RadixZip: Linear Time Compression of Token Streams

Binh Dao Vo
Google, USA
binh@google.com

Gurmeet Singh Manku
Google, USA
manku@google.com

ABSTRACT

RadixZip is a block compression technique for token streams. It introduces RadixZip Transform, a linear time algorithm that rearranges bytes using a technique inspired by radix sorting. For appropriate data, RadixZip Transform is analogous to the Burrows-Wheeler Transform used in bzip2, but is both simpler in operation and more effective in compression. In addition, RadixZip Transform can take advantage of correlations between token streams with no computational overhead. Experiments over practical data show that for common token streams, RadixZip is superior to bzip2.

1. INTRODUCTION

Token streams arise naturally in logging systems and data warehouses, both of which store large volumes of data. Log files created by Google's web-servers, for example, record a variety of events such as clicks on advertisements, queries submitted at `http://www.google.com` and requests to web services like GMail. A log record has (key, value) pairs [24] where the key represents the name and type of the value. At Google, a stream of log records is divided into blocks. Within each block, values are first grouped by key and then compressed. A stream of values with the same key constitutes a token stream.

In relational data warehouses, tables are commonly stored in row-major format, also known as the n-ary Storage Model (NSM). Tables can instead be stored in column-major format, also called the fully decomposed storage model (DSM) by Copeland and Khoshafian [11]. DSM optimizes performance for SQL queries that involve a small set of columns. However, record deletion and updates to multiple columns of a record become expensive. A useful compromise between NSM and DSM is to divide a stream of records into blocks and to store records within a block in column-major format. This idea was introduced in the "partition attributes across" (PAX) format by Ailamaki *et al* [2]. In PAX, a column within a block constitutes a token stream.

VLDB '07, September 23-28, 2007, Vienna, Austria.

Compression of token streams can be carried out by a variety of techniques. If the domain of token values is small and known *a priori*, then the simplest compression technique is to employ a static dictionary that maps values to fixed codes, which themselves can be encoded with standard compression techniques such as Huffman encoding [18]. gzip (based on Liv-Zempel coding [37, 38]) and bzip2 (based on the Burrows-Wheeler Transform [9]) are slower but more powerful techniques. Both can work with token values whose domain is not known. Different methods are used in practice based on access performance requirements. Huffman encoding or gzip are common low-overhead choices for frequently accessed data, and bzip2 is a stronger but more costly choice for long-term storage.

RadixZip Transform is a linear time algorithm for rearranging bytes in a block of tokens using a technique inspired by radix sorting. The algorithm is simple, fast and easy to implement. It replaces the Burrows Wheeler Transform (BWT) in bzip2 for dealing with tokenized streams, and improves in both run-time performance and compression ratio for such data. We also show how RadixZip Transform can be used without additional performance cost to improve compression of multiple correlated token streams.

Experiments over three real world datasets confirm that RadixZip beats bzip2 in terms of both compression ratio and throughput (compression and decompression rates). RadixZip can achieve compression ratios as high as 15:1, beating bzip2 by as much as 10-20%. Compression and decompression rates were also high, beating bzip2 by about 10% overall; for certain streams, the improvement was as high as 10:1. For highly correlated token streams, RadixZip was able to improve over bzip2 by a factor of over 300:1.

Roadmap: An overview of RadixZip is presented in §2. The core routine RadixZip Transform is described in detail in §3. Its output is fed to the backend compressor, which is described in §4. Comparison with Burrows-Wheeler Transform and bzip2 is done in §5. We extend RadixZip to work over correlated streams in §6. Experiments over real datasets are described in §7. The applicability of RadixZip to relational data, XML documents and on Google logs data is outlined in §8. Finally, we survey table compression techniques in §9 before concluding the paper in §10.

2. OVERVIEW OF RadixZip

RadixZip divides a stream of tokens into blocks and compresses each block individually. Block boundaries coincide with token boundaries. Tokens come in two flavors:

Fixed-width tokens: All tokens have identical length. For example, a sequence of 32-bit IP addresses, or a series of strings of equal length can be used as fixed-width tokens.

EXAMPLE 2.1. *A block with fixed-width 3 letter words:*

a s p c o t a s p b o p a s p

The same block in matrix format:

a	s	p
c	o	t
a	s	p
b	o	p
a	s	p

Variable-width tokens: Tokens may not have the same length. For example, a sequence of strings terminated by a special character can constitute variable-width tokens.

EXAMPLE 2.2. *A block of ✧-terminated strings:*

p o t ✧ i t ✧ p o t ✧ a ✧ i t ✧

The same block in matrix format:

p	o	t	✧
i	t	✧	
p	o	t	✧
a	✧		
i	t	✧	

RadixZip compresses a block by applying RadixZip Transform (§3), followed by backend compression (§4). We will show that all of these steps are linear time, giving us:

THEOREM 2.1. *RadixZip is linear time.*

RadixZip shares its backend with bzip2, but replaces the Burrows-Wheeler Transform with RadixZip Transform.

3. RadixZip Transform

The input to RadixZip Transform is a block of b bytes with t tokens. Let c denote the length of the longest token. RadixZip Transform consists of two steps: TRANSPOSE and PERMUTE, which are described in §3.1 and §3.2. Each of these steps preserves size, producing a block with b bytes.

3.1 TRANSPOSE

TRANSPOSE performs matrix transposition by storing tokens in column-major format instead of row-major format.

Fixed-width tokens: The matrix is dense, so the transformation is straightforward. For instance, the block in Example 2.1 is transformed into

a c a b a s o s o s p t p p p

which is a concatenation of the three columns of the matrix: abaca, sosos and ptppp.

Variable-width tokens: The matrix is sparse. For $i \in [1, c]$, the i-th column is stored by concatenating together the i-th bytes of those tokens whose length is i or more. Algorithm 1 contains pseudo-code showing this in detail. For instance, the block in Example 2.2 is transformed into

p i p a i o t o ✧ t t ✧ t ✧ ✧ ✧

which is a concatenation of the four columns of the matrix: pipai, oto✧t, t✧t✧ and ✧✧.

Algorithm 1 TRANSPOSE (*tokens*[1:t])

```
output ← empty
for i ← 1 to t do
  len ← tokens[i].length
  for j ← 1 to len do
    if len ≥ i then
      APPEND(output, tokens[i][j])
    end if
  end for
end for
```

TRANSPOSE is an invertible transformation. For fixed-width tokens, inversion amounts to matrix transposition of a dense matrix. For variable-width tokens, inversion is complicated by the fact that widths must be deduced from token delimiters in the block. Still, a single scan over the block suffices for inversion. The first t bytes correspond to the first bytes of the t tokens. Subsequent bytes are appended to tokens whose delimiters have not yet been encountered.

