External Sorting with On-the-Fly Compression

John Yiannis and Justin Zobel

School of Computer Science and Information Technology
RMIT University, Melbourne, Australia, 3000
{jyiannis,jz}@cs.rmit.edu.au

Abstract. Evaluating a query can involve manipulation of large volumes of temporary data. When the volume of data becomes too great, activities such as joins and sorting must use disk, and cost minimisation involves complex trade-offs. In this paper, we explore the effect of compression on the cost of external sorting. Reduction in the volume of data potentially allows costs to be reduced – through reductions in disk traffic and numbers of temporary files – but on-the-fly compression can be slow and many compression methods do not allow random access to individual records. We investigate a range of compression techniques for this problem, and develop successful methods based on common letter sequences. Our experiments show that, for a given memory limit, the overheads of compression outweigh the benefits for smaller data volumes, but for large files compression can yield substantial gains, of one-third of costs in the best case tested. Even when the data is stored uncompressed, our results show that incorporation of compression can significantly accelerate query processing.

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. With the amount of data being so large, each stage of the entire storage hierarchy of disk, controller caches, main-memory, and processor caches becomes a bottleneck. Processors are not keeping pace with growth in data volumes [1], 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 focus on reducing the costs of external sorting through 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 [2]. Thus, during an external sort, total processing time is only a tiny fraction of elapsed time.

A. James, B. Lings, M. Younas (Eds.): BNCOD 2003, LNCS 2712, pp. 115–130, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Most of the time is spent writing temporary files containing sorted runs to disk, then reading and merging the runs to produce the final sort; each run is the result of sorting one buffer of data. This imbalance, where disk activity dominates, can be partly 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. 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 needed for sorting. No standard technique meets these constraints.

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. In this framework we test several compression techniques: canonical Huffman coding and two new methods that we have developed, both of which are based on identifying the commonest letter sequences and representing them in computationally efficient bytewise codes. Our experiments show that these methods reduce sorting costs for large files. 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 [3].

Previous research [4,5,6,7,8,9] has shown the benefits of decompressing data on the fly where the data is stored compressed. However, it was found [9] 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 Compression in Retrieval Systems

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. A model is in principle a set of symbols and probabilities; 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.

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 [10], but at the start of the next record it is necessary to revert to the original model.) Further difficulties are that the data changes as records are added, modified, or deleted; that the volume of memory available to store a compression model is very much smaller than the volume of data to be compressed, a situation that is likely to lead to poor compression; that the presence of a compression model reduces the buffer space available to evaluate the query; and that coding and decoding must have low processor overhead so as not to eliminate the benefits of reduced data transfer times.

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, and so on) 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 [11,12].

Semi-static compression has been successfully integrated into text information retrieval systems, resulting in savings in both space requirements and query evaluation costs [12,13,14,15,16]. The compression techniques used are relatively simple – Huffman coding for text, and integer coding techniques [15] for indexes – but the savings are dramatic. Index compression in particular is widely used in commercial systems, from search engines such as Google to content managers such as TeraText. Moreover, integer coding is extremely fast. We have shown [14] that even the cost of transferring data from main-memory to the on-processor cache can be reduced through appropriate use of compression based on elementary byte-wise codes.

However, compression has not traditionally been used in commercial database systems [4,7], and has been undervalued in query processing research [3]. Earlier papers investigated the benefits of compression in database query evaluation theoretically [6,7,8], and only in the last few years have researchers reported compression being incorporated into database systems [4,5,9].

Most of the research in this area has focused on reducing storage and query processing costs when data is held compressed. Graefe et al. [6] recommend compressing attributes individually, employing the same compression scheme for all attributes of a domain. Ng et al. [7] 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. Ray et al. [8] compared several coding techniques (Huffman, arithmetic, LZW, 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. [5] described a page-level compression algorithm that allows decompression at the field level. However, like the scheme described by Ng et al. [7], this technique is only useful for records with low cardinality fields. Westmann et al. [9] used compression at the attribute level. For numeric fields they used null suppression and encoding of the resulting length of the compressed integer [17]. 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. [4] used the same scheme as Westmann et al. 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, Westmann et al. [9] 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.

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 [3].

