High Throughput Data-Compression for Cloud Storage

Bogdan Nicolae

High Throughput Data-Compression for Cloud Storage

Bogdan Nicolae

University of Rennes 1
IRISA, Rennes, France

Abstract. As data volumes processed by large-scale distributed data-intensive applications grow at high-speed, an increasing I/O pressure is put on the underlying storage service, which is responsible for data management. One particularly difficult challenge, that the storage service has to deal with, is to sustain a high I/O throughput in spite of heavy access concurrency to massive data. In order to do so, massively parallel data transfers need to be performed, which invariably lead to a high bandwidth utilization. With the emergence of cloud computing, data intensive applications become attractive for a wide public that does not have the resources to maintain expensive large scale distributed infrastructures to run such applications. In this context, minimizing the storage space and bandwidth utilization is highly relevant, as these resources are paid for according to the consumption. This paper evaluates the trade-off resulting from transparently applying data compression to conserve storage space and bandwidth at the cost of slight computational overhead. We aim at reducing the storage space and bandwidth needs with minimal impact on I/O throughput when under heavy access concurrency. Our solution builds on BlobSeer, a highly parallel distributed data management service specifically designed to enable reading, writing and appending huge data sequences that are fragmented and distributed at a large scale. We demonstrate the benefits of our approach by performing extensive experimentations on the Grid'5000 testbed.

1 Introduction

As the rate, scale and variety of data increases in complexity, the need for flexible applications that can crunch huge amounts of heterogeneous data (such as web pages, online transaction records, access logs, etc.) fast and cost-effective is of utmost importance.

Such applications are *data-intensive*: in a typical scenario, they continuously acquire massive datasets (e.g. by crawling the web or analyzing access logs) while performing computations over these changing datasets (e.g. building up-to-date search indexes). In order to achieve scalability and performance, data acquisitions and computations need to be *distributed at large scale* in infrastructures comprising hundreds and thousands of machines [1].

However, such large scale infrastructure are expensive and difficult to maintain. The emerging cloud computing model [2, 20] is gaining serious interest

from both industry and academia, as it provides a new paradigm for managing computing resources: instead of buying and managing hardware, users rent virtual machines and storage space. In this context, data-intensive applications become very attractive, because users that need to process huge amounts of data and cannot afford to maintain their own large-scale infrastructure can rent the necessary resources to run their applications, paying only for the resources the application has consumed throughout its execution time.

Since data intensive applications need to process huge amounts of data, a huge amount of storage space is required. Moreover, processing such huge amounts of data in a scalable fashion involves massively parallel data transfers among the participating nodes, which invariably leads to a high bandwidth utilization of the underlying networking infrastructure. In the context of cloud computing, storage space and bandwidth are resources the user has to pay for. It is therefore crucial to *minimize storage space and bandwidth utilization* for data-intensive applications, as this directly translates into lower overall application deployment costs.

In order to achieve scalable data processing performance, several paradigms have been proposed, such as MapReduce [3], Dryad [9] and parallel databases [4]. To optimally exploit the data parallelism patterns that are application-specific, such approaches typically force the developer to *explicitly* handle concurrent data accesses. Consequently, these highly-scalable approaches place a heavy burden on the data storage service, which must deal with massively parallel data accesses in an efficient way. Thus, the storage service becomes a critical component on which the whole system performance and scalability depend [6, 8].

To efficiently deal with massively parallel data accesses, the service responsible for data storage needs both to provide a *scalable aggregation of storage space* from the participating nodes of the distributed infrastructure with minimal overhead, as well as to sustain a *high throughput under heavy access concurrency*. This last issue is particularly important in the context of data-intensive applications, because a significant part of the processing is actually spent on accessing the data, which means a high data access throughput is a key factor in reducing the overall computation time.

Several techniques exist (such as data striping and avoiding synchronization as much as possible) that are highly scalable at achieving a high throughput under heavy access concurrency, but they invariably lead to a high bandwidth utilization.

Therefore, we are faced with a dilemma: on one side it is important to conserve storage space and bandwidth, but on the other side it is important to deliver a high data-access throughput under heavy access concurrency, which means a high bandwidth utilization.

This paper focuses on evaluating the benefits of applying data compression *transparently* at the level of the storage service in the context of data-intensive applications, with the purpose of solving the dilemma mentioned above: conserving both storage space and bandwidth while delivering a high throughput under heavy access concurrency.

