

Using Column Dependency to Compress Tables

Binh Dao Vo*

Kiem-Phong Vo†

Abstract

Large amounts of business data are kept in tables of fixed-length records. Columns in such a table may be functionally dependent on one another, resulting in low overall information content. This paper shows how to exploit this source of information redundancy to compress table data. Experiments with a wide variety of massive tables including telecom data and stock quotes show that this technique compresses table data well, up to 48:1 or even 100:1 reduction in some cases.

1 Introduction

Tables are sequences of fixed-length records. Corporate data warehouses generate massive tables daily to keep track of on-going operations. For example, a large telephone company may record all telephone calls flowing through its network in daily call detail tables. Each of these tables ranges up to more than two hundred million records. Depending on table types, call detail records may be anywhere between a few hundred to a few thousand bytes long. On a monthly basis, the data generated ranges up to terabytes. Further, these data tables spawn many other tables for purposes such as billing, maintaining business relationships and managing fraud. The cost to store and manage such data can run up to tens and hundreds millions of dollars. It was this high cost that motivated the ground-breaking work by Buchsbaum et al to develop the compressor *Pzip* [5, 6] to compress massive tables.

Tables are similar to databases in that different parts of a record may be functionally related. For example, in a customer data table, the zipcode in an address is usually predictable from the area code of the corresponding telephone number since area codes are by and large defined geographically. A major part in designing a database is to analyze dependency among fields and factor the database tables appropriately to maintain consistency and ease maintenance. The database factorization process helps to reduce storage but this is not its only goal. From an information-theoretic point of view, functional dependency is a form of information redundancy since data in the range of a function can be computed from that in the domain. As such, functional dependency could be exploited to enhance data compression. However there are issues to consider before this can be effectively done. First, where functional dependency among fields in a database can be analyzed from the design of the database, i.e., database schemas, such relations among columns of a table are not readily known. For example, a telephone call detail table may be computed from a database on some legacy system already outsourced to a different company. Thus, even if compression is desirable to reduce data transmission, the operator or tool in use may not know or even have permission to know how the data was generated. Further, relationships among parts of a record in a table are not always perfectly functional even if they may be sufficiently predictable within some limit. For example, an area code may span multiple zipcodes so the area code and zipcode relation is not truly functional even though the correspondence may hold over many records.

Thus, any structure in given data must be automatically learned for the purpose of compression. Following *Pzip*, we treat a table as a two-dimensional array of bytes. Then, we introduce *k-transforms* as a way to capture column dependency to transform table columns in an invertible way. The transformed data can be better compressed by some entropy coding techniques [10, 15] so this approach is similar to block sorting techniques such as the famed Burrows-Wheeler transform [7]. Unfortunately, computing an optimum *k*-transform for $k \geq 2$ is NP-hard. However, when $k = 1$, the problem reduces to computing optimum

*Massachusetts Institute of Technology 77 Massachusetts Avenue, Cambridge, MA 02139, USA, bdv@mit.edu

†AT&T Labs, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA, kpv@research.att.com.

branchings on directed graphs [8] and has efficient solutions. Unfortunately, $k = 1$ does not always produce good compression. So we present a practical, fast, and space-efficient method to compute suitable 2-transforms. A new table compressor was constructed based on the proposed technique. This compressor fully automates the process by learning the table nature of a given dataset (i.e., automatically computing its record length), computing dependency relations among the columns, then compressing the data. We measured compression performance based on an extended set of files starting from the corpus used for testing the *Pzip* program. Experimental results showed that our method could compress 50% better than *Pzip* with faster compression speed.

2 Transforming a table by column dependency

If S is a string of bytes or a sequence of objects, $S[i]$ denotes the i^{th} element of S while $S[i..j]$ denotes the subsequence between positions i and j where $i \leq j$. If S is a sequence and t is an object or another sequence, $S \circ t$ denotes the sequence built by concatenating S and t .

A table T is a sequence of m records, each of length n . That is, T is a 2-dimensional byte array with m rows and n columns. The superscripts r and c are used to denote rows and columns in T . For example, $T^r[i]$ denotes the i^{th} row or the i^{th} record while $T^c[j]$ denotes the j^{th} column or the sequence formed by concatenating the j^{th} byte of each record. We shall think of both $T^r[i]$ and $T^c[j]$ as strings of bytes even though T is in row-major order on input. The superscripts may be omitted whenever the context is clear whether rows or columns are involved.