3 External Sorting

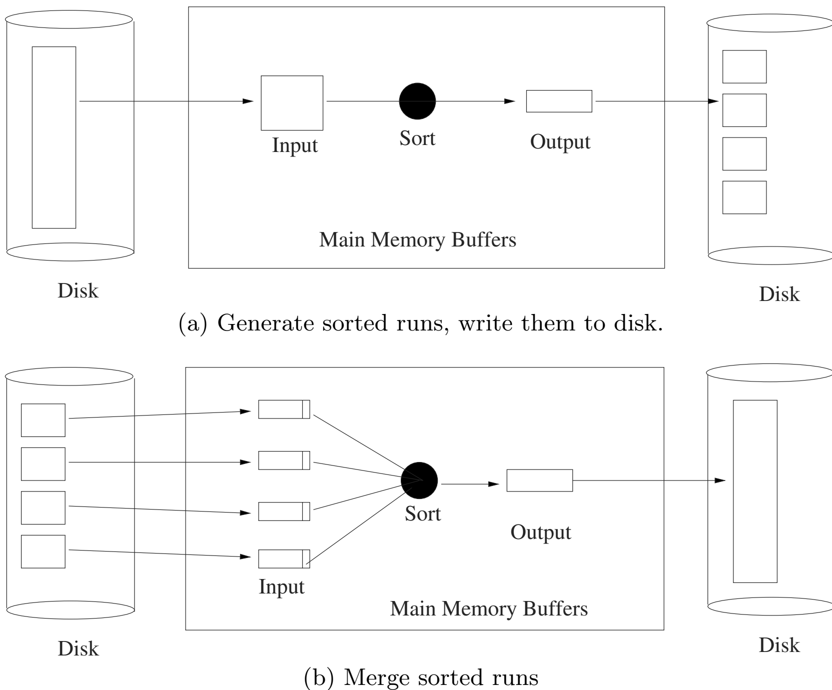
External sorting is used when the data does not fit into available memory. It is of general value for sorting large files, but is of particular value in the context of databases, where a machine may be shared amongst a large number of users and queries, and per-query buffer space is limited. External sorting has two phases [18,19], as below. The process assumes that a fixed-size buffer is available, which is used for sorting in the first phase and for merging in the second. The process is illustrated in Figure 1, and is described in detail by Knuth [20].

Phase 1: Process buffer-sized amounts of data in turn. The following is repeated as many times as necessary:

1. Fill the buffer with records from the relation to be sorted.
2. Sort the records in memory.
3. Write records in sorted order into new blocks, forming one sorted run.

Phase 2: Merge all the sorted runs into a single sorted list, repeating until all runs have been processed:

1. Divide input buffer space amongst the runs, giving per-run buffers, and fill these with blocks from runs.
2. Using a heap, find the smallest key among the first remaining record 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



(a) Generate sorted runs, write them to disk.

(b) Merge sorted runs

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

There are many variants on these algorithms. One 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 then to merge the runs hierarchically, which however incurs significant penalties in data transfer, and should be avoided if at all possible. The Unix command-line sort utility takes this approach. (We do not test hierarchical merging in our experiments.) Another 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.

4 External Sorting with Compression

By incorporating compression into the sort algorithm, we aim to reduce the time taken to sort due to better use of memory, reduced transfer costs, and generation of fewer runs. The two key questions to answer when integrating compression into external sorting are, first, at what stage should the data be compressed, and, second, what compression technique to use.

Considering the first of the key questions, 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.

In this approach, external sorting with compression proceeds as follows. Referring to Figure 2, assume that we have an input buffer of size A and an output buffer of size B, and that compression model size is M.

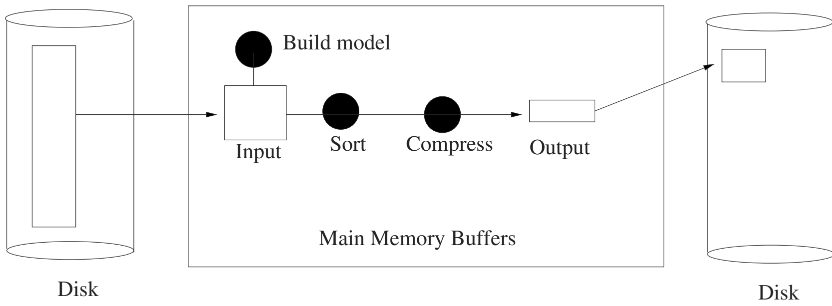
Phase 1: Build the compression model.

