The VLDB Journal (2007) 16(2): 269–291
DOI 10.1007/s00778-006-0005-2

ORIGINAL ARTICLE

John Yiannis · Justin Zobel

Compression techniques for fast external sorting

Received: 22 June 2004 / Accepted: 8 December 2004 / Published online: 12 July 2006

Abstract External sorting of large files of records involves use of disk space to store temporary files, processing time for sorting, and transfer time between CPU, cache, memory, and disk. Compression can reduce disk and transfer costs, and, in the case of external sorts, cut merge costs by reducing the number of runs. It is therefore plausible that overall costs of external sorting could be reduced through use of compression.

In this paper, we propose new compression techniques for data consisting of sets of records. The best of these techniques, based on building a trie of variable-length common strings, provides fast compression and decompression and allows random access to individual records. We show experimentally that our trie-based compression leads to significant reduction in sorting costs; that is, it is faster to compress the data, sort it, and then decompress it than to sort the uncompressed data. While the degree of compression is not quite as great as can be obtained with adaptive techniques such as Lempel-Ziv methods, these cannot be applied to sorting. Our experiments show that, in comparison to approaches such as Huffman coding of fixed-length substrings, our novel trie-based method is faster and provides greater size reductions.

Keywords External sorting · Semi-static compression · Query evaluation · Sorting

1 Introduction

Relational database systems, and more recent developments such as document management systems and object-oriented database systems, are used to manage the data held by virtually every organisation. Typical relational database systems contain vast quantities of data, and each table in a database may be queried by thousands of users simultaneously. However, the increasing capacity of disks means that more data can be stored, escalating query-evaluation costs. Each stage of the entire storage hierarchy of disk, controller caches, memory, and processor caches can be a bottleneck. Processors are not keeping pace with growth in data volumes [43], particularly for tasks such as joins and sorts where the costs are superlinear in the volume of data to be processed.

In this paper we propose the use of compression of data to reduce the costs of external sorting, thus making better use of the storage hierarchy. A current problem is that tens to hundreds of processor cycles are required for a memory access, and tens of millions for a disk access, a trend that is continuing: processor speeds are increasing at a much faster rate than that of memory and disk technology [5]. During an external sort, total processing time is only a tiny fraction of elapsed time. Most of the time is spent writing sorted runs to disk, then reading and merging the runs. This imbalance can partly be redressed through use of compression.

For external sorting, it should in principle be possible to use spare cycles to compress the data on the fly, thus reducing the number of runs. We propose that compression proceed by allowing pre-inspection of the first buffer-load of data, and building a model based on this data alone. This partial (and probably non-optimal) model can then be used to guide compression and decompression of each subsequent run. It can also reduce the temporary space required to store the runs.

However, a compression technique for this application must meet strong constraints. First, in contrast to adaptive compression techniques, which treat the data as a continuous stream and change the codes as the data is processed, it must allow the records to be accessed individually and reordered. Second, in contrast to standard semi-static techniques, the data cannot be fully pre-inspected to determine a model. Third, the coding and decoding stages must be of similar speed to the transfer rate for uncompressed data. Last, the compression model must be small, so that it does not consume too much of the buffer space, which is needed for sorting.

Preliminary versions of parts of this paper, not including the work on vargram compression" [41]

J. Yiannis (✉) · J. Zobel
School of Computer Science and Information Technology,
RMIT University, Melbourne 3000, Australia
E-mail: jyiannis@cs.rmit.edu.au

None of the commonly used compression methods meets these constraints. Huffman coding is a possibility, although model construction can be slow and, for tokens of sufficient length to give good compression, model size is a potential problem. As our experiments show, it is unsatisfactory. Nonetheless, data compression can be an effective means of increasing bandwidth – not by increasing physical transfer rates, but by increasing the information density of transferred data—and can relieve the I/O bottlenecks found in many high-performance database management systems [13].

We have therefore developed several new compression techniques designed to solve the problem of rapidly compressing records while allowing random access. The first two are based on the simple heuristic of extracting common fixed-length sequences of letters and allocating them bytewise codes. (Bitwise codes offer slightly greater compression, but are considerably more expensive to process.) These succeed in reducing the cost of sorting large files, but further gains would be desirable.

For this reason, we have investigated novel compression techniques based on constructing in-memory tries of common strings, inspired by our previous success with use of tries as a fast space-efficient data structure [15] and for internal sorting [34, 35]. We show experimentally that using bytewise codes to represent common variable-length strings can greatly reduce the space required to store the runs. For smaller files, the time required to compress and decompress outweighs the savings, but, for our largest file, use of compression reduced sorting time by a third. With current trends in computer hardware, the gains due to compression are expected to increase.

Previous research [7, 12, 14, 23, 29, 37] has shown the benefits of decompressing data on the fly where the data is stored compressed. However, it was found [37] that compression on the fly had significantly higher processor costs, indicating that compression is only beneficial to read-only queries. Our results show, in contrast to previous work, that compression is useful even when the data is stored uncompressed.

2 External sorting

External sorting is used when data to be sorted does not fit into available memory. It can be used for sorting any large file, but is of particular value in the context of databases, where a machine may be shared amongst a large number of users and queries, and per-query buffer space is limited. In the context of databases we handle records in units of blocks. Vitter [36] surveys algorithms and data structures for large applications that do not fit entirely in memory. Sorting can be used as the basis for several of the relational-algebra operations. Garcia-Molina et al. [11] outline algorithms for duplicate elimination, grouping and aggregation using sorting, as well as sort-based union, intersection, and difference.

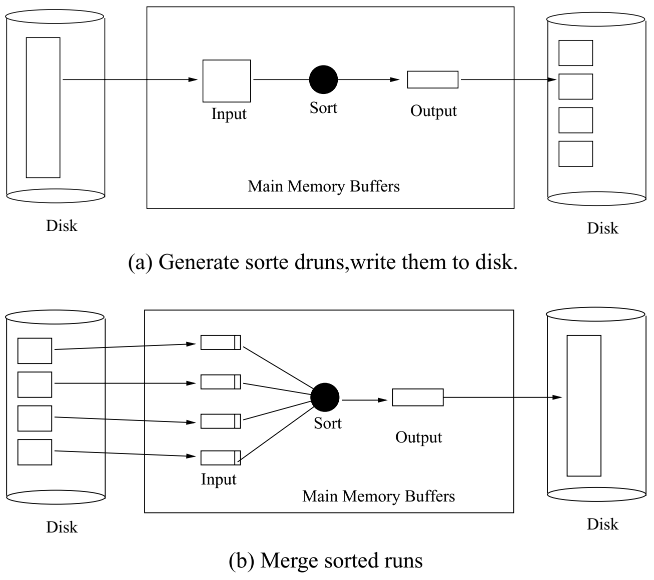


Fig. 1 A simple external merge sort, with sorted blocks of records written to intermediate runs that are then merged to give the final result

The external merge sort has two phases [11, 27]. The process assumes that a fixed-size buffer is available for sorting in the first phase and for merging in the second. The external merge sort process is illustrated in Fig. 1, and is described in detail by Knuth [16]. In detail, external sorting proceeds as follows. Assume that M is the number of blocks in the buffer space to be used to hold the input and any intermediate results, $B(R)$ denotes the number of blocks that are required to hold all the tuples of relation R, and $T(R)$ is the number of tuples in relation R. If we have a large relation R, where $B(R) > M$, a two-pass algorithm for the external merge sort is as follows:

Two-pass external merge sort:

Build runs: Process buffer-sized amounts of data in turn to produce a sorted list of records that just fits in the available memory, yielding several sorted sublists or *runs*, which are merged in the next phase. The following is repeated until all input is consumed:

1. Fill the buffer with records from the relation to be sorted (that is, read M blocks of R into memory).
2. Sort the records in memory using an efficient sorting algorithm.
3. Write records in sort order into new blocks, forming one run of M blocks on disk.

Merge runs: Merge the N runs into a single sorted list. The following is repeated until all runs have been processed:

1. Divide input buffer space amongst the runs, giving per-run buffers of size M/N blocks, and fill these with blocks from each of the runs.
2. Using a heap, find the smallest key among the first remaining records in each buffer, then move the corresponding record to the first available position of the output buffer.

3. If the output buffer is full, write it to disk and empty the output buffer.
4. If the input buffer from which the smallest key was just taken is now exhausted, fill the input buffer from the same run. If no blocks remain in the run, then leave the buffer empty and do not consider keys from that run in any further sorting.

The number of disk I/Os used by this algorithm, ignoring the handling of the output is $B(R)$ reads to fetch each block of R when creating the runs, $B(R)$ writes to store each block of the runs on disk, and $B(R)$ reads to fetch each block from the runs at the appropriate time when merging. Thus the total cost of this algorithm is $3B(R)$.

Assume we have a relation R consisting of 10,000,000 tuples. Each tuple is represented by a record with several fields, one of which is the sort key. For simplicity assume that the records are of a fixed length, say 100 bytes. Also, assume that the block size is 8 Kb, and we have 64 MB of main memory available as buffer space for the algorithm. We can fit eighty 100-byte records in each block. The relation thus occupies 125,000 blocks. The number of blocks that can fit in 64 MB of memory is 8192. In the first phase of the external merge sort we fill memory 16 times, sort the records in memory, and write the sorted sublists to disk. At the conclusion of this phase we have 16 runs on disk. We read each of the 125,000 blocks once and we write out 125,000 new blocks, resulting in 250,000 disk I/Os. If each access takes 1 ms, the I/O time for the first phase is 250 s or 4.2 min. In the second phase of the external merge sort we read 125,000 blocks (if we include the output, we also write 125,000 blocks). The I/O time for the second phase is half of the first or 125 s (if we include the time to write the result, the time is the same or 250 s). Both the read phase and the write phase are more or less ordered and will make good use of disk cache.

There are many variants on this algorithm. One variant is that, if the merged results are to be written to disk, they can be written in-place; in the context of database query processing, however, it is often the case that the results are immediately used and discarded. Another is that, with a large number of runs, there can be housekeeping problems for the operating system, and the per-run buffers may become too small. A solution is to merge the runs hierarchically, so that each pass reads in all the data from disk and writes it out again. Hierarchical merge incurs significant penalties in data transfer, and should be avoided. The Unix command-line utility `sort` takes this approach.

We do not test hierarchical merging in our experiments. Even for large relations, two passes are usually sufficient. Consider the following example. Suppose that we have a block size of b, memory available for buffering of m bytes, and records of size r bytes. The number of buffers available in memory is thus m/b. In the second phase each buffer can be devoted to one of the runs. Thus the number of runs that can be created in the first phase is m/b. Each time memory is filled to produce a run, we can sort m/r records. Thus the total number of records that we can sort is $(m/r)(m/b)$ or m^2/rb records. If we use the parameters from the example above, then $m =$67,108,864, $b = 8192$, and $r = 100$. We can thus sort up to $m^2/rb = 5.5$ billion records, occupying 0.5 terabytes of storage. Given the current limits on secondary storage, a two-phase merge sort is likely to be sufficient for most practical purposes.

Many sorting techniques are based on the assumption that memory access costs are homogeneous. However, the costs of sorting can be greatly reduced if more realistic models of memory are used. As we discuss later, even internal sorting can be accelerated through appropriate use of the memory hierarchy. At the lowest level – that is, closest to the processor – are the processor registers and caches that consist of the fastest but most expensive memory. At the next level, internal memory, is dynamic random access memory, which is slower but less expensive. Inexpensive disks are used for external mass storage. In database systems, the amount of data to be stored is typically large, and it would be prohibitively expensive to store the data entirely in memory; and, as disk volumes have increased, so have the volumes of data that have to be managed. Data is also stored on disk for reasons of permanency [36].

Each level of the memory hierarchy acts as a cache for the next. When data to be referenced is not cached, we incur the additional cost of fetching the data from a higher level of storage. Each level has its own cost and performance benefits. The time for accessing a level increases for each new level, with the largest bottleneck being between memory and disk. The main constituents of disk access costs are seek time, rotational delay, and transfer time. To amortize these delays, large contiguous amounts of data or *blocks* are transferred at a time. Aspects that determine memory access cost include: latency and address translation.

When dealing with large volumes of data, all levels of the memory hierarchy must be used. On typical hardware, loading a register takes on the order of a quarter of a nanosecond (10^{-9} s), and accessing memory takes tens of nanoseconds. Accessing data from a disk requires several milliseconds (10^{-3} s), which is several orders of magnitude slower [36]. For applications that must process large amounts of data, the I/O costs, that is, the times to transfer data between levels of storage, are a bottleneck of increasing significance. Processor speeds have been doubling every 18 months, and processors have additionally become faster through internal parallelism. The speed of commercial microprocessors has increased about 70% every year, while the speed of random-access memory has improved by a little more than 50% over the past decade, and the speed of access to disk has increased by only 7% a year. That is, processor speeds are increasing at a much faster rate than that of memory and disk technologies [5, 19]. These issues are particularly acute for applications such as external sorting.