Our contribution can be summarized as follows:

- We propose a generic sampling-based compression technique that dynamically adapts to the heterogeneity of data in order to deal with the highly concurrent access patterns issued by data-intensive applications.
- We apply our proposal to improve *BlobSeer* [13–15], a data management service specifically designed to address the needs of data-intensive applications.
- We perform extensive experimentations on the Grid5000 testbed [11] in order to demonstrate the benefits of our approach.

2 Our approach

In this section we present an adaptive transparent compression technique which aims at reducing the storage space and bandwidth needs with minimal impact on I/O throughput when under heavy access concurrency. We introduce a series of key constraints and design principles and show how to apply them to real-life storage services by implementing them into BlobSeer.

2.1 General considerations

Compression does not come for free. We identified the following factors that influence the benefits of adopting data-compression in the context of data-intensive applications:

Transparency. Compression/decompression can be performed either at application level by the user explicitly or it can be handled transparently by the storage service. Explicit compression management may have the advantage of enabling the user to tailor compression to the specific needs of the application, but this is not always feasible.

Many applications are build using high-level paradigms specifically designed for data-intensive applications (such as MapReduce [3]). This paradigms abstract data access, forcing the application to be written according to a particular schema which makes explicit compression management difficult.

For this reason it is important to integrate compression in the storage service and handle it *transparently*.

Heterogeneity of data. First of all, compression is obviously only useful as long as it shrinks the space required to store a chunk of data. Data-intensive applications typically process a wide range of unstructured data.

One type of input is text, such as huge collections of documents, web pages and logs [17]. This is an advantageous scenario, because high compression ratios can be achieved.

Another type of input is multimedia, such as images, video and sound [18]. This type of data is virtually not compressible and in most cases trying to apply any compression method on it actually increases the required storage space. For this reason, the choice of applying compression is highly dependent on the type of data to be processed and the storage service needs to adapt accordingly.

Computational overhead. In the case of data-intensive applications, a big proportion of the processing is spent on transferring huge data sizes. Maximizing the data access throughput is therefore a high priority.

Compression and decompression invariably leads to a computation overhead that diminishes the availability of compute cores for effective application computations. Therefore, this overhead must be taken into account when calculating the data-access throughput. With modern high-speed networking interfaces, high compression rates might become available only at significant expense of computation time. Since the user is paying not only for storage space and bandwidth, but also for compute-time, choosing the right trade-off is difficult and depends both on the offer of the provider and the access pattern of the application.

Memory overhead. Processing huge volumes of data in a distributed fashion generally uses up a large amount of main memory from the participating machines. Moreover, it is common to use the machines that perform the computation for storage as well, which in turn needs significant amounts of main memory for caching purposes. Given this context, main memory is a precious resource that has to be carefully managed. It is therefore crucial to apply a compression method that consumes a minimal amount of extra memory.

2.2 Design principles

In order to deal with the issues presented above, we propose the following set of design principles:

Overlapping of compression with I/O. A straight-forward way to apply compression is to compress the data before sending it for storage when writing, and receive the compressed data and decompress it when reading respectively. However, this approach has a major disadvantage: the compression/decompression does not run in parallel with the data transfer, potentially wasting computational power that is idle during the transfer. We propose the use of data striping: the piece of data is split into chunks and each chunk is compressed independently. This way, in the case of a write, a successfully compressed chunk can be sent before all other chunks have finished compressing, while in the case of a read, a fully received chunk can be decompressed before all other chunks have been successfully received. Moreover, such an approach can benefit from multicore architectures, avoiding having cores sit idle during I/O.

Sampling of chunks. Since the system needs to adapt to both compressible and incompressible data, we need a way to predict whether it is useful to apply compression or not. For this reason, each chunk is sampled, that is, compression is attempted on a small random piece of it. Under the assumption that the obtained compression ratio predicts the compression ratio that would have been obtained by compressing the whole chunk itself, the chunk will be compressed only if the compression ratio of the small piece of random data is satisfactory.

Configurable compression algorithm. Dealing with the computation and memory overhead of compressing and decompressing data is a matter of choosing the right algorithm. A large set of compression algorithms have been proposed in the literature that trade off compression ratio for computation and memory overhead. However, since compression ratio relates directly to storage space and bandwidth costs, the user should be allowed to configure the algorithm in order to be able to fine-tune this trade-off according to the needs.

2.3 BlobSeer

This section introduces BlobSeer, a distributed data management service designed to deal with the needs of data-intensive applications: *scalable aggregation of storage space* from the participating nodes with minimal overhead, support to store *huge data objects*, *efficient fine-grain access* to data subsets and ability to sustain a *high throughput under heavy access concurrency*. BlobSeer provides the ideal premises to integrate our design principles presented in Section 2.2.

