Performing Joins without Decompression in a Compressed Database System

S.J. O'Connell*, N. Winterbottom
Department of Electronics and Computer Science,
University of Southampton, Southampton, SO17 1BJ, UK
* Corresponding author: soc@ecs.soton.ac.uk

Abstract

There has been much work on compressing database indexes, but less on compressing the data itself. We examine the performance gains to be made by compression outside the index. A novel compression algorithm is reported, which enables the processing of queries without decompressing data needed to perform join operations in a database built on a triple store. The results of modelling the performance of the database with and without compression are given and compared with other recent work in this area. It is found that for some applications, gains in performance of over 50% are achievable, and in OLTP-like situations, there are also gains to be made.

1 Introduction

1.1 Background

The benefit of compressing indexes in a database has long been established, but recent work has focussed on compressing the data itself. In one paper, Westmann et al [Wes00] recommended lightweight compression techniques in the context of a TPC-D benchmark database, but queried the benefit of compressing data in an OLTP environment. In other research, Chen et al [Chen01] focussed on compression-aware query optimization in a TPC-H database.

Both of these approaches deal with decision support queries in a 'traditional' n-ary relational database with large numbers of records, which require heavy processing in query optimization and execution. In this research, we focus on some more fundamental issues which come to light when an alternative database architecture is employed, in this case, a binary relational database. Here, information concerning relationships is held directly in a triple store, with data held in a lexical store (see below). This architecture greatly reduces the need for expensive join processing, but the trade-off is extensive processing of triple store records, and the question is then whether compression can benefit this processing.

A new compression algorithm has been developed for the database. Using this, records are compressed when initially inserted into the triple store, but from then on, processing can be carried out efficiently without needing to decompress the records again. We describe a modelling exercise carried out to explore the extent of the performance improvement, and suggest that although there are theoretical limitations, there is often significant gain to be made, even for OLTP type queries.

The triple store is not a new concept, see for example [Shar78, Shar88], but continues to draw interest [TriStarp] with a new commercial database [Sen01] now on the market. In the present implementation, a triple store is the central repository for information concerning relationships in both the data and the meta-data, and this is coupled with a lexical store which holds each separate data value just once. The number of triples in the triple store is related to the number of instances of each field in the database, and the structure means that there is indexed access to every field in the database.

To guide the design process, a tool has been developed to model the performance of the emerging implementation. The approach taken was to use the facilities provided by a spreadsheet, rather than to develop a separate bespoke modelling program. The performance model is constructed around the interface at which commands are submitted to the database to enter or retrieve entities and their attributes. Applications may be developed within the model by assembling sequences of the operations, and the model is then used to predict behaviour as various parameters are altered.

1.2 Related Work

Many efforts in the context of relational databases have dealt with compression in the index. Here, successive entries are sequenced, and various techniques such as prefix compression and suffix compression have been employed, as described in standard works such as Gray and Reuter [Gray93] or Ramakrishnan [Ram00].

With respect to compression of non-index data in a database, techniques such as Huffman coding [Huff52], Lempel-Ziv [Ziv77] work well for compressing certain types of data, such as medical images [Kar97] or text [Mof97], but are not applicable to compressing string fields in a database due to the CPU cost. There are other algorithms for compressing numeric data, and these are well described in the papers mentioned earlier, [Wes00] and [Chen01]. The reader is therefore referred to them for further related work.

2 Compression in Databases

The most obvious reason to consider compression in a database context might seem to be to reduce the space required on disk. However, as disk space becomes rapidly less expensive, this is no longer such an important concern. The more important issue is to see whether the processing time for queries can be reduced by limiting the amount of data that needs to be read from disk to satisfy the query. By compressing data, can the number of blocks to be read be reduced?

Speed-up can come from reducing the number of disk I/Os, (as long as the CPU cost of achieving this is not too high) and frequently the only way to do this is by reducing the number of accesses required in traversing the index. The height of the index tree is given by a logarithmic formula:-

$$H = \frac{Ln(RBlkNum)}{Ln(INum)}$$

where H is the height of the tree, RBlkNum is the number of blocks containing data records, and INum is the number of index entries/block. In other words, there is an exponential relationship between H and both RBlkNum and INum.

