Compressing Intermediate Keys between Mappers and Reducers in SciHadoop

Adam Crume, Joe Buck, Carlos Maltzahn, Scott Brandt
University of California, Santa Cruz
{adamcrume,buck,carlosm,scott}@cs.ucsc.edu

***Abstract*—In Hadoop mappers send data to reducers in the form of key/value pairs. The default design of Hadoop's process for transmitting this intermediate data can cause a very high overhead, especially for scientific data containing multiple variables in a multi-dimensional space. For example, for a 3D scalar field of a variable "windspeed1" the size of keys was 6.75 times the size of values. Much of the disk and network bandwidth of "shuffling" this intermediate data is consumed by repeatedly transmitting the variable name for each value. This significant waste of resources is due to an assumption fundamental to Hadoop's design that all key/values are independent. This assumption is inadequate for scientific data which is often organized in regular grids, a structure that can be described in small, constant size.**

Earlier we presented SciHadoop, a slightly modified version of Hadoop designed for processing scientific data. We reported on experiments with SciHadoop which confirm that the size of intermediate data has a significant impact on overall performance. Here we show preliminary designs of multiple lossless approaches to compressing intermediate data, one of which results in up to five orders of magnitude reduction the original key/value ratio.

I. Introduction

Many scientific applications, such as simulations, operate on a regular grid. The most direct representation of the data in a key/value system is to use grid coordinates as keys and store an integer per dimension. However, this is extremely inefficient. Generating a field of 4-byte floats on a $100 \times 100 \times 100$ grid and including a variable index as part of the key, Hadoop creates an intermediate file of 26,000,006 bytes. Since the data is $4 \times 100 \times 100 \times 100$ bytes, this yields an overhead of 450%. (Using a variable name of "windspeed1" instead of a variable index yields a file size of 33,000,006 bytes and an overhead of 625%.) Alternatively, if values can be stored in order and keys are represented in aggregate as a (corner, size) pair, the overhead is reduced to a constant. We show how to accomplish this in SciHadoop[1], a version of Hadoop with changes made to target structured data.

This paper starts with an overview of MapReduce and motivation for this work. Section III describes a byte-level approach for compressing keys, and Section IV describes an application-level approach.

II. Background

A. MapReduce overview

A MapReduce program consists mainly of two parts, a Map function and a Reduce function. The Map function takes a key/value pair in the input key space and outputs key/value pairs in an intermediate key space. The Reduce function takes a key and a set of values in the intermediate key space and outputs key/value pairs in the output key space

The flow of data looks like this (see Figure 1):

1) Several Mappers read the input from HDFS, each taking a portion
2) The Mappers apply the Map function and write the output to disk
3) Optionally, combiners partially reduce Mapper output
4) The intermediate data is transferred across the network from Mappers to Reducers and stored to disk. In general, every Mapper sends a portion of its data to every Reducer.
5) Each Reducer groups intermediate values together by key, possibly requiring multiple on-disk sort phases
6) The Reducers read the sorted data from the disk and apply the Reduce function
7) Output is written back to HDFS

Reducing intermediate data can therefore speed up a write/read cycle on the Mapper hard drives, reduce network transfer sizes, and possibly several read/write cycles on the Reducer hard drives.

B. Assumptions in Hadoop

Hadoop (a MapReduce implementation) makes a number of assumptions about key/value pairs that complicates the use of aggregate keys.

a) Key/value pairs are independent of each other: Hadoop assumes that every key/value pair is independent. The optimizations we wish to make rely on the fact that keys are *not* independent, and in fact are related in a regular pattern. Hadoop uses its assumption in its file format for intermediate data, where every key has a separate field. It also assumes that keys are routed independently, and the user has no information about or control over grouping or dispersal of keys. The user may specify how individual keys are routed to reducers, but this level of control is insufficient for our purposes.

b) Keys are serialized (converted to byte representation) immediately when output from a Mapper: Hadoop converts keys to byte form as the keys are generated by a Mapper. This eliminates the possibility of reordering keys for better compression, as well as any possibility of changing the serialization based on subsequent keys.

c) Key/value pairs are atomic: Hadoop assumes that keys are irreducible and do not overlap. This is relevant for routing