Data is abstracted in BlobSeer as long sequences of bytes called BLOBs (Binary Large OBject). These BLOBs are manipulated through a simple access interface that enables creating a blob, reading/writing a range of *size* bytes from/to the BLOB starting at a specified *offset* and appending a sequence of *size* bytes to the BLOB. This access interface is designed to support versioning explicitly: each time a write or append is performed by the client, a new snapshot of the blob is generated rather than overwriting any existing data (but physically stored is only the difference). This snapshot is labeled with an incremental version and the client is allowed to read from any past snapshot of the BLOB by specifying its version.

Architecture. BlobSeer consists of a series of distributed communicating processes. Each BLOB is split into chunks that are distributed among *data providers*. *Clients* read, write and append data to/from BLOBs. Metadata is associated to each BLOB and stores information about the chunk composition of the BLOB and where each chunk is stored, facilitating access to any range of any existing snapshot of the BLOB. As data volumes are huge, metadata grows to significant sizes and as such is stored and managed by the *metadata providers* in a decentralized fashion. A *version manager* is responsible to assign versions to snapshots and ensure high-performance concurrency control. Finally, a *provider manager* is responsible to employ a chunk allocation strategy, which decides what chunks are stored on which data providers, when writes and appends are issued by the clients. A *load-balancing* strategy is favored by the provider manager in such way as to ensure an even distribution of chunks among providers.

Key features. BlobSeer relies on *data striping*, *distributed metadata management* and *versioning-based concurrency control* to avoid data-access synchronization and to distribute the I/O workload at large-scale both for data and metadata. This is crucial in achieving a high aggregated throughput for data-intensive applications, as demonstrated by our previous work [13–15].

2.4 Integration with BlobSeer

The proposed design principles are applied to BlobSeer by introducing an additional compression layer on top of the client-side networking layer, which is responsible for remote communication with the data providers.

This compression layer is responsible to filter data chunks transparently, depending on the operation performed by the application.

In case of a write or append, after the data is split into chunks, a small random sample of each chunk is compressed in order to probe whether the chunk is compressible or not. If the achieved compression ratio is higher than a predefined threshold, then the whole chunk is compressed and the result passed on to the networking layer, which is responsible to send it to the corresponding provider. If the achieved compression ratio is lower than the threshold, then the chunk is passed directly to the networking layer without any modification.

In case a read operation is performed, once a chunk that is part requested dataset has been successfully received from the networking layer, the compression layer first checks whether it was compressed at the time it was written or appended. If this is the case, the chunk is decompressed first. Then, it is placed at its relative offset in the buffer supplied by the application where the result of the read operation is supposed to be stored. The whole read operation succeeds when all chunks that form the requested dataset have been successfully processed this way.

In both cases, the compression layer processes the chunks in a highly parallel fashion, potentially taking advantage of multi-core architectures, which enables overlapping of I/O with the compression and decompression to high degree. Careful consideration was given to keep the memory footprint to a minimum, relying in the case of incompressible chunks on *zero-copy* techniques. This avoids unnecessary copies of large blocks in the main memory, which both leaves more memory for the application and speeds up the processing.

The compression layer was designed to be highly configurable, such that any compression algorithm can be easily plugged in. For the purpose of this paper we adopted two popular choices: Lempel-Ziv-Oberhumer(LZO) [16], based on the work presented in [22], which focuses on minimizing the memory and computation overhead, and BZIP2 [19], a free and open source standard compression algorithm, based on several layers of compression techniques stacked on top of each other.

3 Experimental evaluation

In order to evaluate the benefits of our approach, we conduct a series of large-scale experiments that simulate the behavior of typical distributed data-intensive applications.

Data-intensive applications usually continuously acquire massive datasets while performing large-scale computations on these datasets. In order to simulate this behavior, we perform two types of experiments. The first series of

experiments involves concurrent appends of data to the same BLOB and corresponds to the data acquisition part, while the second series of experiments involves reading different parts of the same BLOB and corresponds to the processing part.

In each of these two settings we evaluate the impact of compression on the achieved aggregated throughput and conclude with a short discussion about the conserved storage space.

