VCODEX: A DATA COMPRESSION PLATFORM

Kiem-Phong Vo
AT&T Labs, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA

Keywords: Data compression, transformation.

Abstract: Vcodex is a platform to compress and transform data. A standard interface, *data transform*, is defined to represent any algorithm or technique to encode data. Although primarily geared toward data compression, a data transform can perform any type of processing including encryption, portability encoding and others. Vcodex provides a core set of data transforms implementing a wide variety of compression algorithms ranging from general purpose ones such as Huffman or Lempel-Ziv to structure-driven ones such as reordering fields and columns in relational data tables. Such transforms can be reused and composed together to build more complex compressors. An overview of the software and data architecture of Vcodex will be presented. Examples and experimental results show how compression performance beyond traditional approaches can be achieved by customizing transform compositions based on data semantics.

1 INTRODUCTION

Modern business systems routinely manage huge amounts of data. For example, the daily volumes of billing and network management data generated by an international communication company involve tens to hundreds of gigabytes (Fowler et al., 2004). Further, certain of such data is required by laws to be kept online for several years. Thus, data compression is a critical component in these systems.

Much of compression research has traditionally been focused on developing general purpose techniques which treat data as unstructured streams of bytes. Algorithms based primarily on either pattern matching, statistical analysis or some combination thereof (Huffman, 1952; Jones, 1991; Witten, 1987; Ziv and Lempel, 1977; Ziv and Lempel, 1978) detect and remove different forms of information redundancy in data. Well-known compression tools such as the ubiquitous Unix Gzip or Windows Winzip are based on such general purpose techniques, for example, the Lempel-Ziv compression method (Ziv and Lempel, 1977) and Huffman coding (Huffman, 1952). A recently popular compressor, Bzip2 (Seward, 1994), first reorders data by the famous BWT or Burrows-Wheeler Transform (Burrows and Wheeler, 1994) to induce better compressibility before passing it to other compressors.

Structures in data often provide a good source of information redundancy that can be exploited to improve compression performance. For example, columns or fields in relational data tend to be sparse and may share non-trivial relationships. Buchsbaum et al (Buchsbaum et al., 2003) developed the Pzip technique that assumes some conventional compressor such as Gzip and groups table columns to improve their compressibility by such a compressor. Vo and Vo (Vo and Vo, 2004; Vo and Vo, 2006) took a different approach to the same problem and showed how to use dependency relations among table columns to rearrange data for better compressibility. For a different class of data, namely XML, the Xmill compressor (Liefke and Suciu, 2000) works in a similar spirit to Pzip by grouping parts of a document with the same tags to be compressed by some conventional compressor.

In practice, data often consist of ad-hoc mixtures of structures beyond just simple tables or single XML documents. For example, a log file generated by some network management system might contain many records pertaining to different network events. Due to their common formats, the records belonging to a particular event might form a table. As such, the compressibility of such a log file could be enhanced by grouping records by event types first before compressing each group by one of the above

Vo K. (2007).
VCODEX: A DATA COMPRESSION PLATFORM.
In *Proceedings of the Second International Conference on Software and Data Technologies - SE*, pages 81-89
DOI: 10.5220/0001344600810089

table compressors. Unfortunately, from the point of view of building compression tools, it is hard, if not impossible, to automatically detect and take advantage of such ad hoc structures in data. Nonetheless, this points out that, in each particular circumstance, optimum compression performance may require the ability to form arbitrary compositions of diverse data transformations, both generic and data-specific. Indeed, this ability is implicitly needed even in implementing general purpose compressors such as Gzip and Bzip2 that combine distinct techniques dealing with different aspects of information redundancy. Thus, a major challenge to compression research as well as to practical use of compression on large scale data is to provide a way to construct, reuse and integrate different data transformation and compression techniques to suit particular data semantics. This is the problem that the Vcodex platform addresses.

Central to Vcodex is the notion of *data transforms* or software components to alter data in some invertible ways. As it is beyond the scope of this paper, we only mention that by taking this general approach to data transformation Vcodex can also accommodate techniques beyond compression such as encryption, translation between character sets or just simple coding of binary data for portability. For maximum usability, two main issues must be addressed:

- *Defining a standard software interface for data transforms*: A standardized software interface helps transform users as it eases implementation of applications and ensures that application code remains stable when algorithms change. Transform developers also benefit as a standard interface simplifies and encourages reuse of existing techniques in implementing new ones.
- *Defining a self-describing and portable standard data format*: As new data transforms may be continually developed by independent parties, it is important to have a common data format that can accommodate arbitrary composition of transforms. Such a format should allow receivers of data to decode them without having to know how they were encoded. In addition, encoded data should be independent of OS and hardware platforms so that they can be easily transported and shared.

