Reihe Informatik
3 / 1998

The Implementation and Performance of Compressed Databases

Till Westmann Donald Kossmann
Sven Helmer Guido Moerkotte

The Implementation and Performance of Compressed Databases

Till Westmann[1] Donald Kossmann[2] Sven Helmer[1] Guido Moerkotte[1]

[1]Universität Mannheim
Informatik III
D-68131 Mannheim, Germany

[2]Universität Passau
FMI
D-94030 Passau, Germany

Abstract

In this paper, we show how compression can be integrated into a relational database system. Specifically, we describe how the storage manager, the query execution engine, and the query optimizer of a database system can be extended to deal with compressed data. Our main result is that compression can significantly improve the response time of queries if very *light-weight* compression techniques are used. We will present such light-weight compression techniques and give the results of running the TPC-D benchmark on a so compressed database and a non-compressed database using the AODB database system, an experimental database system that was developed at the Universities of Mannheim and Passau. Our benchmark results demonstrate that compression indeed offers high performance gains (up to 55%) for IO-intensive queries and moderate gains for CPU-intensive queries. Compression can, however, also increase the running time of certain update operations. In all, we recommend to extend today's database systems with light-weight compression techniques and to make extensive use of this feature.

1 Introduction

Compression is a heavily used technique in many of today's computer systems. To name just a few applications, compression is used for audio, image and video data in multi-media systems, to carry out backups, to compress inverted indexes in information retrieval, and we all know the UNIX *gzip* and the DOS *zip* commands that we use to ship files across the Internet and to store large files and software packages that we do not need very often.

Compression has two advantages: (1) it reduces costs for storage media (main memory, disk, and tape), and (2) it saves IO bandwidth (disk, tape, or network communication) which results in improved performance for IO-bound applications. On the negative side, compression can be the cause of significant CPU overhead to compress the data in the first place and to uncompress the data every time the data is used so that compression can result in reduced performance for CPU-bound applications.

Since many standard (i.e., relational) database applications execute a fair amount of CPU-intensive operations (e.g., joins and aggregation), compression has not yet found wide acceptance in the relational database arena, and database vendors are only slowly adopting compression techniques for their products. A possible reason could be the fear of increasing response time. Just to give an example: without compression, a query could have IO costs of one minute and CPU costs of 30 seconds resulting in an overall response time of one minute since CPU and IO processing can be overlapped. With compression, the IO costs of the same query could easily be reduced to less than 30 seconds, but if an expensive compression technique is used, the CPU costs could just as easily be increased to become more than a minute resulting in an overall higher response time due to compression. In this paper, we will show that, if done right, compression can in fact reduce the response time of most queries. We will show that in a carefully designed system, the CPU overhead of compression is tolerable while getting high benefits from compression due to reduced disk IO at the same time.

Specifically, we will present a set of very simple and light-weight compression techniques and show how a database system can be extended to exploit these compression techniques. We will address storage management issues such as the efficient implementation of small variable-sized fields, query engine issues such as the efficient evaluation of expressions on compressed data (e.g., predicates, aggregate functions, etc.), and query optimization issues such as the necessary refinements to the optimizer's cost model in order to find good query plans for compressed databases. Except for the optimizer's cost model, we have implemented all our proposed techniques in the AODB database system at the Universities of Mannheim and Passau.

We will also give the results of performance experiments that we carried out using the TPC-D benchmark [TPC95]. These experiments demonstrate the reductions in the size of the database that are likely to be achieved using light-weight compression techniques and confirm that compression improves the performance of most queries (by a factor of two, in the extreme case) and only shows weaker performance for certain update operations.

The remainder of this paper is organized as follows: Section 2 lists related work on database compression. Section 3 presents the light-weight compression techniques used in this work. Section 4 explains how queries can be executed in the presence of compressed data. Section 5 discusses our TPC-D results. Section 6 contains our conclusions.

2 Related Work

Most related work on database compression has focussed on the development of new compression algorithms or on an evaluation of existing compression techniques for database systems (e.g., [Sev83, Cor85, RH93, IW94, ALM96, NR95, GRS98]). Our work differs from all this work in two important ways: First, we were interested in showing how compression could be integrated into a database system rather than inventing new compression algorithms. Second, we were interested in the performance aspects of compression (i.e., the running times of queries), and we will, therefore, present the results of performance experiments. All other experimental studies, on the other hand, investigated only the disk savings that can be achieved with database compression. While disk savings are an

important advantage of compression, we believe that the importance of this factor is going to decrease with the continuing trend of dropping disk prices. We do note that there have been a couple of papers that address performance issues of database compression (e.g., [GS91, SNG93, RHS95, GRS98], but, other than us, none of these papers present the results of comprehensive performance experiments.

There have been two other areas in which compression was studied in the context of database systems. First, there has been work on the design of special implementation techniques for, say, joins based on compression (e.g., [GO95]). Second, there has been a significant body of work on compressed indexes; e.g., VSAM [Wag73], prefix compression for B trees [Com79], compression of rectangle descriptions for R trees [GRS98], and compression of bit mapped indexes [MZ92]. All the work in both of these areas is orthogonal to our work: One, we concentrated on studying the performance of compression if well-perceived query techniques (e.g., hash joins) are used, but the techniques we propose would work just as well if specialized query evaluation algorithms are used. Two, we concentrated on the compression of base data (i.e., relations) and made sure that any kind of index (compressed or not) remains applicable in our environment.

3 Light-Weight Database Compression

In this section, we will describe the compression techniques that we considered in this work. We will first describe the characteristics a compression technique must have to be well-suited for general-purpose database systems, and then list the concrete techniques we have chosen to implement in our experimental system. Rather than inventing new compression techniques, our main contribution is to show explicitly how compression techniques can be integrated into the storage manager of a database system; we address this subject in the third part of this section. As stated in the Related Work section, we will only cover compression techniques for base data (i.e., relations); special-purpose compression techniques for indexes are already very well understood and can be used independently of the techniques we propose here.

3.1 Characteristics

The compression techniques we consider have two main characteristics. First, they are *fine-grained*. Principally, compression can be applied to a whole file of the database (i.e., a relation or a partition of a relation), a page of a file, a tuple, or every individual field of a tuple. As suggested in most recent papers on database compression (e.g., [GS91, SNG93, RHS95, GRS98]), we apply field-level compression, the finest possible granularity, which means that every field of every tuple of the database can be compressed and decompressed without reading or updating other fields of the same or of other tuples. Field-level compression also provides the flexibility to use different compression techniques for different fields of the same tuple, including an approach that compresses some fields (e.g., long strings) and does not compress other fields (e.g., short strings that are frequently used in queries). We do require, however, that the same compression technique is applied to a whole column of a table; for example, we require that the *salary* field of

all `Emp` tuples be compressed in the same way because it would be too cumbersome for the user to specify a compression technique for every tuple individually, and it would be quite costly for the system to determine the compression mode of a tuple before accessing the tuple.

