CHAPTER 7

Burrows–Wheeler Compression[1]

PETER FENWICK

7.1 INTRODUCTION

Block-sorting compression, or "Burrows–Wheeler compression" is a relatively new algorithm of good compression and speed, first presented by Burrows and Wheeler in 1994 [1], although Wheeler had discovered the basic algorithm some 10 years earlier. In contrast to most other compression algorithms it treats the incoming text as a block, or sequence of blocks, with transformations on each block.

An initial study of Burrows–Wheeler compression was conducted by Fenwick, with a series of reports [2–4] and papers [5, 6]. Later work on the topic, by many other authors, will be cited throughout this chapter. Particular mention should be made of Seward [7], who took Fenwick's initial implementations and produced a commercial-strength implementation, BZIP, released through the Free Software Foundation.

Most of the compression results here will be for the Calgary Corpus [8], a long-standing and somewhat arbitrary collection of files which has become a *de facto* standard for compression evaluation. Some results also use the newer Canterbury Corpus [9], a rather more systematically designed collection of files, chosen to reflect current compression needs and practices. Compression will be stated as bpc (bits per character), which is generally equivalent to the older term "bits per byte."

[1] Author's note: The material of this chapter, while quoting extensively from other work, is in part a summary of my own experience and thoughts in working with the Burrows–Wheeler compression algorithm. Some of it is accordingly rather less formal in style than might otherwise be the case, as I give more personal opinions on various aspects. Where my own work already appears in the public domain it is cited in the normal way, but unpublished material simply refers to "the author."

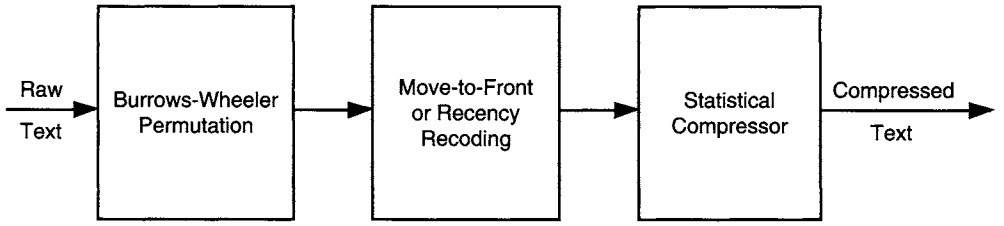

FIGURE 7.1
Stages of Burrows–Wheeler or block sorting compression.

7.2 THE BURROWS–WHEELER ALGORITHM

In its original form, the compression algorithm processes the text in the three stages shown in Fig. 7.1. Later developments may vary from this basic scheme.

For now we just outline each of these three steps:

1. The heart of the compression algorithm is the Burrows–Wheeler transformation on the incoming text, which permutes or reorders the text into a form which is especially suited to processing by the following two stages, with clusters of similar symbols and many symbol runs.
2. The permutation step is followed by a recoding of the permuted text by a Move-To-Front (MTF) or recency recoder. (For a given symbol x, if n different symbols have been seen since the last occurrence of x, then the symbol x is recoded to the integer n.) The essence of this step is that recent symbols recode into small integers and symbol runs recode into runs of zeros; the MTF output is dominated by small values and especially zeros.
3. The final step is some form of statistical compressor. An adaptive Huffman or similar bit-oriented coder is appropriate where speed is more important, but an arithmetic coder may be better if good compression is emphasized.
4. Although it is not shown in the diagram, many implementations use run-length encoding in one or more parts of the process. For example, many statistical encoders are assisted by transforming the zero-runs from the MTF stage. Again, run-encoding the initial text may greatly accelerate some implementations of the permutation stage.

7.3 THE BURROWS–WHEELER TRANSFORM

Burrows–Wheeler, or block-sorting, compression is based upon two quite separate transformations, a forward transformation, which permutes the input data into a form which is easily compressed, and a matching reverse transformation, which recovers the original input from the permuted data. The data is always considered in blocks, which may be as large as the entire file or may be a few tens or hundreds of kilobytes. (The discussion here is based on Fenwick's reports [2–4].)

7.3.1 The Burrows–Wheeler Forward Transformation

There are two approaches to describing the forward transformation. Burrows and Wheeler describe it as the steps:

1. Write the input as the first row of a matrix, one symbol per column.
2. Form all cyclic permutations of that row and write as them as the other rows of the matrix.
3. Sort the matrix rows according to the lexicographical order of the elements of the rows.
4. Take as output the final column of the sorted matrix, together with the number of the row which corresponds to the original input.

In terms which are more familiar to workers in data compression, the transformation may be described as:

1. Sort the input symbols, using as a sort key for each symbol the symbols which immediately follow it, to whatever length is needed to resolve the comparison. The symbols are therefore sorted according to their *following* contexts, whereas conventional data compression uses the *preceding* contexts. (Some implementations do use preceding contexts, but the discussion is easier with following contexts.)
2. Take as output the sorted symbols, together with the position in that output of the last symbol of the input data.

In either case the effect of the sorting is to collect together similar contexts. The (assumed) Markov structure of the input implies that only a few symbols are likely to occur in association with adjacent contexts. Any region of the permuted file will probably contain only a very few symbols and this locality can be captured with a Move-to-Front compressor.

7.3.2 The Burrows–Wheeler Reverse Transformation

Perhaps the most surprising thing about the forward transformation described above is that it is easily reversed! The reverse transformation depends on two observations:

1. The transformed input data is a permutation of the original input symbols.
2. Sorting the permuted data gives the first symbol of each of the sorted contexts.

But the transmitted data is ordered according to the contexts, so the nth symbol transmitted corresponds to the nth ordered context, of which we know the first symbol. So, given a symbol **s** in position i of the transmitted text, we find that position i within the ordered contexts contains the jth occurrence of symbol **t**; this is the next emitted symbol. We then go to the jth occurrence of **t** in the transmitted data and obtain its corresponding context symbol as the next symbol. The position of the symbol corresponding to the first context is needed to locate the last symbol of the output and from there we can traverse the entire transmitted data to recover the original text.

7.3.3 Illustration of the Transformations

To illustrate the operations of coding and decoding we consider the text "`mississippi`", as illustrated in Fig. 7.2. The first context is "`imississip`" for symbol "`p`", the second is "`ippimissis`" for symbol "`s`", and so on. The permuted text is then "`pssmipissii`", and the initial index is 5 (marked with "→"), because the fifth context corresponds to the original text.

To decode we take the encoded string "`pssmipissii`", sort it to build the contexts ("`iiiimppssss`"), and then build the links shown in the last column in Fig. 7.2. The four "`i`" contexts link to the four "`i`" input symbols in order (to 5, 7, 10, and 11, respectively). The "`m`" context links to the only "`m`" symbol, and the two "`p`"s and four "`s`"s link to their partners in order.

Symbol	Context	Index	Symbol	Context	Link
p	imississip	1	p	i...	5
s	ippimissis	2	s	i...	7
s	issippimis	3	s	i...	10
m	ississippi	4	m	i...	11
→i	mississipp	5	i	m...	4
p	pimississi	6	p	p...	1
i	ppimississ	7	i	p...	6
s	sippimissi	8	s	s...	2
s	sissippimi	9	s	s...	3
i	ssippimiss	10	i	s...	8
i	ssissippim	11	i	s...	9

FIGURE 7.2
The forward and reverse transformations.

To finally recover the text, we start at the indicated position (5) and immediately link to 4. The sorted received symbol there yields the desired symbol "m" and its immediately following context symbol, the fourth "i". We then link to 11 get the "i", and so on for the rest of the data, stopping on a symbol count or the return to the start of the file.

7.3.4 Algorithms for the Reverse Transformation

The forward transformation is largely concerned with sorting and will be described later. Here we describe the steps of the reverse transformation, assuming the notation:

N denotes symbols in the file	
n denotes the symbols in the alphabet	
S denotes the array of received symbols	S[1..N]
T is the array of recovered symbols	T[1..N]
K denotes the counts of each symbol	K[0..n-1]
L denotes the links to resolve permutation	L[1..N]
M is the mapping array for symbols	M[0..n-1]

The received (permuted) symbols are assumed in S, with counters, etc., appropriately initialized. The first step is to just count the occurrences of each symbol; it may be performed as the file is read in, decompressed, and the symbols recovered from the inverse Move-To-Front.

```
for(i = 1; i <= N; i++)
  K[S[i]]++;              /* count input symbols */
```

The next step involves building the table of the initial positions of each context. Remember that these are ordered according to the lexicographical ordering of the symbols and that there are as many contexts starting with "a" as there are occurrences of "a" in the input.

```
M[0] = 1;
for (j = 1; j < n; j++)
  M[j] = M[j-1]+K[j-1];
```

We are now in a position to build the links which will be used to traverse the received data. The mapping array M initially points to the first positions of the contexts. As a symbol s is used from a context, M[s] is incremented so that it tracks the next context starting with "s".

```
for(i = 1; i <= N; i++)
   {
   s = S[i];            /* get current symbol */
   L[i] = M[s];         /* set link from mapping table */
   M[s] ++;             /* increment mapping */
   }
```

Although we never actually prepare an explicit list of the contexts (or rather their initial symbols), the steps so far are equivalent to using such a sorted list. Finally we can recover the original text.

```
ix = initial_position;          /* start at correct position */
for (i = 1; i <= N; i++)
   {
   ix = L[ix];                  /* step on via the links array */
   T[i] = S[ix];                /* copy symbol to output */
   if (ix == initial_position)
     break;                     /* an alternative termination */
   }
```

There are three passes through the entire file (one of which may be combined with reading) and one pass through the alphabet. The operations are all quite simple and usually add little to the input/output costs of handling the file or, in most cases, to the cost of decompressing the file.

