Andreas Dehmel

A Compression Engine for Multidimensional Array Database Systems

Herbert Utz Verlag · Wissenschaft
München

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist
bei Der Deutschen Bibliothek erhältlich

Zugleich: Dissertation, München, Techn. Univ., 2002

ISBN 3-8316-0139-9

Printed in Germany

Herbert Utz Verlag GmbH, München
Tel.: 089/277791-00 – Fax: 089/277791-01

Contents

Chapter 1

Introduction

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say.

J.R.R. Tolkien, *The Lord of the Rings*

1.1 Motivation

Integrating compression in Database Management Systems (DBMS) has not received very much attention so far. The main reason for this is that the benefits of compression in traditional, relational DBMSs (RDBMS) are rather limited due to the nature of relational data which typically consists of small units like numbers or short strings, which have a size of several bytes rather than kilo- or even megabytes. Compressing these small units individually results in very poor space savings, whereas grouping units into compounds and then compressing them improves the space savings per unit considerably, but burdens the system with much higher random access times, as will be shown below. This is often unacceptable since the resulting system is usually expected not to be noticeably slower despite the compression/decompression overhead, but possibly even faster due to reduced data size which allows more information to be transfered in fewer IO operations.

Compression is always a tradeoff between savings in storage space and compression overhead, where on average complex compression algorithms save more space while taking longer, whereas simpler algorithms compress worse but are faster; it is therefore obvious that in order to minimize compression overhead, simple algorithms should be chosen. We will also see that fine access granularity, as is typically required for RDBMSs, is another point in favour of simple compression variants as well. In contrast to RDBMSs, array DBMSs (ADBMS) deal with much larger units which makes compression considerably

more efficient as well as more attractive for the space savings alone; it also encourages the use of more advanced compression techniques, even lossy ones which have never been an option in RDBMSs, in the same way that lossy compression is viable for images but not for text.

Compression aims to minimize the redundancy in data by finding a representation for the data that requires less storage. This is often an adaptive process that performs better the more data it processes, i.e. a large text compressed as a whole will usually have a higher compression ratio than any subset of the text, depending on the compression algorithm used. Analogously, while all data contained in a relational table may compress well in its entirety, the individual attribute values are typically too short to allow gathering any statistically meaningful information to aid compression. The techniques used for the compression of single attribute values are therefore very basic since more complex approaches like adaptive arithmetic coding can't accumulate enough statistical data to outperform the simpler and therefore faster techniques.

The alternative of grouping together attribute values to allow more efficient compression severely compromises execution time because accessing any value contained in such a compound structure requires uncompressing the whole, or at least all data up to the desired value, since compressed data normally does not allow random access any more. Moreover, updating data in a compressed format is not localized, i.e. changing a local part in the uncompressed data usually implies global changes of the compressed data, typically from the point corresponding to the start of the update of the uncompressed data to the end of the compressed data stream. Caching uncompressed compounds can alleviate this phenomenon in some cases, but introduces other problems like an increase of memory requirements as well as more complex IO and transaction management. Considering these points it comes as no surprise that there has been little research on compression in traditional DBMSs and the work done on the subject has focussed on relatively simple algorithms, some of which will be introduced in the *Related Work* section on page 10.

Not all DBMSs are relational, nor do all systems require as fine an access granularity, however. Array DBMSs deal with rastered data of varied dimensionality and differ considerably from traditional DBMSs both in terms of access granularity and data volume typically transfered to clients. Whereas the access granularity of RDBMs is in the area of bytes, for array DBMSs it is in the area of kilobytes or even megabytes, which makes these systems very interesting for compressed storage. The high data volume transferred to client applications also makes transfer compression a viable approach. Moreover, for multidimensional data there are often local correlations between data samples which can be exploited by compression, like for instance a rectangular area of uniform colour in an image which will compress better if the compression algorithm is aware of the 2D nature of the source data rather than being applied to a sequence of 1D data sequences. Things like this are usually resolved in a *model* (or transformation) layer which transforms the original data according to a data model into a different (possibly similar, i.e. lossy) representation which compresses better.

