Adaptive Tuple Differential Coding

Jean-Paul Deveaux, Andrew Rau-Chaplin, and Norbert Zeh

Faculty of Computer Science, Dalhousie University, Halifax NS Canada,
jpdeveaux@starcatcher.ca, arc@cs.dal.ca, nzeh@cs.dal.ca

Abstract. It is desirable to employ compression techniques in Relational OLAP systems to reduce disk space requirements and increase disk I/O throughput. Tuple Differential Coding (TDC) techniques have been introduced to compress views on a tuple level by storing only the differences between consecutive ordered tuples. These techniques work well for highly regular data in which the differences between tuples are fairly constant but are less effective on real data containing either skew or outliers. In this paper we introduce Adaptive Tuple Differential Coding (ATDC), which employs optimization techniques to analyze blocks of tuples to detect large tuple differences, with the purpose of isolating them to minimize their negative effect on the compression of neighbouring tuples. Our experiments show that this new algorithm provides an increase in compression ratio of 15–30% over TDC on typical real datasets.

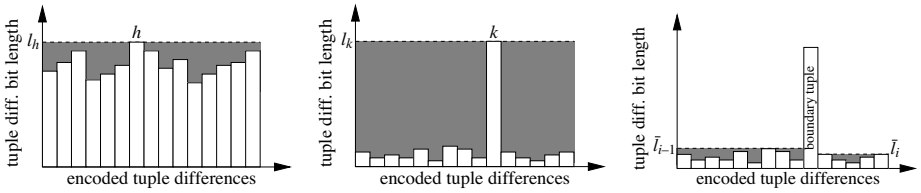
1 Introduction

Many types of information systems, particularly Relational On-Line Analytical Processing (ROLAP) systems, must store ordered multi-dimensional views on disk. Data compression is often critical to their success due to the massive size of the views involved. A properly implemented compression algorithm can save disk space and reduce the overall amount of time required to answer queries, as long as the overhead required to compress and decompress the data is less than the reduction in disk I/O time resulting from the compression.

In order to use compression in a live database environment, compression and decompression has to be fast, and the basic database functionality (insert, query, update in place) has to be retained. This rules out standard general-purpose compression techniques, as they are too computationally expensive and their strength lies in compressing large files rather than, say, individual disk blocks.

Tuple Differential Coding (TDC) techniques initially introduced by Ng and Ravishankar [6] work by storing differences between consecutive tuples and provide view compression that is often superior to traditional compression techniques both in terms of compression ratio achieved and compression and decompression time required. However, while TDC methods perform well on databases where the gaps between successive tuples are reasonably small and constant, they often deteriorate on real data containing either skew or outliers. For an overview of compression techniques as applied to information systems, see [3, 5].

Given a sequence of n tuples $T = [t_1, \ldots, t_n]$ to be encoded, TDC breaks it into subsequences, each of which is stored in a separate block. The subsequence



(a) TDC: Similar differences

(b) TDC: Dissimilar differences

(c) ATDC: Effect of a boundary tuple

Fig. 1. Encoded tuple difference values where the shaded area represents wasted space.

$T[a, b] = [t_a, \ldots, t_b]$ stored in a given block is encoded as follows: For a tuple t_j, let $\phi(t_j)$ be its "standard" encoding discussed in Section 2.1, let $\Delta(t_j) = \phi(t_j) - \phi(t_{j-1})$, let $l_j = \lceil \log_2(\Delta(t_j) + 1) \rceil$, and let l^* be the number of bits required to encode any tuple using encoding ϕ. Then the sequence $T[a, b]$ is stored as the sequence $[\phi(t_a), \Delta(t_{a+1}), \ldots, \Delta(t_b)]$, plus a constant amount of meta-data discussed later. The $\phi(t_j)$ value is stored as a length-l^* bit string. Each of the $\Delta(t_j)$ values is stored using $l_{a+1,b} = \max_{a+1 \le j \le b} l_j$ bits. This saves $(b-a)(l^* - l_{a+1,b})$ bits of space compared to storing values $\phi(t_a), \ldots, \phi(t_b)$ explicitly. The number of tuples stored in a block varies and depends on the number of bits needed to encode the tuples. In particular, tuples are added to blocks one by one. If so far, tuples $t_a, \ldots, t_j$ have been added to the current block, the next tuple t_{j+1} is added to the same block if $l^* + (j + 1 - a)l_{a+1,j+1}$ is no more than the number of bits that fit in a block; otherwise, t_{j+1} starts a new block.