7.4 BASIC IMPLEMENTATIONS

We now describe an implementation of the entire compressor, generally leading to the version described by Fenwick [4–6]. It will bring in some of the experience gained from those early implementations and that acted a basis for later developments described in Section 7.5. When this version was completed in 1995 (published in 1996) it was nearly as good as the best PPM compressors at the time and certainly illustrated the potential of the Burrows–Wheeler or block-sorting algorithm.

7.4.1 The Burrows–Wheeler Transform or Permutation

In this section we deal only with the forward transform; the reverse transform is quite straightforward and independent of the forward implementation. In the original report Burrows and Wheeler present the algorithm in terms of sorting, but also mention that suffix trees provide an alternative and possibly better implementation. Much of the later work on the algorithm has actually dealt with suffix tree implementations. The author, however, with a greater interest in improving the final coding, has persisted with the sorting algorithm as described here.

All of the author's implementations have used a combination of a bucket sort (65,536 buckets, based on symbol digrams) and the standard C *qsort* implementation of QuickSort. An initial pass over the input string builds up, for each digram, a table of the indices of occurrences of

Comparisons for string "mississippi"	Symbol	Word
	m	miss
	i	issi
First comparison →	s	ssis
	s	siss
	i	issi
	s	ssip
Next comparison →	s	sipp
	i	ippi

FIGURE 7.3
Example of character striping.

that digram. The sort is a tag sort, permuting these indices. We also build a word array, striping symbols across successive words allowing 4 (or possibly 8) symbols to be compared at a time, with the comparison striding 4 or 8 words through the array, as shown in Fig. 7.3.

When presented with the string "mississippi", based on the digram "mi", the comparison procedure supplied to *qsort* will first test the word "ssis" (the first two letters must match because they are in the same digram bucket) and then the word "sipp". Only if these two both succeed will the main comparison loop be entered.

A walk through the digrams in order and the sorted tags within the digrams gives the complete ordering by context; the symbols immediately before each context are those emitted.

Two problems occurred with this sort routine. The first is that some implementations of *qsort* will compare an entity against itself; self-comparisons must be trapped at the very beginning of each string comparison (are the initial indices identical?).

The second problem is that some files, such as the Calgary Corpus pic, have long runs of identical symbols, and sorts on these runs may require enormous numbers of comparisons to resolve. In Fenwick's original implementation a run of four or six identical symbols always signals a run. All immediately following occurrences of that symbol are replaced by a run-length or symbol count (which may be zero). This preprocessing step is intended only to help files such as pic and has little effect on other files.

Wheeler [10] described a sorting technique which starts as a radix-256 sort, based on the initial symbols of the contexts and with symbols striped across successive long words, as described above. He then sorts the buckets into increasing size and sorts the smaller ones first. After each bucket is sorted, each of the words (32 bits) referred to by that bucket has its rightmost 24 bits replaced by its ordinal position in the sorted bucket, leaving the symbol itself in the leftmost 8 bits. Thus each symbol in the bucket is uniquely tagged according to its lexicographical ordering and whenever two already processed symbols are compared (as words of four symbols or as position+symbol) the comparison is resolved immediately.

7.4.2 Move-To-Front Recoding

The aim and object of any statistical text compressor is to produce a highly skewed symbol distribution so that some symbols are known to be much more likely than others in a particular context or position in the output. In general, the greater the skewness, the better the compression, as the few likely symbols can be encoded in very few bits. In the standard Burrows–Wheeler compressor this skewed distribution is produced by a MTF recoding of the permuted text.

Table 7.1 Move-To-Front Statistics from Calgary Corpus

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans
Average MTF movement													
2.50	2.45	2.25	36.05	3.80	23.47	10.26	3.27	2.81	1.30	3.90	1.99	2.18	1.96
Fraction of MTF codes which are zero													
66.8%	49.8%	60.8%	35.8%	57.9%	50.6%	68.1%	58.4%	55.4%	87.4%	60.3%	72.9%	74.0%	79.2%

Symbol	MTF List Before	MTF List After	Emitted Code
p	abcd..	pabc..	?
s	pabc..	spab..	?
s	spab..	spab..	0
m	spab..	mspa..	?
i	mspa..	imsp..	?
p	imsp..	pims..	3
i	pims..	ipms..	1
s	ipms..	sipm..	3
s	sipm..	sipm..	0
i	sipm..	ispm..	1
i	ispm..	ispm..	0

FIGURE 7.4
Example of MTF recoding.

While the term Move-To-Front is strictly a descriptive one based on the usual implementation, the technically more accurate description is "recency" recoding. Each symbol is replaced by the number of different symbols which have appeared since its last occurrence in the data stream.

In Fig. 7.4 we show the symbols from the earlier Burrows–Wheeler permutation of "`mississippi`" and the recoding using a MTF list. (Visual inspection shows that it implements recency recoding.) The MTF list is initially some permutation of the full symbol alphabet. Each symbol is recoded as its current position in the MTF list; the list is then shuffled to bring that symbol to the front of the list, retaining the order of the others. Here the *first* occurrence of each symbol is coded as some arbitrary value, shown as "?" Thereafter, recent symbols tend to be coded into small values, irrespective and independent of the value for the first reference.

Move-To-Front recoding was first used in data compression by Bentley *et al.* [11], who used a word-based compressor with quite large movements of large objects through the MTF list or table. Much of their discussion concerned the merits of various data structures to maintain the table and allow updates at low cost. Here, however, the objects to be moved are simple (just single 8-bit bytes or symbols) and the average movement is small, often only two or three positions, as shown for the Calgary Corpus in Table 7.1, although the movement is much greater for the three less compressible binary files. The table also shows the proportion of the MTF codes which are zero; the overall average is about 62%.

The initial implementation used two parallel byte arrays: one the current list itself and the other a mapping array to find the current position of each byte or symbol value. Later implementations dispensed completely with the mapping array, with a simple search along the list to find the index

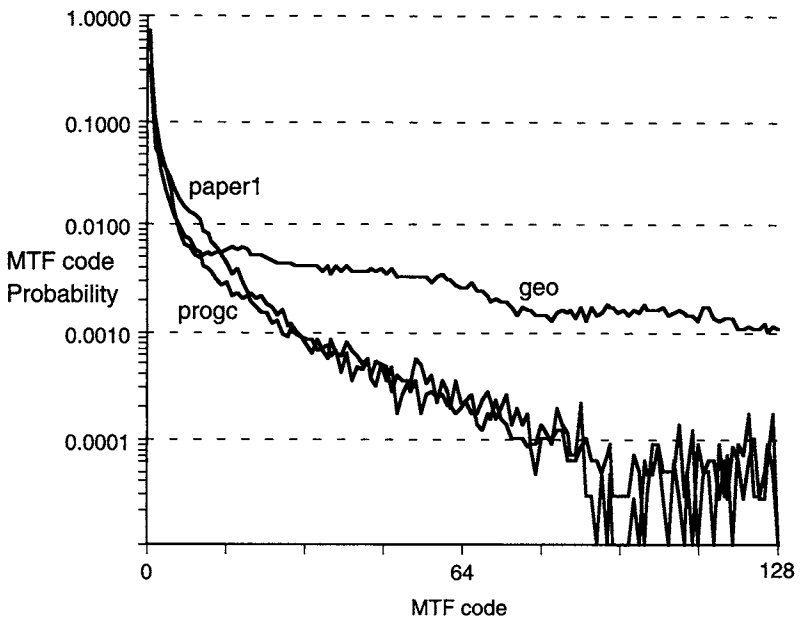

FIGURE 7.5
MTF code probabilities for three files.

of the symbol. The average overhead of the search is less than the average overhead of shuffling the mapping array.