2.1 Cache-aware sorting

With the speed of processors increasing at a greater rate than that of both disk and memory, algorithms and data structures must be cache-aware, and tuned to the memory access pattern imposed by a query and hardware characteristics such as cache size [18, 19, 24, 34]. An approach analogous to that for reducing disk I/O can be applied to in-memory processing, where the aim is then to reduce memory access costs such as cache misses.

The external merge sort algorithm above reduces disk I/O by sorting memory buffer-sized loads of data at a time. A similar approach can be used to reduce memory I/O. For an in-memory sort, the R-MERGE algorithm of Wickremesinghe et al. [38] sorts cache-sized loads of data with quicksort, which are then merged (as in an external merge sort). The merge heap is accessed multiple times for every key, and is relatively small. This allows the heap to be stored in registers, which reduces the cost to access it, and eliminates the need to load heap elements from memory. Another advantage of limiting the merge order is that memory accesses tend to be in the set of pages cached in the translation lookaside buffer.

Nyberg et al. [24] describes a cache-aware sort algorithm, AlphaSort, which uses clustered data structures to get good cache locality, file striping to get high disk bandwidth, quicksort to generate runs, and replacement selection to merge the runs. While loading all records into memory, AlphaSort sorts small batches of records to create mini-runs. When memory is full AlphaSort merges all mini-runs in a single, wide merge using a selection tree in the same way as for replacement selection. To reduce the disk I/O bottleneck, disk striping is used to spread the input and output file across several disks. This allows parallel disk reads and writes.

Larson [18] introduces batched replacement selection, which is a cache-aware version of replacement selection that works for variable length records. The algorithm resembles AlphaSort in that it creates small in-memory runs and merges these mini-runs to produce the final runs. The creation and merging of mini-runs in memory, and the processing of both input and output in batches rather than single records in the replacement selection algorithm reduces the number of cache misses.

In the burstsort algorithm of Sinha and Zobel [34], data structures are again chosen to minimise accesses to memory and so make better use of the cache. In burstsort, a trie is dynamically constructed as strings are sorted, and is used to allocate a string to a bucket. Like MSD radix-sorts, the leading character of each string is inspected only once, however, the pattern of memory accesses better utilises the cache. In an MSD radix-sort, before the bucket-sorting phase, each string is repeatedly accessed, once for each character. For a large collection of strings it is likely that the string will no longer be in the cache the next time it is accessed. In contrast, each string in burstsort is accessed once only, while the trie nodes are accessed repeatedly. Burstsort is currently the fastest sorting algorithm for strings or integers [33].

The run-generation phase of large external sorts with compression should benefit from these faster in-memory sorts. The overall external sort time will be dominated even more by disk I/O and merge costs as the time for in-memory sorting is reduced. It is these costs that we aim to reduce by the use of compression. The relative difference in speed between level-one cache and level-two cache, or between level-two cache and memory, is orders of magnitude smaller than that between memory and disk, so our focus is to use compression to reduce the more significant disk I/O bottleneck.

3 Database compression techniques

The cost of I/O can be reduced through the use of compression. That is, it is possible to trade off the increase in processor overhead in compressing and decompressing data on the fly against reduced I/O costs by transferring compressed data. Data compression is therefore an effective means of increasing bandwidth – not by increasing physical transfer rates, but by increasing the information density of transferred data – and can relieve the I/O bottlenecks found in many high-performance database management systems [13]. In other work [31], we have shown that even the cost of transferring data from internal memory to the on-processor cache can be reduced through appropriate use of compression.

In operations where intermediate results must be written to disk, data can first be compressed, reducing volumes of data to be transferred. If the overhead of compressing and decompressing data combined with the I/O cost of transferring the compressed representation is smaller than the I/O cost of transferring the uncompressed data, we can reduce the overall query time. Compression may also lead to other benefits in addition to reduced I/O costs. For example, in external sorting, smaller records due to compression may result in the generation of fewer runs, thus reducing the merge cost of the operation. In hash partitioned joins, compression of records will result in smaller partitions, and these smaller partitions are more likely to fit entirely in memory (compared to uncompressed partitions) and hence will not require repartitioning.

The value of compression in communications is well-known: it reduces the cost of transmitting a stream of data through limited-bandwidth channels. Much of the research into compression has focused on this environment, in which the order of the data does not change and pre-inspection of the data is not necessarily available, leading to the development of high-performance *adaptive* techniques. Compression depends on the presence of a *model* that describes the data and guides the coding process; in adaptive compression, the model is changed with each symbol encountered. Compression is achieved by using short codes for highly probable symbols, and longer codes for rarer symbols.

Compression is not straightforward in the context of databases. There are several constraints on compression when used in the context of reducing database query evaluation costs, particularly if compression and decompression need to be performed on-the-fly:

- To allow random access and atomic compression and decompression of records, modeling schemes must be *semi-static*. Adaptive techniques are largely inapplicable to the database environment, in which the stored data is typically a bag of independent records that can be retrieved or manipulated in any order. In such applications, the only option is to use semi-static compression, in which the model is fixed after some training on the data to be compressed, so that the code allocated to a symbol does not change during the compression process. Adaptation can be used while a record is being compressed [21], but at the start of the next record it is necessary to revert to the original model.
- It can be impractical to inspect an entire relation to build a model during query evaluation. To reduce processor and I/O overhead, only a sample of the data, not the entire database, is inspected to determine symbol frequencies. This can lead to sub-optimal compression, as inspecting the entire database can produce a more accurate model of the data and hence give better compression effectiveness.
- Compression of individual records or attributes allows random access to data, but is less effective than compressing entire files.
- The presence of a compression model reduces the buffer space available to evaluate the query.
- Even with the increasing speed of processors, it has been shown that existing compression schemes can lead to an increase in query times [37]. That is, coding and decoding be fast so as not to eliminate the benefits of reduced data transfer times.
- Database systems typically have large numbers of users running multiple queries, so buffer space per query is limited. To reduce memory usage, we must limit the model size, restricting the compression effectiveness of the compression technique, as the compression model is likely to be only a tiny fraction the volume of data to be compressed, a situation that is likely to lead to poor compression. This too yields sub-optimal compression; larger models would allow a better model of the data and hence increase compression effectiveness. For example, in algorithms such as Lempel-Ziv coding, the entire text to be compressed is used as a random-access model.

The best-known semi-static compression technique is zero-order frequency modeling coupled with canonical Huffman coding, in which the frequency of each symbol (which might be a byte, Unicode character, character-pair, English word, or any other such token) is counted, then a Huffman code is allocated based on the frequency. In canonical Huffman coding, the tree is not stored and decompression is much faster than traditional implementations [17, 40].

Semi-static compression has been successfully integrated into text information retrieval systems, resulting in savings in both space requirements and query evaluation costs [2, 31, 39, 40, 42]. The compression techniques used are relatively simple – Huffman coding for text, and integer coding techniques [39] for indexes – but the savings are dramatic. Index compression in particular is widely used in commercial systems ranging from search engines such as Google to content managers such as TeraText. Moreover, integer coding is extremely fast. As mentioned above, in other work [31] we found that even the cost of transferring from memory to processor cache can be reduced through appropriate use of compression based on elementary bytewise codes.

However, compression has traditionally not been used in commercial database systems [7, 23], and data compression has been undervalued in database query processing research [13]. Earlier papers investigated the benefits of compression in database query evaluation theoretically [14, 23, 29], and only in the last few years have researchers reported compression being incorporated into database systems [7, 12, 37]. An exception is text retrieval systems, where compression has been widely used in indexing [40].

Most of the research in database compression has focused on reducing storage and query processing costs when data is held compressed. Graefe and Shapiro [14] recommend compressing attributes individually, employing the same compression scheme for all attributes of a domain. Ng and Ravishankar [23] describe a page-level compression scheme based on a lossless vector quantisation technique. However, this scheme is only applicable to discrete finite domains where the attribute values are known in advance and the cardinality of each domain is low. Ray et al. [29] compared several coding techniques (Huffman, arithmetic, Lempel-Ziv, and run-length) at varying granularity (file, page, record, and attribute). They confirm the intuition that attribute-level compression gives poorer compression, but allows random access.

Goldstein et al. [12] described a page-level compression algorithm that allows decompression at the field level. However, like the scheme described by Ng and Ravishankar [23], this technique is only useful for records with low-cardinality fields. Westman et al. [37] used compression at the attribute level. For numeric fields they used null suppression and encoding of the resulting length of the compressed integer [30]. For strings they used a simple variant of dictionary-based compression. This is particularly effective if a field can only take a limited number of values. For example, a field that can only take the values "male" and "female" could be represented by a single bit which could then be used to look up the decompressed value of the field in the dictionary. They saw a reduction in query times for read-only queries, but significant performance penalties for insert and modify operations. Chen et al. [7] used the same scheme as Westman et al. [37] for numerical attributes, and developed a new hierarchical semi-static dictionary-based encoding scheme for strings. They also developed a number of compression-aware query optimization algorithms. Their results for read-only queries showed a substantial improvement in query performance over existing techniques. A consensus from this work is that, for efficient query processing, the compression granularity should be small, allowing random access to the required data and thereby minimising unnecessary decompression of data; and the compression scheme should be lightweight, that is, have low processor costs, so as not to eliminate the benefits of reduced data transfer times

When examining the benefits of compression, Westman et al. [37] saw that compression of a tuple had significantly higher processor costs than decompression, and so did not believe that compression could improve the performance of online transaction processing (OLTP) applications. All the other papers presupposed a compressed database, so the only compression-related cost involved in query resolution was the decompression of data.

Note that, for query processing, compression has value in addition to improved I/O performance, because decompression can often be delayed until a relatively small data set is determined. Exact-match comparisons can be on compressed data. During sorting, the number of records in memory and thus per run is larger, leading to fewer runs and possibly fewer merge levels [13].

Given the constraints outlined earlier, off-the-shelf compression systems are not useful for external sorting, as also noted by Witten et al. [40]. Arithmetic coding, Lempel-Ziv, and PPM methods do not allow random access to data; thus we cannot use, for example, `gzip` or `bzip2`, where decompression must start at the beginning of the data. For efficient query processing, compression must allow random access to data, either individual records or fields, and so the compression granularity must be small. In an external sort, individual records need to be sorted on a key, and written to different runs. In hash-partitioned joins, records are spread among several different partition files. These compression techniques could be used to compress individual records or attributes, however Ray et al. [29] show that, as the unit of compression decreases, adaptive schemes progressively produce poorer compression relative to non-adaptive schemes.

In our work, we aim to create new efficient decompression and compression methods for data stored in general purpose database systems. Thus the database can be left uncompressed, with any necessary compression and decompression performed on the fly during query execution. This approach eliminates the necessity of maintaining a model for the relations, as is necessary in previous methods. The model must be stored with the relations, and will become out of date as data is modified in the database. By compressing on-the-fly, a model does not need to be explicitly stored and is always up to date. This also allows both read and write queries to be evaluated on the data.

Several observations can be made. We need to investigate which semi-static coding techniques can be used in conjunction with a model based on inspection of only part of the data. If we were to read the entire relation to build a compression model, and read it again to perform the query, any possible gains through the use of compression would be lost. (Moreover, if the data to be sorted is the output of a subquery, it is not available for pre-inspection.) To avoid any additional I/O cost, in all our methods discussed below we build a model based solely on the data contained in the first buffer load of data we read in for the database operation. Also, if some symbol does not occur in this part of the data, it is nonetheless necessary that it have a code.

Bitwise or bytewise codes are much faster than arithmetic coding [40], which is too slow for this application. Bytewise codes are much faster than bitwise codes [31], but may lead to poor compression effectiveness. Both coding and decoding must be highly efficient: for example, given a symbol it is necessary to find its code extremely fast. Zero-order models are an obvious choice, because higher-order models lead to high symbol probabilities – and thus poor compression efficiency – with bitwise or bytewise codes (for a given model size), and model size must be kept small.

In view of these observations, Huffman coding is one choice of coding technique, based on a model built on symbol frequencies observed in the first buffer-load of data. Bytewise codes are another option. Possible heuristic approaches to compression that meet the constraints stated above are discussed later in this section. Another choice would be to use a semi-static scheme such as XRAY [6], in which an initial block of data is used to build a model. Each symbol, including all unique characters, is then allocated a bitwise code. XRAY provides high compression effectiveness and fast decompression; in both respects it can be superior to `gzip` on text data, for example, even though the whole file is compressed with regard to one model. (In `gzip`, a new model is built for each successive block of data. Compression effectiveness depends on block size, which is around 64 Kb in standard configurations; with small blocks no size reduction is achieved.) However, the training process in XRAY is slow.