The arrangement of buffers in shown in Figure 2(a). The input buffer has capacity $A - M$ for records.

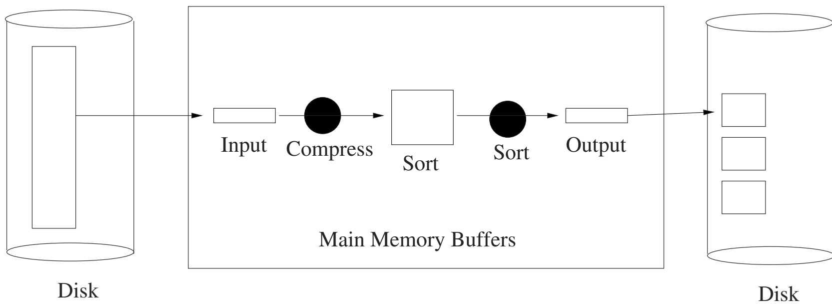
1. Fill input buffer with records.
2. Build a model based on the symbol frequencies in these records.

Phase 2A: Generate the first compressed run.

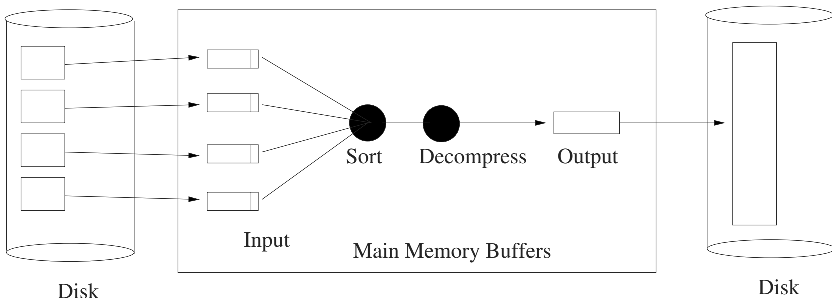
1. Sort the keys of the records in the input buffer.
2. In sorted order, compress the records then write them to disk as a sorted run.



(a) Generation of first compressed run.



(b) Generation of subsequent compressed runs.



(c) Merge of compressed runs.

Fig. 2. 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.

Phase 2B: Generate the remaining compressed runs.

The arrangement of buffers is shown in Figure 2(b); note that the data can no longer be read directly into a buffer for sorting, as it must first be compressed. The input and output buffers are of size $B/2$ each, and the sort buffer is of size $A - M$.

Repeat the following until all data has been processed:

1. Fill the input buffer with data.

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 the records in the sort buffer.
4. In sorted order, write the compressed records to disk, forming one run.

Phase 3: Merge all the runs into a single sorted list.

The arrangement of buffers in shown in Figure 2(c). The input buffer is of size $A - M$, the output buffer is of size B. Note that the model required for decoding may be smaller than that required for encoding.

1. Divide the input buffer space amongst the runs and fill with data from the compressed runs.
2. Find the smallest key among the first remaining record in each buffer, and move the corresponding record to the first available position of the output block and decompress.
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.

In this algorithm, the sort key must be left uncompressed, and to simplify data management each compressed record should be prefixed with a bytelength. In 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.

5 Compression Techniques for External Sorting

The second key question of this research program is choice of compression technique. As outlined earlier, "off the shelf" compression systems (with one exception, XRAY, discussed below) do not satisfy the specific constraints of this application.

Several observations can be made. We need to investigate semi-static coding techniques that can be used in conjunction with a model based on inspection of only part of the data; 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 [12], which is too slow for this application. Bytewise codes are much faster than bitwise codes [14], but may lead to poor compression efficiency. 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 bufferload of data. Bytewise codes are another option. These are discussed below. Another choice would be to use XRAY [21], in which an initial block of data is used to train up a model. Each symbol, including all unique characters, is then allocated a bitwise code. XRAY provides high compression efficiency and fast decompression; in both respects it is 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 efficiency depends on block size, which is around 100 Kb in standard configurations; with small blocks no size reduction is achieved.) However, the training process is much too slow. Development of new XRAY-based compression techniques for this application is a topic for further research.

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 18.5 Mb and 37 Mb buffers. In this context, model sizes need to be restricted to at most a couple of megabytes.

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. Using variable-length symbols requires an XRAY-like training process. 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 (528 Kb for the decode part and 264 Kb for the additional encode part).

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 [22].