There are several reasons why it is important to provide very fine-grained compression in database systems. We have already seen that fine-grained compression provides the flexibility to employ different compression techniques for different types of data without vertically partitioning tables. As discussed in [GS91, SNG93], this flexibility is very useful for query execution because it allows to generate temporary results in which some fields are compressed and others are not so that *lazy decompression* becomes possible. A query that asks for the *name*, *address*, and *department info* of all employees of a particular set of departments could, for example, be executed by decompressing the relevant join columns for the `Emp` $\bowtie$ `Dept` join and only decompressing the *name, address* and *department info* fields of those tuples which survive the join; other fields (e.g., *salary*) would not have to be decompressed at all. Another argument for fine-grained compression is that any compression technique that compresses more than one tuple at a time does not interact well with other components of the database system. It must, for instance, be possible to compress tuples individually so that update and insert operations can be carried out without affecting any other tuples than the target tuples of the update operation; affecting other tuples would severely impact the locking and recovery components of a database system if protocols such as ARIES are used [MHL+92]. Likewise, it must be possible to decompress individual tuples so that the system can take advantage of indexes in the best possible way and in order to provide efficient navigational access to the database as is done by object database systems. As a result of relying on very fine-grained compression techniques, the compression techniques we consider must work well even for fairly small amounts of data (e.g., one integer).

The second characteristic of the compression techniques we consider is that they are very *fast* in terms of CPU overhead to compress and decompress data. As stated in the introduction, this feature is very important since many database operations are CPU intensive so that there is not much CPU time to waste. Fast compression and decompression is particularly important in the presence of fine-grained compression because executing a single query might involve the decompression of millions of data items. When we pick a compression technique, we are therefore willing to sacrifice a couple of percent of disk savings in order to achieve speed-of-light decompression performance.

3.2 Concrete Techniques

In the following, we will discuss a number of compression techniques that we found useful in order to improve the performance of database systems. Specifically, we will describe three compression techniques: *numeric* compression, *string* compression, and *dictionary-based* compression. Furthermore, we will describe how compression works in the presence of `NULL` values.

Since all three compression techniques are applicable in a variety of cases, there are, in general, many different options to compress a table (including the option not to compress certain fields at all) and choosing the wrong technique can impact the performance of a

database. Nevertheless, we do not think that we are adding another heavy burden to the job of a database system administrator because it is usually embarrassingly obvious what the right compression techniques for a given application are. For example, we did not hesitate to use *dictionary-based* compression for *flag* fields and *numeric* compression for all fields of type decimal in our implementation of the TPC-D benchmark. (Dictionary-based and numeric compression are described below).

3.2.1 Numeric Compression

The technique we use to compress integers is based on *null suppression* and encoding of the resulting length of the compressed integer [RH93]. This technique has also been built into ADABAS, a commercial relational database system by Software AG [AG94]. The idea is to cross out leading 0's of the representation of an integer. In most systems, integers are represented using four bytes so that Integer 3 is represented by 30 bits that are set to 0 and two bits that are set to 1. With this kind of compression, Integer 3 could, therefore, be represented using two bits. Of course, the crux of numeric compression is to keep the information of how many bits are used to represent a specific integer because this information is needed to decompress the integer.[1] We will discuss a technique to encode and decode this information in Section 3.3, after having presented all the other compression techniques, because our encoding and decoding techniques are not specific to this or any other particular compression technique; for the moment, however, keep in mind that our coding scheme only works well if compressed fields are aligned to bytes. That is, Integer 3 will be represented by one byte rather than two bits. Alignment to bytes is one example of how we trade disk savings for high-speed decompression.

The same compression technique as for integers can also be applied to dates. Often a date is represented by the number of days the date is before or after some certain base date. If the base date is November 3, 1997, then the Date November 5, 1997 could be represented by the Integer 2, and the Date October 23, 1997 could be represented by the Integer -11 and both dates could, therefore, be compressed just like any other integer.

We apply a special compression technique to floating point numbers that are represented using eight bytes in their uncompressed state (i.e., doubles). In many cases, an eight byte floating point number can be represented using only four bytes and without loosing any information. We will take advantage of this fact and represent floating point numbers using four bytes whenever this is possible.

Other forms of numeric compression that we did not consider in our work, but that might be helpful in some situations can be found in [NR95, GRS98].

3.2.2 String Compression

SQL allows to define strings in two different ways: `CHAR(n)` or `VARCHAR(n)`. In most database systems, `CHAR(n)` fields are represented by allocating a fixed chunk of bytes of length n, whereas `VARCHAR` fields are typically implemented by recording the length of the

[1] In some cases, we also need to encode the sign of the integer in order to achieve effective compression for, e.g., Integer -3.

string and storing the string in a variable chunk that can shrink and grow depending on the current state of the string. A simple way to compress `VARCHAR` fields is to simply compress the part that records the length of the string using the numeric compression technique for integers defined above. `CHAR` fields can be compressed by converting them into a `VARCHAR` in a first step, and then achieving further compression by, again, compressing the length of the resulting variable string in a second step.

If the strings are very long, it is sometimes beneficial to further compress string fields. If order preservation is not important, such an additional compression can be done by using the classic compression techniques such as Huffman coding [Huf52], Arithmetic coding [WNC87], or the LZW algorithm [Wel84]. If an order preserving technique is needed, then this additional compression can be done using the techniques proposed in [BCE76, ALM96]. All these compression techniques can be carried out independently and in addition to the compression of the part that records the length of the (compressed) string.

3.2.3 Dictionary-based Compression

Dictionary-based compression is a very popular compression technique that can be used for any data type. Dictionary-based compression is particularly effective if a field can take only a small number of different values, and it is based on storing all the different values a field can take in a separate data structure, the *dictionary*. If, for instance, a field can only take the values "Mannheim" and "Passau," then the value of the field could be represented by a single bit, and this bit could be used to look up the decompressed value of the field in the dictionary.

There are many different variants of dictionary-based compression conceivable. We have chosen to implement a very simple and somewhat limited variant in which the maximum size of the dictionary is known in advance and, therefore, the number of bits required to represent a field are known in advance, too. Another interesting and more general variant of dictionary-based compression is presented in [ALM96].