The rest of the paper gives an overview of the Vcodex platform and how it addresses the above software and data issues. Experiments based on the well-known Canterbury Corpus (Bell and Powell, 2001) for testing data compressors shows that Vcodex could far outperform conventional compressors such as Gzip or Bzip2.

2 SOFTWARE ARCHITECTURE

Vcodex is written in the C language in a style compatible with C++. The platform is divided into two main layers. The base layer is a software library defining and providing a standard software interface for data transforms. This layer assumes that data are processed in segments small enough to fit entirely in memory. The library can be used directly by applications written in C or C++ in the same way that other C and C++ libraries are used. A command tool Vczip is written on top of the library layer to enable file compression without low-level programming. Large files are handled by being broken into chunks with suitable sizes for in-memory processing.

2.1 Library Design

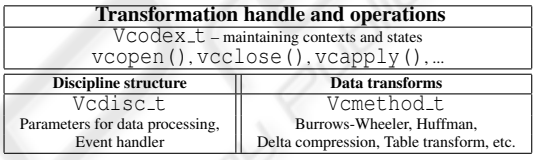

Figure 1: A discipline and method library architecture for Vcodex.

Figure 1 summarizes the design of the base Vcodex library which was built in the style of the Discipline and Method library architecture (Vo, 2000). The top part shows that a *transformation handle* of type `Vcodex_t` provides a holding place for contexts used in transforming different data types as well as states retained between data transformation calls. A variety of functions can be performed on such handles. The major ones are `vcopen()` and `vcclose()` for handle opening and closing and `vcapply()` for data encoding or decoding. A transformation handle is parameterized by an optional discipline structure of type `Vcdisc_t` and a required data transform of type `Vcmethod_t`. Each discipline structure is supplied by the application and provides additional information about the data to be processed. On the other hand, a data transform is selected from a predefined set and specifies the transformation technique. Section 2.2 describes a few common data transforms such as Burrows-Wheeler, Huffman, etc. Complex data transformations can be composed from simpler ones by passing existing handles into newly opened handles for additional processing.

Figure 2 shows an example to construct and use a transformation handle for delta compression by composing two data transforms: `Vcdelta` and `Vchuffman` (Section 2.2). Here, delta compression (Hunt et al., 1998) is a general technique to compress a *target*

```
1. typedef struct _vcdisc_s
2. {     void*     data;
3.       ssize_t   size;
4.       Vcevent_f eventf;
5. } Vcdisc_t;
6. Vcdisc_t disc = { "Source data to compare against", 30, 0 };

7. Vcodex_t* huff = vcopen(0, Vchuffman, 0, 0, VC_ENCODE);
8. Vcodex_t* diff = vcopen(&disc, Vcdelta, 0, huff, VC_ENCODE);

9. ssize_t cmpsz = vcapply(diff, "Target data to compress", 23, &cmpdt);
```

Figure 2: An example of delta compression.

data given a related *source data*. It is often used to construct software patches or to optimize storage in revision control systems (Tichy, 1985). The transform `Vcdelta` implements a generalized Lempel-Ziv parsing technique for both delta and normal compression whose data format was the subject of the IETF (Internet Engineering Task Force) VCDIFF Proposed Standard RFC3284 (Korn et al., 2002; Korn and Vo, 2002).

