabyte scale data warehouse using hadoop. In: Data Engineering (ICDE), 2010 IEEE 26th International Conference on. pp. 996 –1005 (2010)
11. Warneke, D., Kao, O.: Nephele: efficient parallel data processing in the cloud. In: Proc- of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers. pp. 8:1–8:10. MTAGS '09 (2009)