3.2.4 Dealing With `NULL` Values

SQL also allows `NULL` values for every data type. In realistic applications, integrity constraints disallow `NULL` values for many fields, but in the absence of such constraints, the system must take into account that fields may have `NULL` values, and a compression technique must uniquely represent and identify `NULL` values.

If dictionary-based compression is used, `NULL` values can easily be represented by defining `NULL` as one of the possible values and recording it in the dictionary. If numeric or string compression are used, then a field with value `NULL` can be represented as a field with length 0; this does not cause much additional trouble because both numeric and string compression force the system to record the length of every compressed field independently of the presence of `NULL` values. However, we do need to distinguish a `NULL` value from Integer 0, Double 0.0, or the empty string. To do so, we represent Integer 0 as one byte with all bits turned off (i.e., a compressed 0 has length 1) and Double 0.0 using 4 bytes.

The empty string is represented as a string with length 0 (compressing the length using 1 byte) and the `NULL` string is represented as a string with length `NULL`.

3.3 Encoding and Decoding Compression Information

We now turn to the question how all these (and many other) compression techniques can be integrated into the storage manager of a database system. As seen in the previous subsection, the effectiveness of many (variable-length) compression techniques depend on efficiently encoding and decoding length information for compressed data items. Another issue is finding the right offset of a field, if a tuple contains several variable-length (compressed) fields. The approach we take encodes the length information of every field into a fixed number of bits and packs the length codes of all compressed fields together into a special part of the tuple. In the following, we will describe the resulting overall layout of compressed tuples, and then our encoding and decoding algorithms.

3.3.1 Layout of Compressed Tuples

Figure 1 shows the overall layout of a compressed tuple. The figure shows that a tuple can be composed of up to five parts:

- The first part of a tuple keeps the (compressed) values of all fields that are compressed using dictionary-based compression or any other fixed-length compression technique.
- The second part keeps the encoded length information of all fields compressed using a variable-length compression technique such as the numerical compression techniques described above.
- The third part contains the values of (uncompressed) fields of fixed length; e.g., integers, doubles, `CHARs`, but not `VARCHARs` or `CHARs` that were turned into `VARCHARs` as a result of compression.
- The fourth part contains the compressed values of fields that were compressed using a variable-length compression technique; for example, compressed integers, doubles, or dates. The fourth part would also contain the compressed value of the size of a `VARCHAR` field if this value was chosen to be compressed. (If the size information of a `VARCHAR` field is not compressed, then it is stored in the third part of a tuple as a fixed-length, uncompressed integer value.)
- The fifth part of a tuple, finally, contains the string values (compressed or not compressed) of `VARCHAR` fields.

While all this sounds quite complicated, the separation in five different parts is very natural. First of all, it makes sense to separate fixed-sized and variable-sized parts of tuples. The first three parts of a tuple are fixed-sized which means that they have the same size for every tuple of a table. As a result, compression information and/or the value of a field can directly be retrieved from these parts without further address calculations. In

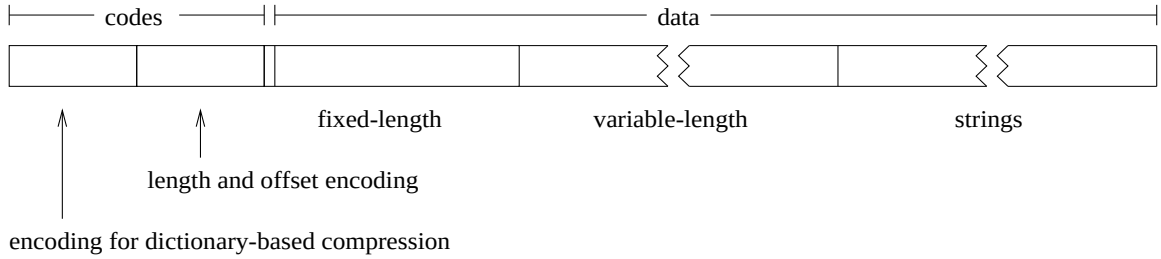


Figure 1: Layout of a Compressed Tuple

particular, uncompressed integer, double, date, ...fields can directly be accessed regardless of whether other fields are compressed or not. Furthermore, it makes sense to pack all the length codes of compressed fields together because we will exploit this bundling in our fast decoding algorithm, as we will see soon. Finally, we separate small variable-length (compressed) fields from potentially large variable-length string fields because the length information of small fields can be encoded into less than a byte whereas the length information of large fields is encoded in a two step process as described in Section 3.2.2.

Obviously, not every tuple of the database consists of these five parts. For example, tuples that have no compressed fields consist only of the third and, maybe, the fifth part. Furthermore keep in mind that all tuples of the same table have the same layout and consist of the same number of parts because all the tuples of a table are compressed using the same techniques.

3.3.2 Length Encoding

From the discussion of the layout of (compressed) tuples, it is fairly obvious how uncompressed fields and fixed-length compressed fields are accessed. The open question is how variable-length compressed fields are accessed. In the following, we will describe how the length of such fields is encoded and packed into the second part of a tuple, and then in the next subsection, we will describe how this information is decoded.

Recall that we are mostly interested in encoding the length of compressed integer and double values. (Dates can be compressed and represented as integers and for strings, we keep the length information separately and compress it just like an integer.) Also recall that due to byte alignment, a compressed integer can be 1, 2, 3, or 4 bytes long. As a consequence, we can encode the length of a compressed integer using two bits as shown in the `NOT NULL` column of Table 1. If the integer can take `NULL` values, then we need three bits to encode the length of the compressed integer because a compressed integer can be 0, 1, 2, 3, or 4 bytes long in this case (the `NULL allowed` column of Table 1).

Analogously, Table 2 shows the codes for the lengths of a compressed double. Here, recall that a compressed double can either be 4 or 8 bytes long if `NULL` values are not allowed, and 0, 4, or 8 bytes long if `NULL` values are allowed so that the length can be coded using only one bit in the first case and two bits in the second case.