If $P = (p_1, p_2, \dots, p_k)$ is a sequence of distinct column indices, $T[P]$ is defined as the table formed by juxtaposing in order the columns $T^c[p_1], T^c[p_2], \dots, T^c[p_k]$. To simplify presentation, we shall abuse notation and use P to denote $T[P]$ whenever the context is clear. We occasionally consider an empty sequence P . In that case, we think of $T[P]$ as a sequence of m empty strings.

A *dependency relation* is a pair (P, c) in which P is a sequence of distinct column indices (possibly empty) and c a column index not in P . If the length of P is less than or equal to k , we say that (P, c) is a k -*relation*. For reasons that will become clear later, we say that P is the *predictor* of c or, alternatively, c is *predicted* by P .

2.1 Reversible column sorting

Given a dependency relation (P, c) , data in c can be made more amenable to compression by sorting c based on P . To this end, we define the sorted index \mathcal{I}_P and sorted column $\mathcal{S}_{P,c}$ as follows:

- If P is empty, \mathcal{I}_P is defined to be the identity permutation of the index sequence $(0, 1, \dots, m - 1)$. Otherwise, \mathcal{I}_P is the unique permutation formed by stably sorting the same indices using the lexicographic order of the rows of $T[P]$.
- $\mathcal{S}_{P,c}$ is formed by permuting column $T[c]$ based on \mathcal{I}_P . That is, $\mathcal{S}_{P,c}[k] = T[c][\mathcal{I}_P[k]]$ for $0 \leq k < m$.

Column $T[c]$ can be recovered from $\mathcal{S}_{P,c}$ as long as we have P . This is done by first computing \mathcal{I}_P , then setting $T[c][\mathcal{I}_P[k]] = \mathcal{S}_{P,c}[k]$ for $0 \leq k < m$.

908771aaaa 07922	908	2
973360bbbb 07932	973	3
908464cccc 07922	908	2
973360dddd 07932	973	3

Figure 1: A table of phone numbers and zipcodes

Figure 1 shows examples of the defined constructs. The left side of the figure shows a table T with 4 records, each of which consists of a 10-digit phone number and a corresponding 5-digit zipcode separated by a blank. Let $P = (0, 1, 2)$ and $c = 14$. The right side of the figure shows $T[(0, 1, 2)]$ and $T[14]$. We see that $\mathcal{I}_{(0,1,2)} = (0, 2, 1, 3)$ and $\mathcal{S}_{(0,1,2),14} = 2233$.

2.2 Estimating predictability

Algorithm $\pi(P, c)$

1. Set $\pi = 0$;
2. For $0 \leq i < m$: initialize $map[P^r[i]]$;
3. For $0 \leq i < m$:
 - (a) If $map[P^r[i]] = T^c[c][i]$ then $\pi = \pi + 1$;
 - (b) Set $map[P^r[i]] = T^c[c][i]$;
4. Return π ;

Figure 2: Estimating how well P predicts c .

The example in Figure 1 shows that for a good dependency relation (P, c) , column $T[c]$ is either completely or largely predictable from $T[P]$. Predictability means that the sorted column $\mathcal{S}_{P,c}$ tends to collect same bytes together so it becomes more compressible than the original column $T[c]$. Thus, to help compression, it is beneficial to find for each column some suitable predictor sequence. As there may be many predictor sequences for any given column, the first step is to find a way to quantify predictability. A simple weighting for a dependency relation (P, c) is the length of $T[c]$ minus the number of runs in the sorted column $\mathcal{S}_{P,c}$. Figure 2 shows a weighting function $\pi(P, c)$ that is faster to compute. Step 1 and 2 initialize data. In particular, elements of an associative array map indexed by the rows of table P (i.e., $T[P]$) are initialized to some preselected value (e.g., 0). Step 3 iterates over each row of P to see whether the current mapping matches the corresponding element in column c and, if so, increases the weight π . The element in the map is then updated with the respective element of column c . Note that when P is empty, the convention of treating P as a sequence of m empty strings means that the algorithm estimates the length of column $T[c]$ minus the number of runs in $T[c]$.