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 [14]. In these 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 bigrams'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 uncompressed sort key and record length). The model sizes are identical to those for Huffman coding of bigrams above.

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 [23] 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. Determining an ideal set of quadgrams is almost certainly NP-hard. However, the frequencies are in any case only an approximation, as only part of the data has been inspected. Also, 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.

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.

6 Results

To test the effect of compression on external sorting, we implemented a fast external sorting routine, and added as options the three compression schemes described above. Runs were sorted with the best implementation of sorting we able to locate; as part of a separate project we are investigating high-performance sorting algorithms [24,25].

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. The larger was 37 Mb, chosen as 36 Mb for data plus 1 Mb for housekeeping; the smaller was half that, 18.5 Mb. These choices were arbitrary.

For data, we required a large number of records representing a realistic task. We used a log derived from a web cache, in which each line includes information such as file size, time and date, and HTML page request. Data volumes tested ranged from 100 Mb to 10 Gb of distinct records. The task was to sort these on one of the numerical fields.

All experiments were carried out on an Intel 1 GHz Pentium III with 512 Mb of memory running the Linux operating system. Other processes and disk activity were minimised during experiments, that is, the machine was under light load.

Tables 1 and 2 show the effect that incorporating compression into external sorting has on elapsed time and temporary disk requirements. 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 Figure 3, 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, which can clearly 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 common-quadgram coding, 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.) However, because Huffman coding

Table 1. Results for sorting with 18.5 Mb of buffer space, using no compression and using three alternative compression techniques. Results shown include time to sort and temporary space required.

	No compression	Huffman	Bigram	Quadgram
Build model (sec)	—	0.34	0.26	2.84
Generate runs (sec)	10.62	14.37	11.29	15.69
Merge runs (sec)	12.85	19.31	14.99	13.27
Total time to sort (sec)	23.47	34.02	26.54	31.80
Comparative (%)	100.0	145.0	113.1	135.5
Number of runs	6	5	5	5
Size of runs (Gb)	0.101	0.066	0.078	0.057
Comparative (%)	100.0	64.6	76.5	56.5

(a) Results for sorting 100 Mb of data with 18.5 Mb of buffer space

	No compression	Huffman	Bigram	Quadgram
Build model (sec)	—	0.33	0.26	2.84
Generate runs (sec)	109.23	134.49	115.68	146.83
Merge runs (sec)	176.24	197.58	161.82	142.09
Total time to sort (sec)	285.47	332.41	277.76	291.76
Comparative (%)	100.0	116.4	97.3	102.2
Number of runs	56	39	46	36
Size of runs (Gb)	0.976	0.641	0.757	0.561
Comparative (%)	100.0	65.7	77.6	57.5

(b) Results for sorting 1 Gb of data with 18.5 Mb of buffer space

	No compression	Huffman	Bigram	Quadgram
Build model (sec)	—	0.34	0.26	2.83
Generate runs (sec)	1305.25	1562.52	1262.95	1699.38
Merge runs (sec)	6140.86	4736.35	4769.87	3237.10
Total time to sort (sec)	7446.11	6299.21	6033.08	4939.32
Comparative (%)	100.0	84.6	81.0	66.3
Number of runs	568	394	462	372
Size of runs (Gb)	9.921	6.584	7.742	5.848
Comparative (%)	100.0	66.4	78.0	58.9

(c) Results for sorting 10 Gb of data with 18.5 Mb of buffer space

and common-quadgram coding achieve greater compression than does bigram coding, the sort times decrease faster as database size grows. This is most noticeable in the upper graph in Figure 3, which shows that, for smaller volumes of data, compression and decompression speed is the dominating factor; and that at larger volumes, the amount of compression achieved becomes the dominating factor in determining the sort time.

Despite the greater compression achieved by Huffman coding in comparison to bigram coding, the latter is always faster. This confirms that bytewise codes are much more efficient, with the loss of compression efficiency more than com-

Table 2. Results for sorting with 37 Mb of buffer space, using no compression and using three alternative compression techniques. Results shown include time to sort and temporary space required.

