SciHadoop Semantic Compression

Adam Crume, Joe Buck, Noah Watkins, Carlos Maltzahn, Scott Brandt, Neoklis Polyzotis

University of California, Santa Cruz
{adamcrume,buck,jayhawk,carlosm,scott,alkis}@cs.ucsc.edu

August 16, 2012

Abstract

Many scientific applications, when written in a MapReduce paradigm, naturally use grid coordinates as keys. Unfortunately, a straightforward representation of intermediate keys leads to an enormous amount of overhead. We show how grid coordinates can be stored compactly, yielding a significant reduction in data size. This is an important step in making MapReduce systems such as Hadoop more attractive for developers of scientific applications.

1 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. For a scalar field on a 3D grid, the ratio of metadata to data is 3:1. 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[2], a version of Hadoop with changes made to target structured data.

2 Background

2.1 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:

1. Several Mappers read the input, each taking a portion

2. The Mappers apply the Map function and write the output to disk

3. The intermediate data is transferred across the network from Mappers to Reducers (in general requiring an M to N shuffle, although SIDR[1] reduces this)

4. Each Reducer groups intermediate values together by key, possibly requiring multiple on-disk sort phases

5. The Reducers apply the Reduce function and write the output to disk

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.

2.2 Assumptions in Hadoop

Hadoop (a MapReduce implementation) makes a number of assumptions about key/value pairs that make it difficult to introduce the described optimization.

2.2.1 Key/value pairs are independent of each other

Hadoop assumes that every key/value pair is independent of every other. 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 this assumption in its file format for intermediate data, where every key has a separate field, rather than allowing any form of aggregation or group compression by the user. 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.

2.2.2 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.

2.2.3 Key/value pairs are atomic

Hadoop assumes that keys are irreducible and do not overlap. This is relevant for routing and sorting, and it hampers the usage of aggregate keys. Our changes, detailed in section 4.2, remove this assumption.

3 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. 1). 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. However, by running on top of a generic compression scheme, we retain the ability to compress other data in the stream such as values.

Notice that the marked bytes in figure 1 increase in a linear sequence. This is typical of traversing a regular grid. Simple and accurate predictions can be made by detecting and predicting 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, running about 4x as slow as the algorithm below 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. For the dataset used in Figure 2, 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 de-

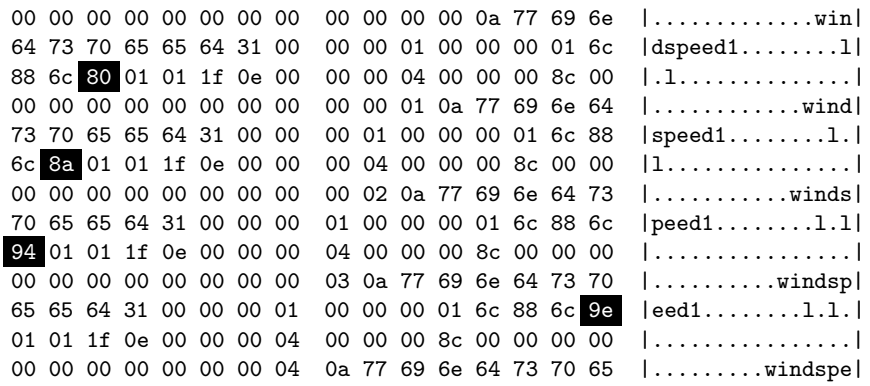


Figure 1: Encoded stream of keys

termine 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.

As an example of the difficulty of specifying a good stride length, we tried to compress a dataset that contained groups of fixed-length records separated by small markers. The obvious choice for the stride is the length of a record, but the markers prevented the algorithm from achieving good compression. The optimal stride actually turned 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 100x2 block of keys and a 2x100 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. 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, and an aggregate-key-aware file format would probably be more feasible. Unfortunately, this is also difficult, especially due to Hadoop's eager serialization of keys.

3.1 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 be the original byte stream, and let y be the transformed stream. A sequence is defined by a stride s and a phase ϕ, and has a difference δ as an attribute. This means that

$$x[\phi + ks] = x[\phi + (k-1)s] + \delta \qquad (1)$$

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.

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. 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, a stride is chosen to be added to the active set. 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.