It is clear that the cost of running the above algorithm depends on the cost of indexing the associative array map . When $k > 0$, indices are strings of length k so a trie [11] can be used resulting in total cost of $O(km)$ time and space. When $k = 0$, the cost is $O(m)$ for time and $O(1)$ for space (not counting input).

2.3 k -transforms

Let D be a collection of dependency relations $\{(P, c)\}$ and $\omega : D \rightarrow \mathcal{N}$ a weight function mapping elements of D into the natural integers. The weight $\omega(D)$ is defined as $\sum_{(P,c)} \omega(P, c)$. In addition, the dependency graph $\mathcal{G}(D)$ is defined by taking nodes as columns appearing in any dependency relation and expanding each relation (P, c) to edges $p \rightarrow c$ where p is in P . A collection D of dependency relations is said to be a k -transform for table T if:

- Each column of T appears exactly once as the right part of some dependency relation (P, c) ,
- The dependency graph $\mathcal{G}(D)$ is acyclic, and
- Each dependency relation (P, c) is a k -relation.

A k -transform D can be used to transform the data in columns of T by sorting them based on their predictors. Figure 3 shows how to do that. Step 1 of the algorithm is valid because the graph $\mathcal{G}(D)$ is acyclic. Step 2 is well-defined since $(P, \Gamma[c])$ is unique by definition. We note that the transformed data is invertible by following the topological order and inverting each sorted column based on their predictors as discussed in Section 2.1. The sorted permutation \mathcal{I}_P is well-defined at each step since the topological sort order of the columns guarantees the reconstruction of all columns in the predictor P of a column c before c itself is reconstructed.

Algorithm *Transform*(T, D)

1. Sort the nodes of $\mathcal{G}(D)$ topologically to create a sequence of columns Γ .
2. For $0 \leq c < n$: compute and output the sorted column $\mathcal{S}_{P, \Gamma[c]}$ where $(P, \Gamma[c])$ is a relation in D .

Figure 3: Transforming table T by a k -transform D .

The cost of the first step in the algorithm is $O(kn)$ since kn is the maximum number of edges in the graph $\mathcal{G}(D)$. The cost of the second step depends on the cost of sorting the rows of the predictors. Since these are fixed length strings of bytes, they can be sorted using bucket-sort for a total cost of $O(kmn)$ where m is the number of rows in table T . Thus, the total cost of transforming data is bounded by $O(kmn)$. In practice, we use $k = 1$ or $k = 2$ so the transforming cost is linear in the size of data.

2.4 Computing a practical 2-transform

There are many possible k -transforms for any given table T . A k -transform D is said to be optimum if it has maximum weight. For each table T , let $\mathcal{R}(T)$ be the set of all k -relations for T . We construct a hypergraph $\mathcal{H}(T)$ as follows. The hypernodes are either predictor sequences of the k -relations in $\mathcal{R}(T)$ or single columns. The hyperedges are members of $\mathcal{R}(T)$. The hypergraph $\mathcal{H}(T)$ is similar to the k -supergraph considered by Adler and Mitzenmacher [2] except that our hypernodes are ordered. In addition, computing an optimum k -transform for T is equivalent to computing an optimum branching on $\mathcal{H}(T)$. Thus, the results by Adler and Mitzenmacher [2] carry over and show that computing an optimum k -transform is NP-hard for $k \geq 2$.

Algorithm *Plan*(T)

1. Set $D = \phi$.
2. For $0 \leq c < n$:
 - (a) $P = \phi$.
 - (b) For $0 \leq p < c$: If $\pi((p), c) > \pi(P, c)$, set $P = (p)$.
 - (c) Add (P, c) to D .
3. Sort the nodes of $\mathcal{G}(D)$ topologically to create a sequence of columns Γ .
4. For $0 \leq c < n$:
 - (a) Let $(P = (p), \Gamma[c])$ be the 1-relation in D corresponding to $\Gamma[c]$.
 - (b) For $0 \leq q < c$: If $\Gamma[q] \neq p$ and $\pi(p \circ \Gamma[q], \Gamma[c]) > \pi(P, \Gamma[c])$, set $P = p \circ \Gamma[q]$.
5. Return D .