If a tuple has several variable-length compressed fields, then we will try to pack the length codes of as many fields as possible into a single byte, but we will make sure at the same time that the code of a single field can be retrieved by probing a single byte only. We

Length	NOT NULL	NULL allowed
0	—	000
1	00	001
2	01	010
3	10	011
4	11	100

Table 1: Length Encoding for Integers

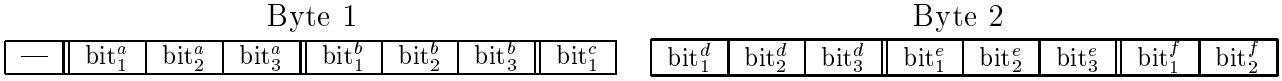
Length	NOT NULL	NULL allowed
0	—	00
4	0	01
8	1	10

Table 2: Length Encoding for Doubles

will illustrate this in the following example. The example compresses tuples that consist of four integer fields with NULL allowed, one integer field with a NOT NULL constraint, and a double field with a NOT NULL constraint:

$$\mathcal{R} = \langle \texttt{a:int, b:int, c:double not null, d:int, e:int, f:int not null} \rangle$$

If all fields are compressed as described in Section 3.2.1, then the length codes of all six fields are packed into two bytes in the following way:



Here, bit_i^x refers to the ith bit of the length encoding for Attribute x, and the first bit of Byte 1 is not used.

3.3.3 Length Decoding

Given the encoding scheme from the previous section, it is easy to determine the length of a compressed field of a specific tuple: we simply need to access the right bits for this field in the length-encoding part of the tuple and look up the length in an encoding table such as Table 1 or 2. Before, we can actually access the field, however, we need to solve another problem: we have to determine the *offset* of the field which depends on the length of all the other (compressed) fields stored in the tuple before that field.

A naive algorithm to determine the offset of, say, the ith compressed field would be to loop from $j = 1$ to $i - 1$, decode the length of the jth field, and compute the offset as the sum of these lengths. This algorithm would, however, have very high CPU overhead because it would involve decoding the length information of $i - 1$ fields. Fortunately, we can do much better by materializing all possible offsets a field can have in so-called *decoding tables*. To see how this works, let us continue our example from above and look at one concrete tuple of Relation $\mathcal{R}$. If the compressed value of a of this tuple has 2 bytes, the value of b has 0 bytes, the value of c has 8 bytes, the value of d has 3 bytes, the value of e has 0 bytes, and the value of f has 4 bytes, then the two bytes that encode the length of the tuple would look as follows (using the codes of Tables 1 and 2 and setting the unused bit of Byte 1 to 0):

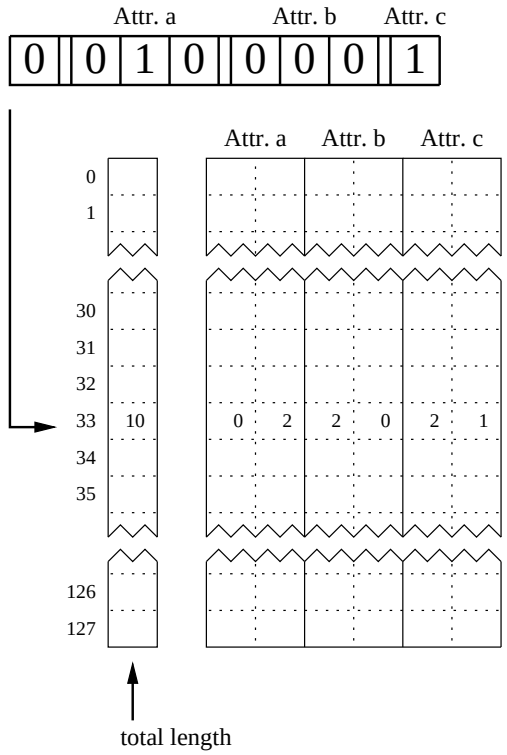


Figure 2: Decoding Byte 1

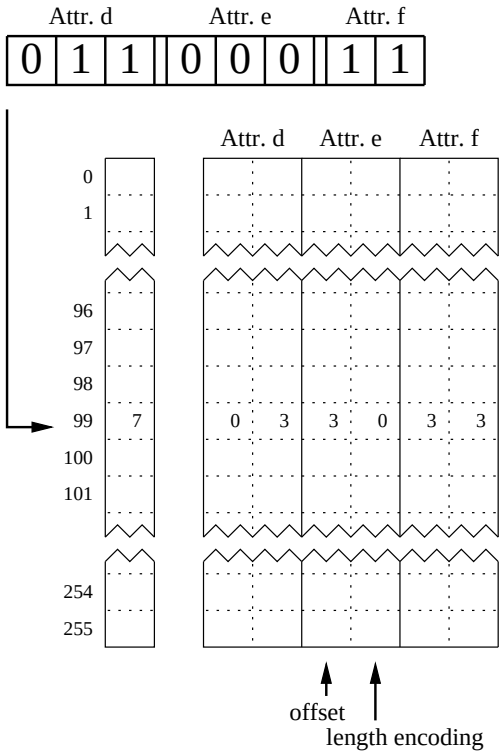
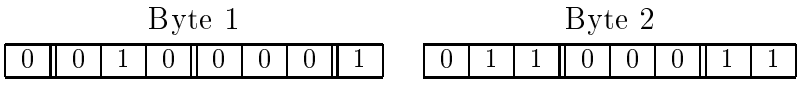


Figure 3: Decoding Byte 2



Now, let us determine the offset of Attribute e for this tuple. We first probe the decoding table of Figure 2 with Byte 1 to find out that with this length encoding, Attributes a, b and c combined consume 10 bytes. Then, we probe the decoding table of Figure 3 with Byte 2 to find out that we need to add another 3 bytes to determine the full offset for Attribute e in this tuple, and to get the length of Attribute e.

In general, we maintain d decoding tables for a relation whose tuples have d bytes in their length-encoding part. Every decoding table has 2^b entries, where b is the number of bits used in the corresponding length-encoding byte; e.g., $b = 7$ for Byte 1 and $b = 8$ for Byte 2 of our example. Every entry of a decoding table has one *total length* field and an *offset* and *length* field for every attribute that is encoded in the corresponding byte (see Figures 2 and 3). Given this design, the offset and length of a compressed field of a tuple can be determined by the algorithm shown in Figure 4.

To go back to our example, we note that the decoding tables are really very small. Relation $\mathcal{R}$ from our example would require one decoding table of size 2 KB ($128\text{entries} * 16\text{bytes}$) and one decoding table of size 4 KB ($256\text{entries} * 16\text{bytes}$). So, we expect that these decoding tables can be kept in main memory most of the time just like any other meta data. If we are willing to invest more memory, we could materialize all possible offsets into a single table (rather than one table per byte) and achieve decoding in constant time. For Relation $\mathcal{R}$, such a *universal* decoding table would require about 0.5 MB of main memory ($2^{15}\text{entries} * 16\text{bytes}$). (Of course, there are also many ways of compromise conceivable.)