Likely buffer sizes are a crucial factor in design of algorithms for this application. We have assumed that tens of megabytes are a reasonable minimum volume for sorting of data of up to gigabytes; in our experiments we report on performance with buffers sizes of 18.5 and 37 MB. (These were arbitrary choices based on design constraints in the system in which we undertook these experiments.) In this context, model sizes need to be restricted to at most a few of megabytes.

We now consider simple compression techniques that may be applicable for external sorting.

3.1 Huffman coding of bigrams

In compression, it is necessary to choose a definition of symbol. Using individual characters as symbols gives poor compression; using all trigrams (sequences of three distinct characters) consumes too much buffer space. We therefore chose to use *bigrams*, or all character pairs, as our symbols, giving an alphabet size of 2^{16}. The amount of memory required for the model is approximately 800 Kb, comprised of 528 Kb for the decode part (the mapping from codes to symbols, that is, the symbol table) and 264 Kb for the additional encode part (the reverse mapping).

Huffman coding yields an optimal bitwise code for such a model. Standard implementations of Huffman coding are slow; we used canonical Huffman coding, with the implementation of Moffat and Turpin [20].

3.2 Bytewise bigram coding

Bitwise Huffman codes provide a reasonable approximation to symbol probabilities; a symbol with a 5-bit code, for example, has a probability of approximately 1 in 32. Bytewise codes can be emitted and decoded much more rapidly, but do not approximate the probabilities as closely, and thus have poorer compression efficiency. However, their speed makes them an attractive option.

One possibility is to use radix-256 Huffman coding. However, given that the model is based on partial information, it is attractive to use simple, fast approximations to this approach – in particular, the bytewise codes that we have found to be highly efficient in other work [31]. In these *variable-byte* codes, a non-negative integer is represented by a series of bytes. One flag bit in each byte is reserved for indicating whether the byte is final or has a successor; the remaining bits are used for the integer. Thus the values 0 to 2^7-1 can be represented in a single byte, 2^7 to $2^{14}-1$ in two bytes, and so on. Using these bytewise codes, the calculation of codes for bigrams can be dispensed with. The bigrams are simply sorted from most to least frequent and held in an array, and each bigram's array index is its code. The first 2^7 or 128 most frequent bigrams are encoded in one byte, the next 2^{14} are be encoded in two bytes, and so on.

This scheme is simple and fast, but does have the disadvantage that compression can no more than halve the data size, regardless of the bigram probabilities, whereas Huffman coding could in principle provide reduction by around a factor of 16 (ignoring the sort key and record length, which in our application must be kept uncompressed). The model sizes are identical to those for Huffman coding of bigrams above.

3.3 Bytewise common-quadgram coding

To achieve better compression than is available with bigrams, we need to include more information in each symbol. Longer character sequences can yield better compression, but models based on complete sets of trigrams and quadgrams are too large. Another approach is to model common grams and use individual characters to represent other letter sequences. In a 32-bit architecture, it is efficient to process 4-byte sequences, and thus we explored a compression regime based on quadgrams and individual characters.

Because buffer space is limited, we cannot examine all quad-grams and choose only the commonest. As a heuristic, our alphabet is the first L quadgrams observed, together with all possible 256 single characters. We use a hash table with a fast hash function [26] to accumulate and count the first L overlapping quadgrams, and simultaneously count all character frequencies. The symbols – quadgrams and characters together – are then sorted by decreasing frequency, and indexed by bytewise codes as for bytewise bigram coding. This scheme is not perfect; for example, "ther" and "here" may both be common, but they often overlap, and if one is coded the other isn't. We believe that determining an ideal set of quadgrams is NP-hard. However, the frequencies are in any case only an approximation, as only part of the data has been inspected. In the presence of overlap, choosing which quadgram to code (rather than greedily coding the leftmost) can improve compression, but is slower. We use the simple greedy approach.

We varied L for the two buffer sizes tested, using $L = 2^{16}$ for the 18.5 Mb buffer and $L = 2^{17}$ for the 37-Mb buffer. The amount of memory required for the model is approximately 1.8 Mb (528 Kb for the decode part and 1.3 Mb for the encode part) or 3.0 Mb (528 Kb for the decode part and 2.5 Mb for the encode part). As for bigram coding, the commonest 2^7 symbols are represented in a single byte. In detail, the method for building the model is as follows:

Construction of a model of common quadgrams:

1. Initialise the model to contain all possible 256 single characters.
2. As characters are encountered, update their frequency in a *char-table*.
3. Use the character to create a new quadgram and update its frequency in a *quadgram table*, which contains entries for the first L quadgrams observed.
4. When the end of the input buffer in reached, copy the quadgrams and their frequencies to a *decode table*.
5. Sort symbols (characters and quadgrams) into order of decreasing frequency.
6. For each symbol in the decode table:
   (a) If it is a character, replace the frequency in the char-table with the ordinal position in the decode table.
   (b) If it is a quadgram, replace the frequency in the quadgram hash table with the ordinal position in the decode table.

Coding then proceeds as follows. If the current four characters from the input form a valid quadgram, its code is emitted, and the next four characters are fetched. Otherwise, the code for the first character is emitted, and the next character is fetched. Decoding proceeds by replacing successive codes by the corresponding symbols, which can be characters or quadgrams. (We observed in our experiments that about two-thirds of the output codes represented quadgrams.)

In some data, patterns of much more than four characters may be common, however, and fixed-length methods such as these – although efficient to compute – do not achieve high degrees of compression. For this reason we explored a more principled alternative.

4 Vargram compression

ptMany of the most effective compression techniques, such as Lempel-Ziv coding and XRAY, rely on identification of long repeated strings. In this section we propose a novel technique for rapidly identifying such strings. In this *variable-gram* or *vargram* compression, we gradually increase the length of the symbols added to the model as data is inspected. The Lempel-Ziv family and XRAY schemes use an in-memory tree structure to store phrases of variable length, which grow as the data is inspected. In our vargram techniques, we use in-memory trie-based structures to both store and accumulate statistics on common strings observed in the data.

That is, as data is processed, the trie is grown to represent strings of increasing length; a path through the trie represents a string. As we plan to use bytewise codes, we need to avoid having symbols of excessively high frequency, or compression efficiency will be low; hence the attractiveness of trying to identify symbols of lower, similar frequency but of varying length. By – hopefully – identifying long, not-excessively-common strings, and allocating a single short code to each such string, significant compression can be achieved.

The main advantages of tries are speed of storage and access, ease of addition or updating, facility in handing information sequences of diverse lengths, and the ability to take advantage of redundancies in the information stored. The main disadvantage is relative inefficiency in utilising storage space [10]. Various techniques haven been investigate to reduce the space requirements of tries [1, 9, 25, 28], but these techniques are generally best suited to static tries. The most successful method for dynamic tries is the burst trie [15], where leaves are bags of strings that are burst to give a new subtrie when they become too large.

There are several approaches to implementing tries, depending on the structure of the nodes and the method used to guide descent [8]. An *array-trie* uses an array of pointers to access subtrees directly, one for each character of the alphabet. To search for or insert an n-gram, each character is used in turn to determine which pointer to follow. Access is fast, with one node traversal for each character, but, for high-cardinality alphabets, utilisation can be low and most positions in the array only contain null pointers.

A *list-trie* represents non-empty subtrees as linked lists, reducing the storage costs, but access is slower as an array lookup is replaced by linked list traversal. We have not used an implementation based on list-tries, as Clement et al. [8] showed that list-tries require approximately three times as many comparisons as the TST discussed below.

A *BST-trie* uses binary search trees as the subtree access method. An effective structure of this type is the *ternary search trie* or TST [3, 32]. In a TST, each node has a character and three links. Two of the links, the left and right, point to nodes with characters less than and greater than the current node, and are used to link to characters on the same level of the trie. Thus for a particular level – which corresponds to a particular character in the n-gram – characters are searched for as in a BST. When a match has been found, the third link or middle pointer is used to descend to the next level of the trie. The next character in the n-gram is then searched for as describe above, and the trie traversed until all characters of the n-gram have been found. While this structure has lower memory requirements compared to the array-trie (there are no null pointers), this comes at the expense of increased access cost. While the array-trie only requires a simple array lookup, the TST requires a binary search for each character in the n-gram.

We use array tries and TSTs to represent models. Due to the tight constraints on buffer space available for a model, a parameter was used to bound the size of the model. To avoid storing n-grams that appear only a small number of times, the creation of a new node is delayed until a predetermined number of instances of an n-gram have been encountered, which we term the *burst limit*. A similar idea is that of *inhibiting rule formation* [22]. The number of nodes in the trie and the number of symbols or n-grams stored were also limited to ensure that the model would fit within specified memory limits. The model parameters are discussed in more detail below.

4.1 Array-trie implementation

The first stage of compression is building a model. We propose that the encode part of the model consist of an array-trie structure. Each node consists of two arrays of 256 positions, one to count occurrences of each n-gram (and later used to contain the ordinal position of the symbol in the code table), and one containing pointers to the next node in the sequence, as illustrated in Fig. 2. The arrays are indexed by the current character or byte value.

The decode part of the model consists of a symbol buffer containing all the n-grams identified during encoding, accessed via a code table of which each element contains the frequency of the n-gram, a pointer to the n-gram in the symbol buffer, and its length, as illustrated in Fig. 3. This is analogous to the dictionary used in other compression techniques.

Two of the parameters used to bound the size of the model are the number of nodes and the number of symbols or codes. The other parameter is the burst limit, which delays the creation a new node until a fixed number of occurrences of an n-gram have been observed. The root node of the trie is initialised with all counts set to 1 to ensure that all possible 256 single characters are assigned codes. As each character is read from the buffer, the count for that character in the current node is incremented. When the count for an n-gram reaches the burst limit, a new node is created. Then, when the next character is read, it becomes the last character of a new $(n + 1)$-gram. Consider the example in Fig. 4. In part (a), the character ‘b’ has been observed five times. The next character in the sequence is another ‘b’. If the burst limit is set at 5, a new node is created as displayed in part (b). In part (c), the character ‘c’ is observed, its count is incremented by one, creating a new n-gram of ‘bc’. At the next character, updating starts again from the root node of the trie.

After all characters in the buffer have been inspected, the trie is traversed (see Fig. 2) and all the n-grams are copied into a symbol buffer to be used for decoding. An array is

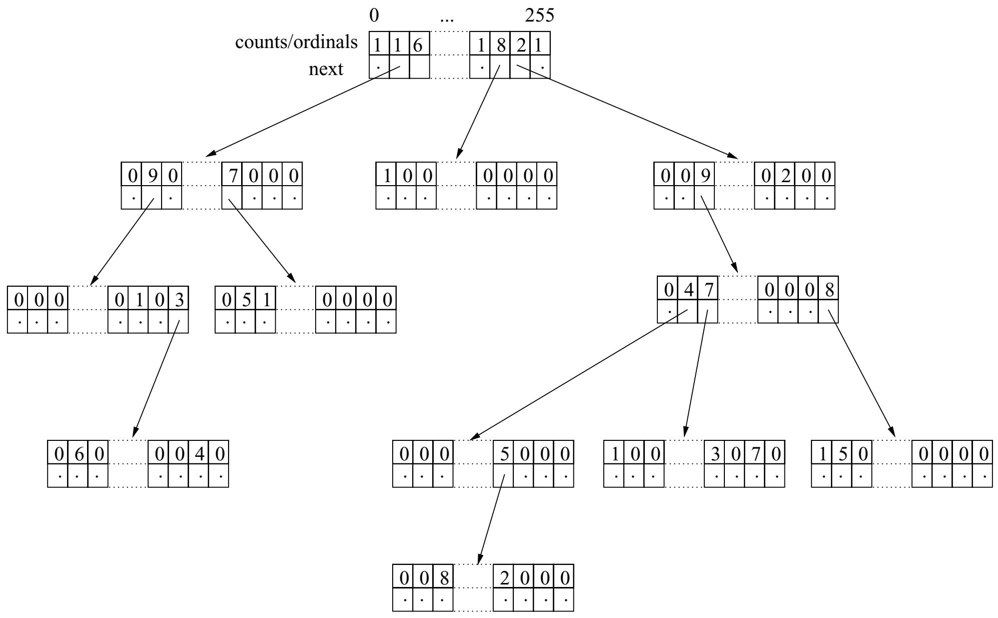
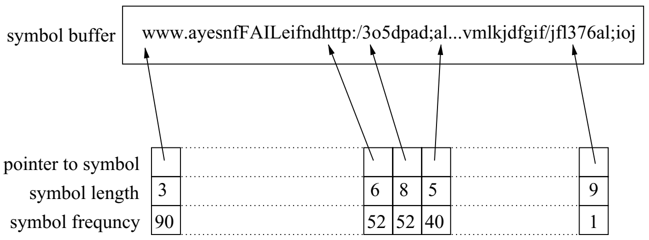


Fig. 2 An example trie structure. Each position in the array corresponds to one character or byte value


symbol buffer
www.ayesnfFAILeifndhttp:/3o5dpad;al...vmlkjdfgif/jfl376al;ioj
pointer to symbol
symbol length
symbol frequncy

symbol length	3	6	8	5	9
symbol frequncy	90	52	52	40	1

Fig. 3 An example sequence of n-grams in a model. The symbol buffer contains all the symbols of the model and is accessed via the code table. The first (and most frequent) symbol in the buffer is "www" and is 3-bytes long. The last (and least frequent) symbol is "367al;ioj" and is 9-bytes long

used to record the position of each symbol in the buffer, its length, and its observed frequency. This array is then sorted in decreasing frequency order. In the trie (which is used at encoding), the frequency of each symbol is then replaced by its ordinal position in the sorted array.