Figure 4: Computing a 2-transform for table T .

When $k = 1$, $\mathcal{H}(T)$ reduces to a directed graph and the induced optimum branching problem has an efficient solution [9]. But our experimentation showed that using two predictors per column tends to produce better compression than just one. In addition, the graph $\mathcal{H}(T)$ requires $O(mn^2)$ time and $O(n^2)$ space to construct. Figure 4 shows how to compute a practical 2-transform based on the prediction weight π (Section 2.2). It works in two phases. In the first phase, a 1-transform is computed by finding for each column c the preceding column p with maximum predicting weight $\pi((p), c)$. For each 1-relation computed in the first phase, the second phase finds an additional predictor column that, together with the first predictor, maximizes predictability.

The algorithm takes $O(n)$ space to store the computed relations. The total number of prediction weight calculations in both phases is bounded by $n(n - 1)$ so the entire algorithm runs in $O(mn^2)$ time. Note that step 2 could be slightly enhanced by computing an optimum branching of the graph $\mathcal{H}(T)$. That would add a factor $\log n$ to the time bound and require $O(n^2)$ space. In practice, the compression improvement is too small to justify this additional complexity.

3 Table compression

3.1 Computing the record length

Data in a table often repeat across records even if the repeated parts may not always be the same. For example, in a telephone call detail table, an area code will likely be repeated across some subset of records due to customers living in the same area. Likewise, many street addresses would have the same town or city names and zipcodes. Due to the fixed length and fixed format nature of the records, such data would likely occur at the same positions in many records. Further, a table is often a step toward reports summarizing decision-making information so records with similar data tend to cluster together. This means that many parts of such a table in row-major order can be thought of as being a quasi-periodic strings where the period is the record length.

Algorithm *RecordLength*(T)

1. For $0 \leq i < t$: set $L[i] = 0$.
2. For $0 \leq i < t$:
 - (a) Let $j > i$ be the least index s.t. $T[i..t - 1]$ and $T[j..t - 1]$ have the longest matching prefix.
 - (b) If there is no such j , skip to next i ;
 - (c) Otherwise, let p be the length of the match, and
 - (d) Set $L[j - i] = L[j - i] + p$.
3. Set $n = 1$.
4. For $2 \leq i < t$:
 - (a) If $L[i] \leq L[n]$, skip to next i .
 - (b) If $n > 1$ and $i \bmod n = 0$, skip to next i .
 - (c) Set $n = i$.
5. Return n .

Figure 5: Estimating the record length of table T .

Let T be a table of t bytes in row-major order with an unknown number of records. Figure 5 shows how to compute n , the number of columns or record length. Here, T is treated as a string of length t . The algorithm works on two assumptions: (1) a significant number of matches occur at the same positions in different records, and (2) records with similar data occur together. The first assumption is necessary for the correctness of the algorithm since it means that the array L will be likely to have peaks appearing at multiples of the record length. The second assumption implies that similar records are clustered together so a high peak in L will likely appear at the record length. This assumption can be alleviated by changing steps 3 and 4 of the algorithm to look for some appropriate common divisor of the high peaks in array L . Although this is an interesting algorithmic variation of the problem, it adds significant complexity and we have not found it to be necessary in practice. The run time of the algorithm depends on finding longest matching prefixes of suffixes in T . This can be done in $O(t)$ time and space using suffix trees [14].

3.2 Compression strategy

Based on the described techniques, the overall strategy of our table compressor is as follows:

1. Use algorithm *RecordLength* to automatically determine the number of columns.
2. Use a suitable sample S of data to compute a 2-transform D , i.e., $D = Plan(S)$.
3. Transform table T according to D , i.e., run $Transform(T,D)$.
4. Run a back-end compressor to compress the transformed data.

Step 1 allows transparent use of the table compressor without any a priori knowledge about the data to be compressed including record length. Steps 2 computes a 2-transform as described in Section 2.4. Typical tables can be very large but only a small sample of data would be needed to compute a good transform. In practice, for tables with a few hundred columns, about a thousand rows of data would suffice. Step 3 applies the computed transform to the data as described in Section 2.3. The transformed data is amenable to compression in the same way that data transformed by the block-sorting Burrows-Wheeler transform [7] would be. In fact, our back-end compressor has the same form as that of a typical Burrows-Wheeler compressor [3], i.e., a composition of move-to-front processing, run length encoding and entropy coding.