One option is therefore to increase INum by compressing index entries, which is the route taken in many databases today. The second option, in which we are interested, is to decrease RBlkNum, by compressing the data itself. In order to reduce the height of the index tree by one, and thus eliminate one disk I/O, we could calculate

$$H'' - H' = \frac{Ln(RBlkNum'')}{Ln(INum'')} - \frac{Ln(RBlkNum')}{Ln(INum')} = 1$$

If we assume that INum, the degree of index compression, is the same in both cases, this simplifies to

$$Ln(RBlkNum'') - Ln(RBlkNum') = Ln(INum)$$

or

$$\frac{RBlkNum''}{RBlkNum'} = INum$$

So if the number of index entries per block were, say, 100 (a relatively low figure), then to achieve a consistent performance improvement by reducing the number of disk accesses by one for all database sizes, a compression factor of over 100 is needed, a fairly aggressive target!

This sort of analysis might lead one to abandon interest in data compression immediately, but in fact things are not quite so simple, as the following work will show. Nevertheless, the basic facts above are worth bearing in mind and will be discussed later.

3 Towards a Compression Algorithm

In the triple store, sorting ensures that the first part of the triple will be repeated for successive entries, which immediately suggests scope for compression. Each entry in the triple store contains three parts: the identity of the relationship (RelId), the identity of the entity that the relationship runs from (FromId) and the identity of the entity that the relationship runs to (ToId). The triples are stored in sorted order in two ways: <RelId, FromID, ToId> and <RelId, ToId, FromId>. Each logical triple is therefore actually stored twice, and query processing is optimized to use the appropriate sort order depending on the search criteria. As entity sets increase in size, there are increasing numbers of triples for each relationship type.

The three identities are each currently represented by a four-byte integer, which gives a symmetrical implementation. The triple store is accessed by means of a B-Tree type of index. It is worth noting that while compression in indexes can be lossy (if index entries are over-compressed, the situation can be recovered by retrieving additional data blocks), in the triple store itself, any algorithm must not lose information.

3.1 The Scope for Compression in the Triple Store

Most queries applied to the database will result in the direct retrieval of one or a small number of triples by

means of the index. The only queries where this is not the case, and a range of triples is retrieved in sequence, are where the database is being searched to perform a join on a non-key field (in n-ary terms). The DBMS contains its own cache, and the size of this will affect the number of blocks that must be read from the disk. Cache size and block size are parameters in the performance model. If the triples can be reduced in size, more triples can be held in a block. The size of the index is therefore reduced, and this is also modelled.

Two observations are worth making at this point:

1) The number of different RelIds in a given database is quite small. In the database described below, there are fewer than one hundred different RelIds. The Id allocation algorithm is designed to pack numbers into as few low-order bytes as possible, and it is likely that there will be 'spare' bytes at the start of the RelId that are never used.

2) A 16k block can store about 1000 uncompressed triples at 70% occupancy. The triples are sorted, and if the Ids are allocated so that 50% of the numbers in a given range are actually used, the range of FromIds in a block could be as little as 2000 (Hex 7D0), needing only one and a half bytes. This figure is even less if a smaller block size is used. Within one block, therefore, it is quite likely that the high order bytes will be repeated for many successive triples.

3.2 Possible Approaches

Two contrasting approaches were considered. The first was typified by an algorithm which made use of a 'compression byte' prefixed to the triple. The bits in the prefix are set to indicate which bytes in the present triple are repeated from the previous triple, and are therefore omitted. Application of the algorithm to a sample triple store indicated that the store could be compressed to about 60% of its original size.

However, there is a major disadvantage to this approach, which applies in some degree to compression in most databases. In order to carry out any processing, the block will need to be decompressed, as the offset of a record depends on the size of the previous records in the block. While the reduction in size potentially gives a significant reduction in I/O, the intensity of processing in the triple store, where relationships are followed from one entity to another, led to consideration of another algorithm.

The second approach was designed to permit the processing of a block in its compressed state. The principle is that once the block has been initially compressed, subsequent operations, particularly binary searches, can be performed on the block in its compressed state, *without needing to decompress it every time*, which will clearly benefit performance considerably. The algorithm used to achieve this was termed 'the block mask algorithm'.