The same variable-byte integer coding scheme as for the simple bytewise schemes can be used on the ordinal position values to produce codewords. As for the other bytewise schemes, the commonest 2^7 symbols are represented in a single byte, the next 2^{14}–2^7 in two bytes, and so on. Thus the most common symbols are assigned shorter codewords, and the less frequent symbols progressively longer codewords. In detail, the model construction algorithm is as follows.

Constructing a vargram model using an array trie:

1. Initialise the model: create the root node and initialise the count for each character to 1.
2. Build the trie as data is inspected.
   (a) Start at the root node.
   (b) Until all characters have been read from the buffer,
      i. Read a character from the buffer and increment its frequency
      ii. If the current node is a leaf node, then
         A. If frequency for the character is 0 (character sequence not observed yet) then increment the number of n-grams observed, and return to the root node.
         B. If the frequency is equal to the burst limit then create a new child node corresponding to the current character, and start the next iteration at the new node.
         C. Otherwise, return to the root node.
      iii. Otherwise, traverse the appropriate pointer.

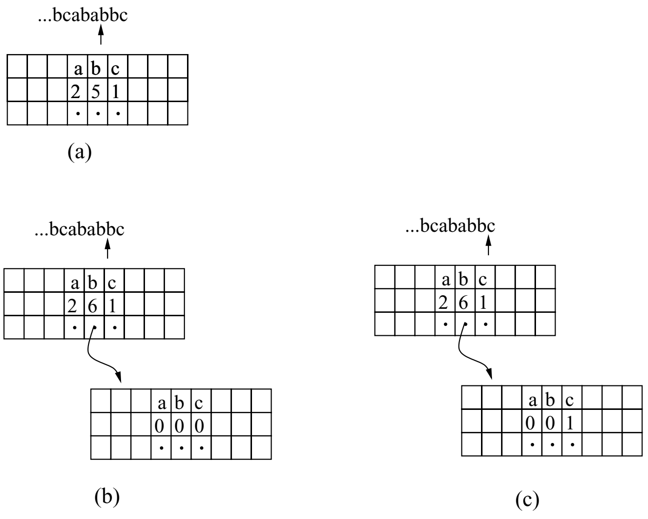


Fig. 4 Creation of a new node in the trie when the burst limit is reached

3. Traverse the trie and for each symbol, copy the symbol and its frequency to the symbol buffer, and copy the symbol length to the code table.
4. Sort the code table in decreasing frequency order.
5. For each symbol replace its frequency in the trie with its ordinal position in the sorted code table.

Coding proceeds as follows. As characters are read from the input, the trie is traversed until a leaf node, or a non-valid (that is, unobserved) n-gram is encountered. This finds a match for the longest symbol stored in the model. For a leaf node, the codeword for the current n-gram is emitted. For a non-valid n-gram, the codeword representing all the previous characters (or current $(n-1)$-gram) is emitted, and trie traversal then commences from the root node for this character. The algorithm for encoding is as follows.

Encoding of data using an array trie:

```
1. Start at the root node.
2. Until all characters have been encoded,
    (a) Read a character from the buffer.
    (b) If the current $n$-gram represented by path from
        root to this character has not been observed,
        i. Variable-byte encode the ordinal number for
           symbol containing the previous characters of
           the $n$-gram (that is, the $n$-gram excluding this
           character).
        ii. Return to the root node and traverse from the
            pointer for this character.
    (c) If the current node is a leaf node then variable-
        byte encode the $n$-gram and return to the root
        node.
    (d) Otherwise, use the current character to traverse
        the trie.
```

Decoding proceeds by replacing successive codewords by the corresponding symbol or n-gram. Since the codeword is the variable-byte encoding of the ordinal position of the symbol in the symbol buffer, decoding is fast and is simply a matter of decoding the variable-byte integer, and using this as an offset from which the symbol or n-gram can be emitted. The algorithm for decoding is as follows.

Decoding using an array trie:

```
1. Until the number of symbols decoded is equal to the
   number of symbols encoded,
    (a) Decode one variable-byte integer.
    (b) Use this value as an index into the code table and
        read the length and position of the symbol in the
        symbol buffer.
    (c) Emit the symbol stored in the symbol buffer.
```

The burst limit and the maximum number of codewords and nodes were varied on some sample data to observe the effect on compression effectiveness, in experiments not reported here. We then chose settings that achieved a good balance between model size and compression effectiveness. The maximum number of codewords was to limited to 50,000, the maximum number of nodes was limited to 2500, and the burst limit was set at 100. These values were chosen using a range of preliminary experiments. This resulted in a total model size of approximately 5.64 MB, with the decode part of the model only requiring approximately 0.76 MB. Results for these and some other combinations are shown later.

When building the models above, the input buffer was simply parsed character by character from the beginning of the buffer to the end. Consider the example sequence `abbcadbac`. If the character sequence `abbc` (positions 1–4) has just been added to the trie, one option is to go back to position 2 and continue looking at the overlapping sequences, such as `bbca`, or another option is to ignore the overlapping sequences and simply continue from position 5 searching for new substrings beginning with `adbac`. In experiments not reported here, we examined at the effects of building a model using overlapping n-grams. However, for models based on overlapping n-grams to be effective, they require model build times and memory requirements much too large to make them efficient enough for use in this application, and so were not used when building the compression models in the experiments.

As noted earlier, we believe that determining an ideal set of n-grams is NP-hard. The frequencies are in any case only an approximation, as only part of the data has been inspected. Also, choosing which n-gram to code (rather than greedily coding the leftmost) can improve compression, but is slower. We again use the simple greedy approach.

4.2 TST implementation

In our experiments, we observed that node utilisation was low. On average only about 10% of the 256 positions in the array of the array-trie implementation were occupied. An alternative is to use a TST, with, for efficiency, an array trie node as the root. Thus we have an array of 256 TSTs, one for each possible value of the first character of an n-gram. An example is illustrated in Fig. 5.

With a TST, the decode part of the model is identical to that of the trie implementation. The algorithms for building the model and encoding are similar to the array trie implementation, except that the root node is treated differently to the other nodes, and, when a character is first observed following a given prefix, a node must be created for it in the TST.

As before, the burst limit and maximum number of codewords and nodes were varied on sample data to observe the effect on compression effectiveness. We then chose a set that achieved a good balance between model size and compression effectiveness. The maximum number of codewords and nodes was limited to 150,000, and the burst limit was set at 2. This resulted in a total model size of approximately 5.55 MB (3.15 MB for the decode part and 2.40 MB for the encode part).

As for the array trie implementation, models based on overlapping n-grams require model build times and memory

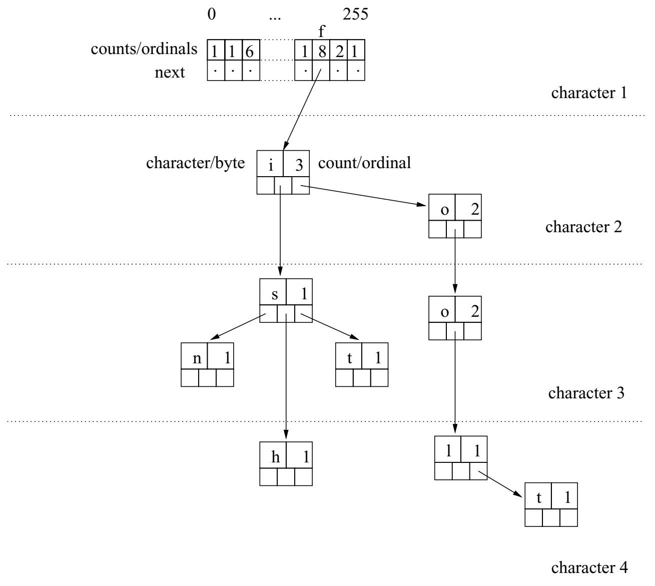


Fig. 5 An example tst structure containing the n-grams "fish", "fin", "fit", "fool" and "foot"

requirements much too large to make them efficient enough for use in this application, and so were not used when building the compression models in the experiments.

5 Comparison of compression techniques

We have proposed compression techniques for the specific task of compressing data record by record, using a semi-static model. The performance of the different approaches varies considerably in both speed and compression efficiency, and varies from file to file.

One way of benchmarking them is to compare them to general-purpose compression methods. These do not allow random access to compressed data, and so have fewer constraints on the way data can be represented or manipulated, but – even though they cannot be used for sorting – do provide a point of reference. Our compression techniques, and XRAY, are semi-static and have been developed to allow random access to compressed data. However, even these are not directly comparable. XRAY's primary aim is for effective compression and fast decompression, with the memory and processor resources required for compression being a secondary consideration. Our techniques, while also semi-static, are intended to be efficient in both compression and decompression.

We use a range of data sets to compare the compression methods and their impact on sorting, to explore performance on data with different characteristics. The data sets are as follows:

Proxy log. Records from a web proxy cache log. Each consists of a fixed series of fields, including URL, file size, time and date, and file type. Each record in the relation has eleven fields, eight of which are string fields, two are floating point numbers, and one is an integer.

Web crawl. Lines of text from a large web crawl that has been widely used in TREC experiments (see `trec.nist.gov`). This data contains nine fields: two string and seven integer, including the text itself, counters, and lengths.

Low-cardinality data. As discussed earlier, some other database compression research focuses on low-cardinality data. We used the proxy log as above, but replaced each of the eleven fields by the ASCII representation of its length in bytes.

Canterbury corpus. The large corpus component of the Canterbury Corpus (`corpus.canterbury.ac.nz`), which is a mix of documents of a mix of sizes. It is a benchmark that enables researchers to evaluate lossless compression methods. We used *Escherichia coli* (the complete genome of the E.Coli bacterium) of 4,638,690 bytes and and world192.txt (the CIA world fact book) of 2,473,400 bytes.

For each of the first three sets we have 10 GB of data.

The first data set used is the proxy log. The top third of Table 1 shows results for the compression utilities `gzip` and `bzip2`, which cannot be used in conjunction with sorting but – as commonly used utilities – provide an interesting benchmark. The middle third shows results for XRAY [6] and our implementation of canonical Huffman coding based on algorithms described by Moffat and Turpin [20] for single characters and bigrams. The bottom third of the table contains results for our new compression techniques. In these timings, our compression techniques build a model based on only the first 36 MB of data. This value can be varied, but, in our sorting experiments – where we are compressing on the fly – it will be inefficient to examine the entire database, and so we need to sacrifice some compression effectiveness for increased efficiency by building a compression model on only a small portion of the data. There are two sets of results for each of our vargram implementations, the first with parameters as used in the sort experiments (v1), and another set from parameters that give better compression effectiveness (v2), at the cost of larger model sizes and compression times.

While XRAY achieves the highest compression effectiveness and is fast to decompress, it is, like `bzip2`, slow to compress, and it requires large amounts of memory. For the default configuration of XRAY (v1) the peak memory usage was approximately 170 MB. The Huffman and fixed-length n-gram methods are relatively fast to compress or decompress, but the compression efficiency is low.

The compression achieved by our semi-static vargram techniques is comparable to `gzip`, although our techniques are slightly slower. TST vargram v2 achieves the most effective compression, but is the slowest of our schemes and requires a model size in excess of 100 MB. The bytewise codes have been surprisingly effective; for these methods, we have shown entropies, which are the sizes that would be

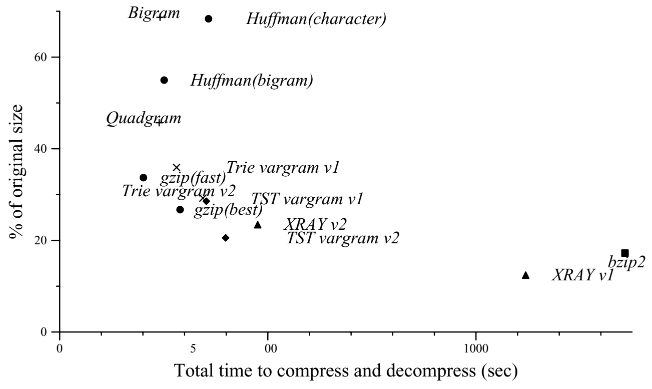


Fig. 6 Size of the compressed data as a percentage of the original size versus the total time to compress and decompress 1 GB of text data from a web proxy cache log

achieved with a perfect code; the bytewise codes are consistently about 20% greater.