3.1 Experimental setup

We performed our experiments on the Grid'5000 [11] testbed, a highly configurable and controllable experimental Grid platform gathering 9 sites in France. We used 110 nodes of the Rennes site, which are outfitted with dual-core and quad-core x86_64 CPUs and at least 4 GB of RAM. We measured raw buffered reads from the hard drives at an average of about 60MB/s, using the *hdparm* utility. Internode bandwidth is 1 Gbit/s (we measured 117.5 MB/s for TCP end-to-end sockets with MTU of 1500 B) and latency is 0.1 ms.

3.2 Concurrent appends of data

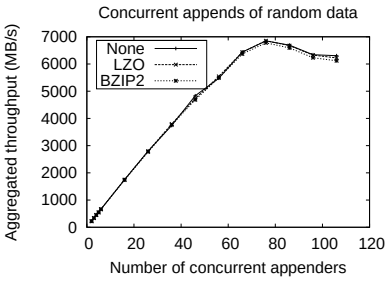
This scenario corresponds to a highly concurrent data acquisition phase, where a data is appended to a BLOB in parallel. We aim at evaluating our approach on the total aggregated throughput, both in the case when the data to be processed is compressible and in the case when it is not.

In each of the cases we measure the aggregated throughput achieved when N concurrent clients append 512 MB of data in chunks of 64MB. We deployed 110 data providers on different nodes, while each of the N clients is co-deployed with a data provider on the same node. We have chosen to co-deploy data providers with clients in order to follow the regular deployment of data-intensive applications: each machine acts both as a storage element and as a processing element. 10 dedicated nodes are reserved to deploy the metadata providers, while the version manager and the provider manager are deployed on dedicated nodes as well. Each data provider is configured to use a cache of 512MB.

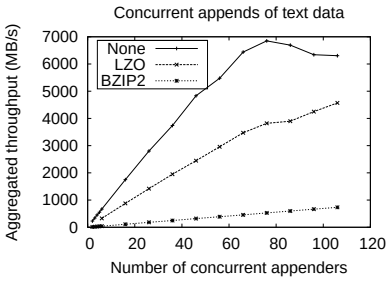
In the first case that corresponds to compressible data, we use the text of books available online. Each client builds the sequence of 512MB by assembling text from those books. In the second case, the sequence of 512MB is simply randomly generated, since random data is the worst case scenario for any compression algorithm.

We perform experiments in each of the cases using our implementation (for both LZO and BZIP2) and compare it to the reference implementation that does not use compression. Each experiment is repeated three times for reliability and the results are averaged. The sample size that enables to decide whether to compress the chunk or not is fixed at 64KB.

The obtained results are represented in Figure 1. The curves corresponding to random data (Figure 1(a)) are very close, clearly indicating that the *impact*



(a) Writing uncompressible random data: high aggregated throughput is ensured by negligible sampling overhead.



(b) Writing compressible text data: high aggregated throughput when using LZO.

Fig. 1. Impact of our approach on aggregated throughput under heavy concurrency. In both cases concurrent clients append each 512 MB of data which is transparently split into 64MB chunks.

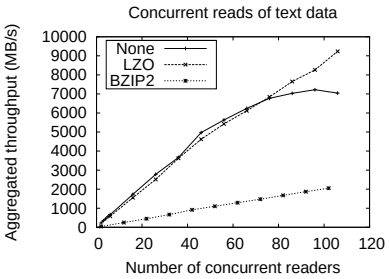
of sampling is negligible, both for LZO and BZIP2. On the other hand, when using compressible text data (Figure 1(a)), the aggregated throughput in the case of LZO, although scaling, is significantly lower than the total aggregated throughput achieved when not compressing data. With less than 1 GB/s maximal aggregated throughput, performance levels in the case of BZIP2 are rather poor.

When transferring uncompressed data, an interesting effect is noticeable: past 80 concurrent appenders, the aggregated throughput does not increase but rather slightly decreases and then stabilizes. This effect is caused both by reaching the total system bandwidth limit and by heavy disk activity caused by the cache of the data providers rapidly filling up.

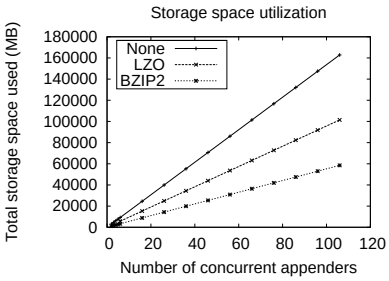
3.3 Concurrent reads of data

This scenario corresponds to a highly concurrent data processing phase, where different parts of the BLOB are read in parallel by different clients in order to be processed. We aim at evaluating the impact of reading compressed data on the total aggregated throughput, assuming it has been written as presented in the previous section. Since reading data that was stored in uncompressed form does not depend on the data type, is sufficient to perform a single set of experiments for text data only.