3.3 Implementation notes

Below are a few engineering choices in the compressor implementation that have significant effects on compression:

- *Column sorting*: Step 1 of algorithm $Transform(T,D)$ in Figure 3 and step 3 of algorithm $Plan(T)$ in Figure 4 sort columns topologically according to the dependency graph $\mathcal{G}(D)$ but with two further constraints:
 - Columns without predictors are kept together at the front of the sorted list. Since these columns are not transformed by column dependency, our compressor provides an option to transform them together using the Burrows-Wheeler transform. As a non-table dataset can be thought of as a table with a single column, this option would provide for better compression of such data. However, many types of table data have low information content in columns so the back-end compressor would perform adequately already. Applying the Burrows-Wheeler transform in such cases may not be worth the incurred computing cost.
 - The natural order of the columns are maintained as much as possible without violating the above condition and the topological sort order. This is done by modifying the topological sort algorithm to maintain column indices with already sorted predictors in a priority queue. This adds a $\log n$ factor to sorting. Compression is improved somewhat for the data that we deal with, perhaps due to certain data dependency implicit in the column ordering not yet captured by the computed 2-transforms.
- *Back-end compressor*: The table transform is implemented as a data transforming method in the Vcodex package [16]. It can then be composed with the predictive move-to-front transform, the 0-run encoder and the Huffman coder with grouping already available in Vcodex to build a complete compressor. The predictive move-to-front transform of Vcodex uses a learning strategy that creates more 0's in the transformed data than the normal move-to-front transform. The Huffman encoder with grouping computes clusters of data segments that compress well with single Huffman tables. This strategy performs much better than a pure Huffman coder and is competitive to other entropy coders such as arithmetic coding but runs faster.
- *Windowing for large data*: Certain tables to be compressed may be in the hundreds megabytes or even gigabytes range. The compressor divides large tables into segments or windows of sizes that can be entirely processed in memory, typically a few tens of megabytes. The 2-transform described in Section 2.4 is computed only once, then applied to all windows.

4 Performance

Our table compressor was implemented as the option *-t* of the compressor *Vczip* in the Vcodex package [16]. We shall present data comparing *Vczip-t*, *Pzip*, the Burrows-Wheeler compressor *Bzip2* and the Lempel-Ziv compressor *Gzip*. All experiments were performed on a Pentium 4 2.8GHZ processor with 1GB RAM running the Linux Redhat 8.0 operating system. Timing results were obtained by averaging data from 5 different runs. Variations among runs were negligible.

For test data, we started with the corpus used to test the *Pzip* program [5], then added an Excel spreadsheet taken from the Canterbury corpus*, a set of telephone call detail records on the AT&T network, and a one-day set of stock transactions on the New York Stock Exchange. Below are brief descriptions of these files:

- EGF, LRR, PF00032, BACKPQQ, CALLAGEN, CBS: files from the Pfam database of multiple alignments of protein domains [4].
- CYTOB: a file from the AMmtDB database of multi-aligned sequences of vertebrate mitochondrial genes for coding proteins [12].
- CARE: a set of event records from a customer care database of voice call activity.
- NETWORK: a set of event records from a system of network status monitors.
- CENSUS: a portion of the US 1990 *Census of Population and Housing Summary Tape File 3A* [1], field group 301, level 090 for all states.
- LERG: a file from a database describing local telephone switches. Spaces were appended as necessary to make all records have the same length.
- KENNEDY: the *kennedy.xls* Excel spreadsheet from the Canterbury corpus.
- EMC: a set of telephone call records. The records are large but sparse.
- NYSE: a complete record of stock trades on the New York Stock Exchange on September 18, 2003.