3.2.1 The Block Mask Algorithm

At the beginning of each block, a mask is stored, indicating which of the twelve bytes in each triple are *not* constant throughout the block, as shown in Table 1. The next record in the block contains a full triple, a 'starter record', with the values of the fixed bytes in the appropriate position. The remainder of the block stores short fixed length records containing only the bytes that vary. Each block will contain a different mask, so that the length of the fixed length records in each block might be different.

Mask	0001 0011 0111
Starter Triple	0010 4000 5000
Subsequent Triples	345987 (= 0013 4045 5987)
…	446678 (= 0014 4046 5678)
…	… and so on …

Table 1 – Example of the Block Mask Algorithm

When a block is retrieved into the DBMS, it is then possible to use the mask and the starter record to reconstruct any individual triple without the need to decompress the whole block. As described above, the algorithm works in terms of bytes. A further refinement is possible to store only the bits that change, rather than whole bytes, which allows further compression to be achieved.

3.2.2 Evaluation of the algorithm

Application of this algorithm can lead to compression down to a third of the original size of the triple, or a quarter if bit level compression is being used. Triples are compressed when being placed in the triple store. For retrieval, the search string is compressed, the required triple is located in the compressed block, typically using a binary search, and the selected triple is decompressed when located. The block mask algorithm only needs a few lines of code to pick up the mask and the starter record, and then apply these to the selected triple.

There are further detailed decisions that a final implementation would require. For example, it would be possible to insist that each block contained only triples relating to one RelId. This would enhance compression, and if data sets are large so that one RelId spans several blocks, would lead to a worthwhile saving. For a small database, however, this could result in an unnecessary proliferation of blocks, adversely affecting the performance. This sort of refinement is beyond the current study, however.

4 Modelling the Performance Improvement

4.1 The Database

For this exercise, a database for a wholesaler buying in goods from a number of suppliers, and shipping smaller quantities to various customers was used. In conventional n-ary database terms, the database had 8 tables, with a number of relationships between them. The scenario assumed was that a variety of mainly OLTP transactions would be carried out, at normal volumes. All queries in the present experiments were read-only.

The 8 tables represented customers, suppliers, orders, products and so on. The average number of fields per table was taken as 10. This translates into a triple store database with 8 entity sets with 80 attribute sets, requiring 80 different entity-attribute relationships. The foreign key relationships between the tables translate into 10 entity-entity relationships. Thus 90 different relationships were required.

In considering the compression ratio achievable, it is necessary to consider the range of values for various aspects. The following discussion is in terms of a triple store sorted in the primary order, that is, on RelId and FromId.

1) **RelId**'s: For this database, 90 different ids are required, plus the small number required to handle meta-data. The Ids for this could therefore be handled within one byte. For any database other than the smallest, however, most blocks will contain triples relating to only one relationship. The RelId will therefore compress out completely, and be held only in the block mask.

2) **FromId**'s: Following the discussion in 3.1 above, two bytes will be needed, which gives a range of 64k for the values of the Ids in one 16k block. (If a smaller block size is used, the range is reduced. However, the greater compression is not significant unless very small blocks are used, and the increase in processing then outweighs the benefit. A 16k block size was used throughout this series of experiments.)

3) **ToId**'s: If the database is sorted on the first two fields, then the values in the ToIds will be randomly scattered across the range for each set. For a database up to one million triples (which corresponds to about 50,000 entities per set or 25 Mb of actual data), two bytes will suffice to cover the range of ids; for a database up to 200 million triples and beyond (about 1 million entities/set, 500 Mb), three bytes will be needed.

The 12 bytes required before compression can therefore be reduced to four or five bytes after compression for this scenario. If the compression is carried down to the bit level, then the FromId could be held in 12 bits, and the smaller ranges for the ToId could also be held in 12 bits, so the compressed triple could then be just three bytes.

In the direction FromID to ToId, each triple captures one instance of a m:1 relationship, so that when the triple store is sorted in the primary order, the third field will not have any particular sequence, as reflected above. In the inverse sorted triple store, the order is RelId, ToId, FromId, which represents the relationships in the 1:m order. Successive triples may now have identical fields in both the RelId and the ToId, and the FromId will be in sorted order, so that triples can be further compressed. To model this, however, would require more detailed examination of the distribution of data in the various domains, and this was not deemed appropriate to the present level of analysis.