Since the number of bits used to store the difference values in each block is determined by the largest difference value in the block, TDC performs best when the differences $\Delta(t_j)$ in a block are small and do not vary much; more precisely, when the $l_{a+1,b}$ value for the sequence $T[a + 1, b]$ is by only a small constant factor $\alpha > 1$ greater than the average number of bits required to encode these difference values: $l_{a+1,b} \le \alpha \cdot \sum_{j=a+1}^{b} l_j/(b - a)$.

This is illustrated in Figure 1(a). In this figure, the largest difference $\Delta(t_h)$ occurs in position h, forcing us to encode all differences in $T[a + 1, b]$ using $l_{a+1,b} = l_h$ bits. The shaded area represents the number of bits wasted by storing the differences as fixed-length bitstrings compared to encoding each value $\Delta(t_j)$ in l_j bits. However, since most encoded tuple difference values in this block require close to l_h bits to be encoded, padding them to length l_h does not waste much space in this case. Figure 1(b), on the other hand, shows a scenario where a single difference value (at position k) forces us to use significantly more than $\sum_{j=a+1}^{b} l_j$ bits to encode the difference values in the block. As a result, the wasted bits represented by the grey area now account for a major portion of the space used to store the difference values in the block.

This phenomenon should not be unexpected in a dataset made up of real (as opposed to synthetic) data. Real data will contain clusters and other patterns not always found in synthetic datasets. Large tuple differences are representative of "gaps" in real data and should at least be tolerated, if not anticipated.

In this paper we introduce Adaptive Tuple Differential Coding (ATDC), which employs greedy optimization techniques to analyze blocks of tuples to detect large tuple differences, with the purpose of isolating them to minimize their negative effect on the compression of neighbouring tuples. Our experiments show that ATDC provides an increase in compression ratio of 15–30% over standard TDC on typical real datasets, depending on the distribution of the data in the domain space. Additionally, the ATDC algorithm proves to be very robust in situations where there are large gaps due to data skew or outliers. Our experiments show that a good implementation of the ATDC algorithm does not incur any performance penalty compared to reading and writing uncompressed data and compared to a variant of TDC called XTDC [5]. Given the increasing gap between processor speeds and disk access rates, we believe that the savings in I/O-time obtained using the ATDC algorithm will outweigh the investment in compression and decompression time in the near future, thus making ATDC a simple and effective tool for increasing the performance of modern information systems that store relational tables on disk.

Section 2 discusses the ATDC algorithm. Section 3 discusses our experimental results. Concluding remarks are given in Section 4.

2 The ATDC Algorithm

As we have already done for the TDC algorithm in the introduction, we discuss the ATDC algorithm in the context of storing a sequence $T = t_1, \ldots, t_n$ of tuples in a sequence of blocks so that each block can be decoded in isolation.

Structurally, ATDC follows the basic framework of the TDC method. The tuples in the given sequence $T = [t_1, \ldots, t_n]$ are converted into integers $\phi(t_i)$ using an encoding function ϕ. Then the sequence T is divided into subsequences $T_1, \ldots, T_k$, where $T_i = [t_{a_i}, \ldots, t_{b_i}]$, $1 = a_1 < \ldots < a_k \leq b_k = n$, and $b_i = a_{i+1} - 1$, for $1 \leq i < k$. We call the sequences $T_1, \ldots, T_k$ *chunks*. The first tuple t_{a_i} in each chunk T_i is stored explicitly using its encoding $\phi(t_{a_i})$, while all subsequent tuples t_j in T_i are represented by their difference values $\Delta(t_j)$. We call t_{a_i} the *boundary tuple* of T_i. Each chunk T_i is stored as the sequence $[n_i, \bar{l}_i, \phi(a_i), \Delta(t_{a_i+1}), \ldots, \Delta(t_{b_i})]$, where $n_i = b_i - a_i + 1$ is the number of tuples in T_i and $\bar{l}_i = l_{a_i+1,b_i}$; $\phi(a_i)$ is stored as a length-l^* bit string and each $\Delta(t_j)$ is stored as a length-$\bar{l}_i$ bit string.