- Lines 1–6 show the type of a discipline structure and `disc`, an instance of it. The fields `data` and `size` of `disc` provide any source data available to compare against in delta compression. Even if no source data is given, the target data would still be compressed in the usual Lempel-Ziv fashion, i.e., by factoring out repeating patterns. The field `eventf` of `disc` is optional and, if used, specifies a function to process events such as handle opening and closing.
- Lines 7 and 8 show handle creation and composition. The Huffman coding handle `huff` is created first. The first and second arguments to `vcopen()` specify the optional discipline structure, e.g., `disc` on line 8, and the selected data transform, e.g., `Vchuffman` on line 7. Variants or modes of a transform can be given in the third argument (see the examples in Figure 5). The fourth argument is used to compose a transform with another one. For example, on line 8, the handle `huff` is used to further compress the output of `diff`. The last argument of `vcopen()` must be one of `VC_ENCODE` or `VC_DECODE` to tell if the handle is opened for encoding or decoding.
- Line 9 gives an example of calling `vcapply()` to compress some given data. The compressed result is returned in `cmpdt` while its size is returned by the call. A transform such as `Vcdelta` creates multiple output data segments coding different types of data (Section 3). Each such segment would be separately passed to the follow-on transformation handle, in this case, `huff`. As such, `huff` can generate the appropriate Huffman codes to match with each segment and improve compression performance. Note also that the source data in the discipline structure `disc` could be changed before each `vcapply()` call to match with the data to be compressed.

The above example shows the encoding side. The decoding side is exactly the same except that `VC_DECODE` replaces `VC_ENCODE` and the data passed to `vcapply()` would be some previously compressed data. An advantage in being able to write application code based on a standard interface as provided is that algorithmic enhancement of the data transform in use does not require changes at the application level. In addition, experimentation with different data transforms only need minimal code editing.

2.2 Data Transforms

```
1. typedef struct _vcmethod_s
2. {     ssize_t (*encodef)(Vcodex_t*, void*, ssize_t, void**);
3.       ssize_t (*decodef)(Vcodex_t*, void*, ssize_t, void**);
4.       int     (*eventf)(Vcodex_t*, int, void*);
5.       char*   name;
6.       ...
7. } Vcmethod_t;
```

Figure 3: The type of a data transform.

Figure 3 shows `Vcmethod_t`, the type of a data transform. Lines 2 and 3 show that each transform provides two functions for encoding and decoding. An optional event handling function `eventf`, if provided, is used to process certain events. For example, some transforms maintain states throughout the lifetime of a handle. The structures to keep such states would be created or deleted at handle opening and closing via such event handling functions. Each transform also has a `name` that uniquely identifies it among the set of all available transforms. This name is encoded in the output data (Section 3) to make the encoded data self-describing.

Vcodex provides a large collection of transforms for building efficient compressors of general and structured data including a number that are application-specific. Below are brief overviews of a few important data transforms:

- `Vcdelta`: This implements a delta compressor based on a generalized Lempel-Ziv parsing method. It outputs data in the Vcdiff encoding format as described in the IETF Proposed Standard RFC3284 (Korn et al., 2002).
- `Vcsieve`: This transform used approximate matching to perform delta compression as well as compression of genetic sequences, a class of data notoriously difficult to compress (Manzini and Rastero, 2004).
- `Vcbwt`: This implements the Burrows-Wheeler transform.

- `Vctranspose`: This treats a dataset as a table, i.e., a two-dimension array of bytes, and transposes it.
- `Vctable`: This uses column dependency (Vo and Vo, 2006) to transform table data and enhance their compressibility.
- `Vcmtf`: This provides the move-to-front data transform (Bentley et al., 1986) and a predictive variant of it. The predictive variant uses a heuristic algorithm that keeps track of how often a character follows another and, when favorable, moves it to the front as soon as its predecessor is encoded.
- `Vcrle`: This provides the run-length encoder and a variant that only encodes runs of zeros using a special binary coding method (Deorowicz, 2000).
- `Vchuffman`: This implements static Huffman coding.
- `Vchuffgroup`: This is implemented on top of `Vchuffman`. It divides data into short segments of equal length and groups segments compress well together with a single Huffman code table.
- `Vcama`: This collects records with the same length together to form tables that could then be compressed via `Vctable`. It was originally written to deal with an arcane data format specific to telephone data called AMA, hence the name. But it can also handle other general record types.
- `Vcmap`: This provides a collection of methods to map data from one character set to another. For example, different variants translate data between ASCII and various versions of EBCDIC.

2.3 The Vczip Command

The Vczip command enables usage of the available data transforms without low level programming. It provides a syntax to compose different transforms together to process a data file. In this way, end users can experiment with different combinations of transforms on their data to optimize compressibility.