	No compression	Huffman	Bigram	Quadgram
Build model (sec)	—	0.66	0.51	6.02
Generate runs (sec)	8.39	13.41	12.17	15.13
Merge runs (sec)	12.79	19.37	15.09	13.20
Total time to sort (sec)	21.18	33.44	27.77	34.36
Comparative (%)	100	157.9	131.1	162.2
Number of runs	3	3	3	3
Size of runs (Gb)	0.101	0.0652	0.0773	0.0561
Comparative (%)	100	64.3	76.3	55.3

(a) Results for sorting 100 Mb of data with 37 Mb of buffer space

	No compression	Huffman	Bigram	Quadgram
Build model (sec)	—	0.66	0.52	5.99
Generate runs (sec)	108.67	146.36	126.43	143.71
Merge runs (sec)	168.81	213.62	177.93	155.36
Total time to sort (sec)	277.49	360.65	304.88	305.06
Comparative (%)	100	130.0	109.9	110.0
Number of runs	28	19	23	18
Size of runs (Gb)	0.976	0.640	0.752	0.552
Comparative (%)	100	65.5	77.0	56.6

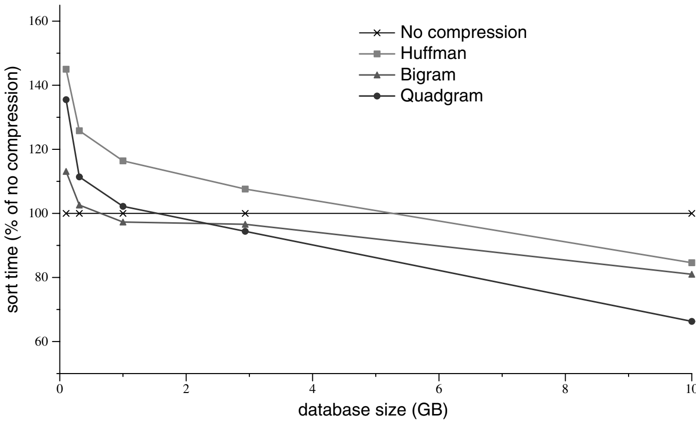
(b) Results for sorting 1 Gb of data with 37 Mb of buffer space

	No compression	Huffman	Bigram	Quadgram
Build model (sec)	—	0.66	0.51	5.98
Generate runs (sec)	1291.94	1675.52	1395.19	1593.98
Merge runs (sec)	2852.07	2442.40	2245.25	1796.71
Total time to sort (sec)	4144.02	4118.58	3640.95	3396.66
Comparative (%)	100	99.4	87.9	82.0
Number of runs	287	193	226	181
Size of runs (Gb)	9.918	6.569	7.694	5.751
Comparative (%)	100	66.2	77.6	58.0

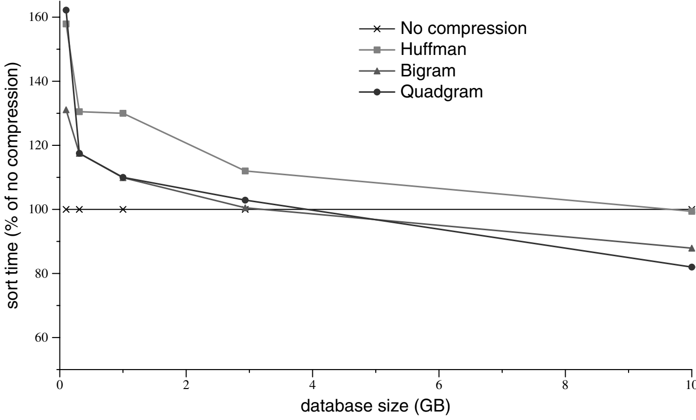
(c) Results for sorting 10 Gb of data with 37 Mb of buffer space

pensated for by the gain in processing speed. The common-quadgram method had both better compression efficiency and high processing efficiency, and thus was for large files superior to the other methods.

The tables also include the size of the resulting runs, giving an indication of the amount of compression achieved. Because we are making a number of compromises (compression and decompression must be fast, and the model must not consume too much memory), only modest compression was achieved. Also, 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



(a) Smaller buffer



(b) Larger buffer

Fig. 3. Sort times as a percentage of the time to sort without compression. (a): with 18.5 Mb of buffer space. (b): with 37 Mb of buffer space.

by the decoder. 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 1(c) and 2(c), using 36 Mb to build the model instead of 18 Mb only resulted in an extra 1 to 2 percent decrease in size.