The key difference between TDC and ATDC is the definition of chunks and their association with physical disk blocks. TDC declares a tuple t_j to be the boundary tuple of a new chunk T_i whenever adding t_j to T_{i-1} would increase the number of bits required to encode T_{i-1} beyond the capacity of a disk block. Each disk block then stores one chunk.

ATDC on the other hand chooses boundary tuples to be those tuples t_j whose difference values $\Delta(t_j)$ are significantly higher than those of the tuples in their vicinity, as these are exactly the tuples that negatively affect the compression ratio of TDC. (We describe the exact choice of these tuples in Section 2.2.) The space savings resulting from this strategy are visualized in Figure 1(c), which

shows the same sequence as Figure 1(b), but encoded using ATDC. By storing tuple t_k as a boundary tuple, we can store the tuples preceding and succeeding t_k in significantly fewer bits than using TDC. The penalty we pay is that we need to store $\phi(t_k)$ explicitly, as well as values n_i and $\bar{l}_i$ for the chunk T_i starting with boundary tuple t_k. Let c_{boundary} be the number of bits required to store the values n_i, $\bar{l}_i$, and $\phi(t_k)$. It is worthwhile to make t_k a boundary tuple if the space savings for storing the difference values before and after t_k exceed $c_{\text{boundary}} - l_k$.

As a result of choosing boundary tuples as described above, chunks may vary greatly in size, and we may end up storing more than one chunk in each disk block. The boundary tuples in all chunks in a disk block are encoded using the same number of bits, as are the tuple differences in each chunk. The number of bits used to encode tuple differences in different chunks, however, may differ.

More precisely, in order to fill disk blocks completely and in order to avoid loading the whole data set to be encoded into memory, we apply the following strategy to pack the data into blocks: Assuming that the blocks we have filled so far store tuples $t_1, \ldots, t_{a-1}$, we load the next $m = b - a + 1$ tuples $t_a, \ldots, t_b$ into memory, encode these tuples using one of the encoding functions discussed in Section 2.1 and apply one of the boundary tuple selection algorithms from Section 2.2 to this sequence. We then iterate over the sequence of tuples and pack them into the current physical block until it is full. For each tuple t_j, if it is marked as a boundary tuple, we add the meta-information of its chunk and $\phi(t_j)$ to the physical block. For any other tuple t_j, we encode $\Delta(t_j)$ using $\bar{l}_i$ bits, where T_i is the chunk containing t_j.

We choose the number m of tuples to be packed into a disk block so that we expect that their encoding requires at least the number of bits that can fit into a block. Thus, we may end up storing only a subsequence $[t_a, \ldots, t_{b'}]$, $b' \leq b$, of tuples in the current block. Tuples $t_{b'+1}, \ldots, t_b$ are reconsidered when filling the next block. On the rare occasion when encoding tuples $t_a, \ldots, t_b$ uses less than a block-full of space, we double the number of tuples we consider and restart the whole packing procedure for the current block.

The following two subsections discuss the two main factors affecting compression ratio and compression/decompression time: the choice of the encoding function ϕ and the selection of boundary tuples.

2.1 Encoding Tuples

We consider two encoding functions: a *mixed-radix* (MR) and a *bit-shift* (BS) encoding. Using the MR-encoding, a tuple $t = (a_1, \ldots, a_d)$ is encoded as

$$\phi(a_1, \ldots, a_d) = \sum_{i=1}^{d} \left(a_i \prod_{j=i+1}^{d} \text{card}(j) \right),$$

where $\text{card}(j)$ denotes the cardinality of dimension j (ie, the values in this dimension are integers between 0 and $\text{card}(j) - 1$). Using the BS-encoding, each value a_j is encoded using $\lceil \log_2 \text{card}(j) \rceil$ bits, and the value of $\phi(a_1, \ldots, a_d)$ is the concatenation of these bit strings.

The advantage of the MR-encoding is that it uses the minimum number of bits necessary to encode all possible tuple values in the given view. It is computationally expensive to decode, however, as computing $\phi^{-1}(e)$ incurs one division per dimension. An encoding using the BS-encoding can be expected to waste half a bit per dimension on average, but its inverse is extremely fast to compute using simple bit-shift operations. We investigate the impact of this trade-off on compression ratio and compression/decompression time in our experiments.