Input: Attribute identifier *attr*, encoding information *codeByte[]* of a tuple, an array of decoding tables *table[][]*, and *encIn* which is the number of the Byte in *codeByte* that records *attr*'s length code

Output: offset of *attr* in the tuple, length of the value of *attr* in the tuple

```
1: offset = 0;
2: for j = 1 to encIn - 1 do
3:     offset = offset + table[j][codeByte[j]].total
4: return ⟨ offset + table[encIn][codeByte[encIn]].offset(attr),
                    table[encIn][codeByte[encIn]].length(attr) ⟩
```

Figure 4: Algorithm to Decode Compression Information

4 Executing Queries on Compressed Data

In the previous section, we showed how compression techniques can be integrated and implemented efficiently in the storage manager of a database system. In this section, we will describe the necessary adjustments to carry out queries efficiently in the presence of compression. Obviously, compression could be integrated into a database system without any adjustments to the query execution engine and query optimizer of a database system by simply encapsulating compression in the storage manager. Recall from the introduction, however, that compression can easily turn an IO-bound query into a CPU-bound query, and since textbook query execution engines have been designed to minimize IO costs, they need to be extended to minimize CPU costs, too, in order to take full advantage of database compression. Furthermore, compression can impact the choices made by an optimizer; for example, the best join orders in a compressed and uncompressed database may differ.

In the following, we will first present the design of our fully tuned query execution engine and then discuss the query optimization issues. At the end of this section, we will give a brief report on the status of our system and on our experiences in building the system. Section 5, then, presents the results of experiments that evaluate the performance of our engine.

4.1 Query Execution Engine

To achieve lowest-possible CPU overhead during query execution, we propose the following two techniques: (1) an extended query iterator model, and (2) evaluation of expressions using a virtual machine which interprets assembler-like programs. The first extension is necessary in order to avoid unnecessary copying of tuples during query execution and to avoid decompressing certain fields twice for the same query. The second extension is necessary in order to minimize the cost to evaluate expressions (e.g., predicates and aggregate functions).

4.1.1 Extended Iterator Model

In the classic iterator model, every query operator such as *table scan, index scan, sort,* etc. (called iterator) provides three methods [Gra93]: *open*, *next*, and *close*. *open* allocates resources (main memory, disk for temporary results) and does all computations that need to be carried out before the iterator can actually produce results. *next* delivers the result tuples of an iterator one at a time. *close* releases all the allocated resources and does other cleaning up work.

Here, we are concerned about the interface of the *next* method of an iterator. In the classic iterator model, *next* delivers the next result tuple by returning a (main memory) pointer to that tuple. We generalize this interface and allow the *next* method to return an *array* of pointers (rather than just a single pointer). This extension is necessary to implement a technique called *implicit joins* which is known since the seventies and which avoids copying tuples when pipelined join methods are used [Pal74]. To see how implicit joins save CPU costs to copy tuples, consider an index nested loop join (NLIJ) between Relations A and B. In the classic iterator model, the NLIJ iterator must copy matching tuples from A and B into a result tuple in order to return a pointer to the result tuple. With implicit joins, the NLIJ iterator simply returns two pointers for every pair of matching tuples from A and B without copying any tuples from either relation.

The second step we take to generalize the interface of the *next* method in the iterator model is to allow iterators to return values of fields of tuples in addition to just returning pointers to tuples. This extension is necessary to avoid decompressing a field twice in the course of evaluating a query. Consider the following example of an Employee database: a table scan iterator on `Emp` evaluates the predicate `Emp.salary > 100,000` and pipes the result into an NLIJ iterator that evaluates the predicate `Emp.salary < 1% * Dept.budget` (among others). Now, assume that the `Emp.salary` field is compressed and consider that the *next* method of an iterator could only return pointers to tuples. In this case, the table scan iterator would decompress the `Emp.salary` fields of all `Emp` tuples a first time to evaluate the `Emp.salary < 100,000` predicate. The table scan iterator would pass pointers to the resulting (compressed) `Emp` tuples to the NLIJ iterator so that the NLIJ iterator would have to decompress the `Emp.salary` field of all resulting tuples a second time in order to evaluate its join predicate. If we generalize the interface of the *next* method, then the table scan iterator could return the decompressed values of the `Emp.salary` fields in addition to the pointers to the resulting `Emp` tuple, and the NLIJ iterator could use the uncompressed values generated by the table scan iterator.

To conclude, the interface of the *next* method in our extended iterator model now becomes (in C++):

```
bool next(ZReg[]& regSet);
```

The *next* method of an iterator returns `FALSE` when the iterator is done and cannot find any (new) result tuples; *next* returns `TRUE` otherwise. Results (i.e., pointers to tuples and values of fields) are passed through `regSet` which is an array of type `ZReg`. `ZReg` is a C++ `union` type that can take the uncompressed value of the C++ pendant of any common SQL data type (i.e., integer, double, date, etc.) as well as internal types used

```
LEQ_C_D_AC  10  14  ''1998-02-09''  13
AVM_STOP
```

Figure 5: AVM Prg: shipdate $\leq$ 1998-02-09

```
SUB_C_F_CZ  1.0  5   7
ADD_C_F_CZ  1.0  6   8
MUL_C_F_ZZ   7   8   9
MUL_A_F_ZZ   4   9   10
AVM_STOP
```

Figure 6: AVM Prg: Aggregate Function

by the query engine such as RID and `void` $*$. For reasons that will become clear in the next subsection, we call every entry of the `regSet` array a *register*.

4.1.2 The AODB Virtual Machine

To date, there has been very little published work on the efficient evaluation of expressions in database systems. The intuitive approach is to generate an operator tree for every expression at compile time of a query and to interpret the operator tree during query execution, and as far as we know, this approach is used by Oracle. The operator-tree approach was also the approach that we had initially implemented for the AODB system. Going through an operator tree, however, involves high CPU costs, and in the experiments carried out with our first version of AODB, these CPU costs were so high that they ate up all the benefits we achieved by saving IO costs with compression. We, therefore, developed a more efficient method which is based on generating assembler-like statements at compile time and interpreting these statements with a special virtual machine that we call AVM[2]. The only references to a similar idea we found in the literature are an (old) IBM technical report [LW79] and a paper that describes how a rule-based system can be used to generate such statements for a given query [FG89]. We were also told that IBM DB2 uses assembler-like statements to evaluate expressions, but such details of DB2 have not yet been published.