LEMMA 3.1. TRANSPOSE *and its inverse take $O(b)$ time each where b denotes the number of bytes in the block.*

PROOF. For variable-width columns, time taken is proportional to the sum of widths of all tokens. Since TRANSPOSE and its inverse both read and write every byte exactly once regardless of whether the input is fixed-width or variable-width, both are linear time. □

3.2 PERMUTE

Algorithm 2 PERMUTE (*perm*[1:t]), *columns*[1:c][1:t]

```
output ← empty
π ← perm
for i ← 1 to c do
  APPEND-COLUMN(output, columns[i], π)
  π ← STABLE-SORT(columns[i], π)
end for
```

Algorithm 2 shows the next step in RadixZip Transform: PERMUTE. The first input parameter is *perm*[1:t], which is a permutation over the set $[1, t]$. We explain the role of this parameter in §6, where we study compression of multiple correlated blocks. For the current section, where we deal with a single block, it suffices to assume that *perm* is the identity permutation, i.e., $\forall i$: $perm[i] = i$. The second parameter for PERMUTE is *columns*[1:c][1:t], which denotes the dense matrix representation of the output produced by TRANSPOSE. For fixed-width columns, all entries of the form *columns*[i][j] are defined. However, in the case of variable-width columns, *columns*[i][j] is undefined if j-th token is less than i bytes long.

The first method used by PERMUTE is APPEND-COLUMN, which is defined below. Permutation π is applied to *column* and the result is appended to *output*. After permuting a column containing undefined characters, those characters are removed before appending the result to *output*.

```
APPEND-COLUMN(output, column[1:t], π[1:t])
for j ← 1 to t do
  if (column[π[j]] is defined) then
    append column[π[j]] to output
  end if
end for
```

The next method used is STABLE-SORT(*column*, π), which first applies permutation π to *column* to yield π(*column*) and then computes the unique permutation corresponding to stable sorting of π(*column*). This permutation is returned.

Some observations about PERMUTE:

- At the end of the i-th iteration in PERMUTE, permutation π corresponds to the sort order of columns 1 thru i taken together, with the i-th byte being the most significant. So at the end of the last iteration, the final permutation corresponds to the (stable) sort order of all tokens, with the last byte of each token being treated as the most significant byte.
- PERMUTE (*perm*, *columns*) is equivalent to first permuting all tokens by *perm*, and then invoking PERMUTE (*identity*, *columns*), where *identity* is equal to the identity permutation.
- The final permutation could be passed as a parameter to further invocations of RadixZip. This is useful when compressing multiple blocks where corresponding tokens in the blocks are correlated (see §6).

Example 3.1 shows PERMUTE run across our original fixed-width example. The input permutation is the identity permutation. For each column, a new permutation is computed by sorting that column. To break ties, we obtain the indices of all the matching characters, and extract those indices from the previous permutation in order. The column itself is permuted by the *previous* permutation before being appended to the output.

EXAMPLE 3.1. PERMUTE *run across example 2.1:*

1	a	1	**a**	s	4	**s**	p	4	**p**
2	c	3	**c**	o	2	**s**	t	1	**t**
3	a	5	**a**	s	1	**s**	p	3	**p**
4	b	4	**b**	o	3	**o**	p	5	**p**
5	a	2	**a**	s	5	**o**	p	2	**p**

The final permutation 41352 represents the sort order of the five tokens using last byte most significant order, and can be used as input to future calls to PERMUTE. The block in Example 2.1 is thus transformed by TRANSPOSE and PERMUTE into the block shown below:

a c a b a s s s o o p t p p p

Similarly, example 3.2 shows PERMUTE run across our original variable-width example. Note that although later columns are smaller due to terminated tokens, the permutations are still equal in size to the total number of tokens.

EXAMPLE 3.2. PERMUTE *run across example 2.2:*

1	p	4	**p**	o	1	**◆**	t	1	**t**	✧	1	**◆**
2	i	2	**i**	t	3	**t**	✧	3	**t**		3	**◆**
3	p	5	**p**	o	2	**t**	t	2	**◆**	✧	2	
4	a	1	**a**	✧	5	**o**		5	**◆**		5	
5	i	3	**i**	t	4	**o**	✧	4			4	

The final permutation 13254 again represents the sort order of the tokens. However, this time the length of the tokens is the most significant ordering factor; shorter tokens are last regardless of the value of the final byte. The block in Example 2.2 is transformed into

p i p a i ✧ t t o o t t ✧ ✧ ✧ ✧

Inversion of PERMUTE: PERMUTE is invertible without requiring any additional information about variable widths. Each column c can be recovered given the permutation resulting from the recovery of column $c-1$. Thus, given the same input permutation we can sequentially recover all columns. We first retrieve the first column from the first t bytes. We then identify the permutation π corresponding to its stable sort order, apply π^{-1} to the next column to retrieve that column, and so on. We illustrate the idea by walking through the output of TRANSPOSE and PERMUTE on the variable-width tokens shown in Example 2.2. The first 5 bytes `pipai` constitute the first column because there were 5 tokens in the block. We set π to the permutation that denotes the stable sort order of `pipai`. Since no terminators were encountered in the first column, we read the next 5 bytes: ✧`ttoo`. We apply π^{-1} to these bytes to obtain `oto`✧`t`. π is now set to the stable sort order of the recovered column `oto`✧`t`.

Since the second column has 1 terminator, we know that the third column has only 4 bytes. Moreover, we know that the undefined character(s) are always last, since they follow either terminators or other undefined characters which are last alphabetically. Thus, we can read 4 bytes, `tt`✧✧, and pad the column with an undefined character to obtain a 5 byte column `tt`✧✧_. We apply π^{-1} to recover `t`✧`t_`✧. Similarly, we know the last column has 2 bytes and 3 undefined characters since we have observed an additional 2 terminators in the third column. 2 bytes are read and padded to a 5 byte column ✧✧___ before recovering the last column ✧_✧__ and recomputing the final π by sorting the result.

LEMMA 3.2. PERMUTE *and its inverse are $O(b)$ time operations, where b is the number of bytes in the block.*

4. BACKEND COMPRESSION

Output from RadixZip Transform is passed through a series of three steps: Move To Front, Run-Length and Group Huffman. These three steps are also used in bzip2 after the Burrows-Wheeler Transform has been applied to a block. These components may be reproduced using the Vcodex package, written by Kiem-Phong Vo at AT&T Labs[1].