2.2 Selection of Boundary Tuples

The key step in the ATDC algorithm is the selection of boundary tuples. We describe this process as if we were to apply it to the whole sequence $T = [t_1, \ldots, t_n]$. Recall, however, that we apply it only to subsequences $[t_a, \ldots, t_b]$ expected to be long enough to fill a disk block in encoded form.

We define the set of boundary tuples incrementally, starting with t_1 as the only initial boundary tuple. Given the current sequence of boundary tuples, we inspect subsequences $T[a, b]$ that currently do not contain any boundary tuples, choose a candidate tuple t_j, $a \leq j \leq b$, in each such sequence and decide whether making t_j a boundary tuple reduces the amount of space necessary to encode $T[a, b]$; if so, we add t_j to the set of boundary tuples.

We present two algorithms that choose subsequences $T[a, b]$ and the candidate boundary tuples t_j in these subsequences differently. Both algorithms satisfy the condition that encoding $T[a, b]$ takes $(b - a + 1) \cdot l_j$ bits if t_j is not chosen as a boundary tuple, and $(j - a) \cdot l_{a,j-1} + c_{\text{boundary}} + (b - j) \cdot l_{j+1,b}$ bits if t_j is chosen as a boundary tuple. Thus, choosing t_j as a boundary tuple reduces the amount of space required to store the sequence $T[a, b]$ if and only if $(j-a) \cdot l_{a,j-1} + c_{\text{boundary}} + (b-j) \cdot l_{j+1,b} < l_j \cdot (b-a+1)$. In this case, we say that tuple t_j satisfies the *boundary tuple condition with respect to sequence* $T[a, b]$.

Top-down boundary tuple selection. The tuples that are most likely to decrease the cost of storing subsequences of T when chosen as boundary tuples are those whose tuple differences are large relative to the tuple differences of their neighbours. Our first algorithm, called the *top-down boundary tuple selection algorithm* or Top-Down, is based on this observation and can be described recursively: Initially, let t_1 be the only boundary tuple and invoke Top-Down on the sequence $T[2, n]$. Given a sequence $T[a, b]$, let i be the index such that $a \leq i \leq b$ and $l_i = \max_{a \leq j \leq b} l_j$. If tuple t_i satisfies the boundary tuple condition w.r.t. sequence $T[a, b]$, we add t_i to the set of boundary tuples and recurse on sequences $T[a, i - 1]$ and $T[i + 1, b]$. Otherwise, we choose no boundary tuples in $T[a, b]$.

When implemented naively, this algorithm takes $O(n^2)$ time. Next we describe a faster method to implement the Top-Down algorithm, which runs in $O(n)$ time. The key is to store the entire tuple sequence in a *Cartesian tree* [8]. This is a binary tree in which each node has a *key* and a *priority*. The tree is a binary search tree w.r.t. the keys, that is, for every node, the keys in its left subtree are less than the key of the node itself, which in turn is less than the keys in the right subtree; and it is heap-ordered w.r.t. the priorities, that is, no

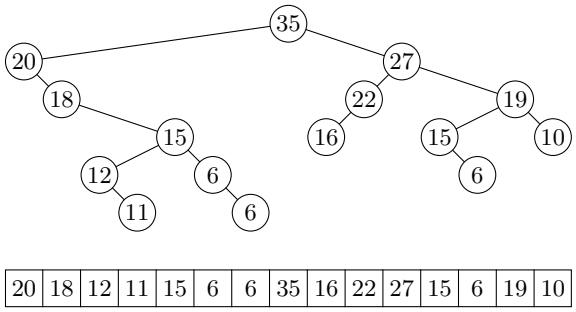


Fig. 2. Example of a Cartesian tree. The values in the boxes denote tuple differences.

node has a greater priority than its parent. In the context of boundary tuple selection, the key of a tuple t_j is its position j in the tuple sequence; its priority is the number of bits l_j required to store $\Delta(t_j)$ (see Figure 2).

A Cartesian tree for a sequence of elements sorted by their keys can be built in linear time [1]. Every node v in the tree can be seen as representing the subsequence T_v consisting of all tuples stored at descendant nodes of v. The construction algorithm is easily augmented to compute the size of T_v for every node v and store this size with v. Once the tree is given, implementing procedure Top-Down in $O(n)$ time is straightforward. In particular, testing whether a node v satisfies the boundary tuple condition translates into the condition $|T_v| \cdot l_v > |T_x| \cdot l_x + c_{\text{boundary}} + |T_y| \cdot l_y$, where x and y are the children of v in T.