Figure 4 shows the transform composition syntax. Lines 1 and 2 show that each transform is specified by a *name*, e.g., `delta` or `table`, followed by an optional sequence of arguments depending on the particular transform. Lines 3 and 4 show that such arguments are separated by periods. Lines 5 and 6 show that the list of transforms can be empty to signify usage of a default compression method defined by Vczip. But if not, it must start with `-m` and follow by a comma-separated sequence of transforms.

Figure 5 shows a snapshot of an experiment to explore different ways to compress data using Gzip, Bzip2 and various compositions of Vcodex data transforms. The file *kennedy.xls* was an Excel spreadsheet taken from the Canterbury Corpus (Bell and Powell, 2001).

- Lines 1–6 show the raw size of *kennedy.xls* and compression results by Gzip and Bzip2. Gzip compressed the data by roughly a factor of 5 while Bzip2 achieved a factor of 8 to 1.
- Lines 7–10 show Vczip instantiated as different variations of a BWT compressor. The `0` argument to the run length encoding transform `rle` meant that only runs of `0`'s were coded. The same argument to the move-to-front transform `mtf` restricted it to moving a data byte only after access. As that was analogous to Bzip2, about the same performance resulted. On the other hand, line 9 used a variant that aggressively predicted and moved data (Section 2.2) resulting in a 12 to 1 compression factor.
- Lines 11-14 show the use of the table transform `Vctable` (Vo and Vo, 2006) to treat the data as a 2-dimensional array of bytes and reorder data by column dependency before running same backend data transforms as earlier. The compression factor improved to about 19 to 1 with that. Inserting the Burrows-Wheeler transform after the table transform further improved the compression factor to nearly 30 to 1.
- Line 15 shows that the compressed data was decoded into a file `x` and compared against the original data to test correctness. As shown, decompression would always be done with the option `-u`, i.e., no knowledge of transforms used on encoding required.

Figure 6 shows another experiment to compress a large file of telephone data in a proprietary format called `AMA`. This type of data consists of records in which the first 4 bytes of a record tell the length of the record. The `Vcama` transform collected records of the same length together before passing them to the column dependency table transform `Vctable`. Again, this ability to exploit structures in data resulted in the best compression performance.

3 DATA ARCHITECTURE

The output of a transformation is data to store or transport. For maximal usability, the main issues to be addressed are: portability and self-description. Portability means that primitive types such as bits, integers and strings should be standardly encoded so that data

```
1. Transform -> {delta, table, bwt, huffman, ama, ...}
2. Transform -> {delta, table, bwt, huffman, ama, ...} . Arglist(Transform)
3. Arglist(Transform) -> Nil
4. Arglist(Transform) -> Arg(Transform) . Arglist(Transform)
5. TransformList -> Nil
6. TransformList -> "-m" TransformList , Transform
```

Figure 4: Language syntax to compose transforms.

```
1.   ls -l kennedy.xls
2.   -rw-------  4 kpv 1029744 Nov 11  1996 kennedy.xls

3.   gzip < kennedy.xls > out; ls -l out
4.   -rw-r--r--  1 kpv 206767 Apr 11 12:30 out

5.   bzip2 < kennedy.xls > out; ls -l out
6.   -rw-r--r--  1 kpv 130280 Apr 11 12:30 out

7.   vczip -mbwt,mtf.0,rle.0,huffgroup < kennedy.xls > out; ls -l out
8.   -rw-r--r--  1 kpv 129946 Apr 11 12:31 out

9.   vczip -mbwt,mtf,rle.0,huffgroup < kennedy.xls > out; ls -l out
10.  -rw-r--r--  1 kpv 84281 Apr 11 12:31 out

11.  vczip -mtable,mtf,rle.0,huffgroup < kennedy.xls > out; ls -l out
12.  -rw-r--r--  1 kpv 53918 Apr 11 12:31 out

13.  vczip -mtable,bwt,mtf,rle.0,huffgroup < kennedy.xls > out; ls -l out
14.  -rw-r--r--  1 kpv 35130 Apr 11 12:31 out

15.  vczip -u < out >x; cmp x kennedy.xls
```

Figure 5: Experimenting with different combinations of data transforms.

can be produced on one OS/hardware platform and transparently decoded on another. Self-description means that data can be decoded without knowledge of how they were encoded.