3.2 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

$$\hat{x}[i] = x[i-s] + \delta \tag{2}$$

The value written to the output stream is

$$\begin{aligned} y[i] &= x[i] - \hat{x}[i] \\ &= x[i] - x[i-s] - \delta \end{aligned} \tag{3}$$

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, $y[i] = x[i]$.

3.3 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

$$x[i] = y[i] + x[i-s] + \delta \tag{4}$$

if a prediction was made, and $x[i] = y[i]$ otherwise.

3.4 Performance

The performance can be seen in Figure 2. The input was a raw stream of triples of 32-bit integers, taken by walking a 100x100x100 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.

4 Key aggregation

When compression is performed at the logical level, higher compression is obtained with less overhead. This is the motivation for this section of the work, which works close to the application instead of at the byte level.

4.1 Algorithm

Due to the limitations listed in section 2.2, the code is not fully integrated with Hadoop. We have made one set of changes inside Hadoop (detailed in the next section), which allows aggregate keys to be split during the routing and sorting phases. Aggregation itself, however, is performed by the user's mapper code calling our library.

Ideally, aggregation would be performed directly in the keys' N-dimensional space. Unfortunately, this is difficult (see fig. 3). Simple keys

Method	File size (bytes)	Time (seconds)
Original	12000000	
gzip	1630016	0.977
transform+gzip	33858	4.366
bzip2	512740	1.269
transform+bzip2	468	5.440

Figure 2: Byte-level compression tested on grid points from a 100x100x100 rectangle

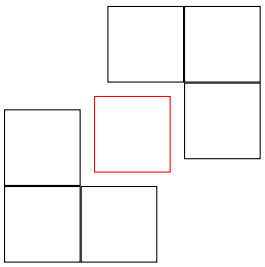
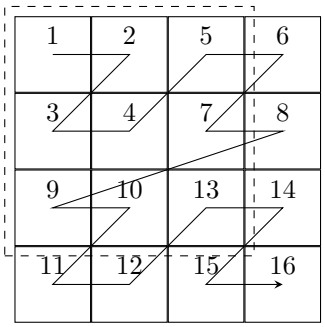


Figure 3: The middle cell may be put in either group, and the optimal choice is not obvious



1-5, 7, 9-10, 13

Figure 4: Cells are numbered with a space-filling curve, and contiguous numbers are collapsed into ranges

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 reduce the difficulty, the space is reduced to one dimension using a space filling curve. An aggregate key is then expressed as a range of contiguous values. Currently, a Z-order curve is used, partly 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 is known to have more overhead.

4.2 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. 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 will 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.

4.3 Avoiding key overlap

Key splitting could be eliminated if the application developer can guarantee that aggregate keys

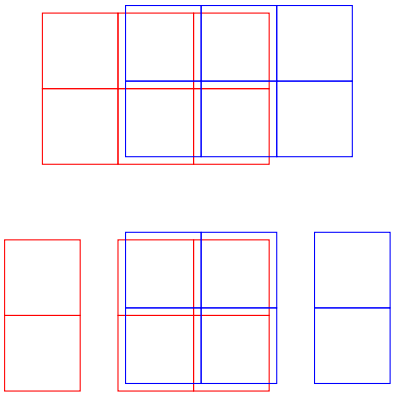


Figure 5: Key splitting - overlapping ranges are split on the overlap boundaries

will never overlap. Unfortunately, this seems difficult or impossible in the general case.

For example, consider computing the median in a sliding 3x3 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.

5 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[3] is a method for compressing floating-point scientific data. While extremely interesting, it is lossy, general purpose, and focuses on data values. Our scheme is intended to compress keys (grid points) based on our knowledge of their regular layout and denseness.

6 Conclusion

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

References

[1] Joe Buck, Noah Watkins, Greg Levin, Adam Crume, Kleoni Ioannidou, Scott Brandt, Carlos Maltzahn, and Neoklis Polyzotis. SIDR: Efficient structure-aware intelligent data routing in SciHadoop. Technical report, Santa Cruz, CA, USA, 2012.

[2] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. SciHadoop: array-based query processing in Hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 66:1–66:11, New York, NY, USA, 2011. ACM.

[3] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 366–379. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-23400-2_34.

[4] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):124 –141, jan/feb 2001.

[5] Kai Wang, Jizhong Han, Bibo Tu, Jiao Dai, Wei Zhou, and Xuan Song. Accelerating spatial data processing with mapreduce. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 229 –236, dec. 2010.