Bottom-up boundary tuple selection. The problem with the top-down approach is that it is short-sighted in nature: by deciding that a tuple t_j is an unsuitable choice for a boundary tuple, we do not consider any further tuples in its subtree. It may be, however, that t_j by itself is not a good boundary tuple, while choosing t_j together with additional tuples in its subtree leads to significant compression.

Our second tuple selection algorithm, the *bottom-up boundary tuple selection algorithm* or Bottom-Up, shown in Algorithm 1, tries to address this weakness. In order to choose the boundary tuples in a subsequence T_v represented by a node v, it first considers v's left and right subtrees in isolation and chooses boundary tuples for the subsequences represented by these trees. It then considers the subsequence $T[a, b]$ of T_v, where $a - 1$ and $b + 1$ are the indices of the last boundary tuple chosen in the left subtree and the first boundary tuple chosen in the right subtree, respectively. If v satisfies the boundary tuple condition w.r.t. sequence $T[a, b]$, it is added to the sequence of boundary tuples.

In procedure Bottom-Up, not choosing a node v as a boundary tuple does not prevent us from choosing boundary tuples in v's subtree, as was the case in Top-Down. The down-side of Bottom-Up is that it always traverses the whole tree, while Top-Down can be expected to stop recursing after visiting only a small portion of the tree. Our experiments investigate the resulting trade-off between compression time and compression ratio.

Algorithm 1 BOTTOM-UP(v): Returns a triple (B, l_p, l_s), where B is a list of boundary tuple indexes in T_v, l_p is the number of bits required to encode each tuple difference up to the first boundary tuple in B, and l_s is the number of bits required to encode each tuple difference after the last boundary tuple in B.

```
1  if v has a left child
2     then (L, l_{p,l}, l_{s,l}) ← BOTTOM-UP(left(v))
3     else (L, l_{p,l}, l_{s,l}) ← (∅, 0, 0)
4  if v has a right child
5     then (R, l_{p,r}, l_{s,r}) ← BOTTOM-UP(right(v))
6     else (R, l_{p,r}, l_{s,r}) ← (∅, 0, 0)
7  if L = ∅
8     then a ← index of the leftmost tuple in T_v
9     else a ← (last entry in L) + 1
10 if R = ∅
11    then b ← index of the rightmost tuple in T_v
12    else b ← (first entry in R) − 1
13 if v satisfies the boundary tuple condition w.r.t. sequence T[a, b]
      ▷ This is easily checked using indices a, b, values l_{s,l} and l_{p,r}, and l_v = priority(v).
14    then return the triple (B, l_{p,l}, l_{s,r}), where B is the concatenation of L, a new list
                 node storing the index of v, and R.
15    else if L = ∅
16            then l_p ← l_v
17            else l_p ← l_{p,l}
18         if R = ∅
19            then l_s ← l_v
20            else l_s ← l_{s,r}
21         return the triple (B, l_p, l_s), where B is the concatenation of L and R.
```

3 Experimental Analysis

We evaluated the performance of the ATDC algorithm on a collection of different types of datasets, both in terms of the compression ratios achieved and the time required for compression and decompression. Both the BOTTOM-UP and TOP-DOWN boundary tuple selection algorithms were included in the tests, as were the bit-shifting and mixed-radix encoding techniques. The results of these tests were compared with tests run using an implementation of the XTDC variant of TDC proposed in [5], as well as a Bit Compression algorithm [6].

We conducted all our experiments on an Intel P4 2.8GHz processor with 512KB L2 cache and 1GB Dual-channel DDR333 RAM on a motherboard using the Intel 875P chipset and equipped with a 3ware 7506-8 parallel ATA RAID controller and 3 Maxtor MaxLine Plus II 250GB drives (ATA/133) in RAID Level 5 configuration. The operating system was Debian Linux 2.6.8.

The compression achieved by the Bit Compression algorithm served as the baseline against which the compression ratios of the XTDC and ATDC algorithms were compared. Also, as with the tests conducted in [5], we considered only the attribute dimensions in the compression ratios: since the measure dimensions could have been of some non-categorical datatype, it could not be safely assumed that they could be compressed using the same technique that was applied to the attribute dimensions; so we decided to store them explicitly.