3.1 Portable Data Encoding

Transform writers must take care to ensure portability of output data. Vcodex provides a variety of functions to encode strings, bits, and integers in portable forms. String data are assumed to be in the ASCII character set. When running programs on a platform not based on ASCII, for example, IBM mainframes with EBCDIC, a function `vcstrcode()` can be used to transcribe string data from the native character set into ASCII for storage. For example, the identification string of a data transform is often constructed from its name (Section 2.2). As the name is in the native character set, it would need to be translated to ASCII for portability.

For bit encoding, it is assumed that the fundamental unit of storage is an *8-bit byte*. The bits in a byte are position from left to right, i.e., the highest bit is at position 0, the second highest bit at position 1, and so on. Then a sequence of bits would be simply imposed onto a sequence of bytes in that order. For example, the 12-bit sequence `101010101111` would be coded in the 2-byte sequence `170 240`. That is, the first eight bits, `10101010`, are stored in a byte so that byte would have value `170`. The last four bits, `1111`, are padded with `0`'s before storage so the representing byte would have value `240`.

Integers are unsigned and encoded in a variable-sized format originally introduced in the Sfio library (Korn and Vo, 1991). Each integer is treated as a number in base 128 so that each digit can be stored in a single byte. Except for the least significant one, each byte would turn on its most significant bit, MSB, to indicate that the next byte is a part of the encoding. For example, consider the integer 123456789 which is represented by four digits 58, 111, 26, 21 in base 128. Below are the coding of these digits shown in bits for clarity.

```
1.  ls -l ningaui.ama
2.  -rw-r--r--  1 kpv      kpv      73452794 Sep 12  2005 ningaui.ama

3.  gzip < ningaui.ama >out; lsl out
4.  -rw-r--r--  1 kpv      kpv      30207067 Apr 11 12:55 out

5.  bzip2 < ningaui.ama >out; lsl out
6.  -rw-r--r--  1 kpv      kpv      22154475 Apr 11 12:57 out

7.  vczip -mama,table,mtf,rle.0,huffgroup < ningaui.ama >out; lsl out
8.  -rw-r--r--  1 kpv      kpv      20058182 Apr 11 13:00 out
```

Figure 6: Compression based on data-specific structures.

10111010	11101111	10011010	00010101
MSB+58	MSB+111	MSB+26	0+21

3.2 Self-describing Data

A large file might need to be divided into segments small enough to process in memory. Each such segment is called a *window*. The internal representation of a transformed window would depend on what data transforms were used. But at the file level, that can be treated simply as a sequence of bytes.

```
1.  [ID_file:] 0xd6 0xc3 0xc4 0xd8
2.  [Reserved:] Size Data
3.  [Transforms:] Size
4.      [Transform1:] ID_transform
5.          [Argument:] Size Data
6.      [Transform2:] ID_transform
7.          [Argument:] Size Data
8.          ...
9.  [Window1:] Indicator
10      [Transformed data]: Size Data
11. [Window2:] Indicator
12      [Transformed data]: Size Data
13. ...
```

Figure 7: The self-describing format of transformed data.

Figure 7 shows the structure of encoded data in which each file consists of a set of header data followed by one or more `Window` sections and gives an example of header data.

- Line 1 shows that each file is identified by four starting bytes which are the ASCII letters `V`, `C`, `D` and `X` with their high bits on.
- Line 2 shows a `Reserved` section usable by an application to store additional data. As shown here and later, each data element is represented by a `Size` and `Data` pair that tells number of data bytes followed by the data itself.
- Lines 3–8 show the `Transforms` section which encodes the original composition of transforms. This starts with `Size`, the length in bytes of the rest of the section. Each transform consists of an identification string which, by convention, must be in ASCII followed by any additional data.
- Lines 9–13 show the list of `Window` sections. Each window section starts with an `Indicator` byte which can be one of the values `VC_RAW`, `VC_SOURCEFILE` and `VC_TARGETFILE`. `VC_RAW` means that the encoded data were not transformed. The latter two values are as defined in the Vcdiff Proposed Standard RFC3284 (Korn et al., 2002) to indicate that the data was delta compressed (Hunt et al., 1998) against data from either the source or target file respectively. Following the `Indicator` byte is the transformed data.

```
0xd6 0xc3 0xc4 0xd8   [ID_file]
0                     [Reserved size]
16                    [Transforms size]
delta\0               [ID_transform]
0                         [Argument size]
huffman\0             [ID_transform]
0                         [Argument size]
```

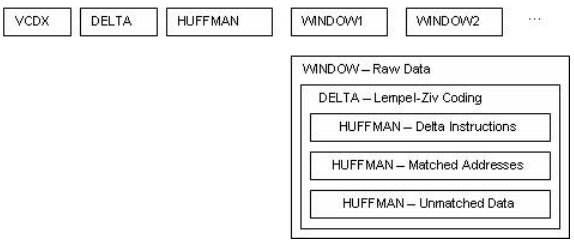

Figure 8: Header data and data schema of a delta compressor.