- Move To Front coding was introduced by Bentley *et al* [5], which reduces entropy by moving recent characters to the front of an encoding array. We further use an enhanced version discovered by Vo and Vo [33] that uses a heuristic to predict and move characters to the front before their turn. When Move To Front is applied to the output of Burrows-Wheeler Transform or RadixZip Transform, it tends to produce long runs of zeros interspersed by other characters.
- Run-Length coding replaces a run of zeros by encoding its length, using two special characters. For example,

[1] `http://www.research.att.com/~gfs/download/ref/vcodex/vcodex.html`

lengths 0 thru 13 could be encoded by `H, T, HH, TT, HT, TH, HHH, TTT, HHT, TTH, THH, HHT, HTH, THT`. Such an encoding is logarithmic in the size of the run. We used the Run-Length encoder deployed in Julian Seward's bzip2 implementation, the details of which are not published anywhere.

➢ Group Huffman is an improvement over the well known Huffman coder [18] which reduces string size to approach its 0-th order entropy by assigning variable length prefix-free codes to every character. Group Huffman uses a clustering method first proposed by David Wheeler, which is also used in bzip2.

LEMMA 4.1. *Compression and decompression by Move To Front, Run-Length and Group Huffman are linear time.*

5. COMPARISON WITH bzip2

In this Section, we show how RadixZip Transform improves upon the Burrows-Wheeler Transform (BWT) in terms of both performance (compression and decompression speed) and compression ratio. For illustration, we will use the following block of variable-width tokens:

o n e ✧
t w o ✧
o n e ✧
o n e ✧
t h r e e ✧
t w o ✧
o n e ✧
t h r e e ✧
o n e ✧

Figure 5.1 shows the output of both BWT and RadixZip Transform run on this input. Traditionally, BWT rearranges bytes in a block by the sort order of all its suffixes. However, we have chosen to re-arrange bytes by the sort order of prefixes read right to left. The final compression ratio achieved by bzip2 should not be altered significantly with this modification to BWT, and this simplifies our comparison to RadixZip Transform. RadixZip Transform rearranges bytes by the sort order of its prefixes read right to left, where prefixes are limited to token boundaries. RadixZip Transform could also instead be reversed to use suffixes with little impact on compressed size. However, this results in slower compression and decompression due to cache behavior.

5.1 Better Compression

As shown in Figure 5.1, both BWT (using full prefixes up to the beginning of the block) and RadixZip Transform (using prefixes limited to token boundaries) produce a large number of runs. This behavior is expected because many tokens are repeated and the total number of unique tokens is quite small. However, as we will see, RadixZip Transform improves in a number of ways.

The prefix for a character constitutes its *context*. The basic philosophy underlying both BWT and RadixZip Transform is that a character can be predicted by its context. So sorting by prefixes should create regions of low local entropy. By limiting prefixes to token boundaries, RadixZip Transform avoids *context pollution*. For example, all of the `'n'` characters are grouped by RadixZip Transform, because the prefix `'o'` is always followed by `'n'`. However, in BWT, since prefixes extend all the way to the first character of the block, context pollution occurs: the `'o'` prefixes that begin `'one'` are interspersed with the `'o'` prefixes that end `'two'`. This causes the `'n'` run to be broken up by `'✧'` characters.

Furthermore, RadixZip Transform restricts each character to remain within its column. It is a common case that various columns will have different character distributions. In these cases, RadixZip Transform ensures that these local statistics are preserved. For example, the terminating character ✧ appears in exactly two columns because there are only two distinct token lengths. RadixZip Transform results in the total area that ✧ appears in being smaller than with the BWT. This impacts the Move To Front encoder: since each character in Move To Front coding is encoded as the number of distinct characters that have appeared since it last occurred, restricting the occurrences of a character to a smaller region results in a greater frequency of low numbers. This in turn benefits the final entropy coder.

Burrows Wheeler Transform		RadixZip Transform	
o			o
one✧two✧one✧one✧three✧two✧one✧three	✧		t
one✧two✧one✧one✧three	✧		o
one	✧		o
one✧two✧one✧one✧three✧two✧one✧three✧one	✧		t
one✧two✧one✧one	✧		t
one✧two✧one✧one✧three✧two✧one	✧		o
one✧two✧one	✧		t
one✧two✧one✧one✧thre	e		o
one✧two✧one✧one✧three✧two✧one✧thre	e	o	n
one✧two✧one✧one✧th	r	o	n
one✧two✧one✧one✧three✧two✧one✧th	r	o	n
on	e	o	n
one✧two✧one✧one✧three✧two✧one✧three✧on	e	o	n
one✧two✧one✧on	e	t	w
one✧two✧one✧one✧three✧two✧on	e	t	h
one✧two✧on	e	t	w
o	n	t	h
one✧two✧one✧one✧three✧two	✧	th	r
one✧two	✧	th	r
one✧two✧one✧one✧three✧two✧one✧three✧o	n	on	e
one✧two✧one✧o	n	on	e
one✧two✧one✧one✧three✧two✧o	n	on	e
one✧two✧o	n	on	e
one✧two✧one✧one✧thr	e	on	e
one✧two✧one✧one✧three✧two✧one✧thr	e	tw	o
one✧two✧one✧one✧three✧t	w	tw	o
one✧t	w	one	✧
one✧two✧one✧one✧t	h	one	✧
one✧two✧one✧one✧three✧two✧one✧t	h	one	✧
one✧two✧one✧one✧three✧tw	o	one	✧
one✧tw	o	one	✧
one✧two✧one✧one✧three✧	t	two	✧
one✧two✧one✧one✧three✧two✧one✧three✧	o	two	✧
one✧	t	thr	e
one✧two✧one✧one✧	t	thr	e
one✧two✧one✧one✧three✧two✧one✧	t	thre	e
one✧two✧one✧	o	thre	e
one✧two✧one✧one✧three✧two✧	o	three	✧
one✧two✧	o	three	✧

Figure 1: Re-arrangement of bytes by BWT is done by sorting prefixes by reading them right to left. In RadixZip Transform, sorting is first done by prefix length and then sorting prefixes of identical length by reading them right to left.

5.2 Improved Performance

Complexity: There are a number of linear-time suffix sorting algorithms for the BWT. As these algorithms are based upon complicated data partitioning strategies, they are not always efficient in practice [26]. By contrast, RadixZip Transform mirrors a simple radix sort while rearranging input bytes. Thus it can run much faster than linear-time suffix sorting algorithms. In addition, because of its simplicity, an implementation of RadixZip Transform would be less likely to contain bugs or present security holes.

Memory: Linear time suffix sorting algorithms require additional memory for their data structures. The best known algorithm requires three integers per input byte. On the other hand, RadixZip Transform requires additional main memory that is proportional to the number of tokens. Specifically, it requires two permutations which are written back and forth as successive columns are processed. Each permutation is an array of size equal to the number of tokens, thus RadixZip Transform requires two integers per token.

Cache performance: RadixZip Transform rearranges only one column at a time, whereas suffix sort can move bytes from anywhere in the input to anywhere else. For large-sized tokens, the size of a column is much smaller than the block size. So byte movements incur fewer cache miss penalties.