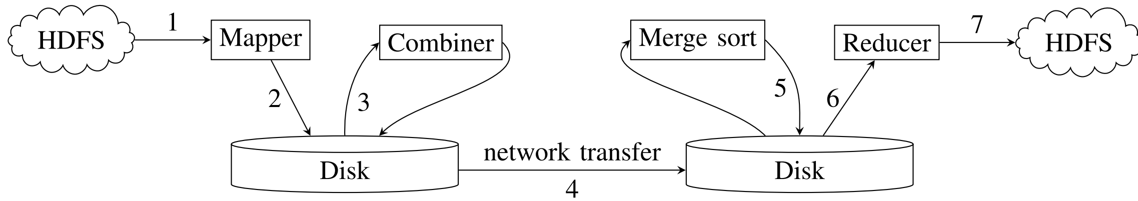

Fig. 1. Data movement in Hadoop

and sorting, and it hampers the usage of aggregate keys. Our changes, detailed in section IV-B, remove this assumption.

III. Semantically-informed byte-level compression

Given the difficulty of changing core Hadoop code, our first approach was to take advantage of Hadoop's pluggable compression and write a custom compression module. Generic compression methods such as GZIP rely on repeating sequences of bytes. A stream of keys generated by walking a grid in a regular patterns creates *almost* identical sequences of bytes (see Fig. 2). The changing bytes greatly hamper compression by introducing bytes that must be literally encoded as well as forcing GZIP to use shorter sequences, rather than using long sequences such as multiples of the simple sequence.

Our approach is to predict these changing bytes and output deltas from our predictions, then run the result through a generic compression such as GZIP. This essentially amounts to using predictive coding[2]. However, by running on top of a generic compression scheme, we retain the ability to compress other data in the stream such as values.

The purpose of transforming the byte stream before compression is to make the byte stream much more amenable to compression. In other words, the compression ratio is much improved by transforming the data beforehand. This improved compression ratio trades extra computation time for less disk and network I/O.

Notice that the marked bytes in Figure 2 increase in a linear sequence. This is typical of traversing a regular grid. Simple and accurate predictions can be made based on linear sequences. Initially, we attempted to detect linear sequences of almost any length at every location. This gives the most accurate predictions and therefore the best compression. Unfortunately, it is far too slow to be practical. In Section III-A, we describe a modification of this algorithm which only attempts to detect a small subset of sequences at any given time. The brute force method is about 4x as slow as the algorithm in Section III-A for a maximum stride length of 100. This grows to a 17x slowdown for a maximum stride length of 1000.

We noticed that one or two linear sequences are enough to achieve most of the compression. These are input-dependent and typically equal to, or a small multiple of, the size of the serialized key/value pair. For the dataset used in Figure 3, a single stride length of 12 yields a bzip2 compressed size of 1619 bytes, versus 701 bytes obtained by using all stride lengths less than 100, which is 99.987% compression versus 99.994% compression. Therefore, the salient question is how to determine the relevant sequences. The most accurate approach is to have the user specify lengths. However, this relies on the user providing accurate information, and we decided an automated approach would be preferable.

Manually specifying an optimal stride length can be difficult with some inputs. A good example is a dataset that contains groups of fixed-length records separated by small markers. The obvious choice for the stride is the length of a record, but the markers break the stride's regularity and reduce compression. The optimal stride actually turns out to be the size of an entire group plus a marker. This would not have been apparent without examining the raw byte stream.

Another method of determining stride length would be to derive it from metadata. This would include the dimensionality of the data, the length of the variable name, and the shape of the data. (The optimal stride length may differ for a 100×2 block of keys and a 2×100 block of keys.) If multiple variables are output, this would require determining where one ends and another begins in the byte stream, because they may have different stride lengths due to different shapes. The same difficulty arises if there are multiple contiguous blocks, even with one variable. This can theoretically be accomplished but requires detailed knowledge of the file format.

A. Detection of linear sequences

During detection, two sets of stride lengths are relevant. One is the full set, which is every stride less than the configured maximum. The other is the active set, which consists of the strides currently being considered. For performance, the active set is usually smaller than the full set, meaning that not every stride is considered at any given moment.