Figure 8 gives an example based on a delta compressor resulted from composing the transforms `Vcdelta` and `Vchuffman`. The top part shows first the literal bytes defining the header data which essen-

tially only coded the names of the transforms. The bottom part shows pictorially the various components of a compressed file. Each file starts with the header data which tells the file type via the four magic bytes `VCDX` and the transform composition sequence. The compressed data are divided into a sequence of windows. The raw data in each window is processed first by `Vcdelta` to generate the so-called delta instructions defined in the IETF RFC3284 (Korn et al., 2002). In this method, three different sections of data are generated, instructions, addresses of matches and unmatched data. Each section would then be processed separately by the following Huffman compressor `Vchuffman`. Partitioning of data by types in this way helps to improve compression as the different sections often have distinct statistical properties.

4 PERFORMANCE

An experiment was done to test compression performance by tailoring transform compositions to particular data. The small and large files from Canterbury Corpus (Bell and Powell, 2001) were used to compare Vczip, Gzip and Bzip2. Sizes were measured in bytes and times in seconds. Timing results were obtained on a Pentium 4, 2.8GHZ, with 1Gb RAM, running Redhat Linux.

Figure 9 presents compression and timing data in two tables. The top and bottom parts of each table show results of the small and large files respectively. The *Algs* entries show the algorithm compositions used to compress the files. Below are the algorithms:

- `B`: The Burrows-Wheeler transform `Vcbwt`.
- `M`: The move-to-front transform `Vcmtf` with prediction.
- `R`: The run-length encoder `Vcrle` with coding only runs of `0`'s.
- `h`: The Huffman coder `Vchuffman`.
- `H`: The Huffman coder with grouping `Vchuffgroup`.
- `T`: The table transform `Vctable`.
- `S`: The delta compressor `Vcsieve` using approximate matching.
- `D`: The transform `Vcsieve` with reverse matching and mapping of the letter pairs `A` and `T`, `G` and `C`, etc.

The *Bits* entries show the average number of bits needed to encode a byte with each row's overall winner in bold. Vczip only lost slightly to Bzip2 on *cp.html* and to Gzip on *grammar.lsp*. Its weighted compression rates of 1.21 bits per byte for the small files and 1.68 for the large files outdid the best results of 1.49 and 1.72 shown at the Corpus website. The pure Huffman coder `Vchuffman` outperformed its grouping counterpart `Vchuffgroup` on small files as the cost of coding groups became too expensive in such cases.

The best cases of Vczip typically ran somewhat slower than Bzip2 and Gzip. Although some speed loss was due to the general nature of transform composition, the major part was due to sophisticated algorithms such as move-to-front with prediction. In practice, applications often make engineering choices between faster data transforms with worse compression vs. slower ones with better compression.

For any collection of transforms, the number of algorithm combinations that make sense tend to be small and dictated by the nature of the transforms. For example, Huffman coding should never be used before other transforms as it destroys any structure in data. Thus, although not done here, it is possible to find an optimal composition sequence by trying all candidate composition sequences on small data samples.

5 RELATED WORKS

Vcodex introduces the notion of *data transforms* as standard software components to encapsulate compression algorithms for reusability. A variety of library packages such as *zlib* (Gailly and Adler, 2005) or *bzlib* (Seward, 1994) also aim at improving reusability. However, except for their overall external compression interface, the constituent algorithms in these packages are buried in the implemented code. For example, although Huffman coding is required in both *zlib* and *bzlib*, different versions were hard-coded with incompatible data formats. By contrast, new Vcodex data transforms are often crafted from existing ones via either composition or direct usage in implementation. Application systems with proprietary data types may also develop special data transforms for such types to enhance compressibility. In this way, exotic compositions of compression algorithms can be made to fit any particular data semantics and gain optimal compression performance.