6. CORRELATED STREAMS

Two token streams are said to be correlated if the tokens in one stream can be predicted by the tokens of the other. In the real world, token streams corresponding to columns of a relation in an RDBMS are often correlated. For example, country codes of incoming requests at Google servers are likely to be correlated with IP addresses of front-end servers. Or in a database of addresses, area codes of telephone numbers are likely to be correlated with zip codes.

In Vczip [33], Vo and Vo demonstrated that applying sort orders from one byte column to another can remove redundancy present in correlated byte-columns. Byte-columns in such a relationship are called predictor and predictee respectively; the predictor passes a permutation to the predictee. We use a similar idea in RadixZip: given two blocks with correlated tokens, we first apply RadixZip Transform to the first block and identify the permutation π returned by PERMUTE (see §3.2 for details). We then pass π to PERMUTE when it is invoked for the second block. This is logically equivalent to applying the permutation to all the tokens in the second block before running RadixZip over it.

Vo and Vo also showed that usage of multiple predictors for breaking ties in sort order often improves compression. It is easy to see that these ideas carry over to RadixZip over multiple token streams. Consider the example in Figure 2.

In this case, using column `IP Address` as a predictor improves the compression of column `Browser`. This is because each user prefers a particular browser on each computer. However, computers are used by more than one user, who may use different browsers. Thus, breaking ties in the sort order using the column `Client ID` can improve compression. Using `Client ID` alone does not suffice either, since users use different browsers on different computers. This is demonstrated in Figure 3.

Vczip handles the situation described above by computing a secondary predictor for each column. This secondary predictor is used to break ties. However, with RadixZip, the permutation passed from one predictor is already influenced by the permutation passed to that predictor when it was compressed. This is a powerful effect: all prior sort orders are used to break ties (this is because stable sort was performed for each block). For the table in Figure 3, one might imagine that `IP Address` was used as a predictor for `Client ID` to some benefit because each user had a preferential computer, shown below.

Original order	a	b	a	b	b	a	a		a	b	a	b	a
Sorted by IP address	a	a	b	a	a	a	a		b	b	a	b	b

As we can see by the spacing, with only two exceptions, using `IP Address` as a predictor for `Client ID` results in a complete division between the two IDs. Thus, passing a sort order from `IP Address` to the PERMUTE method for `Client ID` improves compression. Doing so would also change the permutation that is produced by RadixZip Transform to include `IP Address` as a secondary predictor, since it uses stable sorting. It could use whatever predictor was used for IP address as a tertiary predictor and so on. The sort orders produced by RadixZip Transform can thus incorporate many predictors with no additional overhead. That said, determining what predictors to use from amongst a collection of streams is left as a subject for future work.

IP Address	Client ID	Browser
1.1.1.1	a	Firefox
2.2.2.2	b	Explorer
1.1.1.1	a	Firefox
1.1.1.1	b	Maxthon
2.2.2.2	b	Explorer
2.2.2.2	a	Opera
1.1.1.1	a	Firefox
1.1.1.1	a	Firefox
2.2.2.2	b	Explorer
1.1.1.1	a	Firefox
2.2.2.2	b	Explorer
1.1.1.1	a	Firefox

Figure 2: Hypothetical log with two users on two computers

By IP Address	By Client ID	By both
Firefox	Firefox	Firefox
Firefox	Firefox	Firefox
Maxthon	Opera	Firefox
Firefox	Firefox	Firefox
Firefox	Firefox	Firefox
Firefox	Firefox	Firefox
Firefox	Firefox	Opera
Explorer	Explorer	Maxthon
Explorer	Maxthon	Explorer
Opera	Explorer	Explorer
Explorer	Explorer	Explorer
Explorer	Explorer	Explorer

Figure 3: Sorting of column `Browser` by various predictors. Using *both* `IP Address` and `Client ID` gives the best compression ratio.

7. EXPERIMENTAL RESULTS

We experimented with three datasets:

- **Census data**: The current population survey in USA is carried out by the Bureau of Labor Statistics and the Census Bureau. It is available at `http://www.bls.census.gov/cps_ftp.html`. The survey report consists of a stream of fixed-width records, each of which is sub-divided into fixed-width fields described in `ftp://www.bls.census.gov/pub/cps/basic/200508/augnov05dd.txt` We divided the overall stream into individual streams, one per field. Each stream is tokenized into fixed-width tokens.

- **Forest cover data**: Forest cover data is collected by the US Forest Service Region 2 Resource Information System. The data describes measurements of landscape attributes in 30×30 meter cells. It is available as part of the UC Irvine Knowledge Discovery in Databases corpuses of large data at `http://kdd.ics.uci.edu/databases/covertype/covtype.data.gz`. The data consists of a stream of records, one record per cell. Each record is a comma-separated list of integers in text format. Each integer represents some attribute of the cell. We divided the dataset into multiple streams, one per attribute.

- **Ads clicks logs**: Google web-servers create a record of every click-through of Google ads. Such records contain a variety of attributes like timestamps, IP addresses and details of ad impressions. Many of these attributes are categorical in nature. We divided ad records into individual streams, one stream per attribute. These streams are grouped into four types: integers, strings, floats and doubles. Streams of integers and strings constitute variable-width streams whereas streams of floats and doubles constitute fixed-width streams.

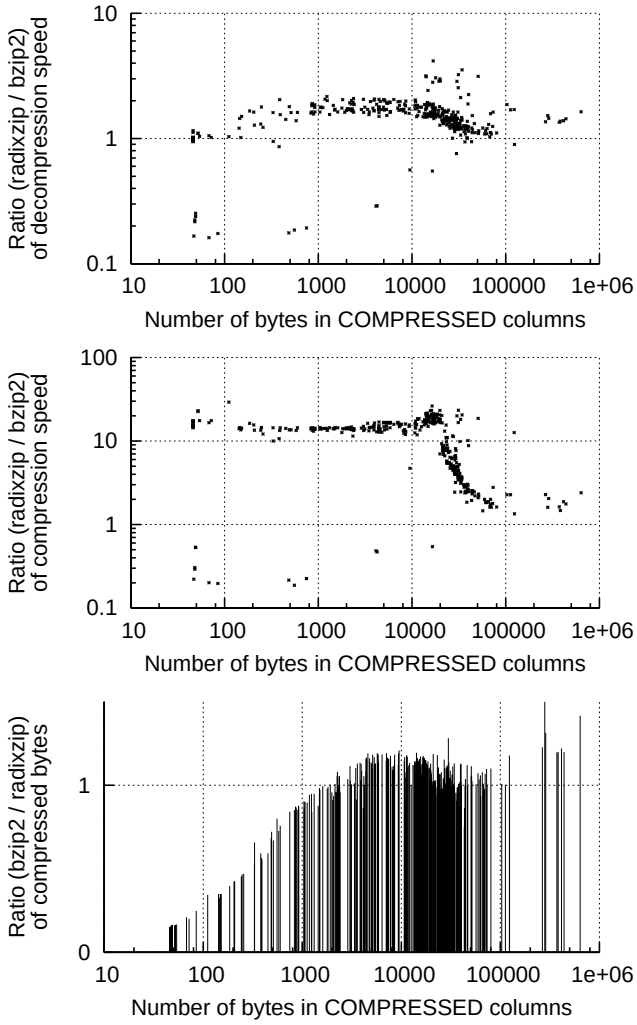

Figure 4: RadixZip to bzip2 ratios v compressed stream size across all streams in census data.