Table 1: Compression results

File	Size	Vczip-t	Pzip	Bzip2	Gzip
EGF	533,920	36,264	46,267	51,545	72,305
LRR	235,440	35,059	45,705	49,828	61,745
PF00032	402,512	19,142	28,254	28,811	34,225
BACKPQQ	22,356	5,488	6,477	7,284	7,508
CALLAGEN	242,816	39,808	48,206	57,664	67,338
CBS	73,834	14,723	17,415	21,283	23,207
CYTOB	579,425	49,417	67,157	82,559	109,681
CARE	8,181,810	1,051,539	1,272,841	1,661,885	2,036,277
NETWORK	60,889,500	1,333,435	1,827,589	2,627,757	3,749,625
CENSUS	332,959,796	12,706,667	17,364,441	20,178,769	30,692,815
LERG	3,480,030	137,375	187,162	355,739	454,975
KENNEDY	1,029,744	53,730	53,391	130,280	206,767
EMC	1,146,789,242	11,381,760	14,959,540	27,392,245	53,291,017
NYSE	3,160,251,353	65,174,400	110,992,016	148,847,896	212,052,052

Table 1 shows raw and compressed file sizes for all combinations of files and compressors. Figure 6 shows compression factors, i.e., the ratios of raw sizes over compressed sizes. *Vczip-t* performed best in all

* Available at <http://corpus.canterbury.ac.nz/>.

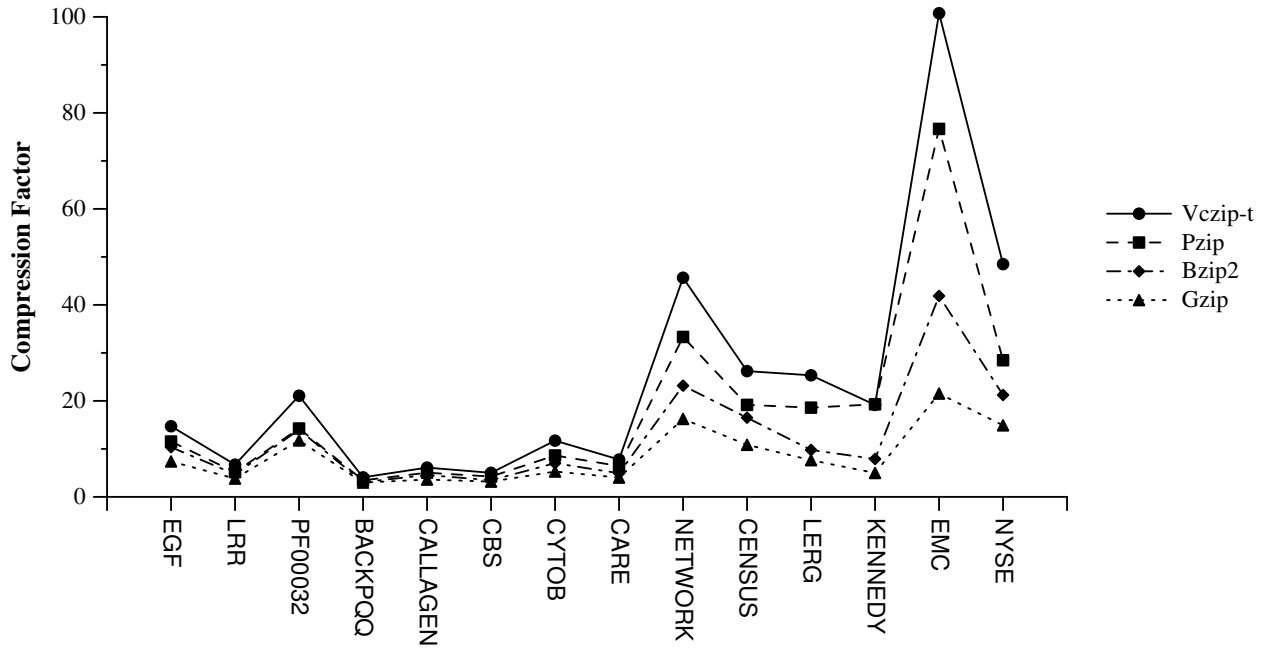


Figure 6: Compression factors.

cases except for KENNEDY where it was slightly worse than *Pzip*. *Vczip-t* did particularly well on the two largest files, EMC and NYSE where the compression factors were about 100:1 and 48:1 respectively. *Bzip2* compressed better than *Gzip* but far worse than *Pzip* and *Vczip-t*.