A side of data compression often overlooked is the design of the output data. If not carefully done, such data may not be sharable across machines due to different architectures or may become stale and unusable over time due to algorithm evolution. Unfortunately, few compression tools have ever published their cod-

File	Size	Vczip Algs	Vczip Cmpsz	Vczip Bits	Bzip2 Cmpsz	Bzip2 Bits	Gzip Cmpsz	Gzip Bits
alice29.txt	152089	BMRH	42654	**2.24**	43202	2.27	54423	2.86
ptt5	513216	TBMRH	33149	**0.52**	49759	0.78	56438	0.88
fields.c	11150	BMRh	3036	**2.18**	3039	2.18	3134	2.25
kennedy.xls	1029744	TBMRH	35130	**0.27**	130280	1.01	206767	1.61
sum	38240	SBMRH	12406	**2.60**	12909	2.70	12920	2.70
lcet10.txt	426754	BMRH	105778	**1.98**	107706	2.02	144874	2.72
plrabn12.txt	481861	BMRH	143817	**2.39**	145577	2.42	195195	3.24
cp.html	24603	BMRh	7703	2.50	7624	**2.48**	7991	2.60
grammar.lsp	3721	BMRh	1274	2.74	1283	2.76	1234	**2.65**
xargs.1	4227	BMRh	1728	**3.27**	1762	3.33	1748	3.31
asyoulik.txt	125179	BMRH	39508	**2.52**	39569	2.53	48938	3.13
Total	2810784		426183	**1.21**	542710	1.54	733662	2.09
E.coli	4638690	DH	1137801	**1.96**	1251004	2.16	1341243	2.31
bible.txt	4047392	BMRH	786709	**1.55**	845623	1.67	1191061	2.35
world192.txt	2473400	BMRH	425077	**1.37**	489583	1.58	724593	2.34
Total	11159482		2349587	**1.68**	2586210	1.85	3256897	2.33

File	Vczip Cmptm	Vczip Dectm	Bzip2 Cmptm	Bzip2 Dectm	Gzip Cmptm	Gzip Dectm
alice29.txt	0.05	0.01	0.03	0.01	0.01	0.01
ptt5	0.27	0.08	0.03	0.01	0.02	0.01
fields.c	0.01	0.01	0.01	0.01	0.01	0.01
kennedy.xls	0.27	0.18	0.23	0.07	0.11	0.01
sum	0.03	0.01	0.01	0.01	0.01	0.01
lcet10.txt	0.17	0.08	0.12	0.06	0.05	0.01
plrabn12.txt	0.21	0.11	0.15	0.09	0.08	0.01
cp.html	0.01	0.01	0.01	0.01	0.01	0.01
grammar.lsp	0.01	0.01	0.01	0.01	0.01	0.01
xargs.1	0.01	0.01	0.01	0.01	0.01	0.01
asyoulik.txt	0.04	0.01	0.02	0.01	0.01	0.01
E.coli	7.85	0.10	1.92	1.03	1.93	0.08
bible.txt	2.47	1.04	1.64	0.72	0.55	0.06
world192.txt	1.31	0.62	1.04	0.42	0.23	0.03

Figure 9: Compression size (bytes), rate (bits/byte) and time (seconds) for the Canterbury Corpus.

ing formats. Two rare exceptions are the *Deflate* Format (Deutsch, 1996) used in *Gzip* and the *Vcdiff* Format (Korn et al., 2002) for delta encoding which have been standardized and sanctioned by the Internet Engineering Task Force to facilitate transporting of compressed data over the Internet. An appealing feature of the Vcodex self-describing data architecture is that the data format of each transform can be kept independent from others that may be composed with it in applications. This allows any specification and standardization effort to focus on the format of each transform independently from others. This should be contrasted, for example, with *Deflate* which must define in details both the output of its main Lempel-Ziv coder and also that of the low level Huffman coder used to clean up any remaining statistical redundancy in the Lempel-Ziv output.