Figure 4 compares RadixZip and bzip2 for various streams in census data, there being one stream per attribute. The X axes denote *compressed* stream sizes. The larger the value along the X axes, the more that stream contributes to overall compression ratio. For larger streams, RadixZip generally outperforms bzip2. Overall, RadixZip achieved a compression ratio of 15:1 which represents a 10-20% improvement.

For some streams, RadixZip was worse than bzip2. This is probably because data in these streams was not well-structured, making the columns produced by byte transposition meaningless.

RadixZip performs poorly against bzip2 on streams whose compressed size is 1000 or less. This suggests that RadixZip probably needs sufficient average width of tokens to become effective. This is supported by plots in Figure 5, which show RadixZip performs better when width of tokens is larger.

Figure 4 also shows the compression and decompression speed of RadixZip as a ratio against bzip2. In decompression we see a consistent gain of about 10% over bzip2. There are a few outliers where RadixZip loses by significant amounts to bzip2, which appear to be mostly on very small compressed streams. This is likely due to the data size being insufficient for the difference between RadixZip Transform and the Burrows-Wheeler Transform to be significant. In this case, other factors such as our use of an additional TRANSPOSE step and a predictive step in Move To Front dominate.

In compression, we also see consistent gains, but they also get markedly better for streams which compress to less than about 20k, barring the same outliers as with decompression. As we can see in Figure 6, RadixZip tends to compress faster on streams that compress better. This is likely due to improved cache performance when recalculating permutations between the byte columns. Better compression tends to coincide with more repetitive tokens in the streams. This creates smaller alphabets in the byte columns, which leads to reduced modification of the carried permutations. Conversely, with bzip2, repetitive substrings adversely impact the underlying suffix sort algorithm and force it to use its slower fallback algorithm. Using more modern suffix sorting algorithms in bzip2 would reduce this difference, however RadixZip remains a fundamentally faster approach.

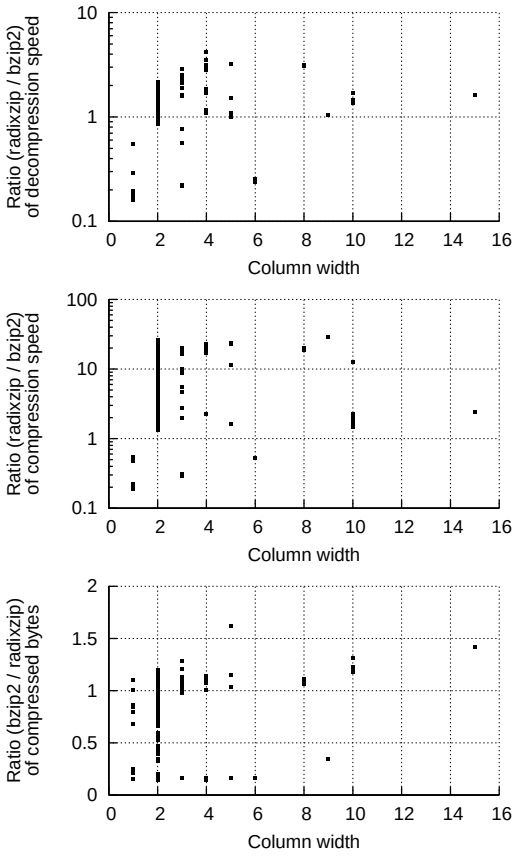

Figure 5: RadixZip to bzip2 ratios v stream width across all streams in census data.

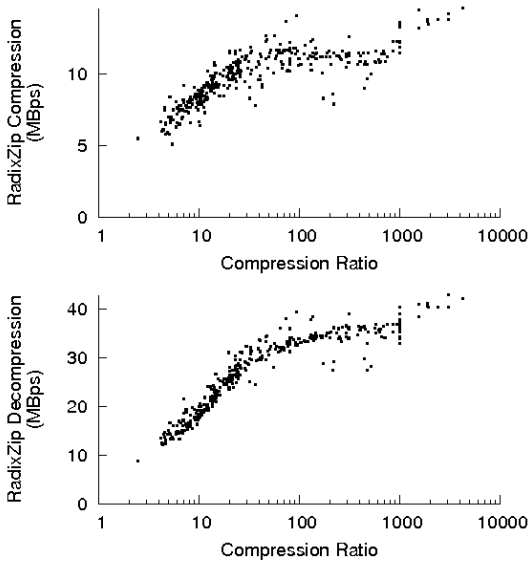

Figure 6: (De)compression speed v compression ratio for all streams in census data.

RadixZip	bzip2
613195	324651
606062	295820
570618	500948
509128	409766
507856	461411
478430	382884
452378	157132
444642	393210
400000	349486
329478	274870

Figure 7: Compressed stream sizes in forest cover data.

Figure 7 shows the compression results of RadixZip as a ratio against bzip2 on the largest streams in the forest cover data. In this case, we see that RadixZip consistently loses. The forest cover data contains columns with measurements of various terrain attributes, which are fairly random within a range. As such most digits after the first are randomly distributed. After TRANSPOSE, the resulting byte-columns after the first have little meaning, and so RadixZip performs poorly. For some of the columns, the losses are significant, where's RadixZip's byte column based approach has misaligned the digits of measurements which spanned different powers of 10 (i.e. 99 and 100). It is thus important to note that not all token streams are amenable to RadixZip.

Figure 8 shows the compression results of RadixZip as a ratio against bzip2 on the ad clicks data. For variable-length integer and float columns, we see results similar to the census data: a net improvement of about 10%. For string data, we see a net loss of about 1%. In this case, we see benefits for some string columns which contain categorical data such as region codes or browser languages, but losses for many columns which contain unorganized text such as long URLs. This was partially offset by the best gaining string column for RadixZip which contained a stream of identifying strings of Google's advertising partners. This data is both large and categorical. Again, the best compression would result from selective use of the two compressors.