RDBMSs can only handle vector data efficiently, which can be translated into tuples containing coordinate attributes and value attributes. However, many kinds of data can be

modelled efficiently as multidimensional arrays, especially dense data, of course; but with ever increasing memory capacity, even modelling sparse data as multidimensional arrays becomes feasible, especially with the addition of compression, since sparse representation in the form of vectorized data is a kind of offset compression in itself. In [44], for instance, this development is quoted in the context of visualization to predict a move away from surface-oriented models (vector) to volume-oriented models (arrays) in a similar way as raster images have grown ever more popular compared to vector graphics in many cases, especially when it comes to digital representations of real world phenomena, typical examples of which are sampled analogue data such as images, volumetric data like tomograms, or simulation data (spatial or spatio-temporal, like for instance fluid flow or climate simulations). The disadvantages of dense modelling, foremost of all memory requirements, are becoming less relevant as more memory is readily available; in many cases the vector data used in e.g. visualization has to be calculated from a dense representation like a 3D data cube anyway (e.g. isosurface algorithms [34] which calculate a triangulated surface from 3D array data). At the same time the advantages of dense data modelling are becoming more and more attractive in many application areas:

- compact storage (at least for dense data), i.e. no coordinate overhead because the coordinates are implicitly given by the offset in the data and the linearization scheme;
- constant access time of arbitrary coordinates independently of the data distribution, which also includes finding neighbouring cells in constant time.

Returning to the example of visualization, volume rendering has considerable advantages compared to surface-oriented (= vector-oriented) approaches as it allows exploring internal structures [44]. In numerical simulation, using arrays rather than vectors for storage allows simpler and more efficient algorithms for the solution of partial differential equation systems. Using hierarchical grids makes it possible to use more sample points within areas of rapid change than in areas with little activity, thereby addressing the major problem of dense modelling; wavelet-based compression implicitly uses a similar approach, as will be shown later in section 3.3.

Naturally, dense modelling can't replace vector data entirely, especially when the coordinate system is not discrete but real-valued as in CAD systems or to a certain extent documents, although thanks to specialized compression techniques documents are actually becoming a borderline case [8]; it must be added, however, that rasterized documents are mostly of interest to digital libraries where the vector data is not available or never existed in the first place. Looking especially at the developments in visualization and numerical simulation, there is a clear trend away from vector data towards array data.

The object of this work was the implementation of a compression engine for the multidimensional Array DBMS $\mathcal{R}as\mathcal{D}a\mathcal{M}an$, which was originally developed at FORWISS and supports data of arbitrary dimensionality and base type. Data is modelled as multidimensional arrays, i.e. dense storage; the addition of compression capabilities to the DBMS also allows sparse data to be handled efficiently, however. There is a wealth of literature on

specialized compression, especially the compression of raster images, i.e. 2D arrays over a small number of possible base types, but there is scarcely any work on integrating these techniques into DBMSs or similar products dealing with multidimensional arrays. Furthermore, the approaches are usually restricted to a specific number of dimensions – foremost of all 2D for images – which requires generalization before they can be integrated into a truly *multidimensional* system. The purpose of this thesis is

1. the design of a generic compression framework for storage- and transfer-compression and its integration into the kernel of a multidimensional Array DBMS, $\mathcal{RASDAMAN}$ in this case;

2. the evaluation of compression classes and the inclusion of promising candidates in the compression framework. The main focus here lies on techniques developed in image compression, because many MDD show exactly the same properties exploited in image compression, foremost of all local smoothness caused by correlations between neighbouring cells;

3. provide compression algorithms with different properties dependening on the application. For transfer compression, (de)compression overhead is usually the decisive factor, whereas for long-term storage it is the compression ratio. For mostly read-only data an algorithm with asymmetric complexity may be ideal, such as (adaptive) dictionary techniques where compression can take considerably longer than decompression.

4. performance measurements on the resulting system and evaluation of the various compression classes with respect to the data types they are applied to and the scenarios they are used in (for instance storage compression vs. transfer compression).

The goal in integrating a compression engine into the DBMS kernel is to reduce storage requirements on one hand as well as transfer times on the other. As always in compression, the trade-off between storage reduction and compression overhead plays a central role and will be evaluated in more depth in sections 3.7 and 4.4.

1.2 Compression Basics

The fundamental idea of data compression is to find shorter – and in case of lossy compression approximate – representations for given data (= a sequence of symbols). Without compression, the number of bits required for a symbol is determined by the data type of the symbol, where typically only a few different types are supported by computer systems, regardless of the actual symbol values. While this property allows very fast data access, it usually takes up more storage than strictly required, for instance when only the 26 lower case characters appear in a data stream but 8 bits have to be used for storage.