Even though the degree of compression is relatively small, from Table 2(c) for bigram coding, we can see that a 22.4% saving in space due to the use of compression has resulted in a 12.1% saving in time. From Table 1(c) for bigram coding, a 22.0% saving in space has resulted in a 19.0% saving in time. For common-quadgram coding, a 41.1% reduction in the size of the runs has

resulted in a 33.7% reduction in the sort time. It seems clear that more effective compression should lead to further reduction in costs in both space and time.

7 Conclusions

We have developed new compression methods that accelerate external sorting for large data files. The methods are simple; the most successful is based on the expedient of identifying common quadgrams and replacing characters and quadgrams by bytewise codes. The gain is greatest when memory is limited, showing that the reduction in merging costs is a key reason that time is saved. Even though the compression gains were only moderate, significant reductions in costs were achieved.

For the largest file 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 modelling, compression, and decompression, and low memory use – continue to be met. It is also likely that similar techniques could accelerate other database processing tasks, in particular large joins. That is, our results indicate that compression of this kind could be used to reduce costs for a range of applications involving manipulation of large volumes of data.

Acknowledgements. This research was supported by the Australian Research Council.

References

1. Zobel, J., Williams, H.E., Kimberley, S.: Trends in retrieval system performance. In Edwards, J., ed.: Proceedings of the Australasian Computer Science Conference, Canberra, Australia (2000) 241–248
2. Boncz, P.A., Manegold, S., Kersten, M.L.: Database architecture optimized for the new bottleneck: Memory access. In: The VLDB Journal. (1999) 54–65
3. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surveys **25** (1993) 152–153
4. Chen, Z., Gehrke, J., Korn, F.: Query optimization in compressed database systems. In: Proceedings of ACM SIGMOD international conference on Management of Data, Santa Barbara, California, USA (2001) 271–282
5. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, IEEE Computer Society (1998) 370–379
6. Graefe, G., Shapiro, L.: Data compression and database performance. In ACM/IEEE-CS Symposium On Applied Computing (1991) 22–27
7. Ng, W.K., Ravishankar, C.V.: Relational database compression using augmented vector quantization. In: Proceedings of the Eleventh International Conference on Data Engineering, Taipei, Taiwan, IEEE Computer Society (1995) 540–549
8. 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)

9. Westman, T., Kossmann, D., Helmer, S., Moerkotte, G.: The implementation and performance of compressed databases. ACM SIGMOD Record **29** (2000)
10. Moffat, A., Zobel, J., Sharman, N.: Text compression for dynamic document databases. IEEE Transactions on Knowledge and Data Engineering **9** (1997) 302–313
11. Larmore, L.L., Hirschberg, D.S.: A fast algorithm for optimal length-limited Huffman codes. Journal of the ACM **37** (1990) 464–473
12. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Second edn. Morgan Kaufmann, San Francisco, California (1999)
13. Bell, T.C., Moffat, A., Nevill-Manning, C.G., Witten, I.H., Zobel., J.: Data compression in full-text retrieval systems. Journal of the American Society for Information Science **44** (1993) 508–531
14. 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. (2002) 222–229
15. Williams, H.E., Zobel, J.: Compressing integers for fast file access. Computer Journal **42** (1999) 193–201
16. Zobel, J., Moffat, A.: Adding compression to a full-text retrieval system. Software Practice and Experience **25** (1995) 891–903
17. Roth, M., Horn, S.V.: Database compression. ACM SIGMOD Record **22** (1993) 31–39
18. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems Implementation. First edn. Prentice Hall (2000)
19. Ramakrishnan, R., Gehrke, J.: Database Management Systems. Second edn. McGraw-Hill (2000)
20. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, Massachusetts (1973)
21. Cannane, A., Williams, H.: A general-purpose compression scheme for large collections. ACM Transactions on Information Systems **20** (2002) 329–355
22. Moffat, A., Turpin, A.: Compression and Coding Algorithms. First edn. Kluwer (2002)
23. Ramakrishna, M.V., Zobel, J.: Performance in practice of string hashing functions. In: Proceedings of the Databases Systems for Advanced Applications Symposium, Melbourne, Australia (1997) 215–223
24. Sinha, R., Zobel, J.: Efficient trie-based sorting of large sets of strings. In Oudshoorn, M., ed.: Proceedings of the Australasian Computer Science Conference, Adelaide, Australia (2003) 11–18
25. Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. In Ladner, R., ed.: Proceedings of the ALENEX Workshop on Algorithm Engineering and Experiments, Baltimore, Maryland (2003)