Results for compressing 1 GB of text data from a web proxy cache log are graphed in Fig. 6. The distribution of the results is similar to that of Table 1. Both XRAY and `bzip2` provide good compression, but are slow, while `gzip` is both efficient and effective. Our semi-static schemes again have comparable compression effectiveness to `gzip`. However, compared to compression of 100 MB, our semi-static schemes have declined slightly. In both experiments, only the first 36 MB of data is inspected to build the compression model, so presumably the model is in this case less representative.

Table 2 shows results of similar experiments on 100 Mb of the web crawl data. The compression effectiveness is less for all of the techniques compared to Table 1. The distribution of results is similar to those for the web proxy cache log, but our vargram techniques are closer in performance to `gzip`.

Table 3 shows results for 100 Mb of low-cardinality data. As expected, this data is more compressible and all the compression schemes achieve better results. The relativities are much as the same as before, and the vargram compression schemes again have effectiveness and efficiency comparable to `gzip`. The innate defects of the fixed-length *n*-gram methods are, however, more obvious.

We also tested these methods on documents from the Canterbury Corpus. The results in Tables 4 and 5 illustrate the compression effectiveness and efficiency of the various schemes. In the context of databases, these file sizes are very small. In our other experiments we use data sets of several gigabytes. Even though the data is probably not likely to represent that in a typical database, it is still informative to compare these methods with different types of data.

The results for our methods compared to the others is best for the *E. coli* file (Table 4). For this file, with trie vargram v1 the compression effectiveness falls between `gzip`-fast and `gzip`-best, but with a time closer to `gzip`-fast. The compression effectiveness is also close to that of XRAY and with lower compress and decompress times. The result for trie vargram v2 in this table, in contrast to all the others, is worse than for trie vargram v1. However, as mentioned above, the second set of parameters are those that gave better compression for the web proxy cache log. For the text file (Table 5) our methods do not perform as well. This is probably due to a lower degree of repetition compared to *E.coli*, and demonstrates the advantage of adaptive schemes that are able to continuously update their compression models as the data is being compressed. Also, this material has locality that is absent in the proxy log (and cannot be used in semi-static compression), giving `gzip` and `bzip2` opportunities for improved compression.

It is possible that, for these small files, our vargram compression schemes would be able to provide better compression with a different set of parameters. In these experiments

Table 1 Results for compression utilities `gzip` and `bzip2`, XRAY-best (default), XRAY-fast (limited to 36 MB sample and five passes), canonical Huffman coding of single characters and bigrams, and our new methods: bigram, quadgram, trie vargram v1 (burst limit 100, max. nodes 2500, max. codes 50,000), trie vargram v2 (burst limit 5, max. nodes 50,000, max. codes 100,000), TST vargram v1 (burst limit 2, max. nodes 150,000, max. codes 150,000), TST vargram v2 (burst limit 1, max. nodes 2,000,000, max. codes = 2,000,000) on 100 MB of text data from a web proxy cache log

	Compress time (s)	Decompress time (s)	% of original size	Entropy
`gzip`-best	23.3	6.3	25.22	–
`gzip`-fast	13.3	6.5	32.30	–
`bzip2`	109.3	33.3	15.72	–
XRAY-best	358.4	6.4	11.89	–
XRAY-fast	109.3	7.4	23.17	–
Huffman (character)	18.0	18.6	68.39	–
Huffman (bigram)	14.0	12.1	54.93	–
Bigram	16.8	8.3	68.60	–
Quadgram	22.1	7.1	45.15	–
Trie vargram v1	23.1	7.1	34.53	29.50
Trie vargram v2	34.9	7.7	27.55	22.74
TST vargram v1	35.3	6.5	26.76	22.06
TST vargram v2	46.6	5.4	18.28	16.15

Table 2 Results for compression utilities `gzip` and `bzip2`, XRAY -best (default), XRAY -fast (limited to 36 MB sample and 5 passes), canonical Huffman coding of single characters and bigrams, and our new methods: bigram, quadgram, trie vargram v1 (burst limit 100, max. nodes 2500, max. codes 50,000), trie vargram v2 (burst limit 5, max. nodes 50,000, max. codes 100,000), TST vargram v1 (burst limit 2, max. nodes 150,000, max. codes 150,000), TST vargram v2 (burst limit 1, max. nodes 2,000,000, max. codes = 2,000,000) on 100 MB of text data created from TREC wt10g web data

	Compress time (s)	Decompress time (s)	% of original size	Entropy
`gzip` -best	23.06	4.41	33.66	–
`gzip` -fast	13.73	6.30	37.82	–
`bzip2`	148.58	36.02	24.27	–
XRAY -best	523.86	8.31	24.07	–
XRAY -fast	158.80	8.29	28.34	–
Huffman (bigram)	14.00	10.45	55.05	–
Bigram	14.04	6.58	69.56	–
Quadgram	25.59	4.88	47.25	–
Trie vargram v1	24.04	8.52	41.50	35.56
Trie vargram v2	33.33	8.80	38.75	31.18
TST vargram v1	40.94	9.89	38.43	31.27
TST vargram v2	55.85	7.93	30.00	26.52

Table 3 Results for compression utilities `gzip` and `bzip2`, XRAY -best (default), XRAY -fast (limited to 36 MB sample and 5 passes), canonical Huffman coding of single characters and bigrams, and our new methods: bigram, quadgram, trie vargram v1 (burst limit 100, max. nodes 2500, max. codes 50,000), trie vargram v2 (burst limit 5, max. nodes 50,000, max. codes 100,000), TST vargram v1 (burst limit 2, max. nodes 150,000, max. codes 150,000), TST vargram v2 (burst limit 1, max. nodes 2,000,000, max. codes = 2,000,000) on 100 MB of low cardinality data

	Compress time (s)	Decompress time (s)	% of original size	Entropy
`gzip` -best	42.64	5.59	5.41	–
`gzip` -fast	7.35	5.87	9.68	–
`bzip2`	287.97	24.95	3.25	–
XRAY -best	147.86	2.68	3.11	–
XRAY -fast	53.99	4.36	9.85	–
Huffman (bigram)	13.78	9.49	25.17	–
Bigram	13.75	5.39	50.00	–
Quadgram	16.15	3.10	26.95	–
Trie vargram v1	22.03	4.26	8.65	6.86
Trie vargram v2	33.25	3.07	6.34	5.40
TST vargram v1	19.99	3.16	6.69	5.60
TST vargram v2	26.43	3.10	6.73	5.55

Table 4 Results for compression utilities `gzip` and `bzip2`, XRAY, canonical Huffman coding of single characters and bigrams, and our new methods: bigram, quadgram, trie vargram v1 (burst limit 100, max. nodes 2500, max. codes 50,000), trie vargram v2 (burst limit 5, max. nodes 50,000, max. codes 100,000), TST vargram v1 (burst limit 2, max. nodes 150,000, max. codes 150,000), TST vargram v2 (burst limit 1, max. nodes 2,000,000, max. codes = 2,000,000) on *E.coli* from the Canterbury Corpus (file size of 4.42 MB)

	Compress time (s)	Decompress time (s)	% of original size	Entropy
`gzip` -best	12.48	0.25	28.00	–
`gzip` -fast	1.08	0.49	32.91	–
`bzip2`	5.09	1.84	26.97	–
XRAY	21.18	0.58	28.53	–
Huffman (character)	0.68	0.64	28.07	–
Huffman (bigram)	0.52	0.44	25.28	–
Bigram	0.43	0.31	50.00	–
Quadgram	1.08	0.22	34.12	–
Trie vargram v1	1.75	0.25	30.24	25.67
Trie vargram v2	3.59	0.49	32.22	25.03
TST vargram (1)	2.79	0.39	32.62	25.51
TST vargram (2)	3.76	0.42	31.00	24.69

Table 5 Results for compression utilities `gzip` and `bzip2`, XRAY, canonical Huffman coding of single characters and bigrams, and our new methods: bigram, quadgram, trie vargram v1 (burst limit 100, max. nodes 2500, max. codes 50,000), trie vargram v2 (burst limit 5, max. nodes 50,000, max. codes 100,000), TST vargram v1 (burst limit 2, max. nodes 150,000, max. codes 150,000), TST vargram v2 (burst limit 1, max. nodes 2,000,000, max. codes = 2,000,000) on world192.txt from the Canterbury Corpus (file size of 2.36 MB)

	Compress time (s)	Decompress time (s)	% of original size	Entropy
`gzip`-best	0.80	0.75	29.17	–
`gzip`-fast	0.53	0.29	37.11	–
`bzip2`	2.33	1.20	19.79	–
XRAY	10.22	0.71	24.69	–
Huffman (character)	0.43	0.44	63.02	–
Huffman (bigram)	0.42	0.31	54.87	–
Bigram	0.32	0.22	68.97	–
Quadgram	1.01	0.16	49.70	–
Trie vargram v1	0.93	0.25	49.89	43.99
Trie vargram v2	2.76	0.35	40.02	33.26
TST vargram (1)	1.99	0.29	37.81	31.07
TST vargram (2)	3.27	0.43	34.91	26.97

on the files from the Canterbury Corpus, we used the parameters we found to work well with 100 MB of data; in particular, we did not want to exceed the memory limit on the compression model too early, so that a greater percentage of the data could be inspected when gathering statistics. With these smaller files, the limits used in our vargram compression schemes could have been relaxed, as we did not use all the memory allocated for the model.

Overall, these results show that our fast semi-static compression techniques achieve reasonable compression performance compared to general-purpose adaptive compression techniques and the semi-static XRAY, and that the speed of compression is of similar magnitude to that of disk transfer rates. The compression techniques are versatile and can be parameterised to balance memory requirements, compression efficiency, and compression effectiveness. Our vargram compression techniques have not been designed to achieve the best possible compression, but are designed to trade the various requirements of compression effectiveness against processor and memory utilisation; yet compression efficiency is close to that of `gzip`. In the sections following, we make use of our compression schemes in sorting operations.

6 External sorting with compression

By incorporating compression into sorting, we aim to reduce the time taken to sort due to better use of memory, reduced I/O transfer costs, and the generation of fewer runs, which will reduce the costs of merging. One of the two key questions of this research is how to integrate compression into standard database operations such as sort.

Compression could be used simply to speed memory-to-disk transfers, by compressing runs after they have been sorted and decompressing them as they are retrieved. This approach has the advantage that high-performance adaptive compression techniques could be used, but also has disadvantages. In particular, it does not allow reduction in the number of runs generated, and at merge time a separate compression model must be used for each run.

The alternative is to compress the data as it is loaded into the buffer, prior to sorting. This allows better use to be made of the buffer; reduces the number of runs; and, since semi-static compression must be used, the same model applies to all runs. However, the compression is unlikely to be as effective. Nonetheless, given the cost of adaptive compression and the advantages of reducing the number of runs – such as increasing the buffer space available per run and reducing disk thrashing – it is this alternative that we have explored in our experiments.

Figure 7 illustrates the entire external sorting process with compression. In this figure, the input buffer is of size A, the output buffer is of size B, and the compression model size is C.

External sorting with compression:

Build the compression model The arrangement of buffers is shown in Fig. 7a. The input buffer has capacity $A - C$ to store records.

1. Fill the input buffer with records from the relation to be sorted
2. Build a model based on the symbol frequencies in these records.

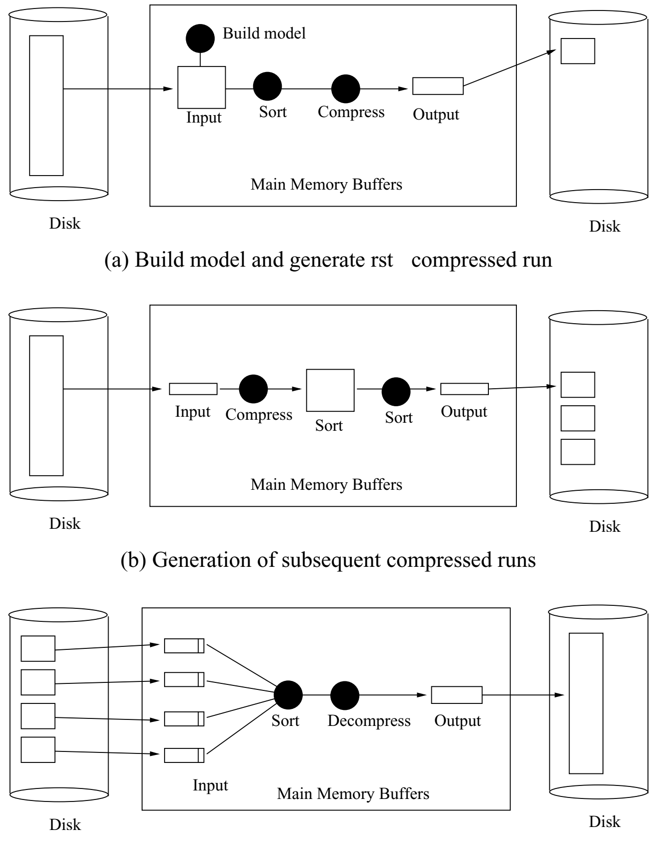
Generate the first run

1. Sort on the keys of the records in the input buffer.
2. In sorted order, compress the records then write them to disk as a sorted run.