Figure 9 shows a small sampling of hand-picked dependency relationships from the census data. We compare compression using bzip2 alone, RadixZip alone, and using RadixZip when passed a permutation from a predictor column from the same dataset. As we can see, potential compression benefits are large, reaching as high as 99.8%. When correlation is accurate enough to approach functional dependency, using a predictor can compress a stream to nearly zero, representing the fact that it contains no new information over its predictor. This shows that there are situations in real data where passing permutations in RadixZip is a valuable tool. Note also that there are cases where using a predictor leads RadixZip to win over bzip2 where it would not have before. These correlations are not necessarily the best possible that could have been chosen to minimize the size of the set as a whole; a good method for determining the best set of dependency relationships is still necessary.

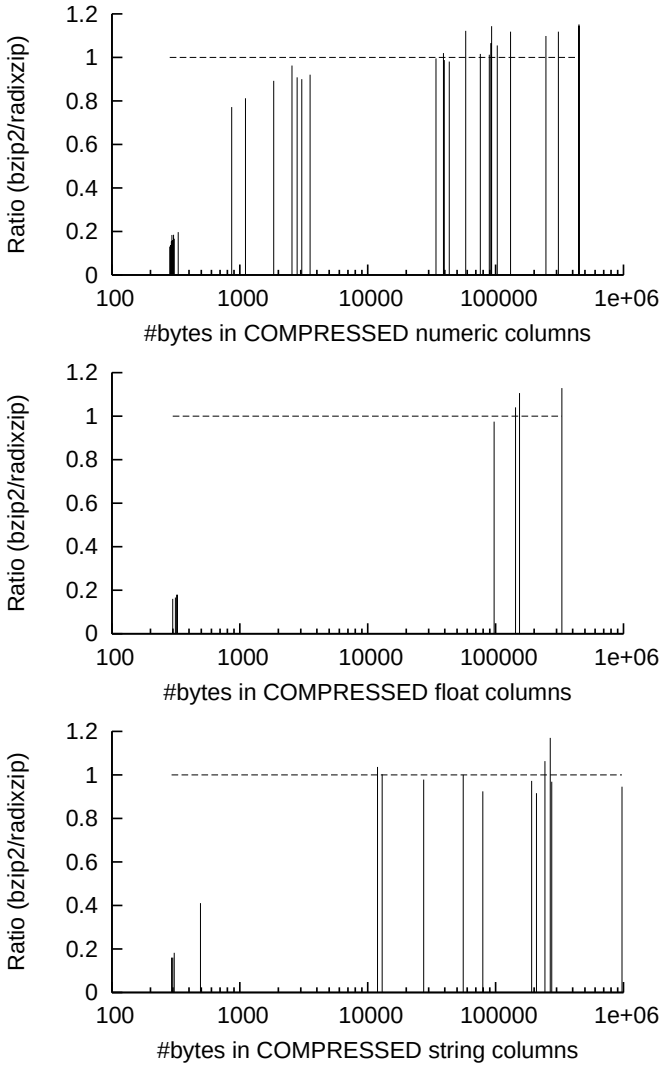

Figure 8: RadixZip to bzip2 ratios v compressed stream size across streams in AdClicks data

bzip2	RadixZip	predictor	% reduction
16003	16357	32	99.8 %
13741	13348	28	99.8 %
21559	20555	63	99.7 %
16156	16310	45	99.7 %
14518	19751	86	99.6 %
12076	12398	52	99.6 %
10078	10664	44	99.6 %
26213	25187	120	99.5 %
24948	23392	124	99.5 %
12084	12416	62	99.5 %
29503	27586	153	99.4 %
11650	12091	73	99.4 %
11992	12346	99	99.2 %
11681	12109	110	99.1 %
10681	10185	95	99.1 %

Figure 9: Compression of census data streams with prediction

8. RadixZip IN PRACTICE

There are many real world datasets which consist of record sequences that can be decomposed into fields. Any such sequence of records can be split into token streams, one stream per field, which may be amenable to RadixZip. For example:

- *Relational tables*: Records in a relational table consist of tuples. Each tuple has as many members as the number of columns in the table schema. A sequence of tuples can be broken into token streams in a natural way with one stream per column.
- *XML data*: As shown in the XMill [21], an effective compression technique is to store the XML tree separately, and to group the values in leaf nodes into containers such that there is one container per unique XML path expression. Each container amounts to a token stream.
- *Log records at Google*: The storage format at Google for storing log records is simply a byte string that encodes a list of (key, value) pairs [24]. Four kinds of values are supported: strings, doubles, floats and integers. The *key* associated with a value denotes its type and its name. The record format at Google provides two interesting features: (a) sets are supported by allowing multiple (key, value) pairs to have the same key, and (b) embedding is supported by allowing strings to represent byte-strings that encode other records recursively. Thus a sequence of records can be compressed by separating structure from content, along the lines of XMill: the parse tree of a record is stored separately and values are valued if they share the same key.
- *Serialized objects*: For transmission of objects across the wire, and for persistent storage, *pickling* [6] or *serialization* of objects in languages like Java [14] and CORBA [32] is employed. For compression, a sequence of objects can be transformed by separating structure (the definition of objects) from content (the values associated with members of object instances). Values can be grouped to form token streams, some of which would be compressed best by RadixZip.

Warehousing and Archival: Large data warehouses store hundreds of terabytes of raw data. Storage requirements for log records collected by companies like Google are two orders of magnitude larger. Query workload over such data is typified by long-running queries for analysis, reporting and data mining. Google has developed the Sawmill system [24] for such queries, which carry out sequential scans over raw data.

The overall cost of infrastructure for logs storage and analysis is proportional to the total number of bytes occupied by log files. However, not all records are popular: recently generated records are read far more often than old ones. Since the majority of bytes correspond to rather old records, it makes sense to maximize compression ratio (by a suitable combination of gzip, bzip2 and RadixZip) at the cost of slower decompression times.

Categorical datasets: RadixZip works especially well when tokens are drawn from a categorical domain. Examples include strings of browser names, IP addresses from a

specific country, language codes, country codes, etc. Non-categorical domains include search queries strings posted at `http://www.google.com`, timestamps.

Impact of block size: Although RadixZip is linear time, careful coding is necessary because cache-performance can be heavily impacted by block size. Computing the permutation for each column in PERMUTE (see §3.2 for details) is the bottleneck operation since it involves writes to random indices in a large integer array. So choosing the block size such that a permutation can fit entirely in cache should improve throughput. Note that the size of the permutation is proportional to the number of tokens, which is much less than the number of bytes in a block. Thus the block size can sometimes be much larger than the cache size.