Table 2: Compression and decompression times in seconds

Data		Vczip-t		Pzip			Bzip2		Gzip	
File	#Cols	Comp	Deco	Plan	Exec	Deco	Comp	Deco	Comp	Deco
EGF	188	0.31	0.03	15.54	0.05	0.02	0.15	0.05	0.04	0.01
LRR	72	0.12	0.02	9.02	0.05	0.02	0.05	0.03	0.02	0.01
PF00032	176	0.30	0.02	261.08	0.05	0.02	0.18	0.03	0.02	0.01
BACKPQQ	81	0.02	0.01	8.474	0.04	0.01	0.01	0.01	0.01	0.01
CALLAGEN	112	0.16	0.01	28.33	0.05	0.02	0.07	0.03	0.04	0.01
CBS	134	0.09	0.01	13.92	0.03	0.01	0.02	0.01	0.01	0.01
CYTOB	1225	3.46	0.04	2488.63	0.09	0.02	0.20	0.09	0.13	0.01
CARE	90	1.16	0.48	316.79	0.85	0.14	3.33	1.13	0.84	0.13
NETWORK	126	3.11	2.46	42.93	1.92	0.48	32.88	4.03	2.43	0.47
CENSUS	932	12.17	9.90	915.48	14.08	2.16	39.55	21.57	17.49	3.27
LERG	30	0.33	0.15	4.54	0.19	0.05	1.18	0.28	0.25	0.04
KENNEDY	13	0.28	0.08	6.91	0.10	0.03	0.25	0.08	0.12	0.01
EMC	2042	30.32	31.51	66.57	13.64	6.37	467.24	86.82	33.80	8.17
NYSE	92	95.80	98.10	79.86	120.90	23.75	1217.39	193.17	135.49	25.05

Table 2 shows timing results. The first column shows the record sizes which range from 13 to 2042. These sizes had substantial effects on the training time of *Pzip* and the time used to compute the k -transforms in our method. They were computed by the algorithm *RecordLength* in Section 3.1. For the files in the *Pzip* corpus, the results matched with the description given by Buchsbaum et al [5]. For KENNEDY, EMC and NYSE, it was unknown whether or not the true record lengths were computed. But, neither way mattered as long as the compression was effective. Compression times for *Pzip* were separated into two parts: computing

a compression plan and executing it. The default dynamic programming training method of *Pzip* was used in the experiments since it produced the best compression results. This algorithm ran in $O(n^3)$ time with n being the number of columns so it was slow for tables with large number of columns. It performed seemingly well for EMC only because this table had many low-entropy columns and *Pzip* excluded them from the dynamic programming computation. Note that the computation of the 2-transforms in *Vczip-t* was naturally included as a part of the compression process. Even with that, *Vczip-t* compressed faster than the execution phase of *Pzip* on CENSUS and NYSE. *Bzip2* compressed extremely slowly the two largest files, EMC and NYSE. This is probably due to the information sparsity in a number of columns in these tables which exerted adverse effect on the suffix sorting algorithm used in computing the Burrows-Wheeler transform. As typical with string matching compressors, *Gzip* decompressed extremely fast. *Pzip* decompressed at about the same rate as *Gzip* since it used the same underlying compression technique after training. For large files, *Vczip-t* compressed and decompressed at about the same speed because, except for the k -transform computation on compression, the compressor performed about the same computation both ways. Thus, *Vczip-t* decompressed slower than *Pzip* and *Gzip*. However, it easily won over *Bzip2*.

5 Discussion

Columns in a table are like fields in a database and dependency among columns is like functional dependency among fields. The sorting technique introduced in Section 2.3 can be extended to sort fields. It would be interesting to see if functional dependency among fields can be used this way to improve compression beyond the use of column dependency.

The *Pzip* compressor first separates columns into groups such that each group is amenable to compression by some preselected conventional compressor. *Gzip* was the preferred choice for its speed and relatively good compression performance. Buchbaum et al showed that a variation of computing an optimum column grouping was MAX-SNP-hard but also presented polynomial time heuristic solutions based on combinatorial optimization. Even these efficient heuristics might take several minutes or hours of CPU time for tables with large number of columns, so the column grouping process could only be done off-line for each class of tables. This approach is vulnerable to degraded performance if columns are changed in some way by addition, deletion or just rearrangement. Our way of computing the 2-transform is fast enough so that it can be applied to each table on-line.