Let x_i be the ith byte in the original stream. A sequence is defined by a stride s and a phase/offset ϕ, and has a difference δ as an attribute. This means that

$$x_{\phi+ks} = x_{\phi+(k-1)s} + \delta \quad (1)$$

for most k.

A table of sequences is maintained, keyed by stride and phase. This table comprises the full set of strides. The difference δ is tracked, as well as the run length, which is the current k minus the largest k for which equation 1 did not hold. In other words, the run length is the number of times in a row that the sequence has predicted the correct value.

```
00 00 00 00 00 00 00 00  00 00 00 00 0a 77 69 6e  |............win|
64 73 70 65 65 64 31 00  00 00 01 00 00 00 01 6c  |dspeed1........l|
88 6c 80 01 01 1f 0e 00  00 00 04 00 00 00 8c 00  |.l..............|
00 00 00 00 00 00 00 00  00 00 01 0a 77 69 6e 64  |...........wind|
73 70 65 65 64 31 00 00  00 01 00 00 00 01 6c 88  |speed1........l.|
6c 8a 01 01 1f 0e 00 00  00 04 00 00 00 8c 00 00  |l...............|
00 00 00 00 00 00 00 00  00 02 0a 77 69 6e 64 73  |..........winds|
70 65 65 64 31 00 00 00  01 00 00 00 01 6c 88 6c  |peed1........l.l|
94 01 01 1f 0e 00 00 00  04 00 00 00 8c 00 00 00  |................|
00 00 00 00 00 00 00 00  03 0a 77 69 6e 64 73 70  |.........windsp|
65 65 64 31 00 00 00 01  00 00 00 01 6c 88 6c 9e  |eed1........l.l.|
01 01 1f 0e 00 00 00 04  00 00 00 8c 00 00 00 00  |................|
00 00 00 00 00 00 00 04  0a 77 69 6e 64 73 70 65  |........windspe|
```

Fig. 2. Encoded stream of keys. Highlighted sequence has δ=0xA, $s = 47$, and $\phi = 34$.

Method	File size (bytes)	Time (seconds)
Original	12,000,000	
gzip	1,630,016	0.977
transform+gzip	33,266	2.858
bzip2	512,740	1.245
transform+bzip2	254	4.004

Fig. 3. Byte-level compression tested on grid points from a $100 \times 100 \times 100$ rectangle

The active set is initialized to be the full set. Predictions are based only to the active set. If the hit rate for a stride falls below a certain threshold (currently 5/6 in the code), and it has been active for at least $2s$ bytes, it is removed from the active set. Hit rate is defined as the percentage of correct predictions for all sequences with a given stride. The $2s$ requirement is tunable and prevents strides from being removed from the active set until the hit rate has a chance to settle. Note that a value of 0 for δ is still valid. This is important because, for keys of length s with only one byte that changes, the hit rate will still be high due to predictions with $\delta = 0$ due to the constant bytes.

Every 256 bytes (one selection cycle), a stride is chosen to be added to the active set. The selection cycle length was chosen to be large enough to reduce CPU overhead and small enough to quickly react to input changes. Priority is given to the strides that have been out of the active set the longest: A stride of s is eligible to be selected only once every s selection cycles. This balances the fact that big strides take longer to get removed from the active set, due to the fact that strides remain in the active set for at least $2s$ bytes.

B. Prediction

For each input byte, the algorithm looks at the table of sequences, one for each stride in the active set. (Because a sequence is determined by stride and phase, there is exactly one sequence for a given stride and byte offset.) The sequence with the longest run length is found. If the run length is greater than a threshold (currently 2), a prediction is made. Letting the byte offset be i, the previous value is x_{i-s}. The prediction is then