We use the same deployment settings as with our previous experimentations: 110 data providers on different nodes while each of the N clients is co-deployed with a data provider on the same node; 10 metadata providers, one version manager, one provider manager. We measure the aggregated throughput achieved when N concurrent clients read 512 MB of data stored in compressed chunks, each corresponding to 64MB worth of uncompressed data. Each client is configured to read a different region of the BLOB, such that no two clients access



(a) Reading compressed text data: negligible decompression overhead for LZO reaches high aggregated throughput and outperforms raw data transfers.



(b) Counting the totals: BZIP2 saves more than 60% of storage space and bandwidth utilization. LZO reaches 40%.

Fig. 2. Impact of our approach on compressible text data: concurrent clients read 512MB of compressed data saved in chunks of 64MB (left); total bandwidth and storage space conserved (right).

the same chunk concurrently, which is the typical case encountered in the data processing phase.

As in the previous setting, we perform three experiments and average the results. All clients of an experiment read from the region of the BLOB generated by the corresponding append experiment, i.e. the first experiment reads data generated by the first append experiment, etc. This ensures that no requested data can be found in the cache of the data providers, which have to read the data from the disk in order to satisfy the read requests.

The results are represented in Figure 2(a). Unlike the situation of appends, the transfer of smaller compressed chunks combined with the fast decompression speed on the client side contribute to a steady increase in aggregated throughput that reaches well over 9 GB/s when using LZO compression. In the case of uncompressed data transfers, the aggregated throughput stabilizes at about 7 GB/s in the case of uncompressed data transfers, both because of having to transfer larger data sizes and because of reaching the limits of the system bandwidth. With a maximal aggregated throughput of about 2 GB/s, BZIP2 performs much better at reading data, but the results obtained are still much lower compared to LZO.

3.4 Storage space and bandwidth gains

The storage space gains from storing text data in compressed form are represented in Figure 2(b). With a consistent gain of about 40% of the original size, LZO compression is highly attractive. Although not measured explicitly, the same gain can be inferred for bandwidth utilization too. In the case of BZIP2, the poor throughput described in the previous sections makes up with the storage space and bandwidth gains, which reach well over 60%.

4 Related work

Data compression is highly popular in widely used data-intensive application frameworks such as Hadoop [7]. In this context, compression is not managed transparently at the level of the storage layer (Hadoop Distributed File System [8]), but rather explicitly at the application level. Besides introducing complexity related to seeking in compressed streams, this approach is also not aware of the I/O performed by the storage layer in the background, which limits the choice of optimizations that would otherwise be possible, if the schedule of the I/O operations was known.

Adaptive compression techniques that apply data compression transparently have been proposed in the literature before.

In [10], an algorithm for transferring large datasets in wide area networks is proposed, that automatically adapts the compression effort to currently available network and processor resources in order to improve communication speed. A similar goal is targeted by ACE [12] (Adaptive Compression Environment), which automatically applies on-the-fly compression at the network stack directly to improve network transfer performance. Other work such as [5] applies on-the-fly compression at higher level, targeting an improve in response time of web-services by compressing the exchanged XML messages. Although these approaches conserve network bandwidth and improve transfer speed under the right circumstances, the focus is end-to-end transfers, rather than total aggregated throughput. Moreover, compression is applied in-transit only, meaning data is not stored remotely in a compressed fashion and therefore requests for the same data generate new compression-decompression cycles over and over again.

Methods to improve the middleware-based exchange of information in interactive or collaborative distributed applications have been proposed in [21]. The proposal combines methods that continuously monitor current network and processor resources and assess compression effectiveness, deciding on the most suitable compression technique. While this approach works well in heterogeneous environments with different link speeds and CPU processing power, in clouds resources are rather uniform and typically feature high-speed links, which shifts the focus towards quickly deciding if to apply compression at all, and, when it is the case, applying fast compression techniques.

5 Conclusions

This paper evaluates the benefits of applying *transparent* data compression at the level of the storage service in the context of large-scale, distributed data-intensive applications. As data volumes grow to huge sizes in such a context, we are interested both in *conserving storage space and bandwidth* in order to reduce associated costs. Unlike work proposed so far that focuses on end-to-end data transfer optimizations, we target achieving a *high total aggregated throughput*, which is the relevant I/O metric for large-scale deployments.