6 CONCLUSION

Advances in compression are continually pulled by two opposing forces: *generalization* to devise techniques applicable to all data types and *specialization* to devise techniques exploiting specific structures in certain classes of data. General techniques such as Lempel-Ziv or Burrows-Wheeler Transform are easy to use and do perform well with most data. However, as seen in our experiments earlier, such techniques could seldom match the performance of algorithms specifically designed for classes of data such as tables, DNA sequences, etc. Unfortunately, the cost to develop data-specific compression tools can quickly become prohibitive considering the plethora of arcane data types used in large application systems. Vcodex provides a unique solution to this problem by applying a software and data engineering approach to compression. Its *data transform* interface frees algorithm designers to focus only on the data transfor-

mation task at hand without having to be concerned with other transforms that they may require. Further, users of compression also have the opportunity to mix and match data transforms to optimize the compressibility of their data without being locked into some compositional sequences picked by the tool designers. The best indication of success for Vcodex is that the platform has been in use for a few years in a number of large data warehouse applications handling terabytes of data daily. Tranform compositions properly tailored to data types help achieving compression ratios up to hundreds to one, resulting in significant cost savings. A variety of data transforms have been continually developed along with new data types without disrupting ongoing usage. Further, as the number of data types is limited per application system, simple learning algorithms based on training with small samples of data could often be developed to automatically find optimal combinations of transforms for effective compression. In this way, the platform has proven to be an effective tool to help advancing compression not just in specialization but also in generalization.

REFERENCES

T. Bell and M. Powell. The Canterbury Corpus, http://corpus.canterbury.ac.nz. Technical Report, 2001.

J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A Locally Adapative Data Compression Scheme. *Comm. of the ACM*, 29:320–330, 1986.

A. Buchsbaum, G.S. Fowler, and R. Giancarlo. Improving Table Compression with Combinatorial Optimization. *J. of the ACM*, 50(6):825–51, 2003.

M. Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Report 124, Digital Systems Research Center, 1994.

S. Deorowicz. Improvements to Burrows-Wheeler Compression Algorithm. *Software—Practice and Experience*, 30(13):1465–1483, 2000.

P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. In *http://www.ietf.org*. IETF RFC1951, 1996.

G.S. Fowler, A. Hume, D.G. Korn, and K.-P. Vo. Migrating an MVS Mainframe Application to a PC. In *Proceedings of Usenix'2004*. USENIX, 2004.

J. Gailly and M. Adler. Zlib, http://www.zlib.net. Technical report, 2005.

D.A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proc. of the IRE*, 40(9):1098–1101, Sept 1952.

J.J. Hunt, K.-P. Vo, and W.F. Tichy. Delta Algorithms: An Empirical Analysis. *ACM Transactions on Software Engineering and Methodology*, 7:192–214, 1998.

D.W. Jones. Practical Evaluation of a Data Compression Algorithm. In *Data Compression Conference*. IEEE Computer Society Press, 1991.

David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proc. of the Summer '91 Usenix Conference*, pages 235–256. USENIX, 1991.

D.G. Korn, J. MacDonals, J. Mogul, and K.-P. Vo. *The VCDIFF Generic Differencing and Compression Data Format*. Internet Engineering Task Force, www.ietf.org, RFC 3284, 2002.

D.G. Korn and K.-P. Vo. Engineering a Differencing and Compression Data Format. In *Proceedings of Usenix'2002*. USENIX, 2002.

H. Liefke and D. Suciu. Xmill: an efficient compressor for xml data. In *Proc. of SIGMOD*, pages 153–164, 2000.

G. Manzini and M. Rastero. A Simple and Fast DNA Compression Algorithm. *Software—Practice and Experience*, 34:1397–1411, 2004.

J. Seward. Bzip2, http://www.bzip.org. Technical report, 1994.

W. F. Tichy. RCS—a system for version control. *Software—Practice and Experience*, 15(7):637–654, 1985.

B.D. Vo and K.-P. Vo. Using Column Dependency to Compress Tables. Data Compression Conference, 2004.

B.D. Vo and K.-P. Vo. Compressing Table Data with Column Dependency. *Theoretical Computer Science, accepted for publication*, 2006.

K.-P. Vo. The Discipline and Method Architecture for Reusable Libraries. *Software—Practice and Experience*, 30:107–128, 2000.

I.H. Witten, M. Radford, and J.G. Cleary. Arithmetic Coding for Data Compression. *Comm. of the ACM*, 30(6):520–540, June 1987.

J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. on Information Theory*, 23(3):337–343, May 1977.

J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Trans. on Information Theory*, 24(5):530–536, 1978.