7.4.3 Statistical Coding

Most of the work on high-performance Burrows–Wheeler compressors has used arithmetic coders, or a complex of such coders, in the final stage. Huffman and similar bit-wise coders can be used, and are used, but will not be discussed here.

As stated earlier, good compression implies a skewed probability distribution of the encoded symbols. Normal text follows a Zipf distribution in which the rth most frequent symbol has a frequency proportional to r^{-1}. Work by Fenwick [4, 6] showed that the MTF output in Burrows–Wheeler compression has a distribution similar to r^{-2}, with the exponent being larger for more compressible files. The more skewed distribution and much greater range of probabilities have major consequences for a final arithmetic compressor.

Figure 7.5 shows the distribution of MTF code probabilities for three files of the Calgary Corpus. Two are text files (but the line for `paper1` is generally above that for `progc`, showing its lesser compressibility) while `geo` has many symbols at higher ranks and is correspondingly less compressible. The text files show a range of probabilities exceeding 10,000:1.

Initial work (by the author) used a combination of Witten's "CACM" arithmetic coder [12], in conjunction with a new data structure for holding the frequency counts[13] which allowed greater choice of the coding range and per-symbol increment. When Moffat *et al.* [14] released their improved coder, that was substituted and immediately gave much poorer compression! Solving this problem led to a much better understanding of Burrows–Wheeler compression.

Moffat's coder is designed for a conventional PPM-style compressor where each context has its own coding model and statistics accumulate for many models as compression proceeds. Most versions of Burrows–Wheeler compression maintain just a single coding model (if only because it can be very difficult to decide when to switch between models, but see later work in Section 7.11).

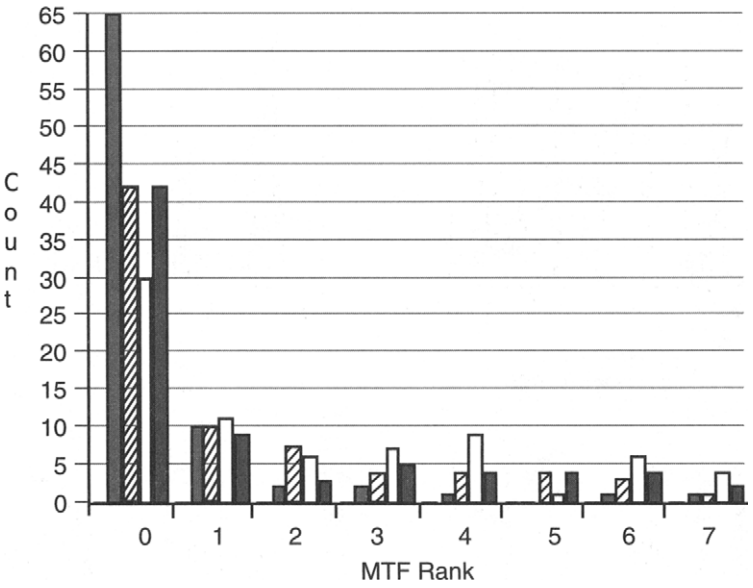

FIGURE 7.6
MTF code frequencies for successive areas of a sorted file.

The coding statistics, however, are far from stable; as an entropy source the MTF output is far from ergodic. We find that a file, almost any file, transforms into bursts of high entropy on a background of quite low entropy. The difference between files is that in a relatively compressible file the bursts of high entropy tend to be shorter rather than lower in absolute value. The Moffat coder had to be redesigned to increment and scale counts in a manner similar to that of the original CACM coder.

Figure 7.6 shows the counts of various small MTF ranks in four successive 100-symbol blocks of the file `paper1`. The variation is considerable, with many MTF counts varying by more than 2:1 over even this small section of file. Thus statistics accumulated for one section of the file may have to be adjusted, or even forgotten completely, quite soon afterward. The statistical coder needs considerable agility to track these changes.

Figure 7.7 shows the approximate coding cost in bits for each symbol of sequences of 200 symbols from the files `paper1` and `obj1`. We see a general mixture of extreme activity, alternating with very quiet periods, even for a relatively incompressible file like `obj1`. The horizontal axis is in each case the symbol position in the permuted file.

An arithmetic coding model for Burrows–Wheeler compression must combine the two attributes:

1. It must handle a wide range of symbol probabilities, from 10,000:1 for a typical text file to 65,000:1 for some binaries. This requires a significant difference between the limit for the accumulated count and the per-symbol increment. The ratio need not be as high as 10,000 or 50,000 to 1, but certainly should be high.
2. It must be able to adapt quickly as the output moves between the bursts and the quiescent background. Fast adaptation usually implies a small ratio of limit: increment to force frequent overflows and scaling of the counts.

At first sight the two requirements of range and agility are largely incompatible, but may be reconciled by combining coding models. Two forms are the cascaded model and the structured model, both outlined in Fig. 7.8. The extension to handle larger alphabets should be obvious.

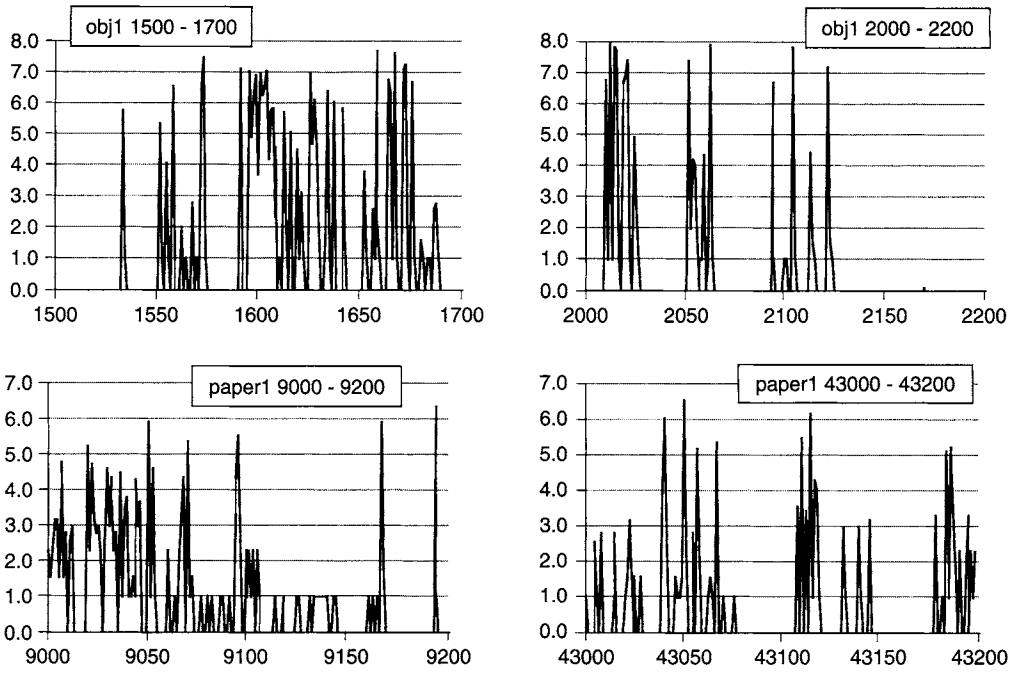

FIGURE 7.7
Coding cost (bits per symbol) for sections of files `obj1` and `paper1`.

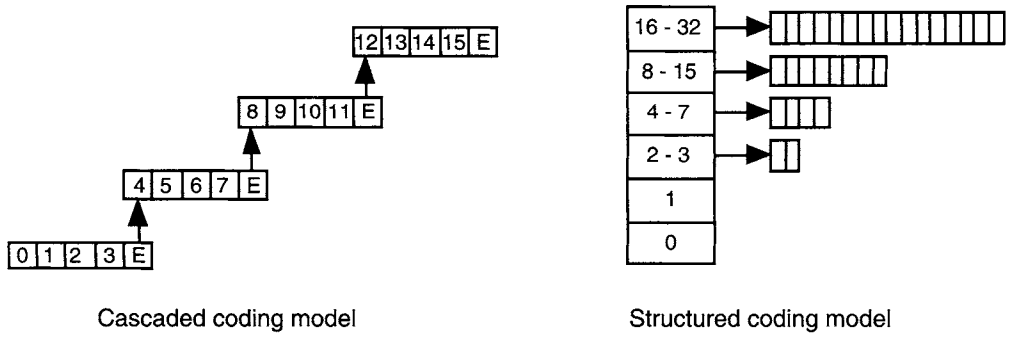

FIGURE 7.8
Cascaded and structured coding models.

In both cases the barred rectangles are arithmetic coding models handling an appropriate number of symbols.