The *XMILL* compressor [13] compresses XML data by grouping related items based on their tags, then applying specialized compressors based on data types to gain better compression. This approach is a direct analogue of column grouping in *Pzip*. That it works means that certain relationships among data with the same tags are being taken advantage of by the specialized compressors. We have shown that column dependency can be directly used for compression. A natural question is to see whether dependency among different parts of XML data can also be directly computed to gain better compression for this class of data.

Pzip, *XMILL* and our approach exploit special structures in data to expose sources of information redundancy and gain better compression. The *RecordLength* heuristic in Section 3.1 provides a way to automatically deduce the table structure of a dataset by computing its number of columns. Beyond this, much of real world data contain mixtures such as multiple tables in the same data file, records with variable lengths, and semi-structured data such as XML or HTML. But as we pointed out, users of a compression tool may not always know the nature of the data that they compress. Thus, a question is how to automatically compute these structures in general data so that proper compression techniques can be applied.

6 Conclusion

We presented a technique to automatically learn the dependency among the columns of a table and use that to transform data to make it amenable to better compression by more conventional entropy encoding methods. We also showed how to automatically discover the table nature of a dataset by computing its number of columns. This enables the new compression technique to be used without requiring users to know anything about the data. Indeed, we have built this technique into a general purpose compressor [16]. Experiments based on a variety of table data from diverse applications ranging from biology to telecom and stock quotes showed that transforming data based on column dependency enabled excellent compression, always several

times smaller than that achievable with well-known conventional compressors such as *Bzip2* and *Gzip* and up to 48:1 and 100:1 size reduction for some common types of data. Our technique also outperformed *Pzip* [5], the only other table compressor known to us. In addition, *Pzip* required a separate and slow training phase before compression while ours was usable on-line. Our compression and decompression speeds were competitive to *Bzip2* and *Gzip*.

Acknowledgements

We would like to thank Adam Buchsbaum, Glenn Fowler and David Korn for many fruitful discussions.

References

- [1] Census of population and housing, 1992.
- [2] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Data Compression Conference*, pages 203–212, 2001.
- [3] B. Balkenhol and S. Kurtz. Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice. *IEEE Transactions on Computers*, 49(10):1043–1053, 2000.
- [4] A. Bateman, E. Birney, R. Durbin, S.R. Eddy, K.L. Howe, and E.L.L. Sonnhammer. The Pfam protein families database. *Nucleic Acids Research*, 28(1):263–266, 2000.
- [5] A. Buchsbaum, D. Caldwell, K. Church, G.S. Fowler, and S. Muthukrishnan. Engineering the Compression of Massive Tables: An Experimental Approach. *Proc. 11th ACM-SIAM Symp. on Disc. Alg.*, pages 175–184, 2000.
- [6] A. Buchsbaum, G.S. Fowler, and R. Giancarlo. Improving Table Compression with Combinatorial Optimization. *Proc. 13th ACM-SIAM Symp. on Disc. Alg.*, pages 213–222, 2002.
- [7] M. Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Report 124, Digital Systems Research Center, 1994.
- [8] J. Edmonds. Optimum Branchings. *J. of Res. of The National Bureau of Standards*, 71B:233–240, 1967.
- [9] T. Spencer H.N. Gabow, Z. Galil and R.E. Tarjan. Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs. In *Proc. 25-th Annual IEEE Symp. on Found. of Comp. Sci.*, pages 347–357, 1984.
- [10] D.A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.
- [11] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.
- [12] C. Lanave, S. Liuni, F. Licciulli, and M. Attimonelli. Update of AMmtDB: A database of multi-aligned Metazoa mitochondrial DNA sequences. *Nucleic Acids Research*, 28(1):153–154, 2000.
- [13] H. Liefke and D. Suciu. Xmill: an efficient compressor for xml data. In *Proceedings of SIGMOD*, pages 153–164, 2000.
- [14] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23:262–272, 1976.
- [15] A. Moffat, R.M. Neal, and I.H. Witten. Arithmetic Coding Revisited. *ACM Trans. on Information Systems*, 16:256–294, 1998.
- [16] Kiem-Phong Vo. Vcodex: A Platform of Data Transformers. *In preparation*, 2003.