In lossless compression there is a hard limit on achievable compression, determined by the so called *entropy* of the data. This part of information theory was introduced by

Shannon [48] in 1948 where the entropy H of a sequence S of independent, identically distributed symbols over an alphabet $\mathcal{A} = \{X_1, \ldots, X_m\}$ with probabilities $P(X_i)$ was defined as

$$H(S) = -\sum_{i=0}^{m} P(X_i) \log P(X_i). \tag{1.1}$$

The entropy represents the average amount of storage space per symbol (= *rate*) required to encode S losslessly. The unit of the storage space depends on the base of the logarithm function in equation (1.1); typically $\log_2$ is used, in which case the entropy is the average number of bits per symbol. Shannon proved that it is impossible for *any* compression algorithm to encode a given symbol sequence in fewer bits than specified by the entropy, provided the assumption about the independence of the symbols holds.

In many cases, the symbols are dependent, however. A typical example of this is of course text, where preceding symbols severely restrict the possible values of following symbols, for example the probability of a vowel following a "th" in an english text is considerably higher than that of another consonant. Symbols are often correlated in other cases as well, such as in "smooth" signals where the next symbol only differs from the preceding one by a small amount which can be coded in fewer bits than the actual symbol value. These dependencies are usually resolved via a *data model*, for instance "english text" or "smooth values" or "values lying approximately on a straight line" etc. The better a model matches a symbol sequence the better this sequence can be compressed, even well below the entropy based on the assumption of independent symbols. Therefore most modern compression techniques consist of two layers, a top layer which transforms the data according to a data model, and a bottom layer which actually compresses the transformed data; this architecture will be discussed in more depth in chapter 3.

There is a very limited number of traditional techniques in the bottom layer and even modern compression techniques rely on one of these for actual data compression. In contrast, there is a large number of data models in the top layer and most "new" compression techniques introduce new algorithms in the top layer only, this thesis being no exception. Traditional data compression techniques can be divided into four elementary techniques and two basic classes which may also be combined to improve the compression ratio (and often are so); I will only give a short overview here since they will be covered in much more depth in chapter 3.

Pattern-oriented techniques:

RLE: (*Run-Length Encoding*) compresses consecutive symbols of the same value. Very low complexity, but also relatively poor compression ratio for dense data;

Dictionary Techniques: find patterns in the symbol sequence and replace these literal patterns with references into a dictionary. Very high complexity during compression, but usually a good compression rate;

Variable-length coding:

Huffmann Coding: represents symbols with high probability with fewer bits than those with lower probability, thereby achieving data reduction. Because every symbol must be represented by an integral number of bits, only symbol probabilities which are (negative) powers of 2 can be modelled exactly;

Arithmetic Coding: the current state-of-the-art in variable length coding which recently superceded Huffmann coding as the de-facto standard. Arithmetic coding represents a symbol sequence of arbitrary length by a number in the unit interval $[0, 1[$ with arbitrary precision. By representing symbol probabilities as subintervals of the unit interval with a width proportional to their probability, arbitrary symbol probabilities can be modelled exactly;

An important property of these compression techniques is whether they are static or whether they can adapt to the data they are applied to. For example a dictionary technique could use a static dictionary or build the dictionary during operation; or the variable-length coding techniques may need to know the probability distribution in advance or they may adapt the probabilities according to the data automatically. In some cases, like text in a natural language, static variants are usually sufficient or actually better than adaptive ones because the sub-optimal "learning" phase of the adaptive process can be disposed of and the dictionary doesn't have to be stored with the compressed data. But in the majority of cases – especially when compressing binary data – adaptivity is an important criterion because no generally applicable dictionary or probability distribution exists. Regarding adaptivity, all dictionary techniques that are in common use today are adaptive. Huffman coding is very hard to use adaptively because the Huffman tree has to be fully materialized (and therefore constantly recalculated for adaptivity), making adaptivity very expensive. Arithmetic coding can be made adaptive with much less overhead, which is another substantial advantage over Huffman coding: even one of the first implementations of arithmetic coding was adaptive [63].

Typically, compression in the bottom layer is a combination of a pattern-oriented technique followed by variable-length coding, e.g. for a compact representation of references in dictionary coding like in the *ZLib* compression library [67] where dictionary offsets and lengths are Huffmann-coded.

1.3 Related Work

There are not many publications on the subject of compression in databases and the existing ones are based on mostly text-based databases like RDBMSs. As noted in section 1.1, the situation regarding compression in traditional, text based DBMSs differs considerably from that in an Array DBMS, where typically large blocks of contiguous data are processed and transferred in one go rather than single cell values. However, the basic goals of using compression in a DBMS are the same for all kinds: reduction of storage and in addition a potential IO speedup due to the reduced data volume. I will therefore discuss some of

the literature on compression in RDBMSs here. It must be noted that the older the literature gets the less relevant its conclusions usually are from today's perspective because what used to be an expensive compression technique at that time may well be considered light-weight compression today.