The timing tests we performed on the datasets consisted of measuring the *round-trip compression time* (RTT). The RTT consists of compression time and

decompression time. The *compression time* represents the amount of time required to compress the tuples from their normalized form into a set of boundary tuples and tuple difference values and write them to disk. The *decompression time* refers to the amount of time required to read every boundary tuple and tuple difference value from the disk and convert them back to normalized tuple form. The results of these timing tests include the read and write times for the raw datasets that consisted of 32-bit integers, to show that even with virtually unlimited storage space, there are still speed advantages to using compression.

The uniform and skewed synthetic datasets used in our tests were created using the OLAP data generator described in [2]. Real data was extracted from the HYDRO1K database developed by the U.S. Geological Survey [7].

3.1 ATDC vs. GZIP Compression

Our first experiment compares the performance of ATDC to that of GZIP, a popular compression tool based on the Lempel-Ziv compression scheme.

Table 1 represents a comparison between GZIP compression and ATDC compression on a selection of the datasets used in this project. It should be noted that the compression ratios shown in this table are relative to the size of the original database file and not to the Bit-compressed version of the file as in the other experiments documented in this paper. The reason is that the GZIP algorithm did not achieve any compression whatsoever on the Bit-compressed version of the database, due to the fact that packing tuples tends to eliminate the long common sequences in the input that make GZIP effective.

As we can see, ATDC achieves compression ratios that are roughly twice as high as those achieved by GZIP on the same dataset. Not only are the ATDC compression results much better, the ATDC compression algorithm compresses data in approximately 1/4th the time required by GZIP.

3.2 ATDC vs. TDC Compression

Our final set of experiments compares the performance of ATDC with that of the XTDC algorithm. We consider the standard bit compression [6] to be the baseline against which we compare our compression ratios. To demonstrate that compression in general can lead to significant overall performance improvements, we also include round-trip time measurements for reading and writing the raw tuple sequence without even bit compressing it.

Database	GZIP Size	GZIP Ratio	ATDC Size	ATDC Ratio
Uniform Synthetic	123,749,391	5.818:1	57,230,264	12.581:1
Skewed Synthetic (1.0)	59,534,163	9.406:1	22,217,928	25.205:1
HYDRO1K-Africa	131,143,020	8.232:1	75,554,360	14.228:1
HYDRO1K-North America	96,134,225	8.271:1	62,448,276	12.732:1

Table 1. GZIP compression vs. ATDC compression

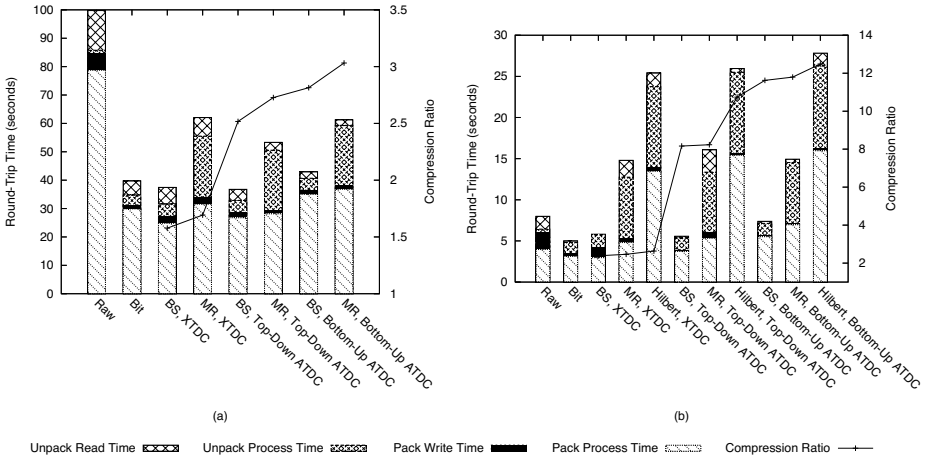


Fig. 3. Average round-trip times and compression ratios: (a) HYDRO1K Africa dataset, $n = 29,987,509$ tuples. (b) Synthetic data, $n = 20,000,000$ tuples. Zipf skew $z = 1.5$.