$$\hat{x}_i = x_{i-s} + \delta \quad (2)$$

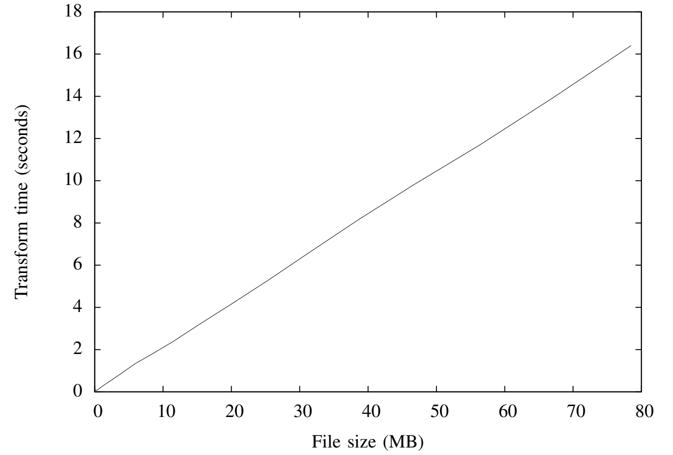

Fig. 4. Transform time versus file size

Let y_i be the ith byte in the transformed stream. The value written to the output stream is

$$\begin{aligned} y_i &= x_i - \hat{x}_i \\ &= x_i - x_{i-s} - \delta \end{aligned} \quad (3)$$

If no sequence has a run length greater than the threshold, the value written to the output stream is simply the input, or in other words, $y_i = x_i$.

C. Inverse transform

The code for the inverse transform is almost identical to that for the forward transform. Data in the sequence tables is computed from the reconstructed original stream. Rearranging (3), we get

$$x_i = y_i + x_{i-s} + \delta \quad (4)$$

if a prediction was made, and $x_i = y_i$ otherwise.

D. Performance

The performance can be seen in Figure 3. The input was a raw stream of triples of 32-bit integers, taken by walking a $100 \times 100 \times 100$ grid. Note that the transform appears to be synergistic with bzip2 and improves compression even more than it does with gzip.

The adaptive transform actually performs better than an exhaustive search of all stride lengths for this dataset (468 bytes versus 701 bytes using bzip2). The reason for this is not immediately clear. An important point, though, is that the compression is applied after the strides are chosen and bytes are predicted, meaning that the algorithm does not directly optimize the size of the compressed data.

Figure 4 shows transform time for varying file sizes. Input files were generated by walking an $n \times n \times n$ grid for varying n. The time to transform the data is linear in the file size. This is expected, since the transform has constant-sized in-memory state and does not look ahead or behind in the input.

E. Results

We ran the sliding median query described in Section IV-C on a 5-node cluster with 5 reducers and 10 map slots, using an $800 \times 800 \times 800$ grid of integers. A custom codec applied the transform and then compressed the data with the built-in zlib compressor. The intermediate data ("Map output materialized bytes") was reduced by 77.8% (from 55.5 GB to 12.3 GB). Unfortunately, total runtime increased by 106% (from 183 minutes to 377 minutes). This is due to the runtime cost of the transform, which is roughly 2.9 times the cost of gzip alone.

IV. Key aggregation

When compression is performed directly on keys instead of on bytes, higher compression is obtained with less overhead[3]. This is the motivation for this section of the work, which is an alternative to Section III and works close to the application instead of at the byte level.

A. Algorithm

Due to the limitations listed in section II-B, the code for the algorithm in the current section is not fully integrated with Hadoop. We have made one set of changes inside Hadoop (detailed in section IV-B), which allows aggregate keys to be split during the routing and sorting phases. Aggregation itself, however, is performed by the user (from Hadoop's point of view). Instead of passing intermediate key/value pairs directly to Hadoop, the user's code passes the key/value pairs to our library. The library aggregates key/value pairs and periodically passes the aggregated key/value pairs to Hadoop.

Ideally, aggregation would be performed directly in the keys' N-dimensional space. Unfortunately, this is difficult (see Fig. 5). Individual keys may join together in multiple ways to form aggregate keys, and some choices may result in more or fewer aggregate keys. We suspect (but have not proven) that this is an NP-hard problem.

To simplify aggregation, the space is reduced to one dimension using a space filling curve (Fig. 6). Each coordinate is mapped to an index on the curve. (Note that the size of each key is not reduced yet, since the mapping is from n 32-bit integers to a single $32n$-bit integer.) Aggregation is then simple; each contiguous range of indices becomes an aggregate key.