Generate the remaining compressed runs The arrangement of buffers is shown in Fig. 7b; note that to increase the number of records per run, they must first be compressed. The input and output buffers are of size $B/2$ each, and the sort buffer is of size $A - C$.

Repeat the following until all data has been processed:

1. Fill the input buffer with data.



(a) Build model and generate rst compressed run

(b) Generation of subsequent compressed runs

(c) Merge of compressed runs

Fig. 7 External sorting with compression. The first stage is using the initial data to determine a model. Then runs are generated and merged as before, but compression is used to increase the number of records per run

2. Compress each record in the input buffer, then write it to the sort buffer. Continue until the sort buffer is full, reloading the input buffer as necessary.
3. Sort on the keys of the records in the sort buffer.
4. In sorted order, write the compressed records to disk, forming one run.

Merge all runs The arrangement of buffers is shown in Fig. 7(c). The input buffer is of size $A - C$, the output buffer is of size B. Note that only the decoding part of the model is required in this phase, so C is smaller.

1. Divide the input buffer space amongst the runs, giving per-run buffers, and fill with data from the compressed runs.
2. Find the smallest key among the first remaining record in each buffer, decompress the corresponding record and move it to the first available position of the output block.
3. If the output buffer is full, write it to disk and reinitialise.
4. If the input buffer from which the smallest key was taken is exhausted, read from the same run and fill the same input buffer. If no blocks remain in the run, then leave the buffer empty and do not consider keys from that run in any further sorting.

number of bytes encoded (vbyte)	key (uncompressed)	record key compressed and padded to byte boundary

Fig. 8 Format of record in compressed run

5. Repeat until all buffers are empty.

In this algorithm, the sort key must be left uncompressed, and to simplify processing each compressed record should be prefixed with a byte-length. Compressed records start with the number of bytes encoded in the compressed part of the record (represented as a variable-byte integer) followed by the key attribute, which is left uncompressed. The remainder contains the original record, excluding the key, compressed and then padded to a byte boundary. The format of compressed records is shown in Fig. 8.

When comparing sorting techniques, each should use the same fixed amount of buffer space. If compression is not used, all the buffer space is available for sorting. For the compression-based sort algorithms, the buffer space available for sorting will be reduced by the memory required by the compression model. In the run generation phase, both the encode model and the decode model are used, so the available buffer space is reduced by the combined size of the models. However, once the runs have been generated, only the decode model is required for merging, so the buffer space during the merge phase is only reduced by the memory required to store the decode model.

In Sect. 2, we noted that if $B(R)$ is used to denote the number of blocks that are required to hold all the tuples of relation R, then the number of disk I/Os used by the external sort algorithm without compression, ignoring the handling of the output, is $B(R)$ to read each block of R when creating the runs, $B(R)$ to write each block of the runs to disk, and $B(R)$ to read each block from the runs at the appropriate time when merging. Thus, the total cost of this algorithm is $3B(R)$ I/Os.

If the compression scheme used is able to reduce the size of the runs by, say, 50%, the I/O cost becomes $B(R)$ to read each block of R when creating the runs, $B(R)/2$ to write each block of the runs to disk, and $B(R)/2$ to read each block from the runs at the appropriate time when merging. This results in a total cost of $2B(R)$ I/O's for the compression based algorithm.

As the volume of data increases, compression will reduce the total I/O cost of the algorithm. However, this does come at the expense of higher processor costs for compressing and decompressing the data. A further benefit of compression is a reduction in the merge costs. With compression it is possible for each run (excluding the first) to contain more records, thereby reducing the number of runs, which will allow larger per-run buffers in the merge phase. Reducing the number of runs also reduces the cost of the merge. Again, this is more difficult to quantify. The overall effect of the addition of compression to the external sort algorithm is expected to reduce the execution time of the sort, with the amount of reduction dependent on the different trade-offs of processor and I/O costs.

Our experiments were run on a typical personal computer with one processor and one non-RAID disk. We do not investigate other approaches reported in the literature to reduce I/O bottlenecks such as systems with multiple processors or disks. In these approaches the reduction in I/O access times reduces the time the processor is idle, and therefore reduces the effectiveness of the use of compression which relies on idle processor time for the compression and decompression of data. On the other hand, with an additional processor the benefit of compression could be much greater, and in both cases we would retain the benefits of reduced merge costs due to the generation of a smaller number of runs. As with approaches such as AlphaSort, we do not sort entire records, but rather sort key–pointer pairs, making more effective use of the cache.

7 External sort experiments

To test the effect of compression on external sorting, we implemented a fast external sorting routine, and added as options the compression schemes described earlier. Runs were sorted with an efficient implementation of quicksort [4]. We are confident that the implementation is of high quality. For example, on the same data and with similar parameters, the Unix sort utility takes almost twice as long (or four times as long to sort on strings, for experiments not reported here).

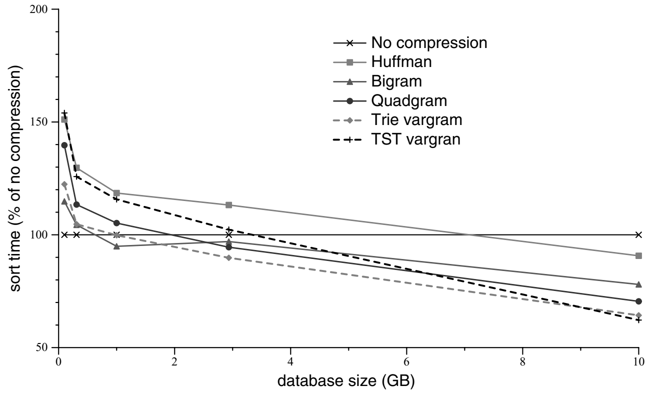
Two buffer sizes were used, as earlier, 18.5 and 37 MB. Data volumes tested ranged from 100 Mb to 10 Gb of distinct records.

All experiments were carried out on an Intel 1 GHz Pentium III with 512 Mb of memory and a 120 Gb hard drive (Ultra ATA/100 interface, average seek time of 14.3 ms, 7200 rpm) and running the Linux operating system (RedHat 8.0). Other processes and disk activity were minimised during experiments, that is, the machine was under light load.

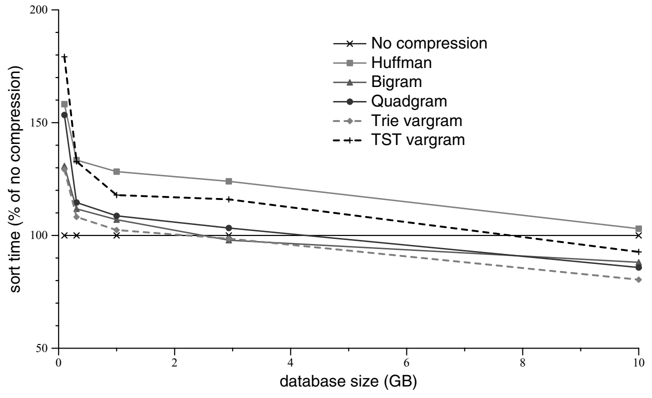
Proxy log. For the proxy data, Tables 6–11 show the effect that incorporating compression into external sorting has on elapsed time and temporary disk requirements. The task was to sort these on one of the numerical fields. The tables show experimental results for external sorting without compression, with Huffman coding of bigrams, and with the four new compression schemes we have developed: bigram, quadgram, trie vargram, and TST vargram. The "build model" time is the time to determine the model. The "generate runs" time is the time to read in the data and write out all the runs. The "merge runs" time is the time to read in and merge the runs and write out the result. The total sort times are illustrated in Fig. 9, including additional data points.

These results show that, as the volume of data being sorted grows – or as the amount of buffer space available decreases – compression becomes increasingly effective in reducing the overall sort time. The gains are due to reduced disk transfer, disk activity, and merging costs, savings that eventually outweigh the increased processor cost incurred by compression and decompression of the data. In the best case observed, with an 18.5 Mb buffer on 10 Gb of data, total time is reduced by a third. The computationally more expensive methods, such as Huffman coding and the quadgram and vargram schemes, are slow for the smaller data sets, where the disk and merging costs are a relatively small component of the total. For a given buffer size, the cost of building each run is more or less fixed, and thus run construction cost is linear in data size; merge costs are superlinear in data size, as there is a log K search cost amongst K runs for each record merged. Use of hierarchical merge and other similar strategies does not affect the asymptotic complexity of the merge phase. The results in Tables 8 and 11 show that, with compression, the most benefit was achieved when there is a large difference in the the number of runs.

For the compression schemes that have a greater effectiveness (quadgram and vargram), the sort times increase more slowly with database size compared to the less effective schemes (Huffman and bigram). This is most noticeable in the upper graph in Fig. 9, which shows that, for smaller volumes of data, compression and decompression speed is the dominating factor; and that, at larger volumes,



(a) Smaller buffer



(b) Larger buffer

Fig. 9 Sort times as a percentage of the time to sort without compression for the proxy cache log. **a** With 18.5 Mb of buffer space. **b** With 37 Mb of buffer space

Table 6 Results for sorting 100 Mb created from web proxy cache log data with 18.5 Mb of buffer space, using no compression and using five alternative compression techniques

	No compression	Huffman	Bigram	Quadgram	Trie vargram	TST vargram
Build model (s)	–	0.35	0.27	2.90	0.92	2.61
Generate runs (s)	10.39	14.54	12.21	14.40	11.17	17.78
Merge runs (s)	12.91	20.31	14.26	15.24	16.42	15.50
Total time to sort (s)	23.30	35.20	26.74	32.54	28.51	35.89
Comparative (%)	100.0	151.1	114.8	139.7	122.4	154.0
CPU utilisation (%)	46.8	82.3	62.9	75.4	73.8	85.5
Number of runs	6	5	5	5	4	4
Size of runs (Gb)	0.101	0.066	0.078	0.057	0.048	0.037
Comparative (%)	100.0	64.6	76.5	56.4	47.5	36.6
Disk reads	52	398	401	408	412	408
Disk writes	427	477	525	445	398	361

Results include time to sort and temporary space required

Table 7 Results for sorting 1 Gb created from web proxy cache log data with 18.5 Mb of buffer space, using no compression and using five alternative compression techniques

	No compression	Huffman	Bigram	Quadgram	Trie vargram	TST vargram
Build model (s)	–	0.35	0.27	2.90	0.93	2.66
Generate runs (s)	114.74	148.16	120.26	144.32	117.28	166.72
Merge runs (s)	178.65	199.12	157.86	161.32	174.49	169.99
Total time to sort (s)	293.39	347.63	278.39	308.54	292.70	339.37
Comparative (%)	100.0	118.5	94.9	105.2	99.8	115.7
CPU utilisation (%)	37.5	74.3	58.7	72.3	69.4	85.6
Number of runs	56	39	46	36	36	28
Size of runs (Gb)	0.976	0.641	0.757	0.559	0.484	0.380
Comparative (%)	100.0	65.7	77.6	57.3	49.6	38.9
Disk reads	3443	5713	6347	5604	5394	5059
Disk writes	4111	4847	5349	4488	4066	3678

Results include time to sort and temporary space required

Table 8 Results for sorting 10 Gb created from web proxy cache log data with 18.5 Mb of buffer space, using no compression and using five alternative compression techniques

	No compression	Huffman	Bigram	Quadgram	Trie vargram	TST vargram
Build model (s)	–	0.35	0.27	2.92	0.95	2.64
Generate runs (s)	1239.80	1573.24	1231.39	1598.39	1257.93	1774.68
Merge runs (s)	5374.66	4423.15	4355.40	3062.31	2992.15	2337.15
Total time to sort (s)	6614.46	5996.74	5587.06	4663.62	4251.03	4114.47
Comparative (%)	100.0	90.7	84.5	70.5	64.3	62.2
CPU utilisation (%)	17.4	42.9	31.0	41.6	42.7	59.0
Number of runs	568	394	462	368	364	292
Size of runs (Gb)	9.921	6.584	7.742	5.834	5.057	4.095
Comparative (%)	100.0	66.4	78.0	58.8	50.9	41.3
Disk reads	434594	331316	381977	294669	254341	216721
Disk writes	41716	49705	54756	46330	41965	38406