1. In the cascaded model each component handles a small range of consecutive values, with an "escape" to overflow into the next model. In the example here, coding a "13" would result in the codes E, E, E, 1 as the coder drives through the successive components of the whole model. If symbols in the range 0–3 are emitted, only the very first coding model is used and later models remain untouched. A sequence of larger symbols will enter the later coding models but the early models affect only the probability of the escape symbol(s). Successive levels of the cascade need not have the same size as in the example. It is probably more sensible to have them growing as do the leaf nodes of the structured model.

Table 7.2 Typical Parameters for Structured Coding Model

Group	Range	Count Limit	Count Increment
Trunk	—	2000	20
0	0	—	—
1	1	—	—
2	2–3	256	1
3	4–7	256	1
4	8–15	512	1
5	16–31	1024	1
6	32–63	2048	1
7	64–127	4096	1
8	128–255	8192	1

2. The structured model has a "trunk" model with several "leaf" models. Instead of a sequence of possible escapes, most symbols have an initial trunk code, followed by an appropriate leaf code coding a residue. For example "6" would be encoded as an initial "3" (element 3 of the trunk) and then "2" as the residue within the leaf. Exactly as with the cascaded model a sequence of small codes leaves the "high-value" portion of the model untouched. The structured model was first devised by Fenwick [6] and designed to equalize the probabilities served by each entry of the trunk model. In retrospect this equalization is relatively unimportant. Much more significant is the ability of the trunk model to decouple the effects of high and low code values.

Both of the models rely on one or more small coding models which handle only a few symbols and can adapt very quickly to changing statistics. Experience shows that both give a similar performance. Fenwick [3] used a cascade model with initial binary models feeding into a full background model, for a Calgary Corpus result of 2.363 bpc, which is very close to his final 2.338 bpc for the structured model. (Much of the difference probably comes from using run-encoding in the structured model but not with the cascade.)

By way of illustration, Table 7.2 shows typical arithmetic coding parameters for a structured model; the values for the leaf nodes are not critical.

7.4.3.1 Run-Length Coding

A final way of improving compression is to run-encode the streams of 0's which dominate the MTF output. Strictly this should be unnecessary with a good arithmetic compressor, because over a stream of N consecutive 0's, the cost per encoded digit tends to zero, while the cost of encoding any other non-0 symbol to terminate the run tends to $\log_2 N$ bits. But as the count of N can also be encoded in $\log_2 N$ bits, we can expect the costs to be similar. In practice the arithmetic coder does not adjust immediately to the ideal coding and an explicit run-length coding gives slightly better results.

A suitable method of encoding zero runs, suggested by Wheeler, is to expand the alphabet by one symbol so that most values x are encoded as the value $x+1$. For a run of N zeros, the actual code values {0, 1} are used to encode the binary representation of $(N + 1)$, least significant bit first and omitting the most significant 1 bit. The leading 1 is implied by the code >1, which follows the run.

An example of this encoding is shown in Table 7.3. At the small price of increasing the alphabet by one symbol it gives a very efficient coding of runs of zeros, never increasing the data length. In Table 7.3 it saves 4 symbols out of the original 15. Using the run-length coding gave about a 1% improvement in compression.

Table 7.3 Wheeler's Run-Length Recoding of a Data Stream

Original Data	6	2	1	0	1	0	0	2	3	0	0	0	0	0	4
Recoded Data	7	3	2	0	2		1	3	4				0	1	5

Table 7.4 Fenwick's 1996 Compression of Calgary Corpus Files: Average = 2.338 bpc

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans
1.946	2.391	2.030	4.500	2.499	3.865	2.457	2.458	2.413	0.767	2.495	1.715	1.702	1.495

One final, small, improvement comes from limiting the alphabet to 128 symbols for text files, giving the final results for Fenwick's 1996 implementation [4–6] as shown in Table 7.4.

7.5 RELATION TO OTHER COMPRESSION ALGORITHMS

Almost as soon as it was first described, Cleary and Teahan [15] showed that the Burrows–Wheeler transform had a formal correspondence to their PPM* compressor and was described by a similar context tree. However, an important difference lies in the manner by which that tree is used and traversed.

In on-line or sequential compressors such as PPM the tree is built "on the fly" and functions as a storage or repository for information on contexts which have been seen already. The active area moves all around the tree as compression proceeds, moving to an existing leaf or context for each symbol and extending that information or extending the tree in accordance with the current symbol and its context.

In Burrows–Wheeler compression, the tree is built, explicitly or implicitly, by the initial transform and is then traversed in lexical order. The context described by a given node may be extended into longer contexts for a while but once left is abandoned entirely and never revisited. The context tree of PPM is thereby replaced by a stack and the main function of the tree, storing context history and statistics for future reuse, is eliminated completely.

Effros [16] shows the equivalence of the context trees of PPM*, the suffix trees of some Burrows–Wheeler implementations, and the pattern-matching trees often used in Ziv–Lempel compressors, emphasizing the essential unity of these three apparently disparate methods. She then uses this knowledge to implement a compressor combining "PPM performance with BWT [Burrows–Wheeler Transform] complexity," albeit closer to PPM than to BWT.

7.6 IMPROVEMENTS TO BURROWS–WHEELER COMPRESSION

We now take the compressor of the previous section and examine various improvements which have been proposed. The paper by Deorowicz [17] includes an excellent review of much of this material and readers should consult that paper for further details. In line with the compression stages as used above, the improvements are described here under the broad headings of:

1. Preprocessing the input.
2. The Burrows–Wheeler transform.

3. The Move-To-Front operation.
4. The final statistical compressor.

In fact the division is not as simple as this because often a problem in one area must be solved or have repercussions in another.

One point which must be raised is the question of overheads. It is all too easy to propose a solution which looks good until one counts the cost of the overheads. For example, suppose that we want to recode the input in some way and experiments show that this recoding improves compression of text files by 1%, a possibly useful amount. A text file has an alphabet of around 100 symbols, or perhaps a few less. The simplest way of describing the recoding is to transmit a recoding vector for these 100 symbols. This is unlikely to be very compressible and adds 100 characters to the compressed data. Given that ordinary text compresses at around 2.5 bpc, the 100 characters used to describe the recoding is equivalent to 320 characters of input text. Only if the input file exceeds 32,000 characters does the 1% gain exceed the 100-byte cost. Although some clever techniques may reduce the 100-byte description overhead, the overhead is still a cost which must be balanced against the gain. All too often the benefits are minimal.

7.7 PREPROCESSING

Two of the simpler types of preprocessing have been mentioned already, modifying the compression for a partial input alphabet and run-encoding the input before the permutation step. Partial input alphabets will be considered in Section 7.9 when the Move-To-Front operation is considered. Run-length encoding was introduced into the preprocessor only to aid in sorting files such as `pic`, which have long symbol runs. It was apparent from the beginning that run-length encoding destroys some of the symbol context and reduces compression by perhaps 0.1%. Balkenhol *et al.* [18] suggest that run-encoding should be avoided if it reduces the file size by less than 30%. This step is in any case unnecessary if a suffix-tree algorithm is used for the permutation.

More significant preprocessing was proposed by Chapin and Tate [19]. Their techniques apply some permutation (or Caesar cipher) to the input text with the idea of minimizing the costs of the MTF recoding. If symbols which are in some sense "similar" are brought together in the recoding, the cost of moving from one to the other in the MTF recoding will be smaller and the compression will improve. They found that just bringing the vowels together (a permutation to "`aeioubcdfg...`") improved compression of text files by 0.3–0.5% but a full analysis, akin to a Traveling Salesman Problem, had less effect.

Overall, they improved the compression from 2.410 to 2.340 bpc for the Calgary Corpus, an improvement of about 3% in their implementations. Unfortunately all of the recoding methods must transmit the recoding information. This may be a simple codebook number (for a predefined recoding like `aeioubcdfg...`) but may require a dictionary containing the entire input alphabet. When we consider that Chapin reports that the saving of a recoding seldom exceeds 100 bytes for the smaller text files, it is doubtful whether anything more complex than a simple "vowel" codebook heuristic is worthwhile.

7.8 THE PERMUTATION

We might consider that an ideal permutation simply lists the input symbols or letters with their frequency counts, followed by the reordering rules. Unfortunately for most permutations the reordering rules are comparable in size to the original text! It appears that the Burrows–Wheeler

transform is almost unique; it is conceptually simple to perform and invert and adds just one integer of overhead to facilitate the inverse transform. Thus while we must remain open to the possibility of a better permutation algorithm, Burrows–Wheeler remains the best known.

There are nevertheless two other permutations to consider here

- Chapin and Tate [19] also introduce the idea of a "reflected sort," analogous to the reflected binary or Gray code. In the Gray code, representations of successive integers differ in an most one place. Chapin modifies the sort comparison to alternate increasing and decreasing comparisons on successive bytes of the comparand strings. To quote Chapin "Whenever a symbol in a column changes between string i and string $i + 1$, the sort order of all following columns is inverted."
- Arnavut and Magliveras [20] describe the *Lexical Permutation Sorting Algorithm*, a generalization of the Burrows–Wheeler transform.