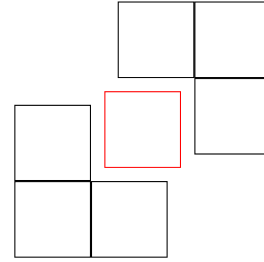

Fig. 5. The middle cell may be put in either group, and the optimal choice is not obvious

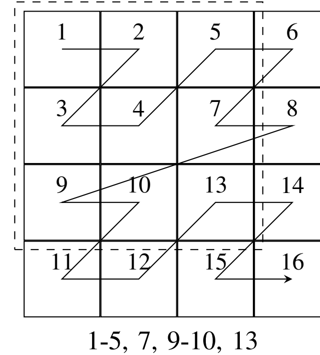

Fig. 6. Cells are numbered with a space-filling curve, and contiguous numbers are collapsed into ranges

Currently, a Z-order curve is used due to speed and ease of implementation. Other curves, such as the Hilbert curve or Peano curve could be used. Moon et al. have shown the Hilbert curve to have better clustering properties than the Z-order curve[4], but the Hilbert curve has more overhead.

Aggregation is performed on subsets of the intermediate data due to memory limitations. Whenever the size of the aggregation buffer reaches a set threshold, the results are written out and the buffer is cleared. This slightly reduces the effectiveness of aggregation, since keys generated after a flush cannot be aggregated with keys generated before a flush, but the effect should be minimal.

B. Key splitting

To support aggregate keys, we must also support splitting of keys. There are two cases where an aggregate key may need to be split:

- A mapper may generate an aggregate key whose simple keys do not all route to the same reducer.
- When sorting keys at a reducer, overlapping keys are split along the overlap boundaries (Fig. 7). This is necessary because unequal overlapping keys contain data that map to the same simple keys, but since the aggregate keys are unequal, the data would not be reduced together.

Aggregation is currently performed only inside mappers. It could also be performed in other places to offset the increase in key count caused by key splitting. We have not yet determined how much the key count is increased by key splitting, or whether further aggregation would be worth the overhead.

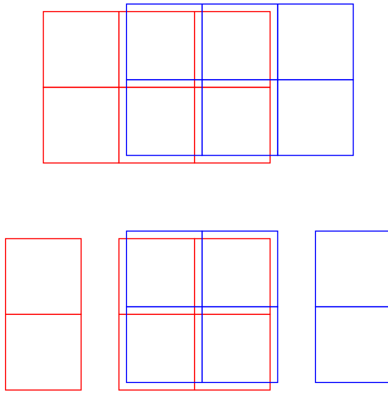

Fig. 7. Key splitting - overlapping ranges are split on the overlap boundaries

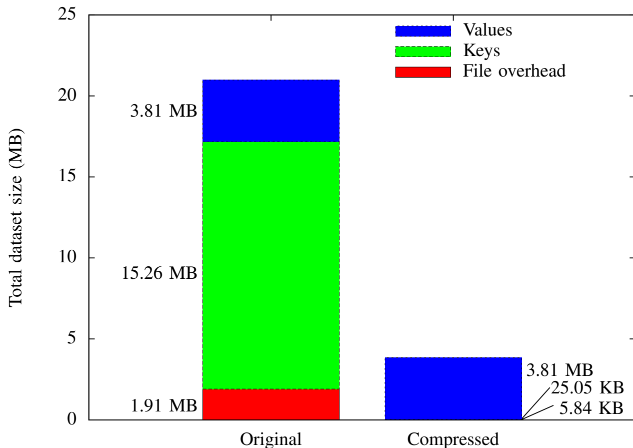

Fig. 8. Effect of key aggregation on total data size for a $100 \times 100 \times 100$ grid of integers

C. Avoiding key overlap

Key splitting could be eliminated if the application developer can guarantee that aggregate keys will never overlap. Unfortunately, this seems difficult or impossible in the general case.

For example, consider computing the median in a sliding 3×3 rectangle. This cannot be computed entirely in the mapper, because the sliding rectangle may cross mapper boundaries. Assume mappers take a value with key (x, y) and output the value for keys (x, y), $(x+1, y)$, $(x+1, y+1)$, etc. Reducers then group the values by key and take the median for each key. This means that a mapper taking input for (0,0)-(9,9) will produce output in (-1,-1)-(10,10). A neighboring mapper responsible for (0,10)-(9,19) will produce output in (-1,9)-(10,20). Note that these aggregate keys are unequal, but overlap in (-1,9)-(10,10).