We tested these methods on a range of datasets. Due to lack of space, we report results on only two data sets here: the Africa data set from the HYDRO1K database [7] (Figure 3(a)) and a synthetic data set skewed with Zipf factor 1.5 (Figure 3(b)). The behaviour of the algorithms on other datasets is similar.

For the Africa dataset, we observe that *all* compression methods significantly decrease the round-trip time compared to the raw dataset. All our adaptive methods produce significantly higher compression ratios than the XTDC algorithm, with the bottom-up approach outperforming the top-down approach and the mixed-radix encoding leading to better compression ratios than the bit-shift approach. This aligns well with our expectations about the performance of these algorithms. Also in line with our expectations, the higher compression ratio in both cases is bought at the expense of increased round-trip times. In the case of the bottom-up algorithm, the time is lost during the compression phase; the mixed-radix encoding requires more time to decode. Note, however, that even using the bottom-up algorithm in combination with bit-shift encoding, the round-trip time is only slightly higher than using bit compression and XTDC. For the top-down algorithm and bit-shift encoding, the round-trip time is slightly lower than using bit compression and XTDC. In general, in terms of compression ratio bottom-up ATDC is superior; however, top-down ATDC may be preferred in applications where the round-trip compression time is critical.

For our skewed data set, the round-trip time improvement over the raw tuple sequence is less pronounced than on the Africa data set; the mixed-radix methods even lead to significantly higher round-trip times, due to high decompression cost. Consistent with the behaviour on the Africa dataset, the bit-shift compression methods produce round-trip times that are competitive with bit compression and XTDC but, due to the skew, now lead to significantly improved compression

ratios. It is interesting to note that, on this data set, there seems to be little gain in compression ratio when using mixed-radix instead of bit-shift encoding. Since the mixed-radix encoding leads to a significantly increased decompression cost, it therefore cannot be considered useful for compressing this type of data.

Figure 3(b) also includes the compression ratios and compression times when applying our ATDC variants to a tuple sequence based on a Hilbert space encoding, which has been shown to be effective in improving query time in parallel OLAP query processing [4]. Although we used an efficient Hilbert implementation that was largely based on simple bit-shifts, the increased processing cost had a detrimental effect on round-trip times, increasing it even beyond that incurred by the mixed-radix methods. It is interesting to note, however, that the compression ratio does not deteriorate; on the contrary, the Hilbert encoded sequences lead to the highest observed compression ratios for each of the XTDC, top-down ATDC, and bottom-up ATDC algorithms. Thus, our compression method can be applied effectively in combination with Hilbert encodings in applications where the cost required for computing the Hilbert encoding is justified. In particular, our ATDC method is potentially a useful tool for reducing the amount of data exchanged between processors in parallel processing of OLAP queries.

4 Conclusion

We have demonstrated that the use of the ATDC algorithm has the potential of providing improved compression over existing algorithms on tuple-based statistical datasets. The algorithm was shown to be effective and robust on synthetically-generated datasets, both uniform and skewed, as well as on real-world datasets. In particular ATDC using a bit-shift encoding achieves both high compression ratios and low round-trip compression times. Furthermore, it must be emphasized that, as the gap between processor and I/O speeds grows, optimizing compression methods like ATDC, which may be encapsulated in the storage layer, will become increasingly appealing.

References

1. M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Alg.*, 57:75–94, 2005.
2. cgmLab. OLAP data generator. `http://cgmlab.cs.dal.ca/downloadarea/`, 2000.
3. Z. Chen and P. Seshadri. An algebraic compression framework for query results. In *Proc. 16th Int. Conf. on Data Eng.*, pp. 177–188, 2000.
4. F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel multi-dimensional ROLAP indexing. In *Proc. 3rd Int. Symp. on Cluster Comp. and the Grid*, pp. 86–93, 2003.
5. B. Liang. *Compressing Data Cube in Parallel OLAP Systems.* Master's thesis, Carleton University, 2004.
6. W.K. Ng and C.V. Ravishankar. Block-oriented compression techniques for large statistical databases. *Knowledge and Data Eng.*, 9(2):314–328, 1997.
7. US Geological Survey. HYDRO1k elevation derivative database, 2003. `http://edcdaac.usgs.gov/gtopo30/hydro/index.asp`.
8. J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23:229–239, 1980.