Suffix trees were introduced by Burrows and Wheeler as an alternative to the explicit sorting algorithm. In one sense the suffix trees change nothing, being just an alternative way of implementing the Burrows–Wheeler transform. But particularly in the worst case they are faster and may be more space-efficient than a sorting implementation. They are therefore preferred by many authors for situations where compression speed is important.

Several authors consider other improvements to the sorting algorithm. Schindler [21] sorts using finite-length contexts as the keys, but at the cost of a more expensive reverse transform and slightly poorer compression. Sadakane [22] describes a fast and economical algorithm for sorting suffix arrays, suggesting that the *average match length* or average number of comparisons needed to verify the correct sorting is a measure of the difficulty of producing the Burrows–Wheeler transform. Seward [23] compares several sorting algorithms.

One final point to consider is the direction of the sort comparison (or its equivalent with suffix trees). One natural direction is a conventional lexicographical comparison in the forward direction (so that "`somewhat`" precedes "`somewhere`"). But equally natural is a reverse comparison ("`anywhere`" follows "`somewhere`") because in compression we traditionally lead into the symbol under consideration. In his very early work on the entropy of English, Shannon [24] found that although the second approach is more natural for people to use, there is little real difference between the prediction ability of the two directions.

For the most part, the choice seems to be according to the whim of the implementer. Table 7.5 shows some results from Fenwick [3], for a very early version of the compressor and also for a much later version of the compressor with better final coding and other optimizations. Both cases show results for forward comparisons (trailing contexts) and reverse comparisons (leading contexts) of the incoming text.

Boldface text in Table 7.5 indicates where the compression is better by about 1% than in the other direction. In few cases are the differences significant, except in the file `geo`, which shows a change of nearly 4% between the two directions. This difference has been noted by

Table 7.5 Comparisons of Context Directions for Two Compressors

	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans	Average
Early compressor, little optimization															
Forward	**2.132**	**2.524**	2.198	4.810	2.678	4.202	2.709	2.606	**2.570**	0.830	2.680	1.854	1.837	**1.613**	2.517
Reverse	2.182	2.549	2.218	**4.748**	2.695	4.201	**2.730**	2.629	2.593	0.830	2.683	1.851	1.838	1.625	2.527
Later compressor, more optimization															
Forward	**1.926**	2.356	2.012	4.430	2.464	3.796	**2.433**	**2.439**	**2.387**	0.753	2.476	1.697	1.702	**1.488**	2.311
Reverse	1.953	2.363	2.012	**4.268**	2.476	3.765	2.478	2.464	2.420	0.758	2.478	1.697	1.716	1.503	2.311

C2	66	D8	00	C2	34	DC	00	42	10	60	00	42	35	4C	00
42	32	54	00	42	38	BC	00	42	63	9C	00	42	96	3C	00
42	99	28	00	42	9C	8C	00	42	9B	B8	00	42	89	BC	00
42	6C	D4	00	42	48	90	00	41	6B	40	00	C2	2D	90	00

FIGURE 7.9
Sixty-four consecutive bytes from the file `geo`, broken into floating point values.

several authors; for example, Balkenhol *et al.* [18] recommend using leading contexts or reverse comparisons where the file has an alphabet of 256 symbols or nearly so. However, `obj1` and `obj2` also have large alphabets and `obj1` shows a negligible difference, while `obj2` is better with trailing contexts or forward comparisons.

The explanation is just that `geo` consists of 4-byte (32 bit) floating-point numbers whose least significant (rightmost) byte is usually zero and independent of the next byte of that word. The zero byte is, however, adjacent to the first byte of the *next* word, which has relatively few values and is a good predictor of the zero byte *preceding* it in the file, as shown in Fig. 7.9. Working only from alphabet size is quite misleading. While some files do compress much better with comparisons in one direction, that direction must be decided individually, on a case by case basis. (In any case it should be noted that just reversing the context direction for `geo` typically gives an improvement of 0.012 bpc on the Calgary Corpus average, which is a significant part of the improvement quoted by many authors.)

7.8.1 Suffix Trees

In their initial paper Burrows and Wheeler suggested the use of suffix trees (see McCreight [25]) as an alternative to an explicit sort. That route has been followed by many workers, especially by those interested in improving the compression speed. But while they can give good speed, the suffix-tree algorithms often have high memory consumption. Larsson and Sadakane [26] have shown that it is often better to use the more complex algorithms which are non-linear in time. Some examples of algorithms quoted by Deorowicz are as follows:

- Manbers–Myer [28] has a worse-case time complexity $O(n \log n)$ and memory complexity $8n$.
- Bentley–Sedgwick [29] has a worst-case complexity $O(n^2)$, an average complexity $O(n \log n)$, and memory consumption $5n$ (plus a quicksort stack).
- Sadakane [22] describes an algorithm for producing a *suffix array*, as an alternative to a suffix tree. His measurements show a generation time which is generally the best or second best of the Manbers–Myer and Bentley–Sedgwick algorithms and an algorithm of Larsson [31].
- Larsson and Sadakane [26] describe an algorithm with worst-case complexity $O(n \log n)$ and memory complexity $8n$.

7.9 MOVE-TO-FRONT

The Move-To-Front step is usually regarded as an essential part of Burrows–Wheeler. That this is not so was demonstrated by Fenwick [3], who without much effort achieved a compression of 2.52 bpc on the Calgary Corpus in a Burrows–Wheeler compressor with no MTF stage.

Although not a good result by PPM and Burrows–Wheeler standards, it is nevertheless quite respectable by most compression standards.

Many authors have looked very suspiciously at the MTF recoding, realizing that by mixing the information from many different contexts it actually involves a considerable loss of possibly useful information. Despite this doubt, most have accepted the Move-To-Front recoding as necessary (a necessary evil?) and a great deal of work has been expended on improving the Move-To-Front step, with varying amounts of success. But more recently Wirth has applied PPM techniques and shown that excellent performance is possible with no MTF at all. We will defer that topic until later.

7.9.1 Move-To-Front Variants

Move-To-Front is one of the classical examples of a self-organizing list structure which dynamically restructures itself to facilitate access to recent or seemingly important data. It is the exact analog of the hierarchical caches found in modern computers, but providing a smooth transition instead of distinct levels. In comparison with many other self-organizing list structures (see Fenwick [32]), it is characterized by a very rapid movement of referent data to the preferred position at the front of the list. Many authors look with suspicion at this aspect and propose schemes with slower movement. Some examples developed for Burrows–Wheeler compression are as follows:

- **Move-One-From-Front (M1FF).** If the symbol σ is at rank 2 or higher, move it to position 1; otherwise move it to position 0 [18].
- **Move-One-From-Front-2 (M1FF2).** If σ is at rank 2 or higher and if the head symbol was requested last time or the time before, move it to position 1; otherwise move it to position 0.
- **Best x of $2x - 1$.** Put σ in front of all others that have been requested the minority of (fewer than x) times in the last $2x - 1$ requests of themselves and σ [33].
- **Best 2 of 3.** The previous method, with $x = 2$.

The author has developed another method described as "sticky MTF," which works very well on most files. In this method the symbol is brought to the front immediately, but if it is not used on the very next symbol it is moved back to about 40% of its original position. The method is unpublished, although it was communicated privately to Seward and used in some of the BZIP compressors.

To justify sticky MTF, note that for most text files there is a 60% probability that a symbol, having been moved to the list head, will be reused immediately. It therefore makes sense to move it right to the front because this is where it will be needed. If, however, it is not reused immediately, the symbol will "pollute" the MTF list until it shuffles back behind the active symbols. Each one of these more recent symbols will incur a cost from the presence of this unwanted symbol. We therefore move the unused symbol well out of the way, far enough to clear the MTF "working set" but not so far as its original position, because it is likely to be reused. A return to about 40% of its original rank is a good compromise.