As an example, Figures 5 and 6 show two AVM programs that we used to implement Query 1 of the TPC-D benchmark [TPC95]. The first program implements the `shipdate` predicate of this query, and the second AVM program implements the `sum(extended_price * (1 - discount) * (1 + tax))` aggregate function of this query. The instructions of all AVM programs operate on the registers that are passed around in the iterators; that is, the instructions take their parameters from registers and they write their results into registers. For example, the `LEQ` statement ($\leq$) of Figure 5 compares the 10th attribute of the tuple pointed to by Register 14 with the constant `1998-02-09` and writes the result into Register 13. Going into the details of the instruction set supported by AVM is beyond the scope of this paper; it should, however, become clear that AVM allows more efficient expression evaluation than operator trees just as *compiled* programming languages are more efficient than *interpreted* programming languages. Note, however, that the statements of AVM are machine-independent. Also note that the AVM statements have nothing compression-specific about them.

On the iterator side, iterators get AVM programs (instead of operator trees), and iterators call AVM to execute these programs, evaluate expressions, and load registers as a side

[2]AVM stands for AODB Virtual Machine.

effect: scan iterators get an AVM program for all the predicates they apply, join iterators get separate AVM programs for their primary and secondary join predicates, and group-by iterators get several AVM programs in order to compute aggregate values (separate programs to initialize, compute, aggregate, and store the aggregate values). All iterators, except pipeline breakers such as *temp*, get an AVM program to be applied to the result tuples of the iterators; usually, these AVM programs only involve copying a pointer to the result tuple(s) into a register so that it can be consumed by the next iterator and used as a parameter in the AVM program of the next iterator. Again note, that the implementation of the iterators have nothing compression-specific about them (just like AVM) so that the same set of query iterators can be applied to compressed and uncompressed databases.

4.2 Query Optimization

To get the best possible query plans in the presence of compression, two small adjustments to the (physical) query optimizer are required. These adjustments are necessary because the best query plan for an uncompressed database might not be the best query plan for a compressed database. Since compression impacts the size of base relations and intermediate results, compression (ideally) also impacts join ordering and the choice of join methods. An (index) nested-loop join might, for example, be favorable for a query in a compressed database because the inner relation fits in memory due to compression, whereas a merge join might be favorable for the same query in an uncompressed database because neither relation fits in memory due to the absence of compression. One important point to notice, however, is that a query plan that shows good performance in an uncompressed database will also show good performance in a compressed database so that an optimizer that lacks these adjustments will produce acceptable plans for compressed databases. The purpose of the two adjustments we list in the following, therefore, is to find *as-good-as-possible* query plans for compressed databases.

The first and most important adjustment we suggest is to make the cost model of the optimizer "compression-aware." If the optimizer's cost model is compression-aware, the optimizer will automatically choose the (index) nested-loop join plan in the example of the last paragraph because this plan has lower cost than the merge join plan, and the optimizer will also automatically generate a plan with the right join order in the presence of compression. To make the optimizer's cost model "compression-aware," we need to carry out the following two steps:

1. The cost model ought to account for the CPU costs to decompress fields of tuples. Given the techniques presented in Section 3 and the optimizer's estimates for the cardinality of base relations and temporary results, these CPU costs are very easy to predict, and therefore, can easily be considered in the optimizer's cost model.

2. The memory requirements and disk IO estimates of every iterator need to be adjusted because they are significantly smaller in the presence of compression. The exact savings due to compression are difficult to predict because they depend on the characteristics of the data set; for the light-weight compression techniques we used in this work, however, we found that savings of 50% per compressed attribute is a good rule of thumb.

The second adjustment we propose is to make the optimizer decide whether temporary results should be compressed or not. If large temporary results must be written to disk (e.g., for sorting), it is sometimes good to compress those fields that are not already compressed (e.g., results of aggregate functions) first in order to save disk IO. If, however, the sort can be carried out in memory completely because the temporary results are small or if there are no CPU cycles left to be spent, then it does not make sense to compress fields because no advantage can be expected from compression.

4.3 Implementation Status

We have implemented the extensions to the storage manager described in Section 3 and most of the extensions to the query engine described in this section as part of the AODB project using C++ as a programming language. At this point, the only component that is not working well is the query optimizer. This deficiency is, however, not due to compression. It is due to the fact that the system has been growing dramatically in the last couple of months, so that we have not been able to keep the optimizer's up-to-date in order to consider all different join and group-by methods we have implemented.

One important observation we made is that our changes to the query execution engine (extended iterator model and AVM) helped to improve the performance of queries on compressed *and* uncompressed databases: in both cases, we observed savings in CPU costs by more than a factor of two due to these changes. Also, integrating compression into the storage manager did not affect the performance of queries on uncompressed data (see Section 3.3.1). Not affecting the performance of queries on uncompressed data was very important for us because it allowed us to explicitly study the performance tradeoffs of database compression, which is the subject of the next section.

5 TPC-D Benchmark Results

In this section, we will present the results of running the TPC-D benchmark [TPC95] on a compressed database using the techniques described in the previous two sections. To study the performance tradeoffs of database compression, we compare these results with the results of the TPC-D benchmark on an uncompressed database. We will first describe the details of our implementation of the TPC-D benchmark, and then the database sizes, the bulk loading times, and the running times of the seventeen queries and two update functions.

5.1 Implementation of the TPC-D Queries and Update Functions

As mentioned before, we used the AODB database system as an experimental platform. AODB is pretty much a textbook relational database system with the special features described in Sections 3 and 4 (e.g., extended iterator model, AVM, etc.). Since the AODB optimizer does not currently work very well, we had to craft the query plans for

the TPC-D queries and update functions by hand. In doing so, we were guided by the query plans produced by commercial database systems for the TPC-D queries. The plans were mostly left-deep with group-by operators sitting at the top of the plans (i.e., we did not consider any early aggregation alla [YL94, CS94]). Grace-hash and block nested-loop were the preferred join methods [HR96, HCLS97], and we also used hashing for most of our group-by operations. The plans (including the memory allocation) we used for the compressed and uncompressed databases were identical; that is, we first found a good plan for a query using the uncompressed database, and then ran this plan on the compressed and the uncompressed database. As described in Section 4.2, this approach was conservative: it might have been possible to achieve better response times on the compressed database with compression-specific query plans so that the results we present can, in some sense, be seen as a lower bound for the performance improvements that can be achieved by compression.