Results include time to sort and temporary space required

Table 9 Results for sorting 100 Mb created from web proxy cache log data with 37 Mb of buffer space, using no compression and using five alternative compression techniques

	No compression	Huffman	Bigram	Quadgram	Trie vargram	TST vargram
Build model (s)	–	0.68	0.54	6.11	2.11	5.59
Generate runs (s)	9.38	14.85	13.26	14.47	11.55	19.10
Merge runs (s)	12.72	19.43	15.10	13.33	14.89	14.92
Total time to sort (s)	22.10	34.96	28.90	33.91	28.55	39.61
Comparative (%)	100	158.2	130.8	153.4	129.2	179.2
CPU utilisation (%)	47.4	76.6	63.0	76.1	73.0	83.2
Number of runs	3	3	3	3	3	2
Size of runs (Gb)	0.101	0.065	0.077	0.056	0.044	0.035
Comparative (%)	100	64.3	76.3	55.4	43.6	34.7
Disk reads	18	151	153	156	161	156
Disk writes	214	220	242	207	187	170

Results include time to sort and temporary space required

Table 10 Results for sorting 1 Gb created from web proxy cache log data with 37 Mb of buffer space, using no compression and using five alternative compression techniques

	No compression	Huffman	Bigram	Quadgram	Trie vargram	TST vargram
Build model (s)	–	0.69	0.54	6.09	2.09	5.62
Generate runs (s)	125.84	162.89	140.78	152.77	125.29	176.84
Merge runs (s)	172.07	218.66	177.34	164.96	177.69	168.78
Total time to sort (s)	297.91	382.24	318.66	323.82	305.07	351.24
Comparative (%)	100	128.3	107.0	108.7	102.4	117.9
CPU utilisation (%)	39.2	67.7	57.7	69.6	64.0	82.4
Number of runs	28	19	23	18	17	13
Size of runs (Gb)	0.976	0.640	0.752	0.551	0.460	0.364
Comparative (%)	100	65.5	77.0	56.5	47.1	37.3
Disk reads	862	2351	2507	2280	2221	2142
Disk writes	2056	2342	2571	2172	1977	1784

Results include time to sort and temporary space required

Table 11 Results for sorting 10 Gb created from web proxy cache log data with with 37 Mb of buffer space, using no compression and using five alternative compression techniques

	No compression	Huffman	Bigram	Quadgram	Trie vargram	TST vargram
Build model (s)	–	0.68	0.54	6.03	2.05	5.51
Generate runs (s)	1257.43	1637.50	1368.51	1565.75	1308.87	1817.42
Merge runs (s)	2671.49	2408.52	2092.86	1799.51	1847.65	1818.28
Total time to sort (s)	3928.92	4046.70	3461.91	3371.30	3158.57	3641.21
Comparative (%)	100	103.0	88.1	85.8	80.4	92.7
CPU utilisation (%)	29.6	64.3	52.1	65.9	62.1	81.3
Number of runs	287	193	226	184	165	135
Size of runs (Gb)	9.918	6.569	7.694	5.738	4.810	3.918
Comparative (%)	100	66.2	77.6	57.9	48.5	39.5
Disk reads	87446	61897	76971	52156	44215	37935
Disk writes	20861	24116	26450	22496	20449	18683

Results include time to sort and temporary space required

Table 12 Results for sorting a 10-Gb database created from TREC wt10g web data with 18.5 Mb of buffer space, using no compression and using five alternative compression techniques

	No compression	Huffman	Bigram	Quadgram	Variable - trie	Variable - TST
Build model (s)	–	0.39	0.29	3.10	0.96	3.15
Generate runs (s)	1262.28	1741.97	1228.75	1777.49	1277.13	1990.12
Merge runs (s)	5998.76	4789.49	4645.26	4170.47	3916.25	3142.54
Total time to sort (s)	7261.04	6531.46	5874.30	5951.06	5194.34	5135.81
Comparative (%)	100.0	90.0	80.9	82.0	71.5	70.7
Number of runs	586	387	447	382	371	333
Size of runs (Gb)	10.238	6.458	7.490	6.022	5.123	4.669
Comparative (%)	100.0	63.1	73.2	58.8	50.0	45.6

Results include time to sort and temporary space required

the amount of compression achieved becomes the dominating factor in determining the sort time.

For our hardware, with otherwise light load, the processor utilisation for sorting without compression shows that there are spare processor cycles that can be made use of by compression. Tables 6 to 8 show that as the volume of data and hence I/O increases, processor utilisation changes from 46.8% to 17.4%. From Tables 6 to 8, as we sort from 100 Mb to 10 Gb with TST vargram compression, processor utilisation reduces from 85.5 to 59.0%. A similar trend can be observed as the amount of available buffer space decreases, that is the utilisation of the processor again decreases decreases. Then as the amount of data increases, or the available buffer space decreases we have an even greater margin of unused processor capacity, which may become useful for example if the system is under heavier load. The processor utilisation for each compression scheme is as expected. The bit bitwise Huffman and the TST vargram scheme required larger use of the processor, while the simplest scheme, Bigram, had the lowest processor utilisation. This hardware is now several years old, and it is expected that with more modern processors, the amount of unused processor time will increase.

Despite the greater compression achieved by Huffman coding compared to bigram coding, the latter is always faster. This confirms that bytewise codes are more efficient, with the loss of compression effectiveness more than compensated for by the gain in processing speed. The quadgram and vargram compression methods had better compression effectiveness and enough processing efficiency to give results superior to the other methods for large files. The upper graph in Fig. 9 shows that external sorting with TST vargram compression was the fastest. While it was not the fastest in the lower graph, it has the steepest gradient, so it is likely that at larger data volumes it will be the most efficient.

The tables also include the size of the resulting runs, giving an indication of the amount of compression achieved. As discussed earlier the key is not compressed, and there is the extra overhead of storing the number of bytes encoded in the record, as this value is needed by the decoder. To allow random access, records must end on a byte boundary. The model is only built from symbols encountered in the first buffer, not the entire database, so the model may not be optimal. However it is worth noting that when compressing 10 Gb of data, comparing the values in Tables 8 and 11, using 36 Mb to build the model instead of 18 Mb only resulted in an extra 1–3% decrease in size.

Even though the degree of compression is only moderate (compared to the best general compression schemes), from Table 11 for Trie vargram coding, we can see that a 50.0% saving in space due to the use of compression has resulted in a 20% saving in time. From Table 8, for Trie vargram coding, a 50% reduction in the size of the runs has resulted in a 35% reduction in the sort time.

Web crawl data. Table 12 – a small buffer and a large volume of data, seen earlier to be a best case for the use of compression – and Fig. 10 – for both buffer sizes and a range of data volumes – contain results for experiments on the web crawl data. Results are consistent with those discussed above. As the amount of data to sort increased, or the amount of buffer space available decreased, compression becomes a more effective means of reducing query times, while also decreasing the amount of temporary storage required to execute the query. On this data the compression effectiveness of the techniques is similar, except for the TST vargram method, which is only slightly better than the trie vargram method. Results for compression effectiveness are consistent across all data volumes.

The best result is for the trie-based technique, as shown in Table 12, in which the sort time is about 70% that of the sort without compression. For the proxy log data, the TST vargram method had much better compression that the trie-based method, and achieved the best execution time, of about 62% of the sort not using compression.

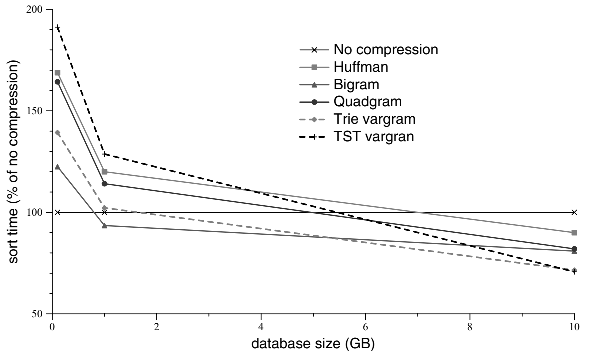
*Low cardinality data.*Fields with low cardinality are common in databases, so we experimented with low-cardinality data to contrast with the previous results. Results are presented in Table 13 (again, a best case for the use of compression) and Fig. 11. The table shows that variable-bit Huffman and the vargram schemes achieve much better compression – runs are about half that for the other data, while the fixed length bigram and quadgram bytewise schemes result in runs about two thirds the size of those for the other data. Consequently, the vargram compression schemes again lead to the

Table 13 Results for sorting a 10-Gb database created from low cardinality data with 18.5 Mb of buffer space, using no compression and using five alternative compression techniques

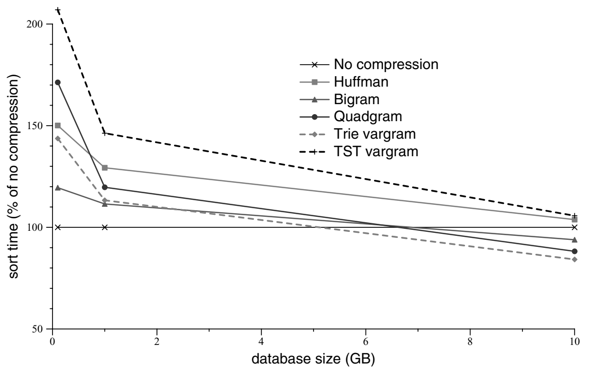
	No compression	Huffman	Bigram	Quadgram	Variable–trie	Variable–TST
Build model (s)	–	0.33	0.24	1.80	1.06	1.19
Generate runs (s)	1046.86	1789.22	1613.15	1429.82	1400.99	1257.13
Merge runs (s)	6420.39	3800.79	4677.68	3473.12	3004.90	2976.32
Total time to sort (s)	7467.25	5590.34	6291.07	4904.74	4406.95	4234.64
Comparative (%)	100.0	74.9	84.2	65.7	59.0	56.7
Number of runs	615	240	388	288	144	132
Size of runs (Gb)	10.755	3.964	6.453	4.512	1.960	1.809
Comparative (%)	100.0	36.9	60.0	42.0	18.2	16.6

Results include time to sort and temporary space required

fastest execution. The best result as a percentage of the sort time without compression was again achieved when sorting the lager amount of data, with the smaller buffer size (56.7% in Table 13). This execution time is the best result for all the data, but the difference compared to the space savings is not as great. The times for the other data are 62.2% (Table 8) and 70.7% (Table 12). This seems to indicate that the time to do the actual sorting during the run generation and merge phases still consumes a large portion of the time, and will be present in all the figures – suggesting that faster processors will lead to further relative gains for compression.

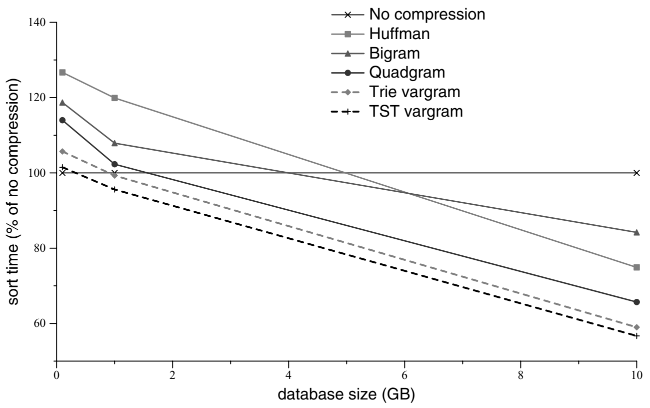


(a) Smaller buffer

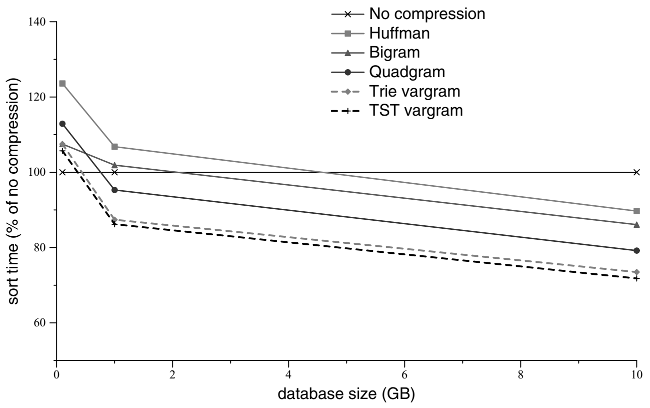


(b) Larger buffer

Fig. 10 Sort times as a percentage of the time to sort without compression for web crawl data. **a** With 18.5 Mb of buffer space. **b** With 37 Mb of buffer space



(a) Smaller buffer



(b) Larger buffer

Fig. 11 Sort times as a percentage of the time to sort without compression for low cardinality data. **a** With 18.5 Mb of buffer space. **b** With 37 Mb of buffer space

*Sorted data.*To further stress-test our techniques, we now consider a pathological case of already sorted data. Tables 14 to 17 contain results for sorting without compression and for our two best schemes, for the task of sorting pre-sorted web proxy cache data. In comparing these tables to the results of the original unsorted data (Tables 6, 7, 9, and 10) several observations can be made. The merge-runs times, which make use of heapsort, are similar. The degree of compression for each scheme is also comparable. However, the generate-runs times, which make use of quicksort, are much larger, and vary greatly between columns. The values vary from 5 to 30 times larger than for the unsorted data, demonstrating the known instability of quicksort. Comparing Table 14 to Table 9, the sort times with compression as a percentage of the sort without compression are much less than for the unsorted data; for sort with trie vargram compression, the results are 48.3% compared to 129.2%. Conversely, in comparing Table 17 to Table 7, the sort times with compression as a percentage of the sort without compression are much larger than for the unsorted data; for sort with TST vargram compression the results are 155.8% compared to 115.7%.