Results for both the Calgary Corpus and the newer Canterbury Corpus are shown in Table 7.6. With one glaring exception, sticky MTF gives an improvement of about 1% across most files. That exception is the file `excl` from the Canterbury Corpus, a file of spreadsheet data. Examination of the MTF output from this file shows that while most of the MTF is very quiet with most values predicted very well, it has large bursts of very high entropy, higher than is usual in other files. Many symbols are referred to once and then again after a break of a few other intervening symbols. With "non-sticky" MTF those symbols remain near the head of the list and

Table 7.6 Results from Sticky MTF on Calgary and Canterbury Corpora

	Files from Calgary Corpus													
	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans	Average
Normal	1.943	2.390	2.037	4.482	2.497	3.865	2.456	2.452	2.411	0.779	2.492	1.708	1.699	1.491	2.336
Sticky	1.927	2.357	2.014	4.428	2.465	3.797	2.433	2.440	2.389	0.753	2.478	1.698	1.702	1.489	2.312
Reduction	0.8%	1.4%	1.1%	1.2%	1.3%	1.8%	0.9%	0.5%	0.9%	3.5%	0.6%	0.6%	−0.2%	0.1%	1.0%

	Files from small Canterbury Corpus											
	csrc	excl	fax	html	lisp	man	play	poem	sprc	tech	text	Average
Normal	2.081	0.971	0.779	2.436	2.536	3.108	2.508	2.390	2.608	1.993	2.249	2.151
Sticky	2.097	1.237	0.753	2.410	2.547	3.098	2.483	2.361	2.571	1.969	2.228	2.159
Reduction	−0.8%	−22%	3.5%	1.1%	−0.4%	0.3%	1.0%	1.2%	1.4%	1.2%	0.9%	−0.4%

are accessed with low cost. With sticky MTF they are well removed from the head and are much more expensive to recover. This file is a salutary lesson in the ability of special cases to defeat optimization!

Another technique which impinges on MTF is to restrict the alphabet. This has a very small one-off effect for each symbol as it may have to move a shorter initial distance when it becomes active. (Most text files use only three or four of the ASCII "control" symbols and packing these reduces the initial MTF movement by about 30 positions.) Reducing the alphabet often has a useful effect on the final statistical compressor as it means that many symbols can be omitted entirely and do not clutter the model with available but unused counts.

Some caution is needed though in transmitting the alphabet of active symbols, as sending information on the reduced alphabet may cost more than any gain. The author has used a system which sends a bit vector of active symbols. Explicit full-mode (>240 symbols) and half-mode ($120 < n < 128$) cases are signaled if appropriate to avoid the cost of a bit vector. It gives an improvement of about 0.05 bpc on the Calgary Corpus.

7.10 STATISTICAL COMPRESSOR

Early work by Fenwick [2] indicated that the output of the Burrows–Wheeler permutation is compressed very poorly by "good" compressors such as PPM. Wirth [36] interprets this negative result as due to the context mechanism becoming overwhelmed. Although much of the input context structure is absorbed by the permutation, there is still some context structure in the recoded MTF output. The problem is to interpret that structure, while keeping the number of contexts within reasonable bounds.

A major work on the statistical compressor is by Balkenhol *et al.* [18]. They note that the output from the MTF recoding may be loosely divided into "good" and "bad" fragments, with the goodness defined according to compressibility. The symbol probabilities change little within fragments and sometimes very little between "close" fragments with different true contexts. While the statistics of individual fragments vary widely, the number of fragments shows little variation. While each fragment corresponds to a context (in some approximate sense) the number of fragments grows quite slowly as the file size increases. While a large file has a few more fragments, these fragments are themselves much larger. They conclude that it is sensible to base contexts on the fragment structure using the current "noisiness" to determine the position with respect to fragments.

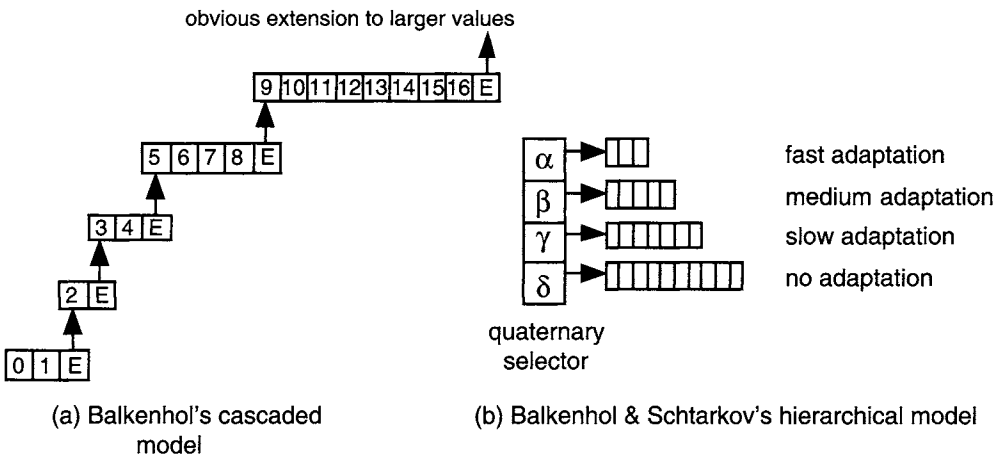

FIGURE 7.10
Balkenhol and Schtarkov's models.

Symbol	Code
0_a	0 0
0_b	0 1
1	1 0
2–7	1 1 0 b_2 b_1 b_0
8–15	1 1 1 0 b_2 b_1 b_0
16–31	1 1 1 1 0 b_3 b_2 b_1 b_0
32–63	1 1 1 1 1 0 b_4 b_3 b_2 b_1 b_0
64–127	1 1 1 1 1 1 0 b_5 b_4 b_3 b_2 b_1 b_0
127–255	1 1 1 1 1 1 1 b_6 b_5 b_4 b_3 b_2 b_1 b_0

FIGURE 7.11
Deorowicz' coding model.

They first used a cascaded model as shown in Fig. 7.10a. It is truncated to the actual maximum symbol so that the coder wastes no code space in allowing for symbols which will never occur. They then use an order-3 context, based on the rank r, according to $\{r = 0, r = 1, r > 1\}$, to determine whether the coding is within a good fragment, within a bad fragment, or between fragments. Their compressor achieves 2.30 bpc, at the time the best result from a Burrows–Wheeler compressor. In later work, Balkenhol and Schtarkov [27] use the hierarchical model shown in Fig. 7.10b. This has a small model with fast adaptation, two progressively larger ones with slower adaptation and a final constant or order(–1) model in the background. This model has been adopted by other workers as noted later.

Deorowicz [17] also examines the statistical coder, producing the coding model shown in Fig. 7.11, which combines aspects of both the cascaded and the structured coding models. The coding is done with binary encoders (each of the b_i represents a single bit), with the code having some resemblance to an Elias γ code [34].

Note that he includes run-length coding and therefore needs the two symbols 0_a and 0_b to encode the lengths.

Table 7.7 Recent Results on Calgary Corpus Files

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans
Fenwick (unpublished); average = 2.298 bpc													
1.926	2.356	2.012	4.268	2.464	3.765	2.433	2.439	2.387	0.753	2.476	1.697	1.702	1.488
Balkenhol *et al.* (1998); average = 2.30 bpc													
1.93	2.33	2.00	4.27	2.47	3.79	2.47	2.44	2.39	0.75	2.47	1.70	1.69	1.47
Deorowicz 2000; average = 2.271 bpc													
1.904	2.317	1.983	4.221	2.450	3.737	2.427	2.411	2.369	0.741	2.446	1.678	1.665	1.448

Code Bit	Number of Possible Contexts	Possible Contexts
first bit	6	last char = 0; length of 0-run ≤ 2
		last char = 0; length of 0-run > 2
		last char = 1; preceding char = 0
		last char = 1; preceding char $\neq 0$
		last char > 1; preceding char = 0
		last char > 1; preceding char $\neq 0$
second bit (first = 0)	$\log_2(n)$	length of 0-run is context
second bit (first = 1)	2	last char = 0 or last char = 1
successive code bits	254	all previous code bits form a context

FIGURE 7.12
Deorowicz' context generation.

But Deorowicz then treats the sequence of bits in a context c as a Markov chain of order d and uses a statistical estimator; the details of his context generator are shown in Fig. 7.12. The estimator is complex and the reader is referred to his paper for details.

These methods are among the mathematically most complex used in a Burrows–Wheeler compressor; the results are the best known for a Burrows–Wheeler compressor. Table 7.7 shows some of these results, together with those for Balkenhol *et al.* [18] and some unpublished results of the author (shown in the first row), which retained the structured coder but included a reduced alphabet and sticky MTF. It may be concluded that although improved encoders can give certainly better compression, the gain is difficult and expensive to achieve.

7.11 ELIMINATING MOVE-TO-FRONT

Arnavut and Magliveras [20] introduce *inversion frequency vectors* as an alternative to Move-To-Front recoding. We illustrate with the earlier "`mississippi`" example, which permutes to "`iiiimppssss`". First transmit the alphabet {`imps`} and then the symbol frequencies {4, 1, 2, 4}. Then send the positions of each symbol by encoding the gaps of available spaces between symbol occurrences. The symbol ordering is unstated, but Wirth [36] finds that symbols should be sent in decreasing frequency. Here we encode in the order {`ispm`}, breaking the tie on `s` and `p` by lexicographical ordering. The operations are then as shown in Fig. 7.13.