In our implementation of the TPC-D benchmark we did not use any indexes. The current version of AODB supports a rudimentary implementation of B-trees as the only index structure, and our B-tree index scans were simply too expensive for the kind of heavy-weight TPC-D queries—regardless of whether compression was used or not. It should be noted that indexes do not affect the performance tradeoffs of database compression. First, indexes can be implemented and work completely independently from base data compression if a fine-grained compression technique is used. Second, indexes reduce the number of disk IOs to execute a query; at the same time, however, indexes also reduce the amount of data that needs to be decompressed so that indexes reduce the benefits and overhead of compression by the same proportion.

To compress the TPC-D database, we used the following compression techniques: we used numeric compression as described in Section 3.2.1 for all integers and decimals, and we used dictionary-based compression for all *flag* fields as described in Section 3.2.3. We turned all `CHAR(n)` strings into `VARCHARs` and compressed the length information of `VARCHARs` as described in Section 3.2.2. We did not use any Huffman coding or so to further compress strings. We felt that using such sophisticated compression techniques for long string fields (e.g., *comments* in the TPC-D tables) would have been unfair in favor of compression because they would have resulted in high compression rates (i.e., high IO savings) without paying the price for this compression as these fields are rarely used in the TPC-D queries. Also, we did not compress any date fields, and we did not compress the `REGION` and `NATION` tables as they fit into a single page on disk.

We ran all TPC-D queries and update functions on compressed and uncompressed databases with scaling factor SF=1. The machine we used was a Sun Ultra I workstation with a 167 MHz processor and 400 MB of main memory. To execute the queries, however, we limited the size of the database buffer pool to 24 MB. The databases were stored on a 4 GB Seagate Barracuda disk drive, and we had another 4 GB Seagate Barracuda disk drive to store temporary results of queries. The operating system was Sun Solaris Version 2.5. We configured AODB to use 4 KB as the page size.

Table	*Compressed*	*Uncompressed*
lineitem	427,360	758,540
order	132,164	177,900
partsupp	112,588	124,600
part	24,120	29,680
customer	22,916	28,256
supplier	1,412	1,640
nation	4	4
region	4	4
total	720,568	1,120,634

Table 3: Size in KB of the Compressed and Uncompressed Tables (SF=1)

5.2 Size of the Compressed and Uncompressed Databases

Table 3 shows the size of the compressed and uncompressed databases. Depending on the TPC-D table, we achieved compression rates between 10% and 45%. The highest compression rate we achieved was for the `LINEITEM` table because the `LINEITEM` table has many numerical values (integers, decimals etc.) that we did compress. On the other hand, the fraction of (long) string values for which we only compressed the length information was quite high in the other tables so that we did not achieve high compression rates for these tables. As a rule of thumb, we found that our light-weight compression techniques achieve about 50% compression rate on those fields which we did compress. As stated above, we did not compress the `REGION` and `NATION` tables because they fit into a single 4 KB page.

5.3 Loading the Compressed and Uncompressed Databases

Table 4 shows the times to bulkload the compressed and uncompressed databases. We observe that the bulkloading times tend to be 20% to 50% higher for compression. While compression does reduce the cost to write (new) pages to disk, bulkloading is a CPU-bound operation because bulkloading involves parsing the input file a character at a time. Typically, we had more than 90% CPU utilization during bulkloading both the compressed and the uncompressed databases, and *compressing* tuples which is part of bulkloading has very high CPU overhead; much higher in fact than *decompressing* tuples. We do not show the bulk loading times for the `REGION` and `NATION` tables in Table 4 because they were too small to be measured.

5.4 Running Times of the TPC-D Queries and Update Functions

Table 5 lists the running times, CPU costs, and CPU utilization of the seventeen queries and two update functions of the TPC-D benchmark. Just looking at the results for the seventeen queries, we observe that compression never looses; that is, in all these queries

Table	*Compressed*	*Uncompressed*
lineitem	17:13.8	11:31.5
order	3:06.9	2:08.1
partsupp	1:35.2	1:14.3
part	27.3	17.4
customer	22.4	15.1
supplier	1.5	1.2
total	22:47.1	15:27.6

Table 4: Time (min:secs) to Bulkload the TPC-D Database (SF=1)

the benefits due to reduced IO costs outweigh the CPU overhead of compression. The performance improvements due to compression, of course, depend on the kind of query; i.e., the selectivity of the predicates, the number of joins and the tables involved in the joins, the presence of `ORDER BY` and `GROUP BY` clauses, and on the columns used in the expressions and in the result of the query. In six cases (Q1, Q6, Q14, Q15, Q16, and Q17), we found compression to improve the running time by a factor of two or even more, whereas we only found two cases (Q3 and Q11) in which the savings in running time were less than 20%.

Looking closer at the CPU costs and CPU utilizations of the queries, we can see how important it is to implement compression in such a way that the CPU overhead of compression is as small as possible. None of these queries has 100% CPU utilization, but CPU utilizations of more than 75% are pretty common, if compression is used. A naive decoding algorithm that does not take advantage of *decoding tables* (see Section 3.3.3), alone, would triple the CPU costs of decompression for most queries. Increasing the CPU costs by a factor of three would not only eat up all the benefits of reduced disk IO, it would result in overall much higher running times for the compressed database than for the uncompressed database.

Turning to the results of the update functions, we observe that compression does increase the running time of update function UF1 by a factor of two. UF1 inserts about 6000 tuples which are specified in a text file into the database, and we see basically the same performance degradation due to compression as in the bulkloading experiments (see Table 4). The tuples can be appended to the `ORDER` and `LINEITEM` tables with fairly little disk IO so that CPU costs for parsing the text file dominate the running time of UF1 and compression looses due to the high cost to compress tuples. For update function UF2, on the other hand, compression wins by a significant margin. Update function UF2 deletes 6000 `ORDER` and `LINEITEM` tuples and involves a fair amount of disk IO in order to find the right tuples to delete. Compression saves costs to carry out these disk IOs and has an additional advantage in this particular case: compression has (almost) no additional CPU overhead in this case because tuples that are deleted need not be decompressed. (Only the primary keys need to be decompressed in order to find the right tuples to delete.)