Our approach integrates with the storage service and adapts to heterogeneous data dynamically, by *sampling small portions of data on-the fly* in order to avoid compression when it is not beneficial. We *overlap compression and decompression with I/O*, by splitting the data into chunks and taking advantage of multi-core architectures, therefore minimizing the impact of compression on total throughput. Finally, we enable *configurable compression algorithm selection*, which enables the user to fine-tune the trade-off between computation time costs and storage and bandwidth costs.

We show a negligible impact on aggregated throughput when using our approach for uncompressible data thanks to negligible sampling overhead and a high aggregated throughput both for reading and writing compressible data with massive storage space and bandwidth saves.

Thanks to our encouraging results, we plan to explore in future work more adaptability approaches that are suitable in the context of data-intensive application. As mentioned before, uncompressible data is often in the form of multimedia, such as images, video and sound. Because so far we have used lossless compression techniques that cannot be applied successfully to such data, it would be interesting to explore the benefits of lossy compression. Moreover, we have experimented so far with chunk sizes that directly correspond to the chunks sizes internally managed by BlobSeer. Another interesting direction to explore is dynamic adaptation of chunk sizes to the chosen compression algorithm, such as to enable the algorithm to run on optimal chunks sizes.

Acknowledgments

The experiments presented in this paper were carried out using the Grid'5000/ ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/ for details).

References

1. Bryant, R.E.: Data-intensive supercomputing: The case for disc. Tech. rep., CMU (2007)
2. Buyya, R.: Market-oriented cloud computing: Vision, hype, and reality of delivering computing as the 5th utility. Cluster Computing and the Grid, IEEE International Symposium on 0, 1 (2009)
3. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
4. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Commun. ACM 35(6), 85–98 (1992)
5. Ghandeharizadeh, S., Papadopoulos, C., Pol, P., Zhou, R.: Nam: a network adaptable middleware to enhance response time of web services. In: MASCOTS '03: 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems. pp. 136 – 145 (12-15 2003)

6. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS - Operating Systems Review 37(5), 29–43 (2003)
7. The Apache Hadoop Project. http://www.hadoop.org
8. HDFS. The Hadoop Distributed File System. http://hadoop.apache.org/common/docs/r0.20.1/hdfs_design.html
9. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. 41(3), 59–72 (2007)
10. Jeannot, E., Knutsson, B., Björkman, M.: Adaptive online data compression. In: HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing. p. 379. IEEE Computer Society, Washington, DC, USA (2002)
11. Jégou, Y., Lantéri, S., Leduc, J., Noredine, M., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Iréa, T.: Grid'5000: a large scale and highly reconfigurable experimental grid testbed. International Journal of High Performance Computing Applications 20(4), 481–494 (November 2006)
12. Krintz, C., Sucu, S.: Adaptive on-the-fly compression. IEEE Trans. Parallel Distrib. Syst. 17(1), 15–24 (2006)
13. Nicolae, B., Antoniu, G., Bougé, L.: BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency. In: Data Management in Peer-to-Peer Systems. St-Petersburg, Russia (2009), workshop held within the scope of the EDBT/ICDT 2009 joint conference.
14. Nicolae, B., Antoniu, G., Bougé, L.: Enabling high data throughput in desktop grids through decentralized data and metadata management: The blobseer approach. In: Proc. 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09). Lect. Notes in Comp. Science, vol. 5704, pp. 404–416. Springer-Verlag, Delft, The Netherlands (2009)
15. Nicolae, B., Moise, D., Antoniu, G., Bougé, L., Dorier, M.: BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications. In: Proc.24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010) (2010), in press.
16. Oberhumer, M.F.X.J.: Lempel-ziv-oberhumer. http://www.oberhumer.com/opensource/lzo (2009)
17. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data. pp. 165–178. ACM, New York, NY, USA (2009)
18. Raghuveer, A., Jindal, M., Mokbel, M.F., Debnath, B., Du, D.: Towards efficient search on unstructured data: an intelligent-storage approach. In: CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management. pp. 951–954. ACM, New York, NY, USA (2007)
19. Seward, J.: Bzip2. http://bzip.org (2001)
20. Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M.: A break in the clouds: towards a cloud definition. SIGCOMM Comput. Commun. Rev. 39(1), 50–55 (2009)
21. Wiseman, Y., Schwan, K., Widener, P.: Efficient end to end data exchange using configurable compression. SIGOPS Oper. Syst. Rev. 39(3), 4–23 (2005)
22. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23, 337–343 (1977)