Action	Sym	Vector											
Prepare 11 symbols			X	X	X	X	X	X	X	X	X	X	X
Send vector for i	i	0000	i	i	i	i	X	X	X	X	X	X	X
Send vector for s	s	3000	i	i	i	i	X	X	X	s	s	s	s
Send vector for p	p	100	i	i	i	i	X	p	p	s	s	s	s
Fill remaining space with m	m		i	i	i	i	m	p	p	s	s	s	s

FIGURE 7.13
Inversion frequency vector coding.

1	Symbol Index	0	1	2	3	4	5	6	7	8	9	10
2	array	p	s	s	m	i	p	i	s	s	i	i
3	FC	i	i	i	i	m	p	p	s	s	s	s
4	CSPV	4	6	9	10	3	0	5	1	2	7	8
5	LBIV	0	0	0	0	4	5	3	6	6	2	2
6	decorrelated vector	0	0	0	0	4	5	2	6	0	4	0

FIGURE 7.14
Arnavut's inversion coding.

The successive numbers of the vectors indicate the number of *available* spaces to skip over before the current symbol is entered; consecutive symbols have 0's from the zero space. Note that the very last symbol (least frequent) can be inferred as it simply fills the still available spaces.

Arnavut later improved and modified his technique of inversion frequencies, with his system of "inversion coding" [30], illustrated in Fig. 7.14 for the string "`iiiimppssss`" (the Burrows–Wheeler permutation of "`mississippi`" used earlier). Despite the similarity in names, these are really two quite different approaches. His inversion frequencies were really a substitute for Move-To-Front coding, whereas inversion coding applies at a much later stage of the process sequence.

Figure 7.14 shows the following:

- Row 1 is an index of the symbols in the second row, placed here only for reference.
- Row 2 contains the permuted symbols from the Burrows–Wheeler transform.
- Row 3 "FC" (first symbol) has the symbols in order as prepared during the first stage of the decoding step. These symbols form the last symbol of the preceding context for the permuted symbols in row 2.
- Row 4 is Arnavut's *canonical sorting permutation vector*. Inspection shows that this is neither more nor less than the "link" column of Fig. 7.2, but now 0-origin rather than 1-origin. His coder goes partway into what is traditionally thought of as the *decoding* process.
- Row 5 he calls the *left bigger inversion vector*. The value at `LBIV[i]` is the number of values in `CSPV[0...i-1]` which exceed `CSPV[i]`.
- Row 6 is the "decorrelated" values of Row 5. The LBIV entries occur in chunks, one chunk for each symbol. Within each chunk, the values are non-increasing, which allows us to *decorrelate* the values by taking the difference between adjacent values of the same chunk. It is this decorrelation vector which is actually transmitted. The vertical lines here divide the sequence into blocks, one block for each symbol. The initial values in each block are

Table 7.8 Wirth's and Other Recent Results on Calgary Corpus Files

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans
Balkenhol *et al.* (1998); average = 2.30 bpc													
1.93	2.33	2.00	4.27	2.47	3.79	2.47	2.44	2.39	0.75	2.47	1.70	1.69	1.47
Deorowicz (2000); average = 2.271 bpc													
1.904	2.317	1.983	4.221	2.450	3.737	2.427	2.411	2.369	0.741	2.446	1.678	1.665	1.448
Wirth (2001); average = 2.349 bpc													
1.990	2.330	2.012	4.390	2.487	3.811	2.514	2.492	2.424	0.743	2.518	1.763	1.792	1.622

rare (only one for each symbol) and can be transmitted by almost any convenient manner without affecting the final compression.

Arnavut's technique therefore breaks the sequence

```
forward_permutation — encode — compress — decode
  — inverse_permutation
```

at a different point from what is usual, placing the statistical encode–decode step in the middle of the `inverse_permutation` rather than between `encode` and `decode`. Using Balkenhol's hierarchy as described in the previous section [18] Arnavut achieves a compression of 2.30 bpc, but with outstanding results on the large text files of the Canterbury Corpus.

In a quite different approach, Wirth (see Refs. [36, 37]) abandons the Move-To-Front operation completely and attacks the permuted data directly using techniques taken from PPM compressors. His compressor divides the permuted output into "segments" of like symbols and processes the segments with a compressor based on that of Balkenhol and Schtarkov, employing conditioning and exclusion as from PPM compressors. Run-lengths (the length of a segment) are encoded separately.

Wirth's results may be compared with those of other recent workers, copied from Fig. 7.8 earlier and combined in Table 7.8. These results demonstrate that Burrows–Wheeler compression without the Move-To-Front stage can certainly achieve good performance.

7.12 USING THE BURROWS–WHEELER TRANSFORM IN FILE SYNCHRONIZATION

Tridgell [35] designed his program `rsync` to synchronize widely separated files by transmitting only changes and differences. Such differences can be detected by checksumming blocks of the files, comparing checksums, and converging on areas of difference. An immediate problem arises with line breaks, which may be sometimes CR, sometimes LF, and sometimes a CR–LF pair, depending on the computer. Thus text blocks which are visually and functionally identical may indicate frequent differences and require a great deal of unnecessary synchronization. Tridgell solves this problem by applying a Burrows–Wheeler transform to both files and then comparing and synchronizing the permuted data. The permutation has the effect of sweeping the CR, LF, or other separators into a few isolated areas of the files, leaving the text areas largely unchanged. Differences due to line breaks are concentrated in a few blocks which are relatively inexpensive to reconcile. He quotes an example where applying the Burrows–Wheeler transform gives a speed-up (effectively a reduction of synchronization traffic) of 27 times (see p. 78, Table 4.5, of [35]).

7.13 FINAL COMMENTS

An enormous amount of work has been done on Burrows–Wheeler compression in the short time since it was introduced, perhaps more than for any other compression technique in a comparable interval. It is new, it is different, it is conceptually simple, and with relatively little effort it gives excellent results. The initial observation by Burrows and Wheeler that "it gives PPM compression with Ziv–Lempel speed" is certainly justified. (Compare this with Effros' claim, just 6 years later, to combine "PPM compression with BWT complexity"; Burrows–Wheeler compression has certainly arrived!)

Despite all of this work, however, the author feels that Burrows–Wheeler compression is still not *really* understood. The suspicion remains that the Move-To-Front stage is an embarrassing and unnecessary complication, as supported by the work by Arnavut and Wirth. There is undoubtedly some sort of context information in the BWT output, which has been used with some difficulty by Balkenhol *et al.* [18], Deorowicz [17], and Wirth [36]; but is there more, and do we have to find it in some other way?

The improvements so far seem to be largely those of chipping away at a monolithic structure in the hope of getting occasional improvement. While some of the work is solid research with a good theoretical grounding, much other work (and the author includes some of his own in this category!) is little better than optimistic "hacking," some of which works. The difficulty of improvement is seen in that Fenwick obtained a Calgary Corpus result of 2.52 bpc with a trivial single-model coder and improved this with little trouble to 2.34 bpc (a 7% improvement). Later work has improved that to about 2.27 bpc, a further improvement of 3%. In practical terms this change is negligible.

But this pessimistic tone is hardly justified as an end to the chapter. Burrows–Wheeler compression *does* work. It is competitive in compression with all but the very best statistical compressors. And it achieves this compression with relatively few memory and processor resources, far less than most top-of-the-line PPM methods. The pessimism of the previous paragraph should perhaps be reinterpreted as a call for yet more research on a challenging topic. In summary the algorithm works and it works well, with prospects for future improvement.

7.14 RECENT DEVELOPMENTS

This section presents some Burrows–Wheeler results which became available between the preparation of the chapter and its going to press around July 2002. Interestingly, the four papers represent different approaches, two examining the recoding stage and two the final statistical coder. Quite clearly, the topic is far from worked out. Results for the Calgary Corpus are shown in Table 7.9.

Arnavut [38] continues his development of inversion ranks as an alternative to MTF recoding. With this paper he achieves compression comparable with the best MTF coders, showing that inversion ranks are a viable option.

Balkenhol and Shtarkov [27] concentrate on the final coder, improving their previous context models.

Deorowicz [39] makes further modifications to the MTF recoder, using a "weighted frequency count" to control movement in the ordered list of symbols. This paper reviews other recent alternatives to MTF coding.

Fenwick [40] simplifies the final coder by using "variable-length integers" or "universal codes" instead of the conventional arithmetic or dynamic Huffman coders. Even with this extreme simplification he still obtains compression within about 15% of the best results.