If keys are allowed to contain empty space, overlap may be reduced by actually expanding the key to a predetermined alignment. If the alignment is large enough, this will increase the probability that overlapping keys will actually be equal. This also adds complexity, storage overhead per aggregate value, and false sharing, so it may not be worthwhile. Also note that in the sliding rectangle case, no alignment is large enough to completely eliminate overlap, because there are always rectangles that straddle the alignment boundary.

Even if key overlap cannot be avoided entirely, reducing key overlap where possible will reduce the amount of key splitting and thereby improve performance.

D. Results

In the ideal case, key aggregation yields up to 84.5% reduction in the size of the intermediate data (Fig. 8), depending on data types. Partitioning the data set across Map tasks results in less aggregation.

We ran the sliding median query described in Section IV-C on a 5-node cluster with 5 reducers and 10 map slots, using an $800 \times 800 \times 800$ grid of integers. The intermediate data ("Map output materialized bytes") was reduced by 60.7% (from 55.5 GB to 21.8 GB). Total runtime was reduced by 28.5% (from 183 minutes to 131 minutes).

The reduction in data is due to reduction in key data and reduction of file overhead. The file format used by Hadoop adds a non-zero overhead per key/value pair. Aggregation greatly reduces the number of key/value pairs, thereby mitigating this overhead.

Due to time and hardware constraints, we only tested input datasets up to $800 \times 800 \times 800$. However, we believe the results hold for larger datasets. All changes were at the level of individual mappers and reducers, leaving communication patterns unchanged, so they should not affect scaling by adding nodes. Additionally, the aggregation and sort/merge/split code is all based on streaming algorithms, so adding more data per node should not be detrimental.

V. Related work

Wang et al. applied MapReduce to spatial data processing and used a space-filling curve for partitioning data among mappers[5]. However, their paper does not mention the size of the intermediate data, which is the primary focus of this work.

ISABELA[6] is a method for compressing floating-point scientific data. While extremely interesting, it is lossy, general purpose, and focuses on data values. The work by Ibarria et al. [7] compresses n-dimensional scalar fields, but again, this operates on data values. Our scheme is intended to compress keys (grid points) based on our knowledge of their regular layout and denseness, and is not intended to compress the values.

Goldstein et al. [8] show how to compress multi-dimensional integer-valued keys for relational database tables. Our work currently focuses on dense keys, but adapting their work may be useful for sparse data.

VI. Conclusion

Combined with SciHadoop, SIDR, and other efforts by our lab, key compression brings traditional scientific HPC applications on an easy to program, fault tolerant system like Hadoop closer to reality.

REFERENCES

[1] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt, “SciHadoop: array-based query processing in Hadoop,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11. New York, NY, USA: ACM, 2011, pp. 66:1–66:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063473

[2] P. Elias, “Predictive coding–i,” *Information Theory, IRE Transactions on*, vol. 1, no. 1, pp. 16 –24, march 1955.

[3] D. A. Lelewer and D. S. Hirschberg, “Data compression,” *ACM Comput. Surv.*, vol. 19, no. 3, pp. 261–296, Sep. 1987. [Online]. Available: http://doi.acm.org.oca.ucsc.edu/10.1145/45072.45074

[4] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz, “Analysis of the clustering properties of the hilbert space-filling curve,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 13, no. 1, pp. 124 –141, jan/feb 2001.

[5] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, and X. Song, “Accelerating spatial data processing with mapreduce,” in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, dec. 2010, pp. 229 –236.

[6] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. Samatova, “Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data,” in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin / Heidelberg, 2011, vol. 6852, pp. 366–379, 10.1007/978-3-642-23400-2_34. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23400-2_34

[7] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, “Out-of-core compression and decompression of large n-dimensional scalar fields,” *Computer Graphics Forum*, vol. 22, no. 3, pp. 343–348, 2003. [Online]. Available: http://dx.doi.org/10.1111/1467-8659.00681

[8] J. Goldstein, R. Ramakrishnan, and U. Shaft, “Compressing relations and indexes,” in *Data Engineering, 1998. Proceedings., 14th International Conference on*, feb 1998, pp. 370 –379.