The newest of the publications introduced here is [16] and is actually more about database normalization than compression in the classical sense. The idea introduced there is to reduce the data volume by adding an indirection level for attributes which can have a limited number of different values, i.e. rather than encode the attribute value inside the table, the attribute becomes a reference into an additional dictionary table. The example given there is a table encoding computer chip specifications where one field of characters can take on the values `CMOS` and `TTL` only. These can be "compressed" by using a dictionary table containing these two possible values and transforming the original table by storing the offset into this dictionary table in place of the attribute value, which can be done in just 1 bit in this example. While some of the most widely used compression algorithms like LZ77 [64] and LZ78 [65] are based on the idea of a dictionary, it still seems odd to categorize transformations like the above as compression rather than database normalization. There is common ground between schema design and compression in that good schema design strives to minimize redundancy (at least when ignoring preaggregation), just like compression; nonetheless what this paper actually describes is much closer to schema design or even plain programming style than to (database) compression.

A paper covering actual compression in a RDBMS is [62] which introduces several compression techniques, discusses the ones chosen in more depth and concludes with measurements after integrating the compression engine into their AODB system. Low compression overhead and fine access granularity are stressed as points of primary importance from the beginning. Compressed tuples are divided into five fields with different properties, namely

1. values of fields compressed to constant length (no address calculations), e.g. using dictionary techniques with a known dictionary size;

2. lengths of all fields compressed to variable length (note that the length of this section is also of constant length);

3. values of uncompressed fields of constant length (this section is of constant length too);

4. values of fields compressed to variable length;

5. string values of `VARCHAR` fields; `CHAR` fields of fixed (maximum) length are converted to `VARCHAR` fields to ensure only the data actually required for the string is stored.

This separation aims to allow constant random access time by putting all fields of constant size in a block at the beginning. The actual compression techniques used are rather simple, e.g. integer numbers are compressed by storing the minimum number of bytes needed to represent the integer as length information, plus the actual bytes as values.

Since all length information is packed in one block, it can be packed into bytes without unused gaps, i.e. at most 7 bits are wasted per block. Applying the TPC-D benchmark to a database using this compression approach was quite successful with the size of the compressed database reduced to 64% and the total query time on the compressed database reduced to 62%, both compared to the uncompressed database. However, the time for bulkloading the compressed database went up to 146%.

A more advanced compression method based on Huffman coding is suggested in [17] and implemented in the IMS DBMS as a segment store/retrieve filter. A segment in IMS is a concatenation of tuple values of various types, typically considerably shorter than a database page. Because the access granularity of the filter is on segment level, the decompression algorithm can only use (statistical) data stored in the compressed segment and due to the size of the segments this means only very little compression meta data can be stored. Furthermore, the exact boundaries of the tuple values are not known to the filter, although they would aid the handling of compression considerably. The solution suggested in [17] was to use a fixed set of contexts, each with its own set of Huffman codes optimal for the context, and switching the context adaptively whenever a symbol is encountered that is improbable in the current context but probable in another. Thus, in a context for alphabetic symbols, a number is relatively improbable, so after encountering a number the context would be switched to one for numbers. The Huffman codes for the various contexts are generated once for a given database by gathering statistical information about its contents. That is probably the biggest shortcoming of this approach because it works only on (at least statistically) static databases, as changing the Huffman codes to compensate changes in the distribution requires recompressing the entire database. When applied to entire, existing databases, the approach worked quite well, however: the authors compressed a database consisting of student records to 58% with only 17% CPU overhead due to compression.

The above publications share little common ground with this thesis because of the considerable differences between text and array data. Of course basic techniques like variable length or dictionary coding are used in some parts of the compression engine described in this thesis as well, as in most sophisticated compression algorithms, but the model layer differs completely. There is a large body of work using similar model layers in image compression, however, which was used as a design template for parts of the compression engine. The related literature on image compression is too numerous to quote at this point, but a good overview on existing techniques can be found in [47].

1.4 Structure

The thesis starts with an overview on the terminology and data types used as well as the $\mathcal{RasDaMan}$ architecture and its implementation of these data types in chapter 2. This is followed by the design of the compression engine as an object-oriented two-layer architecture (model- and compression layer) supporting a large range of compression techniques in chapter 3. This includes wavelets in theory and their applications in data compression,

predictors, dynamic parameter system, and closes with an analysis of transfer compression. This will be followed by an evaluation of the engine on different data types in chapter 4, and the thesis will close with some comments on the current state of the system and future work in chapter 5.