Training: RadixZip outperforms bzip2 often, but not in all cases. For maximum compression of a set of records that has been decomposed into multiple token streams, we should identify the best possible algorithm per token stream. When the number of records is very large, say of the order of billions, it suffices to *train* over a small sample (of the order of tens of thousands of records). The goal of training is to identify the best algorithm per token stream by comparing various algorithms over the sample. Generally speaking, the nature of such token streams does not change significantly over long periods of time, nor do the correlations between them [7, 33]. So training does not have to be repeated often. Finally, invocations of RadixZip over correlated blocks can be improved by passing permutations from one block to another. Identifying correlated blocks automatically is an interesting research problem.

9. TABLE COMPRESSION: A SURVEY

We will now survey existing compression methods that are being used to encode database or tabular data. These range from simple techniques for use on frequently accessed data, to heavier compression algorithms for long-term storage.

9.1 Lightweight Compression Techniques

9.1.1 Column-oriented Storage

Traditionally, relational databases stored tuples in a page in row-major format. An alternative called "vertical partitioning" was proposed by Copeland and Khoshafian [11]. A relational table was instead divided into as many tables as the number of columns. That way, one page would contain values in a single column. This scheme slows down updates since multiple pages have to be modified. However, selection conditions over columns run much faster. A compromise between traditional format and vertical partitioning was suggested by Ailamaki *et al* [2]; paging would still be done on whole rows, but tuples within a page were stored in column-major format. Subsequently, research into column-oriented data storage has continued, as exemplified by experimental systems like C-Store [30] and commercial systems like Sybase IQ [22]. Recent work by Harizopoulos *et al* [16] identifies workloads under which column-oriented systems outperform row-oriented systems.

9.1.2 Compression by Simple Encoding Techniques

In both row-oriented and column-oriented storage systems, individual members of a tuple can be compressed by a variety of lightweight encoding techniques:

- **Null value suppression**: It is common for large portions of database or tabular data to have a lot of NULL values; so compression can be improved by suppressing them [28, 36]. INGRES [31], one of the earliest relational databases, employed NULL value suppression.

- **Dictionary/Domain encoding**: A global dictionary that maps values to fixed-length codes is created. Individual values in a column are replaced by their codes. For example, if there are only eight possible values for a column of type CHAR(12), then only 3 bits per value are necessary. Domain encoding has been explored by various groups [10, 13, 15, 28, 30, 39]. Recently, Raman and Swart [27] have developed this idea further by using Huffman encoding [18] and exploiting correlations between columns by pairing them into single tuples.

- **Delta compression**: In a relational database, tables are commonly stored by sorting the tuples with some attribute treated as the sort key. Occasionally, multiple columns are concatenated to derive the sort key. Sorted keys can be efficiently compressed by storing only the *differences* between successive keys.

- **Bit-mapped indices**: These are commonly used in data warehouses. The basic idea originated in work by O'Neil and Graefe [23]. Compressed bitmaps were investigated by Amer-Yahia and Johnson [3]. For a survey of other space-saving tricks and techniques used in data warehouses, see the survey by Sarawagi [29].

Lightweight compression is now supported by commercial databases, such as IMS [12], DB2 [19] and Oracle [25]. Its impact on query performance has been studied by various groups: Graefe and Shapiro [15], Amer-Yahia and Johnson [3], Goldstein, Ramakrishnan and Shaft [13], Johnson [20]. Recently, Holloway *et al* [17] have studied the performance of various lightweight techniques over modern processors. Abadi, Madden and Ferriera [1] show how compression can be integrated into column-oriented systems.

9.2 General-Purpose Compression

Two general purpose compressors used widely are gzip, based on Liv-Zempel encoding [37, 38], and bzip2, based on the Burrows-Wheeler Transform [9] (BWT). gzip achieves compression by identifying repeated occurrences of long sub-strings and replacing them by an encoding of the length of the sub-string and the offset at which the previous occurrence of this sub-string can be found. Offsets are limited to a maximum value called the "window size". The BWT rearranges characters in a block by the sort order of the suffixes of these characters. If suffixes provide a good context for characters, this creates regions of locally low entropy, which can be exploited by various back-end compressors. This requires sophisticated data structures for suffix arrays.

9.3 Specialized Techniques for Tabular Data

Two techniques have recently been discovered at AT&T Labs for compression of large tables with fixed-width columns. PZip [7, 8] is a stream compressor while Vczip [33, 34] is a block compressor. Both algorithms require *training*, i.e., some data structure is computed over a small subset of the records, which is then used during compression.

9.3.1 PZip

PZip, by Buchsbaum *et al* at AT&T Labs [7, 8], is the first compressor to address tabular data with fixed width records. Although records may contain fields, they are ignored and only byte-columns are handled. The ordering of these columns is permuted and they are then divided into disjoint subsets.

Each subset of columns is traversed in row major format and fed to a backend compressor like gzip. Computing the optimal ordering and partitioning of columns is an NP-hard problem and expensive even to approximate. As such, a heuristic is used, and then is only trained over a small subset of records with the results being used for a much larger set. A complete graph is created with one node per column. The edges between two nodes are labeled by an estimates of the *co-compressibility* of the columns corresponding to the two nodes. A Traveling Salesman Problem approximation is used to compute a near-optimal ordering of the columns. Subsets are identified by dynamic programming. Note that an optimal ordering of pair-wise co-compressibilities does not necessarily result in an optimal compression across all columns. Moreover, an optimal partitioning of an optimal ordering is not necessarily optimal over all possible orderings. However, the algorithms employed by PZip work very well; the authors claim 55%-75% improvement over gzip.

An interesting experiment would be to study the efficacy of a PZip-like approach as a pre-processor for compression by RadixZip instead of gzip, as a group of columns produced by PZip can be treated as a stream of tokens of fixed width.

9.3.2 Vczip

Vczip is compressor based on the Vcodex framework developed by Kiem-Phong Vo at AT&T Labs [35]. Vczip includes a table compression technique based on column dependency [33, 34]. Byte-column dependencies are relationships in which the contents of one byte-column can be predicted given the contents of some others. For example, in a table of phone records and addresses, one might imagine the contents of a byte-column from a person's area code might be better guessed given the contents of byte-columns from their zip code. Again, the problem of computing the optimum predictors for each column is NP-hard and an efficient and effective heuristic was developed to find effective sets of predictors with maximum size two. Like with PZip, this heuristic is applied on a small header of records and the results are used for a larger set. Each predictor set is used as a context to sort its predicted columns. Like RadixZip, this is similar in principle to the Burrows-Wheeler Transform. Vczip improves over PZip by an average of 30% and is currently the best known compressor for table data.

As mentioned earlier, a stream of records in a table is a subclass of a token stream, that additionally contains several internal fields. Vczip and RadixZip share the idea of applying a permutation from one context to a target column. Whereas Vczip passes these permutations across a distance and uses a number of predictors bounded to two, RadixZip carries them sequentially and uses a potentially great number of predictors. These distinctions are due to the fact that Vczip aims to work with 'tokens' that contain some unknown internal structure, whereas RadixZip assumes its tokens are minimal units. Still, RadixZip's method for carrying sorted permutations across multiple columns might be an improvement to Vczip worth investigating.