Query Update	*Compressed*			*Uncompressed*		
	Time	CPU	%-CPU	Time	CPU	%-CPU
Q1	64.7	60.6	93.7	121.2	40.3	33.3
Q2	24.2	8.8	36.4	28.7	6.0	20.9
Q3	166.5	112.5	67.6	188.7	84.6	44.8
Q4	113.6	76.3	67.2	177.8	65.3	36.7
Q5	96.0	76.4	79.6	154.0	59.2	38.4
Q6	53.9	32.0	59.4	121.0	24.8	20.5
Q7	107.0	81.0	75.7	162.6	60.9	37.5
Q8	86.6	66.2	76.4	146.4	41.8	28.6
Q9	226.5	167.0	73.7	280.4	138.2	49.3
Q10	133.1	84.5	63.5	197.5	63.1	31.9
Q11	21.5	8.0	37.2	24.5	5.0	20.4
Q12	107.7	88.7	82.4	168.1	76.5	45.5
Q13	80.5	47.5	59.0	149.8	34.3	22.9
Q14	63.6	42.0	66.0	128.0	31.3	24.5
Q15	60.8	45.0	74.0	120.7	34.2	28.3
Q16	84.3	44.4	52.7	175.3	42.8	24.4
Q17	109.3	68.8	62.9	246.9	57.0	23.0
UF1	1.2	1.2	100.0	0.6	0.6	100.0
UF2	253.2	50.5	19.9	418.6	63.3	15.1
total	1854.2	1161.4	62.6	3010.8	929.2	30.9

Table 5: TPC-D Power Test: Running Time (secs), CPU Cost (secs), and CPU Utilization (%)

6 Conclusion

In this work, we showed how compression can be integrated into a database system. While there has been a large body of related work describing specific database compression algorithms, this paper is the first paper that really describes in full detail and at a low level how these algorithms can be built into a database system. Our experience shows that such low-level considerations are very important to make compression perform well because the execution of many queries involves the decompression of millions of tuples. We defined the layout of compressed tuples in which variable-length fields are separated from fixed-length fields and compressed fields are separated from uncompressed fields. This layout allows direct access to fixed-length fields, and it allows to compress individual fields without affecting other (uncompressed) fields. Furthermore, we developed very fast encoding and decoding algorithms so that compressed fields can be accessed almost as fast as uncompressed fields.

In addition to the low-level storage manager issues for the integration of compression, we described the (novel) design of our AODB query execution engine with its extended iterator model and AVM for very fast expression evaluation during query execution. The development of the highly tuned AODB query engine was triggered by our work on high-performance compression, but there is nothing compression-specific about the AODB

query engine and the enhancements are good for any kind of database system, including systems that provide no support for compression at all.

As a result of all these efforts, we were able to show that compression can significantly improve the performance of database systems, in addition to providing disk space savings. We implemented the TPC-D benchmark on a compressed and an uncompressed database and saw that compression improves the running times of queries by a factor of two or slightly more in many cases. For read-only queries, we could not find a single case in which compression would result in worse performance. These observations indicate that compression should definitely be integrated into decision support systems (or data warehouses) that process a great deal of TPC-D style queries. The experiments also showed that CPU costs continue to be an important factor of query performance, even considering today's hardware trends. It is, therefore, important to implement the techniques proposed in this work in order to achieve good performance with compression.

While compression shone brightly for read-only queries, we did see significant performance penalties for *insert* and *modify* operations. In particular, *insert* operations tend to be very CPU-bound and with the techniques we use, compressing a tuple has significantly higher CPU cost than decompressing a tuple so that we cannot foresee that compression is going to improve the performance of OLTP-style applications any time soon.

References

[AG94] Software AG. ADABAS im überblick. Technical Report ADA5000D006IB, Software AG, Munich, Germany, January 1994.

[ALM96] G. Antoshenkov, D. Lomet, and J. Murray. Order preserving compression. In *Proc. IEEE Conf. on Data Engineering*, pages 655–663, New Orleans, LA, USA, 1996.

[BCE76] M. Blasgen, R. Casey, and K. Eswaran. An encoding method for multifield sorting and indexing. Technical Report RJ 1753, IBM Research, San Jose, CA, 1976.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[Cor85] G. Cormack. Data compression in a database system. *Communications of the ACM*, 28(12):1336, December 1985.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In VLDB [VLD94], pages 354–366.

[FG89] J.C. Freytag and N. Goodman. On the translation of relational queries into iterative programs. *ACM Trans. on Database Systems*, 14(1):1, March 1989.

[GO95] G. Graefe and P. O'Neil. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, October 1995.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GRS98] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proc. IEEE Conf. on Data Engineering*, Orlando, FL, USA, 1998.

[GS91] G. Graefe and L. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, Kansas City, MO, April 1991.

[HCLS97] L. Haas, M. Carey, M. Livny, and A. Skukla. Seeking the truth about ad hoc join costs. *The VLDB Journal*, 6(3):241–256, July 1997.

[HR96] E. Harris and K. Ramamohanarao. Join algorithm costs revisited. *The VLDB Journal*, 5(1):64–84, January 1996.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40(9):1098–1101, September 1952.

[IW94] B. Iyer and D. Wilhite. Data compression support in databases. In VLDB [VLD94], pages 695–704.

[LW79] R. Lorie and B. Wade. The compilation of a high level data language. Technical Report RJ 2598, IBM Research, San Jose, CA, 1979.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Systems*, 17(1):94–162, March 1992.

[MZ92] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *SIGIR*, pages 274–285, 1992.

[NR95] W. Ng and C. Ravishankar. Relational database compression using augmented vector quantization. In *Proc. IEEE Conf. on Data Engineering*, pages 540–549, Taipeh, Taiwan, 1995.

[Pal74] F. Palermo. A database search problem. In *Information Systems*, pages 67–101. Plenum Publ., New York, NY, 1974.

[RH93] M. Roth and S. Van Horn. Database compression. *ACM SIGMOD Record*, 22(3):31–39, September 1993.

[RHS95] G. Ray, J. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *Proc. COMAD*, Pune, India, December 1995.

[Sev83] D. Severance. A practitioner's guide to data base compression. *Information Systems*, 8(1):51–62, 1983.

[SNG93] L. Shapiro, S. Ni, and G. Graefe. Full-time data compression: An ADT for database performance. Portland State University, OR, USA, 1993.

[TPC95] Transaction Processing Performance Council TPC. TPC benchmark D (decision support). Standard Specification 1.0, Transaction Processing Performance Council (TPC), May 1995. `http://www.tpc.org/`.

[VLD94] *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Santiago, Chile, September 1994.

[Wag73] R. Wagner. Indexing design considerations. *IBM Systems Journal*, 12(4):351–367, 1973.

[Wel84] T. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[WNC87] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[YL94] W. Yan and P. Larson. Performing group-by before join. In *Proc. IEEE Conf. on Data Engineering*, pages 89–100, Houston, TX, 1994.