4.2 Establishing the Model

The size of the cache has a critical impact on performance. As the cache size increases, more levels of index and more data records can be held in the cache, and the overall performance will improve. Cache size was therefore varied to see the effect of this as it interacts with compression.

The vast majority of normal queries involve searches where the RelId is known and either the FromId or the ToId is also known. In either of these cases, the blocks can be accessed directly through the index, if both sort orders are held (RFT and RTF). The main interest is therefore in the retrieval time for such queries. Access to the triple store which is primarily

sequential with just a few index accesses is required only for a query where, in traditional RDB terms, there is no foreign key linking two tables, such as might be used in some decision support enquiries. An example in the Wholesale database would be "Find me the suppliers and customers who share a postcode". In this case, all or part of the triple store has to be scanned looking for matches. Compression will obviously speed these searches, and this was also considered.

5 Results

5.1 Direct Access

The model was run for the wholesale database. The cache size was varied, and in each case, results were recorded for various sizes of database, both with and without compression. Figures 1 and 2 show the results for two different cache sizes. The graphs show the average number of disk accesses required for the retrieval of a triple. A database operation will often require a number of triples to be retrieved, so that variations in the number of disk accesses will be evened out, and the average is a useful figure to work with. The complex interaction between index size, database size and cache size yields local variations, such as that in Figure 2, where the first two points for the compressed database both show the database almost entirely in the cache, but there is a broad similarity in the results.

The effect of increasing the cache by a factor of 4 can be seen in the reduction of the number of accesses by a half to three quarters of one access depending on the size of the database. Increasing the cache size would be expected to improve performance, and the model helps quantify the degree of improvement.

The particular interest, however, is in the effect of compression. Each graph shows the effect of this, which is to reduce the number of accesses by a significant amount ranging from a quarter to three quarters of an access. This leads to an improvement by a factor of almost two in smaller databases, dropping to 1.25 in large databases. This result corresponds to the OLTP situation, where each query looks for a record which may be unrelated to the previous one, and stands in contrast to the conclusions drawn by Westmann et al, who do not expect compression to improve the performance of OLTP-style applications.

5.2 Sequential Access

For queries which do not involve a significant degree of index access, then compression produces a straightforward benefit. Each retrieved block contains more triples, in direct proportion to the compression ratio, and the model confirms this. One therefore sees an improvement of 2:1 or better, and this is very much in line with the results from Westmann et al and Chen et al, which both deal with decision support situations.