The results for sorting with compression on sorted data compared with unsorted data were sometimes much worse, sometimes similar, and sometimes much better. In general, sorting with compression on sorted data is likely to be less effective. On sorted data, quicksort approaches its worst case behavior of $N^2/2$ comparisons [32]. Better compression schemes allow more compressed records to fit into a buffer, potentially greatly increasing the sort time. Also, as the generate runs time increases, this phase will take up a larger percentage of the overall external sort time, lessening any I/O saving gained through the use of compression. However, as shown by these empirical results, the behavior is difficult to predict, due to the instability of quicksort.

Table 14 Results for sorting a 100 Mb database created from sorted sorted web proxy cache data with 37 Mb of buffer space, using no compression and using the two best compression techniques

	No compression	Trie vargram	TST vargram
Build model (s)	–	1.82	5.26
Generate runs (s)	282.61	125.97	180.73
Merge runs (s)	12.13	14.64	15.07
Total time to sort (s)	294.73	142.43	201.06
Comparative (%)	100.0	48.3	68.2
Number of runs	3	3	2
Size of runs (Gb)	0.101	0.046	0.037
Comparative (%)	100.0	45.5	36.6

Results include time to sort and temporary space required

Table 15 Results for sorting a 100 Mb database created from sorted web proxy cache data with 18.5 Mb of buffer space, using no compression and using the two best compression techniques

	No compression	Trie vargram	TST vargram
Build model (s)	–	0.88	2.46
Generate runs (s)	155.84	138.90	261.08
Merge runs (s)	11.89	14.89	14.82
Total time to sort (s)	167.73	154.67	278.36
Comparative (%)	100.0	92.2	166.0
Number of runs	6	5	4
Size of runs (Gb)	0.101	0.052	0.041
Comparative (%)	100.0	51.5	40.6

Results include time to sort and temporary space required

Table 16 Results for sorting a 1 Gb database created from sorted web proxy cache data with 37 Mb of buffer space, using no compression and using the two best compression techniques

	No compression	Trie vargram	TST vargram
Build model (s)	–	1.98	4.99
Generate runs (s)	958.81	1054.54	1238.75
Merge runs (s)	156.34	163.21	162.99
Total time to sort (s)	1115.15	1219.73	1406.73
Comparative (%)	100.0	109.4	126.1
Number of runs	28	17	15
Size of runs (Gb)	0.976	0.467	0.408
Comparative (%)	100.0	47.8	41.8

Results include time to sort and temporary space required

Table 17 Results for sorting a 1 Gb database created from sorted web proxy cache data with 18.5 Mb of buffer space, using no compression and using the two best compression techniques

	No compression	Trie vargram	TST vargram
Build model (s)	–	1.03	2.67
Generate runs (s)	522.02	739.23	888.91
Merge runs (s)	158.31	167.63	168.29
Total time to sort (s)	680.33	907.89	1059.87
Comparative (%)	100.0	133.4	155.8
Number of runs	56	38	32
Size of runs (Gb)	0.976	0.518	0.434
Comparative (%)	100.0	53.1	44.5

Results include time to sort and temporary space required

8 Conclusions

We have developed new compression methods that accelerate external sorting for large data files. The most successful, our TST vargram method, is based on the expedient of using ternary search tries to identify common strings, which can be replaced by bytewise codes. Our novel strategy of using an in-memory trie (or TST) that slowly grows, thus gradually capturing information about strings of increasing length, means that the resulting model can be small yet representative of a large volume of data. The technique can be parameterised to balance memory requirements, compression efficiency, and compression effectiveness.

The TST vargram method is efficient enough in memory and processor use to enable on-the-fly compression and decompression during database query processing while still providing enough compression effectiveness to reduce the overall sorting time. In the best case, on a 10 GB file, times were reduced by about 36% (or 44% for low-cardinality data), and the temporary disk space requirements by about 60% (or 80% for low-cardinality data). These results compare well to existing general-purpose sorting routines. The slopes on the graphs indicate that the reductions are likely to increase as the volume of data to be sorted increases. The gain is greatest when memory is limited, showing that the reduction in merging costs is a key reason that time is saved.

For the largest files considered, most of the savings in data volume translate directly to savings in sorting time. This strongly suggests that more effective compression techniques will yield faster sorting, so long as the other constraints – semi-static coding, rapid compression and decompression, and low memory use – continue to be met: in the context of database systems in which the stored data is typically a bag of independent records that can be retrieved or manipulated in any order, we require random access to individual records and atomic compression and decompression. It is also likely that similar techniques could accelerate other database processing tasks, in particular large joins.

In contrast to methods presented in previous work, the data need not be pre-compressed and limited to read-only querying. As the compression is used only within the sorting algorithm, our techniques can be incorporated into existing systems, without the difficulties presented by other approaches in which a model must be maintained and data must be stored compressed. That is, our methods provide a straightforward mechanism for cutting the costs of a range of applications involving manipulation of large volumes of data.

Acknowledgements This research was supported by the Australian Research Council. We thank Adam Cannane for his help with XRAY .

References

1. Al-Suwaiye, M., Horwitz, E.: Algorthims for trie compation. ACM Trans. Database Syst. **9**(2), 243–263 (1984)
2. Bell, T.C., Moffat, A., Nevill-Manning, C.G., Witten, I.H., Zobel, J.: Data compression in full-text retrieval systems. J. Am. Soc. Inf. Sci. **44**(9), 508–531 (1993)
3. Bentley, J., Sedgewick, R.: Fast alogorithms for sorting and searching strings. In: Proceedings of the 8th annual ACM-SIAM Symposium on Discrete algorithms, pp. 360–369. New Orleans, USA (1997)
4. Bentley, J.L., McIlroy, M.D.: Engineering a sort function. Software Pract. Exp. **23**(11), 1249–1265 (1993)
5. Boncz, P.A., Manegold, S., Kersten, M.L.: Database architecture optimized for the new bottleneck: Memory access. In: Proceedings of the Very Large Data Bases VLDB Conference, pp. 54–65. Edinburgh, Scotland (1999)
6. Cannane, A., Williams, H.E.: A general-purpose compression scheme for large collections. ACM Trans. Inf. Syst. **20**(3), 329–355 (2002)
7. Chen, Z., Gehrke, J., Korn, F.: Query optimization in compressed database systems. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 271–282. Santa Barbara, California, USA (2001)
8. Clement, J., Flajolet, P., Vallee, B.: The analysis of hybrid trie structures. In: Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 531–539. San Francisco, USA (1998)
9. Comer, D., Sethi, R.: The complexity of trie index construction. J. ACM **24**(3), 428–440 (1977)
10. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
11. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems Implementation, 1st edn. Prentice-Hall, Upper Saddle River, NJ (2000)
12. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: Proceedings of the 14th International Conference on Data Engineering, pp. 370–379. IEEE Computer Society, Orlando, Florida, USA (1998)
13. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Survey **25**(2), 152–153 (1993)
14. Graefe, G., Shapiro, L.: Data compression and database performance. In: ACM/IEEE-CS Symposium On Applied Computing, pp. 22–27 (1991)
15. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: A fast, efficient data structure for string keys. ACM Trans. Inf. Syst. **20**(2), 192–223 (2002)
16. Knuth, D.E.: The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd edn. Addison-Wesley, Reading, MA (1973)
17. Larmore, L.L., Hirschberg, D.S.: A fast algorithm for optimal length-limited Huffman codes. J. ACM **37**(3), 464–473 (1990)
18. Larson, P.-A.: External sorting: Run formation revisited. IEEE Trans. Knowl Data Eng. **15**(4), 961–972 (2003)
19. Manegold, S., Boncz, P., Kersten, M.: Optimizing main-memory join on modern hardware. IEEE Trans. Knowl Data Eng. **14**(4), 709–730 (2002)
20. Moffat, A., Turpin, A.: Compression and Coding Algorithms, 1st edn. Kluwer, Dordretch (2002)
21. Moffat, A., Zobel, J., Sharman, N.: Text compression for dynamic document databases. IEEE Trans. Knowl Data Eng. **9**(2), 302–313 (1997)
22. Nevill-Manning, C.G., Witten, I.H.: Phrase hierarchy inference and compression in bounded space. In: Proceedings of the Data Compression Conference, pp. 179–188 (1998)
23. Ng, W.K., Ravishankar, C.V.: Relational database compression using augmented vector quantization. In: Proceedings of the Eleventh International Conference on Data Engineering, pp. 540–549. IEEE Computer Society, Taipei, Taiwan (1995)
24. Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet, D.: Alphasort: A cache-sensitive parallel external sort. VLDB J. **4**(4), 603–627 (1995)
25. Purdin, T.D.M.: Compressing tries for storing dictionaries. In: Proceedings of the IEEE Symposium on Applied Computing, pp. 336–340, (1990)
26. Ramakrishna, M.V., Zobel, J.: Performance in practice of string hashing functions. In: Proceedings of the Databases Systems for Advanced Applications Symposium, pp. 215–223. Melbourne, Australia (1997)
27. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 2nd edn. McGraw-Hill, New York (2000)
28. Ramesh, R., Babu, A.J.G., Kincaid, J.P.: Variable-depth trie index optimization: Theory and experimental results. ACM Trans. Database Syst. **14**(1), 41–74 (1989)
29. Ray, G., Harista, J.R., Seshadri, S.: Database compression: A performance enhancement tool. In: Proceedings of the 7th International Conference on Management of Data (COMAD). Pune, India (1995)
30. Roth, M., Van Horn, S.: Database compression. ACM SIGMOD Rec. **22**(3), 31–39 (1993)
31. Scholer, F., Williams, H.E., Yiannis, J., Zobel, J.: Compression of inverted indexes for fast query evaluation. In: Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 222–229. Tampere, Finland (2002)
32. Sedgewick, R.: Algorithms in C, Parts 1–4, 3rd edn. Addison-Wesley, Reading, MA (2002)
33. Sinha, R.: Using tries for cache-efficient efficient sorting of integers. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA International Workshop On Experimental Algorithmics, pp. 513–528. Angra dos Reis, Brazil. Springer, Berlin. Published as LNCS 3059 (2004)
34. Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. In: Ladner, R. (ed.) Proceedings of the 5th ALENEX Workshop on Algorithm Engineering and Experiments, pp. 93–105. Baltimore, Maryland (2003)

35. Sinha, R., Zobel, J.: Efficient trie-based sorting of large sets of strings. In: Proceedings of the Australasian Computer Science Conference, pp. 11–18. Adelaide, Australia (2003)
36. Vitter, J.S.: External memory algorithms and data structures: dealing with massive data. ACM Trans. Database Syst. **33**(2), 209–271 (2001)
37. Westman, T., Kossmann, D., Helmer, S., Moerkotte, G.: The implementation and performance of compressed databases. ACM SIGMOD Rec. **29**(3) (2000)
38. Wickremesinghe, R., Arge, L., Chase, J.S., Scott Vitter, J.: Efficient sorting using registers and caches. J. Exp. Algorithm. **7**, 9–26 (2002)
39. Williams, H.E., Zobel, J.: Compressing integers for fast file access. Comput. J. **42**(3), 193–201 (1999)
40. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann, San Francisco, CA (1999)
41. Yiannis, J., Zobel, J.: External sorting with on-the-fly compression. In: James, A. (ed.) Proceedings of the British National Conference on Databases, pp. 115–130. Coventry, UK, July (2003)
42. Zobel, J., Moffat, A.: Adding compression to a full-text retrieval system. Software Pract. Exp. **25**(8), 891–903 (1995)
43. Zobel, J., Williams, H.E., Kimberley, S.: Trends in retrieval system performance. In: Edwards, J. (ed.) Proceedings of the Australasian Computer Science Conference, pp. 241–248. Canberra, Australia (2000)