9.4 Lossy Compression

SPARTAN is a lossy compressor for large tables by Babu *et al* [4]. SPARTAN aims to exploit correlations between attributes as detected via a Bayesian Network. These correlations are then utilized by constructing `CaRT` models. A `CaRT` model computes a value in the predictee based on the corresponding values in the predictors. They are chosen to be accurate within a provided error bound, and succinctly encodable. These models replace the actual values, which may be discarded.

Although SPARTAN also aims to exploit correlations, it cannot be directly applied to lossless compression since the predictees are not encoded. In order to to achieve lossless compression that exploits correlation, it would need to be extended with some method of encoding the predictee that improves coding based on the accuracy of the estimate provided by `CaRT` model.

An interesting experiment would be to see if the correlations computed by SPARTAN would serve as useful predictors for passing permutations with RadixZip.

10. CONCLUSIONS

RadixZip is a new algorithm for compressing streams of tokens. It improves upon traditional compressors like bzip2 that are oblivious to token boundaries. RadixZip can also efficiently exploit correlations between streams of tokens. The applicability of RadixZip extends to structured datasets in general. These include tables in relational databases, log records created by web servers, XML data and serialized objects created in languages like Java and CORBA.

11. REFERENCES

[1] Daniel J Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column oriented database systems. In *Proc. SIGMOD 2006*, pages 671–682, 2006.

[2] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving relations for cache performance. In *The VLDB Journal*, pages 169–180, 2001.

[3] S Amer-Yahia and T Johnson. Optimizing queries on compressed bitmaps. In *Proc. 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 329–338, 2000.

[4] Shivnath Babu, Minos Garofalakis, and Rajeev Rastogi. SPARTAN: A model-based semantic compression system for massive data tables. In *Proc. ACM SIGMOD 2001*, pages 283–294, 2001.

[5] Jon Louis Bentley, Daniel D Sleator, Robert Endre Tarjan, and Victor K Wei. A locally adaptive data compression scheme. In *Communications of the ACM*, pages 320–330, 1986.

[6] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proc. 14th ACM Symposium on Operating Systems Principles (SOSP 1994)*, pages 217–230, 1994.

[7] Adam L Buchsbaum, Donald F Caldwell, Kenneth Ward Church, Glenn S Fowler, and S Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete*

Algorithms (SODA 2000), pages 175–184, January 2000.

[8] Adam L Buchsbaum, Glenn S Fowler, and Raffaele Giancarlo. Improving table compression with combinatorial optimization. *Journal of the ACM*, 50(6):825–851, 2003.

[9] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, May 1994.

[10] Z Chen, J Gehrke, and F Korn. Query optimization in compressed database systems. In *Proc. ACM SIGMOD 2001*, pages 271–282, 2001.

[11] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *Proc. ACM SIGMOD 1985*, pages 268–279, 1985.

[12] G V Cormack. Data compression on a database system. *Communicatons of the ACM*, 28(12):1336–1342, 1985.

[13] J Goldstein, R Ramakrishnan, and U Shaft. Compressing relations and indexes. In *Proc. 14th International Conference on Data Engineering (ICDE 1998)*, pages 370–379, 1998.

[14] James Gosling, Bill Joy, and Guy L Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., first edition, 1996.

[15] G Graefe and L Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symposium on Applied Computing*, pages 22–27, April 1991.

[16] Stavros Harizopoulos, Velen Liang, Daniel J Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *VLDB 2006*, pages 487–498, 2006.

[17] Allison L Holloway, G Swart, V Raman, and David J DeWitt. How to barter bits for chronons: Compression and bandwidth trade offs for database scans. In *Proc. ACM SIGMOD 2007*, June 2007.

[18] David A Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Institute of Radio Engineers*, pages 1098–1101, 1952.

[19] B R Iyer and D Wilhite. Data compression support in databases. In *Proc. 20th International Conference on Very Large Data Bases (VLDB 1994)*, pages 695–704, 1994.

[20] T Johnson. Performance measurements of compressed bitmap indices. In *Proc. 25th International Conference on Very Large Data Bases (VLDB 1999)*, pages 278–289, 1999.

[21] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. In *Proc. ACM SIGMOD 2000*, pages 153–164, 2000.

[22] R MacNiol and B French. Sybase IQ multiplex – designed for analytics. In *Proc. 30th International Conference on Very Large Data Bases (VLDB 2004)*, pages 1227–1230, 2004.

[23] P O'Neil and G Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–1, 1995.

[24] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming (Special Issue on Dynamic Grids and Worldwide Computing)*, 13(4):277–298, 2005.

[25] Meikel Pöss and Dmitry Potapov. Data compression in Oracle. In *VLDB 2003*, pages 937–947, 2003.

[26] Simon J Puglisi, William F Smyth, and Andrew Turpin. The performance of linear time suffix sorting algorithms. In *Proc. Data Compression Conference (DCC 2005)*, pages 358–367, 2005.

[27] V Raman and G Swart. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proc. 32nd International Conference on Very Large Data Bases (VLDB 2006)*, pages 858–869, 2006.

[28] M A Roth and S J V Horn. Database compression. *SIGMOD Record*, 22(3):31–39, 1993.

[29] Sunita Sarawagi. Indexing OLAP data. *IEEE Data Engineering Bulleting*, 20(1):36–43, 1997.

[30] M Stonebraker, D J Abadi, A Batkin, X Checn, M Cherniak, M Ferreira, E Lau, A Lin, S Madden, E J O'Neil, P E O'Neil, A Rasin, N Tran, and S B Zdonik. C-Store: A column-oriented DBMS. In *Proc. 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 553–564, 2005.

[31] M Stonebraker, E Wong, P Kreps, and G Held. The implementation and design of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

[32] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2):46–55, 1997.

[33] Binh Dao Vo and Kiem-Phong Vo. Using column dependency to compress tables. In *Proc. Data Compression Conference (DCC 2004)*, pages 92–101, 2004.

[34] Binh Dao Vo and Kiem-Phong Vo. Compressing table data with column dependency. In *Theoretical Computer Science*, 2007.

[35] Kiem-Phong Vo. Vcodex: A data compression platform. *International Conference on Software and Data Technologies (ICSOFT2007)*, 2007.

[36] T Westmann, D Kossmann, S Helmer, and G Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[37] J Ziv and A Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[38] J Ziv and A Lempel. Compression of individual sequences via variable-rate encoding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[39] M Zukowski, S Heman, N Nes, and P Boncz. Super-scalar RAM-CPU cache compression. In *Proc. 22nd International Conference on Data Engineering (ICDE 2006)*, page 59, 2006.