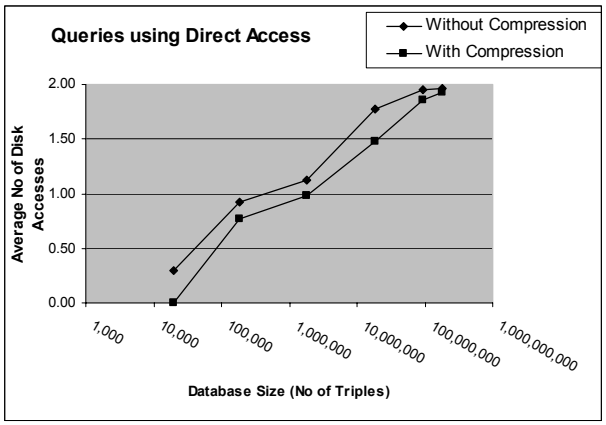

Figure 1. Triple retrieval time with 256k cache

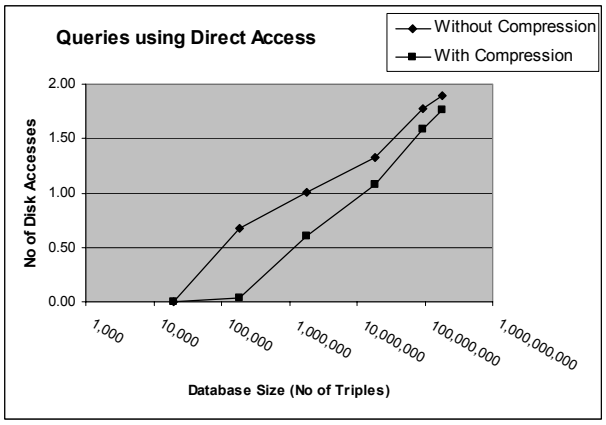

Figure 2. Triple retrieval time with 1 Mb cache

6 Conclusions

The operations that would be carried out by joins in a conventional database are replaced by operations in the triple store, so that any reduction in the number of accesses has a direct effect on performance, whether for a single query or for a sequence of related operations. The block-mask algorithm permits processing to be carried out on compressed data, yielding a very efficient join mechanism. The effect

of this has been modelled, and shown to produce significant benefit.

In the case of sequential operations which would be needed for decision support queries, the results obtained demonstrate an improvement by a factor of two. However, it has also been shown that this approach would benefit OLTP queries, giving a reduction in the number of disk accesses by a factor in the range of 1.25 to 2.

At present, the model takes no account of locality of reference, so is actually unduly pessimistic. One of the advantages of fully decomposing data in the current implementation is that related items will be stored in close proximity; because the triples are sorted, data is automatically clustered by dictionary subject matter. In other words, the whole of a binary relation will be tightly clustered. In practice, therefore, it is expected that the results would be better than predicted by the model.

Further work should include consideration of the additional effect of compression on the indexes. The uniformity of the implementation means that the same code is used to handle both the blocks in the triple store and in the index to the triple store. Any compression algorithm will therefore benefit both, and a further modelling exercise could capture the effect of this.

There is also the possibility of extending the degree of compression. The current assumption is that all data domains are large, but in practice, some are quite small. In the extreme case of a binary domain (e.g. Male, Female), compression down to one bit per triple is possible, as follows. If a block contains one relation, and if FromId's are densely packed, then the initial FromId can be held in the block header, as well as the RelId. If the rest of the block is considered as an array, each bit in the array could represent the monotonically increasing set of FromId's held in the block. Each bit could then be set to indicate whether the ToId took one or other of the two possible values. This would give a compression factor of almost 100 (12 bytes down to one bit). This could be generalised and implemented on a block by block basis. If in the range of one block, the third field only uses two bits, even if the potential domain is larger, the block could be compressed to this level while retaining the higher level advantages of the triple model. This degree of compression would have a major impact on the performance of all types of queries.

References

[Chen01] Z Chen, "Query Optimization in Compressed Database Systems", *ACM SIGMOD Record Vol 30, No 2*, pp 271-282, June 2001

[Gray93] J Gray & A Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993

[Huff52] D Huffman, "A Method for the Construction of Minimum Redundancy Codes", *Proc IRE, 40(9),* pages 1098-1101, Sept 1952

[Kar97] K Karadimitriou, J M Tyler, "Min-Max Compression Methods for Medical Image Databases", *SIGMOD Record,* Vol 26, No 1, March 1997

[Mof97] A Moffatt, J Zobel, "Text Compression for Dynamic Document Databases", *IEEE Transactions on Knowledge and Data Engineering*, Vol 9, No 2, March-April 1997

[Ram00] R Ramakrishnan & J Gehrke, Database Management Systems, McGraw Hill, 2000

[Sen01] "Sentences DB", based on the Associative Model of Data, from Lazy Software, www.lazysoft.com

[Shar78] G C H Sharman and N Winterbottom, "The Data Dictionary Facilities of NDB", *Proc 4th Int. Conf on Very Large Databases (VLDB)*, pp 186-197, IEEE 1978

[Shar88] G C H Sharman and N Winterbottom, "The Universal Triple Machine: a Reduced Instruction Set Repository Manager", *Proceedings of BNCOD 6,* pp189-214, 1988

[TriStarp] TriStarp Web Site: http://www.dcs.bbk.ac.uk/~tristarp

[Wes00] T Westmann, D Kossmann, S Helmer, G Moerkotte, "The Implementation and Performance of Compressed Databases", *ACM SIGMOD Record Vol 29, No 3*, pp 55-67, September 2000

[Ziv77] J Ziv, A Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory,* 22(1), pages 337-343, 1977