Table 7.9 Recent Burrows–Wheeler Compression Results

File	Arnavut BSIC [38]	Balkenhol BS99 [27]	Deorowicz WFC [39]	Fenwick [40]
bib	1.96	1.91	1.90	2.15
book1	2.22	2.27	2.27	2.81
book2	1.95	1.96	1.96	2.31
geo	4.18	4.16	4.15	5.27
news	2.45	2.42	2.41	2.71
obj1	3.88	3.73	3.70	4.05
obj2	2.54	2.45	2.41	2.57
paper1	2.45	2.41	2.40	2.68
paper2	2.36	2.36	2.35	2.70
pic	0.70	0.72	0.72	0.83
progc	2.50	2.45	2.43	2.68
progl	1.74	1.68	1.67	1.84
progp	1.74	1.68	1.67	1.83
trans	1.55	1.46	1.45	1.58
Average	2.30	2.26	2.25	2.57

7.15 REFERENCES

1. Burrows, M., and D. J. Wheeler, 1994. A Block-Sorting Lossless Data Compression Algorithm, SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA. Available at `gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z`.
2. Fenwick, P. M., 1995. Experiments with a Block Sorting Text Compression Algorithm, The University of Auckland, Department of Computer Science Report No. 111, May 1995.
3. Fenwick, P. M., 1995. Improvements to the Block Sorting Text Compression Algorithm, The University of Auckland, Department of Computer Science Report No. 120, August 1995.
4. Fenwick, P. M., 1996. Block Sorting Text Compression—Final Report, The University of Auckland, Department of Computer Science Report No. 130, April 1996.
5. Fenwick, P. M., 1996. Block Sorting Text Compression. In *ACSC-96 Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 1996.* pp. 193–202. Available at `ftp.cs.auckland.ac.nz`.
6. Fenwick, P. M., 1996. The Burrows–Wheeler transform for block sorting text compression—Principles and improvements. *The Computer Journal*, Vol. 39, No. 9, pp. 731–740.
7. Seward, J., 1996. Private communication. For notes on the released BZIP see `http://hpux.cae.wisc.edu/man/Sysadmin/bzip-0.21`.
8. Bell, T. C., J. G. Cleary, and I. H. Witten, 1990. *Text Compression*, Prentice Hall, Englewood Cliffs, NJ. The Calgary Corpus can be found at `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`.
9. Arnold, R., and T. Bell, 1999. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the IEEE Data Compression Conference*, March 1999, pp. 201–210. IEEE Comput. Soc., Los Alamitos, CA. The Canterbury corpus can be found at `http://corpus.canterbury.ac.nz`.
10. Wheeler, D. J., 1995. Private communication. October 1995. [This result was also posted to the `comp.compression.research` newsgroup. Available by anonymous FTP from `ftp.cl.cam.ac.uk/users/ djw3`].
11. Bentley, J. L., D. D. Sleator, R. E. Tarjan, and V. K. Wei, 1986. A locally adaptive data compression algorithm. *Communications of the ACM*, Vol. 29, No. 4, pp. 320–330, April 1986.

12. Witten, I., R. Neal, and J. Cleary, 1987. Arithmetic coding for data compression. *Communications of the ACM*, Vol. 30, pp. 520–540.
13. Fenwick, P. M., 1996. A new data structure for cumulative frequency tables: An improved frequency-to-symbol algorithm. *Software—Practice and Experience*, Vol. 26, No. 4, pp. 399–400, April 1996.
14. Moffat, A., R. M. Neal, and I. H. Witten, 1998. Arithmetic coding revisited. *ACM Transactions on Information Systems*, Vol. 16, No. 3, pp. 256–294, July 1998. Source software available from `http://www.cs.mu.OZ.AU/~alistair/arith_coder/`.
15. Cleary, J. G., and W. J. Teahan, 1997. Unbounded length contexts for PPM. *The Computer Journal*, Vol. 40, No. 2/3, pp. 67–75.
16. Effros, M., 2000. PPM performance with BWT complexity: A fast and effective data compression algorithm. *Proceedings of the IEEE*, Vol. 88, No. 11, pp. 1703–1712, November 2000.
17. Deorowicz, S., 2000. Improvements to Burrows–Wheeler compression algorithm. *Software—Practice and Experience*, Vol. 30, No. 13, pp. 1465–1483, November 2000.
18. Balkenhol, B., S. Kurtz, and Y. M. Shtarkov, 1999. Modifications of the Burrows and Wheeler data compression algorithm. *Proceedings of the IEEE Data Compression Conference*, March 1999, pp. 188–197. IEEE Comput. Soc., Los Alamitos, CA.
19. Chapin, B., and S. R. Tate, 1998. Higher compression from the Burrows–Wheeler transform by modified sorting. *Proceedings of the IEEE Data Compression Conference*, March 1998, p. 532 .IEEE Comput. Soc., Los Alamitos, CA. See also `http://www/cs.unt.edu/home/srt/papers`.
20. Arnavut, Z., and S. S. Magliveras, 1997. Lexical permutation sorting algorithm. *The Computer Journal*, Vol. 40, No. 5, pp. 292–295.
21. Schindler, M., 1997. A fast block sorting algorithm for lossless data compression. *IEEE Data Compression Conference*, March 1997, p. 469. IEEE Comput. Soc., Los Alamitos, CA.
22. Sadakane, K., 1998. A fast algorithm for making suffix arrays and for Burrows–Wheeler transformation. In *IEEE Data Compression Conference*, March 1998, pp. 129–138. IEEE Comput. Soc., Los Alamitos, CA.
23. Seward, J., 2000. On the performance of BWT sorting algorithms. In *Proceedings of the IEEE Data Compression Conference*, March 2000, pp. 173–182. IEEE Comput. Soc., Los Alamitos, CA.
24. Shannon, C. E., 1951. Prediction and entropy of printed English. *Bell Systems Technical Journal*, Vol. 30, pp. 50–64.
25. McCreight, E. M., 1976. A space economical suffix-tree construction algorithm. *Journal of the ACM*, Vol. 23, pp. 262–272, April 1976.
26. Larsson, N. J., and K. Sadakane, 1999. Faster suffix sorting. Technical Report, LU-CS-TR:99-214, *Department of Computer Science, Lund University, Sweden.*
27. Balkenhol, B., and Y. M. Schtarkov, 1999. One attempt of a compression algorithm using the BWT. *Technical Report 99-133, Sonderforschungsbereich: Disktrete Strukturen in der Mathematik, Universität Bielefeld, Germany.*
28. Manber, U., and E. W. Myers, 1993. Suffix arrays: A new method for online string searches. *SIAM Journal on Computing*, Vol. 22, No. 5, pp. 935–948.
29. Bentley, J. L., and R. Sedgwick, 1997. Fast algorithms for sorting and searching strings. *Proceedings of the 8th Annual ACM–SIAM Symposium of Discrete Algorithms*, pp. 360–369.
30. Arnavut, Z., 2000. Move-to-front and inversion coding. In *Proceedings of the IEEE Data Compression Conference*, March 2000, pp. 193–202. IEEE Comput. Soc., Los Alamitos, CA.
31. Larsson, N. J., 1996. Extended application of suffix trees to data compression. In *Proceedings of the IEEE Data Compression Conference*, March 1996, pp. 129–138. IEEE Comput. Soc., Los Alamitos, CA.
32. Fenwick, P. M., 1991. A new technique for self-organizing linear searches. *The Computer Journal*, Vol. 34, No. 5, pp. 450–454.
33. Chapin, B., 2000. Switching between two list update algorithms for higher compression of Burrows–Wheeler transformed data. *Proceedings of the IEEE Data Compression Conference*, March 2000, pp. 183–192. IEEE Comput. Soc., Los Alamitos, CA.
34. Elias, P., 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, Vol. IT-21, No. 2, pp. 194–203, March 1975.

35. Tridgell, A., 1999. *Efficient Algorithms for Sorting and Synchronization*, Ph.D. thesis, Department of Computer Science, Australian National University, Canberra, Australia.
36. Wirth, I. W., 2000. *Symbol Driven Compression of Burrows Wheeler Transformed Text*, M.Sc. thesis, Department of Computer Science, Melbourne University, Melbourne, Australia.
37. Wirth, I. W., and A. Moffat, Can we do without ranks in Burrows–Wheeler transform compression?" In *Proceedings of the IEEE Data Compression Conference*, March 2001, IEEE Comput. Soc., Los Alamitos, CA.
38. Arnavut, A., 2002. Generalization of the BWT transformation and inversion ranks. *Proceedings of the IEEE Data Compression Conference*, March 2002, pp. 477–486. IEEE Comput. Soc., Los Alamitos, CA.
39. Deorowicz, S., 2002. Second step algorithms in the Burrows–Wheeler compression algorithm. *Software—Practice and Experience*, Vol. 32, No. 2, pp. 99–111.
40. Fenwick, P., 2002. Burrows Wheeler compression with variable-length integer codes. *Software—Practice and Experience*, Vol. 32, No. 13, November 2002.